USING INFORMATION RETRIEVAL TO IMPROVE INTEGRATION TESTING

A Dissertation
Submitted to the Graduate Faculty
of the
North Dakota State University
of Agriculture and Applied Science

By

Iyad Alazzam

In Partial Fulfillment
for the Degree of
DOCTOR OF PHILOSOPHY

Major Program:
Software Engineering

April 2012

Fargo, North Dakota

North Dakota State University

Graduate School

Title

USING INFORMATION RETRIEVAL TO IMPROVE INTEGRATION TESTING

By

IYAD ALAZZAM

The Supervisory Committee certifies that this *disquisition* complies with North Dakota State University's regulations and meets the accepted standards for the degree of

DOCTOR OF PHILOSOPHY

SUPERVISORY COMMITTEE:

Dr. Kenneth Magel	
Chair (typed)	Chair (signature)
Dr. Jun Kong	
Dr. Simone Ludwig	
Dr. Volodymyr Melnykov	

Approved by Department Chair:

4/27/2012	Dr. Brian Slator
Date	Signature

ABSTRACT

Software testing is an important factor of the software development process. Integration testing is an important and expensive level of the software testing process. Unfortunately, since the developers have limited time to perform integration testing and debugging and integration testing becomes very hard as the combinations grow in size, the chain of calls from one module to another grow in number, length, and complexity. This research is about providing new methodology for integration testing to reduce the number of test cases needed to a significant degree while returning as much of its effectiveness as possible. The proposed approach shows the best order in which to integrate the classes currently available for integration and the external method calls that should be tested and in their order for maximum effectiveness. Our approach limits the number of integration test cases. The integration test cases number depends mainly on the dependency among modules and on the number of the integrated classes in the application. The dependency among modules is determined by using an information retrieval technique called Latent Semantic Indexing (LSI). In addition, this research extends the mutation testing for use in integration testing as a method to evaluate the effectiveness of the integration testing process. We have developed a set of integration mutation operators to support development of integration mutation testing. We have conducted experiments based on ten Java applications. To evaluate the proposed methodology, we have created mutants using new mutation operators that exercise the integration testing. Our experiments show that the test cases killed more than 60% of the created mutants.

ACKNOWLEDGMENTS

I would like to thank my supervisor, Dr. Kenneth Magel, whose guidance, encouragement, incredible support, and insightful comments from the first level to the final level enabled me to complete my dissertation research.

I would also like to thank Dr. Simone Ludwig, Dr. Jun Kong, and Dr. Volodymyr Melnykov, for accepting to be members of my committee, and for their support and feedback.

Special thanks to my parents, my wife and my kids for their support, encourage, and being a source of love and support.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Figure Page

CHAPTER 1. INTRODUCTION

As computer applications have become larger, more complex, and more varied during the last sixty years, the need to check the computer software for mistakes as it is developed has become more and more critical. While various forms of inspection or walkthrough can be useful and are widely applied, actual testing of the application is still the major technique for determining where and what these almost inevitable mistakes are. Whichever methodology is used for software development, testing forms a critical, expensive part of that methodology.

Testing has many forms. Initially, as individual units (classes in object-oriented code, modules in structured code, and functions in scripting languages and web applications) are developed they are tested in isolation to determine whether or not they contain mistakes. This process is called unit testing. Substantial research has been done into effective methods for unit testing. When mistakes are found in an individual unit, a debugging process is used to find the cause of the mistake and to implement a solution for that mistake. Once units have been tested and debugged to an expected level of quality, the units are gradually combined with other units into larger and larger combinations according to the overall architecture of the intended application. Each time units are combined, a different set of test cases is applied. This process is called integration testing. As mistakes are uncovered, another debugging process is used to identify and correct them. Some research has been done into ways of organizing the units for integration testing, but little effort has been expended on studying the integration testing methods. Once the combinations are sufficiently large to completely implement significant functionality useful to the intended end users, the application moves into system testing. Some research has explored ways to do system testing including user interface testing and functionality

testing. Specialized methods for security testing, performance testing, scaling testing, and so on have been developed.

There are other forms of testing which may or may not be used in a specific development project. For example, acceptance testing is similar to system testing except that the intended user organization does the testing independent of the development organization. Alpha, beta, and gamma testing are ways to involve large numbers of selected, intended users in the process on their own equipment in their own environments. Regression testing is reusing some or all of the test cases to check whether or not a significant change to the application has introduced new mistakes.

Among all these forms of testing, integration testing may be the most costly and the most important[7]. The cost of integration testing may be 50 – 70% of the cost of the entire testing activity [7]. An empirical study reported that 39% of the errors uncovered in the application studied were interface errors [1].Integration testing tries to find mistakes in how one unit uses the public interface of another unit. As the combinations grow in size, the chain of calls from one unit to another which are being tested, grow in number, length, and complexity.

At the same time, integration testing seems to be the neglected form of testing with respect to the amount of research attention paid to it. The research reported in this dissertation makes a small step towards addressing this lack.

My work addresses two main problems: (1). How can integration testing be made less costly while retaining as much as possible of its effectiveness; and (2). How can a given integration testing process be evaluated for effectiveness. This work provides tentative, partial answers to these two problems.

My approach to the first question involves trying to limit the number of integration test cases while still retaining much of the effectiveness of a more complete set of test cases. We start with the assumption that the degree of dependency of one unit on another is an excellent indicator of the sensitivity of the first unit to mistakes in the second unit. Actually, these mistakes often are not really mistakes. Instead they are differences between what the first unit assumes about the second unit and what the second unit actually does. Remember that good unit testing should have revealed most of the mistakes in either unit alone, and an adequate debugging process should have removed these mistakes.

Two examples of differences are described in the rest of this paragraph. Assume we have two units, U1 and U2. U1 uses a method, m, in U2 to implement some functionality for U1. The code in U1 calls the method, m. U1 assumes that the first parameter to method m is a temperature in tenths of a degree on the Kelvin scale. However, in fact, method m treats this parameter as being in units of hundredths of a degree. Thus, if U1 were to call m with a parameter of 372 meaning 37.2 degrees Kelvin, m would interpret the parameter as meaning 3.72 degrees Kelvin. The result returned to U1 by m would almost certainly be incorrect. This is a mistake that would not be caught during unit testing since U1 would call a stub method m that would have the same interpretation of this parameter that U1 did and m would be called by test cases that had the same assumption about this parameter that U2 did. As a second example, consider the same situation where unit, U1, is calling method, m in unit U2. The first two parameters for this call are given in reverse order in U1 from what m is expecting. If they were both the same data type, no compiler error would be given, but the result returned by m to U1 almost certainly would be wrong.

There are more indirect mistakes that could occur during integration testing as well. For example, suppose U1's method, m1, called a method, m2, in U2 which called a method, m3 in U3. Method, m3 computed a result which m2 returned to m1, and which m1 interpreted differently from how m3 intended that result to be. I argue, however, that all these more complex situations can be resolved to a misinterpretation at one more boundaries between units. In this case, either there is a misinterpretation at the boundary where m1 calls m2 or at the boundary where m2 calls m3, or both. Therefore, we need consider only simple boundary misinterpretations in our work.

How do we determine the degree to which one unit depends upon another? Dependency is a semantic concept. Computers require sophisticated artificial intelligence techniques to deal with semantic concepts. In most cases, including unit dependencies, these techniques have not been applied yet. In my work, I decided to use a more primitive technique to produce a crude, but useful approximation to unit dependency. This technique comes from Information Retrieval. Information retrieval is an active research area which tries to effectively characterize documents within large collections to make searching and selection more efficient and effective. For example, suppose I need to find documents that provide information about apricots within a collection of one million such documents. I could spend substantial time searching or reading each document for the word apricot. I would miss documents that contained information about apricots by their scientific name, or about related fruits. As an alternative, I could use information retrieval techniques to index the documents quickly and find all relevant documents much more quickly.

The other problem area addressed by my work involves a method for evaluating the effectiveness of a set of test cases. I need such a method to determine how well my information retrieval-based approximation to dependency works as well as my test case selection method. In the unit testing and systems testing areas, there are two basic approaches: some notion of coverage, and some method for error seeding.

Coverage comes in several forms. The simplest form is statement coverage: what percentage of the total statements in the source code of my application was executed at least once by my set of test cases. More complex forms of coverage include path coverage (of all the possible execution paths through my source code, what percentage was exercised at least once by the set of test cases) and condition coverage (of all the possible values of all the conditional clauses in my source code, what percentage were executed by my set of test cases).

Error seeding involves having a third party place a set of mistakes in the source code. The assumption is that if the test cases found x percent of the seeded mistakes, those test cases found x percent of the actual mistakes. For example, if fifty mistakes were seeded throughout the code and forty of those mistakes were discovered by the test cases, we assume that 80% of the actual mistakes present in the source code were revealed by the test cases.

A major variety of error seeding is mutation testing. Mutation testing has been applied to unit testing. Mutation testing takes the original unit and creates a large set of variants of that unit by applying an operator from a small set of mutation operators. For example, a subtraction sign might be changed to a multiplication sign. Each variant, called a mutant, differs from the original unit in having one change caused by one application of a single mutation operator. All of the test cases are run on all of the mutants as well as the original unit. If a test case causes a

mutant to return something different than the original unit, that mutant is said to have been killed by that test case. The set of test cases is evaluated by the percentage of the total mutants killed by at least one of the test cases in that set.

I decided to extend mutation testing for use in integration testing. Existing sets of mutation operators make changes to the executable statements within a unit. I added new mutation operators to create variants of calls from one unit to another since that is what integration testing evaluates.

The remainder of this dissertation is divided into five chapters. Chapter 2 explains the approaches taken in this research. Chapter 3 presents related work in the areas of integration testing, mutation testing, and information retrieval. Chapter 4 describes the tool I developed to support this research. Chapter 5 explains the experiments done in this research to evaluate those approaches and presents the results with some analysis of those experiments. Chapter 6 concludes the dissertation and proposals for follow on work.

CHAPTER 2. RESEARCH PROBLEMS AND APPROACH

Our research has three components. The first component is the development of an integration testing approach. The second component is the development of a set of integration mutation operators to support development of integration mutation testing. The third component is the implementation of two software tools to assist in using our approach. This chapter concentrates on the first component.

The Context

Software integration is a lengthy process with many possible errors. For example, suppose we are developing a moderate size application consisting of 500 classes with a total of 2,000 methods within those classes. These 500 classes are being developed by three teams of developers. Once development is underway, a class might be made available for integration at any time. Therefore, unless the classes are developed in a very constrained order, we cannot know which classes would be available for integration when. If the average number of external methods called by the methods within each class is five, we have a total of approximately 2,500 external method calls to test. Finally, there are at least the following potential errors for which integration testing needs to be done: (1) the calling method has the parameters in a different order than the called method; (2) the wrong method is called; (3) a method that should have been called is not called; (4) the result of a method call is misinterpreted by the calling method; (5) different methods, perhaps in different classes, interfere with each other (for example, they each read the next record from a file); and (6) methods are called in the wrong order.

As explained in the first chapter, there are three research problems inherent in integration testing: (1) what is the best order in which to integrate the classes currently available for integration; (2) which external method calls should be tested in what order for maximum effectiveness; and (3)

which test cases are likely to be most effective in finding problems such as those listed in the previous paragraph. The work reported in this research deals with the first two questions. The final chapter discusses how the reported approach could be extended to address the third question, but we have left that effort for the future.

Our approach assumes that the developers doing integration testing have a set of classes currently available for integration. Each of those classes has gone through unit testing and debugging. This unit testing and debugging resulted in each of those classes having acceptable quality when considered in isolation. Integration is likely to reveal errors not discovered by unit testing because integration testing deals with how the methods actually are used rather than how the design specifies that they will be used.

The developers have limited time to perform integration testing and debugging. Accordingly, the developers want to use test cases that will provide the maximum possible effectiveness in revealing problems. Moreover, the results of these test cases should provide significant assistance in the integration debugging process when errors are revealed.
We make an assumption in our work. Namely, the likelihood that a method call will be erroneous is correlated significantly to the degree in which the calling method and called method depend upon each other. Unfortunately, dependency is a complex semantic relationship. Artificial intelligence techniques currently are not sufficiently developed to identify the degree of dependency. We will use a measure of information similarity derived from information retrieval as a proxy for dependency.

Finally, we need a means for evaluating the effectiveness of our approach. We extend the mutation testing approach that is a well-accepted alternative in unit testing to integration testing. This extension requires that we replace the usual set of mutation operators for units in

isolation by a newly developed set of integration mutation operators. We selected ten moderate size open source applications as experimental subjects.

Objective

Our objective is to develop a feasible approach to integration testing that reduces the number of test cases needed to a significant degree while still finding at least sixty percent of the possible errors. If we assume that a full testing approach would require at least one test case for each external method call for each of the six types of error presented earlier in this chapter, we want to reduce the number of test cases by at least fifty percent.

For our example application of 500 classes with 2,500 external method calls, a full testing approach would require 2,500 * 6 = 15,000 test cases. We tried to use no more than 3*number of classes or 1,500 test cases for our example.

Our Approach

Assume we have a set, S, of classes ready to be integrated. We start by forming each pair of two different classes from S. For each pair, we use Latent Semantic Indexing, a technique from information retrieval, to calculate a class pair weight [57] [68] [69] [75].

Latent semantic indexing (LSI) uses singular value decomposition (SVD) to identify patterns in relationships among words in a text. We adapt LSI to the tokens within a method. We eliminated the keywords and required punctuation of the programming language. For example, the semicolon at the end of each statement, or the curly brackets around each block of code were not considered. Comments were excluded as well. Compound identifiers were split into their constituents.

For example, if the source code of a three methods was:

```
Method 1: public employee ( String name, double salary)
        {
                this.name= name;
                this.salary= salary;
        }
Method 2: public void raissalary(double amount)
{
        salary += amount;
}
Method 3:publicstaticvoid check(employee emp)
   {
        emp = new employee();

   }
```

The resulting pre-processed method would be:

Method1: "employee name salary namename salary salary"
Method2: "raissalary amount salary amount"
Method3: "Check employee emp emp employee"

LSI starts by constructing a term-document matrix. For our approach, this is the token-method matrix. Each token is represented by a row of the matrix. Each method is represented by a column. A cell gives the number of times that row's token appears in that column's method.

For our example methods, the token-method matrix would be:

$Term$	$Method1$	$Method2$	$Method3$
employee	1	0	2
$name$	3	0	0
$salary$	3	1	0
$raissalary$	0	1	0
$amount$	0	2	0
$Check$	0	0	1
emp	0	0	2

Next, the values of the matrix are weighted using term frequency-inverse document frequency (tf-idf) values which assess how important a particular word is to a given document.

$w_m = f_{w,m} * \log (M/f_{w,M})$

Where $f_{w, m}$ equals the number of times w appears in m, M is the size of the corpus, and $f_{w, M}$ equals the number of methods in which w appears in M [72] [35]. For our example methods, the token-method matrix after tf-idf weighting would be:

$$\begin{bmatrix} 0.176 & 0 & .352 \\ 1.431 & 0 & 0 \\ 0.528 & 0.176 & 0 \\ 0 & 0.477 & 0 \\ 0 & 0.954 & 0 \\ 0 & 0 & 0.477 \\ 0 & 0 & 0.945 \end{bmatrix}$$

Next, singular value decomposition is done. SVD converts the token-method matrix, M, into three other matrices. If M is an axb matrix (a rows and b columns), and T, S, and D^T are the three new matrices

$M = T * S* D^T$

Where $T*T^T= I$, $D^T *D= I$; the columns of T are orthonormal eigenvectors of $M*M^T$, the columns of D are orthonormal eigenvectors of M^T*M, S is a diagonal matrix whose diagonal elements are non-negative and ordered in decreasing order. The elements on the main diagonal of S are known as singular values of M and are the square roots of the eigenvalues of M^T*M and $M*M^T$.T and D are matrices whose columns are left and right singular vectors of M. each row in the D matrix represents method vector.

$$\begin{bmatrix} 0.126 & 0.307 & 0.006 \\ 0926 & -0.064 & -0.107 \\ 0.350 & -0.034 & 0.123 \\ 0.024 & -0.027 & 0.440 \\ 0.048 & -0.055 & 0.881 \\ 0.017 & 0.427 & 0.026 \\ 0.034 & 0.845 & 0.052 \end{bmatrix}$$
T

$$\begin{bmatrix} 1.539 & 0 & 0 \\ 0 & 1.114 & 0 \\ 0 & 0 & 1.078 \end{bmatrix}$$

S

$$\begin{bmatrix} 0.996 & 0.077 & 0.055 \\ -0.050 & -0.064 & 0.997 \\ -0.080 & 0.995 & 0.060 \end{bmatrix}$$

D^T

$$\begin{bmatrix} 0.996 & -0.050 & -0.080 \\ 0.077 & -0.064 & 0.995 \\ 0.055 & 0.997 & 0.060 \end{bmatrix}$$

D

Next, we reduce the high dimensional methods vectors into low dimensional space by using the low rank approximation. We chose two as rank. In our example, the matrices would be:

$$\begin{bmatrix} 0.126 & 0.307 \\ 0926 & -0.064 \\ 0.350 & -0.034 \\ 0.024 & -0.027 \\ 0.048 & -0.055 \\ 0.017 & 0.427 \\ 0.034 & 0.845 \end{bmatrix}$$

T_2

$$\begin{bmatrix} 1.539 & 0 \\ 0 & 1.114 \end{bmatrix}$$

S_2

$$\begin{bmatrix} 0.996 & -0.050 \\ 0.077 & -0.064 \\ 0.055 & 0.997 \end{bmatrix}$$

D_2

The similarity between two methods is calculated through finding the cosine angle value between methods vectors. This can be calculated as inner product between vectors as follows:

Similarity (q, d) = $\frac{q \cdot d}{|q| \, |d|}$

Similarity (M1, M2) = $\frac{(0.996)(0.077)+(-0.050)(-0.064)}{\sqrt{(0.996)^2+(-0.050)^2}\sqrt{(0.077)^2+(-0.064)^2}} = 0.8$

Similarity (M1, M3) = $\frac{(0.996)(0.055)+(-0.050)(0.997)}{\sqrt{(0.996)^2+(-0.050)^2}\sqrt{(0.055)^2+(0.997)^2}} = 0.005$

Similarity (M2, M3) = $\frac{(0.077)(0.055)+(-0.064)(0.997)}{\sqrt{(0.077)^2+(-0.064)^2}\sqrt{(0.055)^2+(0.997)^2}} = -0.6$

Thus from the above results we can see that there is a similarity between Methods (M1and M2). However there is no similarity between methods (M1 and M3) and (M2 andM3). The class pair weight is computed from the method pair weights for the methods in the two classes by adding all the method pair weights. Next, we sum all the class pair weights to form the total pair weight, T. Each class pair weight is divided by T to form the adjusted class pair weights.

For our preliminary work we decided that the number of test cases we would use would be a small multiple of the total number of classes to be integrated. In the work reported here, that multiple is 5. Thus if we have n classes to be integrated, we would have 5*n test cases. These test cases are allocated to each pair of classes by multiplying the adjusted class pair weight by the total number of test cases and rounding up.

