Establishing test-to-code traceability links using dynamic and static techniques for C# Applications

By

Mudassar Saleem

00000273951

Supervisor

Dr. Wasi Haider Butt

Department of Computer and Software Engineering

College of Electrical and Mechanical Engineering

National University of Science and Technology (NUST)

Islamabad, Pakistan

September 2022

Establishing test-to-code traceability links using dynamic and static techniques for C# Applications

By

Mudassar Saleem

00000273951

Supervisor

Dr. Wasi Haider Butt

A thesis submitted in conformity with the requirements for the degree of *Master of Science* in Software Engineering

Department of Computer and Software Engineering

College of Electrical and Mechanical Engineering

National University of Sciences and Technology (NUST)

Islamabad, Pakistan

September 2022

Declaration

I, *Mudassar Saleem* declare that this thesis titled “Establishing test-to-code traceability links using dynamic and static techniques for C# Applications” and the work presented in it are my own and has been generated by me as result of my own original research.

_____

Mudassar Saleem

00000273951

Plagiarism Report

This thesis report has been checked for plagiarism. The Turnitin report is also attached, and approved by the supervisor.

_____

Mudassar Saleem

00000273951

_____

Signature of Supervisor

Copyright Notice

- Copyright in text of this thesis rests with the student author. Copies (by any process) either in full, or of extracts, may be made only in accordance with instructions given by the author and lodged in the Library of NUST College of EME. Details may be obtained by the Librarian. This page must form part of any such copies made. Further copies (by any process) may not be made without the permission (in writing) of the author.

- The ownership of any intellectual property rights which may be described in this thesis is vested in NUST College of EME., subject to any prior agreement to the contrary, and may not be made available for use by third parties without the written permission of CEME, which will prescribe the terms and conditions of any such agreement.

- Further information on the conditions under which disclosures and exploitation may take place is available from the Library of NUST College of EME, Islamabad.

Abstract

With the fast-paced development such as DevOps and other agile development methodologies, the code is rapidly changing with the incoming requirements and it is really hard to maintain the quality of a software. Unit tests are the first step that a developer takes to ensure everything is according to the requirements. Tracing which test case is associated to a code artifact (method or class), is a hard task if done manually. A traceability link is a way to maintain and trace the links between different software artifacts. Test-to-code traceability links help a developer to keep track of the test cases that are related to a specific piece of code. These links restrict a developer to remain consistent with the existing architecture when a change is made to the code. These links also reduce the risk of missed or unseen faults in the code. But most of times, the development processes do not adopt this approach due to the extra burden it puts on the developer. Manually creating these traceability links is a hectic routine for the developer and maintaining these links is another big ask. Also, doing all of these manually makes it prone to the errors. In this research, I am introducing an approach and a tool to automatically develop the test-to-code traceability links using static and dynamic approaches for the C# applications (Libraries, ASP.NET Core application). There are many tools built for the JAVA application, but there's not much work done for the C# applications solely. C# language has been around for many years. It is constantly gaining popularity over the time and many large-scale applications are being developed on it. Especially with the introduction .NET Core (which is an open-source and platform-independent framework), it is getting better and better with respect to performance, and more developers are inclined to it. This tool will help these developers in creating these traceability links.

Keywords: Traceability, Software testing, Unit testing, Software development, Software engineering

Acknowledgment

All praise to Allah (the omnipotent and the omnipresent) who has bestowed me with ardor, courage, and patience with which I have completed another phase of my academic journey.

I would like to dedicate this thesis to *my beloved younger sister* who holds a special place in my heart. I would also dedicate this thesis to my parents, teachers, siblings, and students, who were continuous source of motivation in my tough times. They always encouraged me to continue higher studies and fully supported me to full fill my dream degree. My parents played a pivotal role in my MS degree by providing moral and financial support.

Most importantly, I want to pay special gratitude to my supervisor “Dr. Wasi Haider Butt” without whom I would not have been able to take this task to fruition. He enlightens my path with continuous support and made me competent during the whole duration of research.

Finally, I am thankful to “Dr. Arsalan Shaukat”, “Dr. Ali Hassan” and all my friends who assisted me in this thesis and throughout the whole research process.

