AN EVALUATION OF MUTATION OPERATORS FOR EQUIVALENT MUTANTS

By

Maryam Umar

Supervised By

Mark Harman

A project submitted to the Department of Computer Science.
King's College, London.

In partial fulfilment of the requirement for the degree of Msc in Advanced Software Engineering.

Department of Computer Science
King's College, London.

1st September 2006

ACKNOWLEDGEMENTS

I started this project at a point in my life when I faced numerous *new* responsibilities. Being a newly wed, doing an MSc seemed close to impossible. But I firmly believe that if one wishes, the impossible can be achieved. The idea for this project occurred to me during one of my lectures of Advanced Software Engineering. I wondered why is it so impossible to get rid of this issue being faced in mutation testing. It was at this point that I approached Mark Harman with the proposition to work on the problem of equivalent mutants. I was told that that I might face a number of issues whilst working on this project. Indeed, I did.

I can never thank my parents enough to help me get over those horrible days when I couldn't get my software up and running. It took me the longest time to take the first step in doing this project. I want to thank Dr. Jeff Offutt, whose little, but very important guidance helped me take this step.

I would have never been able to gather all the strength and courage to complete this project without the continuous help of my husband, Umar. I also want to thank Kostas Adamopoulos who provided me with some essential literature to study for this thesis. And ofcourse, I can never forget the gentle pushes giving by my supervisor, Mark Harman, into what we jokingly called *the swimming pool* of mutation testing. Thanks to Mark for teaching me how to swim.

Most importantly, I want to thank the Divine Power that has made me capable of putting all this together.

ABSTRACT

Software testing allows programmers to determine the quality of the software. Mutation testing is a branch of software testing which does more than this. It helps determine whether the test cases that have been created, effectively detect all the possible faults in the software. This allows the development of better test sets, thus, ensuring that maximum software quality is achieved. This may seem very inviting to software testers, but there is a major issue being faced which has hindered the widespread use of mutation testing.

Mutation testing works by seeding faults in the software program. Various mutation operators are used to create these faulty programs. These programs are called mutants. The mutants depict software faults that may be caused by programmers while writing the software. Test cases are then executed on these mutants to determine if they have been *killed* or not. Test sets that kill all the mutants are considered to be good as they successfully detect all the possible program faults.

Sometimes, it is difficult to kill all the mutants. The reason is that some of the mutants, although syntactically different than the original program, are still semantically the same. Any test case will not be able to differentiate between the original program and the mutant. Such mutants are called equivalent mutants.

This thesis targets this issue by determining which mutation operators create mutants, which are more probable to create equivalent mutants. A tool called MuJava is used for this purpose. An empirical analysis has been carried out for this purpose, which helps determine which mutation operators develop more equivalent mutants.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

Computer software can have two kinds of problems: faults or failures. A failure does not permit a program to perform its intended function. A fault is an error in the correctness of a program's semantics. Testing is a process of determining faults. Software testing is performed to ensure that the algorithms used in a program are correct, the program can execute correctly in all possible scenarios and it conforms to all the requirements specified. Correctness, completeness and quality are some of the characteristics that all software programs must meet.

Software testing can be performed in a number of ways. There is no fixed process; it is a simple method of trial and error and investigating various problems encountered whilst execution. Software testing is performed using test cases. These make use of some variables and determine the actual output of the program against the expected output. Good test cases help determine faults that occur rarely.

1.1 Mutation Testing

Mutation testing is one of the many types of methodologies used for testing a program. While other software testing techniques focus on the correct functionality of the program, mutation testing focuses on the test cases used to test the programs. The main idea is to create good sets of test cases rather than trying to find all the faults in a particular program. Good test cases are those which are able to discover all the easy and hard-to-find faults in the program. An ideal test case will detect all the possible faults in a software program.

Mutation testing is a white-box testing technique i.e. it examines the internal structure of a program to detect faults. It is used to determine the effectiveness of test cases i.e. how good they are in detecting faults. Mutation testing has been developed using two basic ideas:

1. Competent Programmer Hypothesis: Software programs usually differ from the correct version of the program in minute ways. Most of the programs written are nearly correct.
2. Coupling Effect Hypothesis: Larger programming faults are coupled with smaller faults of the same nature.

These hypotheses form the platform for mutation testing. Since incorrect programs differ from the correct versions in minute ways, mutation testing works by making the correct programs incorrect by using the concept of fault seeding. This is done by the use of mutation operators. These operators are simple rules which create different versions of the same program with minor changes. These versions are called mutants. Typically, mutation operators replace operands with other similar operands or delete entire statements, etc. The process used in mutation testing can be explained simply in the following steps:

1. Suppose we have a program P
2. Apply mutation operators to P to produce mutant P'
3. Apply test case t to P and P'
4. Analyse outputs of P and P'
5. If the outputs are different, then test case is effective i.e. it can detect the fault and has killed the mutant.
6. If the outputs are same, there could be two reasons:
    i. The mutant is difficult to kill. Write another test case to detect this fault.
    ii. The mutant has the same semantic meaning as the original program i.e. it is equivalent.

We need to achieve a high mutation score for any given set of test cases. This mutation score indicates how well a particular test set has detected faults in a program. Mutation score can be defined as:

Mutation score = # of mutants killed / total # of non-equivalent mutants

Where 0 <= M.S. <= 1 or 0% <= M.S. <= 100%

A low score means that the majority of faults cannot be detected accurately by the test set. A higher score indicates that most of the faults have been identified with this particular test set. A good test set will have a mutation score close to 100% or ideally 100%.

A simple example of mutation testing has been illustrated below:

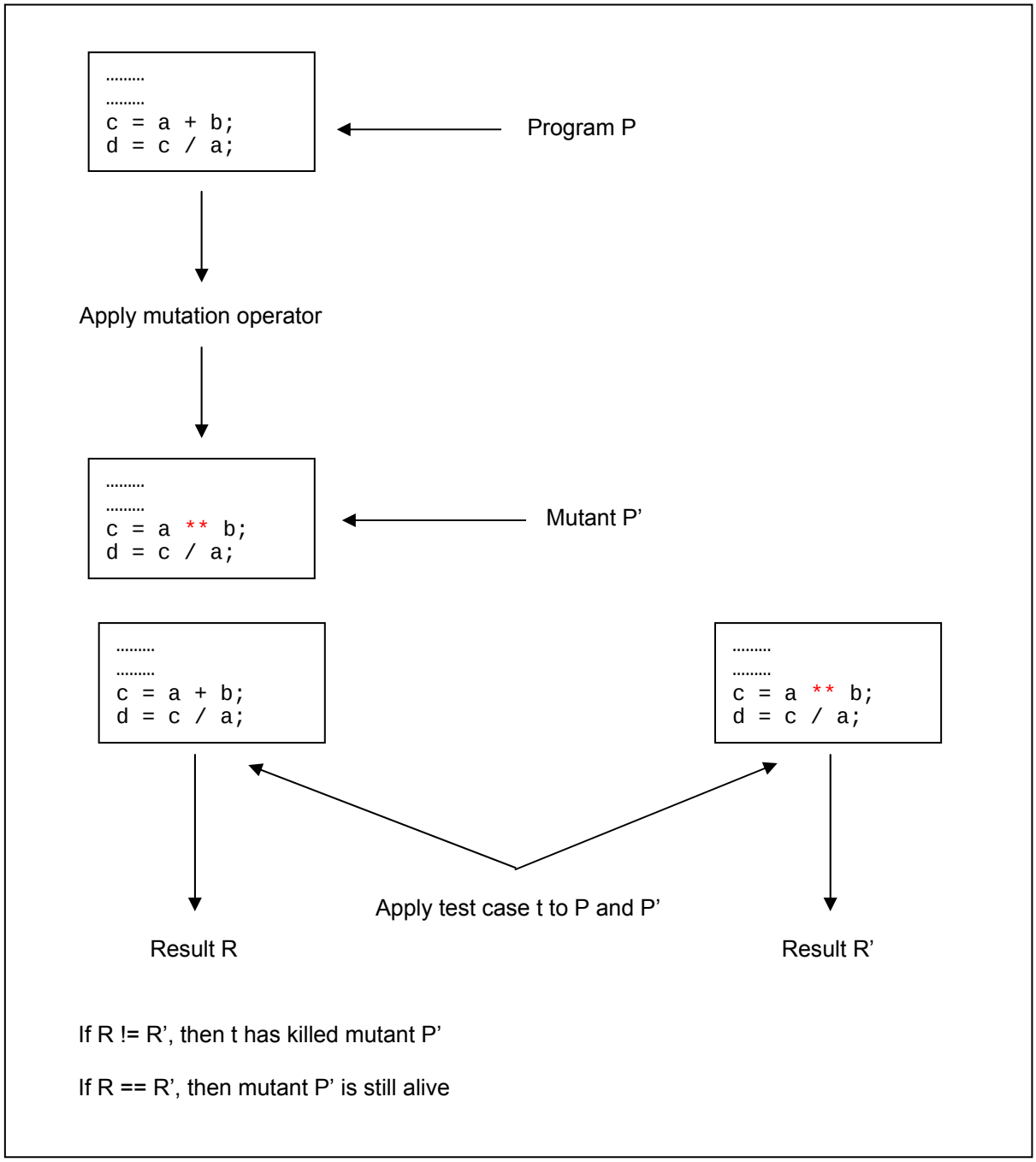

Figure 1: Process of Mutation testing

Mutation testing faces a huge challenge. Since every program may have a fault in many possible ways, one problem is that a large number of mutants are created for very small programs. Another problem is that sometimes most of these programs are equivalent to the original program. The latter problem is the topic of discussion for this thesis.

This project focuses on the use of mutation operators to eliminate the problem of equivalent mutants. The technique adopted suggests that some mutation operators may contribute more towards the creation of equivalent mutants. Hence, a detailed analysis is carried out for all the mutation operators used in MuJava (mutation testing tool). Scenarios are sketched which indicate situations always creating equivalent mutants for certain operators. For the extension of this thesis, it is suggested that the scenarios sketched out be transformed into algorithms and incorporated into the respective mutation operators.

2. LITERATURE REVIEW

The aim of mutation testing is to generate a good set of test cases that successfully help find all possible faults in a software program. We need to achieve a high mutation score i.e. a score close to 100%. To get the maximum mutation score, we need a test set which successfully kills all the mutants. But some of the mutants created are equivalent. Therefore, to get the highest possible score, all the non-equivalent mutants need to be detected.

An equivalent mutant is one, which is syntactically different from the original program, but semantically the same. Such mutants only contribute in increasing the computational cost. They do not help in establishing whether a particular test case is effective in discovering faults in a program or not. The problem of determining whether a mutant is equivalent to the original mutant is theoretically undecidable.

Sample Program P

```
……
……
if(x == 2 && y == 2)
       z = x + y;
……
```

Mutant P'

```
……
……
if(x == 2 && y == 2)
       z = x * y;
……
```

The value of Z will be equal to 4; any test set will be unable to determine any faults with this program because the value will always be equal to 4.

Figure 2: Example of Equivalent Mutants

Mutation testing is not popular in the industry. Reason being, that a large number of mutants are generated for very small programs. This incurs a high computational cost, as each test case needs to be executed on each mutant. The number of mutants generated for a software unit is proportional to the product of the number of data references and the number of data objects [8]. To reduce the computational cost, some approaches have been proposed which are explained below:

2.1 'Do Smarter' approaches

These approaches work by dividing computational cost amongst several machines or by preserving some information, such as compiled code, so that the same code need not be generated repetitively[8]. They may also avoid complete execution of the entire program. E.g. suppose there is a program that has 10,000 lines of code. Assuming the 5th line is changed to create 3 different mutants. It will take a long time to compile the mutated program again and again. To avoid this, the compiled form of the original program is saved. This compiled form of the original program is changed for the single line modified in the mutants. Thus, computation cost is saved.

2.1.1 Weak Mutation

Weak mutation is an approximation technique. It compares the internal states of the mutated program and the original program immediately after executing the mutated portion of the program [8]. If the state of the mutated program is different from that of the original program, the test case has killed the mutant. Otherwise the mutant is still alive. The first tool developed for mutation testing called Mothra used this strategy.

Original program	Mutant

`result = obj.add(a,b)`
`……`
`……`
`Display(result);`
	`result = obj.multiply(a,b)`
	`/* inspect value of result */`
	`……`
	`Display(result);`

Figure 3: Example of Weak Mutation

2.1.2 Distributed Architectures

This approach works by dividing computational cost over multiple machines. Some work has been carried out on vector processors [8], SIMD machines [8], Hypercube (MIMD) [8] machines, and Network (MIMD) [8] computers to adapt them for mutation analysis. This is easy to perform as each mutant is independent of the other mutants and can be executed individually.

2.2 'Do Faster' approaches

These approaches work by generating and running each mutant program as quickly as possible[8].

2.2.1 Schema-based Mutation Analysis

Many mutation systems compile their programs using interpretive methods. This makes the task of compilation low and more tedious to build. Untch [8] has built a Mutant Schema Generation system for this purpose. This system encodes all the mutants into one source-level program to create a 'metamutant' [8]. This metamutant is compiled once and executed in the same programming environment. Since only a fraction of code is different for each schema, repetitive runs of large programs can be avoided on the compiler, thus saving computational cost.

Original program	Metamutant

`result = obj.add(a,b)`	`Switch(n)`
	`Case 1:`
	`  result = obj.add(a,b)`
	`Case 2:`
	`  result = obj.subtract(a,b)`
	`Case 3:`
	`  result = obj.multiply(a,b)`
	`Case 4:`
	`  result = obj.divide(a,b)`
	`Case 5:`
	`  result = obj.modulus(a,b)`
	`Case 6:`
	`  ……`

Figure 4: Example of Schema-based mutation analysis

2.2.2 Separate Compilation Approach

This method avoids the interpretative style of execution. It creates, compiles, executes and runs each mutant individually. When mutant run times greatly exceed individual compilation/link times, a system based on such a strategy will execute 15 to 20 times faster than an interpretative system [8]. One example of such a system is Proteum [8].

2.3 'Do Fewer' approaches

These approaches run lesser number of mutants. A subset of mutants is selected from all the mutants created in such a way that it is sufficient to determine a good set of test cases.

2.3.1 Selective Mutation

The aim of selective mutation is to achieve maximum coverage by using the least number of mutation operators. Operators are selected in such a way that most of the mutants are generated

from them. E.g. multiple mutation operators, which produce the same mutant, are discarded when generating mutants. This idea was developed by Offutt [8] and was called 'selective mutation'. Since lesser operators are used, the number of mutants produced also decreases significantly.

2.3.2 Mutation Sampling

This technique randomly selects a subset of mutants produced. This subset is then tested with the specified test set to determine its sufficiency. If not satisfied, another subset of mutants is selected randomly. Such subsets are continuously selected till we find one which helps determine the effectiveness of the particular test set.

Another method of sampling does not use a priori fixed size of mutants; it uses a Bayesian sequential probability ratio test to determine whether a statistically appropriate sample size of mutants has been reached [8].

2.4 Techniques

Some techniques have been proposed which deal with the problem of equivalent mutants. Some of these have been implemented using some algorithms. Tools have also been created for mutation testing. It was imperative to study these tools as this thesis involves the use of one of these tools.

2.4.1 Overcoming the Equivalent Mutant Problem and Achieve Tailored Selective Mutation Using Co-evolution [1]

This technique makes use of genetic algorithms. It is suggested that the use of genetic algorithms may help reduce the problem of generating a large number of mutants as well as the problem of equivalent mutants. It is argued that this strategy helps attain selective mutation without the need of decreasing the mutation operators applied to the programs. This technique performs three main operations:

1. Evolution of subsets of mutants against a fixed set of test cases:
   In this step, mutants are created and validated against a fixed set of test cases. The score generated indicates the performance level for the mutant against a particular set of test cases. A low score indicates that the mutant is difficult to kill. Subsets of these mutants are then created. Each subset corresponds to an individual of the genetic algorithm. A subset of non-equivalent mutants is then selected with a higher adequacy score. Genetic algorithms are then used to evolve this set. The fitness function used ensures that the mutants selected are such that they are definitely killed by some test case. Uniform crossover is used to enable the selection of highly fit individuals in the population.
2. Evolution of subsets of test cases against a fixed set of test mutants:
   The same strategy is used to create a subset of test cases. Test sets are generated randomly. Mutation scores are then assigned to each set of test cases by executing them against a fixed set of mutants. This mutation score corresponds to the performance of the test set against a given set of mutants. Again, each test set is evolved using a fitness function; it results in another set of test cases which when executed against a set of mutants give a higher adequacy score.
3. Combine the above two sets using co-evolution:
   The two populations generated above are then combined using a fitness function. The score of each mutant is re-evaluated with respect to the present population of test cases. Similarly, the score of each test case is re-evaluated with respect to the present population of the mutants [1].

It is argued in the paper [1], that the strategy proposed ensures that equivalent mutants are not generated. This is due to the design of the fitness functions. The fitness function created assigns an adequacy score of 0 to all the mutants which are difficult to kill since they are more likely to create equivalent mutants. And as explained, only mutants and test cases with higher adequacy scores are selected.

Summarizing the study presented in this paper, selective mutation has been achieved by the use of genetic algorithms without reducing the mutation operators used thus reducing the computational cost. Also, equivalent mutants are not considered to determine the effectiveness of a particular test set due to the ingenious design of the fitness function.

2.4.2 Using Program Slicing to Assist in the Detection of Equivalent Mutants[2]

It is often argued that it is difficult to apply mutation testing to large programs. This is because the detection of equivalent mutants becomes more tedious and almost impossible since it is done manually. It is debated in this paper that the creation of program slices makes detection of equivalence easier.

A formal notation is used. Let the program be *p* and mutant *p'*. An input σ is applied to both *p* and *p'*. *p* and *p'* can be compared by either considering their internal states or final output[2]. To determine whether a mutant has been killed or not, three different choices are considered:

1. Strong mutation:
   Under strong mutation, either the final output of the two programs is considered (strong output mutation) or the final state of *p* and *p'* is considered (strong state mutation). For strong output mutation, an equivalent mutant will produce the same value of variable x as the original program. Similarly, for strong state mutation, an equivalent mutant will produce the same state of variable x as the original program at the end of execution.
2. Weak mutation:
   In weak mutation, if the value of some variable x is different immediately after the execution of some point in the program, say n, for both programs *p* and *p'* when some input σ is applied, then the mutant is killed. Hence, a mutant will be equivalent only when the value of x is identical in *p* and *p'* at any point n.
3. Firm mutation:
   Firm mutation creates *p'* by mutating some part of *p*, for example a loop, a control structure, etc. It then examines *p* and *p'* after applying some input σ to determine if the mutant has been killed or not. If the value of variable x is the same after the final node in the structure, the mutant is equivalent.

The strategy discussed in this paper uses weak mutation. A new Boolean variable, say z, is introduced in the program *p*. A slice is then created on this variable and analyzed to determine if the mutant has been killed or not. The value of z is initially set to true. Every time the node n is met, the value of z is changed. If the mutant is killed it becomes false, otherwise it remains true. Equivalence is indicated if the value of z remains true at the end of execution.

Program *p*

```
1      substring (int from, int to, char target[], char source[]){
2              from++;
3              start = from;
4              if(from < to){
5                      while(from != to){
6                              target[from-start] = source[from];
7                              from++;
8                      }
9                      target[to-start] = '/0';
10             }
11             else
12                     target[0] = '/0';
13     }
```

Figure 5: Sample program to show strategy using program slicing to detect equivalent mutants

Mutant *p'*

```
substring (int from, int to, char target[], char source[]){
        from++;
        start = from;
        if(from < to){
                while(from != to){
                        target[from-start] = source[from];
                        from++;
                }
                target[from-start] = '/0';
        }
        else
                target[0] = '/0';
}
```

Figure 6: Example of mutant created for use with program slicing

The Boolean variable z is introduced at this point on line 2. The statement on line 10 shows that the variable *to* has been modified to the variable *from* in *p'*. A slice is then created with respect to z and the variable *from*. This slice is then tested to determine if the mutant has been killed or is still alive at the end of test execution.

Mutant *p'* to be sliced

```
substring (int from, int to, char target[], char source[]){
        z = TRUE;
        from++;
        start = from;
        if(from < to){
                while(from != to){
                        target[from-start] = source[from];
                        from++;
                }
                z = z && from == to;
                target[from-start] = '/0';
        }
        else
                target[0] = '/0';
}
```

Sliced mutant

```
substring (int from, int to, char target[], char source[]){
        z = TRUE;
        from++;
        if(from < to)
                while(from != to)
                        from++;
                z = z && from == to;
}
```

Figure 7: Introduction of Boolean variable z in *p'* & the creation of a program slice using z

The authors also discuss that creating amorphous slices may reduce the problem of detecting equivalent mutants even more by applying further transformations to the traditional slices[2]. The slice presented in figure 4 will then become:

Amorphous Slice of mutant *p'*

```
substring (int from, int to, char target[], char source[]){
        z = TRUE;
        from++;
        if(from < to)
                z = TRUE;
}
```

Figure 8: Creation of an amorphous slice from a traditional program slice for the detection of equivalent mutants

An interesting observation stated in the paper states that “an ideal amorphous slicing algorithm would yield this slice for all equivalent mutants, and the equivalent mutant problem would disappear” [2].

In summary, this paper discusses that the use of program slices can ease the process of detection of equivalent mutants. it reduces the work carried out manually in identifying equivalent mutants. Also, it has been suggested that equivalent mutants will *always* create slices which are identical for all mutants.

2.4.3 Using Compiler Optimization Techniques to Detect Equivalent Mutants [4]

The strategy presented in this paper makes use of compiler optimisation techniques. The authors of this paper suggest that “many equivalent mutants are either optimizations or de-optimizations of the original program” [4]. Equivalent mutants can always be detected by their optimised code by the use of algorithms. Following are the 6 techniques explained in this paper:

1. Dead code detection
   Mutants that change dead code, i.e. code that will never be executed or is irrelevant, will not affect the output of the program and will thus be equivalent.
2. Constant propagation
   It creates a *constant table*, which keeps entries of variables with constant definitions. Mutants that are still alive at the end of the execution of a test case can be checked for equivalence using this table. A mutant that has an entry in this table will be equivalent.
3. Invariant propagation
   This technique works by saving the relationship between 2 variables, also called an invariant, in an invariant table. An equivalent mutant will have the same definitions in the invariant table, thus not changing the value in the mutated program form that in the original program.
4. Common sub-expressions
   This technique does not identify equivalent mutants directly. It is used in conjunction with other compiler optimisation techniques. It keeps track of all the temporary variables created during compilation and determines the equality of relationships between variables if any.

Normal Code	Optimized Code
T1 = B + C T2 = T1 - D A = T2 T3 = B + C T4 = T3 – D X = T4	T1 = B + C T2 = T1 – D A = T2 X = T2

Figure 9: Common Sub-expression Detection Example

5. Loop invariant detection
   Here, an equivalent mutant is one, which changes the boundary of a loop by moving it inside or outside when the code is compiled.
6. Hoisting and sinking
   This also uses the same technique as loop invariant detection. Equivalent mutants will generate the same result regardless of *hoisting* or *sinking* the code.

2.4.4 Automatically Detecting Equivalent Mutants and Infeasible Paths [3]

The authors of the paper presume that the problem of detecting equivalent mutants is in-fact a type of a feasible path problem. The feasible path problem states that some test requirements are infeasible because of the nature of the program semantics. These requirements cannot be satisfied by the program.

The strategy presented makes use of the constraint-based testing technique. This technique specifies a number of mathematical properties, which need to be met by any test case. Assuming that *P* is a program, *M* is a mutant of *P* on statement *S*, and *t* is a test case for *P*. three mathematical conditions need to be met to kill a mutant:

1. Reachability:
   The test case *must* execute the mutated statement. I.e. if *t* cannot reach *S*, then *t* will never kill *M* [3].
2. Necessity:
   To kill a mutant, the test case *must* cause the mutant to have an incorrect state if it reaches the mutated statement. I.e. for *t* to kill *M*, it is necessary that if *S* is reached, the state of *M* immediately following some execution of *S* must be different from the state of *P* at the same point [3].
3. Sufficiency:
   The test case *must* cause the final state of the mutant to be different from the original program. I.e. the final state of *M* differs from that of *P* [3].

These mathematical conditions are then used to apply constraints on the programs. The constraints are used to determine which mutants are equivalent and which are not.

2.5 Tools

As part of the literature survey, all the tools that have been created for mutation testing were also studied. None of these tools detect equivalent mutants automatically. The tester has to detect them by hand. Tools have been developed for a number of languages:

2.5.1 Mothra

It is the oldest tool developed for mutation testing. It was developed as a project at Georgia Institute of Technology's Software Engineering Research Centre. This project was initiated in 1986 and developed using C language. It was developed for programs written in FORTRAN and made for the Linux platform.

Mothra is a set of tools that can be called separately or called using one of Mothra's interfaces [5]. Since it is based on a set of data structures, many researchers have tried to expand Mothra by adding new tools. Mothra works by creating intermediate code of the program. This requires more compilations and an increase in performance costs.

2.5.2 Jester/Nester/Pester

The tool that was developed first was Jester. It applied mutation testing to Java programs. Jester works for Java code with JUnit tests. Jester does simple modifications to the programs such as changing If statements to true or false, etc. After making these modifications, it runs tests on the modified programs. It then generated web pages displaying the results of the tests using a built-in script.

Pester is the same as Jester only that it was written for programs created in Python. Test cases are written in PyUnit. Nester is also a variation of Jester written for C# programs. Unfortunately, no literature was available to study it in more detail.

2.5.3 MuJava

MuJava is a tool written for java programs. It uses two sets of mutation operators; method-level and class-level. MuJava creates mutants using the various method-level and class-level operators. It then runs test cases on them and evaluates the mutation coverage for them. Test cases are written as separate classes that call methods in the classes that need to be tested.

3. STRATEGY ADOPTED FOR THIS PROJECT

The tool used in this project is MuJava. Before explaining the strategy adopted for this project, it is imperative to explain the working of MuJava in more detail.

3.1 MuJava

Mutation operators make syntactic changes to the program under test. These syntactic changes depict usual syntactical mistakes made by programmers while writing code. MuJava implements a 'do faster' approach to mutation testing to save compilation time [6]. This approach has been adopted primarily for object-oriented programs.

The architecture of MuJava makes use of the Mutant Schemata Generation (MSG) approach. This approach works by creating one meta-mutant of all the mutant programs and requires only two compilations: compilation of the original program and compilation of the meta-mutant program. MuJava uses two types of mutant operators:

- Operators that change the *structure* of the program.
- Operators that change the *behavior* of the program.

