DESIGNING A TOOL FOR MUTATION TESTING

PAWAN KUMAR - 091276

Under the Supervision of

Dr. PRADEEP KUMAR GUPTA
Sr. Lecturer, CSE & IT

May 2013

Submitted in partial fulfilment of the degree of

BACHELOR OF TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE ENGINEERING AND INFORMATION TECHNOLOGY

JAYPEE UNIVERSITY OF INFORMATION TECHNOLOGY
WAKNAGHAT, SOLAN

Certificate

This is to certify that project report entitled "**DESIGNING A TOOL FOR MUTATION TESTING**", submitted by **PAWAN KUMAR(091276)** in partial fulfillment for the award of degree of Bachelor of Technology in Computer Science Engineering to Jaypee University of Information Technology, Waknaghat, Solan has been carried out under my supervision.

This work has not been submitted partially or fully to any other University or Institute for the award of this or any other degree or diploma.

Date:

Dr. PRADEEP KUMAR GUPTA
Sr. Lecturer, CSE& IT

Acknowledgement

I would like to take this oppurtinity to express my sincere indebtness and sense of graditude to all those who have contributed greatly towards the successful partial completion of my project "Designing a tool for Mutation Testing".

It would not have been possible to see through the undertaken project without the guidance and constant support of our guide Dr. Pradeep Kumar Gupta. For his coherent guidance I feel

fortunate to be taught by him, who gave me his unwavering support. I owe my heartiest thanks to Brig. (Retd.) S.P.Ghrera (H.O.D.-CSE/IT Department) who've always inspired confidence in me to take initiative.

As a final note, I am grateful to CSE and IT Department of Jaypee University of Information and Technology ,who inspired me to undertake difficult tasks by their strength of understanding my calibre and my requirements and taught me to work with patience and provided constant encouragement to successfully complete the project.

Date: **Pawan Kumar**

Table of Content

List of Figures

Title **Page No.**

List of Tables

Title **Page No.**

Abstract

Mutation testing takes a different approach to testing by asking questions about the efficacy of test cases. It is a fault-based technique that measures the effectiveness of test suites for fault localization, base on seeded faults. The fault detection effectiveness of a test suite is defined as the percentage of faults that can be detected by that test suite. A mutation is a change made to the source code by a mutation testing tool. Faults are introduced into the program by creating a collection of faulty versions, called mutants. These mutants are created from the original program by applying mutation operators which describe syntactic changes to the original code. The test suite is then used to execute these mutants and to measure how well it is able to find faults. A test case that distinguishes (returning a different result) the program from one or more of its mutants is considered to be effective in finding faults. A mutation score is a quantitative measurement of the quality of the test suite. This report describes an automated mutation testing tool for Java programs. It describes the framework of

the proposed tool and how the various components of the tool are interconnected to each other. Also it provides the various mutation operators present in the tool.

1. INTRODUCTION

Software testing is an integral part of software development cycle which is aimed at evaluating an attribute or capability of a program or system and determining that it meets the specified requirements. It is crucial for the quality of software and widely deployed by programmers and testers. Due to the limited understanding of the principles of software, Software testing still remains an art. The difficulty associated with software testing arises from the complexity of software. It is practically not feasible to test a program with moderate complexity completely. Testing is more than just debugging. The testing can be done to assure the quality of the software, or to verify and validate the system, or to estimate the reliability of the product. Testing can be used as a generic metric as well. Software testing is a trade-off between budget, time and quality.

1.1 Humans, Errors and Testing

Errors are a part of our daily life. Humans make errors in their thoughts, in their actions, and in the products that might result from their actions. Errors occur almost everywhere. Humans

can make errors in any field, for example in observation, in speech, in medical prescription, in driving, in sports, and similarly even in software development. An error might be insignificant in that it leads to a gentle friendly smile, such as when a slip of tongue occurs. Or, an error may lead to a catastrophe, such as when an operator fails to recognize that a relief valve on the pressurizer was stuck open and this resulted in a disastrous radiation leak.

To determine whether there are any errors in our thought, actions, and the products generated, we resort to the process of testing. The primary goal of testing is to determine if the thoughts, actions, and products are as desired, that is they conform to the requirements. Testing of thoughts is usually designed to determine if a concept or method has been understood satisfactorily. Testing of actions is designed to check if a skill that results in the actions has been acquired satisfactorily. Testing of a product is designed to check if the product behaves as desired. Both semantic and syntax errors arise during programming .Given that most modern compilers are able to detect syntactic errors, testing focuses on semantic errors, also known as faults that cause the program under test to behave incorrectly.

1.1.1 Errors, Faults and Failures

A programmer writes a program. An *error* occurs in the process of writing a program. A *fault* is the manifestation of one or more errors. A *failure* occurs when a faulty piece of code is executed leading to an incorrect state that propagates to the program's output. The programmer might misinterpret the requirements and consequently write incorrect code. Upon execution, the program might display behaviour that does not match with the expected behaviour, implying thereby that a failure has occurred. A fault in the program is also commonly referred to as a *bug* or a *defect*. The term error and bug are by far the most common ways of referring to something wrong in the program text that might lead to a failure. In Figure 1.1, notice the separation of observable from observed behaviour. This separation is important because it is the observed behaviour that might lead one to conclude that a program has failed.

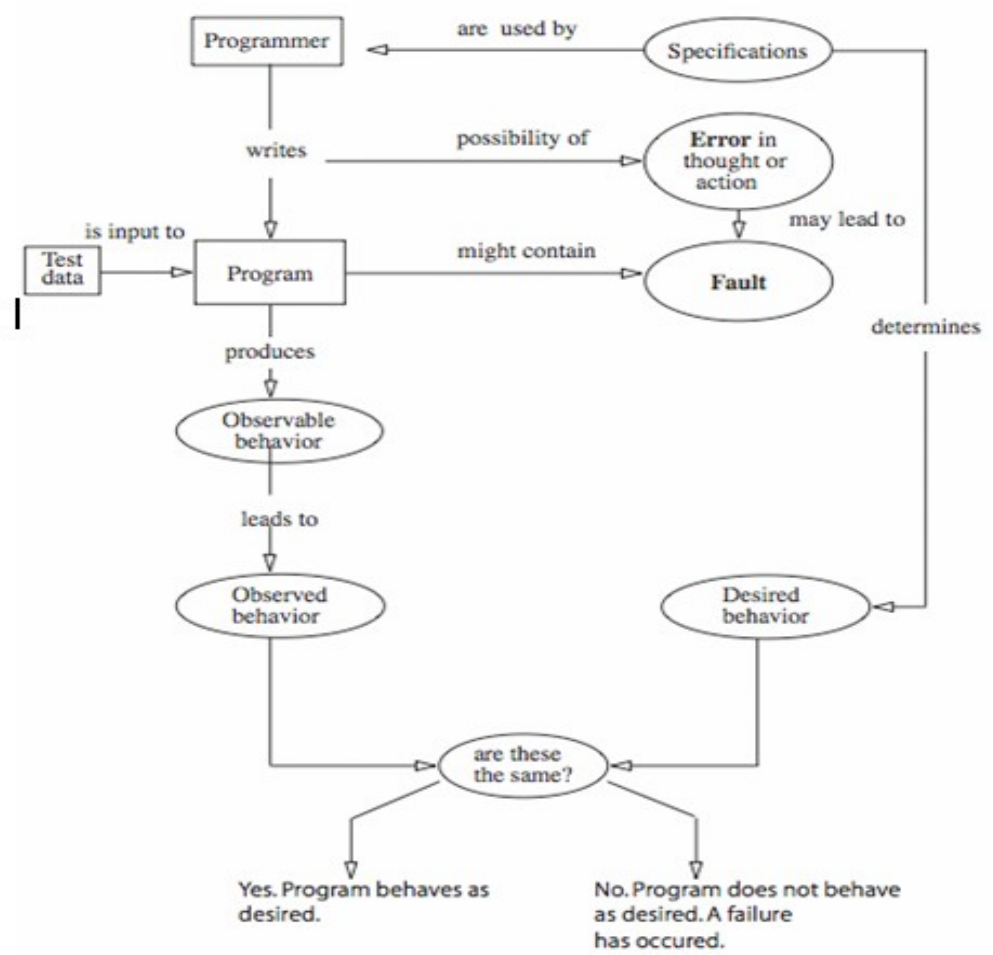

Fig 1.1 Errors, Faults and Failures in the process of programming and testing.

1.1.2 Test Automation

Testing of complex systems can be a human intensive task. Often one needs to execute thousands of tests to ensure that, for example, a change made to a component of an application does not cause previously correct code to malfunction. Execution of many tests can be a tiring as well as error-prone. Hence, there is a tremendous need for automating testing tasks.

Most software development organizations automate test- related tasks such as regression testing, graphical user interface (GUI) testing, and I/O device driver testing. Unfortunately, the process of test automation cannot be generalized. For example, automating regression tests for an embedded device such as a pacemaker is quite different from that for an I/O device driver that connects to the USB port of a PC. Such lack of generalization often leads to specialized test automation tools developed in-house.

Nevertheless, there do exist general-purpose tools for test automation. While such tools might not be applicable in all test environments, they are useful in many of them. Examples of such tools include Eggplant, Marathon, and Pounder for GUI testing; eLoadExpert, DBMonster, JMeter, WAPT, LoadRunner, and Grinder for performance or load testing; and Echleon, TestTube, WinRunner and XTest for regression testing. Despite the existence of a large number and variety of test automation tools, large development organizations develop their own test automation tools due primarily to the unique nature of their test requirements.

1.2 Software Quality

We all want high-quality software. There exist several definitions of software quality. Also, one quality attribute might be more important to a user than another quality attribute. In any case, software quality is a multidimensional quantity and is measurable.

1.2.1 Quality Attributes

There exist several measures of software quality. These can be divided into static and dynamic quality attributes. Static quality attributes refer to the actual code and related documentation. Dynamic quality attributes relate to the behaviour of the application in use.

Static quality attributes include structured, maintainable, and testable code as well as the availability of correct and complete documentation. We might come across complaints such as “Product X is excellent, I like the features it offers, but its user manual stinks!” In this case, the user manual brings down the overall product quality. If a maintenance engineer has been assigned the task of doing corrective maintenance on an application code, he will most likely need to understand portions of the code before he make any changes to it. This is where attributes related to items such as code documentation, understandability, and structure come into play. A poorly documented piece of code will be harder to understand and hence difficult to modify. Further, poorly structured code might be harder to modify and difficult to test.