Table of Contents

Table of Figures

Table of Tables

CHAPTER 1

INTRODUCTION

Software testing is a process in which a program is executed with a goal of discovering errors in an application. Its purpose is not to show that something is working or not, but the underline intent is to improve the quality of an application and add value to it in terms of quality and reliability. Ensuring the quality and reliability of a software application builds up confidence that the application does what it is supposed to do and does not do what it is not supposed to do [1]. And ensuring this is very expensive in terms of personnel, time, and money [2].

Unit testing is the first step to ensure that a unit is working as it is intended. A unit is the smallest module or a block of code (method), that can be tested independently. Unit testing is mainly the responsibility of a developer and it's the first step in ensuring quality. It is very cost-effective as it catches problems with the code at the early stages of development. These small tested units are then integrated, and integration testing is performed [3]. Writing these unit tests and integration tests is among the primary activity in the Test-Driven Development (TDD), and it has been found an effective way of fault detection and localization. [4].

Apart from making sure that the software application does what it is intended to do, the important considerations is to make sure that the documents (test cases) should be traceable [3]. So, whenever some functionality/feature is modified, it can be easily traced which test cases are associated with the specific feature. Software traceability is a vital part of the development process. In the fast-paced agile development process, the traceability links address the need of the industry and it is also implemented successfully in different industries [5]. A trace link represents an association between the source and target software artifacts [6]. In the case of a test-to-code traceability link, it's a link between the test artifact (test class, test method) to code artifacts (class, method). In the software engineering that are a lot of document/artifact repository are maintained and all of these artifacts are isolated from each other. And these artifacts are maintained by many different individuals. A

person working on one artifact may not have knowledge about the other artifact that is related to it, and does not know about the ripple effect it can create or its consequences. The traceability links tends to resolved these and many other issues between the artifacts [7][50]. Figure 1 shows how the artifacts are linked in the different phases of the software engineering process.

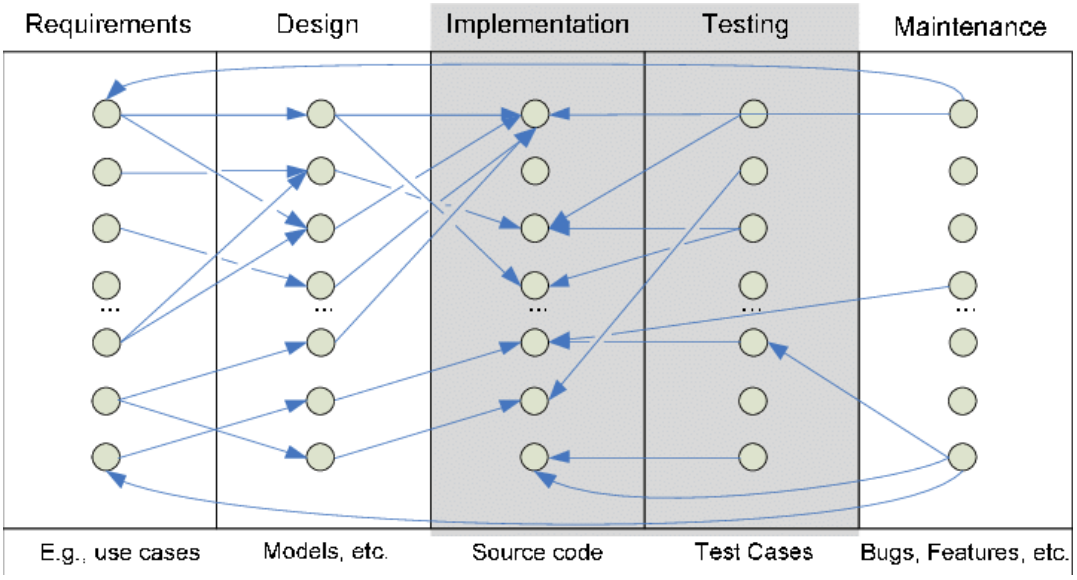

Figure 1 - Hypothetical Traceability graph between Software Artifacts [8]

The test to code traceability links are very important in agile software methodologies. As the code is continuously changing to reflect the customer's needs, the test cases/suits need to be updated as well. The traceability can provide both forward and backward links that help in maintaining these artifacts. The traceability link helps in locating the code to be maintained, or test case to be updated especially in code refactoring [9]. Few of the benefits of test-to-code traceability.