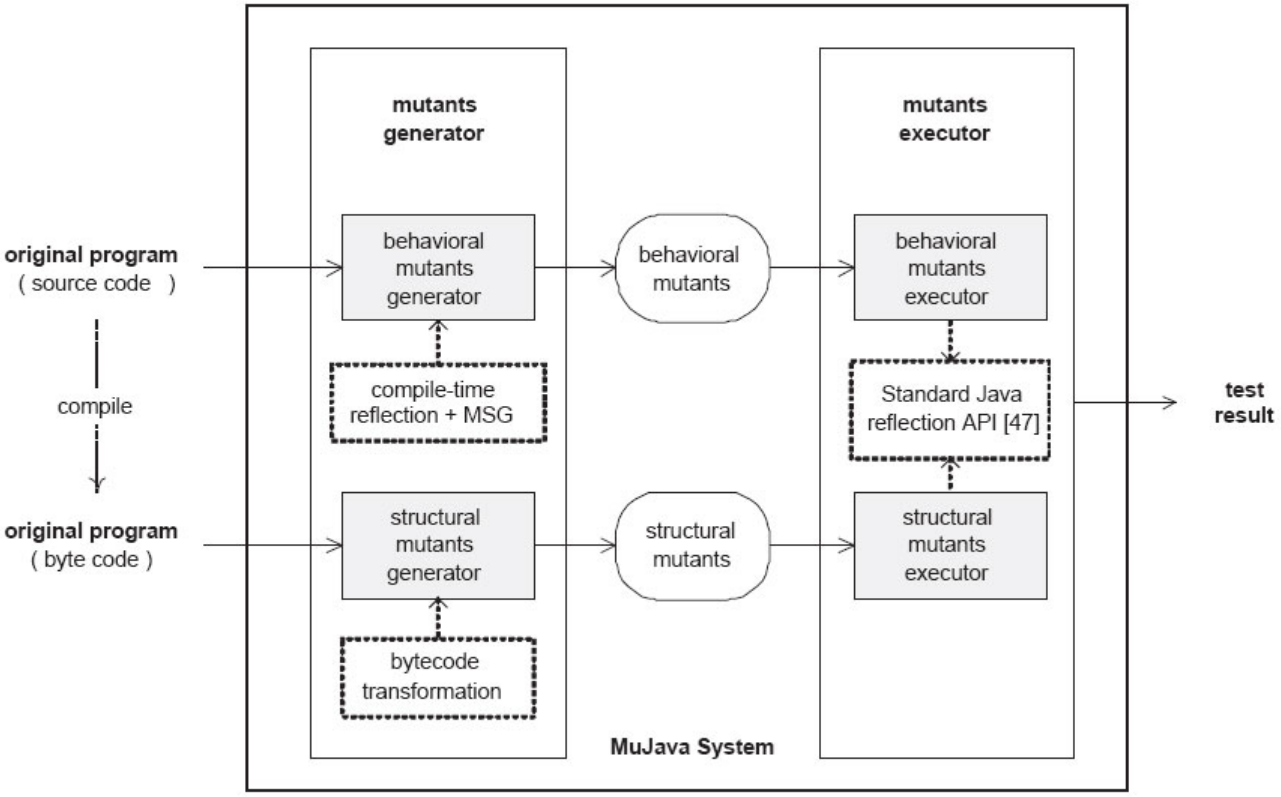

Figure 10: Structure of MuJava [6]

Different engines create the structural and behavioral mutants. For the behavioral mutants, compile time reflection is used to analyze the original program. The MSG engine then uses compile time reflection as well to create a meta-mutant program. For the structural mutants, the original source code is compiled using the Java compiler. BCEL API is then used to add or delete class members in the resulting byte code translation [6].

Details of how MuJava is used are given in Appendix A.

3.2 Motivation

Mutation testing has a high computational cost. A large number of mutants are created for very small programs. This is because each program can have various faulty versions. Each version could be a simple replacement of an addition operator, for example, with all other arithmetic

operators in a particular language. All these mutants need to be compiled and executed against a specific set of test cases.

In addition to this high cost, sometimes most of the mutants created are equivalent. Equivalent mutants produce the same output for the original program as well as the modified program or the mutant.

These issues have prevented mutation testing to gain wide-spread acceptance in the software industry. The aim of this thesis is to help determine a technique by the use of which the number of equivalent mutants produced is reduced.

Quite a few techniques have been proposed to date to deal with this issue as discussed in chapter 2. Some algorithms have been proposed for them, but none of them have been implemented so far. Most authors have resorted to combining other software engineering strategies such as genetic algorithms and compiler optimization techniques with mutation testing. In view of this thesis, this has made the process of detecting equivalence more arduous. The implementation of any of these strategies would require the creation of an entirely new tool for mutation testing rather than reusing any one of those already developed.

The strategy discussed here makes use of MuJava, a mutation testing tool developed in 2003. None of the other mutation testing tools have been used due to lack of support for the Windows platform. Also, some of the tools did not contain enough literature to support their operation. The main idea is that some mutation operators may develop a greater number of equivalent mutants as compared to other operators. To support this argument, the mutation operators of MuJava are discussed in great detail. An analysis is carried out to see which operators create a large number of equivalent mutants.

3.3 The Strategy

For this project, the main aim is to help identify those mutation operators that contribute more towards creating equivalent mutants. The procedure that was followed is explained below:

1. Test programs were created for method-level and class-level operators. These test programs serve as original programs for which mutants will be generated later.
2. Two different approaches were adopted for method-level and class-level operators
    a. For method-level operators, different programs were developed each for arithmetic, shift, logical operators etc. Then all the method-level mutation operators were applied to each type of program to create the mutants.
    b. For the class-level operators, programs were written for the common data structures of Linked list. A Stack class was then derived from it. Programs were then written which used these classes. These programs differed for each of the class-level mutation operators. All the class-level mutation operators were applied to all the programs to create mutants.
3. Empirical data was gathered indicating which mutation operators created a greater number of mutants as compared to other mutation operators.
4. Also, a manual inspection was conducted on all the mutants to discover the equivalent versions.
5. Java programs were then created containing the test cases for each of the method-level and class-level operators original programs created in step 2. These test cases perform branch coverage on the original program.
6. The test cases were executed against the original programs using MuJava.
7. The mutation scores were obtained.
8. Also, it was determined that how many mutants were left alive by each of the operators at the end of test execution.
9. Of the mutants left alive, the equivalent mutants were determined.
10. An empirical analysis was conducted to determine which mutation operators contribute more towards creating equivalent mutants.

The empirical data gathered and its analysis is discussed in great detail in chapters 6 and 7.

4. METHOD-LEVEL MUTATION OPERATORS

This chapter will briefly describe the method-level operators implemented in MuJava. The operators considered for MuJava modify the expressions by inserting, replacing or deleting the primitive operators [9]. MuJava contains six types of primitive operators.

1. Arithmetic operators
2. Relational operators
3. Conditional operators
4. Shift operators
5. Logical operators
6. Assignment operators

Some of these operators have short-cut versions as well. Because of this, some operators are subdivided into binary, unary and short-cut versions. In all, there are 12 method-level operators in MuJava.

Since MuJava has been created for Java programs, only operators used in Java are considered for the creation of the tool.

Category	Operator	Description
Arithmetic	AOR_B AOR_U AOR_S	Arithmetic Operator Replacement (binary) Arithmetic Operator Replacement (unary) Arithmetic Operator Replacement (short-cut)
	AOI_U AOI_S	Arithmetic Operator Insertion (unary) Arithmetic Operator Insertion (short-cut)
	AOD_U AOD_S	Arithmetic Operator Deletion (unary) Arithmetic Operator Deletion (short-cut)
Relational	ROR	Relational Operator Replacement
Conditional	COR	Conditional Operator Replacement
	COI	Conditional Operator Insertion
	COD	Conditional Operator Deletion
Shift	SOR	Shift Operator Replacement
Logical	LOR	Logical Operator Replacement
	LOI	Logical Operator Insertion
	LOD	Logical Operator Deletion
Assignment	ASR_S	Assignment Operator Replacement (short-cut)

Table 1: Method-level operators in MuJava

4.1 Arithmetic Operators

Arithmetic operators perform mathematical computations on all integers and floating-point numbers. The arithmetic operators supported in Java are:

- For binary: op + op (addition), op – op (subtraction), op * op (multiplication), op / op (division) and op % op (modulus).
- For unary: + (plus indicating positive value), - (minus indicating negative value)
- For short-cut: op++ (post-increment), ++op (pre-increment), op-- (post-decrement), --op (pre-decrement)

The arithmetic operators used in MuJava are explained below [9]:

- AOR_B / AOR_U / AOR_S: Arithmetic Operator Replacement
  These operators replace binary, unary or short-cut arithmetic operators with other binary, unary or short-cut operators, respectively.
- AOI_U / AOI_S: Arithmetic Operator Insertion
  These operators insert unary or short-cut arithmetic operators, respectively.
- AOD_U / AOD_S: Arithmetic Operator Deletion
  These operators delete unary or short-cut arithmetic operators, respectively.

4.2 Relational Operators

Relational operators compare the values of two operands. The relational operators supported in Java are:

- op > op (greater than), op >= op (greater than or equal to), op < op (less than), op <= op (less than or equal to), op == op (equal to) and op != op (not equal to).

Since these operators require two operands, only replacement is allowed for these operators. The relational operator used in MuJava is explained below [9]:

- ROR: Relational Operator Replacement
  This operator replaces relational operators with other relational operators.

4.3 Conditional Operators

Conditional operators or bitwise operators perform computations on the binary values of its operands. These operators exhibit short-circuit behavior. The conditional operators supported in Java are:

- For binary: op && op (conditional AND), op || op (conditional OR), op & op (bitwise AND), op | op (bitwise OR) and ^ (bitwise XOR).
- For unary: !op (bitwise logical complement)

The arithmetic operators in MuJava are explained below [9]:

- COR: Conditional Operator Replacement
  This operator replaces binary conditional operators with other binary conditional operators.
- COI: Conditional Operator Insertion
  This operator inserts unary conditional operators.
- COD: Conditional Operator Deletion
  This operator deletes unary conditional operators.

4.4 Shift Operators

Shift operators require two operands. They manipulate the bits of the first operand in the expression by either shifting them to the right or the left. The shift operators supported in Java are:

- op >> op (signed right shift), op << op (signed left shift) and op >>> op (unsigned right shift).

The arithmetic operators in MuJava are explained below [9]:

- SOR: Shift Operator Replacement
  This operator replaces binary shift operators with other binary shift operator.

4.5 Logical Operators

Logical operators perform logical comparisons to produce a Boolean result for comparison statements. The logical operators supported in Java are:

- For binary: op & op (AND), op | op (OR) and op ^ op (XOR).
- For unary: ~op (bitwise complement)

The arithmetic operators in MuJava are explained below [9]:

- LOR: Logical Operator Replacement
  This operator replaces binary logical operators with other binary logical operators.
- LOI: Logical Operator Insertion
  This operator inserts unary logical operators.
- LOD: Logical Operator Deletion
  This operator deletes unary logical operators.

4.6 Assignment Operators

Assignment operators set the values of an operand. The short-cut assignment operators perform a computation on the right hand operand and then assign its value to the left hand operand. The assignment operators supported in Java are:

- For short-cut: op += op (addition assignment), op -= op (subtraction assignment), op *= op (multiplication assignment), op /= op (division assignment), op %= op (modulus assignment), op &= op (bitwise AND assignment), op |= op (bitwise OR assignment), op ^= op (bitwise XOR assignment), op <<= op (right shift assignment), op >>= op (left shift assignment), op >>>= op (unsigned right shift assignment).

The arithmetic operators in MuJava are explained below [9]:

- ASR_S: Assignment Operator Replacement Short-cut
  This operator replaces short-cut assignment operators with other short-cut assignment operators.

5. CLASS-LEVEL MUTATION OPERATORS

This chapter will briefly describe the class-level operators implemented in MuJava. The operators considered for MuJava have been divided in four categories according to their usage in Object Oriented programming. The first 3 groups target features common to all OO languages. The last group includes features that are specific to Java. These groups are:

1. Encapsulation
2. Inheritance
3. Polymorphism
4. Java-specific Features

Like method-level operators, the class-level operators make changes to the program syntax by inserting, deleting or modifying the expressions under test. Operators have been defined for each category. In all, there are 29 class-level operators in MuJava. These operators are explained in detail below. All code examples have been taken from [10].

Language Feature	Operator	Description
Encapsulation	AMC	Access modifier change
Inheritance	IHD	Hiding variable deletion
	IHI	Hiding variable insertion
	IOD	Overriding method deletion
	IOP	Overriding method calling position change
	IOR	Overriding method rename
	ISI	*Super* keyword insertion
	ISD	*Super* keyword deletion
	IPC	Explicit call to parent's constructor deletion
Polymorphism	PNC	New method call with child class type
	PMD	Member variable declaration with parent class type
	PPD	Member variable declaration with child class type
	PCI	Type cast operator insertion
	PCC	Cast type change
	PCD	Type cast operator deletion
	PRV	Reference assignment with other compatible variable
	OMR	Overloading method contents replace
	OMD	Overloading method deletion
	OAN	Arguments of overloading method call change
Java-specific Features	JTI	*this* keyword insertion
	JTD	*this* keyword deletion
	JSI	*static* modifier insertion
	JSD	*static* modifier deletion
	JID	Member variable initialisation deletion
	JDC	Java-supported default constructor creation
	EOA	Reference assignment & content assignment replacement
	EOC	Reference comparison & content assignment replacement
	EAM	Accessor method change
	EMM	Modifier method change

Table 2: Class-level operators in MuJava

5.1 Encapsulation

Encapsulation in OOP deals with data hiding. It is the ability of an object to create a boundary around its data and methods. Encapsulation allows a programmer to define the access levels for various objects. For this purpose many access modifiers are used. Specifying the wrong access modifier can lead to incorrect results.

There is only one mutation operator in MuJava, which deals with encapsulation.

- AMC: Access Modifier Change

  This operator replaces the access modifiers for various instance variables and methods in a program [10]. This allows a tester to ensure that the correct level of accessibility is used in a program.

Original Code	Mutants
public Stack s;	private Stack s; protected Stack s;

5.2 Inheritance

Inheritance allows the data and methods of one class (parent class) to be used by another class (child class). Inheritance promotes code reusability.

Eight mutation operators have been developed for MuJava that deal with inheritance.

Variable shadowing allows a child class to hide or shadow the variables in the parent class. This may cause incorrect variable to be accessed. Mutation operators IHD and IHI test this issue in OOP.

- IHD: Hiding variable deletion

  This operator deletes a hiding variable in a child class. In this way, the variable in the parent class can be accessed.

Original Code	Mutants

class List {
  int size;
  …
}
class Stack extends List {
  int size;
  …..
}
	class List {
	  int size;
	  …
	}
	class Stack extends List {
	// int size;
	  …
	}

- IHI: Hiding variable insertion

  This operator inserts a hiding variable in a child class. It is reverse of IHD. Newly defined and overriding methods in a subclass reference the hiding variable whereas inherited methods reference the hidden variable as before [10].

Original Code	Mutants

class List {
  int size;
  …
}
class Stack extends List {
  …
  …
}
	class List {
	  int size;
	  …
	}
	class Stack extends List {
	  int size;
	  …
	}

A child class can modify the behavior of its parent class by creating a method with the same name and arguments as the parent class. This is called method overriding. Mutation operators IOD, IOP and IOR are used to test issues related to method overriding in MuJava.

- IOD: Overriding method deletion

  This operator deletes the entire declaration of the overriding method in the child class. All references to this function then access the method in the parent class.

Original Code	Mutants

class Stack extends List {
  …
  void push(int a){
  …
  }
}
	class Stack extends List {
	  …
	/* void push(int a){
	  …
	  } */
	}

- IOP: Overridden method calling position change

  Sometimes the overriding method in the child class may need to call the method it overrides in the parent class [10]. It is necessary to test that the method in the parent class is

called at the right point in the program otherwise it would cause the program to be in an incorrect state.

Original Code

```
class List {
    …
    void SetEnv(){
       size = 5;
       …
    }
}
class Stack extends List {
    …
    void SetEnv(){
       super.SetEnv();
       size = 10;
       …
    }
}
```

Mutants

```
class List {
    …
    void SetEnv(){
       size = 5;
       …
    }
}
class Stack extends List {
    …
    void SetEnv(){
       size = 10;
       super.SetEnv();
       …
    }
}
```

- IOR: Overridden method rename

  This operator is used to check if an overriding method adversely affects other methods [10]. It renames the method being overridden in the parent class so that the overriding method in the child class doesn't affect the method in the parent class.

Original Code

```
class List {
    …
    void f(){ … }
    void m(){
        f();
        …
    }
}
class Stack extends List {
    …
    void f(){ … }
    void g(){
        f();
        …
    }
}
```

Mutants

```
class List {
    …
    void f'(){ … }
    void m(){
        f'();
        …
    }
}
class Stack extends List {
    …
    void f(){ … }
    void g(){
        f();
        …
    }
}
```

The *super* keyword allows a programmer to access the member variables and functions of the parent class when using variable shadowing or method overriding. Mutation operators ISI and ISD are used to test issues regarding use of the *super* keyword in MuJava.

- ISI: *super* keyword insertion

  This operator inserts the *super* keyword so that any references to the variable or method go to the overridden method or variable [10].

Original Code

```
class Stack extends List {
   …
   int MyPop(){
      …
      return val * num;
   }
}
```

Mutants

```
class Stack extends List {
   …
   int MyPop(){
      …
      return val * super.num;
   }
}
```

- ISD: *super* keyword deletion

  This operator deletes the *super* keyword so that any references to the variable or method go to the overriding method or variable [10].

Original Code

```
class Stack extends List {
   …
   int MyPop(){
     …
     return val * num;
   }
}
```

Mutants

```
class Stack extends List {
   …
   int MyPop(){
     …
     return val * super.num;
   }
}
```

The constructors of the parent class are not inherited like other methods. Whenever an object of a child class is created, it automatically invokes the default constructor of the parent's class before invoking its own constructor. The child class can also use the *super* keyword to invoke a specific parent class constructor [10]. Mutation operator IPC is used to test this in MuJava.

- IPC: explicit call of a parent's constructor deletion

  This operator deletes calls to the parent class's constructor. This causes the default constructor of the parent class to be called [10].

Original Code

```
class Stack extends List {
   …
   Stack(int a){
     super(a);
     …
   }
}
```

Mutants

```
class Stack extends List {
   …
   Stack(int a){
//     super(a);
       …
   }
}
```

5.3 Polymorphism

Polymorphism allows objects to react differently to the same method. It is implemented by having many methods with the same name.

Ten mutation operators have been developed for MuJava that deal with polymorphism.

- PNC: *new* method call with child class type

  This operator changes the constructor used to instantiate an object i.e. it changes the type with which the object is instantiated. It makes the object reference refer to an object of a different type than that with which it is declared [10].

Original Code

```
Parent a;
a = new Parent();
```

Mutants

```
Parent a;
// a = new Child();
```

- PMD: member variable declaration with parent class type

  This operator changes the declared type of an object reference to the parent of the original declared type [10].

Original Code

```
Child b;
b = new Child();
```

Mutants

```
// Parent b;
b = new Child();
```

- PPD: parameter variable declaration with child class type

  This operator is similar to PPD. The only difference is that it changes the declared type of the parameter object to the parent of the original declared type [10].

Original Code

```
boolean equals (Child o){
   … }
```

Mutants

```
// boolean equals (Parent o){
   … }
```

- PCI: type cast operator insertion

  This operator changes the actual type of an object reference to the parent/child of the original declared type [10]. Different behavior is exhibited when the type of the object reference is changed for overriding methods and hiding variables.

Original Code	Mutants
Child cRef; Parent pRef = cRef; PRef.toString();	Child cRef; Parent pRef = cRef; // ((Child)pRef).toString();

- PCD: type cast operator deletion

  This operator is the opposite of PCI. It deletes the type casting operator from an object reference.

Original Code	Mutants
Child cRef; Parent pRef = cRef; ((Child)pRef).toString();	Child cRef; Parent pRef = cRef; // pRef.toString();

- PCC: cast type change

  This operator changes the type of the casting operator. The type is changed to the ancestors or descendants of the original type.

Original Code	Mutants
((Parent)pRef).toString();	// ((Child)pRef).toString();

- PRV: reference assignment with other compatible type

  This operator changes the assignment to an object reference. This assignment is changed to other compatible types of the object under question that are usually this object's subclasses.

Original Code	Mutants
Object obj; String s = "Hello"; Integer I = new Integer(4); obj = s;	Object obj; String s = "Hello"; Integer i = new Integer(4); // obj = i;

Method overloading allows two or more methods of the same class to have the same name but different arguments. Mutation operators OMR, OMD and OAN are used to test issues related to method overloading in MuJava.

- OMR: overloading method contents change

  This operator checks if overloaded methods are invoked correctly. It does so by replacing the body of one method with the body of another method that has the same name [10].

Original Code	Mutants

class List {
…
void Add(int e){ … }
void Add(int e, int n){
…
}
}
	class List {
	…
	void Add(int e){ … }
	void Add(int e, int n){
	// this.Add(e);
	}
	}

- OMD: overloading method deletion

  This operator works by deleting each of the overloading methods one by one. This operator ensures coverage of all the overloaded methods [10].

Original Code	Mutants

class Stack extends List {
…
void Push(int i){ … }
void Push(float i){
…
}
}
	class Stack extends List {
	…
	// void Push(int i){ … }
	void Push(float i){
	…
	}
	}

- OAN: argument of overloading method change

  This operator works by changing the order or the number of arguments in method invocations. The number of arguments is changed only if there are other overloaded methods that accept the new argument list [10].

Original Code	Mutants

s.push(0.5, 2);	s.push(2, 0.5);
	s.push(2);
	s.push(0.5);
	s.push();

5.4 Java-specific Features

This group of operators is used to cover those aspects of Java that are not covered by other object-oriented programming languages.

Ten mutation operators have been developed for MuJava that deal with Java specific features only.

- JTI: *this* keyword insertion

  This operator inserts the *this* keyword. It helps check if the member variables are used correctly if they are hidden by method parameters [10].

Original Code	Mutants

class Stack {
  int size;
  …
  void setSize(int size){
  this.size = size;
  …
  }
}
	class Stack {
	  int size;
	  …
	  void setSize(int size){
	// this.size = this.size;
	  …
	  }
	}

- JTD: *this* keyword deletion

  This operator deletes *this* keyword anywhere it occurs in the program. It is the opposite of JTI [10].

Original Code	Mutants

class Stack {
  int size;
  …
  void setSize(int size){
  this.size = size;
  …
  }
}
	class Stack {
	  int size;
	  …
	  void setSize(int size){
	// size = size;
	  …
	  }
	}

- JSI: *static* modifier insertion

  This operator adds the *static* modifier to change instance variables to class variables [10].

Original Code	Mutants
public int s = 100;	// public static int s = 100;

- JSD: *static* modifier deletion

  This operator removes the *static* modifier to change class variables to instance variables [10].

Original Code	Mutants
public static int s = 100;	// public int s = 100;

- JID: member variable initialization deletion

  This operator deletes any initialization in the variable declaration and in class constructors. The variables are then initialized to the default values in Java [10].

Original Code	Mutants

class Stack {
  int size = 100;
  …
  Stack(){ … }
}
	class Stack {
	  int size = 100;
	  …
	  Stack(){ … }
	}

- JDC: Java-supported default constructor create

  This operator deletes any implemented default constructors created by the programmer. It creates Java's own default constructor for use [10].

Original Code	Mutants

class Stack {
  …
  Stack(){ … }
}
	class Stack {
	  …
	  Stack(){ … }
	}

- EOA: reference assignment and content assignment replacement

  This operator replaces an assignment of a pointer reference with a copy of the object using the Java convention of a *clone()* method [10]. This method creates a copy of the contents of an object and returns a reference to a new object.

Original Code	Mutants

Stack s1, s2;
s1 = new Stack();
s2 = s1;
	Stack s1, s2;
	s1 = new Stack();
	s2 = s1.clone();

- EOC: reference comparison and content comparison replacement

  This operator is used to check the fault of comparing the contents of an object with the object's references. EOC helps to resolve this issue by using the *equals()* function for comparing the references of two objects [10].

Original Code	Mutants
Integer i1 = new Integer(7); Integer i2 = new Integer(7); boolean b = (i1 == i2);	Integer i1 = new Integer(7); Integer i2 = new Integer(7); boolean b = (i1.equals(i2));

- EAM: accessor method change

  This operator changes the accessor method name for other compatible accessor method names [10].

Original Code	Mutants
point.getX();	point.getY();

- EMM: modifier method change

  This operator changes the modifier method name for other compatible modifier method names [10].

Original Code	Mutants
point.setX(2);	point.setY(2);

6. EMPIRICAL ANALYSIS

For this project, a detailed empirical analysis was carried out for each type of mutation operators. Different strategies were adopted for method-level operators and class-level operators to create mutants using MuJava. This chapter explains in detail how the mutants were created and the results obtained using MuJava.

The aim of this empirical analysis is to determine which mutation operators are more probable to create equivalent mutants. The test cases created for each of these operators ensure that branch coverage is achieved.

6.1 Method-level Mutation Operators

There are 12 method-level operators in MuJava, which have been explained in detail in chapter 4. To develop mutants for these operators the following strategy was used:

1. One type of mutation operator was chosen, e.g. Arithmetic mutation operators i.e. AOR, AOI, AOD.
2. A simple Java program (say 'ArithOper.java') was developed, which used this type of mutation operator, in this case, arithmetic operators.
3. MuJava was started and all the method-level operators were selected.
4. Using MuJava, mutants were created for 'ArithOper.java'.
5. Another java program was created (say 'ArithOperTest.java') which contained the test cases for 'ArithOper.java'. These test cases must exercise branch coverage for 'ArithOper.java'.
6. 'ArithOperTest.java' was run against 'ArithOper.java' to determine how many mutants were killed and the mutation score was calculated.
7. Of the mutants still alive at the end of the execution, the equivalent mutants were determined manually.
8. The entire process from steps 1 to 7 was repeated to develop java programs for each type of method-level mutation operator i.e. relational, conditional, shift, logical and assignment.

Using the above process, the data collected in the first instance indicates the number of mutants created for each method-level operator. Following table explains this.