Dynamic quality attributes include software reliability, correctness, completeness, consistency, usability, and performance. Reliability refers to the probability of failure-free operation. Correctness refers to the correct operation of an application and is always with reference to some artifact. For a tester, correctness is with respect to the requirements; for a user, it is often with respect to a user manual. Completeness refers to the availability of all the features listed in the requirements or in the manual. An incomplete software is one that does

not fully implement all the features required. Consistency refers to adherence to a common set of conventions and assumptions. Usability refers to the ease with which an application can be used. This is an area in itself and there exist techniques for usability testing. Psychology plays an important role in the design of techniques for usability testing. Usability testing refers to testing of a product by its potential users. The development organization invites a selected set of potential users and asks them to test the product. Users in turn test for ease of use, functionality as expected, performance, safety, and security. Usability testing is sometimes referred to as user-centric testing. Performance refers to the time the application takes to perform a requested task. Performance is considered as a non-functional requirement.

1.3 Software Testing

Software Testing is the process of Software Development Life Cycle (SDLC) which aims to find errors in a program or system. It involves various activities to be performed whose aim is to evaluate the capability or attributes of the software and determine that it meets its specified requirements. Software is quite different from other physical processes which involve input and output. Where software differs is in the manner in which it does not satisfy its requirements. Software can fail in many bizarre ways unlike other physical systems which fail in a relatively small set of ways. It is rare and almost impossible to detect all of the different failure modes for the software.

Unlike most physical systems where manufacturing defects occur, most of the defects that occur in software are design errors. Software usually does not suffer from corrosion and wear-and-tear. Software generally does not change until its upgraded version is not available, or until it reaches obsolescence. So, the design defects or bugs will remain buried in the software or system, once the software is shipped.

A software module of considerable size will always have some defects present in it. It is not because the programmers are careless or irresponsible, but because the complexity associated with the software is generally intractable and humans have only limited ability to handle the complexity. It is also true that for any complex systems, design defects can never be completely ruled out.

Discovering the design defects in software is equally difficult, for the same reason of complexity. Because software and any digital systems are not continuous, testing boundary

values are not sufficient to guarantee correctness. All the possible values need to be tested and verified, but complete testing is infeasible. Exhaustively testing a simple program to add only two integer inputs of 32-bits (yielding 2^64 distinct test cases) would take hundreds of years, even if tests were performed at a rate of thousands per second. Obviously, for a realistic software module, the complexity can be far beyond the example mentioned here. If inputs from the real world are involved, the problem will get worse, because timing and unpredictable environmental effects and human interactions are all possible input parameters under consideration.

A further complication has to do with the dynamic nature of programs. If a failure occurs during preliminary testing and the code is changed, the software may now work for a test case that it didn't work for previously. But its behaviour on pre-error test cases that it passed before can no longer be guaranteed. To account for this possibility, testing should be restarted. The expense of doing this is often prohibitive.

Regardless of these limitations, testing is an integral part of software development life cycle. It is broadly deployed in the every phase of development cycle. Typically, more than 50% percent of the software development time is spent in testing the product. Testing is usually performed for the following purposes:

To improve quality

As computers and software are used in critical applications, the outcome of a bug can be severe at times. Bugs can cause huge losses. Bugs in critical systems have caused crashes of the airplane, allowed space shuttle missions to go awry, halted trading on the stock market, and even worse. Bugs can cause disasters. The so-called year 2000 (Y2K) bug has given birth to a cottage industry of consultants and programming tools dedicated to making sure the modern world doesn't come to a screeching halt on the first day of the next century. In a computerized embedded world, the quality and reliability of software is a matter of life and death.

Quality means the conformance to the specified design requirement. Being correct, the minimum requirement of quality, means performing as required under specified circumstances. Debugging, an another approach of software testing, is performed to find out design defects made by the programmer. The imperfection of human nature makes it almost

impossible to make a moderately complex program correct the first time. Finding the problems and get them fixed is the purpose of debugging in programming phase.

For Verification & Validation (V&V)

Another important purpose of testing is verification and validation (V&V). Testing can serve as metrics. It is used as a tool in the V&V process. Testers can make claims based on interpretations of the testing results, which either the product works under certain situations, or it does not work. We can also compare the quality among different products under the same specification, based on results from the same test.

We cannot test quality directly, but we can test related factors to make quality visible. Quality has three sets of factors- functionality, engineering, and adaptability. These three sets of factors can be thought of as dimensions in the software quality space. Each dimension may be broken down into its component factors and considerations at successively lower levels of detail. Table 1 illustrates some of the most frequently cited quality considerations.

Table 1.1 Typical Software Quality Factors

Functionality(exterior quality)	Engineering (interior quality)	Adaptability (future quality)
Correctness	Efficiency	Flexibility
Reliability	Testability	Reusability
Usability	Documentation	Maintainability
Integrity	Structure	

Good testing provides measures for all relevant factors. The importance of any particular factor varies from application to application. Any system where human lives are at stake must place extreme emphasis on reliability and integrity. In the typical business system usability and maintainability are the key factors, while for a one-time scientific program neither may be significant. Our testing, to be fully effective, must be geared to measuring each relevant factor and thus forcing quality to become tangible and visible.

Tests with the purpose of validating the product works are named clean tests, or positive tests. The drawbacks are that it can only validate that the software works for the specified test cases. A finite number of tests are not sufficient to validate that the software works for all situations. On the contrary, only one failed test is sufficient enough to show that the software does not work. The tests aiming at breaking the software, or showing that it does not work are

known as Dirty tests or negative tests. A piece of software must have sufficient exception handling capabilities to survive a significant level of dirty tests.

A testable design is a design that can be easily validated, falsified and maintained. Because testing requires significant time and cost, is a rigorous effort and design for testability is also an important design rule for software development.

For reliability estimation

Software reliability has important relations with many aspects of software, including the structure, and the amount of testing it has been subjected to. Based on an operational profile (an estimate of the relative frequency of use of various inputs to the program), testing can serve as a statistical sampling method to gain failure data for reliability estimation.

Software testing is not mature. It still remains an art, because we still cannot make it a science. We are still using the same testing techniques invented 20-30 years ago, some of which are crafted methods or heuristics rather than good engineering methods. Software testing can be costly, but not testing software is even more expensive, especially in places that human lives are at stake. Solving the software-testing problem is no easier than solving the Turing halting problem. We can never be sure that a piece of software is correct. We can never be sure that the specifications are correct. No verification system can verify every correct program. We can never be certain that a verification system is correct either.

1.4 Key Concepts

1.4.1 Taxonomy

There is a number of testing methods and testing techniques available today, serving different purposes in software development life cycle. Classified by source of test generation, software testing can be classified in two categories: Black-box testing and White-box testing. Classified by life-cycle phase, software testing can be classified into the following categories: Unit testing, Integration testing, System testing, Regression testing and Beta testing. Classified by the nature of the goal for which testing is performed, software testing can be classified into three categories: Robustness testing, Stress testing, Performance testing and Load testing.

1.4.2 Source of Test Generation

Test Generation is an essential part of testing; it is as wedded to testing as the earth is to the sun. There are a variety of ways to generate tests. Tests could be generated from informally or formally specified requirements and with or without the aid of the code that is under test.

Black-box testing

The black-box testing is a testing approach in which test data is generated from the specified functional requirements without even considering the code of the program under consideration. It is also termed as data-driven or requirements-based testing. As the tester is only concerned with the functionality of the software, black-box testing is also referred as functional testing which can be defined as a testing method whose main focus is on executing the functions of the software and examining the input and output data of the software. The tester treats the software under test as a black box - only the inputs, outputs and specification are visible, and the functionality of the software is determined by observing the output set to the corresponding input set. In this testing approach, multiple inputs are exercised and the outputs are observed and compared against specification to validate the correctness of the product. Test cases are generated from the specifications of the software. No details of the code are considered while working on this approach of testing.

As the tester goes more and more deep in the input set, he will tend to explore more problems regarding the software and thereafter his confidence regarding the quality of the software will surely enhance. In an ideal case, the tester would want to exhaustively test the input set. But this approach towards testing the combinations of valid inputs will be a tedious task for most of the programs. Combinatorial explosion is the major roadblock in black-box testing. A tester could never be sure whether the specification provided to him is either correct or complete. Ambiguity is also another problem, due to limitations of the language used in the specifications (usually natural language). Even if we use some type of formal or restricted language, we may still fail to write down all the possible cases in the specification. Sometimes, the specification itself becomes an intractable problem: it is not possible to specify precisely every situation that can be encountered using limited words. Specification problems contributes approximately 30 percent of all bugs in software. People can rarely specify exactly what they want - they usually can tell whether a prototype is, or is not, what they want after they have been finished.

The research in black-box testing mainly focuses on how to maximize the effectiveness of testing with minimum cost, usually the number of test cases. It is not possible to exhaust the input space, but it is possible to exhaustively test a subset of the input space. Partitioning is one of the popular techniques used in black box testing. If we can partition the input space and assume all the input values in a partition is equivalent, then we only need to test one value in each partition to sufficiently cover the whole input space. Domain testing partitions the input domain into regions, and consider the input values in each domain an equivalent class. Domains can be exhaustively tested and covered by selecting a representative value(s) in each domain. Boundary values are of special interest in this approach. Research in this field has shown that test cases that explore boundary conditions have a higher payoff than test cases that do not. Boundary value analysis requires one or more boundary values selected as representative test cases. The difficulties with domain testing are that incorrect domain definitions in the specification cannot be efficiently discovered.

White-box testing

Contrary to black-box testing, in this approach of testing, software is viewed as a white-box, or glass-box, as the tester can see the structure and flow of the software under test. Testing plans are made according to the details of the software implementation, such as programming language, logic, and styles.

White-box testing refers to the test activity wherein code is used in the generation of or the assessment of test cases. It is rare, and almost impossible, to use white-box testing in isolation. As a test case contains of both inputs and expected outputs, one must use requirements to generate test cases, the code is used as an additional artefact in the generation process. However, there are techniques for generating tests exclusively from code and the corresponding expected output from requirements. For example, tools are available to generate tests to distinguish all mutants of a program under test or generate tests that force the program under test to exercise a given path. In any case, when someone claims they are using white-box testing, it is reasonable to conclude that they are using some forms of both black-box and white-box testing.

Code could be used directly or indirectly for test generation. In the direct case, a tool, or a human tester, examines the code and focuses on a given path to be covered. A test is generated to cover this path. In the indirect case, tests generated using some black-box

testing techniques are assessed against some code-based coverage criterion. Additional tests are then generated to cover the uncovered portions of code by analyzing which parts of the code are feasible. Control flow, data flow and mutation testing can be used for direct as well as indirect code-based test generation.

1.4.3 Life Cycle Phase based Testing

Testing activities take place throughout the software lifecycle. Each artifact produced is often subject to testing at different levels of rigor and using different testing techniques. Testing is often categorized based on the phase in which it occurs.