CHAPTER 3. LITERATURE REVIEW

This research concerns integration testing, mutation testing, and information retrieval. Associated literature on the above mentioned areas is discussed in this chapter. This chapter consists of three main sections: the first section presents integration testing definitions, integration faults and integration testing strategies for structured programming languages and object-oriented languages. The second section illustrates mutation testing in general, and shows different mutation operators. The last section describes generally information retrieval techniques and then specifically Latent Semantic Indexing.

Integration Testing

Testing separate classes in the software independently assists in removing errors at the class level, but does not guarantee that the software is error free. Unit testing does not have the ability to reveal errors that might happen when integrating classes together including, interface problems and missing functionalities. Integration testing is the activity of bringing together the different classes that compose the software to ensure that these classes are interacting together without causing any error or system failure [34][16] [9] [15].

Integration Faults

The types of discovered faults at the unit testing level are not the same as faults at the integration testing level. Leung and White, 1990 [32] categorize integration faults into four main categories:

1. Interpretation errors: interpretation error happens when integrating two classes together and the type of behavior expected through a user of a class is not equivalent to the functionality

offered by the class. The developer of a calling class might get the wrong idea about the functionality of the called class. Interpretation errors are further classified as:

- Wrong function errors: This happens when the functionalities offered by the called class are not same as the required by the calling class. For example, a calling class may incorrectly presume that the called class will return a sorted array rather than an unsorted array.

- Extra function errors: This is when the called class provides certain functionalities which are not required by any caller class.

- Missing function errors: This is when the calling class attempts to call a function not available in the called class.

2. Miscoded call errors: This is when the programmer puts the call instruction at the incorrect place in the calling class. Miscoded call errors are further classified as:

- Extra call instruction: This happens when the order that is carrying out the invocation is inserted in a place that should not include the invocation.

- Wrong call instruction placement: this type of error happens when the invocation is inserted on the correct path, but in an incorrect position.

- Missing instruction: this type of error happens when the invocation is not found on the path that should include it.

3. Interface errors: An interface error happens when the defined interface between two or more classes is violated, for example a wrong parameters order, an invalid parameter type or format, and an incorrect parameter order.

4. Global errors: This kind of error relates to the inappropriate use of global variables.

Beizer [5] has categorized integration faults into the following types:

- Protection against corrupted data: The calling class attempts to call a function in the called class with invalid data, and the called class does not use any protection or any checking for the data before using it.
- Input/output format faults: The calling class attempts to call a function in the called class with wrong input format. For example, the date parameter the format of "dd-mm-yyyy" is different from " dd-mm-yy"
- Call parameters faults: The calling class attempts to call a function in the called class with wrong parameters.
- Invalid subroutine call sequence: This happens when calling sequence of functions incorrectly. For example, if the valid sequence of function calls should be f1, f2, f3 and f4, yet the sequence of function calls: f1, f2, f4, and f3.
- Invalid parameter values: the calling class attempts to call a function in the called class with incorrect values parameters**.**

The categorization of integration faults by Leung and White [32] is more accurate and comprehensive than the categorization by [5]. Beizer has focused mainly on the function parameters whereas [32] provide wide variety of errors that may occur at an integration level.

Integration Testing Strategies

Software integration strategy commonly refers to an integration chain or order of integration components or parts for the entire system.

The major traditional integration testing strategies are classified into five strategies as follows:

1. Top-down strategy: In top-down integration testing strategies the integration testing process starts with the highest class, which is marked out by the use relation between classes. To the same degree the integration testing process starts with the class that is not used by other classes. In this strategy the stubs are required. Stubs are dummy implementation or incomplete classes use only to let the higher class to be tested. For example given an system consists of eight modules and the following figure represents the call graph of the system under test:

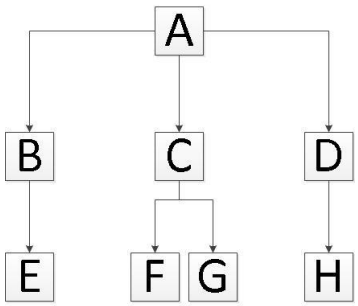

Figure 3.1. Call Graph

Thus the next table shows the steps of top-down integration testing. In other words it shows which modules are integrated first, the order of module integration as well as the required stubs in each step.

Step	Integrated classes	Required stubs
1	A and B	C,D and E Stubs
2	A, B, and C	D, E, F and G Stubs
3	A,B,C and D	E, F, G, and H Stubs
4	A,B,C,D, and E	F, G, and H Stubs
5	A,B,C,D,E, and F	G, and H Stubs
6	A,B,C,D,E, F, and G	H Stub
7	A,B,C,D,E, F, G, and H	No Stubs

Table 3.1. Top-Down Integration Order

This approach permits early verification and proof of high-level behavior. However, the top-down approach postpones the verification of low level behavior, requires creating stubs for each missing or untested module which leads to an increase the cost, raises probability of error prone, and increases the difficulty of test cases input and output preparation [7][20].

2. Bottom-up strategy: This strategy is the opposite of the top-down strategy. The integration testing process starts with the lower class which is marked out by the use relation between classes. It begins with the class that does not depend on other classes. In this strategy the drivers are required to simulate the caller.

   This approach permits early verification of low-level behavior, ease of preparation for inputs and outputs of test cases and does not require stubs. However it does require drivers for missing modules and postpones the verification of high level behavior. For example if we need to do bottom-up integration testing for the same system in the above figure, then the following table shows the steps of bottom-up integration testing and the required driver for each step:

Step	Integrated classes	Required drivers
1	E and B	A Driver
2	F,G, and C	A Driver
3	H and D	A Driver
4	E,B,F,G,C,H,D and A	No drivers

Table 3.2. Bottom-Up Integration Order

Our approach is different from both the top-down and bottom-up approaches in many respects. First we do not need to create any stubs nor drivers which as a results leads to reduce the cost and error prone operations [20]. In addition our approach depends

on class pair weight which is calculated using a latent semantic indexing technique to determine how many test cases should be developed for each class pair. Moreover we do not need to create the call graph of the system under test. Furthermore our approach helps in determining which methods are connecting classes together in order to focus on them from a testing perspective.

3. Big-bang strategy: The integration testing process starts once all the classes are developed and tested separately; it combines all the classes together to see if they are working or not. Although this strategy does not need stubs and drives, it is not recommended because of the difficulty in finding the error causes and the complexity in distinguishing the interface errors from other types of errors.

   Our approach is similar the big-bang approach because it does not need to create neither stubs nor drivers and starts till all the modules are implemented. However, our approach follows systematic calculation based on calculating class pair weight through using latent semantic indexing technique. From these calculations we can determine the number of test cases for the whole application. In addition the calculations help us in specifying which parts of the class pair are most important in order to create test cases for these parts. Furthermore our approach can help in overcoming the huge number of potential test cases [28].

4. Bi-Directional Integration: This kind of approach is a mixture of the top-down and bottom-up integration approaches used jointly. It requires both stubs and drivers since it is a composite of top-down and bottom-up approaches. This approach is also known as either sandwich integration or hybrid integration. This approach is recommended for use when migrating from a two-tier to a three-tier environment [31]. To illustrate this

approach, the following figure represents the system's modules and the table represents the steps of bi-directional integration.

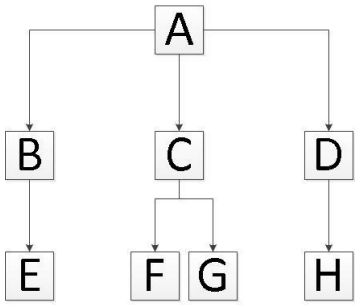

Figure 3.2. Call Graph

Step	Integrated classes	Approach type
1	B and E	Bottom-up
2	C,F,G	Bottom-up
3	D and H	Bottom-up
4	(A,B,E)-(A,F,G)-(A,D,H)	Top-down

Table 3.3. Bi-Directional Integration

Desikan and Ramesh [31] suggest guidelines for choosing which integration testing approach (top-down, bottom-up, bi-directional, and Big bang) based on the factors shown in the following table:

Factors	Suggested integration method
Clear requirements and design	Top-down
Dynamically changing requirements, design,	Bottom-Up
Changing architecture, stable design	Bi-directional
Limited change to existing architecture with less	Big bang

Table 3.4. Integration Strategies

5. Thread strategy: this kind of integration testing combines and integrates classes based on the expected execution threads.
6. Critical classes strategy: classes are merged together based on the class level of criticality, the classes with high critical are combined together first.

Object Oriented Integration Testing Strategies

Object oriented languages have many characteristics that traditional languages do not have. Such characteristics include among others, encapsulation, polymorphism, inheritance, dynamic binding, synchronization, threads, and others. An ordinary difficulty in inter-class integration testing of object-oriented system is the decision about the order in which modules are integrated and tested [21].

When modules are integrated and tested an order of integration should be recognized. The problem occurs when there is a cyclic dependency. This problem is generally called the class integration and test order (CITO) problem [20]. Various researchers [20] [22] [23] [24] [25] [26] [27]have proposed many solutions for the CITO problem.

New integration testing strategies are necessary for object oriented programs. Many likewise researchers have proposed strategies for object oriented integration testing. Jorgensen and Erickson [5] classify five integration levels in object oriented as follows: (1) integration of methods into a single class, (2) integration of two or more classes through inheritance, (3) Integration of two classes through containment, (4) integration of more than one class to form a component and (5) Integration of components into a single application.

In addition, Overbeck[6] identifies three types of typical integration testing strategies: (1) execution based integration testing to uncover wrong interactions of units through tracing the interaction execution;(2) value based integration testing which uses particular values to execute units' interaction and (3) function based integration testing which certifies the functionality of modules while they are interacting.

[7] proposes another approach for integration testing called thin thread. Thin thread is defined in [8] as "A complete trace (E2E) of data/messages using a minimally representative sample of external input data transformed through an interconnected set of systems (architecture) to produce a minimally representative sample of external output data. The execution of a thin thread demonstrates a method to perform a specified function". The following figure shows an example of the thin thread tree for a bank system:

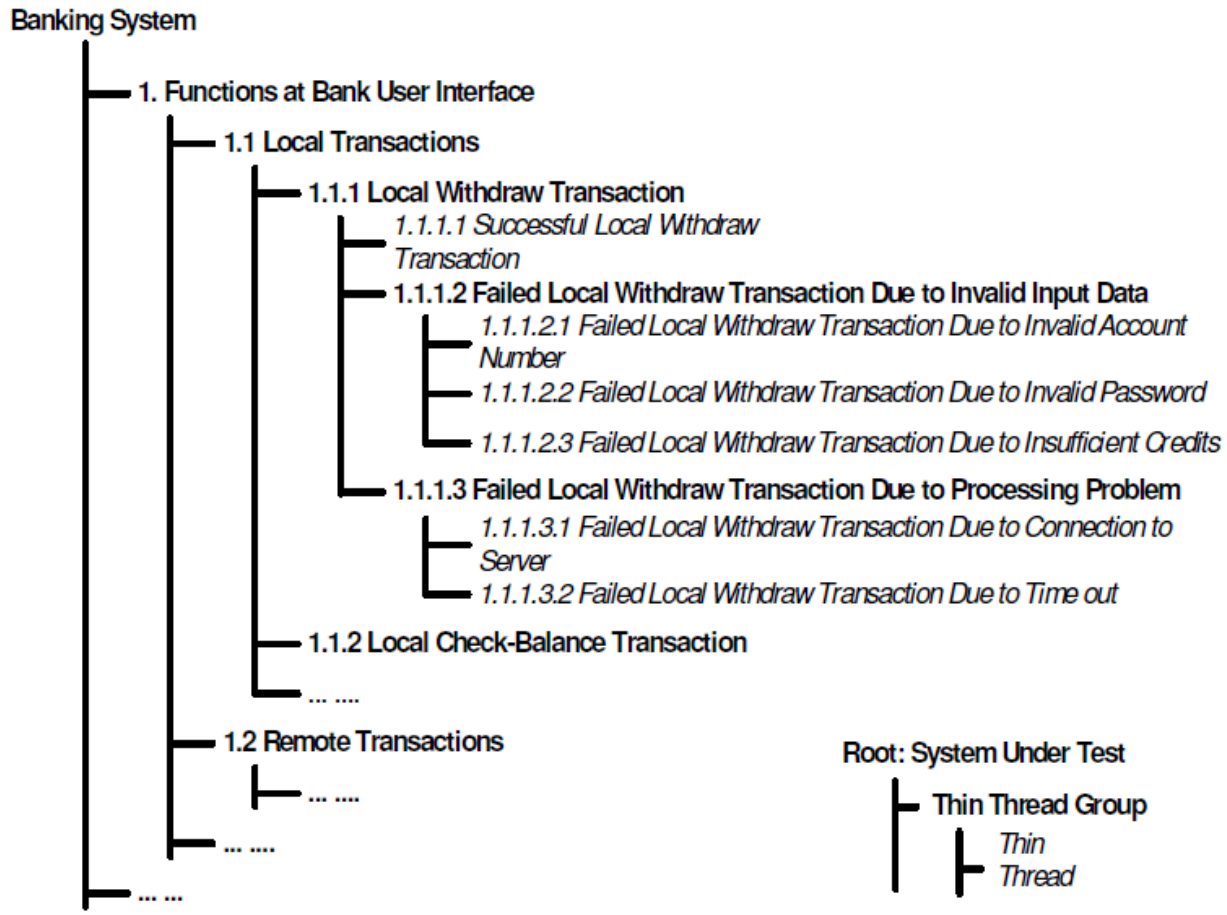

Figure 3.3. Thin Thread Tree For Bank System [7]

The root of a thin-thread tree shows the whole integrated system under test, in which the branch node shows a group of connected thin threads and a leaf shows a concrete and specific

thin thread. The next step after constructing a thread tree is to identify conditions. Conditions are predicates that influence the running of thin threads. A thin thread is considered triggered if, and only if, its conditions are entirely valid. The test cases then are generated from a thin thread identifying through various testing techniques the input data that fulfilled the conditions related to the thread. The expected output is identified from the description of the thread. The problem with this approach is the complexity of building the thin thread which requires the comprehensive and detailed knowledge and awareness of functionalities of the system and the architecture of the system as well.

Our approach does not require any knowledge or comprehensive understanding of the system under test. Since we use the latent semantic indexing technique in determining which method is connecting classes together as well as in specifying how many test cases should be made for the whole system under test. Thus anyone who has no knowledge about the system under test can determine the number of test cases.

Other researchers have utilized Unified Modeling Languages models (UML) in integration testing [9] [10]. [9] provide a new testing method based on collaboration diagrams and state charts in order to reveal state-dependent interaction faults, such as changeable states of classes, incorrect calling state of a class, and incorrect initial state of a class. Their integration testing strategy is based on the concept that the interaction among objects should be exercised for every likely state of included objects. They propose a test model called State Collaboration TEstModel(SCOTEM) which uses the state chart to identify the behavior of each class and uses the collaboration diagram to identify the test directives as shown in the following figure

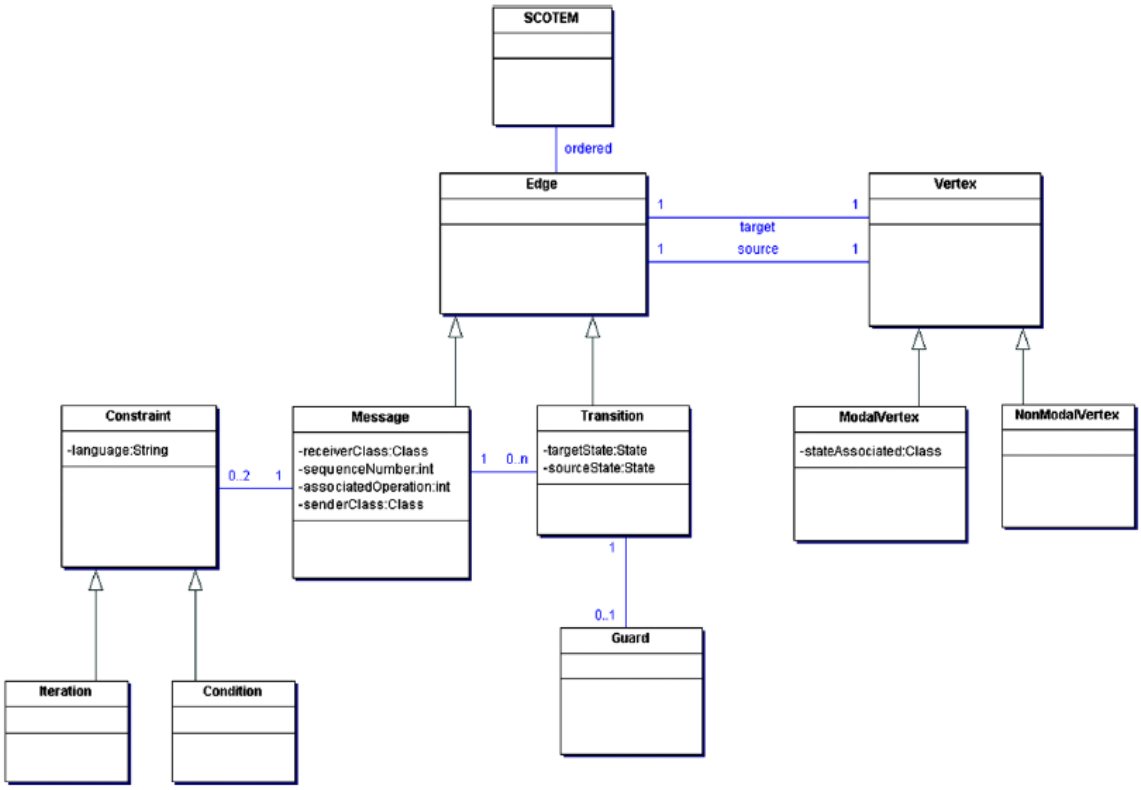

Figure 3.4. State Collaboration TestModel (SCOTEM) [9]

A vertex refers to an object of a class contributing in the integration. A modal class obtains a message in more than one state and shows various behaviors for the equivalent message in distinctive states. This model indicates many vertices, in which every vertex relates to an object of the class in different abstract. On the other hand, a non-modal class needs a single vertex solely in the SCOTEM graph. There are two types of edges: message and transition. A message edge indicates a call action between two objects, and a transition edge indicates a state transition of an object when getting a message. Every message edge might hold a condition or iteration as well. Every message can trigger a state transition. A transition edge links two vertices from the equivalent class. State charts might have many transitions to different states for the equivalent operations. Therefore, there can be many transition edges for the equivalent message edge in SCOTEM. The inner information of a vertex contains the class name and state of the

objects which it relates. Message edge is created in SCOTEM through attributes of a message involving associated operation, message sequence number, the sender object, and receiver object. The transition edge is formed through the attributes of a transition involving sending state, accepting state, associated operation, and sequence number [9].

A test path resulting from the SCOTEM represents a path that begins with the initial vertex and includes the entire message sequence of the collaboration. The overall number of the test paths in SCOTEM can be computed through computing the product of the numbers of the transition paths in every class, where every transition path is an inner transition of a model class from a source state to a target state upon receiving of a specific message. However this approach requires all guards, paths, and loops conditions to be specified using Object Constraint Language (OCL) [9].