Operator	Number of Mutants
AORB	4
AORS	0
AOIU	10
AOIS	120
AODU	0
AODS	0
ROR	20
COR	4
COD	0
COI	2
SOR	2
LOR	2
LOI	33
LOD	0
ASRS	4

Table 3: Number of Mutants for Method-level Operators

6.1.1 Observations

When test cases were executed for all the mutants created by method-level operators, some interesting observations were made:

- No mutants were created for some operators as indicated in table 3.

- The greatest number of mutants, i.e. 120, were created for AOIS, which account for almost 59% of the total number of mutants created for method-level operators.
- MuJava could not compile some mutants that were created by the mutation operator LOI. These mutants contribute as programs that fail the software. So it can be said that MuJava has also helped to identify some of the syntactic changes in the program, which make the program fail. These mutants were not considered when test cases were executed for 'LogicalOper.java'.

6.1.2 An Analysis of the Mutants killed/alive

Test cases were created for each type of method-level operator. When these test cases were executed against the mutants, the following results were obtained (Since, AOIS was dominating the entire graph, two graphs have been drawn, one with and the other without AOIS to give a deeper insight into the analysis.):

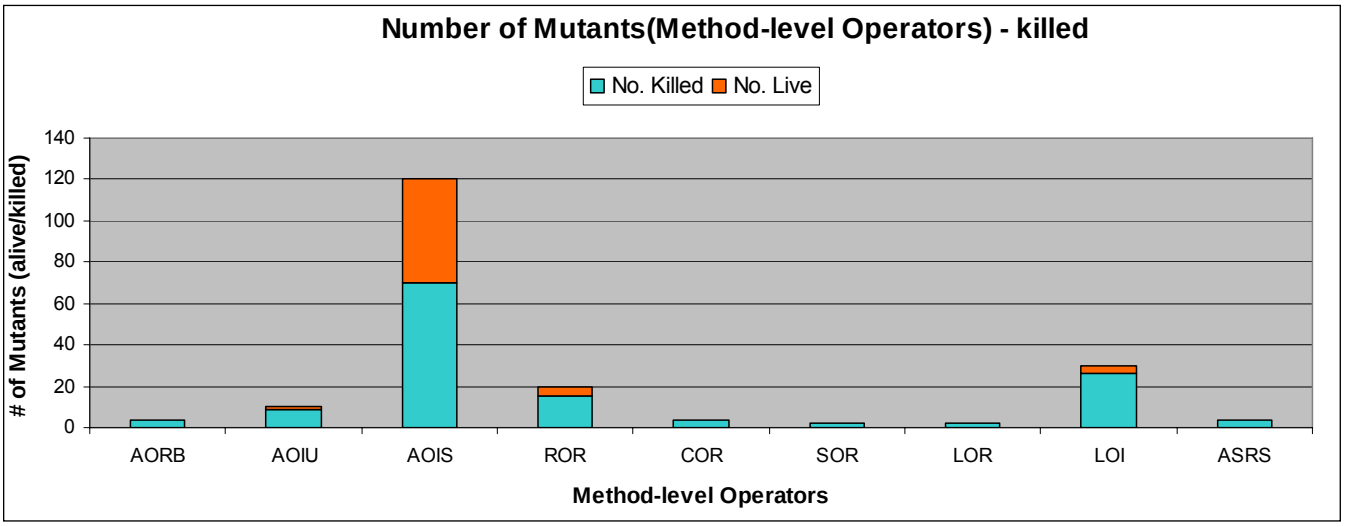

Figure 11: Number of Mutants killed by Method-level Operators

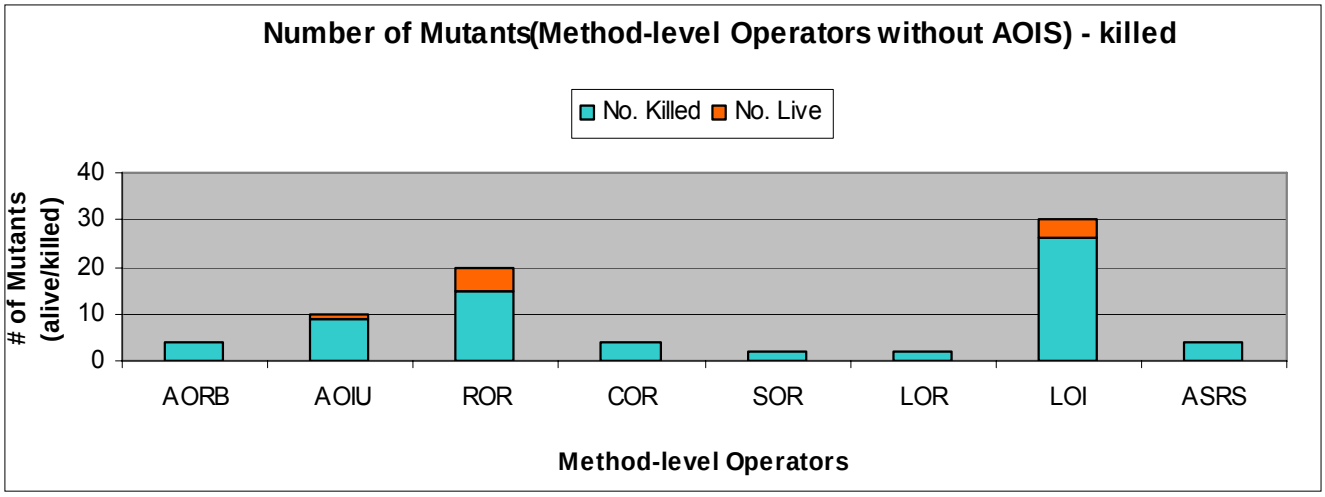

Figure 12: Number of Mutants (excluding AOIS) killed by Method-level Operators

From figure 11, it can be seen that a large number of mutants were created for AOIS, ROR and LOI. From intuition, it can be assumed that most of the equivalent mutants *might* belong to these operators.

From figures 11 & 12, it can be seen that the mutants, which are still alive at the end of the execution, belong to the operators AOIU, ROR, LOI and AOIS. Out of these, the majority of live mutants belong to those created by AOIS. The figure below shows the percentage of live mutants for each type of method-level mutation operator created by AOIS.

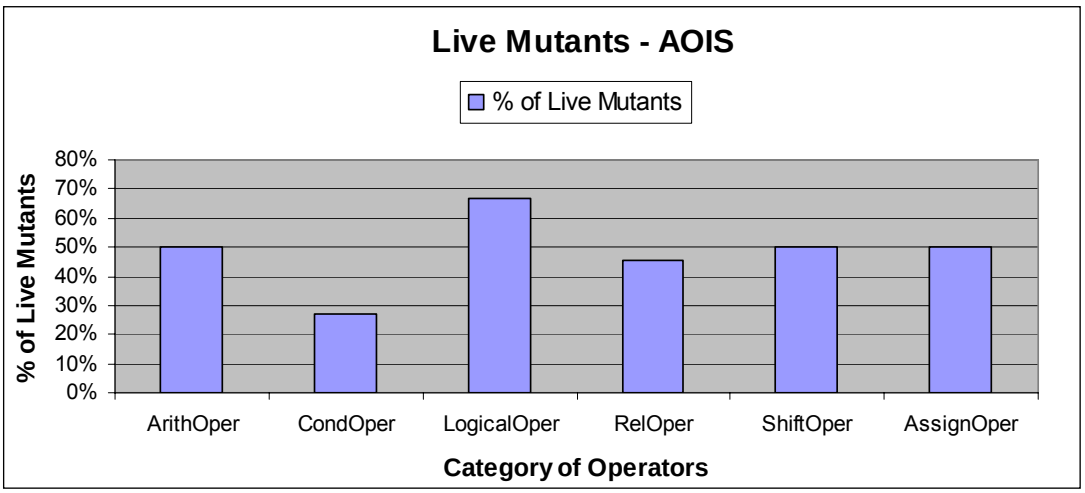

Figure 13: Percentage of live mutants created by operator AOIS

It can be observed that in most cases, almost 50% of the mutants, which are still alive at the end of executing the test cases, are the mutants created by method-level operator AOIS.

An analysis of the mutants that are equivalent from the mutants that are still alive is given in chapter 7.

6.2 Class-level Mutation Operators

There are 29 method-level operators in MuJava, which have been explained in detail in chapter 5. To develop mutants for these operators the following strategy was used:

1. A simple java program for the common data structure linked list was written. A stack class was then written which was derived from linked list. The main program using these data structures varied for each type of operator. The hierarchy for these programs can be shown as:

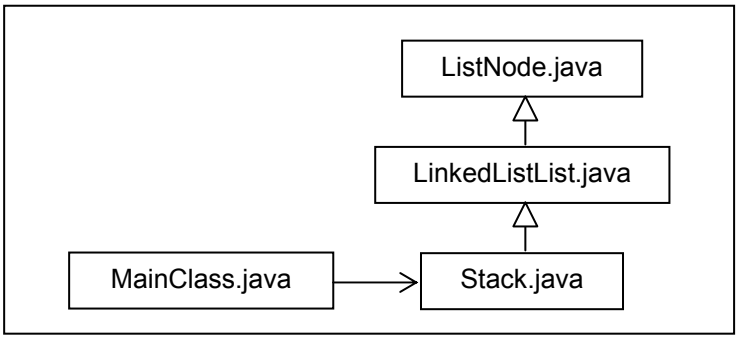

Figure 14: Structure of programs used for Class-level operators

2. Different versions of the three Java programs were created for each of the class-level mutation operators.
3. MuJava was started and all the class-level operators were selected to create mutants for the Java programs.
4. Another Java program (say 'MainClassTest.java') was created containing all the test cases to be applied to 'ListNode.java', 'LinkedList.java', 'Stack.java' and 'MainClass.java'. These test cases must exercise branch coverage.
5. 'MainClassTest.java' was then executed against the other 4 programs to determine how many mutants were killed and the mutation score was then calculated.
6. Of the mutants still alive at the end of the execution, the mutants that were equivalent were determined manually.
7. The entire process from steps 1 to 6 was repeated to develop Java programs for all the class-level mutation operators.

Using the above process the data collected in the first instance indicates the number of mutants created for each class-level operator.

Operator	Number of Mutants
AMC	-
IHD	7
IHI	15
IOD	17
IOP	1
IOR	2
ISI	1
ISD	1
IPC	1
PNC	1
PMD	2
PPD	1
PCI	3
PCC	0
PCD	0
PRV	361
OMR	4
OMD	1
OAN	4
JTI	3
JTD	2
JSI	40
JSD	0
JID	4
JDC	14
EOA	0
EOC	1
EAM	2
EMM	2

Table 4: Number of Mutants for Class-level Operators

6.2.1 Observations

When test cases were executed for all the mutants created by class-level operators, the following observations were made:

- No mutants were created for some operators as indicated in table 5.
- The operator AMC has not been implemented in the current version of MuJava although the literature supporting the tool discusses it.
- The greatest numbers of mutants, i.e. 356, were created for PRV, which account for almost 72% of the total number of mutants created for class-level operators.
- As for method-level operator, LOI, MuJava could not compile some mutants that were created by the mutation operator PRV and PMD. These mutants contribute as programs that fail the software. So it can be said that MuJava has also helped to identify some of the syntactic changes in the program, which make the program fail.
- For PMD, all the mutants that were created made the software crash during execution. Thus, the mutants created by PMD were disregarded completely when executing test cases.
- As for PRV, some of the mutants created incurred failure. The summary for PRV is given in the figure below:

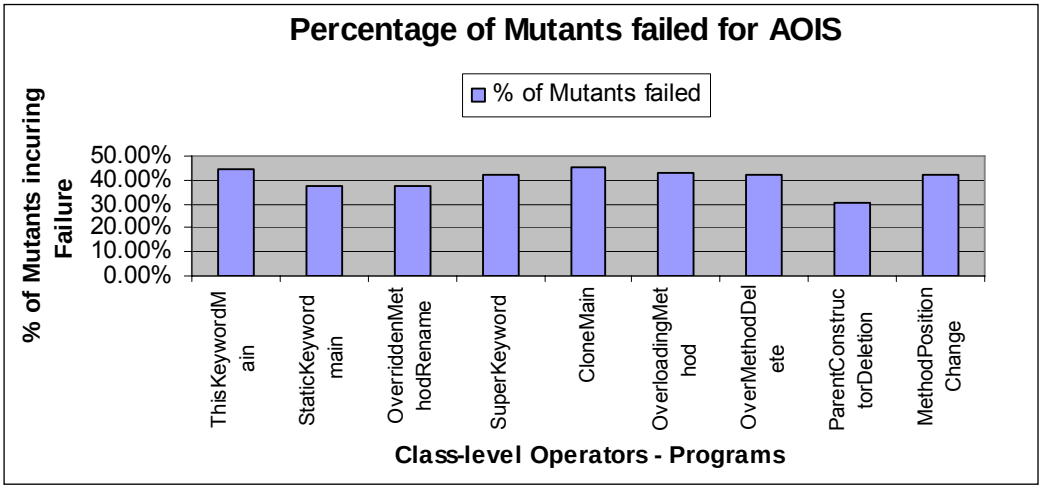

Figure 15: Percentage of mutants incurring failure created by PRV

This graph shows that almost 45% of the total mutants created by PRV for each type of program, failed the software. This means that the mutants that were created made the program fail and can thus, be disregarded when executing test cases to determine mutation scores.

6.2.2 An Analysis of the Mutants killed/alive

Test cases were created for each type of class-level operator. When these test cases were executed against the mutants, the following results were obtained (as in method-level operators, since, PRV was dominating the entire graph, two graphs have been drawn with and without PRV do give a deeper insight into the analysis.):

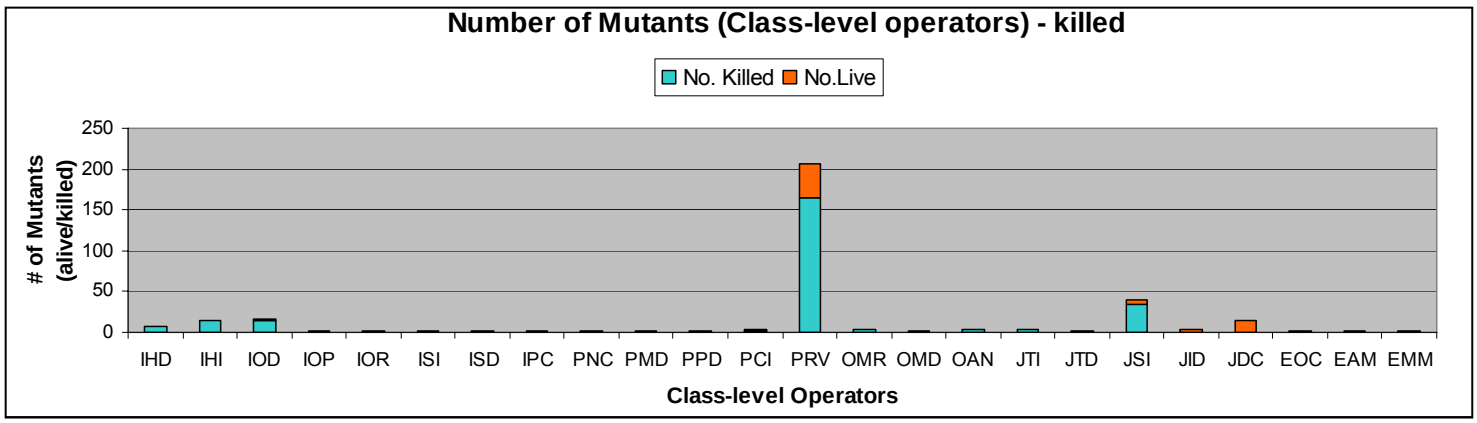

Figure 16: Number of Mutants killed by Class-level Operators

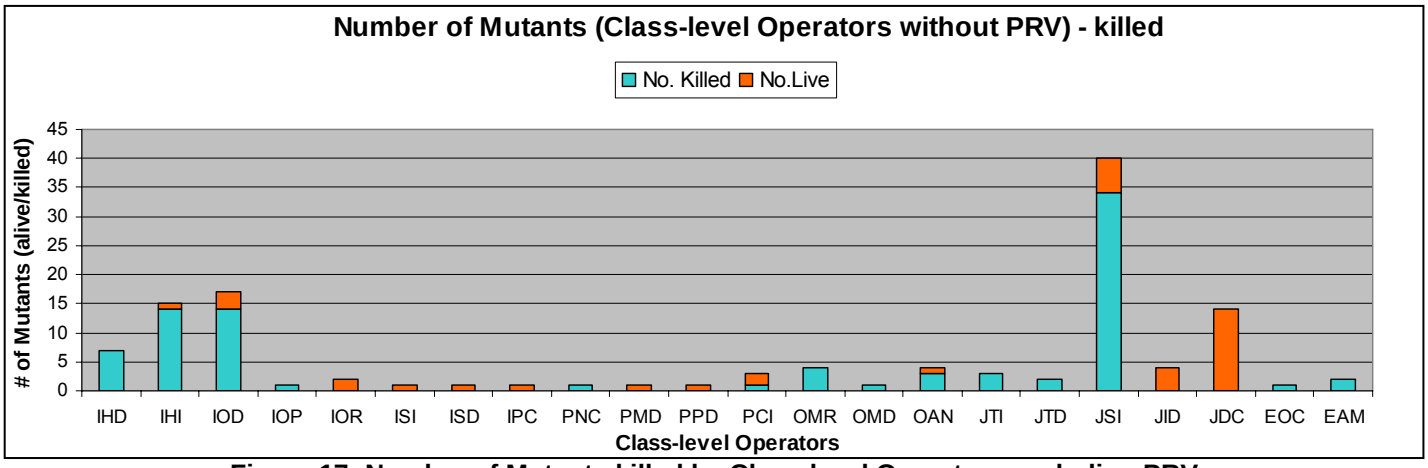

Figure 17: Number of Mutants killed by Class-level Operators excluding PRV

From figure 16, it can be seen that a large number of mutants were created for PRV, JSI, IOD and IHI as compared to the other operators. From intuition, it can be assumed that most of the equivalent mutants *might* belong to these operators.

However, class-level operators behave differently as compared to method-level operators. If it is assumed that most of the mutants that may be still alive at the end of test execution are those created by PRV, JSI and IOD, the reverse happens when testing is performed in practice. From figures 16 & 17, it can be seen that the mutants that are still alive at the end of the execution, belong to the operators PRV, JDC, JID and some other operators. Out of these, the majority of live mutants belong to those created by PRV.

An analysis of the mutants that are equivalent to the mutants that are still alive for class-level operators is given in the next chapter.

7. AN EVALUATION OF EQUIVALENT MUTANTS

For this project, all the test cases have been written such that they ensure that all the branches in the sample programs are covered. This is performed to determine adequacy of the test criterion. To determine the equivalent mutants for method-level operators and class-level operators, a manual inspection of the mutants left alive was carried out. There was no definite process.

For each operator, graphs have been created which show the total number of mutants created, the number of mutants killed, the number of equivalent mutants and the number of mutants that failed the software. These results will help determine which mutation operators are prone to develop more equivalent mutants. Also, scenarios were identified for certain mutation operators. These indicate the situations in which some of the mutation operators were more likely to create equivalent mutants. At the end of the analysis 3 types of results were obtained:

1. The percentage of mutants killed per mutation operator
2. The percentage of non-equivalent mutants per mutation operator
3. The percentage of equivalent mutants per mutation operator

7.1 Evaluation of Method-level Mutation Operators

According to table 3 in chapter 6, MuJava created no mutants for the method-level mutation operators AORS, AODU, AODS, COD and LOD. An analysis has been performed for the remaining operators namely, AORB, AOIU, AOIS, ROR, COR, COI, SOR, LOR, LOI and ASRS. Graphs have been created. These graphs have been divided in arbitrary groups of 2. The last graph depicts the analysis for the method-level operator AOIS since it dominates the behavior of the other operators. This analysis is explained in more detail below:

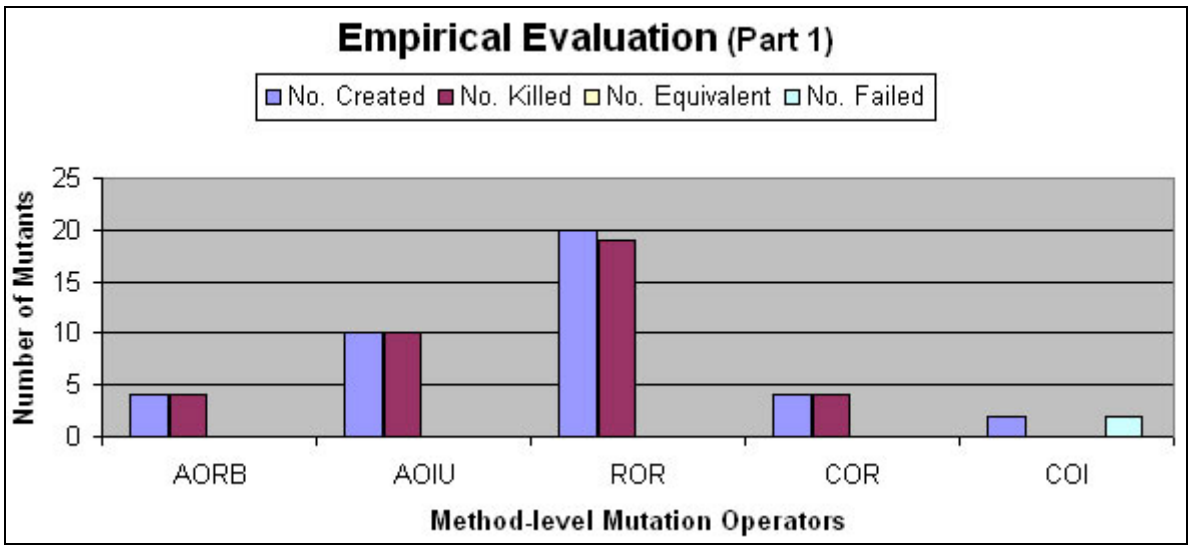

Figure 18: Evaluation of Method-level Operators AORB, AOIU, ROR, COR & COI

Figure 18 shows an evaluation of the operators AORB, AOIU, ROR, COR and COI.

MuJava created a very small number of mutants for AORB. But all the mutants that were created were killed successfully. None of the mutants that were created were found to be equivalent. Also, no mutants caused MuJava to fail. Thus, 100% mutation score was achieved.

As compared to AORB, AOIU still created a considerable number of mutants. But the behavior exhibited by AOIU was identical to that of AORB. 100% of the mutants created for AOIU were killed by MuJava. None of the mutants that were created were found to be equivalent.

The operator ROR works by replacing relational operators with all other relational operators. A significantly good number of operators were created for ROR (20). Of these, 19 mutants were killed. None of the mutants were found to be equivalent. The one mutant which was not killed was just difficult to kill. It was not equivalent.

COR is another mutation operator which replaces conditional operators with other conditional operators. Like AORB, COR also created a very small number of mutants. But all the mutants that were created were killed. None of the mutants were found to be equivalent.

MuJava created only 2 mutants for COI. But all of these mutants failed the software. Thus, none of the mutants were found to be equivalent and none were killed.

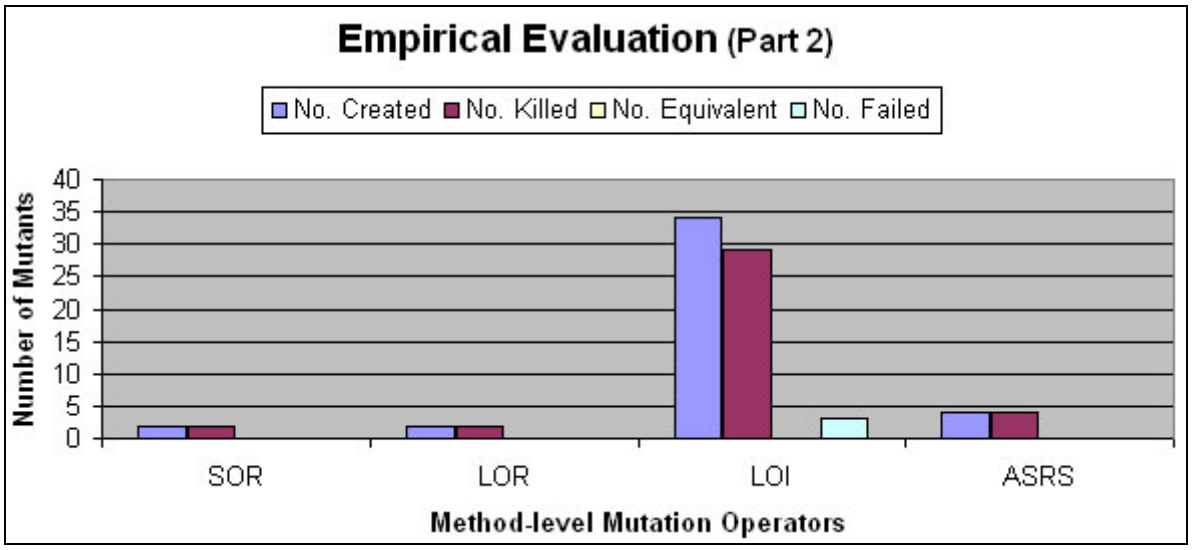

Figure 19: Evaluation of Method-level Operators SOR, LOR, LOI & ASRS

Figure 19 shows an evaluation of the operators SOR, LOR, LOI and ASRS.

A very small number of mutants were created for SOR. The test set created, successfully killed all the mutants. None of the mutants that were created were found to be equivalent.

Similarly, an extremely small number of mutants were created for LOR. None of the mutants that were created were semantically similar to the original program. Also, all the mutants were killed successfully. Thus, 100% mutation score was achieved.

LOI exhibited interesting behaviour. A reasonable number of mutants was created for LOI. Of the 34 mutants created 29 were killed. But 3 of the mutants caused MuJava to fail. Although 85.2% of the mutants were killed, the remaining did not help detect any faults. Most of the remaining mutants caused the program to fail.

ASRS is an operator which replaces short-cut assignment operators with other short-cut assignment operators. All of the mutants that were created for ASRS were killed. No mutants were equivalent.