Programmers write code during the early coding phase. They test their code before it is integrated with other system components. This type of testing is referred to as Unit testing. When units are integrated and a large component or a subsystem formed, programmers do Integration testing of the subsystem. Eventually when the entire system has been built, its testing is referred to as System Testing. Test phases mentioned above differ in their timing and focus. In unit testing, a programmer focuses on the unit or a small component that has been developed. The goal is to ensure that unit functions correctly in isolation. In integration testing, the goal is to ensure that a collection of components function as desired. Integration errors are often discovered at this stage. The goal of system testing is to ensure that all the desired functionality is in the system and works as per its requirements.

Often a selected set of customers is asked to test a system before commercialization. This form of testing is known as Beta-testing. Errors reported by users of an application often lead to additional testing and debugging. Often changes made to an application are much smaller in their size when compared to the entire application. In such situations, one performs a regression test. The goal of regression testing is to ensure that the modified system functions as per its specifications. Test cases selected for regression testing include those designed to test the modified code and any other code that might be affected by the modifications.

1.4.3 Goal directed Testing

There exists a variety of goals. Of course, finding any hidden errors is the prime goal of testing, goal-oriented testing looks for specific types of failures. For example, the goal of vulnerability testing is to detect if there is any way by which system under test can be penetrated by unauthorized users.

Robustness Testing

Robustness testing refers to the task of testing an application for robustness against unintended inputs. It differs from functional testing in that the tests for robustness are derived from outside of the valid input space, whereas in the former the tests are derived from the valid input space.

Stress Testing

In Stress testing one checks for the behaviour of an application under stress. Handling of overflow of data storage, for example buffers, can be checked with the help of stress testing. Web applications can be tested by stressing them with a large number and variety of requests. The goal here is to find if the application continues to function correctly under stress.

Performance Testing

The term performance testing refers to that phase of testing where an application is tested specifically with performance requirements in view. For example, a compiler might be tested to check if it meets the performance requirements stated in terms of number of lines of code compiled per second.

Load Testing

The term load testing refers to that phase of testing in which an application is loaded with respect to one or more operations. The goal is to determine if the application continues to perform as required under various load conditions. For example, a database server can be loaded with requests from a large number of simulated users. While the server might work correctly when one or two users use it, it might fail in various ways when the number of users exceeds a threshold. During load testing one can determine whether the application is handling exceptions in an adequate manner .

2. MUTATION TESTING

2.1 Introduction

Software Testing is the activity aimed at evaluating an attribute or capability of a program or system and determining that it meets its required results. Mutation Testing is a fault based testing technique that measures the effectiveness of a test suite. Unlike other fault-based strategies that directly inject artificial faults into the program, this method generates simple syntactic deviations called mutants of the original program with the help of a set of rules known as Mutation Operators.

The goal of mutation testing is the generation of a test set that distinguishes the behavior of the mutants from the original program. If a test set can distinguish a mutant from the original program (i.e. produce different execution results), the mutant is said to be killed or

distinguished. Otherwise, the mutant is said to be a live mutant. A mutant may remain live because either it is equivalent to the original program (i.e.it is functionally identical to the original program although syntactically different) or the test set is inadequate to kill the mutant. The ratio of distinguished mutants over the total number of mutants, measures the adequacy of the test set. A test case is adequate if it is useful in detecting faults in a program. If the original program and all the mutant programs generate the same output, the test case is inadequate.

Our interest is in using the mutation technique to examine the adequacy of test data for object-oriented programs. Obviously, the traditional mutation method can be applied to OO programs. However, using an existing mutation system it is may not be sufficient to adequately test OO programs because the existing mutation systems were developed in non-OO programming environments. That is, the common programming errors that the traditional mutation systems model were derived from non-OO programming experience and thus they do not consider some kinds of errors likely to appear in OO programs.

In addition, the differences and new features in OO programming are likely to change the requirements for mutation testing. For instance, the conventional mutation systems make mutants of expressions, variables, and statements but do not mutate type and component (e.g. a data structure) declarations. Traditional programming simply

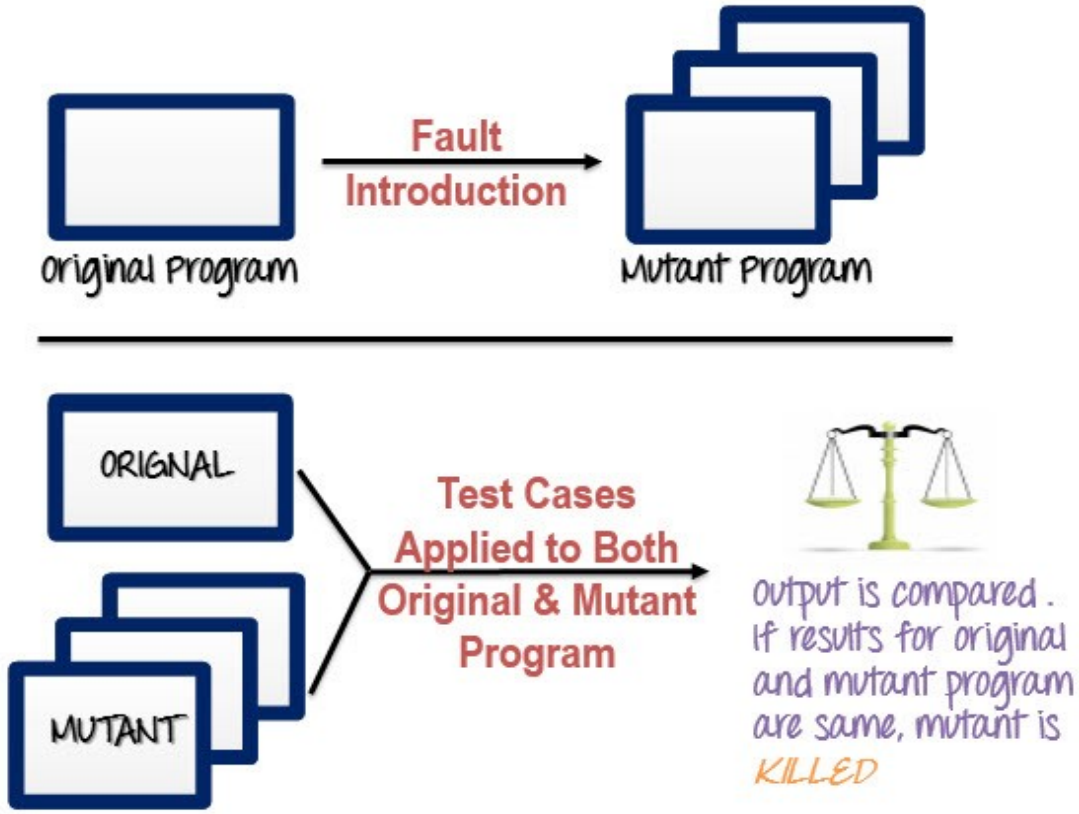

Fig 2.1 Description of Mutation Testing

makes use of the built-in types and entities of a language which are unlikely to contain many errors, so you wouldn't get much benefit by changing the declarations of those pre-defined program entities. However, OO programs are composed of user-defined data types (classes) and references to the user-defined types. It is very likely that user-defined components contain many defects such as mutual dependency between members/classes, inconsistencies or conflicts between the components developed by different programmers etc.

In object-oriented systems, mutation testing should also consider the relationships between components even though it is presently aimed at testing a single method or a class. For example, traditional mutation uses pre-fixed type compatibility (e.g. all arithmetic types are considered compatible) to replace a variable with other variables of compatible types. This is reasonable where there are only pre-defined built-in types. However, type compatibility of OO programs should be flexible because programmers can declare as many types as they want, which will change class structures. Compatibility thus needs to be dynamically determined by considering cluster structure at the time mutation is performed.

The effectiveness of mutation testing, like other fault-based approaches, heavily depends on the types of faults the mutation system is intended to represent, as they actually decide what to test and point out where analysis should be done. In our opinion, it is mainly the flaws

related to OO-specific features such as inheritance, polymorphism, and so on that the current mutation systems fail to adequately handle.

2.2 Founding Principles of Mutation Testing

Mutation Testing is a powerful testing technique for achieving correct or close to correct, programs. It rests on two fundamental principles. One principle is commonly known as the *competent programmer assumption*. The other is known as the *coupling effect.*

2.2.1 The Competent Programmer Hypothesis

The competent programmer hypothesis (CPH) arises from a simple observation made of practising programmers. The hypothesis states that given a problem statement, a programmer writes a program P that is in the general neighbourhood of the set of correct programs.

A reasonable interpretation of the CPH is that the program written to satisfy a set of requirements will be a few mutants away from a correct program. Thus, while the first version of the program might be incorrect, it could be corrected by a series of simple mutations. One might argue against the CPH by claiming something like “What about a missing conditional as the fault? One would need to add the missing conditional in order to arrive at a correct program.” Indeed, given a correct program P, one of its mutants is obtained by removing the condition from a conditional statement. Thus, a missing conditional does correspond to a simple mutant.

The CPH assumes that the programmer knows of an algorithm to solve the program at hand, and if not, will find one prior to writing the program. It is thus safe to assume that when asked to write a program to sort a list of numbers, a competent programmer knows of, and makes use of, at least one sorting algorithm. Certainly mistakes could be made while coding the algorithm. Such mistakes will lead to a program that can be corrected b applying one or more first-order mutations.

2.2.2 The Coupling Effect

While CPH arises out of observations of programmer behaviour, the coupling effect is observed empirically. The coupling effect has been paraphrased by DeMillo, Lipton, and Sayward as follows:

Test data that distinguishes all programs differing from a correct one by only simple errors is so sensitive that it also implicitly distinguishes more complex errors.

A seemingly simple first order mutant could be either equivalent to its program or not. For some input, a non-equivalent mutant forces a slight perturbation in the state space of the program under test. This perturbation takes place at the point of mutation and has the potential of infecting the entire state of the program. It is during an analysis of the behaviour of the mutant in relation to that of its parent that one discovers complex faults.

It may be easy to discover a fault that is a combination of many first-order mutations. Almost any test will likely discover such a fault. It is the subtle faults that are close to first-order mutations that are often difficult to detect. However, due to the coupling effect, a test set that distinguishes first-order mutants is likely to cause an erroneous program under test to fail.

2.3 Related Work

Research in mutation testing focuses on four kinds of activities: 1) defining mutation operators, 2) applying these mutation operators (experimentation), 3) developing tools, and 4) reducing the cost of mutation analysis.