- It helps in the maintenance of source code and test suites i.e. test cases and code remain synced.
- It reduces the risk of unseen faults and test failures.
- It helps in maintaining an accurate model of the system.
- It helps in reducing architectural degradation.

1.1. Motivation

The research and literature (Explained in Chapter 2) that exists till date mainly focuses on the Java Applications. There are a few very well-known tools e.g. SCOTCH+ and TCTracer, which generate the test-to-code traceability links. Some other tools and techniques have also been developed solely for the Java-based applications. But no such tool is developed for C# based applications that generate test-to-code traceability links automatically. C# language has been constantly gaining popularity over time and many large-scale applications are being developed on it [10]. Especially with the introduction of .NET Core (open-source, platform-independent), it is getting better and better with respect to performance, more and more developers are inclined to it [11][12].

C# language is one popular programming language among the developers and many organizations. From mobile applications to video games, desktop applications to web applications, and cloud applications, C# is being widely used to develop these applications. Adobe Photoshop, Adobe Systems, Mozilla Firefox, Stackoverflow, GoDaddy, Bing, UPS, BBC Good Foods, and Microsoft services are among the applications that are developed in C# and ASP.Net [13]. There is a need for the developers who are working on the C# projects, to have a tool that develops the traceability links between the tests and code.

1.2. Problem Statement

Test-to-code traceability links maintain the link between test artifacts (unit, regression test) and the code artifacts (class, method). Developing and maintaining these links manually is a difficult task and it puts an extra burden on the shoulder of developers. To automatically establishes test-to-code traceability links in the software applications developed using the C# language is a crucial element for improving software maintenance, code refactoring, and effective test case selection in case of regression testing. The main goal of the research is to develop a tool that automatically develops these traceability links.

1.3. Aims and Objectives

The objective of the research is to understand the importance of the test-to-code traceability links, how it helps the developers to remain consistent with the

architecture, and how it enables reduction of faults. The aim is to provide a tool that automatically generates these traceability links. Introducing a tool not only pulls off the burden from the developer but it also reduces the risk of errors which may occur when it is done manually.

1.3.1. Literature Review Objectives

The objectives of the literature review are:

- ✓ To identify the importance of traceability among the software artifacts.
- ✓ To identify the importance of traceability between tests and code.
- ✓ Review existing approaches, techniques and tools for test-to-code traceability.
- ✓ Effective selection of test-to-code traceability techniques.

1.3.2. Tool Objectives

The objectives of the tools are:

- ✓ To develop test-to-code traceability links automatically.
- ✓ To take off the burden from the developers' shoulders.
- ✓ To reduce the risk of errors that may occur if it is done manually.

1.4. Structure of Thesis

The rest structure of thesis is followed as: *Chapter 2* provides an overview and discussion of the existing literature. *Chapter 3* explains the approaches and techniques used for tool. *Chapter 4* explains the evaluation criteria and the results. And *Chapter 5* summarizes the research, its goals and the future work.

CHAPTER 2

LITERATURE REVIEW

Literature review is one of the primary components of any research. The main objective of the literature review is to summarize the prior research efforts, what were the findings of that efforts, and what conclusions were made through that effort. Also, it tells about the accuracy and completeness of that knowledge. Instead of reinventing the wheel, it can give ideas about your own research [14].

In this chapter, a review of the literature has been presented. It discusses the different techniques for developing traceability links between test and code, and the tools that can help in developing these traceability links.

2.1. Overview

Unit testing is the early activity that a developer does when writing the code. It is the responsibility of the developer to write the code, and then regularly update test cases as the code changes [15]. A survey shows that creating and maintaining these unit tests is really hard for the developers as it puts an extra responsibility on the developers [16]. One of the main goals of the test cases is that the tests should be traceable. It should tell which test case is associated with a specific feature or requirement [4]. This is called traceability.

Traceability between the software artifacts is managed through the traceability links in which a source artifact is associated with the target artifacts [6]. These links provide a forward and backward flow that can help in locating the ripple effect in a specific artifact. Test-to-Code traceability link help locate which test case needs to be updated when a code is updated, or locating a fault when test case fails [17]. Another main benefit of these links is that it helps in software maintainability. Code refactoring is one of the common practices that is usually done by the developers. The traceability links definitely help in refactoring the test cases as well that are associated with the code. But it is not practically followed in most of the agile methods [9]. In the regression testing, traceability link can also minimize the number

of test cases to be executed for the regression test (as a result of code change) [17].