This approach presents the problem of state explosion since it uses the state diagram. Also this approach does not determine the order of integrating classes as well as the number of test cases for the whole system under test. Our approach does not require state diagram nor collaboration diagram to determine the collaboration among modules in the system under test. Furthermore, in [10] they propose a new algorithm for integration testing called TEst Sequence generaTOR (TeStor). It permits the testers to get test sequences from state diagrams and sequence diagrams. The state diagram provides the component behaviors whereas the sequence diagram identifies what the test should include. In other words, the TESTOR needs a behavioral model of the components as an input in terms of state machines and a sequence diagram denoting the test directives, and it generates a set of sequence diagrams representing the paths which the tests should follow. This algorithm requires the structural specification, as well as the behavioral, specification to be available all together with architectural information that permits the testers to

determine how modules are assumed to interact when they are integrated. In addition, this approach removes loops from the state machines which means some aspects are left without testing.

Our approach is different from the approach in [10] in determining how modules are integrated. The algorithm in [10] requires the structure specification as well as the behavioral specification to be available together with architectural information. While our approach requires only the source code of the system under test to determine interactions and find out how classes are integrating with each other.

Additionally, in [2] they propose a formal specification method for integration testing specifically for object oriented programs. Specifically they formally identify the behavioral dependencies and interactions between objects of various classes formally. Contract is one of the formal languages to specify the behavioral properties. The behavioral property is defined by "message-passing rules" (mp-rules).

Mutation Analysis For Integration Testing

The mutation analysis has been used in integration testing. Delamaro et al. [44] initially illustrated the technique of interface mutation for the integration testing of C programs. The fundamental idea is to produce mutants solely through suggesting minimal changes in the classes belonging to the interface between modules [12]. Mutation testing is used to evaluate the effectiveness of test suite in detecting errors. The use of mutation testing does not provide any guideline to determine how classes are interacting with each other nor how to create test cases at integration level. We use the mutation testing to evaluate our approach.

Tom and Li [13] propose a test framework for testing object oriented systems at the integration level. They create integration test cases based on UML class diagrams and sequence diagrams in terms of coordination contracts. Coordination contracts are connections that are established among a collection of participants objects [14]. After getting class diagram and sequence diagram specification, Tom and Li [13] integration testing process works as follows: (1) the XML Parser parses the class diagram and sequence diagram and represents them in XML notation. (2) Test cases are realized in terms of contacts. What to test and how to test results are described in the contracts rule. (3) The Coordination Development Environment (CDE) is used to create code from the contracts and the components under test to structure the test framework. CDE is a tool to help develop Java applications using coordination contracts [14].

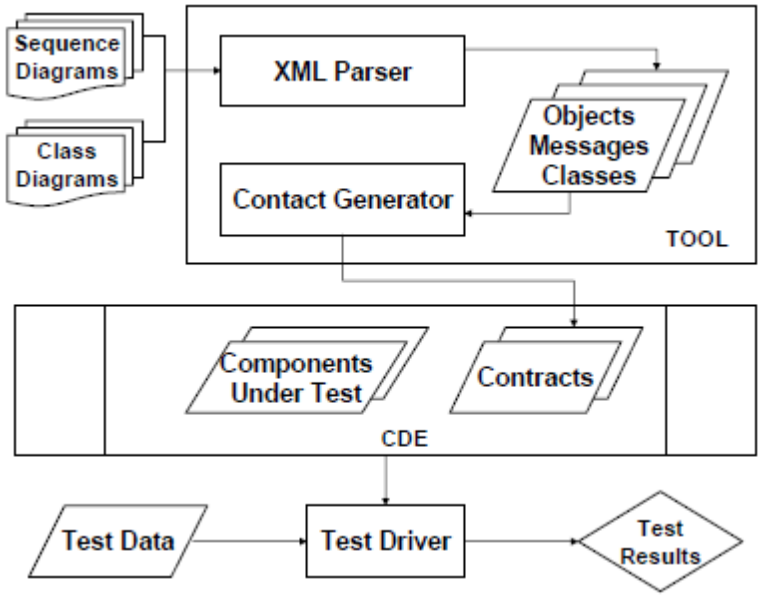

Figure 3.5. Tom And Li Test Framework [13]

This approach depends on the sequence diagram and class diagram to generate the test cases. Thus, if there is mistake in class diagram or sequence diagram then test cases will be inaccurate. Even more, the source code of the system under test may not be compatible with the

class diagram or class diagram. Our approach mainly uses the source code in determining the interactions among modules of the system under test.

Our approach is different from [13] approach in many ways. Our approach does not require either class diagram or sequence diagram. In addition we use Latent Semantic Indexing (LSI) in determining how a pair of classes is related to one another.

Yuan and Xie [18] propose a framework for automatic generation of integration tests called Substra. Their framework depends on call sequence restrictions inferred from initial- test execution or usual runs of the subsystem under test. These restrictions rely on two types of information: shared subsystem states and define-use relationship. Substra employs an object state machine to model these restrictions. A subsystem's state is represented as nodes in the state machine, function calls as transition, and define-use relationships as guard conditions of transitions. Substra proposes an iterative process that uses initial test executions or normal runs of the system to infer sequencing restriction dynamically and uses these restrictions to assist in the creation of new tests. Each one iteration involves six steps: gather execution traces, discover boundary calls, infer define-use relationships, build basic object state machines, build a subsystem state machine, and create a new test as illustrated in the following figure:

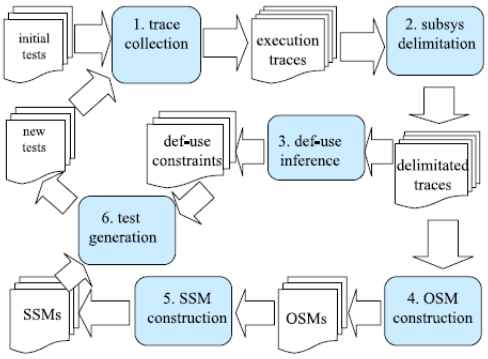

Figure 3.6. Substra Framework [18]

This approach is different from the Tom and Li approach which requires specification in terms of class diagrams and sequence diagrams while the [18] approach does not require any specifications. However their approach supposes that every test in the initial test suite is correct and valid. If the initial test suite employs incorrect behaviors it may not be valid or important. Our approach requires neither any pretests nor any diagram constructions.

Mutation Testing

Mutation testing is a fault based testing technique that evaluates the effectiveness of test cases. Mutation testing, initially proposed in 1978, is based on the fact that software will be well tested if whole simple faults are detected and removed. Simple faults are created in software through producing a collection of faulty versions, called mutants. Test cases are used to carry out the mutants with the goal of leading every mutant to create inaccurate output. A test case that differentiates the software from its mutant is viewed to be effective at discovering faults in the software [33][12] [35] [36] [42] [42] [43] [44]. The mutation process depends on mutation operators in order to generate mutants from the original source code as shown in the following figure:

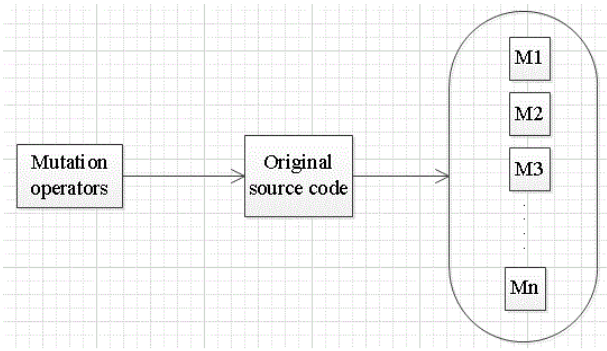

Figure 3.7. Mutation Process

Mutation testing relies on the competent programmer hypothesis and the coupling effect. The competent programmer hypothesis declares that programmers are commonly capable and produce software is close to accurate software. Accurate software can be created from inaccurate software through making modifications that are composed of minor alternations. The coupling effect declares that test cases that differentiate programs with minor modifications from each other are very precise and that they can differentiate software with more compound modifications. The competent programmer hypothesis and the coupling effect express that small modifications in software are sufficient to help discover compound errors [12].

Let S be the system under test and Sa be one accurate version of S. if S is correct, S and Sa are the same. T is the set of tests used to test S. Let the input domain of S be represented by D. Mutation testing depends on a set of Faults F. Every fault f in F is initiated in S individually. Presentation of a fault into S outcome in a program M is named a mutant of S. The application of all faults in F one by one into S generates a collection of mutants M. Factors of F are identified as mutation operators. When a mutant M executes versus a test case t in T and the performance of M is dissimilar from that of S, the mutant M is said to be killed by t. A tester should kill every mutant in M with a minimum of one test case t. Mutants that are not killed throughout testing are called alive mutants [12] [35] [36] [42] [43] [44].

There are some conditions that should exist in order to kill each mutant. [40] propose the three conditions: let L be a line of code in C class which has been mutated to LM to obtain mutant M, so to kill the mutant by a test T, the test T must satisfy the following conditions: (1) Reachability: the line of code L must be reached when the test T is executed; (2) Necessity: The state of M immediately following some execution of LM must be distinctive from the state of C

immediately following the equivalent execution of L; (3) Sufficiency: the differentiation in the states of C and M immediately following the execution of L and LM must continue until the complete of the execution of C or M such that C (t) ≠ M (t).

In mutation testing the tester is looking for to kill every mutant in mutants set with a minimum of one test case. In the situation when mutant remains a live, the tester must explain that the mutant is equal to the original program M ≡ P or update the Tests set T by improving or adding a test t to T in order to kill the mutant. The test adequacy is identified through the ratio of the number of killed mutants to the number of non-equivalent mutants. This ratio is called the mutation score as shown in the following equation:

$$\text{mutation score} = \frac{\text{number of killed mutants}}{\text{number of non equivalent mutants}}$$

Mutation Operators

Mutation testing inserts faults into programs through mutation operators. There are two types of mutation operators (1) mutation operators for procedural languages sometimes called traditional mutation operators. These operators are Absolute Value Insertion (ABS), Arithmetic Operator Replacement (AOR), Logical Connector Replacement (LCR) and Unary Operator Insertion (UOI); and (2) class mutation operators [38]. Many researchers have proposed and classified many class mutation operators [35] [36] [37] [46] [49] [55] [50]. In more detail [35] classifies the class mutation operators into six groups, based on the language feature that is affected: (1) Information Hiding (Access Control), (2) Inheritance, (3) Polymorphism, (4) overloading, (5) Java-Specific Features and (6) Common Programming Mistakes. As shown in the following table:

Operators	Description
AMC	Access Modifier Change
IHD	Hiding Variable Deletion
IHI	Hiding Variable Insertion
IOD	Overriding Method Deletion
IOP	Overridden method calling position change
IOR	Overridden method rename
ISK	Super Keyword deletion
IPC	Explicit call of a parent's constructor deletion
PNC	New method call with child class type
PMD	Instance Variable declaration with Parent Class Type
PPD	Parameter Variable declaration with child class Type
PRV	Reference assignment with other compatible type
OMR	Overloading method contents change
OMD	Overloading Method Deletion
OAO	Argument Order Change
OAN	Argument number Change
JTD	This Keyword Deletion
JSC	Static Modifier Change
JID	Member Variable initialization deletion
JDC	Java-supported default constructor create
EOA	Reference assignment and content assignment replacement
EOC	Reference Comparison and Content Comparison Replacement
EAM	Accessor Method Change
EMM	Modifier Method Change

Table 3.5. Class Mutation Operator

In addition, Offutt et al. [36] adds to the above mutation operators six new operators as shown in the following table three of them are related to type conversion and three of them are related the "this, super, and static" keywords insertions.

Operators	Descriptions
PCI	Type cast operator insertion
PCD	Type cast Operator Deletion
PCC	Cast Type Change
ISI	Super Keyword Insertion
JTI	This Keyword Insertion
JSI	Static Modifier Insertion

Table 3.6. Mutation Operators For Type Conversion And Keywords Insertions [36]

Moreover Kim et al. [37] increase mutation operators for class level by adding three new operators Compatible Reference Type (CRT), Constructor (CON) and Overriding Method (OVM). In CRT, this operator swaps a reference type with all the compatible types found in the classes. In CON, this operator swaps a constructor with other overloaded constructor. In OVM,

this operator deactivates the overriding method so that a reference to the overriding method in fact goes to the overridden method. In addition [39] prove the capability of using mutation analysis and model checkers to create comprehensive test sets from formal specification. They propose eight mutation operators for specifications: Operand Replacement Operator (ORO), Simple Expression Negation Operator (SNO), Expression Negation Operator (ENO), Logical Operator Replacement (LRO), Relational Operator Replacement (RRO), Missing Condition Operator (MCO), Stuck-At Operator (STO), and Associative Shift Operator (ASO). The table below gives short example explaining each operator.

Operator	Example Mutants
ORO	AG (request →AF state =ready)
SNO	AG (!request → AF state=busy) AG(request →AF(!state=busy)
ENO	AG(!(request→AF state =busy)
LRO	AG(request & AF state =busy) AG(request | AF state= busy)
MCO	AG AF state=busy
STA	AG (0→AF state = busy) AG(1→AF state = busy) AG (request →AF 0) AG (request → AF 1)
ASO	AG(x&(y→z))
RRO	AG (WaterPres<=100) AG (WaterPres>100) AG (WaterPres =100) AG (WaterPres !=100)

Table 3.7. Mutation Operators For Specifications [7]

Offutt et al. [46] provide the following mutation operators FORTRAN:

Operator	Description
AAR	Array reference for Array reference Replacement
ABS	ABSsolute value insertion
ACR	Array reference for Constant Replacement
AOR	Arithmetic Operator Replacement
ASR	Array reference for Scalar variable Replacement
CAR	Constant for Array reference Replacement
CNR	Comparable array Name Replacement
CRP	Constants RePlacement
CSR	Constant for Scalar variable Replacement
DER	Do statement End Replacement
DSA	Data Statement Alterations
GLR	Goto Label Replacement
LCR	Logical Connector Replacement
ROR	Relational Operator Replacement
RSR	Return Statement Replacement
SAN	Statement ANalysis
SAR	Scalar for Array reference Replacement
SCR	Scalar for Constant Replacement
SDL	Statement DeLetion
SRC	SouRce Constant replacement
SVR	Scalar Variable Replacement
UOI	Unary Operator Insertion

Table 3.8. Mutation Operators For FORTRAN [46]

Other researchers provide the following concurrency mutation operators for Java [49] [50]:

Operator	Description
MXT	Modify time parameter t of wait(t), sleep(t), join(t), await(t)
MSP	Modify parameter obj of block synchronized(obj)*f...g*
ESP	Exchange parameter obj of block synchronized(obj)*f...g*
MSF	Modify Semaphore Fairness
MXC	Modify Permit Count in Semaphore and Modify Thread Count in Latches and Barriers
MBR	Modify Barrier Runnable Parameter
RTXC	Remove Thread Call wait(), join(), sleep(), yield(), notify(), notifyAll()
RCXC	Remove Concurrency Call (methods in Locks, Semaphores, Latches, Barriers, etc.)
RNA	Replace notifyAll() with notify()
RJS	Replace join() with sleep()
ELPA	Exchange Lock/Permit Acquisition
EAN	Exchange Atomic Call with Non-Atomic
ASTK	Add static Keyword to synchronized Method
RSTK	Remove static Keyword from synchronized Method
RSK	Remove synchronized Keyword from Method
RSB	Remove synchronized block
RVK	Remove volatile Keyword
RFU	Remove finally Around Unlock
RXO	Replace One Concurrency Mechanism-X with Another (Locks, Semaphores, etc.)
SHCR	Shift Critical Region
SKCR	Shrink Critical Region
EXCR	Expand Critical Region
SPCR	Split Critical Region
DelStat	Deletes a statement from a synchronized block
ReplArg	Replaces argument with constant in a synchronized method
DelSync Call	Deletes a call to a synchronized method
ReplMeth	Uses method with same name and other signature
InsNegArg	Inserts unary (negation) operators in an argument
ReplTarg Obj	Replaces the object in a call to synchronized method

Table 3.9. Concurrency Mutation Operators For Java [49]

Praphamontripong and Offutt [55] propose a group of new mutation operators particularly for testing interaction between web components. They propose two categories of mutation operators. One for HTML and the second one for JSP as follow:

Mutation operators for HTML:

- Simple Link Replacement (WLR): the WLR operator changes a destination of a simple link transition identified in the <A> tag with a different destination in the similar domain of the web application.
- Simple Link Deletion (WLD): the WLD operator deletes the destination a destination of a simple link transition identified in the <A> tag.
- Form Link Replacement (WFR): the WFR operator replaces a destination of a form link transition to a different destination in the similar domain of the web application.
- Transfer mode replacement (WTR): the WTR operator changes all GET requests into POST request and all POST requests into GET request.
- Hidden form field replacement (WHR): the WHR operator changes the attribute values of the <input> tag of type hidden with different value.
- Hidden for field deletion (WHD): the WHD operator deletes the whole block of the <input> tag of type hidden.
- Server-side-include replacement (WIR): the WIR operator replaces file attribute of include directives into different destination in the similar domain of the web application.
- Server-side-include deletion: this operator deletes the whole include directive from the HTML file.

Mutation operator for JSP

- Redirect transition replacement: this operator replaces the forward destination of the redirected transition identified in <jsp:forward> tag to different destination.
- Redirect transition deletion (WRD): this operator deletes the whole redirection, as identified in the <jsp:forward> tag.
- Get session replacement (WGR): this operator opens a new connection to the web server each time a client retrieves the webpage.

Moreover Lonetti and Marchetti [56] propose new mutation operators for Extensible Stylesheet Language Transformations (XSLT) through creating six XSLT mutation classes: Arithmetic Operator Replacement (AOR), Variable Manipulation (VM), Arithmetic, Logic and Relational operator Manipulation (ALROM), XPath Expression Manipulation (XPEM), Condition Iteration Manipulation (CIM), Template Manipulation (TM), and Element Attribute Manipulation (EAM).

The mutation operators mentioned above are applied on individual methods or functions consisting of statements and on individual classes or modules consisting of multiple functions or methods. These operators are a concern of unit testing. Since our research is concerned with integration testing we develop mutation operators which are explained in Chapter 4.

Interface Mutation

The researchers in [40] present a technique that employs existing information from the description of the components interface for testing the component. They use this information to generate coverage domains. Components testing are executed throughout their interface as well. Their method does not depend on the existing implementation of the code. [40] propose five operators for interface mutation in CORBA-IDL as follows: (1) Replace: “Replaces an

occurrence of one of ‘in’, ‘out’ and ‘inout’ with another in the list.”; (2) Swap: “Operator for parameter swapping in the method call. Parameters of the same type could be swapped in a method call”; (3) Twiddle: “this operator is used on a numerical or a character variable that is passed to or from the method”; (4) Set:” the set operator assigns a certain (fixed) value to a parameter or to the value returned from a method.” and(5) Nullify: “the nullify operator nullifies an object reference.”