The first one involves defining new mutation operators for different languages. The second research activity is experimentation with mutation operators. Empirical studies have supported the effectiveness of mutation testing. The third kind of activities in mutation testing research is developing mutation tools. Last but not least in terms of research activities is investigating ways to reduce the cost of mutation analysis. The major cost of mutation analysis arises from the computational expense of generating and running large numbers of mutant programs. Mutation testing is a powerful but time-consuming technique which is impractical to use without a reliable, fast and automated tool that generates mutants, runs the mutants against a suite of tests, and reports the mutation score of the test suite.

2.4 Class Mutation

Class Mutation is a mutation technique for OO (Java) programs. The main difference from the traditional mutation method is that it is targeted at plausible faults related to OO-specific features that Java provides – class declarations and references, single inheritance, information

hiding, and polymorphism. Faults are introduced into the program by a set of new mutation operators (predefined program modification rules).

2.5 Class Mutation Techniques

2.5.1 Polymorphic Types

In OO systems, it is common for a variable to have polymorphic types. That is, a variable may at runtime refer to an object of a different type from its declaration and to objects of different types at different times. This raises the possibility that not all objects that become attached to the same variable correctly support the same set of features. This may also cause runtime type errors which cannot always be detected at compile time. Two mutation operators, CRT and ICE, were designed to address this feature.

CRT (Compatible Reference Type replacement)

This operator replaces a reference type with compatible types in a cluster2. The compatible types are the names of other types (classes) that meet the widening/narrowing reference conversion rules in the Java language specification. For instance, the class type S can be replaced with the class type T provided that S is a subclass of T, or S can be replaced with the interface type K provided that S implements K.

```
class T { … }
class S extends T implements K { … }
class U extends S { … }
interface K { … }
```

Original code:

```
S s = new S();
```

CRT Mutants:

```
a) T s = new S();
b) K s = new S();
c) U s = new S();
```

ICE (class Instance Creation Expression changes)

While CRT handles the declared types, the ICE operator was designed to change the runtime type of an object. Java ‘instance creation expression’ creates an object of the class specified in the expression. The ICE operator replaces the class name in ‘instance creation expression’ with compatible class names. This results in calling the constructors of compatible types, which will create the objects of the replaced types.

For example, the original code above can have two mutants created by the ICE operator.

```
ICE Mutants:
a) S s = new T();
b) S s = new U();
```
Mutant a) will call the constructor in class T creating an object of type T (resulting in compilation errors), while mutant b) creates an object of type U through the constructor of class U.

PNC (new method call with child class type)

The POI operator changes the instantiated type of an object reference. This causes the object reference to refer to an object of a type that is different from the declared type.

In the example below, class Parent is the parent of class Child.

```
Original Code:
Parent a;
a = new Parent();

PNC Mutant:
Parent a;
a = new Child();
```

PMD (Member variable declaration with parent class type)

The PMD operator changes the declared type of an object reference to the parent of the original declared type. The instantiation will still be valid (it will still be a descendant of the new declared type). To kill this mutant, a test case must cause the behavior of the object to be incorrect with the new declared type.

In the example below, class Parent is the parent of class Child.

```
Original Code:
Child b;
b = new Child();

PMD Mutant:
Parent b;
b = new Child();
```

PPD (Parameter variable declaration with child class type)

The PPD operator is the same as the PMD, except that it operates on parameters rather than instance and local variables. It changes the declared type of a parameter object reference to be that of the parent of its original declared type.

In the example below, class Parent is the parent of class Child.

```
Original Code:
boolean equals (Child o)
{…..}
```

PPD Mutant:

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

2.5.2 Inheritance

Although a powerful and useful abstraction mechanism, incorrect use of inheritance can lead to a number of faults. We define five mutation operators to try to test the various aspects of using inheritance, covering variable shadowing, method overriding, the use of super, and definition of constructors.

Variable shadowing can cause instance variables that are defined in a subclass to shadow (or hide) member variables of the parent. However, this powerful feature can cause an incorrect variable to be accessed. Thus it is necessary to ensure that the correct variable is accessed when variable shadowing is used, which is the intent of the IHD and IHI mutation operators.

IHD (Hiding variable deletion)

The IHD operator deletes a hiding variable, a variable in a subclass that has the same name and type as a variable in the parent class. This causes references to that variable to access the variable defined in the parent (or ancestor). This mutant can only be killed by a test case that is able to show that the reference to the parent variable is incorrect.

Original Code:

```
class List {
 int size ;
 ... ...
}
class Stack extends List {
int size;
... ...
}
```

IHD Mutant:

```
class List {
 int size;
 ... ...
}
class Stack extends List}
int size;
... ...
}
```

IHI (Hiding variable insertion)

The IHI operator inserts a hiding variable into a subclass. It is a reverse case of IHD. By inserting a hiding variable, two variables (a hiding variable and a hidden variable) of the same name become to be exist. Newly defined and overriding methods in a subclass

reference the hiding variable although inherited methods reference the hidden variable as before.

Original Code:
```
class List {
int size;
... ...
}
class Stack extends List {
... ...
}
```

IHI Mutant:
```
class List {
int size;
... ...
}
class Stack extends List {
int size;
... ...
}
```

IOD (Overriding method deletion)

The IOD operator deletes an entire declaration of an over- riding method in a subclass so that references to the method uses the parent's version.The mutant act as if there is no overriding method for the method.

Original Code:
```
class Stack extends List {
... ...
void push (int a) {... }
}
```

IOD Mutant:
```
class Stack extends List {
... ...
void push (int a) {... }
}
```

IOP (Overridden method calling position change)

Sometimes, an overriding method in a child class needs to call the method it overrides in the parent class. This may happen if the parent's method uses a private variable v, which means the method in the child class may not modify v directly.

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

```
}
```

IOP Mutant:

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

2.5.3 Encapsulation

It is important to note that poor access definitions do not always cause faults initially, but can lead to faulty behavior when the class is integrated with other classes, modified, or inherited from.

AMC (Access modifier change)

The AMC operator changes the access level for instance variables and methods to other access levels. The purpose of the AMC operator is to guide testers to generate test cases that ensure that accessibility is correct.

2.5.4 Method Overloading

A class type may have more than one method with the same name as long as they have different signatures. There is more possibility of an unintended method being called, even if the correct method name is given, when several versions of the same name method are available.

In order to handle the method overloading feature, we manipulate parameters in method declarations and arguments in method invocation expressions. The CRT operator contributes to examining this feature as it changes the types of method parameters in method declarations. We also propose the POC, VMR, AOC, and AND operators for the method overloading feature.

POC (method Parameter Order Change)

The POC operator changes the order of parameters in method declarations if the method has more than one parameter. For example, the Log Message class in our case study has five overloading constructors, and two of them are:

Original Code:
```
1. public LogMessage(int level, String logKey, Object[]
inserts) {…}
2. public LogMessage(int level, String logKey, Object
insert) {…}
```

POC mutant:
```
public LogMessage(String logKey, int level, Object[]
inserts) {…}
```

AOC (Argument Order Change)

The AOC operator changes the order of arguments in method invocation expressions , if there is more than one argument.

Original code:
```
Trace.entry("Logger", "addLogCatalogue");
```

AOC Mutant:
```
Trace.entry ("addLogCatalogue", "Logger");
```

2.5.5 Exceptional Handling

Catch Clauses Deletion (CCD)

In some case, more than one exception could be raised by a single piece of code. To handle this type of situation, two or more catch clauses can specify, each catch block can catch a different type of exception.

The CCD removes catch clauses one by one when there is more than one catch clauses(no effect whether or not the finally clause exist in the code). CCD working is given by the following codes:-

Original code:
```
1. class ccd {
2. try {……}
3. catch (ArithmeticException e){……}
4. catch (ArrrayIndexOutOfBoundsException  e){……}
5. catch (ArrayStoreException e){……}
6. catch (IllegalArgumentException e){…} }
```

CCD mutant:
```
1. class ccd{
2. try{……}
3. catch (ArithmeticException e){……}
4.catch(ArrrayIndexOutOfBoundsException e){……}
5. catch(ArrayStoreException e) {....... .}
```

Throw Statement Deletion (TSD)

TSD operator deletes the throw statement that is shown by the following codes with the different outputs –

Original code:

```
1. class tsd{
2. try{
3. System.out.println("1st time exception throw");
4. Throw new NullPointerException("demo"); }
5. catch (NullPointerExceptione){
6. System.out.println("exception throw");
7. throw e; } } }
```

TSD mutant:

```
1. class tsd{
2. try{
3. System.out.println("1st time exception  throw"); }
4. catch(NullPointerException e){
5. System.out.println("exception throw");
6. throw e; } } }
```

Finally Clause Deletion (FCD)

The FCD operator delete this finally clause and produce an output difference.

Original code:

```
1. class fcd{
2. try{…….
3. ……{catch(……){
4. ……}catch(……){…}
5. ...}finally{……}
```

FCD mutants:

```
1. class fcd{
2. try{…….
3. ……{catch(……..){
4. ……}catch(……){…}
```

2.5.6 String Handling Operators

Value Change Operators (VCO):

String Methods

charAt()

setCharAt()

getChars()

substring()

replace()

deleteCharAt()

Original code:	**Mutants:**
`s.charAt(4);`	`s.charAt(5);`
`s.setCharAt(6);`	`s.setCharAt(7);`
`getChars(10,14,str,0);`	`getChars(11,15,str,1);`
`substring(5,6);`	`substring(6,7);`
`replace(5,7,"new");`	`replace(6,8,"new");`
`deleteCharAt(3);`	`deleteCharAt(4);`

2.6 EXISTING MUTATION TESTING TOOLS

Since mutation testing was proposed in 1987, a number of mutation testing tools have developed in academic world. Mothra is one of the most widely known mutation testing systems for FORTRAN in the early historical development of mutation testing. It was the first tool that implemented mutation analysis as a complete software testing environment. Following the introduction of mutation operators for the programming language C, the first mutation testing tool for C program was developed, called Proteum. After that as Java became more popular, many mutation tools for java were developed. One of the most widely known one is MuJava, which not only supports traditional mutation operators but also provides class-level mutation operators.

In recent years, a number of open source mutation tools have also been implemented for many programming languages. For example, PesTer is a mutation testing tool for Python and PyUnit tests. Nester is a mutation tool for C# code. SQLMutation is a mutation testing tool for database queries. Then SourceForge has an open source mutation tool for java called Jester. However, the efficiency of mutation testing depends largely on the mutation operators and the mutation operators that Jester uses have proven to be rather unstable.

2.6.1 muJava

µJava (***muJava***) is a mutation testing tool for Java programs. It automatically generates mutants for both traditional mutation testing and class-level mutation testing. ***µJava*** can test individual classes as well as packages of multiple classes. Tests are supplied by the users as sequences of method calls to the classes under test encapsulated in methods in separate classes.