Coevolution of both test and code artifacts is not consistently practiced. Over the time, the both of these artifacts get desynchronized. A coevolution analysis is code on the code, which clearly shows that the production code and test code are never in sync. The test code is updated in a separate commit and production code in a separate commit [18]. Traceability links can help the developers to get these two artifacts in sync. Not only that, it also helps in fault localization. Unit tests are generally written to test a specific piece of code i.e. method. Most of the time naming convention is followed that help in tracing. A good coding convention usually follow this approach [19]. In Continuous integration (CI), these traceability links also help while doing integration testing and regression testing as it can identify which test case needs to be executed that is potentially affected by a change [20][49]. Developing and maintaining these links also comes with a lot of challenges as well [21].

2.2. Traceability Approaches

Different traceability approaches are used to create a link between unit tests and code artifacts. Following are few of the techniques [9][20][22][23] that can be used.

2.2.1. Naming Convention (NC)

Naming convention is the simplest way to identify the methods and classes under test and create a link. If the unit test method or unit test class starts or ends with the "test", removing the "test" will give us the name of the method or class under test. For example, if the *testCalculateTax* is a test method, removing the "test" gives us *CalculateTax* i.e. method name under test.

2.2.2. Naming Convention Contains (NCC)

Naming convention contains is derived from the Naming convention approach. It is possible that name does not match exactly the same after removing "test" from it. Because there are times a method under test is tested by more than one unit test. In that case the if the test method name contains the part of method under test name, it is possible that the unit test is testing the specific method.

2.2.3. Lexical Analysis (LA)

Lexical analysis is done based on the lexical token. A vocabulary is defined by a developer that is used in the source code. Vocabulary can be a simple natural language e.g. Type identifiers, names, etc. The unit test and the methods under test should be using same the vocabulary in the code. Based on that it can be determined whether the unit test is testing a method or not.

2.2.4. Longest Common Subsequence (LCS)

Longest common subsequence is a way to find the sequence that is common in the given sequences. This approach can be used to determine if a unit test is testing a specific method. The name of the unit test or class is compared with the method or class under test. Based on longest common subsequence it can be identified whether the unit test or class is testing the specific method or class.

- **Longest Common Subsequence Both (LCS-B):** This is a variant of LCS in which we determine if the unit test name and method name matches exactly the same. This approach is similar to NC that also match the whole name.

- **Longest Common Subsequence Unit (LCS-U):** This is another variant of LCS in which we determine the longest subsequence that is common. And then it can be determined whether the unit test is testing a method or not. This approach is similar to NCC.

2.2.5. Levenshtein Distance

Levenshtein distance is a metric that works on the string. It measures the difference between the two strings [24]. The distance between the unit test name and method name is measured using the Levenshtein Distance, and based on that it can be determined whether the unit test is testing a method or not. The lower the value of distance, higher the chance that unit test is testing a method.

2.2.6. Static Call Graph (SCG)

The Static call graph is a technique in which references are maintained from test case to production code. But not all methods or classes are tested that are called from the test case. All of the classes are collected called from a test case, and then a set of

classes is selected if it has higher references. Based on that it can be determined whether the unit test is testing a method or not.

2.2.7. Last Call Before Assert (LCBA)

A common pattern that many developers follow when creating a unit is [25]:

$$Arrage \rightarrow Act \rightarrow Assert$$

- **Arrange:** In these statements, all the required inputs are arranged and put together.
- **Act:** In these statements, the test cases are called i.e. the object states are modified.
- **Assert:** In these statements, the expected result is compared with the actual result and assertion is made whether the test is successful or not.

In the unit test, it is an assumption that method under test is called before the assert statement [39]. Last call before assert approach works on that assumption, and based on that it can be determined whether the unit test is testing a method or not.