The mutation operators mentioned in [40] are designed for CORBA-IDL. But our mutation operators are designed for Java and include more mutation operators than in [40]. For example we have mutation operators to alter and modify the chains of calls from one module or class to another or itself.

Information Retrieval

Information retrieval (IR) is discovering and returning material, mostly documents of a shapeless environment typically text that complies with the information need from within large collections commonly existing on computers [57].Information retrieval refers to the demonstration, storage, classification of information materials and gain access to information materials. The models of information retrieval are categorized mainly into two groups. The first group is the keywords oriented model, and the second group is matrix oriented model. Keyword based models make use of particular data structures and search algorithms. Matrix oriented models transform the representation of documents keywords into a matrix format [59].

Information retrieval depends on two ratios (Precision and Recall) to assess and calculate the effectiveness of the information retrieval strategies. Precision is the percentage of the number of related documents retrieved to the total number retrieved. Precision gives an indication about

the quality of the answer set. Recall is referred to the total number of related documents. It is the percentage of the number of related documents retrieved to the total number of documents in the corpus that are assumed to be related [58].

Information Retrieval Strategies

Retrieval strategies determine a degree of similarity among a query and document. A retrieval strategy is defined by [58] as "an algorithm that takes a query Q and a set of documents D1, D2... Dn and identifies the Similarity Coefficient SC (Q, Di) for each of the documents $1 \leq i \leq n$". Information retrieval depends on a technique or algorithms in formulating strategies. The following are brief description of information retrieval strategies:

- Vector Space Model (VSM): The vector space model calculates the similarity between query and document by representing them as vectors, a document vector and query vector, then calculating the similarity by finding the cosine angel between two vectors [73]. VSM is based on the idea that the documents words express the documents meaning.

  Given that the individual keywords are not enough and sufficient in discriminating the semantic content of queries and documents, performance of the VSM suffers from two classical problems of synonymy and polysemy [63][68]. Synonymy refers to the different words with same meaning. Polysemy refers to the same words with different meaning. The occurrence of synonymy is likely to reduce the recall performance and the occurrence of polysemy reduces the precision performance [58]. Because term-document matrices are mostly very dimensional and sparse, then the matrices are at risk to noise [59].

- Inference network: a Bayesian network is utilized to infer the relevance of a query to a document. Inference network depends on the "evidence" in a document that permits an inference to be made about the importance of the document. The similarity coefficient is determined by the weight of the inference [58]. The problem with this strategy is the synonymy and polysemy because it depends on the statistical measures that basically depend on matching the terms between the query and the document.
- Neural networks: a chain of "neurons" or nodes in a network, that execute after a query triggering links to documents. Each link has weight which is transmitted and gathered to calculate the similarity coefficient between the query and the document. Network is trained by changing the links weights in return to predetermined related and unrelated documents [58]. A neural network mainly used in the machine learning and it requires high computational resources.
- Fuzzy set Retrieval: a document is mapped to a set that holds elements and number associated with it. The number indicates the strength of the membership and it represents the similarity coefficient between the query and the document [58]. This strategy has limitations. Among these limitations are semantic model needs in order to take the terms meaning into account, needs high computing expenses used for aggregation and membership function, fast expansion of complexity when input variables number increases and does not have the ability to adapt and change via feedback and learning [60].
- Genetic Algorithms: Genetic algorithm based on a best query to locate related documents, which can evolve. An original query is used with either estimated term weights or random. New queries are created through changing these weights. A new

query continues to exist by being near to known related documents and queries with low closeness to documents are dropped from consequent generations [58]. Genetic algorithms are mainly used in machine leaning and needs high computational resources, which prohibits genetic algorithms for more extensive. [58].

- Boolean indexing: “the score is assigned such that an initial Boolean query results in a ranking. This is done by associating a weight with each query term so that this weight is used to compute the similarity coefficient” [58].Drawbacks of the Boolean retrieval model are no official or proper ways for qualifying the task and measure of the terms in differentiating documents’ contents, matching method depends merely the evaluation of the occurrence of a given search keywords in document representation, impossibility of finding out the degree of value of every particular document and troubles with Boolean operators; Disjunctive (OR) queries result in an overload of information as a result of an extreme amount of results. Conjunctive (AND) queries result in decreasing results, and frequently zero results and it leads to decrease in recall [60].

- Probabilistic Retrieval: a probability depends on the possibility that a term will emerge in a related document is calculated for every term in the collection. The similarity coefficient between the query and the document is calculated by combining the probabilities of every term that matches between a query and document. The problem of this strategy is the need of preexisting information to execute correctly [58].
- Latent Semantic Indexing (LSI): is a modification of VSM to overcome the problem of synonym, polysemy and high dimensional space. LSI attempts to create advantage of the conceptual content of documents through searching on concepts rather than looking for

single terms in the documents [59]. LSI employs one technique in algebra called Singular Vale Decomposition (SVD) in order to reduce the dimensional space[58] [60]. Singular value decompositions arrange the space in order to reveal the main associative patterns in the corpus and disregard the less significant effects [61]. Vectors representing the queries and documents are projected in low dimensional space acquired by reduced singular value decomposition.

LSI begins with a term X document matrix A of dimension rXc and rank r and uses the SVD to decompose it into three matrices $A=USV^T$, where U and V are matrices whose columns are left and right singular vectors of A, S is a diagonal matrix whose diagonal elements are non-negative and ordered in decreasing order. The elements on the main diagonal of S are known as singular values of A and are the square roots of the eigenvalues of A^TA and AA^T [67] [68] [69].Computationally, a K-dimensional SVD of A returns $A_k=U_KS_KV^T_K$, where U_k, $R_k$are first k columns of U and V. In this way the rank of A has been reduced from r to k. using this low rank approximation, the high dimensional documents and query vectors are projected and reduced to low dimensional space [59].

LSI Example: the following is an example taken from [58] to illustrate the latent semantic indexing showing how to find the similarity between query and documents:

Q: “gold silver truck”
D1: “Shipment of gold damaged in a fire”
D2:”Delivery of silver arrived in a silver truck”
D3: “Shipment of gold arrived in a truck”

Then the Matrix A is constructed as term-document matrix and the values represent the occurrence of each term in every document as shown below:

$$\begin{bmatrix} Term & D1 & D2 & D3 \\ \text{A} & 1 & 1 & 1 \\ Arrived & 0 & 1 & 1 \\ Damaged & 1 & 0 & 0 \\ Delivery & 0 & 1 & 0 \\ Fire & 1 & 0 & 0 \\ Gold & 1 & 0 & 1 \\ In & 1 & 1 & 1 \\ Of & 1 & 1 & 1 \\ SHipment & 1 & 0 & 1 \\ Silver & 0 & 2 & 0 \\ truck & 0 & 1 & 1 \end{bmatrix}$$

The singular value decomposition (SVD) then is used on the matrix A to generate three matrixes U, S, and V^T.

$$\begin{bmatrix} -0.4201 & 0.0748 & -0.0460 \\ -0.2995 & -0.2001 & 0.4078 \\ -0.1206 & 0.2749 & 0.4078 \\ -0.1576 & -0.3046 & -0.2006 \\ -0.1206 & 0.2749 & -0.4538 \\ -0.2626 & 0.3794 & 0.1547 \\ -0.4201 & 0.0748 & -0.0460 \\ -0.4201 & 0.0748 & -0.0460 \\ -0.2626 & 0.3794 & 0.1547 \\ -0.3151 & -0.6093 & -0.4013 \\ -0.2995 & -0.2001 & 0.4078 \end{bmatrix}$$

U

$$\begin{bmatrix} 4.0989 & 0 & 0 \\ 0 & 2.3616 & 0 \end{bmatrix}$$

S

$$\begin{bmatrix} -0.4945 & -0.6458 & -0.5817 \\ 0.6492 & -0.7194 & -0.2469 \\ -0.5780 & -0.2556 & 0.775 \end{bmatrix}$$

V^T

After that the K rank is chosen to be 2 then the matrixes will be $A_2 = U_2S_2V^T_2$ as shown below:

$$\begin{bmatrix} -0.4201 & 0.0748 \\ -0.2995 & -0.2001 \\ -0.1206 & 0.2749 \\ -0.1576 & -0.3046 \\ -0.1206 & 0.2749 \\ -0.2626 & 0.3794 \\ -0.4201 & 0.0748 \\ -0.4201 & 0.0748 \\ -0.2626 & 0.3794 \\ -0.3151 & -0.6093 \\ -0.2995 & -0.2001 \end{bmatrix}$$

U_2

$$\begin{bmatrix} 4.0989 & 0 \\ 0 & 2.3616 \end{bmatrix}$$

S_2

$$\begin{bmatrix} -0.4945 & -0.6458 & -0.5817 \\ 0.6492 & -0.7194 & -0.2469 \end{bmatrix}$$

V^{T2}

Now the documents are represented as vectors as follow:

D1 (-0.4945, 0.6492)
D2 (-0.6458, -0.7194)
D3 (-0.5817, 0.2469)

The query vector is then found through applying this equation: $q= q^T U_k S^{-1}{}_k$ as follows:

$$\begin{bmatrix} 000001 & 00011 \\ qT & \end{bmatrix} \begin{bmatrix} -0.4201 & 0.0748 \\ -0.2995 & -0.2001 \\ -0.1206 & 0.2749 \\ -0.1576 & -0.3046 \\ -0.1206 & 0.2749 \\ -0.2626 & 0.3794 \\ -0.4201 & 0.0748 \\ -0.4201 & 0.0748 \\ -0.2626 & 0.3794 \\ -0.3151 & -0.6093 \\ -0.2995 & -0.2001 \end{bmatrix} \begin{bmatrix} 0.2440 & 0 \\ 0 & 0.4234 \end{bmatrix} =q= -0.2140, -0.1821$$

Finally the similarity between query and each document is calculated through finding the cosine value between each document and query vectors. This can be calculated as inner product between vectors as follows:

Similarity (q, d) = $\frac{q \cdot d}{|q||d|}$

Similarity (Q, D1) = $\frac{(-0.2140)(-0.4945)+(-0.1821)(0.6492)}{\sqrt{(-0.2140)^2+(-0.1821)^2}\sqrt{(-0.4945)^2+(-0.6492)^2}} = -0.0541$

Similarity (Q, D2) = $\frac{(-0.2140)(-0.6458)+(-0.1821)(-0.7194)}{\sqrt{(-0.2140)^2+(-0.1821)^2}\sqrt{(-0.6458)^2+(-0.7194)^2}} = 0.9910$

Similarity (Q, D3) = $\frac{(-0.2140)(-0.5817)+(-0.1821)(-0.2469)}{\sqrt{(-0.2140)^2+(-0.1821)^2}\sqrt{(-0.5817)^2+(-0.2469)^2}} = 0.4478$

Information Retrieval In Software Engineering

Information retrieval methods and techniques have been used in many aspects and areas in software engineering. Dilucca et al. in [62] apply various information retrieval and machine learning methods involving classification tree, vector space model, support vectors probabilistic model, and K-nearest neighbor classification to the problem of categorizing and ordering incoming maintenance requests and routing them to particular maintenance team automatically. They use a training set of classified maintenance request correctly; recent incoming maintenance request is evaluated versus the maintenance request in the training set and categorized according to certain distance metric varying with the employed method.

Software reuse is an additional software engineering area that has mostly used information retrieval methods. The acceptance of IR has essentially intended to build reusable software libraries automatically through indexing software components [63] [64] [65] [66]. [63]

present an IR method to gather software libraries automatically based on a free text indexing scheme.

Latent semantic indexing (LSI) has been used by Tairas and Jeff in detecting code clones [67]. Code clones are parts of source code that are copied in many places in a program. Clones are created generally as a result of the copy and paste action of developers where one part of code is copied and pasted into other places.

Program comprehension is another field in software engineering where information retrieval methods have been employed to enhance the process. Poshyvanyket al. measure the coupling between modules through using information retrieval methods to help the developers to comprehend how software modules relate to each other [68]. Revelleet al. present a method for feature location through using structural and textual information to capture feature coupling in object oriented [69]. They use the latent semantic indexing technique to measure how the functions are related to each other. Latent semantic indexing is employed by [70] as well to recreate traceability links among requirements and design artifacts as well as among requirements and test case specification.

Settimi et al. study the usefulness of information retrieval methods for tracing requirements to UML artifacts, code, and test cases. In particular, they evaluate the results attained by applying diverse variants of the vector space model to create links among software artifacts [71].

CHAPTER 4. MUTATION INTEGRATION TOOL

We have developed a Mutation Integration Tool (MIT) to create mutants at the integration level for Java programs. Our tool creates integration mutants based on several new mutation operators. For example: deleting a call from a chain of calls, swapping a call in a chain of calls with other methods or functions, duplicating calls, swapping methods' parameters, changing the value of methods' parameters and changing the value of methods' returns. The parameter types that we use in changing values are: integer, double, long, short, byte, boolean, float and string.

The Graphical User Interface Of Mutation Integration Tool

The graphical user interface of the MIT is composed from input and output sections:
Input section: this section consists of three buttons and nineteen check boxes. The user selects the Java file by pressing the "Browse" button. The nineteen check boxes are listed where the user selects the type of mutants (swap parameters, duplicate calls etc.) and the "Create Mutant" Button generates mutants for the selected Java file based on the selected mutant types. "Create Chain of calls Mutant" button is used to create mutants based on the chain of calls.

Output section: this section consists of one test area and eighteen Labels. The text area shows the full path of the selected Java file. The labels show the number of created mutants of each type.

For example, if the user clicks on the button labeled "Browse" and selects the "Elevator" class (Figure 1), a textbox displays the path of the file "Elevator". After the user selects "ALL" (Figure 2) for the types of mutants to be created then the MIT displays the number of created mutants for each mutation type (Figure 3) and the mutants files are saved in the project folder (Figure 4). Another example, if the user clicks on the button labeled "Create Chain of calls

Mutant" the open dialog will appear to select the source code of the whole application as shown in Figure 5. After the user selects the source code then the MIT displays the number of created mutants (deleted call from the chain of calls and the swap call in the chain of calls with another method or function) as shown in Figure 6.