µJava is the result of a collaboration between two universities, Korea Advanced Institute of Science and Technology (KAIST) in S. Korea and George Mason University in the USA. The research collaborators are Yu Seung Ma, PhD candidate at KAIST in Korea, Dr. Yong Rae

Kwon, Professor at KAIST in Korea, and Dr. Jeff Offutt, Professor at George Mason University in the USA. Most of the software development was done by YuSeung.

μJava uses two types of mutation operators, class level and method level. The class level mutation operators were designed for Java classes by Ma, Kwon and Offutt, and were in turn designed from a categorization of object-oriented faults by Offutt, Alexander. ***μJava*** creates object-oriented mutants for Java classes according to 24 operators that are specialized to object-oriented faults. Method level (traditional) mutants are based on the selective operator set by Offutt. After creating mutants, ***μJava*** allows the tester to enter and run tests, and evaluates the mutation coverage of the tests.

2.6.2 Jester

Jester offers a way to the default set of mutation operations, but problems concerning performance and reliability of the tool, as well as a limited range of possible mutation operators (based only on string substitution) remain. Moreover, the mutation operators offered by Jester are nor context-aware, and often lead to broken code. It is worth mentioning that Jester's approach is to generate, compile and run unit tests against a mutant. The process repeats for every mutant of the SUT and, thus is inefficient. Because of these major disadvantages, Jester was not a successful tool.

2.6.3 Jumble

Jumble is a class level mutation testing tool that works in conjunction with JUnit. The purpose of mutation testing is to measure of the adequacy of test cases. A single mutation is performed on the code under consideration; the corresponding test cases are then executed. If the modified code fails the tests, then this increases confidence in the tests. Conversely, if the modified code passes the tests this indicates a testing deficiency.

Jumble was developed in 2003-2006 by a commercial company in New Zealand, Reel Two (www.reeltwo.com), and is now available as open source under the GPL licence.

JUnit has become the *de facto* unit testing framework for the Java language. A class and its corresponding JUnit test is a sensible granularity at which to apply mutation testing. With Java it is feasible to perform mutation testing either at the source code or byte-code level. Jester is a mutation testing tool which operates at the source code level. While Jester proves

useful, it is hampered by the costly cycle of modifying the source, compiling the source, and running the tests.

Jumble is a new mutation tester operating directly on class files. It uses the byte-code engineering library (BCEL) to directly modify class files thereby drastically cutting the time taken for each mutation test cycle.

Jumble has been designed to operate in an industrial setting with large projects. Heuristics have been included to speed the checking of mutations, for example, noting which test fails for each mutation and running this first in subsequent mutation checks. Significant effort has been put into ensuring that it can test code which runs in environments such as the Apache webserver. This requires careful attention to class path handling and co-existence with foreign class-loaders.

At ReelTwo, Jumble is used on a continuous basis within an agile programming environment with approximately 400,000 lines of Java code under source control. This checks out project code every fifteen minutes and runs an incremental set of unit tests and mutation tests for modified classes.

2.6.4 Proteum

Proteum is the first tool to support the testing of C programs based on mutation testing at the unit level. With the proposition of the criterion Interface mutation, that uses a set of mutant operators developed to model integration errors, the Proteum/IM has been developed. At the integration level, Interface mutation is also effective in detecting faults. Recently, Proteum and Proteum/IM have been integrated in a testing environment, named Proteum/IM 2.0. In this way, the tester can use the same concept during the unit and the integration testing phases.

The *Proteum* family is composed of the following tools:

Proteum: supports the unit testing of C programs. It has 71 operators, categorized into four mutation classes: Statement(15), Operator(46), Variable(7) and constant(3).

Proteum/IM: supports the integration testing of C programs based on the Interface Mutation criterion. It has 33 operators divided into two groups: 24 of them Group I, and 9 of Group II.

It provides mechanisms for the assessment of test case adequacy for testing the interactions among the units of a given program.

Proteum/IM 2.0: is an evolution of Proteum and Proteum/IM. It is a single, integrated environment that provides facilities to investigate low-cost and incremental testing strategies based on mutation.

Proteum/FSM: supports the application of mutation testing to validate Finite State Machine based specifications. It has 9 mutant operators. These operators are based on the error classes defined by Chow and on heuristics about typical errors made by designers during the creation of Finite State Machines.

Proteum/ST: supports the application of mutation testing to validate statecharts based specifications. Statecharts are an extension to Finite State Machines. The approach taken to implement Proteum/FSM makes it easier to extend the ideas, concepts and tools to Statecharts considering hierarchy, concurrency, history and other statechart features. Proteum/ST is divided into three categories: 9 Finite State Machine operators; 11 Extended FSM(EFSM) operators; and 17 Statecharts-feature based operators.

2.6.5 Judy

Judy is an implementation of the FAMTA Light approach developed in Java with AspectJ extensions. The core features of Judy are high mutation testing process performance, advanced mutant generation mechanism, integration with professional development environment tools, full automation of mutation testing process and support for the latest version of Java, enabling it to run mutation testing against the most recent Java software systems or components.

Judy, like MuJava, supports traditional mutation operators. These were initially defined for procedural programs and have been identified by Offutt as selective mutation operators. These operators are to minimize the number of mutation operators, whilst maximizing testing strength. The latter was measured by Offutt and Lee by computing the non selective mutation scores of the test sets that were 100% adequate for selective mutation.

Table 2.1 A list of mutation Testing Tools

Language	Tool	Year
Fortran	PIMS	1976
	FMS	1978
	PMS	1978
	EXPER	1978
	Mothra	1988
COBOL	CMS.1	1980
C	Proteum	1993
	PMothra	1993
	CMothra	1989
	Proteum/IM	2000
C#	Nester	2001
Java	Jester	2001
	µJava	2002
	Lava	2005
Python	Pester	2001

.7 Mutation Operators

Mutation operators are designed to model simple programming mistakes that programmers make. Faults in the programs could be much more complex than the simple mistakes modelled by a mutation operator. However, it has been found that, despite the simplicity of mutations, complex faults are discovered while trying to distinguish mutants from their parent. We apply one or more mutation operators to P to generate a variety of mutants. A mutation operator might generate no mutants or one or more mutants. The input statement and declarations are not mutated at all.

While it is possible to categorize mutation operators into a few generic categories, the operators themselves are dependent on the syntax of the programming language. For example, for a program written in ANSI C, one needs to use mutation operators for C. A Java program is mutated using mutation operators designed for the java language.

There are at least three reasons for the dependence of mutation operators on language syntax. First, given that the program being mutated is syntactically correct, a mutation operator must produce a mutant that is also syntactically correct. To do so requires that a valid syntactic construct be mapped to another valid syntactic construct in the same language.

Second, the domain of a mutation operator is determined by the syntax rules of a programming language. For example, in java, the domain of a mutation operator that replaces one relational operator by another is {<,<=,>,>=,!=,==}.

Third, peculiarities of language syntax have an effect on the kind of mistakes that a programmer could make. The aspects of a language such as procedural versus object oriented are captured in the language syntax.

Mujava uses two types of mutation operators. The traditional mutation operators are developed from procedural languages. Object Oriented languages have additional class level mutation operators. They work on the features of object oriented languages like inheritance, polymorphism and dynamic binding.

Mothra uses 22 traditional mutation operators on Fortran. However, running all these mutant operates generate a huge number of mutants and not all of them are effective because of overlaps. The idea of selective mutation was introduced by Wong and Mathur and later experimentally validated by Offutt. Selective mutation states that a subset of all the mutation operators is sufficient to provide same effectiveness as non-selective mutation.

24 class mutation operators were identified for Java classes by Ma, Kwon and Offutt for testing object-oriented and integration issues. There is yet any research on applying selective mutation on these operators. A major issue with class mutation operators is that they are applicable in different levels – intra-method, inter-method, intra-class and inter-class. Traditional mutation operators are all intra-method operators. In general the class mutation operators are intra-class, but inter-class operators are important for traditional integration testing and seldom used subsystem testing.

3. PROPOSED FRAMEWORK OF TOOL

Now that we know what mutation is and what mutants look like, let us understand how mutation is used for assessing the adequacy of test set. The problem of test assessment using mutation can be stated as follows:

Let P be a program under test, T be a test set for P, and R the set of requirements that P must meet. Suppose that P has been tested against all tests in T and found to be correct with respect to R on each test case. We want to know "How good is T?"

Mutation offers a way of answering the question stated above. A quantitative assessment of the goodness of T is obtained by computing a mutation score of T. Mutation score is a number between 0 and 1. A score of 1 means that T is adequate with respect to mutation. A score lower than 1 means that T is inadequate with respect to mutation. An inadequate test set can be enhanced by the addition of test cases that increase the mutation score.

3.1 Software Engineering Approach

We followed SDLC (System Development Life Cycle) for the various phases of project development.

3.1.1 System Development Life Cycle

The Systems Development Life Cycle (SDLC), or Software Development Life Cycle in systems engineering and software engineering, is the process used by the developers to develop software with the help of certain models defined in software engineering.

Software Development Life Cycle (SDLC) is a process used by a software developer to develop an information system for the software under consideration, including its requirements, designing, coding, testing and maintainability. Any SDLC should result in a high quality software that not just meets its customer expectations but also reaches completion within the given time frame and whose cost estimates works effectively and efficiently in the current Information Technology infrastructure, and is inexpensive to maintain and cost-effective to enhance.

Computer systems are complex and often link multiple traditional systems potentially supplied by different software vendors. To manage this level of complexity, a number of SDLC models have been created: waterfall, spiral, build and fix, incremental, and rapid prototyping.

The approach which we followed for the development of the tool is **Incremental development**. The incremental development model is a method of software

development where the model is designed, implemented and tested incrementally (a little more is added each time) until the product is fully developed as per the stated requirements. It involves both development and maintenance. This model combines the elements of the waterfall model with the iterative philosophy of prototyping. In this model, there is an overall lower risk of project failure.

The tool is decomposed into a number of components, each of which is designed and implemented separately. Firstly, the mutation operators are being defined for the tool. After that some mutants are generated of a source code. Then the source code and all the mutants are being run on the test cases to check their adequacy. Next, some more operators are added to the tool and thereafter it generates more mutants of the original program. In this way the tool is being developed and the incremental process model helps in adding more and more functionality at each stage of the product development.