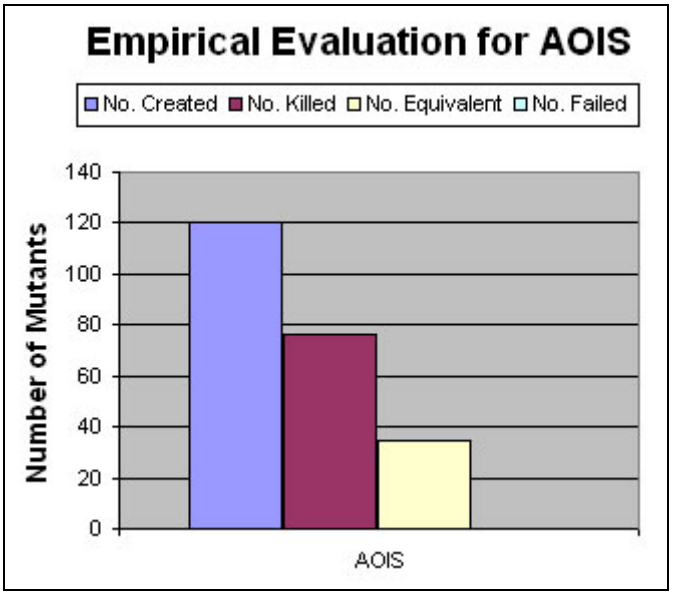

Figure 20: Evaluation of Method-level Operator AOIS

The maximum numbers of mutants were created for AOIS as shown in figure 20. Of the 120 mutants created, 76 were killed giving a mutation score of 63.33% which is not considered too good. Of the remaining mutants, all were found to be equivalent. Hence, 29.17% mutants created for AOIS were unable to capture program faults. This is in agreement to the assumption made in chapter 6 that if more mutants are left alive by AOIS they are more likely to be equivalent.

7.1.1 Summary of Method-level Mutation operators

The following table summarizes the behaviour of method-level mutation operators. It shows the percentage of mutants killed for each mutation operator. It also indicates the percentage of non-equivalent mutants against the percentage of equivalent mutants.

Method-level Operators	Percentage of Mutants Killed	Percentage of Non-equivalent Mutants	Percentage of Equivalent Mutants
AORB	100%	100%	0%
AOIU	100%	100%	0%
AOIS	63.33%	70.83%	29.17%
ROR	95%	100%	0%
COR	100%	100%	0%
COI	0%	100%	0%
SOR	100%	100%	0%
LOR	100%	100%	0%
LOI	96.67%	100%	0%
ASRS	100%	100%	0%

Table 5: Analysis Summary of Method-level Mutation Operators

This table clearly indicates that the only method-level operator which created equivalent mutants was AOIS. This is in partial agreement with the observation made in chapter 6. It was assumed that operators AOIS, ROR and LOI are most probable to create equivalent mutants because they formed the majority of all the mutants that were created for method-level mutation operators. But it can be supported with evidence that the only equivalent mutants produced were those created for AOIS. As for ROR and LOI, a mutation score of 95% and 96.67% was achieved, respectively. The remaining operators performed well and successfully killed all the mutants. However, all the mutants that were created for COI caused the software to fail. Also, the mutants that were left alive by LOI, 75% of them failed MuJava.

Since AOIS created equivalent mutants, some scenarios were discovered under which this operator always creates equivalent mutants. These scenarios can be used to modify the AOIS operator such that it does not create equivalent mutants. This will reduce the computational cost in creating and executing the surplus mutants.

7.1.2 Scenarios for AOIS

Many situations were observed whilst determining equivalent mutants for AOIS. Some of these situations always tend to create equivalent mutants. As a reminder, the AOIS operator works by inserting pre-increment/decrement or post-increment/decrement operators. Most of the equivalent mutants are created by post-increment/decrement operators. These have been chalked out below:

Original Program	Equivalent Mutants

1 public class test {
2 public void abc() {
3 int var1, var2;
4 int result;
5 result = var1 + var2;
6 System.out.println(result);
7 }
8 }
	1 public class test {
	2 public void abc() {
	3 int var1, var2;
	4 int result;
	5 result = var1 + var2;
	6 System.out.println(result++);
	6 System.out.println(result--);
	7 }
	8 }

Figure 21: Equivalent Mutants for method-level operator AOIS

Two of the mutants created for this program are shown. Now, any change in the value of a variable after it has been displayed will not make any change to the output produced by the mutant when compared to the original program. Also, an equivalent mutant will be created if the variable, whose value has been changed, is not being used after it has been displayed. An example of this is:

Non-equivalent Mutant	Equivalent Mutant

1 public class test {
2 public void abc() {
3 int var1, var2;
4 int result;
5 result = var1 + var2;
6 System.out.println(result++);
7 *result = var1 * 10;*
8 }
9 }
	1 public class test {
	2 public void abc() {
	3 int var1, var2;
	4 int result;
	5 result = var1 + var2;
	6 System.out.println(result++);
	7 *var1 += 10;*
	8 }
	9 }

Figure 22: Scenario in which AOIS creates non-equivalent and equivalent mutants

For the non-equivalent mutant, the variable *result* is used again on line 7 after the use of the AOIS operator on line 6. However, the variable *result* is not reused on line 7 for the equivalent mutant.

In summary, whenever a post-increment/decrement operator is used to create a mutant, there are two situations in which AOIS always creates equivalent mutants:

1. When the value of a variable is modified after it is displayed for output.
2. When the variable with the changed value is not modified by the code at any further point in the mutant.

7.2 Evaluation of Class-level Mutation Operators

According to table 4 in chapter 6, MuJava created no mutants for the class-level mutation operators PCC, PCD, JSD and EOA. An analysis has been performed for the remaining operators namely, IHD, IHI, IOD, IOP, IOR, ISI, ISD, IPC, PNC, PMD, PPD, PCI, PRV, OMR, OMD, OAN, JTI, JTD, JSI, JID, JDC, EOC, EAM and EMM. Like method-level operators, the graphs for class-level operators have been divided in arbitrary groups of 4. The last graph depicts an analysis of the operator PRV because it dominates all the other class-level mutation operators. The analysis for these operators is explained in more detail below:

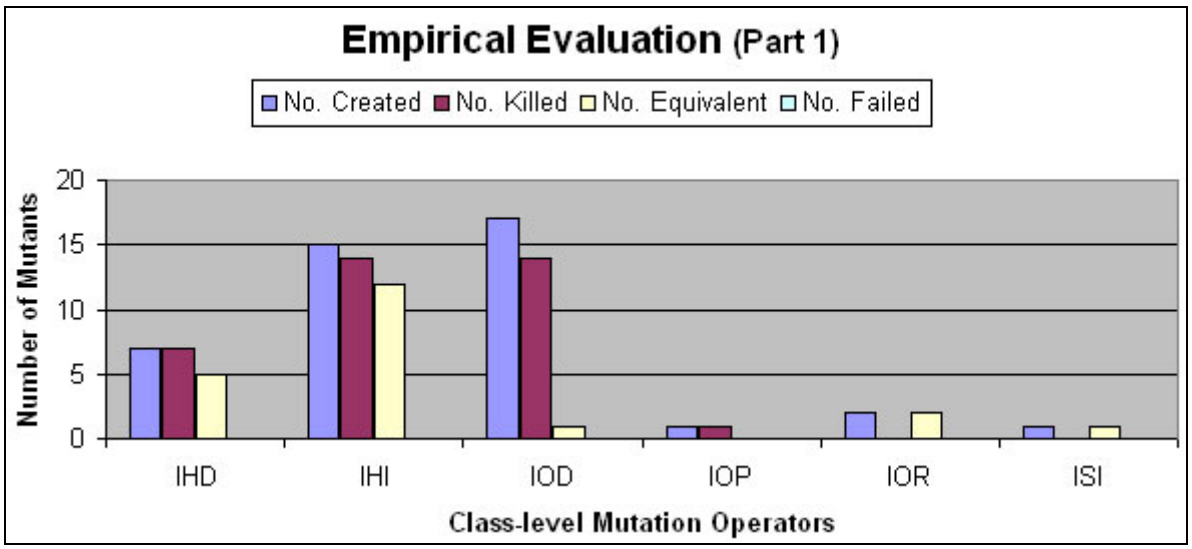

Figure 23: Evaluation of Class-level Operators IHD, IHI, IOD, IOP, IOR & ISI

Figure 23 shows an evaluation of the operators IHD, IHI, IOD, IOP, IOR and ISI.

A relatively good number of mutants were created for IHD. Of all the mutants created, 71.43% were equivalent. But all the mutants that were created were killed successfully. 100% mutation score was achieved using the test set created.

IHI is another class-level operator belonging to the category of inheritance. 15 mutants were created for IHI. 93.33% of the mutants that were created were killed successfully by the test set created.

The number of mutants created for IOD was greater than those created for IHI or IHD. 14 of the 17 mutants created were killed successfully. Interestingly, only one mutant was found to be equivalent. Interestingly, the same mutant was also found to be alive amongst the rest of the alive mutants.

The mutants created for IOP were very few as compared to the other operators. But the success achieved in killing these mutants is much greater than that for other operators. 100% mutation score was achieved. None of the mutants created for IOP were equivalent. Also, none of the mutants failed the software.

Only 2 mutants were created for IOR. All the mutants created were equivalent. None of these mutants were killed successfully. Also, none of them failed MuJava.

ISI works by inserting the *super* keyword so that any call to the variable being referenced goes to the parent class variable. ISI depicted behaviour similar to that of IOR. Although the number of mutants created for IOR was slightly more than those created for ISI, all the mutants that were created for both operators turned out to be equivalent. MuJava killed none of the mutants.

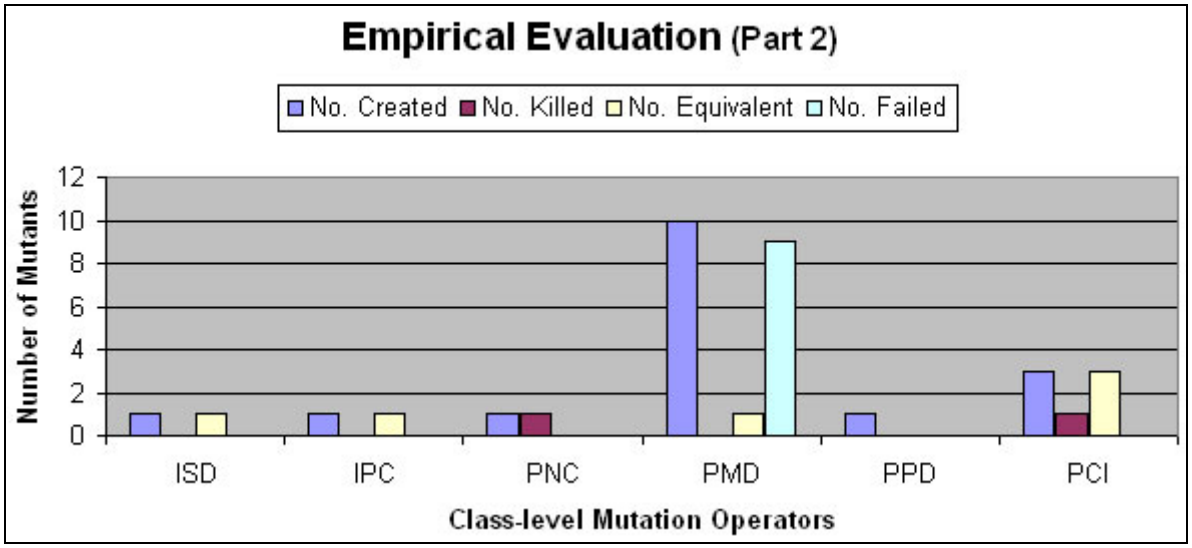

Figure 24: Evaluation of Class-level Operators ISD, IPC, PNC, PMD, PPD & PCI

Figure 24 shows an evaluation of the operators ISD, IPC, PNC, PMD, PPD and PCI.

Very few mutants were created for ISD. None of these mutants were killed. Also, none of the mutants failed. But 100% of the mutants created were equivalent.

Only 1 mutant was created for PNC. But the test set successfully detected the fault in this mutant and killed it. This mutant was non-equivalent.

However, PMD exhibited totally different behaviour. A considerable number of mutants were created for PMD as compared to ISD, IPC and PNC. Yet, MuJava killed no mutants. 90% of the mutants that were left alive failed the software. The remaining 10% were equivalent. Of all the class-level operators examined till now, PMD is the first operator which failed the software.

This graph in figure 24 shows that only one mutant was created for PPD. However, this mutant was not found to be equivalent. It was hard to kill.

In contrast, some of the mutants were created for PCI were killed. It also shows that all the mutants that were created were equivalent. MuJava successfully killed 33.33% of those mutants.

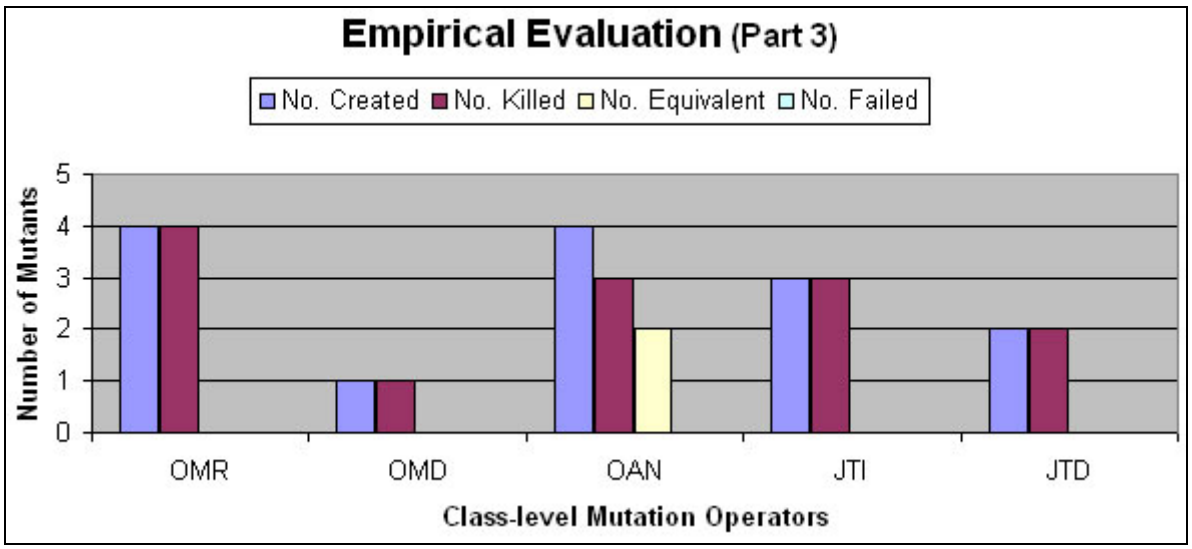

Figure 25: Evaluation of Class-level Operators OMR, OMD, OAN, JTI & JTD

Figure 25 shows an evaluation of the operators OMR, OMD, OAN, JTI and JTD. This figure is interesting in the sense that it appears that these operators created a considerably greater number of mutants as compared to the operators in figures 23 and 24. But the highest scale on the y axis is only 5. So it can be justifiably said that a very small number of mutants were created for operators OMR, OMD, OAN, JTI and JTD as well.

All the mutants created for OMR were killed successfully. 100% mutation score was achieved. None of these mutants were found to be equivalent. Also, none of the mutants created failed the software.

OMD exhibits similar behaviour as OMR. Figure 25 shows that 100% mutation score was achieved for OMD. All the mutants that were created have been killed using MuJava.

The behaviour exhibited by OAN was very different. The figure shows that 75% of the mutants that were created for OAN were killed. 25% of the mutants that were killed were equivalent. The remaining 25% mutants were left alive at the end of test case execution and were also found to be equivalent.

JTI and JTD show identical behaviour as seen in figure 25. All the mutants that were created for JTI and JTD were killed. This means that 100% mutation score was achieved for JTI and JTD using MuJava. None of the mutants were found to be equivalent. Also, none of them failed the software.

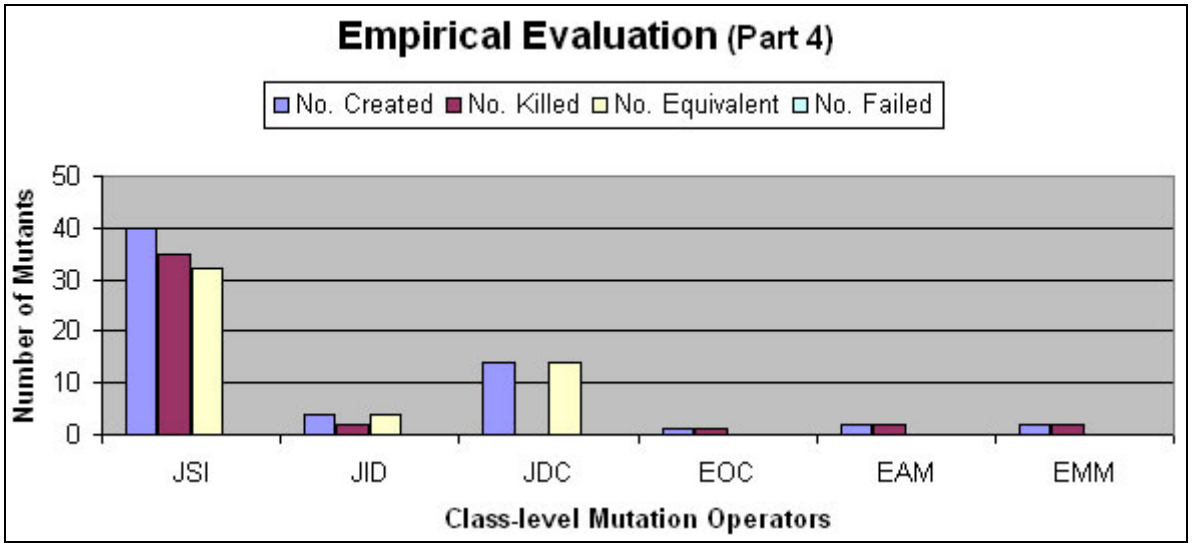

Figure 26: Evaluation of Class-level Operators JSI, JID, JDC, EOC, EAM & EMM

Figure 26 shows an evaluation of the operators JSI, JID, JDC, EOC, EAM and EMM.

JSI was one of the class-level mutation operators for which a considerably large number of mutants was created. 80% of the mutants created were found to be equivalent. But 87.5% of the total number of mutants were killed successfully. All the mutants that were left alive were found to be equivalent. None of the mutants failed the software.

JID revealed very different behaviour than JSI. Figure 26 shows that MuJava killed 50% of the mutants that were created for JID. But, interestingly, all the mutants were found to be equivalent.

JDC also created a good number of mutants. All the mutants that were created were found to be equivalent. None of the mutants were killed by the test set thus giving a mutation score of 0%.

EOC is another operator which created a very small number of mutants. However, all the mutants that were created for EOC were killed by MuJava to achieve a mutation score of 100%. None of the mutants were equivalent.

EAM and EMM exhibit identical behaviour as shown in figure 26. All the mutants that were created for both the operators were killed successfully using MuJava. Thus, a mutation score of 100% was achieved. Also, none of the mutants were equivalent.

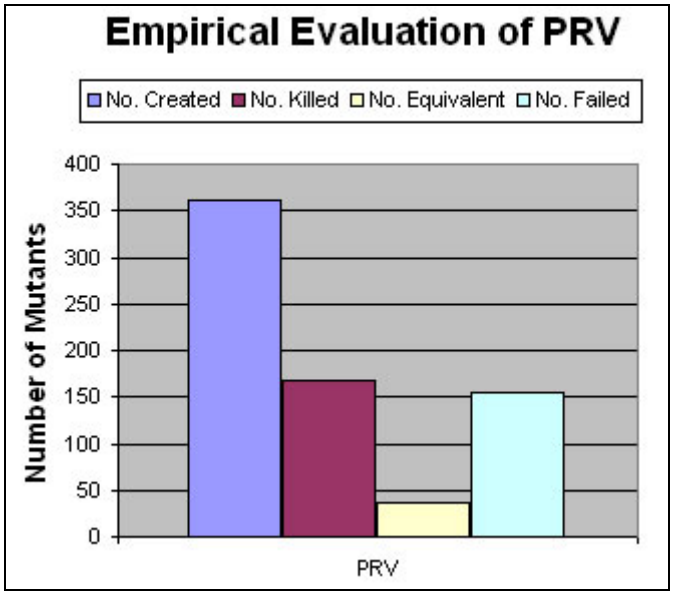

Figure 27: Evaluation of Class-level Operator PRV

Of all the class-level mutation operators, PRV exhibits the most interesting behaviour. The majority of mutants were created for this operator. However, an alarming 42.93% of these mutants failed the software. Of the remaining mutants, 81.07% mutants were killed. The rest of the mutants were found to be equivalent as depicted in figure 18. Also, of the non-equivalent mutants, 7.6% were hard to kill mutants.

7.2.1 Summary of Class-level Mutation operators

The following table summarizes the behaviour of class-level mutation operators. It shows the percentage of mutants killed for each mutation operator. It also indicates the percentage of non-equivalent mutants against the percentage of equivalent mutants.

Class-level Operators	Percentage of Mutants Killed	Percentage of Non-equivalent Mutants	Percentage of Equivalent Mutants
IHD	100%	28.57%	71.43%
IHI	93.33%	20%	80%
IOD	82.35%	94.12%	5.88%
IOP	100%	100%	0%
IOR	0%	0%	100%
ISI	0%	0%	100%
ISD	0%	0%	100%
IPC	0%	0%	100%
PNC	100%	100%	0%
PMD	0%	0%	100%
PPD	0%	0%	0%
PCI	33.33%	0%	100%
PRV	81.07%	82.52%	17.48%
OMR	100%	100%	0%
OMD	100%	100%	0%
OAN	75%	50%	50%
JTI	100%	100%	0%
JTD	100%	100%	0%
JSI	87.5%	20%	80%
JID	50%	0%	100%
JDC	0%	0%	100%
EOA	100%	100%	0%
EAM	50%	100%	0%
EMM	50%	100%	0%

Table 6: Analysis Summary of Class-level Mutation Operators

As opposed to method-level mutation operators, class-level operators created a considerably large number of equivalent mutants. This is due to the design of the operators which affected the operation of the sample programs.

This table shows that 14 of the 24 class-level operators created mutants which were equivalent. An interesting observation made was that operators ISI, ISD and IPC, which created only one mutant each, were found to be equivalent. Hence, the discovery of 100% of equivalent mutants is not as alarming as it may seem when compared to other class-level operators.

The maximum number of mutants (361) was created for PRV as indicated in table 4 chapter 6. This number raises an immediate concern that since these mutants form the majority of the mutants created, they are most probable to be equivalent. But interestingly, a large number of these mutants failed MuJava as discussed earlier. Of the remaining mutants, 81.06% were killed successfully. Also, of all the mutants created for PRV, 17.48% were found to be equivalent.

It can be seen that operators JID and JDC created mutants which were all equivalent. It is interesting to note that these operators work by deleting variable initialization or creating default constructors, respectively.

Of the remaining operators, the greatest percentage of equivalent mutants belonged to operators IHD, IHI and JSI. This indicates that more equivalent mutants created for programs which make use of variable shadowing, method overriding and use of static keyword.

In addition to determining which operators create equivalent mutants, some operators also failed MuJava. These operators are PMD and PRV. Of these, almost 42.93% mutants created for PRV failed. However, the number of mutants created for PMD was significantly small as compared to PRV, almost 95% of those failed MuJava.

Like AOIS, some scenarios were discovered for the class-level operators as well under which these operators always create equivalent mutants. These scenarios can be used to modify the concerned operators such that they do not create equivalent mutants. This will reduce the computational cost in creating and executing the surplus mutants.

7.2.2 Scenarios

The majority of the class-level operators created equivalent mutants. Some scenarios have been sketched out for some of these operators. These scenarios depict situations which will always create equivalent mutants.

7.2.2.1 IHD

The operator IHD works by deleting a hiding variable in a subclass. One situation has been identified when IHD always creates an equivalent mutant.

Original Program	Equivalent Mutants

Public class LinkedList {
protected ListNode start;
protected int size;
public LinkedList() { …… }
public insert(int var) { …… }
……
}
 	
public class Stack extends LinkedList {
protected ListNode start;
protected int size;
public Stack(){ …… }
public insert(int var) {
start = newNode;
size++;
}
……
}
	Public class LinkedList {
	protected ListNode start;
	protected int size;
	public LinkedList() { …… }
	public insert(int var) { …… }
	……
	}
	
	public class Stack extends LinkedList {
	// protected ListNode start;
	protected int size;
	public Stack(){ …… }
	public insert(int var) {
	start = newNode;
	size++;
	}
	……
	}

Figure 28: Scenario in which IHD creates equivalent mutants

Whenever the initialization of a variable is deleted in a subclass, the subclass makes use of the initialization in the parent class. If this initialization is the same in both the parent and the child class, the mutant is equivalent.

7.2.2.2 IHI

The operator IHI works by inserting a hiding variable in a subclass. One situation has been identified when IHI always creates an equivalent mutant.

Original Program	Equivalent Mutants

Public class LinkedList {
protected ListNode start;
protected int size;
public LinkedList() { …… }
public insert(int var) {
start = newNode;
size++;
……
}
……
}
 	
public class Stack extends LinkedList {
protected ListNode start;
public Stack(){ …… }
public insert(int var) {
start = newNode;
……
}
……
}
	Public class LinkedList {
	protected ListNode start;
	protected int size;
	public LinkedList() { …… }
	public insert(int var) {
	start = newNode;
	size++;
	……
	}
	……
	}
	
	public class Stack extends LinkedList {
	protected ListNode start;
	protected int size;
	public Stack(){ …… }
	public insert(int var) {
	start = newNode;
	……
	}
	……
	}

Figure 29: Scenario in which IHI creates equivalent mutants

IHI always creates an equivalent mutant when the initialization of a variable is inserted in a subclass and the subclass is not using this variable in any of its functions.

7.2.2.3 IOR

IOR always creates equivalent mutants when the methods which have been renamed are not called elsewhere in the program. Hence, any change in the name of the function, will not affect the functioning of the rest of the programs. However, it is easy to note that if this same function is called in the main program, it will cause an error and therefore, fail the mutant.

7.2.2.4 PRV

PRV created a large number of equivalent mutants. The situation which caused this is explained more clearly in the example below:

Original program	Equivalent Mutant

`public class Stack extends LinkedList {`
`  protected ListNode start;`
`  public Stack(){ …… }`
`  public insert(int var) {`
`  start = newNode;`
`  end = newNode;`
`  }`
`  ……`
`}`
	`public class Stack extends LinkedList {`
	`  protected ListNode start;`
	`  public Stack(){ …… }`
	`  public insert(int var) {`
	`  start = newNode;`
	`  end = start;`
	`  }`
	`  ……`
	`}`

Figure 30: Scenario in which PRV creates equivalent mutants

Whenever an object reference is referred to other compatible types, sometimes the value of the new object is the same as the value of the object reference in the source program. In the example, the values of objects *start* and *end* are the same. Hence, making the object *end* to point to the object *start* will produce an equivalent mutant.