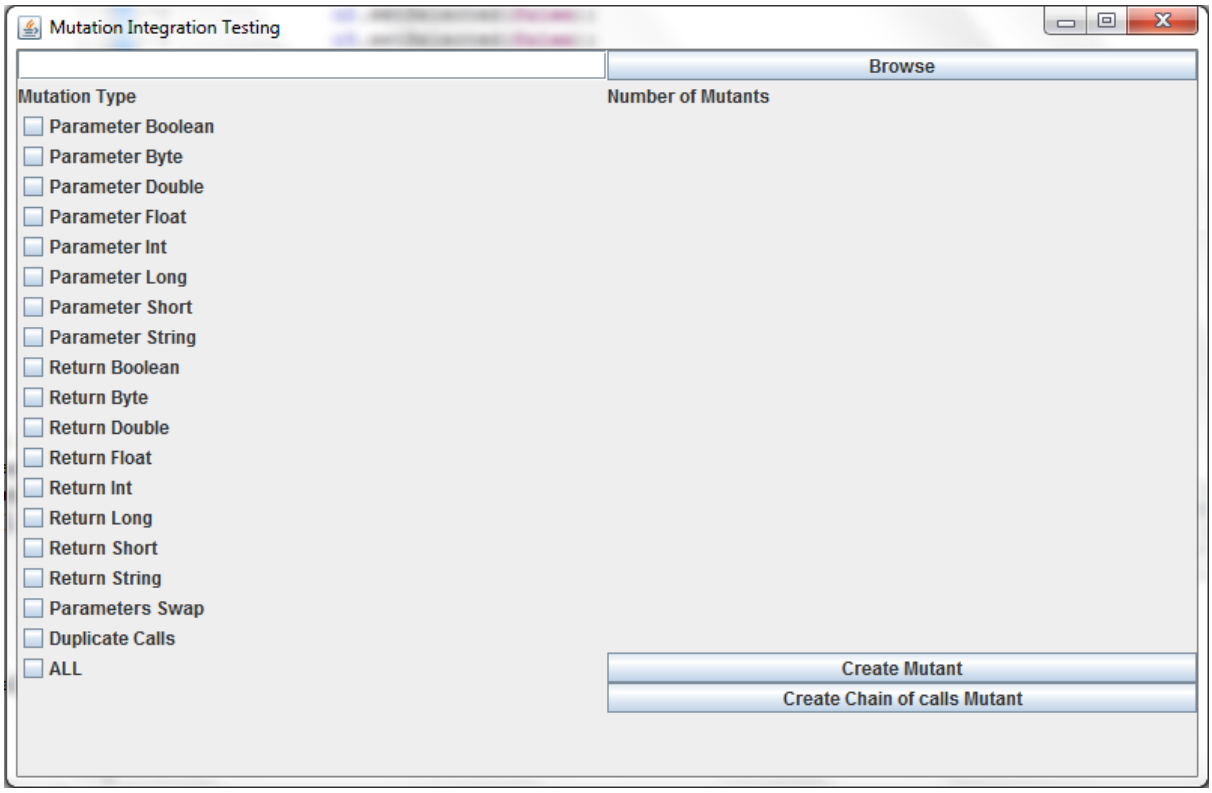

Figure 4.1. Graphical User Interface Of MIT

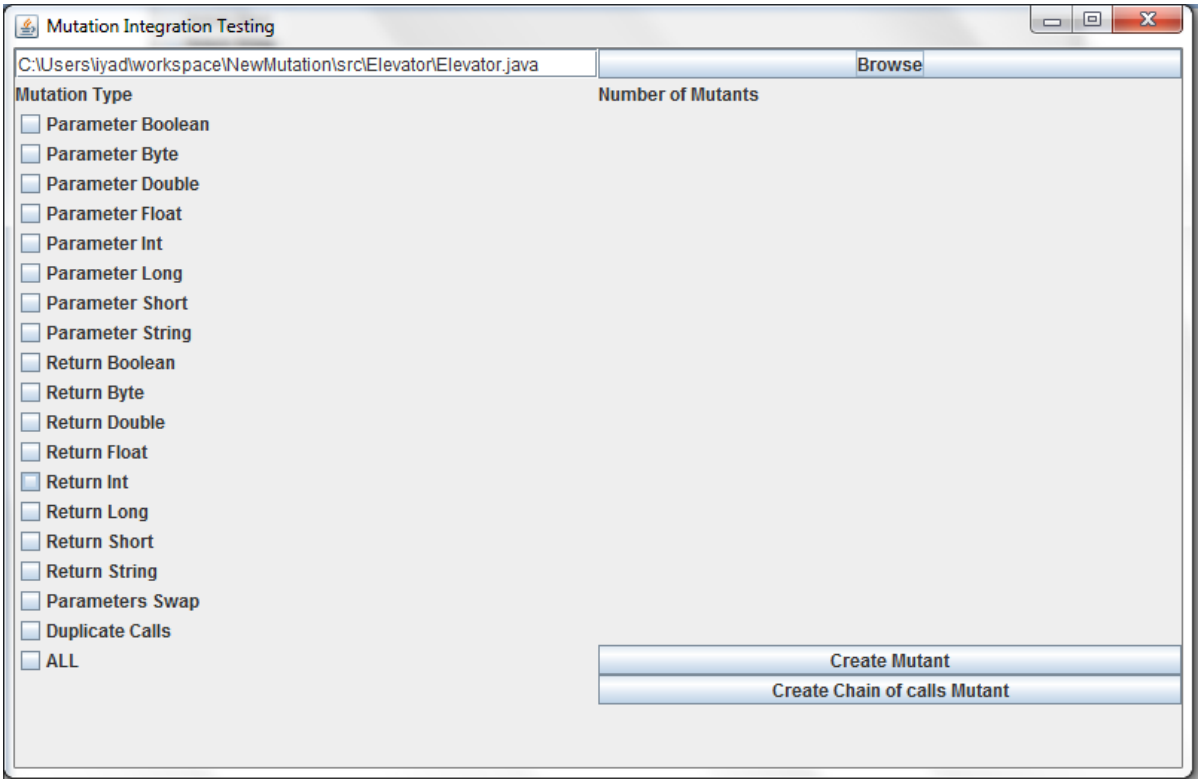

Figure 4.2. Illustration Of Class Selection

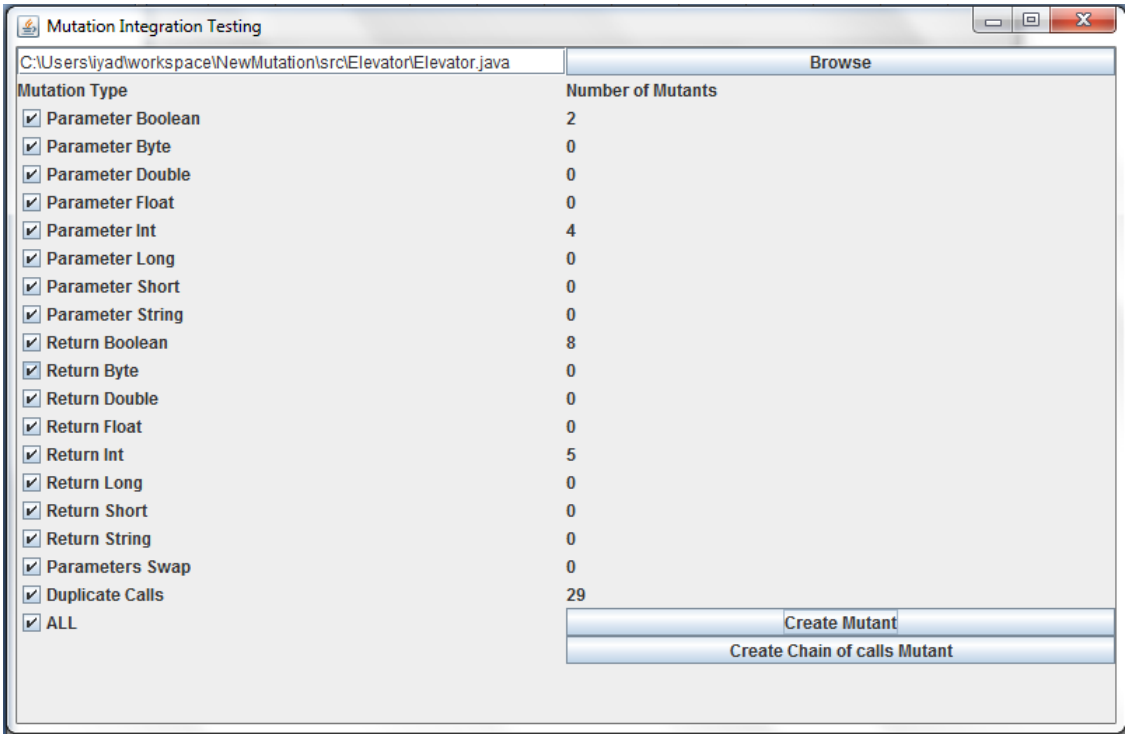

Figure 4.3. Illustration Of The Number Of Created Mutants At Class Level

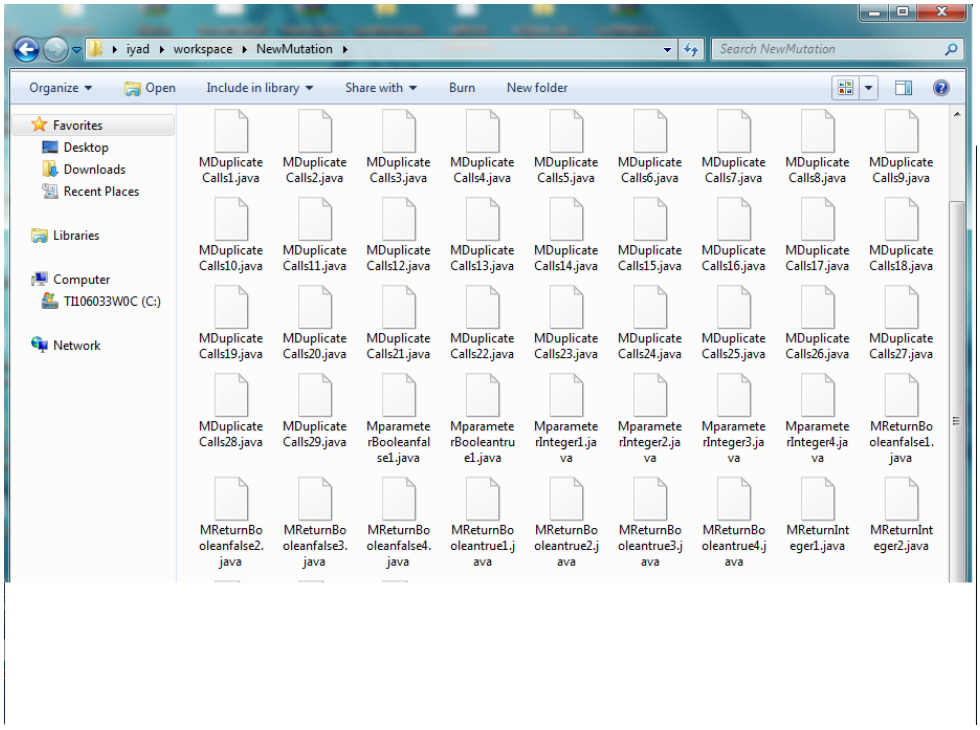

Figure 4.4. Illustration Of The Created Mutants Files

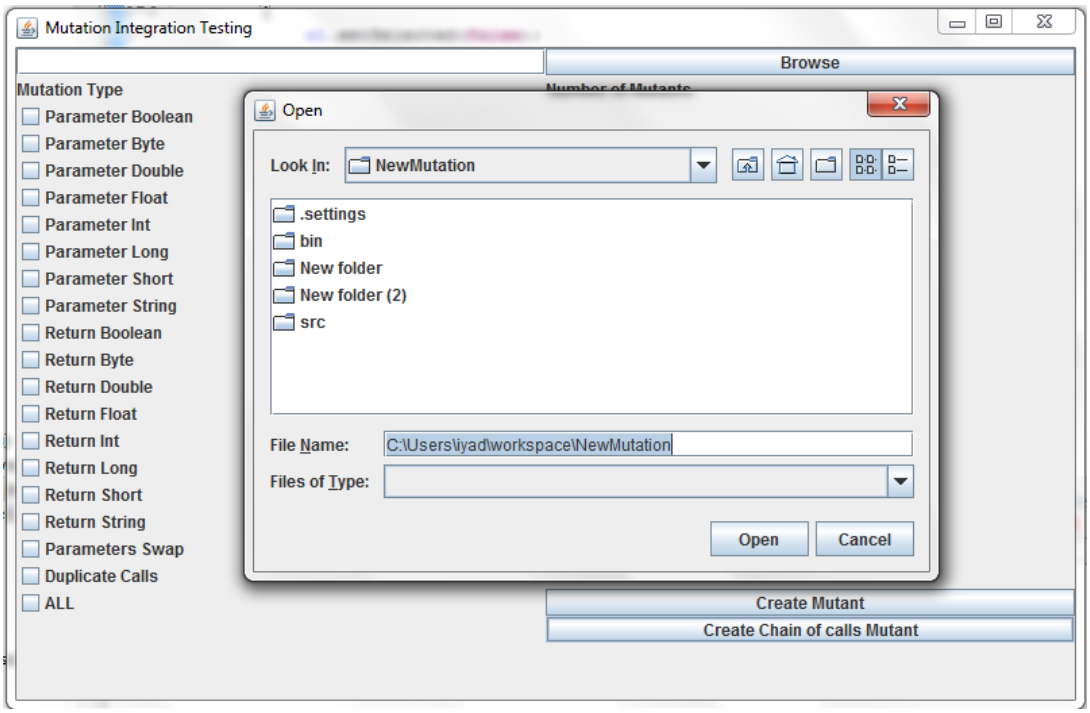

Figure 4.5. Illustration Of Selection A Java Project

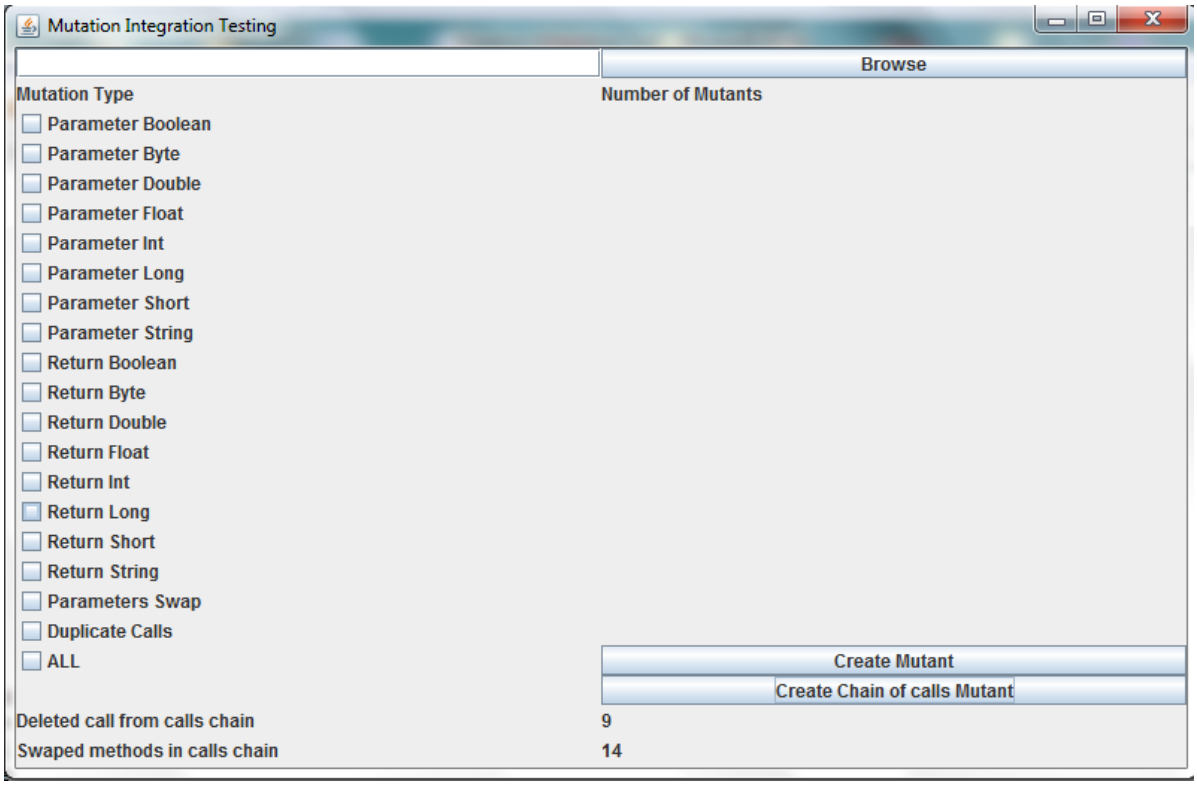

Figure 4.6. Illustration Of The Number Of Chain Of Calls Created Mutants

Integration Mutation Testing Operators

MIT creates mutants based on the following integration mutation testing operators:-

1. Swap parameters: this will swap the parameters in the method declaration if there is more than one parameterwith the same data type. For example:

   Public void calculate (intnum_of_hours, inthour_cost){

   …

   }

   This operator will create a mutant,

   Public void calculate (inthour_cost, intnum_of_hours){

   …

   }

2. Duplicate calling: the mutant will call the same method twice instead of the original one time. For example:

   Public void method1 () {

   CarClass car = new CarClass()

   Car.calculateMilage();

   …

   }

   The mutant will be:

   Public void method () {

   CarClass car = new CarClass()

   Car.calculateMilage();

   Car.calculateMilage();

   ...

   }

3. Return String: for the methods in a specific class that have “String” as their return value the mutant will return a “null” value.

   For example :

   Public String getname(){

   Return name;

   }

   The mutant will be:

   Public String getname(){

   Return null ;

}

4. Return Integer: for the methods in a specific class that have “int” as their return value the mutant will return a “0” value.

   For example

   Public intgetAge (){

   Return age;

   }

   The mutant will be:

   Public intgetAge(){

   Return 0;

   }

5. Return Double: for the methods in a specific class that have “double” as their return value the mutant will return a “0.0” value.

   For example

   Public double getSalary(){

   Return salary;

   }

   The mutant will be:

   Public double getSalary(){

   Return 0.0;

   }

6. Return Float: for the methods in a specific class that have “float” as their return value the mutant will return a “0.0” value.

   For example

   Public floatgetFloatnum(){

   Return floatnum;

   }

   The mutant will be:

   Public floatgetFloatnum (){

   Return 0.0;

   }

7. Return Long: for the methods in a specific class that have “long” as their return value the mutant will return a “0” value.

   For example

   Public longgetLongnum(){

   Return longnum;

   }

   The mutant will be:

   Public longgetLongnum (){

   Return 0;

   }

8. Return Byte: for the methods in a specific class that have “byte” as their return value the mutant will return a “0” value.

   For example

Public bytegetByte(){

Return bytenum;

}

The mutant will be:

Public bytegetByte (){

Return 0;

}

9. Return Short: for the methods in a specific class that have “short” as their return value the mutant will return a “0” value.

   For example

   Public shortgetShortNum(){

   Return shortnum;

   }

   The mutant will be:

   Public shortgetShortNum (){

   Return 0;

   }

10. Return Boolean: for the methods in a specific class that have “boolean” as their return value, two mutants will be created. One mutant will return a “false” value and the other mutant will return a “true” value.

    For example

    Public booleanisEmpty(){

    Return size==0;

}

The mutants will be:

Public booleanisEmpty (){

Return true;

}

Public booleanisEmpty (){

Return false;

}

11. Boolean parameters: the mutation method for Boolean parameters will change the parameters into fixed Boolean values either true or false.

    For Example:

    Public void method1(Boolean is_ready)

    {

    If (is_ready){ …….}

    }

    The mutants will be:

    Public void method1(Boolean is_ready)

    {

    is_ready= true;

    If (is_ready){ …….}

    }

    Public void method1(Boolean is_ready)

    {

is_ready= false;

If (is_ready){ …….}

}

12. String parameters: the mutation method for String parameters will change the parameters into fixed String value “null”.

    For Example:

    Public void method1(String x)

    {

    Function(x);

    …

    }

    The mutant will be:

    Public void method1(String x)

    {

     x= “null”;

    Function(x);

    …..

    }

13. Integer parameters: the mutation method for Integer parameters will change the parameters into integer value “0”

    For Example:

    Public void method1(int x)

    {

Function(x);

…

}

The mutant will be:

Public void method1(int x)

{

x= 0;

Function(x);

…..

}

14. Double parameters: the mutation method for double parameters will change the parameters into double value "0.0"

For Example:

Public void method1(double x)

{

Function(x);

…

}

The mutant will be:

Public void method1(double x)

{

x= 0.0;

Function(x);

…..

}

15. Float parameters: the mutation method for float parameters will change the parameters into float value “0.0”

    For Example:

    Public void method1(float x)

    {

    Function(x);

    …

    }

    The mutant will be:

    Public void method1(float x)

    {

     x= 0.0;

    Function(x);

    …..

    }

16. Long parameters: the mutation method for long parameters will change the parameters into long value “0”

    For Example:

    Public void method1(long x)

    {

    Function(x);

…

}

The mutant will be:

Public void method1(long x)

{

x= 0;

Function(x);

…..

}

17. Short parameters: the mutation method for short parameters will change the parameters into short value “0”

For Example:

Public void method1(short x)

{

Function(x);

…

}

The mutant will be:

Public void method1(short x)

{

x= 0;

Function(x);

…..

}

18. Byte parameters: the mutation method for byte parameters will change the parameters into Byte value “0”

    For Example:

    Public void method1(Byte x)

    {

    Function(x);

    …

    }

    The mutant will be:

    Public void method1(Byte x)

    {

     x= 0;

    Function(x);

    …..

    }

19. Chain call deletion: this mutation removes a call in a chain of calls. For example if we have three classes A, B, and C. where A interacts with B through two methods m1 and m2, and class B interacts with class C through method m3 as shown in the following figure.

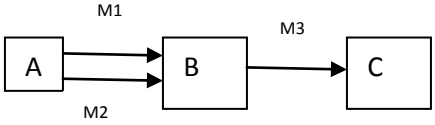

Figure 4.7. Chain Of Calls

This leads to create three mutants as following:

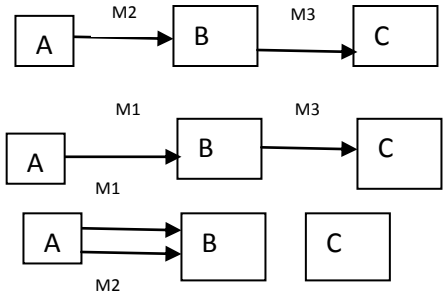

Figure 4.8. Example Of Mutants (Deleting A Call In A Chain Of Calls)

20. Swap methods: if there are two or more methods in a class with the same type of parameter and number and the return type is quite similar (can be cast). This mutation operator will replace a method with another one.