2.2.8. Tarantula

Tarantula is a fault localization algorithm that helps in detecting a fault that is causing a test to fail. The algorithm determines the suspiciousness of each line in a test case. It calculates the ratio of successful and failed test cases. If the ratio for the failed test cases is higher, the suspiciousness is also higher [26]. It can be defined as follows where t is the code entity (test case):

$$suspiciouness(t) = \frac{\frac{passed\ (t)}{totalFailed}}{\frac{passed\ (t)}{totalPassed} + \frac{failed(t)}{totalFailed}}$$

A heuristic can be used to identify a method that is relevant to a unit test. Based on that it can be determined whether the unit test is testing a method or not.

2.2.9. Term Frequency – Inverse Document Frequency (TF-IDF)

Term frequency–inverse document frequency is a statistical and natural language processing (NLP) technique. It tells the significance of a word in a document. The

value is increased as the word appears frequently in a document. It is used for information retrieval and is widely used in the recommender systems and search engines. It can be defined [27] as:

$$tf = \frac{NumberOfTimesWordAppearedInDocument}{TotalWordsInDocument}$$

$$idf = log \frac{TotalDocuments}{DocumentFrequency}$$

Where

$$tf - idf = tf \times idf$$

TF-IDF can be used in linking the test and code artifacts. The tests are considered as the documents, and the methods are considered as the term. The higher the frequency of terms in the document, the higher the possibility that a unit test executes the method. Based on that it can be determined whether the unit test is testing a method or not.

2.2.10. Call Depth Analysis

Call depth analysis looks at the call stack of a method and determine how far it is called from the unit test. The method that is too far way is less likely to be tested by a unit test. Based on that it can be determined whether the unit test is testing a method or not.

2.3. Related Work

Test-to-code traceability links have many benefits in the software development process as explained in Chapter 1. And many researches have been conducted recently due to its significance in the engineering process. Also, different tools have been implemented to develop and maintain these traceability links over the past few years. Establishing the traceability links manually is a difficult job, and later it can't be maintained. Doing all of this manually requires the efforts of the developers and other stakeholders involved with the linked artifacts. TestRoutes is another manually curated dataset for test-to-code traceability. It contains about 2000 methods

classification and maintaining this large number is not an easy task [28]. To improve manual maintenance of the traceability links, and gamification concepts are used [29]. But these approaches are not significantly adopted.

There are generally two ways to generate the test-to-code traceability links; Static and Dynamic. For static traceability, the physical code files are parsed. For dynamic traceability, the source code is executed and code coverage information is gathered to extract the information [30]. The analysis and importance of different static and dynamic approaches used to develop these traceability links have shown these link are helpful when it comes to the maintenance of the test suites [31].

Slicing and Coupling based Test to Code trace Hunter (SCOTCH) is an approach that uses conceptual coupling to differentiate the test class and helper classes. It gives a better accuracy as compared to other techniques such as NC and LCBA because of its limitation [32]. SCOTCH+ works by dynamically slicing to identify the candidate tested classes. These candidates are selected based on the last executed assert statements. And then textual information (Name Similarity - NS) is used to differentiate between the actual methods (that need testing) and the helper methods. This approach gives better results than NC and LCBA [44].

TCTracer is another tool that uses an ensemble of different approaches such as NC, NCC, LCBA, and LCS (described in Section 2.2.). It uses both static and dynamic information to automatically develop the traceability links. Static information is collected through parsing the Java class source files. Dynamic information is collected through code instrumentation and execution of the code. The tool is developed for Java applications and tested on the open source applications as well. The tool not only works on the class level but it also works on the method level. It gives a mean average percentage (MAP) of 85% for test-to-method links and 92% for test-to-class links [34].

In DevOps-based software engineering environments, the production code is continuously updated by the developer. In this environment, the traceability links development has critical importance. SAT-Analyzer is a traceability tool that establishes the links between software artifacts i.e. Source code and unit test code. Junit test code is processed through Java Grammar and ANTLR. Then the

traceability links are developed using string comparison and Levenshtein Distance algorithms. The tool provides an accuracy of 71% on average [35].

Another approach that automatically identifies the methods under test also called focal methods in the unit tests. Discriminating between these focal and non-focal methods manually is hard. The approach uses Data Flow Analysis (DFA) to gather information regarding unit tests (JUnit) and code (Java). The research focuses on the Java classes and it identifies F-MUTs in the unit test case. The prototype implementation of the approach is fully automatic and it has an accuracy of 85% [36].