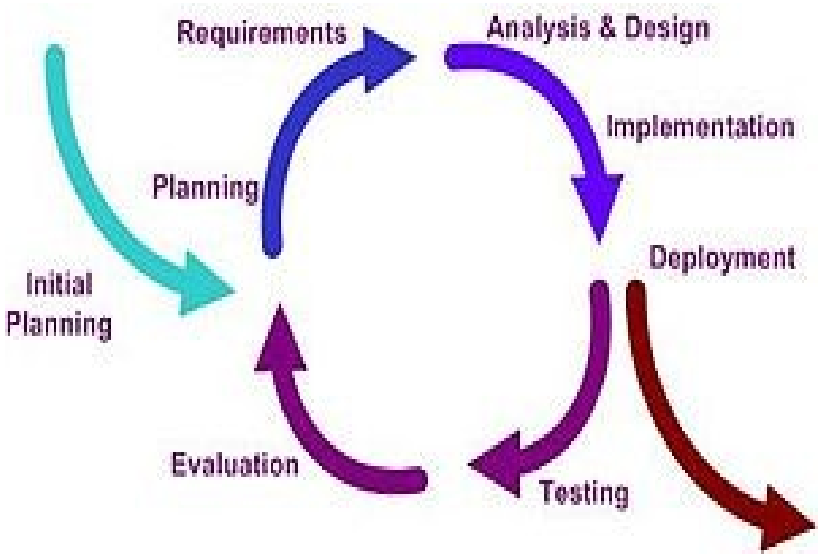

Fig 3.1 An incremental development model

3.2 Project Process

3.2.1 Requirement Phase

This is the starting phase of the project development and consisted of following:

Literature Survey and Research: In this phase, I learned about various aspects of mutation testing. Also learned about mutation operators and how they are applied in practise. I came to know about the traditional mutation operators and class level mutation operators. Also, I learned about some existing mutation testing tools like mujava, how they are run, how they differ from each other and what different functionalities that they perform which make them different from the one another.

Analysis: After going through a lot of research and literature survey, I analysed that the **Eclipse version 3.5 (Java platform)** would be a good platform to develop the tool. Eclipse is a multi-language software development environment comprising an integrated development environment and an extensible plug-in system. Also, I analysed that Java would be the language for which the tool will work initially. After that other object oriented languages will also be included in the tool.

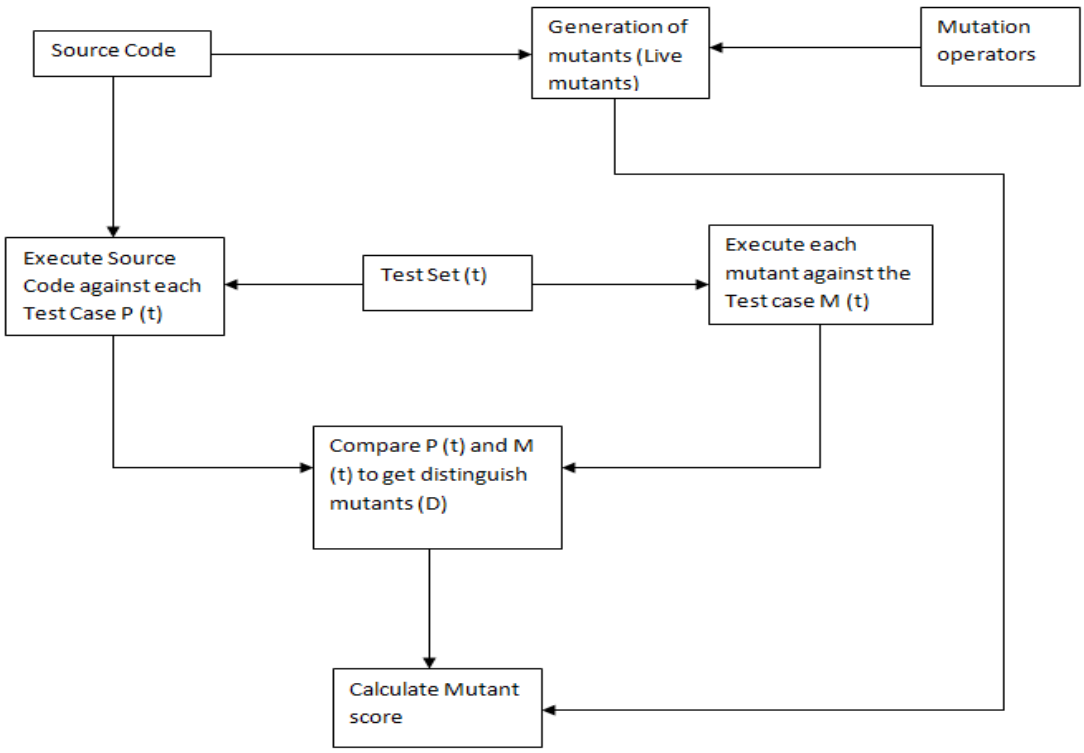

Fig 3.2 Framework of the tool

3.2.2 Design Phase

This phase is the most important phase of the project development:

Framework and Implementation

In this phase, the framework of the tool is modeled. Also It was decided how the various components of the tool will be interconnected to meet the specified requirements. The framework of the tool is depicted in Fig 3.2. The various steps of implementing the tool is given below:

Step 1 Program execution

The first step in assessing the adequacy of a test set T with respect to program P and requirements R is to execute P against each test case in T. Let P(t) denote the observed behaviour of P when executed against t. Generally, the observed behaviour is expressed as a set of values of output variables in P. However, it might also relate to the performance of P.

Step 2 Mutant Generation

The next step in test-adequacy assessment is the generation of mutants. The mutants are generated with the help of Mutation Operators that are defined in the tool. A mutant can be generated from P by altering the arithmetic operators such that any occurrence of the addition operator (+) is replaced by the subtraction operator (-). By mutating the program as mentioned above, we obtain various mutants of the program which are known as Live Mutants(L). These mutants are live because we have not yet distinguished them from the original program. Distinguishing a mutant from its parent is also known as killing a mutant.

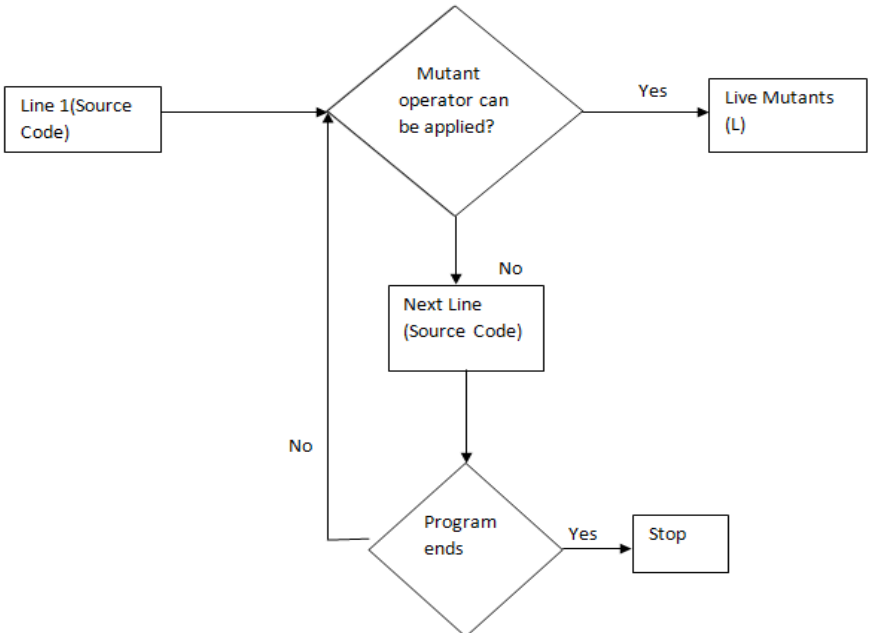

Fig 3.3 Generation of Mutants

Step 3 Select the next mutant

In this step, we select the next mutant to be considered. This mutant must not be selected earlier. At this point a loop is starting that will cycle through all mutants in L until each mutant has been selected. If there are live mutants in L, which have never been selected in any previous step, then a mutant is selected arbitrarily. The selected mutant is removed from L.

Step 4 Mutant execution

Having selected a mutant M, we know attempt to find whether at least one of the tests in T can distinguish it from its parent P. To do so, we need to execute M against tests in T. Thus at this point we enter another loop that is executed for each selected mutant. The loop terminates when all tests are exhausted or M is distinguished by some test, whichever happens earlier. We have selected a mutant M for execution against a test set t. Then we execute M against t and check if the output generated by executing M against t is same or different from that generated by executing P against t.

Step 5 Distinguished and Live Mutants

After executing all mutants, we check for the Live and Distinguished mutants. When none of the tests in T is able to distinguish mutant M from its parent P, then M is placed back into the set of Live Mutants and when any one of the test case in T is able to distinguish the mutant M from its parent P, then that mutant is called Killed or Distinguished Mutant.

Step 6 Computation of mutation Score

This is the final step in the assessment of test adequacy. Mutant score can be computed as the total number of Distinguished Mutants by the total number of Live Mutant. As evident from the formula above, a mutation score is always between 0 and 1. If a test set T distinguishes all mutants, then the mutation score is 1. If T does not distinguish any mutant, then the mutant score is 0. The score of 0 does not imply that the test set is inadequate. In this case, the set of mutants generated is insufficient to assess the adequacy of the test set. In practise, it is rare to find such a situation.

3.3 The Problem of Mutation Analysis

Although Mutation Testing is able to effectively assess the quality of a test set, it still suffers from a number of problems. One problem that prevents Mutation Testing from becoming a practical testing is the high computational cost of executing the enormous number of mutants against a test set. The other problem related to the amount of human effort involved in using Mutation Testing is the human oracle problem. The human oracle problem refers to the process of checking the original program's output with each test case. Strictly speaking this is not a problem unique to Mutation Testing. In all forms of testing, once a set of inputs has been arrived at, there remains the problem of checking output. However, mutation testing is effective precisely because it is demanding and this can lead to an increase in the number of test cases thereby increasing the oracle cost. This oracle cost is often the most expensive part of the overall testing activity.