7.2.2.5 JID

This operator deletes the initialization of instance variables. JID always creates an equivalent mutant when the initialization of a variable is set to the default value of the particular variable. For example:

Original program	Equivalent mutant

`public class Stack extends LinkedList {`
`  int size = 0;`
`  ……`
`}`
	`public class Stack extends LinkedList {`
	`  int size;`
	`  ……`
	`}`

Figure 31: Scenario in which JID creates equivalent mutants

Original program	Non-equivalent mutant

`public class Stack extends LinkedList {`
`  int size = 10;`
`  ……`
`}`
	`public class Stack extends LinkedList {`
	`  int size;`
	`  ……`
	`}`

Figure 32: Scenario in which JID will never create an equivalent mutant

7.2.2.6 JDC

This operator deletes the user-defined default constructor and makes use of Java's default constructor. JDC will create an equivalent mutant whenever the operations performed in the default constructor are simple initializations to the default values of the variables. If the variables or objects are initialized to values other than their default values, no equivalent mutants will be created. Below is an example explaining this scenario:

Original program	Equivalent mutant

public class LinkedList {
protected ListNode start;
protected ListNode end;
protected int size;
public LinkedList(){
start = null; end = null; size = 0;
}
……
}
	public class LinkedList {
	protected ListNode start;
	protected ListNode end;
	protected int size;
	/* public LinkedList(){
	start = null; end = null; size = 0;
	}*/
	……
	}

Figure 33: Scenario in which JDC creates equivalent mutant

Original program	Non-equivalent mutant

public class LinkedList {
protected ListNode start;
protected ListNode end;
protected int size;
public LinkedList(){
start = null; end = null; size = 20;
}
……
}
	public class LinkedList {
	protected ListNode start;
	protected ListNode end;
	protected int size;
	/* public LinkedList(){
	start = null; end = null; size = 20;
	}*/
	……
	}

Figure 34: Scenario in which JDC will never create an equivalent mutant

In summary, there are always some situations which create equivalent mutants. The study of mutation operators has helped identify those situations. Also, sometimes, some operators also make certain changes to the original program in such a way that they cause MuJava to fail.

8. CONCLUSION AND FUTURE WORK

Detailed work has been carried out to determine which mutation operators contribute more towards creating equivalent mutants. The following results were obtained:

- Method-level operators:
  - The majority of mutants were created for AOIS, ROR and LOI.
  - Almost half of the mutants created for AOIS were found to be equivalent.
  - However, majority of the mutants created for ROR and LOI were killed successfully.
  - All the mutants created for COI failed the software.
- Class-level operators:
  - Majority of mutants were created for PRV, JSI, IHI and IOD.
  - Almost 50% of the mutants created for PRV, failed the software. Of the remaining mutants, almost 80% were killed and the remaining were found to be equivalent.
  - Nearly 95% of the mutants created for PMD caused MuJava to fail.
  - All the mutants created for JDC were found to be equivalent.
  - Many class-mutation operators such as IOR, ISI, etc, which created a very low number of mutants, were found to be equivalent.

Various scenarios have been sketched out for method-level operator AOIS and class-level operators IHD, IHI, IOR, PRV, JID and JDC. These scenarios indicate situations which always create equivalent mutants.

Looking at these results, different conclusions can be made for method-level mutation operators and class-level mutation operators. For method-level operators, AOIS is more probable to create equivalent mutants. For class-level operators, PRV usually creates the greatest number of mutants. But most of these mutants are simply program failures. Also, operators that create a very low number of mutants as compared to other class-level operators are also most likely to create equivalent mutants.

For the future work of this project, the operators that create a larger number of equivalent mutants can be tweaked such that they do not create equivalent mutants. This can be done by using the scenarios sketched out in chapter 7. These scenarios can be incorporated in the algorithms implemented for the concerned operators. Thus, the creation of equivalent mutants can be avoided.

APPENDIX A – USER GUIDE for MuJava

MuJava requires 2 jar files and one config file. These files can be found at:
http://ise.gmu.edu/~ofut/mujava/

These 2 files, (mujava.jar & adaptedOJ.jar) should be placed in the same directory. Lets assume the directory is C:\MuJava\.
Before executing the software, the following steps should be taken:

1. Your computer must have a JRE installed and the Java path set. To learn how to do this refer to the java.sun.com website for details.
2. After this, firsts et the classpath using the following command:

```
set CLASSPATH=%CLASSPATH%;C:\mujava\mujava.jar;C:\mujava\adaptedOJ.jar;C:\j2sdk1.4.0_01\lib\tools.jar;C:\mujava\classes
```

This command allows you to run MuJava to create mutants. To run test cases a different path must be set which is explained later.

3. After this open the mujava.config file. You should see 'MuJava_HOME=C:\ofut\mujava'. Change this to point to 'MuJava_HOME=C:\MuJava_Home'. This command tells MuJava where to find the source files and the test programs when executing MuJava.
4. Now we need to make a directory structure for MuJava. All the source programs and test programs will be contained in this directory structure. This structure can be made either manually or by using the command:

   ```
   java mujava.makeMuJavaStructure
   ```

   The directory structure will now look like:
   C:\MuJava_Home\classes
   C:\MuJava_Home\testset
   C:\MuJava_Home\src
   C:\MuJava_Home\result

Now, you will have 2 different directories; one will be 'C:\MuJava' and the other will have 'C:\MuJava_Home\'. The config file should be in the directory of MuJava.

Creating Mutants

1. Create your source file (say 'AccessorModifier.java') that you want to test in C:\MuJava_Home\src.
2. Compile this file using the DOS prompt and the command 'javac AccessorModifier.java'. This command will create a class file in the folder C:\MuJava_Home\src\.
3. Remove this file from the C:\MuJava_Home\src\ and copy it to the folder C:\MuJava_Home\classes\.
4. Now change the path in the dos prompt to point to C:\MuJava\ instead of C:\MuJava_Home\src\.
5. Now set the classpath using:

   ```
   set classpath=%classpath%;c:\MuJava\mujava.jar;c:\MuJava\adaptedOj.jar;c:\J2ee\jdk\lib\tools.jar;c:\MuJava_Home\classes;
   ```

6. Then use the following command to start MuJava:

   ```
   java mujava.gui.GenMutantsMain
   ```

7. Now select the programs from the list shown for which you want to create the mutants. Also select the mutation operators for which you want to create mutants. Then press 'Generate'. Whilst mutants are being generated, many messages will be sent to the command prompt in the background. The 'Generate' button will be disabled. When all the mutants have been generated, this button will turn yellow again and the different mutants that have been generated can then be viewed.

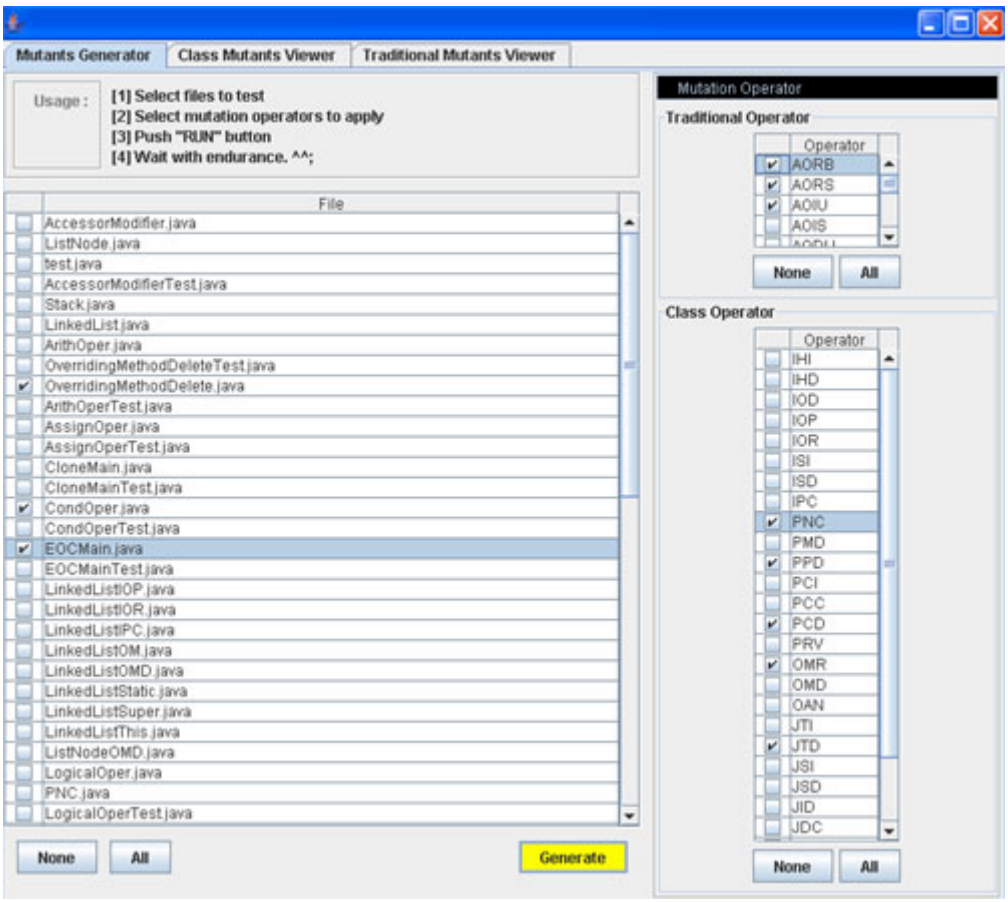

Figure 35: Creating Mutants using MuJava

The mutants that are created for each type of operator can be viewed by navigating through the tabs of ‘Class Mutants Viewer’ or ‘Traditional Mutants Viewer’.

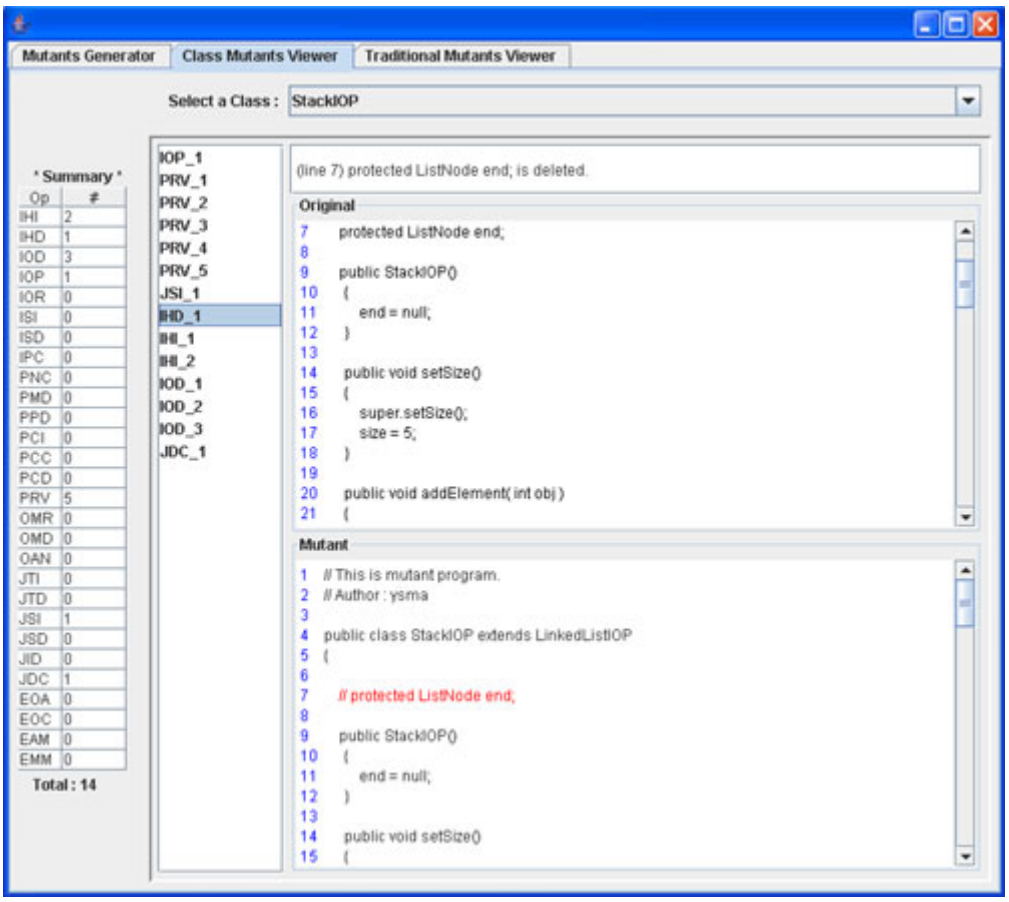

Figure 36: Class Mutants Viewer in MuJava

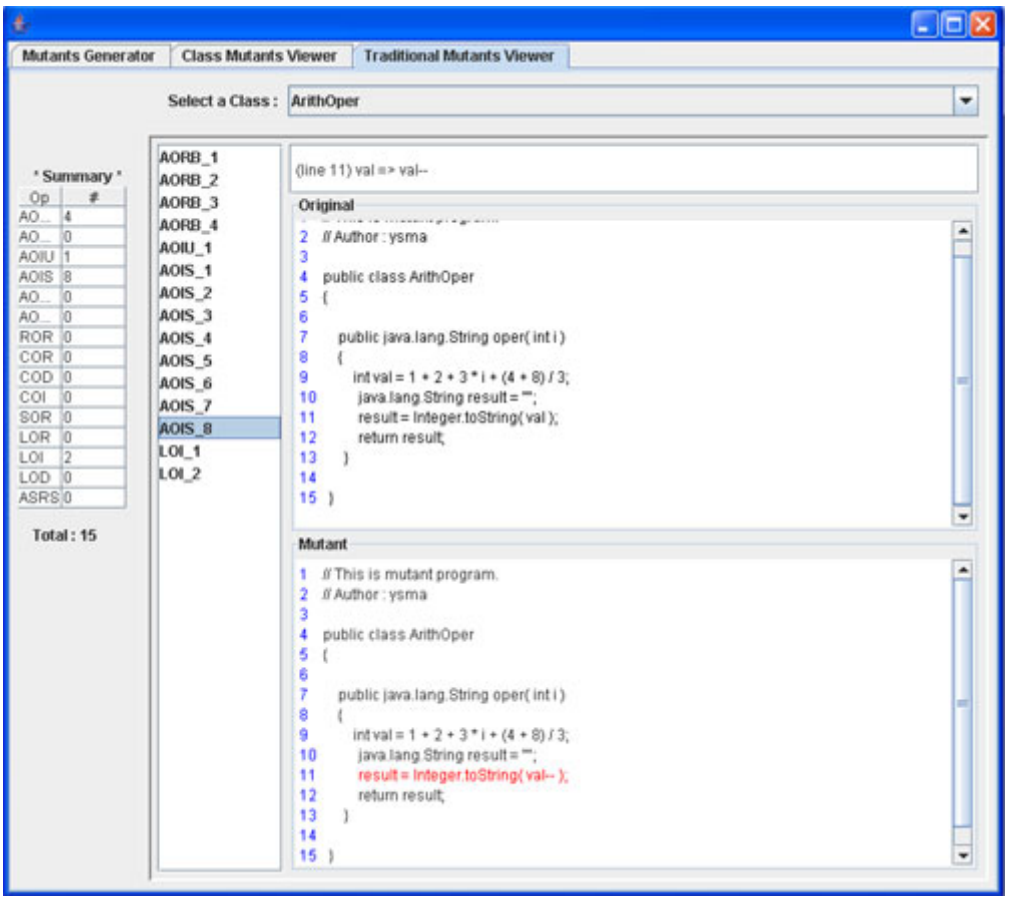

Figure 37: Traditional Mutants Viewer in MuJava

All the mutants that are created can also be found in the directory C:\MuJava_Home\result\.

Creating Test Cases to kill Mutants

Next we need to create test cases to kill the mutants that have been created. MuJava requires these test programs to be written in a specific format. Lets assume we have a program 'AccessorModifier.java' such as:

```
public class AccessorModifier {

	public String AccessorSet(int x1, int y1) {

		String result = "";
		Points p1 = new Points();

		p1.setX(x1);
		p1.setY(y1);

		result = result.concat(Integer.toString(p1.getX()));
		result = result.concat(Integer.toString(p1.getY()));
		return result;
	}
}
```

The important thing to note is that the original program should not have any main function. It is not necessary that the function in the main program should return a string value. But we need at least one function in the original program that returns a string value. This function should represent the data variables used in the original program such that it simply converts/ returns the string value of all the variables we are interested in testing. This is compulsory because MuJava analyses the string values return to determine if the mutant has been killed or not.
The test case for this program should look like this:

```
public class AccessorModifierTest
{
	public String test1()
	{
		String result = "";
		AccessorModifier obj = new AccessorModifier();
		result = obj.AccessorSet(1, 1);
		return result;
	}

	public String test2()
	{
		String result = "";
		AccessorModifier obj = new AccessorModifier();
		result = obj.AccessorSet(1, 2);
		return result;
	}
	public String test3()
	{
		String result = "";
		AccessorModifier obj = new AccessorModifier();
		result = obj.AccessorSet(-1, -1);
		return result;
	}
}
```

For the program that contains the test cases, each test case must be written as a separate function. The name of this function must begin with 'test'. Also, each test case must return a string value as result. The value of this string variable (in our case 'result'), is computed by calling the methods in the main program that return string values as well. You can concatenate the values

returned by multiple functions using the operator '+'. The access modifier for each of the test methods and the test program should be public.
After writing the test program, save it in the folder C:\MuJava_Home\src\. Compile this file using the DOS prompt and the command 'javac AccessorModifierTest.java'. This command will create a class file in the folder C:\MuJava_Home\src\. There will be more than one class file created for this program (in our case 'AccessorModifierTest.class, AccessorModifier.class'). Remove these files and copy them to the folder C:\MuJava_Home\testset\.

Killing Mutants

1. Open a new DOS prompt window. Change the path to C:\MuJava\ to kill the mutants.
2. The catch when killing the mutants is that the classpath needs to be changed such that it does not point to MuJava classes anymore. The classpath therefore becomes:

   set classpath=%classpath%;c:\MuJava\mujava.jar;c:\MuJava\adaptedOj.jar;c:\J2ee\jdk\lib\tools.jar;

3. Then use the following command to start MuJava again to kill the mutants:

   java mujava.gui.RunTestMain

4. Now select the original program from the dropdown list for which mutants have been created already. From the second dropdown list select the corresponding test program. Also, select if you want to apply to the traditional mutants only, or to the class mutants only or to both. Then press 'Execute'. Whilst test cases are run against the original program, many messages will be sent to the command prompt in the background. The 'Execute' button will be disabled. When test cases have finished completion, this button will turn yellow again. Statistics will be shown indicating the number of mutants killed, the mutants left alive and the mutation score.

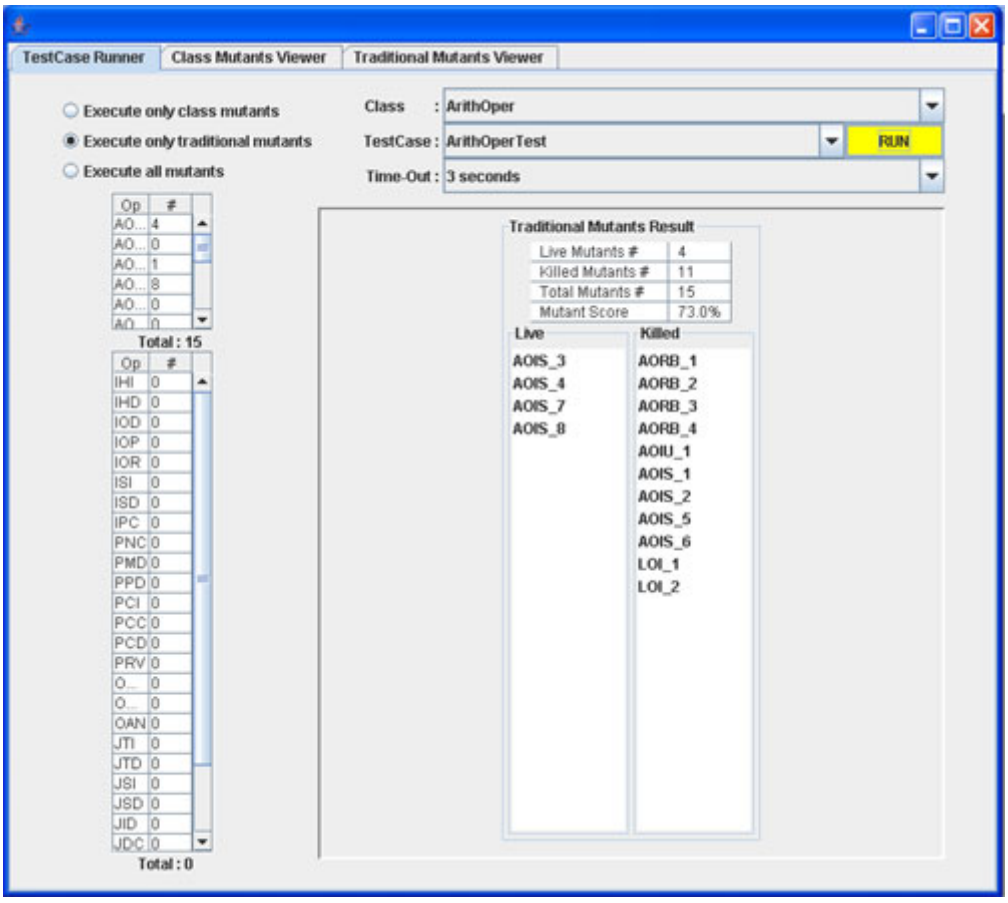

Figure 38: Killing Mutants using MuJava

APPENDIX B – Programs written for MuJava (source code)

Method-level Mutation Operators

The programs written in this section were developed to create mutants for method-level operators in MuJava. They were written arbitrarly. Test cases written for all the programs are also included.

ArithOper.java

```
/* @author: Maryam Umar */
public class ArithOper {
	public String oper(int i){
		int val = 1 + 2 + 3*i + (4 + 8)/3;
		String result = "";
		result = Integer.toString(val);
		return result;
	}
}
```

ArithOperTest.java

```
/* @author: Maryam Umar */
public class ArithOperTest{
	public String test1(){
		String result = "";
		ArithOper obj = new ArithOper();
		result = result + obj.oper(0);
		return result;
	}
	public String test2(){
		String result = "";
		ArithOper obj = new ArithOper();
		result = result + obj.oper(1);
		return result;
	}
	public String test3(){
		String result = "";
		ArithOper obj = new ArithOper();
		result = result + obj.oper(2);
		return result;
	}
	public String test4(){
		String result = "";
		ArithOper obj = new ArithOper();
		result = result + obj.oper(3);
		return result;
	}
}
```

AssignOper.java

```
/* @author: Maryam Umar */
public class AssignOper {
	public String operation(int i){
		String result = "";
		i += 10;
		result = Integer.toString(i);
		return result;
	}
}
```

AssignOperTest.java

```
/* @author: Maryam Umar */
public class AssignOperTest{
	public String test1(){
		String result = "";
		AssignOper obj = new AssignOper();
		result = result + obj.operation(0);
		return result;
	}
	public String test2(){
		String result = "";
		AssignOper obj = new AssignOper();
		result = result + obj.operation(1);
```

```
		return result;
	}
	public String test3(){
		String result = "";
		AssignOper obj = new AssignOper();
		result = result + obj.operation(10);
		return result;
	}
}
```

CondOper.java

```
/* @author: Maryam Umar */
public class CondOper {
	public String LogicOper(int a, int b, int c){
		String result = "";
		if(a > b && b > c)
			result = Integer.toString(a);
		else if(b > a && b > c)
			result = Integer.toString(b);
		else
			result = Integer.toString(c);
		return result;
	}
}
```

CondOperTest.java

```
/* @author: Maryam Umar */
public class CondOperTest{
	public String test1(){
		String result = "";
		CondOper obj = new CondOper();
		result = result + obj.LogicOper(0, 1, 2);
		return result;
	}
	public String test2(){
		String result = "";
		CondOper obj = new CondOper();
		result = result + obj.LogicOper(0, 2, 1);
		return result;
	}
	public String test3(){
		String result = "";
		CondOper obj = new CondOper();
		result = result + obj.LogicOper(1, 0, 2);
		return result;
	}
	public String test4(){
		String result = "";
		CondOper obj = new CondOper();
		result = result + obj.LogicOper(1, 2, 0);
		return result;
	}
	public String test5(){
		String result = "";
		CondOper obj = new CondOper();
		result = result + obj.LogicOper(2, 0, 1);
		return result;
	}
	public String test6(){
		String result = "";
		CondOper obj = new CondOper();
		result = result + obj.LogicOper(2, 1, 0);
		return result;
	}
}
```

LogicalOper.java

```
/* @author: Maryam Umar */
public class LogicalOper {
	public String operation(int i, int j){
		String result = "";
		int val;
```

```
            val = i & j;
            result = Integer.toString(val);
            return result;
      }
}
```

LogicalOperTest.java

```
/* @author: Maryam Umar */
public class LogicalOperTest{
      public String test1(){
            String result = "";
            LogicalOper obj = new LogicalOper();
            result = result + obj.operation(10, 10);
            return result;
      }
      public String test2(){
            String result = "";
            LogicalOper obj = new LogicalOper();
            result = result + obj.operation(10, 20);
            return result;
      }
      public String test3(){
            String result = "";
            LogicalOper obj = new LogicalOper();
            result = result + obj.operation(1, 1);
            return result;
      }
}
```

RelOper.java

```
/* @author: Maryam Umar */
public class RelOper {
      public String Condition(int a, int b, int c){
            String result = "";
            if(a > b){
                  if(b > c)
                        result = Integer.toString(a);
            }
            else if(b > a){
                  if( b > c)
                        result = Integer.toString(b);
            }
            else
                  result = Integer.toString(c);
            return result;
      }
}
```

RelOperTest.java

```
/* @author: Maryam Umar */
public class RelOperTest{
      public String test1(){
            String result = "";
            RelOper obj = new RelOper();
            result = result + obj.Condition(0, 1, 2);
            return result;
      }
      public String test2(){
            String result = "";
            RelOper obj = new RelOper();
            result = result + obj.Condition(0, 2, 1);
            return result;
      }
      public String test3(){
            String result = "";
            RelOper obj = new RelOper();
            result = result + obj.Condition(1, 0, 2);
            return result;
      }
      public String test4(){
            String result = "";
```

```
		RelOper obj = new RelOper();
		result = result + obj.Condition(2, 1, 0);
		return result;
	}
	public String test5(){
		String result = "";
		RelOper obj = new RelOper();
		result = result + obj.Condition(1, 1, 2);
		return result;
	}
	public String test6(){
		String result = "";
		RelOper obj = new RelOper();
		result = result + obj.Condition(7, 5, 5);
		return result;
	}
	public String test7(){
		String result = "";
		RelOper obj = new RelOper();
		result = result + obj.Condition(5, 7, 7);
		return result;
	}
}
```

ShiftOper.java

```
/* @author: Maryam Umar */
public class ShiftOper {
	public String operation(int i){
		String result = "";
		int val;
		val = i << 2;
		result = Integer.toString(val);
		return result;
	}
}
```

ShiftOperTest.java

```
/* @author: Maryam Umar */
public class ShiftOperTest{
	public String test1(){
		String result = "";
		ShiftOper obj = new ShiftOper();
		result = result + obj.operation(20);
		return result;
	}
	public String test2(){
		String result = "";
		ShiftOper obj = new ShiftOper();
		result = result + obj.operation(10);
		return result;
	}
}
```

Class-level Mutation Operators

The programs written in this section were developed to create mutants for class-level operators in MuJava. They were written arbitrarly keeping in mind that they included the features required to create mutants for the specific mutation operators. Test cases written for all the programs are also included.