ETUCA is another automatic approach to generate the links. This approach introduces a custom attribute for .NET Unit tests. The attribute ensures that the traceability links are established at the time of unit test creation. The responsibility of creating the traceability link now falls on the shoulder of developers given they correctly embed the attribute. The quality of this approach is assessed through a survey of questionnaires. Its quality assessment of ETUCA resulted high from the user's perspective [37].

Test-to-code traceability link recovery has received attention due to fast-paced integration and deployment processes. A Hierarchical Trace Map visualization technique is proposed that tries to recover the traceability links [39]. Another visualization technique is proposed that combines approaches such as NC, LCBA, and SCG to recover the traceability links [38]. TCTracVis is another visualization tool that visually shows the links between the unit test and methods/classes. But the literature does not show any accuracy and performance evaluation [33]. Natural language processing methods are extensively used in the traceability recovery techniques. It is found that Word Embedding and Latent Semantic Indexing (LSI) performs better than AST-based identifier extraction and API documentation [40]. LSI has been found very effective among the natural language processing techniques and it can increase the results by 30% [47]. Among the LSI and TF-IDF has been found a good candidate for the traceability links and produced good results [49]. Another automated approach is presented that leverages the semantics of the software artifacts and creates traceability. It generates the domain-specific concept models and creates the trace links. The concept model is created based on the textual

information [41].

Fault localization techniques such as Tarantula have been found useful in finding out the link between the test artifact and the code artifact [20]. But in a different study, a comparison has been done between the Spectrum based Fault Localization with the traditional traceability approaches. After experimenting on the three different projects, it was found that this fault localization approach does not perform better than the tradition traceability approaches [42].

Different machine learning (ML) approaches e.g. neural networks and deep learning have also been used to develop the traceability links between test and code. TestNMT is one such approach that uses neural machine translation to generate the function-to-test links. But the approach has few limitations [43]. In another approach TCTracer, the results are compared with the simple feed forward neural network, and the results were not as satisfactory as compared to simple test-to-code traceability e.g. NS, NCC, LCBA, etc. Also generating the test data (ground truth) for these approaches is not feasible [34].

2.4. Discussion

Traceability links are a way to link the different software artifacts in the software engineering process and there is clearly a need for these links. The literature review presented in this section clearly shows the importance of the test to code traceability. It provides the bi-directional tracing between the tests and code artifacts making it useful when making a change to a system as the production code is changed frequently in the DevOps approaches.

The literature also shows that production code and unit test are not always synced. Production code is updated in different commits and unit tests in a different commit. Co-evolution of both of these artifacts is not possible. Manually creating these links is also not feasible. Having a tool that generated these links automatically is s relief for the developers. The literature also presents the tools such as SCHOTCH+, TCTracer does that automatically.

The literature also shows that there are dynamic and static approaches to create the traceability links. Using one approach to establish the traceability links does not give

an optimal result. The ensemble of these approaches gives better results. Name Similarity (NS), Name Contains (NC), Longest Common Subsequence (LCS), Last call before assert (LCBA), and Levenshtein Distance (LD) are a few of the techniques. Few other fault localization approaches and machine learning approaches are also implemented. But it is found that the traditional test-to-code traceability approaches (textual information based such as NC, LCS, etc.)

The literature also shows that most of the tools and approaches cater to the needs of Java based projects. The tools that are presented use Java languages and Junit for the test suits. Not much is done when it comes to C#. There is clearly a need for such tools that use C# language and test suits.

CHAPTER 3

METHODOLOGY

3.1. Approach

The proposed approach is an embed of different test-to-code traceability techniques. First the candidate traceability links are created between the test artifacts and code artifacts. The candidate links are created using the static information and the dynamic information. These links are evaluated and scores are assigned. Based on the score, it is predicted whether the link is an actual link between test and code artifact.

Static information can be easily collected without executing the actual code. Simple parsing can be done to gather this information. Traditional static techniques such as name similarity (NS) are incorporated in our approach as well. The static information includes the fully qualified names (FQNs) of methods and classes for both test artifacts and code artifacts. The text similarity and comparison-based approaches has been around and used in many approaches. An evaluation of these approaches tells us that the combination of these techniques significantly improves the quality of test-to-code traceability links [44].