4. Results

Original Program:

```
public class kk
{
public static void main(String args[])
{
int a=0;
int u=10;
int l=a+u;
int c=a-u;
int d=a*u;
int f=a/u;
if(a<u)
System.out.println("a is less than u");
System.out.println(l);
System.out.println(c);
System.out.println(d);
System.out.println(f);
}
}
```

Mutated code1

```
public class kk
{
public static void main(String args[])
{
int a=0;
```

```
int u=10;
int l=a-u;
int c=a-u;
int d=a*u;
int f=a/u;
if(a<u)
System.out.println("a is less than u");
System.out.println(l);
System.out.println(c);
System.out.println(d);
System.out.println(f);
}
}
```

Mutated code2

```
public class kk
{
public static void main(String args[])
{
int a=0;
int u=10;
int l=a*u;
int c=a-u;
int d=a*u;
int f=a/u;
if(a<u)
System.out.println("a is less than u");
System.out.println(l);
System.out.println(c);
System.out.println(d);
System.out.println(f);
}
}
```

Mutated code3

```
public class kk
{
public static void main(String args[])
{
int a=0;
int u=10;
int l=a/u;
int c=a-u;
int d=a*u;
int f=a/u;
if(a<u)
System.out.println("a is less than u");
System.out.println(l);
System.out.println(c);
System.out.println(d);
System.out.println(f);
}
}
```

Mutated code 4

```
public class kk
{
public static void main(String args[])
{
int a=0;
int u=10;
int l=a+u;
int c=a/u;
int d=a*u;
int f=a/u;
if(a<u)
System.out.println("a is less than u");
System.out.println(l);
```

```
System.out.println(c);
System.out.println(d);
System.out.println(f)
}
}
```

Mutated code 5

```
public class kk
{
public static void main(String args[])
{
int a=0;
int u=10;
int l=a+u;
int c=a+u;
int d=a*u;
int f=a/u;
if(a<u)
System.out.println("a is less than u");
System.out.println(l);
System.out.println(c);
System.out.println(d);
System.out.println(f);
}
}
```

Mutated code 6

```
public class kk
{
public static void main(String args[])
{
int a=0;
int u=10;
int l=a+u;
int c=a*u;
int d=a*u;
int f=a/u;
if(a<u)
System.out.println("a is less than u");
System.out.println(l);
System.out.println(c);
System.out.println(d);
System.out.println(f);
}
}
```

Mutated code 7

```
public class kk
{
public static void main(String args[])
{
int a=0;
int u=10;
int l=a+u;
int c=a-u;
int d=a/u;
int f=a/u;
if(a<u)
System.out.println("a is less than u");
System.out.println(l);
System.out.println(c);
System.out.println(d);
System.out.println(f);
}
}
```

Mutated code 8

```
public class kk
```

```
{
public static void main(String args[])
{
int a=0;
int u=10;
int l=a+u;
int c=a-u;
int d=a+u;
int f=a/u;
if(a<u)
System.out.println("a is less than u");
System.out.println(l);
System.out.println(c);
System.out.println(d);
System.out.println(f);
}
}
```

Mutated code 9

```
public class kk
{
public static void main(String args[])
{
int a=0;
int u=10;
int l=a+u;
int c=a-u;
int d=a-u;
int f=a/u;
if(a<u)
System.out.println("a is less than u");
System.out.println(l);
System.out.println(c);
System.out.println(d);
System.out.println(f);
}
}
```

Mutated code 10

```
public class kk
{
public static void main(String args[])
{
int a=0;
int u=10;
int l=a+u;
int c=a-u;
int d=a*u;
int f=a+u;
if(a<u)
System.out.println("a is less than u");
System.out.println(l);
System.out.println(c);
System.out.println(d);
System.out.println(f);
}
}
```

Mutated code 11

```
public class kk
{
public static void main(String args[])
{
int a=0;
int u=10;
int l=a+u;
int c=a-u;
int d=a*u;
```

```
int f=a-u;
if(a<u)
System.out.println("a is less than u");
System.out.println(l);
System.out.println(c);
System.out.println(d);
System.out.println(f);
}
}
```

Mutated code 12

```
public class kk
{
public static void main(String args[])
{
int a=0;
int u=10;
int l=a+u;
int c=a-u;
int d=a*u;
int f=a*u;
if(a<u)
System.out.println("a is less than u");
System.out.println(l);
System.out.println(c);
System.out.println(d);
System.out.println(f);
}
}
```

Mutated code 13

```
public class kk
{
public static void main(String args[])
{
int a=0;
int u=10;
int l=a+u;
int c=a-u;
int d=a*u;
int f=a/u;
if(a>u)
System.out.println("a is less than u");
System.out.println(l);
System.out.println(c);
System.out.println(d);
System.out.println(f);
}
}
```

CONCLUSION

Mutation testing is not meant as a replacement for code coverage, but as a complementary approach that is useful in detecting those pieces of the code that are executed by running tests, but are not fully tested. It is not widely used in software engineering due to the limited performance of the existing tools and lack of support for standard unit testing and build tools.

My target of the project is to develop a feasible mutation testing tool with minimal human involvement and significant performance improvement. The tool would provide almost complete automation to the tester. Good coverage is an important criterion and efficient mutation testing provides significantly better coverage than other techniques.