Points.java

```
/* @author: Maryam Umar */
public class Points {
	int x;	int y;
	public Points(){
		x = 0;
		y = 0;
	}
	public int getX() {
		return x;
	}
```

```
	public int getY() {
		return y;
	}
	public void setX(int x) {
		this.x = x;
	}
	public void setY(int y) {
		this.y = y;
	}
}
```

AccessorModifier.java

```
/* @author: Maryam Umar */
public class AccessorModifier {
	public String AccessorSet(int x1, int y1) {
		String result = "";
		Points p1 = new Points();
		p1.setX(x1);
		p1.setY(y1);
		result = result.concat(Integer.toString(p1.getX()));
		result = result.concat(Integer.toString(p1.getY()));
		return result;
	}
}
```

AccessorModifierTest.java

```
/* @author: Maryam Umar */
public class AccessorModifierTest {
	public String test1() {
		String result = "";
		AccessorModifier obj = new AccessorModifier();
		result = obj.AccessorSet(1, 1);
		return result;
	}
	public String test2() {
		String result = "";
			AccessorModifier obj = new AccessorModifier();
		result = obj.AccessorSet(1, 2);
		return result;
	}
}
```

EOCMain.java

```
/* @author: Maryam Umar */
public class EOCMain {
	public String EOCMain(int i, int j) {
		Integer x = new Integer(i);
		Integer y = new Integer(j);
		if(x == y)
			return "0";
		else
			return "1";
		}
}
```

EOCMainTest.java

```
/* @author: Maryam Umar */
public class EOCMainTest {
	public String test1() {
		String result = "";
		EOCMain obj = new EOCMain();
		result = result + obj.EOCMain(-1, -1);
		return result;
	}
}
```

ListNode.java

```
/* @author: Maryam Umar */
public class ListNode {
	protected int data = 0;
	protected ListNode nextPtr = null;
```

```
	public ListNode(int obj){
		if(obj == 0)
			throw new NullPointerException();
		data = obj;
	}
}
```

LinkedList.java

```
/* @author: Maryam Umar */
public class LinkedList {
	protected ListNode start;
	protected ListNode end;
	protected int size;
	public LinkedList(){
		start = null;
		end = null;
		size = 0;
	}
	public void addElement(int obj){
		if (obj == 0)
			throw new NullPointerException();
		ListNode newNode = new ListNode(obj);
		size++;
		if(start == null){
			start = newNode;
			end = newNode;
		}
		else{
			end.nextPtr = newNode;
			end = newNode;
		}
	}
	public ListNode find(int obj){
		if(obj == 0)
			return null;
		ListNode temp = start;
		while(temp != null){
			if(temp.data == obj)
				return temp;
			temp = temp.nextPtr;
		}
		return null;
	}
	public ListNode findBefore(int obj){
		if(obj == 0)
			return null;
		ListNode prevTemp = null;
		ListNode temp = start;
		while(temp != null){
			if(temp.data == obj)
				return prevTemp;
			prevTemp = temp;
			temp = temp.nextPtr;
		}
		return null;
	}
	public void addElementAfter(int obj, int pos){
		if(obj == 0 || pos == 0)
			throw new NullPointerException();
		ListNode posNode = find(pos);
		if(posNode == null)
			throw new NullPointerException();
		ListNode newNode = new ListNode(obj);
		size++;
		if(posNode == end){
			posNode.nextPtr = newNode;
			end = newNode;
		}
		else{
			newNode.nextPtr = posNode.nextPtr;
			posNode.nextPtr = newNode;
		}
```

```
	}
	public int removeNode(int obj){
		if(obj == 0)
			throw new NullPointerException();
		ListNode delNode = find(obj);
		if(delNode == null)
			return -1;
		ListNode prev = findBefore(obj);
		if(delNode == start){
			start = delNode.nextPtr;
			delNode = null;
		}
		else if	(delNode == end){
			end = prev;
			delNode = null;
		}
		else{
			prev.nextPtr = delNode.nextPtr;
			delNode = null;
		}
		size--;
		return 0;
	}
	public String ConvertToString(){
		ListNode tmp = start;
		String result = "";
		while(tmp != end){
			result += Integer.toString(tmp.data) + "";
			tmp = tmp.nextPtr;
		}
		result += Integer.toString(end.data);
		return result;
	}
}
```

Stack.java

```
/* @author: Maryam Umar */
public class Stack extends LinkedList {
	protected int size;	protected ListNode end;
	public Stack(){
		size = 0;
		end = null;
	}
	public void addElement(int obj){
		if (obj == 0)
			throw new NullPointerException();
		ListNode newNode = new ListNode(obj);
		if(size == 0){
			start = newNode;
			end = start;
			size++;
		}
		else{
			end.nextPtr = newNode;
			end = newNode;
			size++;
		}
	}
	public ListNode findBefore(){
		ListNode temp = start;
		int ctr = 1;
		while(ctr != (size-1)){
			temp = temp.nextPtr;
			ctr++;
		}
		return temp;
	}
	public int removeNode(){
		ListNode prev = findBefore();
		int value;
		if(size == 0)
```

```
                return 0;
            else if(size == 1){
                value = end.data;
                start = null;
                end = null;
                size--;
                return value;
            }
            else{
                value = end.data;
                end = prev;
                size--;
                return value;
            }
        }
        public String ConvertToString(){
            ListNode tmp = start;
            String result = "";
            while(tmp != end){
                result += Integer.toString(tmp.data) + "";
                tmp = tmp.nextPtr;
            }
            result += Integer.toString(end.data);
            return result;
            }
}
```

CloneMain.java

```
/* @author: Maryam Umar */
public class CloneMain {
        public String Clonemain() {
            String result = "";
            Stack st1, st2;
            st1 = new Stack();
            st2 = st1;
            st1.addElement(1);
            st1.addElement(2);
            st1.addElement(3);
            st1.addElement(4);
            result = result + st1.removeNode();
            return result;
        }
}
```

CloneMainTest.java

```
/* @author: Maryam Umar */
public class CloneMainTest {
        public String test1()  {
            String result = "";
            LinkedList st = new LinkedList();
            st.addElement(1);
            st.addElement(2);
            st.addElement(3);
            st.addElement(4);
            result = result + st.removeNode(4);
            st.addElement(4);
            result = result + st.ConvertToString();
            return result;
        }
        public String test2() {
            String result = "";
            LinkedList st = new LinkedList();
            st.addElement(1);
            st.addElementAfter(3, 1);
            st.addElement(2);
            result = result + st.ConvertToString();
            return result;
        }
        public String test3() {
            String result = "";
            LinkedList st = new LinkedList();
```

```
		st.addElement(1);
		st.addElement(2);
		st.addElementAfter(3, 1);
		result = result + st.ConvertToString();
		return result;
	}
	public String test4() {
		String result = "";
		LinkedList l1 = new LinkedList();
		LinkedList l2 = new LinkedList();
		l1.addElement(1);
		l2.addElement(4);
		l1.addElement(2);
		l2.addElement(5);
		l2.addElement(6);
		l1.addElement(3);
		result = result + l2.ConvertToString();
		result = result + l1.ConvertToString();
		return result;
	 }
	public String test5() {
		String result = "";
		Stack st = new Stack();
		st.addElement(1);
		st.addElementAfter(2, 1);
		result = result + st.removeNode();
		return result;
	 }
	public String test6()    {
		String result = "";
		Stack st = new Stack();
		st.addElement(1);
		st.addElement(2);
		result = result + st.ConvertToString();
		return result;
	}
	public String test7() {
		String result = "";
		Stack st = new Stack();
		st.addElement(1);
		st.addElement(2);
		st.addElement(3);
		st.addElement(4);
		result = result + st.removeNode(4);
		st.addElement(4);
		result = result + st.removeNode(4);
		return result;
	}
	public String test8() {
		String result = "";
		Stack st = new Stack();
		Stack st2 = new Stack();
		st.addElement(1);
		st.addElement(2);
		st.addElement(3);
		st2.addElement(4);
		st2.addElement(5);
		st2.addElement(6);
		st.removeNode();
		st.removeNode();
		result = result + st.ConvertToString();
		result = result + st2.ConvertToString();
		return result;
	}
	public String test9() {
		String result = "";
		CloneMain st = new CloneMain();
		result = result + st.Clonemain();
		return result;
	}
}
```

PNC.java

```
/* @author: Maryam Umar */
public class PNC {
	public String PNCmain() {
		String result = "";
		LinkedList ll;
		ll = new LinkedList();
		ll.addElement(1);
		ll.addElement(2);
		ll.addElement(3);
		result = result + ll.removeNode(3) + ll.ConvertToString();
		return result;
	}
}
```

PNCTest.java

```
/* @author: Maryam Umar */
public class PNCTest {
	public String test1()  {
		String result = "";
		LinkedList st = new LinkedList();
		st.addElement(1);
		st.addElement(2);
		st.addElement(3);
		st.addElement(4);
		result = result + st.removeNode(4);
		st.addElement(4);
		result = result + st.ConvertToString();
		return result;
	}
	public String test2() {
		String result = "";
		LinkedList st = new LinkedList();
		st.addElement(1);
		st.addElementAfter(3, 1);
		st.addElement(2);
		result = result + st.ConvertToString();
		return result;
	}
	public String test3() {
		String result = "";
		LinkedList st = new LinkedList();
		st.addElement(1);
		st.addElement(2);
		st.addElementAfter(3, 1);
		result = result + st.ConvertToString();
		return result;
	 }
	public String test4() {
		String result = "";
		LinkedList l1 = new LinkedList();
		LinkedList l2 = new LinkedList();
		l1.addElement(1);
		l2.addElement(4);
		l1.addElement(2);
		l2.addElement(5);
		l2.addElement(6);
		l1.addElement(3);
		result = result + l2.ConvertToString();
		result = result + l1.ConvertToString();
		return result;
	 }
	public String test5()  {
		String result = "";
		Stack st = new Stack();
		st.addElement(1);
		st.addElementAfter(2, 1);
		result = result + st.removeNode();
		return result;
	}
	public String test6() {
		String result = "";
```

```
		Stack st = new Stack();
		st.addElement(1);
		st.addElement(2);
		result = result + st.ConvertToString();
		return result;
	}
	public String test7() {
		String result = "";
		Stack st = new Stack();
		st.addElement(1);
		st.addElement(2);
		st.addElement(3);
		st.addElement(4);
		result = result + st.removeNode(4);
		st.addElement(4);
		result = result + st.removeNode(4);
		return result;
	}
	public String test8() {
		String result = "";
		Stack st = new Stack();
		Stack st2 = new Stack();
		st.addElement(1);
		st.addElement(2);
		st.addElement(3);
		st2.addElement(4);
		st2.addElement(5);
		st2.addElement(6);
		st.removeNode();
		st.removeNode();
		result = result + st.ConvertToString();
		result = result + st2.ConvertToString();
		return result;
	}
	public String test9() {
		String result = "";
		PNC p = new PNC();
		result = result + p.PNCmain();
		return result;
	}
}
```

ParameterVariablePPD.java

```
/* @author: Maryam Umar */
public class ParameterVariablePPD {
	public static void ParameterVariable(Stack s){
		s.addElement(1);
		s.addElement(2);
		s.addElement(3);
		s.addElement(4);
	}
	public String ParameterMain() {
		String result = "";
		Stack st = new Stack();
		ParameterVariable(st);
		result = result + st.removeNode() + st.ConvertToString();
		return result;
	}
}
```

ParameterVariableTest.java

```
/* @author: Maryam Umar */
public class ParameterVariableTest {
	public String test1() {
		String result = "";
		LinkedList st = new LinkedList();
		st.addElement(1);
		st.addElement(2);
		st.addElement(3);
		st.addElement(4);
		result = result + st.removeNode(4);
		st.addElement(4);
```

```
        result = result + st.ConvertToString();
        return result;
    }
    public String test2() {
        String result = "";
        LinkedList st = new LinkedList();
        st.addElement(1);
        st.addElementAfter(3, 1);
        st.addElement(2);
        result = result + st.ConvertToString();
        return result;
    }
    public String test3() {
        String result = "";
        LinkedList st = new LinkedList();
        st.addElement(1);
        st.addElement(2);
        st.addElementAfter(3, 1);
        result = result + st.ConvertToString();
        return result;
     }
    public String test4() {
        String result = "";
        LinkedList l1 = new LinkedList();
        LinkedList l2 = new LinkedList();
        l1.addElement(1);
        l2.addElement(4);
        l1.addElement(2);
        l2.addElement(5);
        l2.addElement(6);
        l1.addElement(3);
        result = result + l2.ConvertToString();
        result = result + l1.ConvertToString();
        return result;
     }
    public String test5() {
        String result = "";
        Stack st = new Stack();
        st.addElement(1);
        st.addElementAfter(2, 1);
        result = result + st.removeNode();
        return result;
    }
    public String test6() {
        String result = "";
        Stack st = new Stack();
        st.addElement(1);
        st.addElement(2);
        result = result + st.ConvertToString();
        return result;
    }
    public String test7() {
        String result = "";
        Stack st = new Stack();
        st.addElement(1);
        st.addElement(2);
        st.addElement(3);
        st.addElement(4);
        result = result + st.removeNode(4);
        st.addElement(4);
        result = result + st.removeNode(4);
        return result;
    }
    public String test8() {
        String result = "";
        Stack st = new Stack();
        Stack st2 = new Stack();
        st.addElement(1);
        st.addElement(2);
        st.addElement(3);
        st2.addElement(4);
        st2.addElement(5);
```

```
		st2.addElement(6);
		st.removeNode();
		st.removeNode();
		result = result + st.ConvertToString();
		result = result + st2.ConvertToString();
		return result;
	}
	public String test9() {
		String result = "";
		ParameterVariablePPD st = new ParameterVariablePPD();
		result = result + st.ParameterMain();
		return result;
	}
}
```

TypeCastOperatorTCO.java

```
/* @author: Maryam Umar */
public class TypeCastOperatorTCO {
	public String TypeMain() {
		String result = "";
		Stack st = new Stack();
		LinkedList ll = st;
			((Stack)ll).addElement(1);
		((Stack)ll).addElement(2);
		((Stack)ll).addElement(3);
			((Stack)ll).addElement(4);
		((Stack)ll).addElement(5);
		result = result + st.removeNode() + st.ConvertToString();
		return result;
	}
}
```

TypeCastOperatorTest.java

```
/* @author: Maryam Umar */
public class TypeCastOperatorTest {
	public String test1() {
		String result = "";
		LinkedList st = new LinkedList();
		st.addElement(1);
		st.addElement(2);
		st.addElement(3);
		st.addElement(4);
		result = result + st.removeNode(4);
		st.addElement(4);
		result = result + st.ConvertToString();
		return result;
	}
	public String test2() {
		String result = "";
		LinkedList st = new LinkedList();
		st.addElement(1);
		st.addElementAfter(3, 1);
		st.addElement(2);
		result = result + st.ConvertToString();
		return result;
	}
	public String test3() {
		String result = "";
		LinkedList st = new LinkedList();
		st.addElement(1);
		st.addElement(2);
		st.addElementAfter(3, 1);
		result = result + st.ConvertToString();
		return result;
	 }
	public String test4() {
		String result = "";
		LinkedList l1 = new LinkedList();
		LinkedList l2 = new LinkedList();
		l1.addElement(1);
		l2.addElement(4);
		l1.addElement(2);
```

```
        l2.addElement(5);
        l2.addElement(6);
        l1.addElement(3);
        result = result + l2.ConvertToString();
        result = result + l1.ConvertToString();
        return result;
     }
    public String test5() {
        String result = "";
        Stack st = new Stack();
        st.addElement(1);
        st.addElementAfter(2, 1);
        result = result + st.removeNode();
        return result;
    }
    public String test6() {
        String result = "";
        Stack st = new Stack();
        st.addElement(1);
        st.addElement(2);
        result = result + st.ConvertToString();
        return result;
    }
    public String test7() {
        String result = "";
        Stack st = new Stack();
        st.addElement(1);
        st.addElement(2);
        st.addElement(3);
        st.addElement(4);
        result = result + st.removeNode(4);
        st.addElement(4);
        result = result + st.removeNode(4);
        return result;
    }
    public String test8() {
        String result = "";
        Stack st = new Stack();
        Stack st2 = new Stack();
        st.addElement(1);
        st.addElement(2);
        st.addElement(3);
        st2.addElement(4);
        st2.addElement(5);
        st2.addElement(6);
        st.removeNode();
        st.removeNode();
        result = result + st.ConvertToString();
        result = result + st2.ConvertToString();
        return result;
    }
    public String test9() {
        String result = "";
        TypeCastOperatorTCO TCO = new TypeCastOperatorTCO();
        result = result + TCO.TypeMain();
        return result;
    }
    public String test10() {
        String result = "";
        Stack st = new Stack();
        st.addElement(1);
        st.addElement(2);
        st.addElement(3);
        st.removeNode(2);
        result = result + st.ConvertToString();
        return result;
    }
    public String test11() {
        String result = "";
        LinkedList st = new LinkedList();
        st.addElement(1);
        st.addElement(2);
```

```
		st.addElement(3);
		st.removeNode(1);
		result = result + st.ConvertToString();
		return result;
	}
}
```

OverridingMethodDelete.java

```
/* @author: Maryam Umar */
public class OverridingMethodDelete {
	public String StackMain() {
		String result = "";
		Stack st = new Stack();
		st.addElement(1);
		st.addElement(2);
		st.addElement(3);
		result = result + st.removeNode();
		return result;
	}
}
```

OverridingMethodDeleteTest.java

```
/* @author: Maryam Umar */
public class OverridingMethodDeleteTest {
	public String test1() {
		String result = "";
		LinkedList st = new LinkedList();
		st.addElement(1);
		st.addElement(2);
		st.addElement(3);
		st.addElement(4);
		result = result + st.removeNode(4);
		st.addElement(4);
		result = result + st.ConvertToString();
		return result;
	}
	public String test2() {
		String result = "";
		LinkedList st = new LinkedList();
		st.addElement(1);
		st.addElementAfter(3, 1);
		st.addElement(2);
		result = result + st.ConvertToString();
		return result;
	}
	public String test3() {
		String result = "";
		LinkedList st = new LinkedList();
		st.addElement(1);
		st.addElement(2);
		st.addElementAfter(3, 1);
		result = result + st.ConvertToString();
		return result;
	 }
	public String test4() {
		String result = "";
		LinkedList l1 = new LinkedList();
		LinkedList l2 = new LinkedList();
		l1.addElement(1);
		l2.addElement(4);
		l1.addElement(2);
		l2.addElement(5);
		l2.addElement(6);
		l1.addElement(3);
		result = result + l2.ConvertToString();
		result = result + l1.ConvertToString();
		return result;
	 }
	public String test5() {
		String result = "";
		Stack st = new Stack();
		st.addElement(1);
```

```
		st.addElementAfter(2, 1);
		result = result + st.removeNode();
		return result;
	}
	public String test6() {
		String result = "";
		Stack st = new Stack();
		st.addElement(1);
		st.addElement(2);
		result = result + st.ConvertToString();
		return result;
	}
	public String test7() {
		String result = "";
		Stack st = new Stack();
		st.addElement(1);
		st.addElement(2);
		st.addElement(3);
		st.addElement(4);
		result = result + st.removeNode(4);
		st.addElement(4);
		result = result + st.removeNode(4);
		return result;
	}
	public String test8() {
		String result = "";
		Stack st = new Stack();
		Stack st2 = new Stack();
		st.addElement(1);
		st.addElement(2);
		st.addElement(3);
		st2.addElement(4);
		st2.addElement(5);
		st2.addElement(6);
		st.removeNode();
		st.removeNode();
		result = result + st.ConvertToString();
		result = result + st2.ConvertToString();
		return result;
	}
	public String test9() {
		String result = "";
		OverridingMethodDelete TCO = new OverridingMethodDelete();
		result = result + TCO.StackMain();
		return result;
	}
}
```

LinkedListThis.java

```
/* @author: Maryam Umar */
public class LinkedListThis {
	protected ListNode start;
	protected ListNode end;
	protected int size;
	public LinkedListThis(){
		start = null;
		end = null;
		size = 0;
	}
	public void setSize(int s){
//		size = s;
	}
	public void addElement(int obj){
		if (obj == 0)
			throw new NullPointerException();
		ListNode newNode = new ListNode(obj);
		if(start == null){
			start = newNode;
			end = newNode;
			size++;
		}
		else{
```

```
                end.nextPtr = newNode;
                end = newNode;
                size++;
        }
}
public ListNode find(int obj){
        if(obj == 0)
                return null;
        ListNode temp = start;
        while(temp != null){
                if(temp.data == obj)
                        return temp;
                temp = temp.nextPtr;
        }
        return null;
}
public ListNode findBefore(int obj){
        if(obj == 0)
                return null;
        ListNode prevTemp = null;
        ListNode temp = start;
        while(temp != null){
                if(temp.data == obj)
                        return prevTemp;
                prevTemp = temp;
                temp = temp.nextPtr;
        }
        return null;
}
public void addElementAfter(int obj, int pos){
        if(obj == 0 || pos == 0)
                throw new NullPointerException();
        ListNode posNode = find(pos);
        if(posNode == null)
                throw new NullPointerException();
        ListNode newNode = new ListNode(obj);
        if(posNode == end){
                posNode.nextPtr = newNode;
                end = newNode;
                size++;
        }
        else {
                newNode.nextPtr = posNode.nextPtr;
                posNode.nextPtr = newNode;
                size++;
        }
}
public int removeNode(int obj){
        if(obj == 0)
                throw new NullPointerException();
        ListNode delNode = find(obj);
        if(delNode == null)
                return -1;
        ListNode prev = findBefore(obj);
        if(delNode == start){
                start = delNode.nextPtr;
                delNode = null;
        }
        else if      (delNode == end){
                end = prev;
                delNode = null;
        }
        else{
                prev.nextPtr = delNode.nextPtr;
                delNode = null;
        }
        size--;
        return 0;
}
public String ConvertToString(){
        ListNode tmp = start;
        String result = "";
```

```
		while(tmp != end){
			result += Integer.toString(tmp.data) + "";
			tmp = tmp.nextPtr;
		}
		result += Integer.toString(end.data);
		return result;
	}
}
```

StackThis.java

```
/* @author: Maryam Umar */
public class StackThis extends LinkedListThis {
	public StackThis(){
		end = null;
	}
	public void addElement(int obj){
		if (obj == 0)
			throw new NullPointerException();
		ListNode newNode = new ListNode(obj);
		if(size == 0){
			start = newNode;
			end = start;
			size++;
		}
		else{
			end.nextPtr = newNode;
			end = newNode;
			size++;
		}
	}
	public ListNode findBefore(){
		ListNode temp = start;
		int ctr = 1;

		while(ctr != (size-1)){
			temp = temp.nextPtr;
			ctr++;
		}
		return temp;
	}
	public int removeNode(){
		ListNode prev = findBefore();
		int value;
		if(size == 0)
			return 0;
		else if(size == 1){
			value = end.data;
			start = null;
			end = null;
			size--;
			return value;
		}
		else{
			value = end.data;
			end = prev;
			size--;
			return value;
		}
	}
	public String ConvertToString(){
		ListNode tmp = start;
		String result = "";
		while(tmp != end){
			result += Integer.toString(tmp.data) + "";
			tmp = tmp.nextPtr;
		}
		result += Integer.toString(end.data);
		return result;
	}
}
```

ThisKeywordMain.java

```
/* @author: Maryam Umar */
public class ThisKeywordMain {
	public String ThisMain() {
		String result = "";
		StackThis st = new StackThis();
		st.setSize(4);
		st.addElement(1);
		st.addElement(2);
		st.addElement(3);
		st.addElement(4);
		st.setSize(8);
		st.addElement(5);
		result = result + st.ConvertToString();
		return result;
	}
}
```

ThisKeywordMainTest.java

```
/* @author: Maryam Umar */
public class ThisKeywordMainTest {
	public String test1() {
		String result = "";
		LinkedListThis st = new LinkedListThis();
		st.setSize(3);
		st.addElement(1);
		st.addElement(2);
		st.addElement(3);
		st.addElement(4);
		result = result + st.removeNode(4);
		st.setSize(4);
		st.addElement(4);
		result = result + st.ConvertToString();
		return result;
	}
	public String test2() {
		String result = "";
		LinkedListThis st = new LinkedListThis();
		st.addElement(1);
		st.addElementAfter(3, 1);
		st.addElement(2);
		result = result + st.ConvertToString();
		return result;
	}
	public String test3() {
		String result = "";
		LinkedListThis st = new LinkedListThis();
		st.addElement(1);
		st.addElement(2);
		st.addElementAfter(3, 1);
		result = result + st.ConvertToString();
		return result;
	}
	public String test4() {
		String result = "";
		LinkedListThis l1 = new LinkedListThis();
		LinkedListThis l2 = new LinkedListThis();
		l1.addElement(1);
		l2.addElement(4);
		l1.addElement(2);
		l2.addElement(5);
		l2.addElement(6);
		l1.addElement(3);
		result = result + l2.ConvertToString();
		result = result + l1.ConvertToString();
		return result;
	}
	public String test5() {
		String result = "";
		StackThis st = new StackThis();
		st.addElement(1);
		st.addElementAfter(2, 1);
```

```
		result = result + st.removeNode();
		return result;
	}
	public String test6() {
		String result = "";
		LinkedListThis st = new LinkedListThis();
		st.addElement(1);
		st.addElement(2);
		st.addElement(3);
		st.removeNode(2);
		result = result + st.ConvertToString();
		return result;
	}
	public String test7() {
		String result = "";
		LinkedListThis st = new LinkedListThis();
		st.addElement(1);
		st.addElement(2);
		st.addElement(3);
		st.removeNode(1);
		result = result + st.ConvertToString();
		return result;
	}
	public String test8() {
		String result = "";
		LinkedListThis st = new LinkedListThis();
		st.addElement(1);
		st.addElement(2);
		st.addElement(3);
		st.addElement(4);
		st.addElement(5);
		st.removeNode(3);
		result = result + st.ConvertToString();
		return result;
	}
}
```