    For example: if there are two classes A and B, where class A has three methods as follows:

    public double getBalance(){ …}

    public double calculate(){ …}

    public double getSalary(){ …}

    and in class B there is a method named compute that interacts with class A:

    class B{ …

    public void compute(){…

    A.getBalance();

    …}

    Then the Swap method will replace the getBalance() method which is used in the compute method in class B with calculate and getSalary respectively and create two mutants as:

```
class B{ …
public void compute(){…
A.calculate ();
…}
class B{ …
public void compute(){…
A.getSalary ();
…}
```

The Architecture of Mutation Integration Tool

The MIT is developed using the Java language and creates various integration mutants which are illustrated in the next sections. The MIT consistsofsix main packages as shown in Figure 4.8

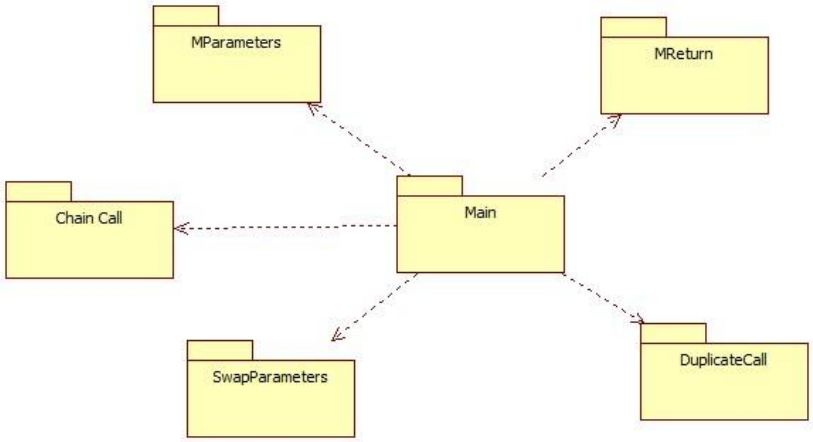

Figure 4.8. Packages In MIT

The Main package consists of three classes, “Method”, “Comment” and “MIT” as shown in Figure 4.9. The DuplicateCall package consists only of one class “DupliacteCall” as shown

Figure 4.10. The SwapParameters package consists of two classes: “Parameters” and “swapParameters” as shown in Figure 4.11. The MParameters package consists of ten classes:“MParameter”, “MParameterByte”, “MParameterInteger”, “MParameterDouble”, “MParameterString”, “MparameterBooleanTrue”, “MparameterBooleanFalse”, “MParameterFloat”, “MparameterShort”, and “MparameterLong” as shown in Figure 4.12. The MReturn package consists of nine classes: “MReturnString”, “MReturnBooleanFalse”, “MReturnBooleanTrue”, ”MReturnLong”, “MReturnShort”, “MRetrunDouble”, “MReturnInteger”, “MReturnByte” and “MReturnFloat” as shown in Figure 4.13. The Chain call package consists of two classes: “ChainMutant” and “ClassMethods” as shown in Figure 4.14.

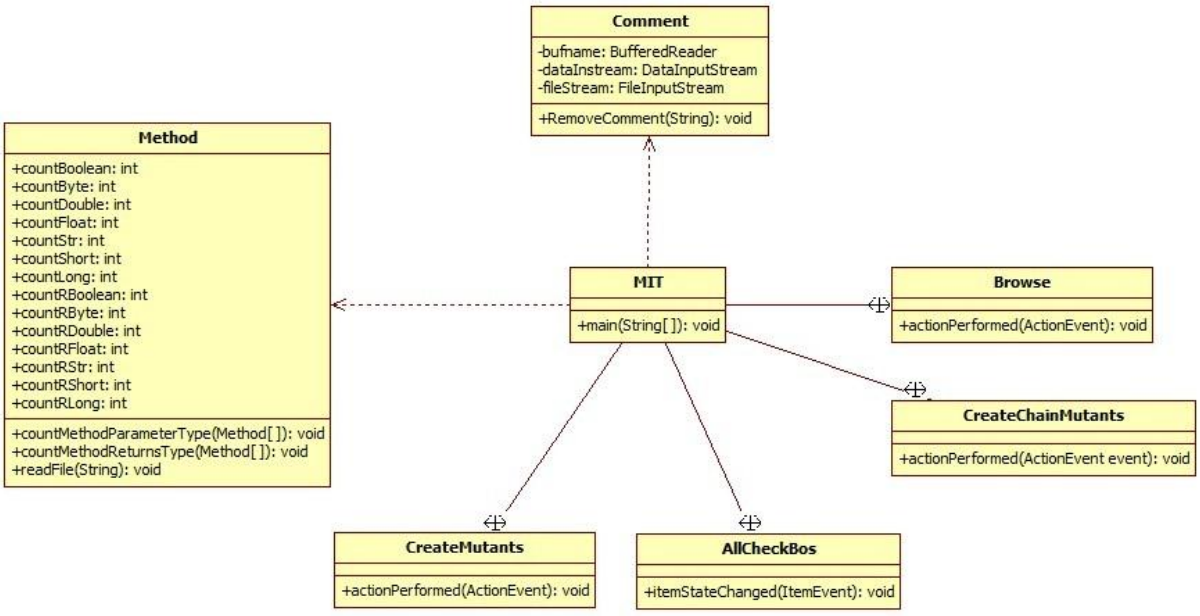

Figure 4.9. The Classes In Main Package

The method of the Comment class:

- RemoveComment(String): Reads the Java file line by line and removes all the comments and blank lines.

The methods of the Method Class:

- CountMethodParametersTypes(Method []): counts the number of Boolean parameters, String parameters, Long parameters, Short parameters, Integer parameters, Double parameters, and Byte parameters for the all methods in the Java class.

- CountMethodReturnsTypes(Method[]): counts the number of return Boolean methods, return String methods, return Integer methods, return Double methods, return Float methods, return Byte methods, return Long methods, and return Short methods, for all methods in the Java class.

- ReadFile(String): reads the Java file in order to do the above calculations.

The methods of the MIT Class

- main (String[]): creates the frame of the graphical user interface and sets the size of the frame.
- ItemStateChanged (ItemEvent) in the inner class "ALLCheckBox": controls the states of every check box in the frame based on the state of ALL check boxes. When the user selects the ALL check box, every check box in the frame will be selected. When the user deselects the ALL check box, every check box in the frame will be deselected.
- ActionPerformed (ActionEvent) in the inner class "Browse": controls the show of open dialog which lets the user select Java file and get the file path, filename and set the file path in the text box.

- ActionPerformed (ActionEvent) in the inner class “CreateMutants”: is the most important method. This triggers the main methods in the other classes.

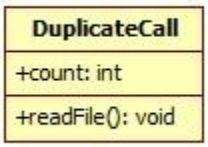

Figure 4.10. The Class In DuplicateCall Package

Method of DuplicateCall class

- ReadFile(): reads the Java file line by line and creates a duplicate line if the line is representing method calling.

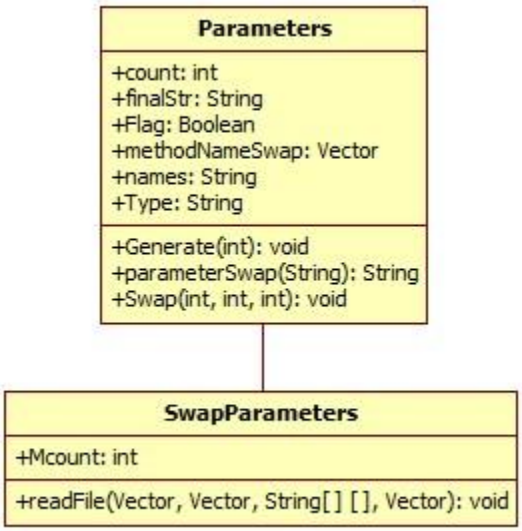

Figure 4.11. The Classes In The SwapParameters Package

Methods of Parameters class

- Generate(int): it produces the method parameters after doing swapping.

- parameterSwap(String) it is responsible for determining if the method is valid for swapping; having two or more parameters with the same data type.
- Swap(int, int, int): it is responsible for performing swapping.

Method of ParameterSwap class

- ReadFile(Vector, Vector, String [][],Vector): reads the Java file line by line and when there is a method signature call the methods in the parameters class for swapping.

Methods of Mreturn class

-Mutation(String,BufferedWriter,Scanner,int): this method is responsible for creating mutants based on the return type.

- processmethod(Vector, String,Vector,Vector,BufferedWriter): it is responsible for deriving the method name from the line, and to determine if the method returns value or not. This method is used in the derived classes in the same way.

- readFile(Vector, Vector, String [][],Vector, String,int): reads the Java file line by line and checks if the line represents a method or not. This method is used in the derived classes in the same way.

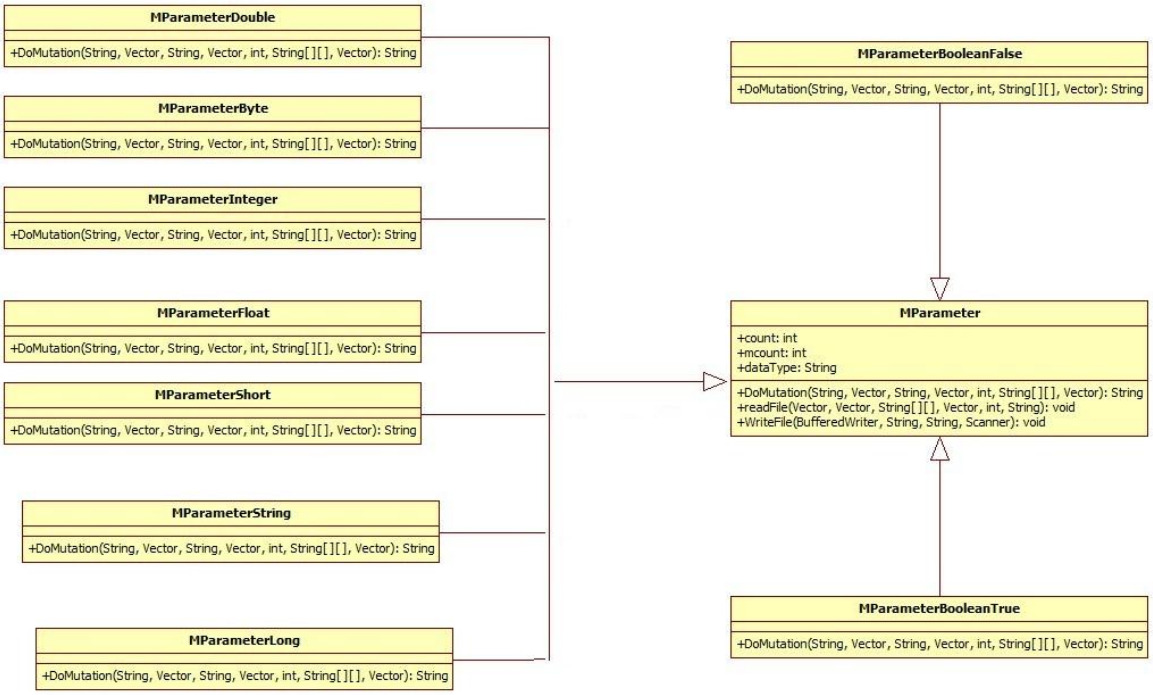

Figure 4.12.The Classes MParameters Package

Methods of MParameters class

- DoMutation(String,Vector, String,Vector,int, String [][], Vector): it is responsible to create mutants based on the types of the methods' parameters.
- readFile(Vector, Vector, String [][],Vector, String,int, String): reads the Java file line by line and checks if the line represents a method or not. This method is used in the derived classes in the same way.
- writefile(BufferedWriter, String, String, Scanner): it is responsible to retrieve the method body after doing mutation based on the parameters' values. This method is used in the derived classes in the same way.

Method of MParameterBooleantrue class

- DoMutation(String,Vector, String,Vector,int, String [][], Vector): it is responsible for each method that has a parameter of type Boolean to set its value to true before using it.

Method of MParameterBooleanfalse class

- DoMutation(String,Vector, String,Vector,int, String [][], Vector): it is responsible for each method that has a parameter of type Boolean to set its value to false before using it.

Method of MParameterByte class

- DoMutation(String,Vector, String,Vector,int, String [][], Vector): it is responsible for each method that has a parameter of type Byte to set its value to 0 before using it.

Method of MParameterDouble class

- DoMutation(String,Vector, String,Vector,int, String [][], Vector): it is responsible for each method that has a parameter of type double to set its value to 0.0 before using it.

Method of MParameterFloat class

- DoMutation(String,Vector, String,Vector,int, String [][], Vector): it is responsible for each method that has a parameter of type Float to set its value to 0 before using it.

Method of MParameterInt class

- DoMutation(String,Vector, String,Vector,int, String [][], Vector): it is responsible for each method that has a parameter of type Integer to set its value to 0 before using it.

Method of MParameterLong class

- DoMutation(String,Vector, String,Vector,int, String [][], Vector): it is responsible for each method that has a parameter of type Long to set its value to 0 before using it.

Method of MParameterShort class

- DoMutation(String,Vector, String,Vector,int, String [][], Vector): it is responsible for each method that has a parameter of type Short to set its value to 0 before using it.

Method of MParameterString class

- DoMutation(String,Vector, String,Vector,int, String [][], Vector): it is responsible for each method that has a parameter of type String to set its value to null before using it.

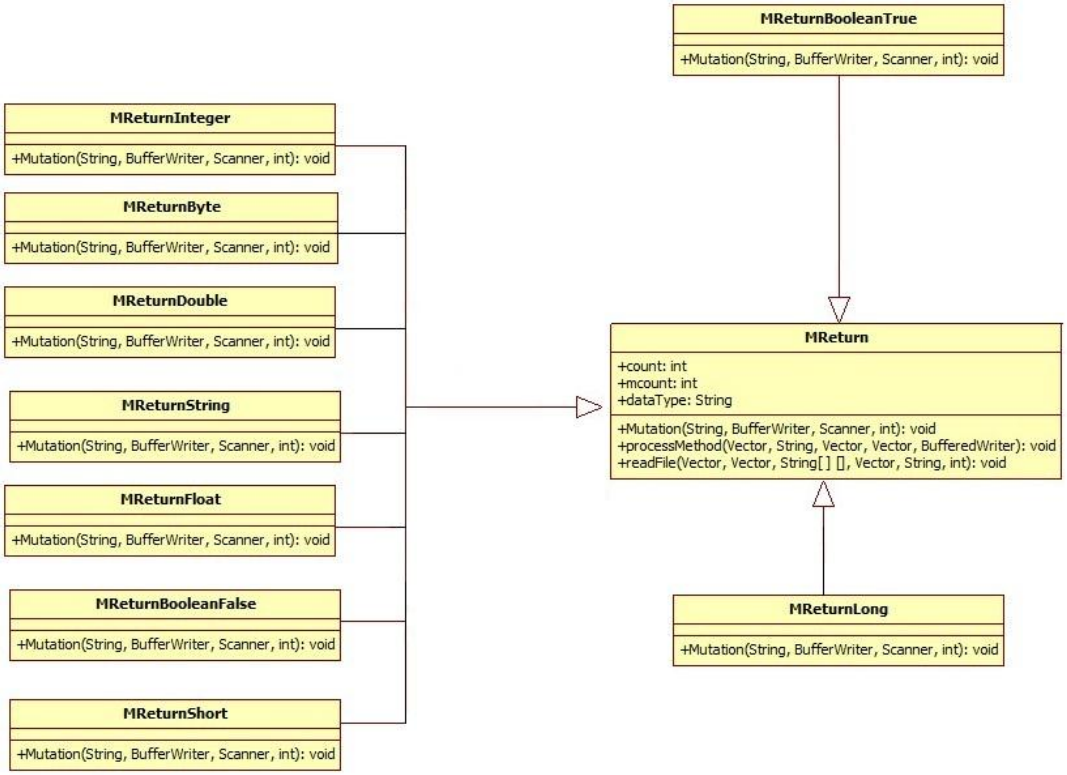

Figure 4.13. The Classes In MReturns Package

Method of MReturnBooleanfalse class

- Mutation(String,BufferedWriter,Scanner,int): this method is responsible for each method that return value of type Boolean to set its return value to false.

Method of MReturnBooleantrue class

- Mutation(String,BufferedWriter,Scanner,int): this method is responsible for each method that return value of type Boolean to set its return value to true.

Method of MReturnByte class

- Mutation(String,BufferedWriter,Scanner,int): this method is responsible for each method that returns value of type Byte to set its return value to 0.

Method of MReturnDouble class

- Mutation(String,BufferedWriter,Scanner,int): this method is responsible for each method that returns value of type Double to set its return value to 0.0.

Method of MReturnFloat class

- Mutation(String,BufferedWriter,Scanner,int): this method is responsible for each method that returns value of type Float to set its return value to 0.0.

Method of MReturnIntclass

- Mutation(String,BufferedWriter,Scanner,int): this method is responsible for each method that returns value of type Integer to set its return value to 0.

Method of MReturnLong class

- Mutation(String,BufferedWriter,Scanner,int): this method is responsible for each method that returns value of type Long to set its return value to 0.

Method of MReturnShort class

- Mutation(String,BufferedWriter,Scanner,int): this method is responsible for each method that returns value of type Short to set its return value to 0.

Method of MReturnString class

- Mutation(String,BufferedWriter,Scanner,int): this method is responsible for each method that returns value of type String to set its return value to null.

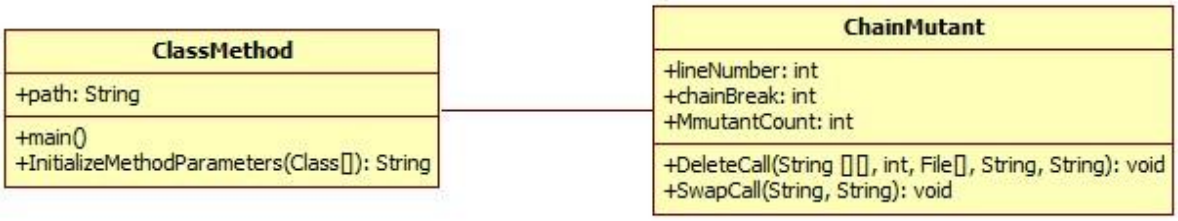

Figure 4.14. The Classes In Chain Call Package

Methods of ClassMethod

-main (): this method is responsible for generating text files that contain the public methods for each class in the application.

- InitializeMethodParameters (Class[]): this method is responsible to assign initial values for the methods parameters.

Methods of ChainMutant

- DeleteCall(String [][],int, file[], String, String): this method creates mutant through deleting a call from a chain of calls. The call should be alone which means it should not be a part of any assignment or use such as in loops or conditions or others.
- SwapCall(String, String): this method creates mutant through replacing a call in a chain of calls with another method or function from the same class which has the same return type and public access modifier.

CHAPTER 5. EMPIRICAL STUDY

In order to evaluate and assess our approach, we have carried out a sequence of experiments based on 10 open source Java applications obtained from different repositories.

In this chapter, we describe the 10 Java applications that we have used in the experiments and present the values of class pair weights for the whole applications. In addition we show the percentage of test cases of each class in every application along with the number of test cases that should go for each class as well. Moreover we present the results of mutation testing and explain them. This chapter also presents the number of created mutants for each application along with the mutation score which is the percentage of killed mutants.

Application Under Test

Table 5.1 lists all the applications that we use in our experiments shows how many classes in each application.

- The “Black Jack” application consists of ten classes: “BustedExceptio”, “Card”, “Dealer”, “DealTemplate”, “FileFacade”, “FileUser”, “Hand”, “LogicFacade”, “Player”, and “User”.
- The “CruiseControl” application consists of four classes: “CarSimulator”, “Controller”, “CruiseControl”, and “SpeedControl”.
- The “Linked List” application consists of four classes: “MyLinkedList”, “MyLinkedListItr”, “MyListBuilder”, and “MyListNode”.
- The “Telephone” application consists of five classes: “RemoteTelNums”, “Setup”, “TelephoneApp”, “TelNums” and “TelNumsProxy”.
- The “Word Processor” application consists of six classes: “CutCommand”, “Document”, “DocumentCommand”, “PasteCommand”, “UndoCommand” and “WordProcessor”.