APPENDIX

CODING

```
import java.io.*;
public class kk
{
public static void main(String args[]) throws IOException
{
File j =  new
File("C:/Users/PARIMALROYCHAUDHURY/workspace/test/src/m.java");
File m1 = new File("C:/Users/PARIMAL
ROYCHAUDHURY/workspace/test/src/m1.java");
File m2 =new File("C:/Users/PARIMAL
ROYCHAUDHURY/workspace/test/src/m2.java");
File m3 =new File("C:/Users/PARIMAL
ROYCHAUDHURY/workspace/test/src/m3.java");
File m4 =new File("C:/Users/PARIMAL
ROYCHAUDHURY/workspace/test/src/m4.java");
File m5 =new File("C:/Users/PARIMAL
ROYCHAUDHURY/workspace/test/src/m5.java");
File m6 =new File("C:/Users/PARIMAL
ROYCHAUDHURY/workspace/test/src/m6.java");
```

```
File m7 =new File("C:/Users/PARIMAL
ROYCHAUDHURY/workspace/test/src/m7.java");
File m8 =new File("C:/Users/PARIMAL
ROYCHAUDHURY/workspace/test/src/m8.java");
File m9 =new File("C:/Users/PARIMAL
ROYCHAUDHURY/workspace/test/src/m9.java");
File m10=new File("C:/Users/PARIMAL
ROYCHAUDHURY/workspace/test/src/m10.java");
File m11=new File("C:/Users/PARIMAL
ROYCHAUDHURY/workspace/test/src/m11.java");
File m12=new File("C:/Users/PARIMAL
ROYCHAUDHURY/workspace/test/src/m12.java");
File m13=new File("C:/Users/PARIMAL
ROYCHAUDHURY/workspace/test/src/m13.java");
m1.createNewFile();
m2.createNewFile();
m3.createNewFile();
m4.createNewFile();
m5.createNewFile();
m6.createNewFile();
m7.createNewFile();
m8.createNewFile();
m9.createNewFile();
m10.createNewFile();
m11.createNewFile();
m12.createNewFile();
m13.createNewFile();
if(m1.isFile())
{
System.out.println("file1 exists");
}
if(m2.isFile())
{
System.out.println("file2 exists");
}
if(m3.isFile())
{
System.out.println("file3 exists");
}
if(m4.isFile())
{
System.out.println("file4 exists");
}
if(m5.isFile())
{
System.out.println("file5 exists");
}
if(m6.isFile())
{
System.out.println("file6 exists");
}
if(m7.isFile())
{
System.out.println("file7 exists");
}
if(m8.isFile())
{
System.out.println("file8 exists");
}
if(m9.isFile())
{
System.out.println("file9 exists");
}
if(m10.isFile())
{
System.out.println("file10 exists");
}
if(m11.isFile())
```

```
{
System.out.println("file11 exists");
}
if(m12.isFile())
{
System.out.println("file12 exists");
}
if(m13.isFile())
{
System.out.println("file13 exists");
}
BufferedReader br1=new BufferedReader(new FileReader(j));
BufferedWriter ty1=new BufferedWriter(new FileWriter(m1));
String line;
String l=null;
while((line=br1.readLine())!=null)
{
l=line.replace("class m", "class m1");
line = l;
l=line.replace('+', '-');
ty1.write(l);
}
ty1.flush();
ty1.close();
BufferedReader br2=new BufferedReader(new FileReader(j));
BufferedWriter ty2=new BufferedWriter(new FileWriter(m2));
l=null;
while((line=br2.readLine())!=null)
{
l=line.replace("class m", "class m2");
line = l;
l=line.replace('+', '*');
ty2.write(l);
}
ty2.flush();
ty2.close();
BufferedReader br3=new BufferedReader(new FileReader(j));
BufferedWriter ty3=new BufferedWriter(new FileWriter(m3));
l=null;
while((line=br3.readLine())!=null)
{
l=line.replace("class m", "class m3");
line = l;
l=line.replace('+', '/');
ty3.write(l);
}
ty3.flush();
ty3.close();
BufferedReader br4=new BufferedReader(new FileReader(j));
BufferedWriter ty4=new BufferedWriter(new FileWriter(m4));
l=null;
while((line=br4.readLine())!=null)
{
l=line.replace("class m", "class m4");
line = l;
l=line.replace('-', '/');
ty4.write(l);
}
ty4.flush();
ty4.close();
BufferedReader br5=new BufferedReader(new FileReader(j));
BufferedWriter ty5=new BufferedWriter(new FileWriter(m5));
l=null;
while((line=br5.readLine())!=null)
{
l=line.replace("class m", "class m5");
line = l;
l=line.replace('-', '+');
```

```
ty5.write(l);
}
ty5.flush();
ty5.close();
BufferedReader br6=new BufferedReader(new FileReader(j));
BufferedWriter ty6=new BufferedWriter(new FileWriter(m6));
l=null;
while((line=br6.readLine())!=null)
{
l=line.replace("class m", "class m6");
line = l;
l=line.replace('-', '*');
ty6.write(l);
}
ty6.flush();
ty6.close();
BufferedReader br7=new BufferedReader(new FileReader(j));
BufferedWriter ty7=new BufferedWriter(new FileWriter(m7));
l=null;
while((line=br7.readLine())!=null)
{
l=line.replace("class m", "class m7");
line = l;
l=line.replace('*', '/');
ty7.write(l);
}
ty7.flush();
ty7.close();
BufferedReader br8=new BufferedReader(new FileReader(j));
BufferedWriter ty8=new BufferedWriter(new FileWriter(m8));
l=null;
while((line=br8.readLine())!=null)
{
l=line.replace("class m", "class m8");
line = l;
l=line.replace('*', '+');
ty8.write(l);
}
ty8.flush();
ty8.close();
BufferedReader br9=new BufferedReader(new FileReader(j));
BufferedWriter ty9=new BufferedWriter(new FileWriter(m9));
l=null;
while((line=br9.readLine())!=null)
{
l=line.replace("class m", "class m9");
line = l;
l=line.replace('*', '-');
ty9.write(l);
}
ty9.flush();
ty9.close();
BufferedReader br10=new BufferedReader(new FileReader(j));
BufferedWriter ty10=new BufferedWriter(new FileWriter(m10));
l=null;
while((line=br10.readLine())!=null)
{
l=line.replace("class m", "class m10");
line = l;
l=line.replace('/', '+');
ty10.write(l);
}
ty10.flush();
ty10.close();
BufferedReader br11=new BufferedReader(new FileReader(j));
BufferedWriter ty11=new BufferedWriter(new FileWriter(m11));
l=null;
while((line=br11.readLine())!=null)
```

```
{
l=line.replace("class m", "class m11");
line = l;
l=line.replace('/', '-');
ty11.write(l);
}
ty11.flush();
ty11.close();
BufferedReader br12=new BufferedReader(new FileReader(j));
BufferedWriter ty12=new BufferedWriter(new FileWriter(m12));
l=null;
while((line=br12.readLine())!=null)
{
l=line.replace("class m", "class m12");
line = l;
l=line.replace('/', '*');
ty12.write(l);
}
ty12.flush();
ty12.close();
BufferedReader br13=new BufferedReader(new FileReader(j));
BufferedWriter ty13=new BufferedWriter(new FileWriter(m13));
l=null;
while((line=br13.readLine())!=null)
{
l=line.replace("class m", "class m13");
line = l;
l=line.replace('<', '>');
ty13.write(l);
}
ty13.flush();
ty13.close();
}
}

import java.io.*;
public class as {
public static void main(String[] args)throws IOException
{
int dm=0,flag=-1,lm=13;
BufferedReader in1  = new BufferedReader(new FileReader("C:/Users/PARIMAL
ROYCHAUDHURY/workspace/test/src/out.txt"));
BufferedReader mn1  = new BufferedReader(new FileReader("C:/Users/PARIMAL
ROYCHAUDHURY/workspace/test/src/out1.txt"));
String lineFromInput  = null ;
String lineFromMutant = null;
while((lineFromInput = in1.readLine())!=null&&(lineFromMutant =
mn1.readLine())!=null)
{
if(lineFromInput.equals(lineFromMutant))
{
flag=1;
 }
else
{
flag=0;
break;
}
}
      if(flag==0)
{
 dm++;
 }
BufferedReader in2  = new BufferedReader(new FileReader("C:/Users/PARIMAL
ROYCHAUDHURY/workspace/test/src/out.txt"));
BufferedReader mn2  = new BufferedReader(new FileReader("C:/Users/PARIMAL
ROYCHAUDHURY/workspace/test/src/out2.txt"));
lineFromInput = null ;
```

```
lineFromMutant = null;
while((lineFromInput = in2.readLine())!=null&&(lineFromMutant =
mn2.readLine())!=null)
{
if(lineFromInput.equals(lineFromMutant))
{
flag=1;
}
else
      {
flag=0;
break;
}
}
if(flag==0)
{
dm++;

}
BufferedReader in3 = new BufferedReader(new FileReader("C:/Users/PARIMAL
ROYCHAUDHURY/workspace/test/src/out.txt"));
BufferedReader mn3 = new BufferedReader(new FileReader("C:/Users/PARIMAL
ROYCHAUDHURY/workspace/test/src/out3.txt"));
lineFromInput = null ;
lineFromMutant = null;
while((lineFromInput = in3.readLine())!=null&&(lineFromMutant =
mn3.readLine())!=null)
{
if(lineFromInput.equals(lineFromMutant))
{
flag=1;
}
else
{
flag=0;
break;
}
}
if(flag==0)
     {
dm++;
}
BufferedReader in4 = new BufferedReader(new FileReader("C:/Users/PARIMAL
ROYCHAUDHURY/workspace/test/src/out.txt"));
BufferedReader mn4 = new BufferedReader(new FileReader("C:/Users/PARIMAL
ROYCHAUDHURY/workspace/test/src/out4.txt"));
lineFromInput  = null ;
lineFromMutant = null;
while((lineFromInput = in4.readLine())!=null&&(lineFromMutant =
mn4.readLine())!=null)
{
if(lineFromInput.equals(lineFromMutant))
{
flag=1;
}
else
      {
 flag=0;
break;
}
}
if(flag==0)
{
dm++;
}
BufferedReader in5 = new BufferedReader(new FileReader("C:/Users/PARIMAL
ROYCHAUDHURY/workspace/test/src/out.txt"));
```

```
BufferedReader mn5 = new BufferedReader(new FileReader("C:/Users/PARIMAL
ROYCHAUDHURY/workspace/test/src/out5.txt"));
lineFromInput = null ;
lineFromMutant = null;

while((lineFromInput = in5.readLine())!=null&&(lineFromMutant =
mn5.readLine())!=null)
{
if(lineFromInput.equals(lineFromMutant))
{
flag=1;
}
else
{
flag=0;
break;
}
}
if(flag==0)
{
dm++;
}
BufferedReader in6 = new BufferedReader(new FileReader("C:/Users/PARIMAL
ROYCHAUDHURY/workspace/test/src/out.txt"));
BufferedReader mn6 = new BufferedReader(new FileReader("C:/Users/PARIMAL
ROYCHAUDHURY/workspace/test/src/out6.txt"));
lineFromInput  = null ;
lineFromMutant = null;
while((lineFromInput = in6.readLine())!=null&&(lineFromMutant =
mn6.readLine())!=null)
{

if(lineFromInput.equals(lineFromMutant))
{
flag=1;
}
else
{
flag=0;
break;
}
}
if(flag==0)
{
dm++;
}
BufferedReader in7 = new BufferedReader(new FileReader("C:/Users/PARIMAL
ROYCHAUDHURY/workspace/test/src/out.txt"));
BufferedReader mn7 = new BufferedReader(new FileReader("C:/Users/PARIMAL
ROYCHAUDHURY/workspace/test/src/out7.txt"));
lineFromInput = null ;
lineFromMutant = null;
while((lineFromInput = in7.readLine())!=null&&(lineFromMutant =
mn7.readLine())!=null)
{
if(lineFromInput.equals(lineFromMutant))
{
flag=1;
}
else
{
flag=0;
 break;
}
}
if(flag==0)
{
```

```
dm++;
 }
BufferedReader in8 = new BufferedReader(new FileReader("C:/Users/PARIMAL
ROYCHAUDHURY/workspace/test/src/out.txt"));
BufferedReader mn8 = new BufferedReader(new FileReader("C:/Users/PARIMAL
ROYCHAUDHURY/workspace/test/src/out8.txt"));
lineFromInput  = null ;
lineFromMutant = null;
while((lineFromInput = in8.readLine())!=null&&(lineFromMutant =
mn8.readLine())!=null)
{
if(lineFromInput.equals(lineFromMutant))
{
flag=1;
}
else
{
flag=0;
break;
}
}
if(flag==0)
{
dm++;
}
BufferedReader in9 = new BufferedReader(new FileReader("C:/Users/PARIMAL
ROYCHAUDHURY/workspace/test/src/out.txt"));
BufferedReader mn9 = new BufferedReader(new FileReader("C:/Users/PARIMAL
ROYCHAUDHURY/workspace/test/src/out9.txt"));
lineFromInput  = null ;
lineFromMutant = null;
while((lineFromInput = in9.readLine())!=null&&(lineFromMutant =
mn9.readLine())!=null
{
 if(lineFromInput.equals(lineFromMutant))
{
flag=1;
}
else
{
flag=0;
 break;
}
}
if(flag==0)
{
dm++;
      }
BufferedReader in10 = new BufferedReader(new FileReader("C:/Users/PARIMAL
ROYCHAUDHURY/workspace/test/src/out.txt"));
BufferedReader mn10 = new BufferedReader(new FileReader("C:/Users/PARIMAL
ROYCHAUDHURY/workspace/test/src/out10.txt"));
lineFromInput = null ;
    lineFromMutant = null;
while((lineFromInput = in10.readLine())!=null&&(lineFromMutant =
mn10.readLine())!=null)
{
if(lineFromInput.equals(lineFromMutant))
{
flag=1;
}
else
{
flag=0;
break;
}
}
if(flag==0)
```

```
{
dm++;
}
BufferedReader in11 = new BufferedReader(new   FileReader("C:/Users/PARIMAL
ROYCHAUDHURY/workspace/test/src/out.txt"));
BufferedReader mn11 = new BufferedReader(new FileReader("C:/Users/PARIMAL
ROYCHAUDHURY/workspace/test/src/out11.txt"));
lineFromInput = null ;
lineFromMutant = null;
while((lineFromInput = in11.readLine())!=null&&(lineFromMutant =
mn11.readLine())!=null)
{
if(lineFromInput.equals(lineFromMutant))
{
flag=1;
}
else
{
flag=0;
break;
}
}
if(flag==0)
{
dm++;
}
BufferedReader in12 = new BufferedReader(new FileReader("C:/Users/PARIMAL
ROYCHAUDHURY/workspace/test/src/out.txt"));
BufferedReader mn12 = new BufferedReader(new FileReader("C:/Users/PARIMAL
ROYCHAUDHURY/workspace/test/src/out12.txt"));
lineFromInput = null ;
lineFromMutant = null;
while((lineFromInput = in12.readLine())!=null&&(lineFromMutant =
mn12.readLine())!=null)
{
if(lineFromInput.equals(lineFromMutant))
{
flag=1;
}
else
{
flag=0;
break;
}
}
if(flag==0)
{
dm++;
}
BufferedReader in13 = new BufferedReader(new FileReader("C:/Users/PARIMAL
ROYCHAUDHURY/workspace/test/src/out.txt"));
BufferedReader mn13 = new BufferedReader(new FileReader("C:/Users/PARIMAL
ROYCHAUDHURY/workspace/test/src/out13.txt"));
lineFromInput = null ;
lineFromMutant = null;
while((lineFromInput = in13.readLine())!=null&&(lineFromMutant =
mn13.readLine())!=null)
{
if(lineFromInput.equals(lineFromMutant))
{
flag=1;
}
else
{
flag=0;
break;
}
}
```

```
if(flag==0)
{
dm++;
}
System.out.println ("No. Of  Live Mutants "+lm);
System.out.println ("No.of  Killed Mutants "+dm);
int MS= (dm*100)/13;
System.out.println("Mutant Score is "+MS+"%");
 }
 }
```

References

1. Aditya P. Mathur, “Foundations of Software Testing,” Pearson, 2008, pp. 1-689.
2. Herbert Schildt, “The Complete Reference Java,” Tata McGraw-Hill, 2006, pp. 1-1024.
3. Kapil Kumar, P.K.Gupta and Roshan Parjapat “New mutants generation for testing java programs,” SPRINGER, Computer Networks and Information Technologies , Vol. 142, No1, pp. 290-294, 2011.
4. Yu-seung Ma and Jeff Offutt “Description of Class Mutation Operators for Java,” 2005.
5. Lech Madeyski and Norbert Radyk “Judy – a mutation testing tool for java,” Institute of Informatics, Wroc law University of Technology, POLAND, pp 1-27.
6. http://www.cs.gmu.edu/~offutt/mujava/
7. http://www.academictutorials.com/testing/introduction.asp/
8. http://jumble.sourceforge.net/