LinkedListStatic.java

```
/* @author: Maryam Umar */
public class LinkedListStatic {
	protected ListNode start;
	protected ListNode end;
	protected int size;
	public LinkedListStatic(){
		start = null;
		end = null;
		size = 0;
	}
	public void setSize(int size){
//		size = 0;
	}
	public void addElement(int obj){
		if (obj == 0)
			throw new NullPointerException();

		ListNode newNode = new ListNode(obj);
		if(start == null){
			start = newNode;
			end = newNode;
			size++;
		}
		else{
			end.nextPtr = newNode;
			end = newNode;
			size++;
		}
	}
	public ListNode find(int obj){
		if(obj == 0)
			return null;
		ListNode temp = start;
		while(temp != null){
```

```
                if(temp.data == obj)
                        return temp;
                temp = temp.nextPtr;
        }
        return null;
    }
    public ListNode findBefore(int obj){
        if(obj == 0)
                return null;
        ListNode prevTemp = null;
        ListNode temp = start;
        while(temp != null){
                if(temp.data == obj)
                        return prevTemp;
                prevTemp = temp;
                temp = temp.nextPtr;
        }
        return null;
    }
    public void addElementAfter(int obj, int pos){
        if(obj == 0 || pos == 0)
                throw new NullPointerException();
        ListNode posNode = find(pos);
        if(posNode == null)
                throw new NullPointerException();
        ListNode newNode = new ListNode(obj);
        size++;
        if(posNode == end){
                posNode.nextPtr = newNode;
                end = newNode;
        }
        else{
                newNode.nextPtr = posNode.nextPtr;
                posNode.nextPtr = newNode;
        }
    }
    public int removeNode(int obj){
        if(obj == 0)
                throw new NullPointerException();
        ListNode delNode = find(obj);
        if(delNode == null)
                return -1;
        ListNode prev = findBefore(obj);
        if(delNode == start){
                start = delNode.nextPtr;
                delNode = null;
        }
        else if      (delNode == end){
                end = prev;
                delNode = null;
        }
        else{
                prev.nextPtr = delNode.nextPtr;
                delNode = null;
        }
        size--;
        return 0;
    }
    public String ConvertToString(){
        ListNode tmp = start;
        String result = "";
        while(tmp != end){
                result += Integer.toString(tmp.data) + "";
                tmp = tmp.nextPtr;
        }
        result += Integer.toString(end.data);
        return result;
    }
}
```

StackStatic.java

```
/* @author: Maryam Umar */
public class StackStatic extends LinkedListStatic {
	protected ListNode end;			//corresponds to Top of Stack
	protected int maxsize;
	public StackStatic(){
		end = null;
		size = 0;
	}
	public void setSize(int size){
		maxsize = size;
	}
	public void addElement(int obj){
		if (obj == 0)
			throw new NullPointerException();
		ListNode newNode = new ListNode(obj);
		if(size < maxsize){
			if(start == null){
				start = newNode;
				end = start;
				size++;
			}
			else{
				end.nextPtr = newNode;
				end = newNode;
				size++;
			}
		}
	}
	public ListNode findBefore(){
		ListNode temp = start;
		int ctr = 1;
		while(ctr != (size-1)){
			temp = temp.nextPtr;
			ctr++;
		}
		return temp;
	}
	public int removeNode(){
		ListNode prev = findBefore();
		int value;
		if(size == 0)
			return 0;
		else if(size == 1){
			value = end.data;
			start = null;
			end = null;
			size--;
			return value;
		}
		else{
			value = end.data;
			end = prev;
			size--;
			return value;
		}
	}
	public String ConvertToString(){
		ListNode tmp = start;
		String result = "";
		while(tmp != end){
			result += Integer.toString(tmp.data) + "";
			tmp = tmp.nextPtr;
		}
		result += Integer.toString(end.data);
		return result;
	}
}
```

StaticKeywordMain.java

```
/* @author: Maryam Umar */
public class StaticKeywordMain {
```

```
	public String StaticMain() {
		String result = "";
		StackStatic st = new StackStatic();
		st.setSize(4);
		st.addElement(1);
		st.addElement(2);
		st.addElement(3);
		st.addElement(4);
		result = result + st.removeNode();
		return result;
	}
}
```

StaticKeywordMainTest.java

```
/* @author: Maryam Umar */
public class StaticKeywordMainTest {
	public String test1() {
		String result = "";
		LinkedListStatic st = new LinkedListStatic();
		st.setSize(3);
		st.addElement(1);
		st.addElement(2);
		st.addElement(3);
		st.addElement(4);
		result = result + st.removeNode(4);
		st.setSize(4);
		st.addElement(4);
		result = result + st.ConvertToString();
		return result;
	}
	public String test2() {
		String result = "";
		LinkedListStatic st = new LinkedListStatic();
		st.addElement(1);
		st.addElementAfter(3, 1);
		st.addElement(2);
		result = result + st.ConvertToString();
		return result;
	}
	public String test3() {
		String result = "";
		LinkedListStatic st = new LinkedListStatic();
		st.setSize(3);
		st.addElement(1);
		st.addElement(2);
		st.addElementAfter(3, 1);
		result = result + st.ConvertToString();
		return result;
	 }
	public String test4() {
		String result = "";
		LinkedListStatic l1 = new LinkedListStatic();
		LinkedListStatic l2 = new LinkedListStatic();
		l1.setSize(3);
		l2.setSize(3);
		l1.addElement(1);
		l2.addElement(4);
		l1.addElement(2);
		l2.addElement(5);
		l2.addElement(6);
		l1.addElement(3);
		result = result + l2.ConvertToString();
		result = result + l1.ConvertToString();
		return result;
	 }
	public String test5() {
		String result = "";
		StackStatic st = new StackStatic();
		st.setSize(2);
		st.addElement(1);
		st.addElementAfter(2, 1);
		result = result + st.removeNode();
```

```
        return result;
    }
    public String test6() {
        String result = "";
        StackStatic st = new StackStatic();
        st.setSize(2);
        st.addElement(1);
        st.addElement(2);
        result = result + st.ConvertToString();
        return result;
    }
    public String test7() {
        String result = "";
        StackStatic st = new StackStatic();
        st.setSize(3);
        st.addElement(1);
        st.addElement(2);
        st.addElement(3);
        st.addElement(4);
        result = result + st.removeNode(4);
        st.setSize(4);
        st.addElement(4);
        result = result + st.removeNode(4);
        return result;
    }
    public String test8() {
        String result = "";
        StackStatic st = new StackStatic();
        StackStatic st2 = new StackStatic();
        st.setSize(3);
        st2.setSize(3);
        st.addElement(1);
        st.addElement(2);
        st.addElement(3);
        st2.addElement(4);
        st2.addElement(5);
        st2.addElement(6);
        st.removeNode();
        st.removeNode();
        result = result + st.ConvertToString();
        result = result + st2.ConvertToString();
        return result;
    }
    public String test9() {
        String result = "";
        StaticKeywordMain st = new StaticKeywordMain();
        result = result + st.StaticMain();
        return result;
    }
    public String test10() {
        String result = "";
        LinkedListStatic st = new LinkedListStatic();
        st.addElement(1);
        st.addElement(2);
        st.addElement(3);
        st.removeNode(2);
        result = result + st.ConvertToString();
        return result;
    }
    public String test11() {
        String result = "";
        LinkedListStatic st = new LinkedListStatic();
        st.addElement(1);
        st.addElement(2);
        st.addElement(3);
        st.removeNode(1);
        result = result + st.ConvertToString();
        return result;
    }
    public String test12() {
        String result = "";
        StackStatic st = new StackStatic();
```

```
		st.addElement(1);
		st.addElement(2);
		st.addElement(3);
		st.addElement(4);
		st.removeNode();
		result = result + st.ConvertToString();
		return result;
	}
	public String test13() {
		String result = "";
		LinkedListStatic st = new LinkedListStatic();
		st.addElement(1);
		st.addElement(2);
		st.addElement(3);
		st.addElement(4);
		st.addElement(5);
		st.removeNode(3);
		result = result + st.ConvertToString();
		return result;
	}
}
```

LinkedListIOR.java

```
/* @author: Maryam Umar */
public class LinkedListIOR {
	protected ListNode start;
	protected ListNode end;
	protected int size;
	public LinkedListIOR(){
		start = null;
		end = null;
	}
	public void setSize(){
		size = 0;
	}
	public void addElement(int obj){
		if (obj == 0)
			throw new NullPointerException();
		ListNode newNode = new ListNode(obj);
		if(start == null){
			setSize();
			start = newNode;
			end = newNode;
			size++;
		}
		else{
			end.nextPtr = newNode;
			end = newNode;
			size++;
		}
	}
	public ListNode find(int obj){
		if(obj == 0)
			return null;
		ListNode temp = start;
		while(temp != null){
			if(temp.data == obj)
				return temp;
			temp = temp.nextPtr;
		}
		return null;
	}
	public ListNode findBefore(int obj){
		if(obj == 0)
			return null;
		ListNode prevTemp = null;
		ListNode temp = start;
		while(temp != null){
			if(temp.data == obj)
				return prevTemp;
			prevTemp = temp;
			temp = temp.nextPtr;
```

```
		}
		return null;
	}
	public void addElementAfter(int obj, int pos){
		if(obj == 0 || pos == 0)
			throw new NullPointerException();
		ListNode posNode = find(pos);
		if(posNode == null)
			throw new NullPointerException();
		ListNode newNode = new ListNode(obj);
		size++;
		if(posNode == end){
			posNode.nextPtr = newNode;
			end = newNode;
		}
		else{
			newNode.nextPtr = posNode.nextPtr;
			posNode.nextPtr = newNode;
		}
	}
	public int removeNode(int obj){
		if(obj == 0)
			throw new NullPointerException();
		ListNode delNode = find(obj);
		if(delNode == null)
			return -1;
		ListNode prev = findBefore(obj);
		if(delNode == start){
			start = delNode.nextPtr;
			delNode = null;
		}
		else if	(delNode == end){
			end = prev;
			delNode = null;
		}
		else{
			prev.nextPtr = delNode.nextPtr;
			delNode = null;
		}
		size--;
		return 0;
	}
	public String ConvertToString(){
		ListNode tmp = start;
		String result = "";
		while(tmp != end){
			result += Integer.toString(tmp.data) + "";
			tmp = tmp.nextPtr;
		}
		result += Integer.toString(end.data);
		return result;
	}
}
```

StackIOR.java

```
/* @author: Maryam Umar */
public class StackIOR extends LinkedListIOR {
	protected ListNode end;			//corresponds to Top of Stack
	public void setSize(){
		size = 0;
		end = null;
	}
	public void addElement(int obj){
		if (obj == 0)
			throw new NullPointerException();
		ListNode newNode = new ListNode(obj);
		if(start == null){
			setSize();
			start = newNode;
			end = start;
			size++;
		}
```

```
		else{
			end.nextPtr = newNode;
			end = newNode;
			size++;
		}
	}
	public ListNode findBefore(){
		ListNode temp = start;
		int ctr = 1;
		while(ctr != (size-1)){
			temp = temp.nextPtr;
			ctr++;
		}
		return temp;
	}
	public int removeNode(){
		ListNode prev = findBefore();
		int value;
		if(size == 0)
			return 0;
		else if(size == 1){
			value = end.data;
			start = null;
			end = null;
			size--;
			return value;
		}
		else{
			value = end.data;
			end = prev;
			size--;
			return value;
		}
	}
	public String ConvertToString(){
		ListNode tmp = start;
		String result = "";
		while(tmp != end){
			result += Integer.toString(tmp.data) + "";
			tmp = tmp.nextPtr;
		}
		result += Integer.toString(end.data);
		return result;
	}
}
```

OverriddenMethodRename.java

```
/* @author: Maryam Umar */
public class OverriddenMethodRename {
	public String OverMain() {
		String result = "";
		StackIOR st = new StackIOR();
		st.addElement(1);
		st.addElement(2);
		st.addElement(3);
		result = result + st.removeNode();
		return result;
	}
}
```

OverriddenMethodRenameTest.java

```
/* @author: Maryam Umar */
public class OverriddenMethodRenameTest {
	public String test1() {
		String result = "";
		LinkedListIOR st = new LinkedListIOR();
		st.addElement(1);
		st.addElement(2);
		st.addElement(3);
		st.addElement(4);
		result = result + st.removeNode(4);
		st.addElement(4);
```

```
        result = result + st.ConvertToString();
        return result;
    }
    public String test2() {
        String result = "";
        LinkedListIOR st = new LinkedListIOR();
        st.addElement(1);
        st.addElementAfter(3, 1);
        st.addElement(2);
        result = result + st.ConvertToString();
        return result;
    }
    public String test3() {
        String result = "";
        LinkedListIOR st = new LinkedListIOR();
        st.addElement(1);
        st.addElement(2);
        st.addElementAfter(3, 1);
        result = result + st.ConvertToString();
        return result;
     }
    public String test4() {
        String result = "";
        LinkedListIOR l1 = new LinkedListIOR();
        LinkedListIOR l2 = new LinkedListIOR();
        l1.addElement(1);
        l2.addElement(4);
        l1.addElement(2);
        l2.addElement(5);
        l2.addElement(6);
        l1.addElement(3);
        result = result + l2.ConvertToString();
        result = result + l1.ConvertToString();
        return result;
     }
    public String test5() {
        String result = "";
        StackIOR st = new StackIOR();
        st.addElement(1);
        st.addElementAfter(2, 1);
        result = result + st.removeNode();
        return result;
    }
    public String test6() {
        String result = "";
        StackIOR st = new StackIOR();
        st.addElement(1);
        st.addElement(2);
        result = result + st.ConvertToString();
        return result;
    }
    public String test7() {
        String result = "";
        StackIOR st = new StackIOR();
        StackIOR st2 = new StackIOR();
        st.addElement(1);
        st.addElement(2);
        st.addElement(3);
        st2.addElement(4);
        st2.addElement(5);
        st2.addElement(6);
        st.removeNode();
        st.removeNode();
        result = result + st.ConvertToString();
        result = result + st2.ConvertToString();
        return result;
    }
    public String test8() {
        String result = "";
        StackIOR st = new StackIOR();
        st.addElement(1);
        st.addElement(2);
```

```
		st.addElement(3);
		st.removeNode(2);
		result = result + st.ConvertToString();
		return result;
	}
	public String test9() {
		String result = "";
		LinkedListIOR st = new LinkedListIOR();
		st.addElement(1);
		st.addElement(2);
		st.addElement(3);
		st.removeNode(1);
		result = result + st.ConvertToString();
		return result;
	}
	public String test10() {
		String result = "";
		StackIOR st = new StackIOR();
		st.addElement(1);
		st.addElement(2);
		st.addElement(3);
		st.addElement(3);
		st.removeNode();
		result = result + st.ConvertToString();
		return result;
	}
}
```

LinkedListSuper.java

```
/* @author: Maryam Umar */
public class LinkedListSuper {
	protected ListNode start;
	protected ListNode end;
	protected int size;
	public LinkedListSuper(){
		start = null;
		end = null;
	}
	public void setSize(){
		size = 0;
	}
	public void addElement(int obj){
		if (obj == 0)
			throw new NullPointerException();
		ListNode newNode = new ListNode(obj);
		if(start == null){
			setSize();
			start = newNode;
			end = newNode;
			size++;
		}
		else{
			end.nextPtr = newNode;
			end = newNode;
			size++;
		}
	}
	public ListNode find(int obj){
		if(obj == 0)
			return null;
		ListNode temp = start;
		while(temp != null){
			if(temp.data == obj)
				return temp;
			temp = temp.nextPtr;
		}
		return null;
	}
	public ListNode findBefore(int obj){
		if(obj == 0)
			return null;
		ListNode prevTemp = null;
```

```
		ListNode temp = start;
		while(temp != null){
			if(temp.data == obj)
				return prevTemp;
			prevTemp = temp;
			temp = temp.nextPtr;
		}
		return null;
	}
	public void addElementAfter(int obj, int pos){
		if(obj == 0 || pos == 0)
			throw new NullPointerException();
		ListNode posNode = find(pos);
		if(posNode == null)
			throw new NullPointerException();
		ListNode newNode = new ListNode(obj);
		size++;
		if(posNode == end){
			posNode.nextPtr = newNode;
			end = newNode;
		}
		else{
			newNode.nextPtr = posNode.nextPtr;
			posNode.nextPtr = newNode;
		}
	}
	public int removeNode(int obj){
		if(obj == 0)
			throw new NullPointerException();
		ListNode delNode = find(obj);
		if(delNode == null)
			return -1;
		ListNode prev = findBefore(obj);
		if(delNode == start){
			start = delNode.nextPtr;
			delNode = null;
		}
		else if	(delNode == end){
			end = prev;
			delNode = null;
		}
		else{
			prev.nextPtr = delNode.nextPtr;
			delNode = null;
		}
		size--;
		return 0;
	}
	public String ConvertToString(){
		ListNode tmp = start;
		String result = "";
		while(tmp != end){
			result += Integer.toString(tmp.data) + "";
			tmp = tmp.nextPtr;
		}
		result += Integer.toString(end.data);
		return result;
	}
}
```

StackSuper.java

```
/* @author: Maryam Umar */
public class StackSuper extends LinkedListSuper {
	protected ListNode end;			//corresponds to Top of Stack
	public StackSuper(){
		end = null;
	}
	public void setSize(){
		size = 0;
	}
	public void addElement(int obj){
		if (obj == 0)
```

```
                throw new NullPointerException();
            ListNode newNode = new ListNode(obj);
            if(start == null){
                super.setSize();
                start = newNode;
                end = start;
                size++;
            }
            else{
                end.nextPtr = newNode;
                end = newNode;
                size++;
            }
        }
        public ListNode findBefore(){
            ListNode temp = start;
            int ctr = 1;
            while(ctr != (size-1)){
                temp = temp.nextPtr;
                ctr++;
            }
            return temp;
        }
        public int removeNode(){
            ListNode prev = findBefore();
            int value;
            if(size == 0)
                return 0;
            else if(size == 1){
                value = end.data;
                start = null;
                end = null;
                size--;
                return value;
            }
            else{
                value = end.data;
                end = prev;
                size--;
                return value;
            }
        }
        public String ConvertToString(){
            ListNode tmp = start;
            String result = "";
            while(tmp != end){
                result += Integer.toString(tmp.data) + "";
                tmp = tmp.nextPtr;
            }
            result += Integer.toString(end.data);
            return result;
        }
}
```

SuperKeyword.java

```
/* @author: Maryam Umar */
public class SuperKeyword {
        public String SuperMain() {
            String result = "";
            StackSuper st = new StackSuper();
            st.addElement(1);
            st.addElement(2);
            st.addElement(3);
            result = result + st.removeNode();
            return result;
        }
}
```

SuperKeywordTest.java

```
/* @author: Maryam Umar */
public class SuperKeywordTest {
        public String test1() {
```

```
        String result = "";
        LinkedListSuper st = new LinkedListSuper();
        st.addElement(1);
        st.addElement(2);
        st.addElement(3);
        st.addElement(4);
        result = result + st.removeNode(4);
        st.addElement(4);
        result = result + st.ConvertToString();
        return result;
    }
    public String test2() {
        String result = "";
        LinkedListSuper st = new LinkedListSuper();
        st.addElement(1);
        st.addElementAfter(3, 1);
        st.addElement(2);
        result = result + st.ConvertToString();
        return result;
    }
    public String test3() {
        String result = "";
        LinkedListSuper st = new LinkedListSuper();
        st.addElement(1);
        st.addElement(2);
        st.addElementAfter(3, 1);
        result = result + st.ConvertToString();
        return result;
     }
    public String test4() {
        String result = "";
        LinkedListSuper l1 = new LinkedListSuper();
        LinkedListSuper l2 = new LinkedListSuper();
        l1.addElement(1);
        l2.addElement(4);
        l1.addElement(2);
        l2.addElement(5);
        l2.addElement(6);
        l1.addElement(3);
        result = result + l2.ConvertToString();
        result = result + l1.ConvertToString();
        return result;
     }
    public String test5() {
        String result = "";
        StackSuper st = new StackSuper();
        st.addElement(1);
        st.addElementAfter(2, 1);
        result = result + st.removeNode();
        return result;
    }
    public String test6() {
        String result = "";
        StackSuper st = new StackSuper();
        st.addElement(1);
        st.addElement(2);
        result = result + st.ConvertToString();
        return result;
    }
    public String test7() {
        String result = "";
        StackSuper st = new StackSuper();
        st.addElement(1);
        st.addElement(2);
        st.addElement(3);
        st.addElement(4);
        result = result + st.removeNode(4);
        st.addElement(4);
        result = result + st.removeNode(4);
        return result;
    }
    public String test8() {
```

```
		String result = "";
		StackSuper st = new StackSuper();
		StackSuper st2 = new StackSuper();
		st.addElement(1);
		st.addElement(2);
		st.addElement(3);
		st2.addElement(4);
		st2.addElement(5);
		st2.addElement(6);
		st.removeNode();
		st.removeNode();
		result = result + st.ConvertToString();
		result = result + st2.ConvertToString();
		return result;
	}
	public String test9() {
		String result = "";
		SuperKeyword st = new SuperKeyword();
		result = result + st.SuperMain();
		return result;
	}
	public String test10() {
		String result = "";
		StackSuper st = new StackSuper();
		st.addElement(1);
		st.addElement(2);
		st.addElement(3);
		st.removeNode(2);
		result = result + st.ConvertToString();
		return result;
	}
	public String test11() {
		String result = "";
		LinkedListSuper st = new LinkedListSuper();
		st.addElement(1);
		st.addElement(2);
		st.addElement(3);
		st.removeNode(1);
		result = result + st.ConvertToString();
		return result;
	}
	public String test12() {
		String result = "";
		StackSuper st = new StackSuper();
		st.addElement(1);
		st.addElement(2);
		st.addElement(3);
		st.addElement(3);
		st.removeNode();
		result = result + st.ConvertToString();
		return result;
	}
}
```

LinkedListOM.java

```
/* @author: Maryam Umar */
public class LinkedListOM {
	protected ListNode start;
	protected ListNode end;
	protected int size;
	public LinkedListOM(){
		start = null;
		end = null;
		size = 0;
	}
	public void addElement(int obj){
		if (obj == 0)
			throw new NullPointerException();
		ListNode newNode = new ListNode(obj);
		if(start == null){
			start = newNode;
			end = newNode;
```

```
                size++;
        }
        else{
                end.nextPtr = newNode;
                end = newNode;
                size++;
        }
}
public ListNode find(int obj){
        if(obj == 0)
                return null;
        ListNode temp = start;
        while(temp != null){
                if(temp.data == obj)
                        return temp;
                temp = temp.nextPtr;
        }
        return null;
}
public ListNode findBefore(int obj){
        if(obj == 0)
                return null;
        ListNode prevTemp = null;
        ListNode temp = start;
        while(temp != null){
                if(temp.data == obj)
                        return prevTemp;
                prevTemp = temp;
                temp = temp.nextPtr;
        }
        return null;
}
public void addElement(int obj, int pos){
        if(obj == 0 || pos == 0)
                throw new NullPointerException();
        ListNode posNode = find(pos);
        if(posNode == null)
                throw new NullPointerException();
        ListNode newNode = new ListNode(obj);
        if(posNode == end){
                posNode.nextPtr = newNode;
                end = newNode;
                size++;
        }
        else{
                newNode.nextPtr = posNode.nextPtr;
                posNode.nextPtr = newNode;
                size++;
        }
}
public int removeNode(int obj){
        if(obj == 0)
                throw new NullPointerException();
        ListNode delNode = find(obj);
        if(delNode == null)
                return -1;
        ListNode prev = findBefore(obj);
        if(delNode == start){
                start = delNode.nextPtr;
                delNode = null;
        }
        else if      (delNode == end){
                end = prev;
                delNode = null;
        }
        else{
                prev.nextPtr = delNode.nextPtr;
                delNode = null;
        }
        size--;
        return 0;
}
```

```
	public String ConvertToString(){
		ListNode tmp = start;
		String result = "";
		while(tmp != end){
			result += Integer.toString(tmp.data) + "";
			tmp = tmp.nextPtr;
		}
		result += Integer.toString(end.data);
		return result;
	}
}
```

OverMethodMain.java

```
/* @author: Maryam Umar */
public class OverMethodMain {
	public String OverMain() {
		String result = "";
		LinkedListOM ll = new LinkedListOM();
			ll.addElement(1);
		ll.addElement(2);
		ll.addElement(3, 2);
		ll.addElement(4, 2);
		result = result + ll.ConvertToString();
		return result;
	}
}
```

OverMethodMainTest.java

```
/* @author: Maryam Umar */
public class OverMethodMainTest {
	public String test1() {
		String result = "";
		OverMethodMain om = new OverMethodMain();
		result = result + om.OverMain();
		return result;
	}
	public String test2() {
		String result = "";
		LinkedListOM st = new LinkedListOM();
		st.addElement(1);
		st.addElement(2);
		st.addElement(3);
		st.addElement(4);
		result = result + st.removeNode(4);
		st.addElement(4);
		result = result + st.ConvertToString();
		return result;
	}
	public String test3() {
		String result = "";
		LinkedListOM l1 = new LinkedListOM();
		LinkedListOM l2 = new LinkedListOM();
		l1.addElement(1);
		l2.addElement(4);
		l1.addElement(2);
		l2.addElement(5);
		l2.addElement(6);
		l1.addElement(3);
		result = result + l2.ConvertToString();
		result = result + l1.ConvertToString();
		return result;
	}
	public String test4() {
		String result = "";
		LinkedListOM st = new LinkedListOM();
		st.addElement(1);
		st.addElement(2);
		st.addElement(3);
		st.removeNode(2);
		result = result + st.ConvertToString();
		return result;
	}
```

```
	public String test5() {
		String result = "";
		LinkedListOM st = new LinkedListOM();
		st.addElement(1);
		st.addElement(2);
		st.addElement(3);
		st.removeNode(1);
		result = result + st.ConvertToString();
		return result;
	}
	public String test6() {
		String result = "";
		LinkedListOM st = new LinkedListOM();
		st.addElement(1);
		st.addElement(2);
		st.addElement(3);
		st.addElement(4);
		st.addElement(5);
		st.removeNode(3);
		result = result + st.ConvertToString();
		return result;
	}
}
```