-The “Bank” application consists of eight classes: “Account”, “AttemptToAddBadBankingComponentException”, “CDAccount”, “CheckAccount”, “Client”, “Customer”, “SavingAcount” and “Setup”.

- The “Elevator” application consists of eight classes: “ArrivalSensor”, “Elevator”, “ElevatorControl”, “ElevatorGroup”, “ElevatorInterface”, “Floor”, “FloorControl” and “FloorInterface”.

- The “Phone Directory” application consists of three classes: “Person”, “PhoneList” and “phoneNumbers”.

- The “Computer” application consists of five classes: “Client”, “Computer”, “CPU”, “NetSystem” and “RAM”.

- The “Coffee Maker” application consists of six classes: “CoffeeMaker”, “RecipeException”, “Inventory”, “InventoryException”, “Recipe” and “RecipeBook”.

Application	Number of classes	Classes
BlackJack	10	BustedException
		Card
		Dealer
		DealTemplate
		FileFacade
		FileUser
		Hand
		LogicFacade
		Player
		User
CruiseControl	4	CarSimulator
		Controller
		CruiseControl
		SpeedControl

Table 5.1. Subject Of The Experiments

LinkedList	4	MyLinkedList
		MyLinkedListItr
		MyListBuilder
		MyListNode
Telephone	5	RemoteTelNums
		Setup
		TelephoneApp
		TelNums
		TelNumsProxy
WordProcessor	6	CutCommand
		Document
		DocumentCommand
		PasteCommand
		UndoCommand
		WordProcessor
Application	Number of classes	Classes
Bank	8	Account
		AttemptToAddBadBankingComponentException
		CDAccount
		CheckAccount
		Client
		Customer
		SavingAcount
		Setup
Elevator	8	ArrivalSensor
		Elevator
		ElevatorControl
		ElevatorGroup
		ElevatorInterface
		Floor
		FloorControl
		FloorInterface

Table 5.1. (Continued)

Phone Directory	3	Person
		PhoneList
		phoneNumbers
Computer	5	Client
		Computer
		CPU
		NetSystem
		RAM
CoffeeMaker	6	CoffeeMaker
		RecipeException
		Inventory
		InventoryException
		Recipe
		RecipeBook

Table 5.1. (Continued)

Class Pair Weight And Test Cases Calculation

Table 5.2 lists the class pair weight for the "Bank" application, test cases percentage for each class, and the number of test cases for each class in the application. The "Account" class has the highest weight among the classes so the majority of test cases, 11 test cases, will go to the "Account" class. On the other hand, the "Setup" class has no class pair weight. In this case, we ensure it has one test case.

Class Name	Weight	Test Cases Percentage	Test Cases Count
Account	11.7638	25.55%	11
CheckAccount	7.017	15.24%	7
Customer	7.719	16.77%	7
CDAccount	6.679	14.51%	6
SavingsAccount	6.781	14.73%	6
Client	4.324	9.39%	4
AttemptToAddBadBankingComponentException	1.757	3.82%	2
Setup	0	0	1

Table 5.2. Bank Test Cases Calculations

Table 5.3 lists the class pair weight, Test cases percentage and the number of test cases for each class in the “Coffee Maker” application. The “Recipe” class has the highest weight “13.7”. The “inventory exception” class has the lowest weight.

Class Name	Weight	Test Cases Percentage	Test Cases Count
Recipe	13.7	27.7%	10
CoffeeMaker	9.762	19.74%	7
RecipeBook	8.435	17.1%	6
Main	8	16.18%	6
Inventory	3.861	7.81%	3
RecipeException	2.91	5.88%	3
InventoryException	2.79	5.64%	2

Table 5.3. Coffee Maker Test Cases Calculations

The Table 5.4 lists the class pair weight, Test cases percentage and the number of test cases for each class in the “Computer” application. The “NetSystem” class has the highest test cases percentage “24.93”. On the other hand, the percentage of test cases for the “Setup” class is zero; however, we ensure that each class at least has one test case.

Class Name	Weight	Test Cases Percentage	Test Cases Count
NetSystem	6.684	24.93%	8
CPU	4.063	15.15%	6
Component	3.83	14.28%	5
RAM	4.007	14.94%	5
Client	1.12	4.18%	2
computer	7.112	26.52%	1
Setup	0	0	1

Table 5.4. Computer Test Cases Calculations

Table 5.5 lists the class pair weight, Test cases percentage and the number of test cases for each class in the “Cruise Control” application. The “CarSimulator” class has the highest weight “7.619”. The “CruiseControl” class has the lowest weight “0”.

Class Name	Weight	Test Cases Percentage	Test Cases Count
CarSimulator	7.619	51.58%	11
Controller	4.468	30.25%	7
SpeedControl	2.683	18.17%	4
CruiseControl	0	0	1

Table 5.5. Cruise Control Test Cases Calculations

Table 5.6 lists the class pair weight, Test cases percentage and the number of test cases for each class in the "Elevator" application. The "Elevator" class has almost third of test cases of the application while the "ElevatorGroup" class has just one test case.

Class name	Weight	Test Cases Percentage	Test Cases Count
Elevator	27.407	29.97%	12
Floor	19.384	21.2%	9
ElevatorControl	12.715	13.9%	6
ElevatorInterface	9.469	10.36%	5
FloorInterface	7.657	8.37%	3
FloorControl	6.894	7.54%	3
ArrivalSensor	4.993	5.46%	2
ElevatorGroup	2.927	3.2%	1

Table 5.6. Elevator Test Cases Calculations

Table 5.7 lists the class pair weight, Test cases percentage and the number of test cases for each class in the "Linked List" application.

Class Name	Weight	Test Cases Percentage	Test Cases Count
MyListNode	7.732	34.85%	7
MyLinkedList	6.496	29.28%	6
MyLinkedListItr	5.103	23%	5
MyListBuilder	2.857	12.88%	3

Table 5.7. Linked List Test Cases Calculations

Table 5.8 lists the class pair weight, Test cases percentage and the number of test cases for each class in the “Phone Directory” application.

Class Name	Weight	Test Cases Percentage	Test Cases Count
Person	7.854	77.24%	12
phonelist	2.314	22.78%	4
phoneNumbers	0	0	1

Table 5.8. Phone Directory Test Cases Calculations

Table 5.9 lists the class pair weight, Test cases percentage and the number of test cases for each class in the “Telephone” application.

Class Name	Weight	Test Cases Percentage	Test Cases Count
TelNums	5.924	41.78%	11
TelNumsProxy	3.822	26.96%	7
RemoteTelNums	2.341	16.51%	5
TelephoneApp	2.093	14.76%	4
Setup	0	0	1

Table 5.9. Telephone Test Cases Calculations

The Table 5.10 lists the class pair weight, Test cases percentage and the number of test cases for each class in the “Word Processor” application.

Class Name	Weight	Test Cases Percentage	Test Cases Count
DocumentCommand	16.78	23.45%	9
CutCommand	13.951	19.5%	7
PasteCommand	13.305	18.59%	7
UndoCommand	11.165	15.6%	6
WordProcessor	10.704	14.96%	6
Document	5.662	7.91%	3

Table 5.10. Word Processor Test Case Calculations

The Table 5.11 lists the class pair weight, Test cases percentage and the number of test cases for each class in the “Black Jack” application.

Class Name	Weight	Test Cases Percentage	Test Cases Count
Hand	18.421	27.09%	13
Card	14.492	21.31%	10
FileUser	7.029	10.34%	7
User	9.892	14.55%	7
Dealer	5.4	7.94%	4
LogicFacade	5.4017	07.94%	4
BustedException	1.75	2.57%	2
DealTemplate	2.499	3.67%	2
FileFacade	2.825	4.15%	2
Player	0.292	0.43%	1

Table 5.11. Black Jack Test Cases Calculations

Developed Test Cases

Table 5.12 shows the number of developed test cases and compares it with the number of test cases that should go for each application. From the table we can see that we developed less than 50% for seven applications: “linkedList”, “computer”, “WordProcessor”, “CruiseContol”, “BlackJack”, “CoffeeMaker” and “Elevator” and we killed more than 80% of the mutants. We stopped developing test cases when we killed 80% of the mutants.

Application Name	Number of Developed Test Cases	Number of Test Cases based on the Calculations	Percentage of Developed Test Cases
linkedList	5	28	17.86%
computer	9	21	42.86%
WordProcessor	13	37	35.14%
CruiseContol	10	44	22.73%
BlackJack	13	38	34.21%
CoffeeMaker	11	28	39.29%
Elevator	21	52	40.39%
Bank	31	41	75.61%
phoneDirectory	15	17	88.24%
Telephone	12	23	52.17%

Table 5.12. Number Of Developed Test Cases

Mutation Testing Results

We use mutation testing in order to evaluate our approach. Table 5.13 lists the percentage of killed mutants. The results represents the mutants generated based on the chain of calls among classes together in the application. The "linked List" and "Computer" application had all the mutants killed with just a few test cases. In the "WordProcessor" application, there are 22 mutants and 20 of them are killed by just 13 test cases. "Elevator" and "Coffee Maker" applications have almost the same percentage of killed mutants 85.12% and 85.19% respectively. Four applications had more than 90% of the mutants killed: "WordProcessor", "BlackJack", "Bank" and "Telephone". So we can see that our test cases killed 80% or more of the mutants.

Application Name	Number of Mutants	Killed Mutants	Live Mutants	Percentage of killed Mutant
linkedList	3	3	0	100%
computer	9	9	0	100%
WordProcessor	22	20	2	90.91%
CruiseContol	31	26	5	83.87%
BlackJack	22	20	2	90.91%
CoffeeMaker	27	23	4	85.19%
Elevator	47	40	7	85.12%
Bank	12	11	1	91.67%
phoneDirectory	25	22	3	88%
Telephone	15	14	1	93.33%

Table 5.13. Chain Calls Mutants Results

Table 5.14 lists the percentage of killed mutant which are created based on the class, specifically methods' parameters, method return types and duplicate calls within the class. In the "computer" application 96% of the mutants are killed. 75% of the mutants are killed in the "Linked List" application. The percentage of killed mutants of "Coffee Maker", "Bank", "Word Processor", "Telephone", "Black Jack", "Elevator", "Phone Directory", and "Cruise Control" are 76.32 %, 76.92%, 71.43%, 62.5%, 59.57%, 57.78%, 59.1% and 33.33% respectively. The big differences in the results between percentage of killed mutants in Table 12 and Table 13 are because of the test cases. The test cases are created to mainly reveal the interaction errors between modules only not the interaction within modules. Since the created mutants at class

level represent all the interaction within the module and among modules while the created mutants based on the chain of calls represent just the interaction among modules, the results in table 5.13 are much better than the results in Table 5.14.

Application Name	Number of Mutants	Killed Mutants	Live Mutants	Percentage of Killed Mutant
Word Processor	49	35	14	71.43%
phone Directory	110	65	45	59.1%
telephone	24	15	9	62.5%
Linked List	16	12	4	75%
elevator	135	78	57	57.78%
cruise Control	39	13	26	33.33%
computer	25	24	1	96%
Coffee Maker	76	58	18	76.32%
black jack	94	56	38	59.57%
bank	26	20	6	76.92%

Table 5.14. Class Mutants Results

Inner Mutants

During the experiments, we have noticed that the class level mutants are reflecting interactions mutants within the class itself and with other classes which come with the built in packages such as String, Integer, Hash table, and others. So we called these kinds of mutants “Inner mutants” and remove them from our calculations because these mutants will be killed at the unit level of testing.

Table 5.15 shows the percentage of mutants killed excluding the inner mutants. The results show that we have killed more than 80% of the mutants after removing inner mutants for the whole applications except juts the “Elevator” application 78.79%. We have killed 100 % of the mutants in the “LinkedList” application. Three applications: ”Computer”, “CoffeeMaker” and “bank” achieved 90% in killing mutants. The percentage of killed mutants after removing inner

mutants of “WordProcessor”, “Telephone”, “BlackJack” and “phoneDirectory” are: 89.74%, 88.24%, 83.58% and 86.67% respectively.

Application Name	Number of Mutants	Killed Mutants	Live Mutants	Inner Mutants	Percentage of Killed Mutant Without Inner Mutants
computer	25	24	1	0	96%
linkedList	16	12	4	4	100%
CoffeeMaker	76	56	18	16	93.33%
bank	26	20	6	4	90.91%
WordProcessor	49	35	14	10	89.74%
Telephone	24	15	9	7	88.24%
BlackJack	94	56	38	27	83.58%
Elevator	135	78	57	36	78.79%
phoneDirectory	110	65	45	32	83.33%
CruiseContol	39	13	26	24	86.67%

Table 5.15. Class Mutants Results Without Inner Mutants

Table 5.16 illustrates the results of Duplicate call mutants excluding the inner mutants. In “CoffeeMaker” and “LinkedList” applications we killed all the Duplicate call mutants. In addition the results show that we have achieved 80% for the whole applications except the “Elevator” application. This is because in the “Elevator” application there are many calls in many basic blocks in the same method. Application Name

Application Name	Duplicate Call Mutants Killed	Duplicate Call Mutants Live	Total of Mutants	Inner Mutants	Percentage of Killed Mutant Without Inner Mutants
Word Processor	26	14	40	10	86.67%
telephone	14	9	23	6	82.35%
phone Directory	47	39	86	29	82.46%
Linked List	3	2	5	2	100%
elevator	40	37	77	24	75.47%
cruise Control	4	21	25	20	80%
computer	21	1	22	0	95.46%
Coffee Maker	17	4	21	4	100%
black jack	14	15	29	12	82.35%
bank	18	6	24	4	90%

Table 5.16. Duplicate Call Mutants Results Without Inner Mutants

Table 5.17 shows the results for the parameter mutants (Giving initial values for the parameter before using it). The results show that the number of created mutants is less than the number of duplicate mutants because the number of parameter mutants is based on the number of methods implementation. In addition the results provide a good indication about our approach in creating test cases. In four applications: “Word Processor”, “Linked List”, “cruise Control” and “Computer” the test cases killed all the mutants. And for the rest of the applications the test cases killed 80% or more of the mutants.

Application Name	Parameter Mutants Killed	Parameter Mutants Live	Total of Mutants	Inner Mutants	Percentage of Killed Mutant Without Inner Mutants
Word Processor	6	0	6	0	100%
phone Directory	8	4	12	2	80%
Linked List	3	2	5	2	100%
elevator	24	7	31	2	82.76%
cruise Control	1	1	2	1	100%
computer	3	0	3	0	100%
Coffee Maker	18	7	25	5	90%
black jack	20	8	28	3	80%

Table 5.17. Parameter Mutants Results Without Inner Mutants

Table 5.18 shows the results for the return mutants (Returning initial values for the methods which return data type). The results show that five applications have achieved 100% in killing mutants, and the other killed 80 % or more except the “Elevator” application.

Application Name	Return Mutants Killed	Return Mutants Live	Total of Return Mutants	Inner Mutants	Percentage of Killed Mutants Without Inner Mutants
Word Processor	2	0	2	0	100%
telephone	1	0	1	0	100%
phone Directory	8	2	10	1	88.89%
Linked List	5	0	5	0	100%
elevator	9	13	22	10	75%
cruise Control	8	4	12	3	88.89%
Coffee Maker	22	7	29	7	100%
black jack	20	15	35	12	86.96%
Bank	2	0	2	0	100%

Table 5.18. Returns Mutants Results Without Inner Mutants

Table 5.19 shows the percentage of killed mutants that are created based on swapping parameter for the whole applications. The results show that the test cases killed all the mutants of each application.

Application Name	Swap Parameter Mutants Killed	Swap Parameter Mutants live	Total of Swap parameter Mutants	Percentage of Killed Mutant
Word Processor	1	0	1	100%
phone Directory	2	0	2	100%
Linked List	1	0	1	100%
elevator	5	0	5	100%
Coffee Maker	1	0	1	100%
black jack	2	0	2	100%

Table 5.19. Swap Parameters Mutants Results

Table 5.20 shows that for four applications: “Word Processor”, ”Linked List”, “Black Jack” and “Bank” all the swap methods mutants are killed. Other applications achieved 80% or more of killed mutants.

Application Name	Swap Method Killed	Swap Method Live	Swap Method Mutants	Percentage of Killed Mutants
Word Processor	9	0	9	100%
Telephone	5	1	6	83.33%
phone Directory	9	1	10	90%
Linked List	1	0	1	100%
Elevator	16	2	18	88.89%
cruise Control	9	2	11	81.82%
Coffee Maker	6	1	7	85.71%
black jack	8	0	8	100%
Bank	4	0	4	100%

Table 5.20. Swap Methods Mutants Results

Table 5.21 shows the results for deleting a call from a chain of calls mutants. From the results we can see that in two applications: “Linked List” and “Computer” the test cases achieved 100% in killing mutants. The test cases have killed 85% and more of mutants in three applications: “Phone Directory”, “Cruise Control” and “Coffee Maker”. While for the other applications the test cases killed 80% or more of the mutants.

Application Name	Deleting a Call Killed	Deleting a Call Live Mutants	Total of Deleting a Call Mutants	Percentage of Killed Mutants
Word Processor	11	2	13	84.62%
Telephone	9	0	9	100%
phone Directory	13	2	15	86.67%
Linked List	2	0	2	100%
Elevator	24	5	29	82.76%
cruise Control	17	3	20	85%
Computer	9	0	9	100%
Coffee Maker	17	3	20	85%
black jack	12	2	14	85.71%
Bank	7	1	8	87.5%

Table 5.21. Deleting A Call Mutation Results

CHAPTER 6. CONCLUSION AND FUTURE WORK

This research has presented a new approach for integration testing. The goal of the approach is to reduce the cost of integration testing while retaining as much as possible of its effectiveness by limiting the number of integration test cases. The second goal of this research is presenting a method for evaluating integration testing.

Contribution

In this research, we developed a methodology to specifically lower the cost of integration testing and generally lower the cost of software. Our methodology assumes that the probability that a method call will be erroneous is correlated significantly to the degree in which the calling method and called method depend upon each other. We used an information retrieval technique called Latent Semantic Indexing (LSI) as a proxy to calculate the dependency among methods since the current artificial techniques are not sufficiently developed to identify the degree of the dependency among methods. The similarity among methods is calculated through representing each method as a vector and finding the cosine angle among them. Next, the class pair weight is computed from the method pair weights for the methods in the two classes by adding all the method pair weights.

We calculated the total pair weight through adding all the class pair weights. Each class pair weight is divided by the total pair weight to form the adjusted class pair weight. Next, we determined the number of test cases by multiplying the total number of the integrated class by 5. The test cases were allocated to each pair of classes by multiplying the adjusted class pair weight by the total number of test cases and rounding up.

Another contribution of this research is developing a new tool to evaluate the integration testing process in order to evaluate our approach. We accomplished this by extending the mutation testing approach from unit testing into integration testing. We developed a set of integration mutation operators to support development of integration mutation testing. The operators seed integration errors into the application under test by, for example, swapping method parameters, calling a wrong method, deleting a call from a chain of calls, and misinterpreting the result of a method call. In addition, we conducted experiments on 10 Java applications: BlackJack, CruiseControl, LinkedList, Telephone, WordProcessor, Bank, Elevator, Phone Directory, Computer, and CoffeeMaker.