ListNodeOMD.java

```
/* @author: Maryam Umar */
public class ListNodeOMD {
	protected Object data = null;
	protected ListNodeOMD nextPtr = null;
	public ListNodeOMD(Object obj){
		if(obj == null)
			throw new NullPointerException();
		data = obj;
	}
}
```

LinkedListOMD.java

```
/* @author: Maryam Umar */
public class LinkedListOMD {
	protected ListNodeOMD start;
	protected ListNodeOMD end;
	protected int size;
	public LinkedListOMD(){
		start = null;
		end = null;
		size = 0;
	}
	public void addElement(int obj){
		if (obj == 0)
			throw new NullPointerException();
		ListNodeOMD newNode = new ListNodeOMD(obj);
		if(start == null){
			start = newNode;
			end = newNode;
			size++;
		}
		else{
			end.nextPtr = newNode;
			end = newNode;
			size++;
		}
	}
	public void addElement(double obj){
		if (obj == 0)
			throw new NullPointerException();
		ListNodeOMD newNode = new ListNodeOMD(obj);
		if(start == null){
			start = newNode;
			end = newNode;
			size++;
		}
		else{
```

```
                end.nextPtr = newNode;
                end = newNode;
                size++;
        }
}
public ListNodeOMD find(int obj){
        if(obj == 0)
                return null;
        ListNodeOMD temp = start;
        while(temp != null){
                if(temp.data.equals(obj))
                        return temp;
                temp = temp.nextPtr;
        }
        return null;
}
public ListNodeOMD findBefore(int obj){
        if(obj == 0)
                return null;
        ListNodeOMD prevTemp = null;
        ListNodeOMD temp = start;
        while(temp != null){
                if(temp.data.equals(obj))
                        return prevTemp;
                prevTemp = temp;
                temp = temp.nextPtr;
        }
        return null;
}
public void addElement(int obj, int pos){
        if(obj == 0 || pos == 0)
                throw new NullPointerException();
        ListNodeOMD posNode = find(pos);
        if(posNode == null)
                throw new NullPointerException();
        ListNodeOMD newNode = new ListNodeOMD(obj);
        if(posNode == end){
                posNode.nextPtr = newNode;
                end = newNode;
                size++;
        }
        else{
                newNode.nextPtr = posNode.nextPtr;
                posNode.nextPtr = newNode;
                size++;
        }
}
public int removeNode(int obj){
        if(obj == 0)
                throw new NullPointerException();
        ListNodeOMD delNode = find(obj);
        if(delNode == null)
                return -1;
        ListNodeOMD prev = findBefore(obj);
        if(delNode == start){
                start = delNode.nextPtr;
                delNode = null;
        }
        else if      (delNode == end){
                end = prev;
                delNode = null;
        }
        else{
                prev.nextPtr = delNode.nextPtr;
                delNode = null;
        }
        size--;
        return 0;
}
public String ConvertToString(){
        ListNodeOMD tmp = start;
        String result = "";
```

```
            while(tmp != end){
                  result += tmp.data.toString() + "";
                  tmp = tmp.nextPtr;
            }
            result += end.data.toString();
            return result;
      }
}
```

OverMethodDelete.java

```
/* @author: Maryam Umar */
public class OverMethodDelete {
      public String OverDeleteMain() {
            String result = "";
            LinkedListOMD ll = new LinkedListOMD();
            ll.addElement(1);
            ll.addElement(2.0);
            ll.addElement(3);
            ll.addElement(4.5);
            result = result + ll.ConvertToString();
            return result;
      }
}
```

OverMethodDeleteTest.java

```
/* @author: Maryam Umar */
public class OverMethodDeleteTest {
      public String test1() {
            String result = "";
            LinkedListOMD st = new LinkedListOMD();
            st.addElement(1);
            st.addElement(2);
            st.addElement(3);
            st.addElement(4);
            result = result + st.removeNode(4);
            st.addElement(4);
            result = result + st.ConvertToString();
            return result;
      }
      public String test2() {
            String result = "";
            LinkedListOMD st = new LinkedListOMD();
            st.addElement(1);
            st.addElement(3, 1);
            st.addElement(2);
            result = result + st.ConvertToString();
            return result;
      }
      public String test3() {
            String result = "";
            LinkedListOMD st = new LinkedListOMD();
            st.addElement(1);
            st.addElement(2.0);
            st.addElement(3, 1);
            result = result + st.ConvertToString();
            return result;
       }
      public String test4() {
            String result = "";
            LinkedListOMD l1 = new LinkedListOMD();
            LinkedListOMD l2 = new LinkedListOMD();
            l1.addElement(1.5);
            l2.addElement(4);
            l1.addElement(2);
            l2.addElement(5.0);
            l2.addElement(6);
            l1.addElement(3);
            result = result + l2.ConvertToString();
            result = result + l1.ConvertToString();
            return result;
       }
      public String test5() {
```

```
		String result = "";
		LinkedListOMD st = new LinkedListOMD();
		st.addElement(1);
		st.addElement(2, 1);
		result = result + st.removeNode(1);
		result = result + st.ConvertToString();
		return result;
	}
	public String test6() {
		String result = "";
		LinkedListOMD st = new LinkedListOMD();
		st.addElement(1);
		st.addElement(2);
		st.addElement(3);
		st.removeNode(2);
		result = result + st.ConvertToString();
		return result;
	}
	public String test7() {
		String result = "";
		LinkedListOMD st = new LinkedListOMD();
		st.addElement(1);
		st.addElement(2);
		st.addElement(3);
		st.removeNode(1);
		result = result + st.ConvertToString();
		return result;
	}
	public String test8() {
		String result = "";
		LinkedListOMD st = new LinkedListOMD();
		st.addElement(1);
		st.addElement(2);
		st.addElement(3);
		st.addElement(4);
		st.addElement(5);
		st.removeNode(3);
		result = result + st.ConvertToString();
		return result;
	}
}
```

LinkedListIPC.java

```
/* @author: Maryam Umar */
public class LinkedListIPC {
	protected ListNode start;
	protected ListNode end;
	protected int size;
	protected int maxSize;
	public LinkedListIPC(){
		start = null;
		end = null;
		size = 0;
	}
	public LinkedListIPC(int size){
		maxSize = size;
	}
	public void addElement(int obj){
		if (obj == 0)
			throw new NullPointerException();
		ListNode newNode = new ListNode(obj);
		if(start == null){
			start = newNode;
			end = newNode;
			size++;
		}
		else if(size < maxSize){
			end.nextPtr = newNode;
			end = newNode;
			size++;
		}
		else{
```

```
                newNode = new ListNode(-1);
                end.nextPtr = newNode;
                end = newNode;
        }
}
public ListNode find(int obj){
        if(obj == 0)
                return null;
        ListNode temp = start;
        while(temp != null){
                if(temp.data == obj)
                        return temp;
                temp = temp.nextPtr;
        }
        return null;
}
public ListNode findBefore(int obj){
        if(obj == 0)
                return null;
        ListNode prevTemp = null;
        ListNode temp = start;
        while(temp != null){
                if(temp.data == obj)
                        return prevTemp;
                prevTemp = temp;
                temp = temp.nextPtr;
        }
        return null;
}
public void addElementAfter(int obj, int pos){
        if(obj == 0 || pos == 0)
                throw new NullPointerException();
        ListNode posNode = find(pos);
        if(posNode == null)
                throw new NullPointerException();
        ListNode newNode = new ListNode(obj);
        if(posNode == end && size < maxSize){
                posNode.nextPtr = newNode;
                end = newNode.nextPtr;
                size++;
        }
        else if(size < maxSize){
                newNode.nextPtr = posNode.nextPtr;
                posNode.nextPtr = newNode;
                size++;
        }
        else{
                newNode = new ListNode(-1);
                end.nextPtr = newNode;
                end = newNode;
        }
}
public int removeNode(int obj){
        if(obj == 0)
                throw new NullPointerException();
        ListNode delNode = find(obj);
        if(delNode == null)
                return -1;
        ListNode prev = findBefore(obj);
        if(delNode == start){
                start = delNode.nextPtr;
                delNode = null;
        }
        else if      (delNode == end){
                end = prev;
                delNode = null;
        }
        else{
                prev.nextPtr = delNode.nextPtr;
                delNode = null;
        }
        size--;
```

```
		return 0;
	}
	public String ConvertToString(){
		ListNode tmp = start;
		String result = "";
		while(tmp != end){
			result += Integer.toString(tmp.data) + "";
			tmp = tmp.nextPtr;
		}
		result += Integer.toString(end.data);
		return result;
	}
}
```

StackIPC.java

```
/* @author: Maryam Umar */
public class StackIPC extends LinkedListIPC {
	protected ListNode end;
	public StackIPC(){
		end = null;
	}
	public StackIPC(int size){
		super(size);
		maxSize = size;
	}
	public void addElement(int obj){
		if (obj == 0)
			throw new NullPointerException();
		ListNode newNode = new ListNode(obj);
		if(size == 0){
			start = newNode;
			end = start;
			size++;
		}
		else if(size < maxSize){
			end.nextPtr = newNode;
			end = newNode;
			size++;
		}
		else{
			newNode = new ListNode(-1);
			end.nextPtr = newNode;
			end = newNode;
		}
	}
	public ListNode findBefore(){
		ListNode temp = start;
		int ctr = 1;
		while(ctr != (size-1)){
			temp = temp.nextPtr;
			ctr++;
		}
		return temp;
	}
	public int removeNode(){
		ListNode prev = findBefore();
		int value;
		if(size == 0)
			return 0;
		else if(size == 1){
			value = end.data;
			start = null;
			end = null;
			size--;
			return value;
		}
		else{
			value = end.data;
			end = prev;
			size--;
			return value;
		}
```

```
	}
	public String ConvertToString(){
		ListNode tmp = start;
		String result = "";
		while(tmp != end){
			result += Integer.toString(tmp.data) + "";
			tmp = tmp.nextPtr;
		}
		result += Integer.toString(end.data);
		return result;
	}
}
```

ParentConstructorDeletionIPC.java

```
/* @author: Maryam Umar */
public class ParentConstructorDeletionIPC {
	public String ParentMain() {
		String result = "";
		StackIPC st = new StackIPC(4);
		st.addElement(1);
		st.addElement(2);
		st.addElement(3);
		st.addElement(4);
		st.addElement(5);
		result = result + st.removeNode();
		return result;
	}
}
```

ParentConstructorDeletionTest.java

```
/* @author: Maryam Umar */
public class ParentConstructorDeletionTest {
	public String test1(){
		String result = "";
		LinkedListIPC st = new LinkedListIPC(4);
		st.addElement(1);
		st.addElement(2);
		st.addElement(3);
		st.addElement(4);
		result = result + st.removeNode(4);
		st.addElement(4);
		result = result + st.ConvertToString();
		return result;
	}
	public String test2() {
		String result = "";
		LinkedListIPC st = new LinkedListIPC(2);
		st.addElement(1);
		st.addElementAfter(3, 1);
		st.addElement(2);
		result = result + st.ConvertToString();
		return result;
	}
	public String test3() {
		String result = "";
		LinkedListIPC st = new LinkedListIPC();
		st.addElement(1);
		st.addElement(2);
		st.addElementAfter(3, 1);
		result = result + st.ConvertToString();
		return result;
	 }
	public String test4() {
		String result = "";
		LinkedListIPC l1 = new LinkedListIPC();
		LinkedListIPC l2 = new LinkedListIPC();
		l1.addElement(1);
		l2.addElement(4);
		l1.addElement(2);
		l2.addElement(5);
		l2.addElement(6);
		l1.addElement(3);
```

```
        result = result + l2.ConvertToString();
        result = result + l1.ConvertToString();
        return result;
     }
    public String test5() {
        String result = "";
        StackIPC st = new StackIPC(2);
        st.addElement(1);
        st.addElementAfter(2, 1);
        result = result + st.removeNode();
        return result;
    }
    public String test6() {
        String result = "";
        StackIPC st = new StackIPC(1);
        st.addElement(1);
        st.addElement(2);
        result = result + st.ConvertToString();
        return result;
    }
    public String test7() {
        String result = "";
        StackIPC st = new StackIPC(3);
        st.addElement(1);
        st.addElement(2);
        st.addElement(3);
        st.addElement(4);
        result = result + st.removeNode(4);
        st.addElement(4);
        result = result + st.ConvertToString();
        return result;
    }
    public String test8() {
        String result = "";
        StackIPC st = new StackIPC(4);
        StackIPC st2 = new StackIPC(2);
        st.addElement(1);
        st.addElement(2);
        st.addElement(3);
        st2.addElement(4);
        st2.addElement(5);
        st2.addElement(6);
        st.removeNode();
        st.removeNode();
        result = result + st.ConvertToString();
        result = result + st2.ConvertToString();
        return result;
    }
    public String test9() {
        String result = "";
        StackIPC st = new StackIPC(4);
        st.addElement(1);
        st.addElement(2);
        st.addElement(3);
        st.addElement(4);
        st.addElement(5);
        result = result + st.removeNode() + st.ConvertToString();
        return result;
    }
    public String test10() {
        String result = "";
        ParentConstructorDeletionIPC IPC = new ParentConstructorDeletionIPC();
        result = result + IPC.ParentMain();
        return result;
    }
    public String test11() {
        String result = "";
        LinkedListIPC st = new LinkedListIPC();
        st.addElement(1);
        st.addElement(2);
        st.addElement(3);
        st.removeNode(1);
```

```
		result = result + st.ConvertToString();
		return result;
	}
}
```

LinkedListIOP.java

```
/* @author: Maryam Umar */
public class LinkedListIOP {
	protected ListNode start;
	protected ListNode end;
	protected int size;
	public LinkedListIOP(){
		start = null;
		end = null;
	}
	public void setSize(){
		size = 0;
	}
	public void addElement(int obj){
		if (obj == 0)
			throw new NullPointerException();
		ListNode newNode = new ListNode(obj);
		size++;
		if(start == null){
			start = newNode;
			end = newNode;
			size = 1;
		}
		else{
			end.nextPtr = newNode;
			end = newNode;
		}
	}
	public ListNode find(int obj){
		if(obj == 0)
			return null;
		ListNode temp = start;
		while(temp != null){
			if(temp.data == obj)
				return temp;
			temp = temp.nextPtr;
		}
		return null;
	}
	public ListNode findBefore(int obj){
		if(obj == 0)
			return null;
		ListNode prevTemp = null;
		ListNode temp = start;
		while(temp != null){
			if(temp.data == obj)
				return prevTemp;
			prevTemp = temp;
			temp = temp.nextPtr;
		}
		return null;
	}
	public void addElementAfter(int obj, int pos){
		if(obj == 0 || pos == 0)
			throw new NullPointerException();
		ListNode posNode = find(pos);
		if(posNode == null)
			throw new NullPointerException();
		ListNode newNode = new ListNode(obj);
		size++;
		if(posNode == end){
			posNode.nextPtr = newNode;
			end = newNode;
		}
		else{
			newNode.nextPtr = posNode.nextPtr;
			posNode.nextPtr = newNode;
```

```
		}
	}
	public int removeNode(int obj){
		if(obj == 0)
			throw new NullPointerException();
		ListNode delNode = find(obj);
		if(delNode == null)
			return -1;
		ListNode prev = findBefore(obj);
		if(delNode == start){
			start = delNode.nextPtr;
			delNode = null;
		}
		else if	(delNode == end){
			end = prev;
			delNode = null;
		}
		else{
			prev.nextPtr = delNode.nextPtr;
			delNode = null;
		}
		size--;
		return 0;
	}
	public String ConvertToString(){
		ListNode tmp = start;
		String result = "";
		while(tmp != end){
			result += Integer.toString(tmp.data) + "";
			tmp = tmp.nextPtr;
		}
		result += Integer.toString(end.data);
		return result;
	}
}
```

StackIOP.java

```
/* @author: Maryam Umar */
public class StackIOP extends LinkedListIOP {
	protected ListNode end;			//corresponds to Top of Stack
	public StackIOP(){
		end = null;
	}
	public void setSize(){
		super.setSize();
		size = 5;
	}
	public void addElement(int obj){
		if (obj == 0)
			throw new NullPointerException();
		ListNode newNode = new ListNode(obj);
		if(size == 0){
			start = newNode;
			end = start;
			size++;
		}
		else{
			end.nextPtr = newNode;
			end = newNode;
			size++;
		}
	}
	public ListNode findBefore(){
		ListNode temp = start;
		int ctr = 1;
		while(ctr != (size-1)){
			temp = temp.nextPtr;
			ctr++;
		}
		return temp;
	}
	public int removeNode(){
```

```
            ListNode prev = findBefore();
            int value;
            if(size == 0)
                  return 0;
            else if(size == 1){
                  value = end.data;
                  start = null;
                  end = null;
                  size--;
                  return value;
            }
            else{
                  value = end.data;
                  end = prev;
                  size--;
                  return value;
            }
      }
      public String ConvertToString(){
            ListNode tmp = start;
            String result = "";
            while(tmp != end){
                  result += Integer.toString(tmp.data) + "";
                  tmp = tmp.nextPtr;
            }
            result += Integer.toString(end.data);
            return result;
      }
}
```

MethodPositionChange.java

```
/* @author: Maryam Umar */
public class MethodPositionChange {
      public String MethodMain() {
            String result = "";
            StackIOP st = new StackIOP();
            st.addElement(1);
            st.addElement(2);
            st.addElement(3);
            result = result + st.removeNode();
            return result;
      }
}
```

MethodPositionChangeTest.java

```
/* @author: Maryam Umar */
public class MethodPositionChangeTest {
      public String test1() {
            String result = "";
            LinkedListIOP st = new LinkedListIOP();
            st.addElement(1);
            st.addElement(2);
            st.addElement(3);
            st.addElement(4);
            result = result + st.removeNode(4);
            st.addElement(4);
            result = result + st.ConvertToString();
            return result;
      }
      public String test2() {
            String result = "";
            LinkedListIOP st = new LinkedListIOP();
            st.addElement(1);
            st.addElementAfter(3, 1);
            st.addElement(2);
            result = result + st.ConvertToString();
            return result;
      }
      public String test3() {
            String result = "";
            LinkedListIOP st = new LinkedListIOP();
            st.addElement(1);
```

```
        st.addElement(2);
        st.addElementAfter(3, 1);
        result = result + st.ConvertToString();
        return result;
    }
    public String test4() {
        String result = "";
        LinkedListIOP l1 = new LinkedListIOP();
        LinkedListIOP l2 = new LinkedListIOP();
        l1.addElement(1);
        l2.addElement(4);
        l1.addElement(2);
        l2.addElement(5);
        l2.addElement(6);
        l1.addElement(3);
        result = result + l2.ConvertToString();
        result = result + l1.ConvertToString();
        return result;
    }
    public String test5() {
        String result = "";
        StackIOP st = new StackIOP();
        st.addElement(1);
        st.addElementAfter(2, 1);
        result = result + st.removeNode();
        return result;
    }
    public String test6() {
        String result = "";
        StackIOP st = new StackIOP();
        st.addElement(1);
        st.addElement(2);
        result = result + st.ConvertToString();
        return result;
    }
    public String test7() {
        String result = "";
        StackIOP st = new StackIOP();
        st.addElement(1);
        st.addElement(2);
        st.addElement(3);
        st.addElement(4);
        result = result + st.removeNode(4);
        st.addElement(4);
        result = result + st.removeNode(4);
        return result;
    }
    public String test8() {
        String result = "";
        StackIOP st = new StackIOP();
        StackIOP st2 = new StackIOP();
        st.addElement(1);
        st.addElement(2);
        st.addElement(3);
        st2.addElement(4);
        st2.addElement(5);
        st2.addElement(6);
        st.removeNode();
        st.removeNode();
        result = result + st.ConvertToString();
        result = result + st2.ConvertToString();
        return result;
    }
    public String test9() {
        String result = "";
        StackIOP ll = new StackIOP();
        ll.setSize();
        ll.addElement(1);
        ll.addElement(2);
        ll.addElement(3);
        result = result + ll.ConvertToString();
        return result;
```

```
	}
	public String test10() {
		String result = "";
		MethodPositionChange TCO = new MethodPositionChange();
		result = result + TCO.MethodMain();
		return result;
	}
	public String test11() {
		String result = "";
		StackIOP st = new StackIOP();
		st.addElement(1);
		st.addElement(2);
		st.addElement(3);
		st.removeNode(2);
		result = result + st.ConvertToString();
		return result;
	}
	public String test12() {
		String result = "";
		LinkedListIOP st = new LinkedListIOP();
		st.addElement(1);
		st.addElement(2);
		st.addElement(3);
		st.removeNode(1);
		result = result + st.ConvertToString();
		return result;
	}
	public String test13() {
		String result = "";
		StackIOP st = new StackIOP();
		st.addElement(1);
		st.addElement(2);
		st.addElement(3);
		st.addElement(3);
		st.removeNode();
		result = result + st.ConvertToString();
		return result;
	}
}
```

REFERENCES

1. Adamopoulos, K., Harman, M. & Hierons, R. “How to Overcome the Equivalent Mutant Problem and Achieve Tailored Selective Mutation Using Co-evolution”. AAAI Genetic and Evolutionary Computation Conference 2004 (GECCO 2004). 26-30 June’04, Seattle, Washington, USA. LNCS 3103, Pages 1338-1349.
2. Hierons, R., Harman, M. & Danicic. S. “Using Program Slicing to Assist in the Detection of Equivalent Mutants”. Journal of Software Testing, Verification and Reliability, 9(4), 233-262, 1999.
3. Offutt, J. & Pan, J. “Automatically Detecting Equivalent Mutants and Infeasible Paths”. The Journal of Software Testing, Verification, and Reliability, Vol 7, No. 3, pages 165--192, September 1997
4. Offutt, J. & Craft, M. “Using Compiler Optimization Techniques to Detect Equivalent Mutants”. The Journal of Software Testing, Verification, and Reliability, 4(3):131--154, September 1994.
5. King, K. N. & Offutt, J. “A Fortran Language System for Mutation-Based Software Testing”. Software Practice and Experience, 21(7):686--718, July 1991.
6. Ma, Y. S., Offutt, J. & Kwon, Y. R. “MuJava: An Automated Class Mutation System”. Journal of Software Testing, Verification and Reliability, 15(2):97-133, June 2005.
7. Ma, Y. S., Kwon, Y. R. & Offutt, J. “Inter-Class Mutation Operators for Java”. Proceedings of the 13th International Symposium on Software Reliability Engineering, IEEE Computer Society Press, Annapolis MD, November 2002, pp. 352-363.
8. Offutt, A. J., & Untch, R. “Mutation 2000: Uniting The Orthogonal”. In Wong, W. E., ed.: Mutation Testing for the New Century (proceedings of Mutation 2000), San Jose, California, USA, Kluwer 2001, pp. 45 –55.
9. Ma, Y. S. & Offutt, J. “Description of Method-level Mutation Operators for Java”. November 2005.
10. Ma, Y. S. & Offutt, J. “Description of Class-level Mutation Operators for Java”. November 2005.
11. Agrawal, H., DeMillo, R.A., Hathaway, B., Hsu, W., Krauser, E. W., Martin, R.J., Mathur, A.P., & Spafford, E.H. “Design of Mutant Operators for the C programming Language”. Software Engineering Research Center, Purdue University. Indiana, March 1989.
12. Alexander, R. T., Bieman, J. M., Ghosh, S. & Ji, B. “Mutation of Java Objects”. Proceedings of the 13th International Symposium on Software Reliability Engineering (ISSRE'02). IEEE Computer Society. 2002.
13. Bybro, M. “A Mutation Testing Tool for Java Programs”. August 2003.
14. Bieman, J., Ghosh, S., & Alexander, R. “A Technique for Mutation of Java Objects”. Proceedings Automated Software Engineering (ASE 2001).
15. Moore, I. “Jester – A JUnit Test Tester”. Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP2001).
16. http://www.xpdeveloper.com/xpdwiki/Wiki.jsp?page=MutationTestingTools
17. http://www.ise.gmu.edu/~offutt/mujava/
18. http://jester.sourceforge.net
19. http://nester.sourceforge.net/

BIBLIOGRAPHY

1. http://en.wikipedia.org/wiki/Software_testing (accessed 20th july '06)
2. Frankl, P. G., Weiss, S. N., & Hu, C. "All-Uses versus Mutation Testing: An Experimental Comparison of Effectiveness". The Journal of Systems and Software, Sept. 1997.
3. Mathur, A. P., & Wong, W. E. "Comparing the Fault Detection Effectiveness of Mutation and Data Flow Testing: An Empirical Study". Technical Report SERC-TR-146-P, Software Engineering Research Center, Purdue University, Indiana, March 1993.
4. Mathur, A. P., & Wong, W. E. "Reducing the Cost of Mutation Testing: An Empirical Study". The Journal of Systems and Software. Vol 31, Issue 3. 1995.
5. Untch, R. H., Offutt, A. J. & Harrold, M. J. "Mutation Analysis using Mutant Schemata". International Symposium on Software Testing and Analysis**.** Proceedings of the 1993 ACM SIGSOFT international symposium on Software testing and Analysis.1993.
6. Offutt, A. J., Lee, A., Rothermel, G., Untch, R. & Zapf, C. "An Experimental Determination of Sufficient Mutation Operators". ACM Transactions on Software Engineering, (Baltimore, MD), IEEE Computer Society Press, , pp. 100 –107. May 1993..
7. Offutt, A. J. "A Practical System for Mutation Testing: Help for the common Programmer". Twelfth International Conference on Testing Computer Software. Pp 99 – 109. June 1995.
8. Offutt, A. J. & Lee, S. "An Empirical Evaluation of Weak Mutation". IEEE Transactions of Software Engineering, Vol 20, Issue 5, pp. 337—345. May 1994.
9. Offutt, A. J., Rothermel, G. & Zapf, C. "An Experimental Evaluation of Selective Mutation". International Conference on Software Engineering**.** Proceedings of the 15th international conference on Software Engineering. Pp 100 – 107. 1993.
10. Offutt, A. J., Voas, J. & Payne, J. "Mutation Operators for Ada". Technical Report ISSE-TR—96-09, Information and Software Systems Engineering, George Mason University. October 1996.