Our experimental results show that the percentage of killed mutants of the chain call mutants reached 100% for two applications, linkedList and Computer; 90% or more for four applications, WordProcessor, BlackJack, Bank and Telephone; and 83% or more for the other four applications, CruiseContol, CoffeeMaker, Elevator and phoneDirectory. In addition, the results show that the percentage of killed mutants of the class mutants reached 100% for one application, linkedList; 90% or more for three applications, Computer, CoffeeMaker, and bank; 80% or more for five applications, WordProcessor, Telephone, BlackJack, phoneDirectory and CruiseContol, and one application, Elevator reached 78.79%. The hypothesis of killing at least 60% of the mutants at integration level was approved since the lowest percentage of the killed mutants in all applications was 78.79%. In our experiments, we developed less than 50% of the number of test cases based on our approach for most of the applications, which means if we develop the exact number of the test cases we would get more killed mutants.

Future Work

This research addresses two main problems in integration testing: what the best order in which to integrate the classes currently available for integration is and which external method calls should be tested and in what order for maximum effectiveness. In this research, we did not explore which the test case selections are likely to be most effective in finding integration problems. We are planning to use the Latent Semantic Indexing (LSI) to find the similarity among test cases and other methods, and based on the methods pair weight, we would choose the test case that is most similar to the methods pair.

Our approach used the methods to determine the dependency among modules. The fields can be used in determining the dependency as well, so we plan to include the fields in computing the dependency. Moreover, we are planning to use basic blocks instead of methods in finding the dependency among modules since one method can have many basic blocks. Furthermore, we use an arbitrary value and multiply it with the number of classes in order to specify the number of test cases for the whole application. We are planning to derive this arbitrary value from the dependency among modules.

We did not evaluate our approach with other integration testing approaches mentioned in Chapter 3 because most of them are theoretical and we did not find any experiment data or tool for other approaches to compare with our approach. We are planning to enhance and add new features to the Mutation Integration Tool (MIT) by running the mutants against test cases automatically, generating reports, and trying to separate integration mutants based on the internal interaction (within the module) from external interaction among the modules. Moreover, we need to expand the MIT to create mutants in other programming languages such as C#, C++, and VB.net.

REFERENCES

[1] Victor Basili and Barry T. Perricone, "Software errors and complexity, an empirical investigation. In Software engineering metrics I, Martin Shepperd (Ed.). McGraw-Hill, Inc., New York, NY, USA, pp. 168-183, 1993.

[2] Huo Yan Chen and T. H. Tse and T. Y. Chen Taccle, "a methodology for object oriented software testing at the class and cluster levels", ACM Transactions on Software Engineering and Methodology, Vol.10, No.1, pp.56-109, 2001.

[3] Martin Jung and Francesca Saglietti, "Supporting Component and Architectural Re-usage by Detection and Tolerance of Integration Faults", In Proceedings of the Ninth IEEE International Symposium on High-Assurance Systems Engineering *(*HASE '05), IEEE Computer Society, Washington, DC, USA, pp.47-55,2005.

[4] Paul C. Jorgensen and Carl Erickson, "Object-oriented integration testing",Commun, ACM ,Vol.37,No.9, pp. 30-38, September 1994.

[5] Boris Beizer,"Software Testing Techniques", (2nd Ed.). Van Nostrand Reinhold Co., New York, NY, USA,1990.

[6] Overbeck, J., "Integration Testing for Object-Oriented Software," Ph.D. thesis, Vienna University of Technology, Vienna, Austria, 1994.

[7] W. T. Tsai and Xiaoying Bai and Ray Paul and Weiguang Shao and Vishal Agarwal, "End-To-End Integration Testing Design", IEEE 2001.

[8] DoD OASD C3I Investment and Acquisition, "Year 2000 Management Plan", 1999.

[9] S Ali, L Briand, M Rehman, H Asghar, M Iqbal, A Nadeem, "A state-based approach to integration testing based on UML models", Information and Software Technology Butterworth-Heinemann,2007.

[10]Patrizio,P, Henry.M. Antonio,B and Fabrizio.F, “TeStor: Deriving Test Sequences from Model-Based Specifications”, Springer-verlag Berlin Heidlberg ,pp.267-282, 2005.

[11] Alessandro Orso Ph.D, Thesis, “Integration Testing of Object-Oriented Software”, Politecnico di Milano. Milan, Italy 1999.

[12] Roger T. Alexander, James M. Bieman, Sudipto Ghosh, and Bixia Ji, 2002. “Mutation of Java Objects”, In Proceedings of the 13th International Symposium on Software Reliability Engineering (ISSRE '02), IEEE Computer Society, Washington, DC, USA, pp.341,2002.

[13] Tom Maibaum and Zhe (Jessie)Li, “A Test framework for integration testing of object oriented programs”, Proceeding CASCON '07 Proceedings of the 2007 conference of the center for advanced studies on Collaborative research, ACM New York, NY, USA©2007.

[14] “The CommUnity team and ATX Software SA. CDE Documentation”, http://www.atxsoftware.com/CDE/.

[15] Dominik Hura and Michał Dimmich, “A method facilitating Integration Testing of Embedded Software”, Proceeding WODA '11 Proceedings of the Ninth International Workshop on Dynamic Analysis, ACM,New York, NY, USA©2011.

[16] Rattikorn Hewett and Phongphun Kijsanayothin, “Automated Test Order Generation for Software Component Integration Testing”, Proceeding ASE '09 Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering IEEE Computer Society Washington, DC, USA ©2009.

[17] Leonard Gallagher and Jeff Offutt, “Automatically Testing Interacting Software Components”, Proceeding AST '06 Proceedings of the 2006 international workshop on Automation of software test ACMNew York, NY, USA©2006.

[18] Hai Yuan and Tao Xie, “substra: a Framework for automatic Generation of Integration Tests”, ACM Press, pp.76-70, 2006.

[19] Nam Hee Lee and Tai Hyo Kim and Sung Deok Cha, “Construction of Global Finite State Machine for Testing Task Interactions written in message Sequence Chart”, In The Fourteenth International Conference on Software Engineering and Knowledge Engineering (SEKE’02), 2002.

[20] Aynur Abdurazik and Jeff Offutt, “coupling-based Class Integration and Test Order”, Proceeding AST '06 Proceedings of the 2006 international workshop on Automation of software test, ACM, New York, NY, USA©2006.

[21] D. Kung, J. Gao, P. Hsia, Y. Toyoshima, and C. Chen, “A test strategy for object-oriented programs”, In 19th Computer Software and Applications Conference (COMPSAC 95), pp.239-244, Dallas, TX, August 1995.

[22] L. Briand, J. Feng, and Y. Labiche, “Using genetic algorithms and coupling measures to devise optimal integration test orders”, In Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering, pp. 43-50, Ischia, Italy, 2002.

[23] L. Briand, Y. Labiche, and Y. Wang, ”Revisiting strategies for ordering class integration testing in the presence of dependency cycles”, Technical report SCE-01-02, Careleton University, 2001.

[24] L. C. Briand, Y. Labiche, and Y. Wang, “An investigation of graph-based class integration test order strategies”, IEEE Transactions on Software Engineering, Vol.29,No.7, pp.594-607, July 2003.

[25] M. J. Harrold and J. D. McGregor, “Incremental testing of object-oriented class structures”, In 14th International Conference on Software Engineering, pp. 68-80, Melbourne, Australia, May 1992.

[26] K.-C. Tai and F. Daniels, “Test order for inter-class integration testing of object-oriented software”, In The Twenty-First Annual International Computer Software and Applications Conference (COMPSAC '97), pp. 602-607, Santa Barbara CA, 1997.

[27] Y. L. Traon, T. Jeron, J.-M. Jezequel, and P. Morel, “Efficient object-oriented integration and regression testing”, IEEE Transactions on Reliability, pp. 12-25, March 2000.

[28] Benz, Sebastian, “Combining Test Case Generation for Component and integration Testing”, Proceedings of the 3rd international workshop on Advances in model based testing AMOST 07, ACM Press, pp.23-30, 2007.

[29] Konstantin Rubinov, “Generating Integration Test Cases Automatically”, Proceeding FSE '10 Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering, ACMNew York, NY, USA©2010.

[30] Harry M. Sneed Anecon GmbH, “testing object oriented software systems”. Proceeding ETOOS '10 Proceedings of the 1st Workshop on Testing Object-Oriented Systems, ACM, New York, NY, USA©2010.

[31] Srinivasan Desikan, Gopalaswamy Ramesh Software Testing Principles and practices. Pearson Education India, Sep 1, 2006 - 486 pages TextBook.

[32] K. N. Leung and L. White, “A study of integration testing and software regression at the integration level”, In Proceedings of the conference on Software. Maintenance-90, pp.290.301, San Diego, California, 1990.

[33]Yu-Seung MaYong-Rae KwonJeff Offutt, “Inter class Mutation Operators for Java”, Proceeding ISSRE '02 Proceedings of the 13th International Symposium on Software Reliability Engineering IEEE Computer Society Washington, DC, USA ©2002.

[34] “IEEE Standard for Software Test Documentation", IEEE Std 829-1998 , vol., no., pp.i, 1998,doi: 10.1109/IEEESTD.1998.88820.
URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=741968&isnumber=16010

[35] Berger, A et al, “Bridging the Lexical Chasm: Statistical Approaches to Answer Finding”, In Proc. Int. Conf. Research and Development in InformationRetrieval, 192-199,2000.

[36] Jeff OffuttYu-Seung Ma and Yong-Rae Kwon, “The Class-Level Mutants of MuJava”, Proceeding AST '06 Proceedings of the 2006 international workshop on Automation of software test, ACM New York, NY, USA ©2006.

[37] Sun-woo Kim and John A. Clark and John A. Mcdermid, “Assessing test set adequacy for object-oriented programs using class mutation”, 28 JAIIO: Symposium on Software Technology (SoST`99) pp. 72-83, 1999.

[38] Yu-Seung MaMary Jean Harrold and Yong-Rae Kwon, “Evaluation of mutation testing for object-oriented programs”, Proceeding ICSE '06 Proceedings of the 28th international conference on Software engineering, ACM New York, NY, USA ©2006.

[39] Paul E. Black, “Mutation Operators for Specifications”, In Proceedings of 15 th IEEE International Conference on Automated Software Engineering, IEEE Computer Society, pp.81-88, 2000.

[40] Máarcio E. Delamaro, José C. Maldonado and Aditya P. Mathur, “Interface Mutation: An Approach for Integration Testing”, Journal IEEE Transactions on Software Engineering archive, Vol.27, No.3, IEEE PressPiscataway, NJ, USA, March 2001.

[41] S. Madiraju and S. Ramakrishnan A. J, “Hurst Towards automated mutation testing”, 2004.

[42] Mresa, Elfurjani S. and Bottaci, Leonardo, “Efficiency of mutation operators and selective mutation strategies: An empirical study”, Softw. Test., Verif. Relia,. pp.205-235, 1999.

[43] David SchulerValentin DallmeierAndreas Zeller, “Efficient Mutation Testing by Checking Invariant Violations”, Proceeding ISSTA '09 Proceedings of the eighteenth international symposium on Software testing and analysis ACM New York, NY, USA ©2009.

[44] M. E. Delamaro, J. C. Maldonado, and A. P. Mathur, “Integration Testing Using Interface Mutation”, In Proceedings of International Symposium on Software Reliability Engineering (ISSRE ’96), pp. 112–121, April 1996.

[45] Sun-woo Kim and John A. Clark and John A. Mcdermid, “Assessing Test Set Adequacy for ObjectOriented Programs Using Class Mutation”, JAIIO: Symposium on Software Technology (SoST`99),pp. 72-83, 1999.

[46] A. Jefferson Offutt, Ammei Lee, Gregg Rothermel, Roland Untch and Christian Zapf, “An Experimental Determination of Sufficient Mutation Operators”, ACM Trans. on Software Engineering & Methodology, Vol. 5, pp. 99–118, April 1996.

[47] Examensarbete I Datalogi and Mattias Bybro and Examinator Prof and Stefan Arnborg, ”A Mutation Testing Tool for Java Programs Ett verktyg för mutationstestning av Javaprogram”, 2003.

[48] Ben H. Smith and Laurie Williams, “On Guiding the Augmentation of an Automated Test Suite via Mutation Analysis”, Journal Empirical Software Engineering archive,Vol.14, No.3, Kluwer Academic PublishersHingham, MA, USA, June 2009.

[49] Bradbury, J.S.; Cordy, J.R.; Dingel, J, “Mutation operators for concurrent java (j2se 5.0)”, in Proceedings of the Second Workshop on Mutation Analysis, IEEE Computer Society, pp. 11–11,2006.

[50] M. Delamaro, M. Pezz, A. M. R. Vincenzi, and J. C. Maldonado, “Mutant operators for testing concurrent java programs”, in XV Simpsio Brasileiro de Engenharia de Software, Rio de Janeiro, RJ, Brasil, pp. 272 – 285,2001.

[51] Leon Wu and Gail Kaiser, “Empirical Study of Concurrency Mutation Operators for Java”, Department of Computer Science Columbia University New York, NY 10027, 2010.

[52] M. Scholive, V. Beroulle, C. Robach, M. L. Flottes and B. Rouzeyre, “Mutation Sampling Technique for the Generation of Structural Test Data”, Proceeding DATE '05 Proceedings of the conference on Design, Automation and Test in Europe, Vol.2, IEEE Computer SocietyWashington, DC, USA©2005.

[53] Lech Madeyski, Norbert Radyk, “Judy - a mutation testing tool for Java", IET Software,vol. 4, No. 1, pp. 32-42,2010.

[54] Macario Polo, Mario Piattini and Ignacio Garc´ıa-Rodr´ıguez, “Decreasing the cost of mutation testing with second-order mutants”, SOFTWARE TESTING, VERIFICATION AND RELIABILITY Softw. Test. Verif. Reliab, (19),pp.111-131, 2009.

[55] Upsorn Praphamontripong and Jeff Offutt, “Applying Mutation Testing to Web Applications”, Software Testing, Verification, and Validation Workshops (ICSTW), 2010.

[56] Francesca Lonetti and Eda Marchetti, “X-MuT: A Tool for the Generation of XSLT Mutants”, Proceeding QUATIC '10 Proceedings of the 2010 Seventh International Conference on the Quality of Information and Communications Technology IEEE Computer Society Washington, DC, USA, 2010.

[57]. Manning, Christopher D. , Raghavan, Prabhakar.Schütze, Hinrich, “Introduction to information retrieval”, New York, Cambridge University Press, 2008.

[58] Grossman, David , and Frieder, Ophir, “Information retrieval algorithms and heuristics”, Boston : Kluwer, 1998.

[59] Cherukuri, Aswani Kumar and Suripeddi, Srinivas, “On the performance of Latent Semantic Indexing-based Information Retrieval”, CIT, (17) pp. 259-264, 2009.

[60] Vincent Wolowski, “Fuzzy Information Retrieva”, Presentation, Seminar Softcomputing, Department of Computer Science, Chair of Applied Computer Science VII, Intelligent Information and Communication Systems Group of Prof. Helbig, University of Hagen, Hagen, Germany, May 27.

[61] Scott Deerwester and Susan T. Dumais and George W. Furnas and Thomas K. Landauer and Richard Harshman, “indexing by latent semantic analysis”, JOURNAL OF THE AMERICAN SOCIETY FOR INFORMATION SCIENCE, Vol.41, No.6, pp. 391—407, 1990.

[62] DI LUCCA, G. A., DI PENTA, M., AND GRADARA, S, “An approach to classify software maintenance requests”, In Proceedings of the IEEE International Conference on Software Maintenance (Montr´eal, Qu´e., Canada), IEEE Computer Society Press, Los Alamitos, CA, pp.93–102, 2002.

[63] MAAREK, Y., BERRY, D., AND KAISER, G, “An information retrieval approach for automatically constructing software libraries”, IEEE Trans. Softw. Eng. Vol.17, NO. 8, pp.800–813, 1991.

[64] ARNOLD, S. P., AND STEPOWAY, S. L, “The reuse system: Cataloging and retrieval of reusable software”, In Software Reuse: Emerging Technology, W. Tracz, Ed. IEEE Computer Society Press, Los Alamitos, CA, pp.138–14,1988.

[65] BURTON, B. A., ARAGON, R. W., BAILEY, S. A., KOELHER, K., AND MAYES, L. A, ”The reusable software library”, In Software Reuse: Emerging Technology,W. Tracz, Ed. IEEE Computer Society Press, Los Alamitos, CA, 129–137,1987.

[66] PIGHIN, M, “A new methodology for component reuse and maintenance”. In Proceedings of 5th European Conference on Software Maintenance and Reengineering (Lisbon, Portugal), IEEE Computer Society Press, Los Alamitos, CA, pp.196–199,2001.

[67] Robert Tairas and Jeff Gray, “An information retrieval process to aid in the analysis of code clones”, Journal Empirical Software Engineering archive,Vol.14, No.1, Kluwer Academic Publishers Hingham, MA, USA, February 2009.

[68] Denys Poshyvanyk , Andrian Marcus , Rudolf Ferenc , and Tibor Gyimóthy, “Using information retrieval based coupling measures for impact analysis”, Journal Empirical Software Engineering archive Vol.14, No.1, Kluwer Academic Publishers Hingham, MA, USA. February 2009.

[69] Meghan Revelle , Malcom Gethers , and Denys Poshyvanyk, “Using structural and textual information to capture feature coupling in object-oriented software”, Journal Empirical Software Engineering archive, Vol.16, No. 6, Kluwer Academic Publishers Hingham, MA, USA December 2011.

[70] LORMANS, M. AND VAN DEURSEN, A, ‘Can LSI help reconstructing requirements traceability in design and test?”, In Proceedings of 10th European Conference on Software Maintenance and Reengineering (Bari, Italy), pp.45–54,2006.

[71] SETTIMI, R., CLELAND-HUANG, J., BEN KHADRA, O., MODY, J., LUKASIK, W., AND DEPALMA, C, “Supporting software evolution through dynamically retrieving traces to

UML artifacts", In Proceedings of 7th International Workshop on Principles of Software Evolution (Kyoto, Japan), IEEE Computer Society Press, Los Alamitos, CA, pp.49–54, 2004.

[72] Salton, G. & Buckley, C, "Term-weighing approaches in automatic text retrieval", In Information Processing & Management, Vol.24, No.5, pp.513-523, 1988.

[73] Salton, A. Wang, C. Yang, "A vector-space model for information retrieval", Journal of the American Society for Information Science, 1975.

[74] Grossman, David , and Frieder, Ophir, "Information retrieval algorithms and heuristics", Boston : Kluwer, 2004.

[75] Trevor.S, Meghan.R, and Denys.p, "FLAT3: Feature Location and Textual Tracing Tool", ICSE '10, Cape Town, South Africa, Copyright © 2010.