Dynamic information is also utilized in our approach. It includes the call traces such as which method is called before the assert statement (LCBA) [45], which method is called from the test method and what is the depth of call. For dynamic information, the source code (system) needs to executed. But for large number of projects, executing each system is not feasible as it consumes a lot of time to run each system. For the very same reason, static information is as used in this approach.

Once information is collected, we have applied different techniques to assign a score. Some techniques simply give a score either 0 or 1. 0 means the candidate link is not a traceability link and 1 means the candidate link is a true traceability link. For some techniques we had to apply score scaling and applied threshold in order to predict the true traceability links.

Our approach also utilizes the method level information along with the class level information. The techniques are described in the following section.

3.2. Techniques

Our approach also utilizes different string comparison techniques. We have selected two variants of name convention (NC, NCC) techniques in our approach as described in Section 2. We have also used the Longest Common Subsequence technique and used two variants i.e. LCS-B and LCS-U as described in Section 2. Furthermore, we have used Tarantula (fault localization technique) and Term Frequency-Inverse Document Frequency (TF-IDF) as describe in section. Both of these techniques are statistical techniques. We have also used Last Call Before Assert (LCBA) as it is seen in the literature that It performs well for test-to-code traceability.

There are mainly two way to gather information and create the test-to-code traceability links: Static and Dynamic. The static analysis requires the physical files to be parsed and dynamic analysis requires the code to be executed and control flow information is collected [45]. Following are the techniques that will be used in our analysis:

3.2.1. Dynamic Techniques

The dynamic techniques that are used to get information are described as follow.

3.2.1.1. Naming Convention (NC)

Name convention (NC) compares the name of the test and name of the method. The "*test"* is removed from the test name and comparison is done. If the name of the test and method is same, the link is established. E.g. *calculateTax* is a method that is tested by test*CalculateTax* test.

If n_t is the name of test t (after removing test from it) and n_m is the name of method m:

$$score(t, m) = \begin{cases} 1 & if\ n_t\ equals\ n_m \\ 0 & otherwise \end{cases}$$

3.2.1.2. Naming Convention Contains (NCC)

Name convention contains (NCC) is a variant of NC but it does not compare the exact name, instead it looks for a substring that matches. After removing the "*test"* from the test name, if part of the method name matches with the test name, the link is established. E.g. *calculateTax* is the method that is tested by test*CalculateTaxPass* test.

If n_t is the name of test t (after removing test from it) and n_m is the name of method m:

$$score(t,m) = \begin{cases} 1 & if\ n_m\ is\ substring\ n_t \\ 0 & otherwise \end{cases}$$

3.2.1.3. Longest Common Subsequence Both (LCS-B)

Longest common subsequence (LCS) is a one of the name similarity (NS) technique. It finds the subsequence that has more characters in common. If the names are exactly the same, it has the highest score. LCS-B works on the same approach. If the test name and the method name are exactly same, the score is maximized at 1.

If n_t is the name of test t (after removing test from it) and n_m is the name of method m:

$$score(t,m) = \frac{|LCS(n_t,\ n_m)|}{\max\left(|n_t|,\ |n_m|\right)}$$

$LCS(n_t,\ n_m) = n_t$ when n_t is equal to the n_m i.e. test name and method name is exactly the same.

3.2.1.4. Longest Common Subsequence Unit (LCS-U)

LCS-U is another variant of LCS. It finds the subsequence that has more characters in common. Instead of finding the exact same name, if finds a substring just like NCC. If the test name contains the method name, the score is maximized at 1.

If n_t is the name of test t (after removing test from it) and n_m is the name of method m:

$$score(t, m) = \frac{|LCS(n_t,\ n_m)|}{|n_m|}$$

3.2.1.5. Levenshtein Distance

Levenshtein distance computes the distance between the test name and the method name. If the distance is lower, it means it takes a smaller number of edits to convert test name into method name, the more chances are the test is testing the method. The distance is normalized as to give a higher value [46].

If n_t is the name of test t and n_m is the name of method m:

$$score(t, m) = 1 - \frac{Levenshtein(n_t,\ n_m)|}{\max(|n_f|,\ |n_m|)}$$

3.2.1.6. Last Call Before Assert (LCBA)

Last call before assert (LCBA) looks for the last method called before the assert statement and assumes that it is the same method that is tested by the current test. Based on the assumption the link is established between the test and method. If the link is established the score is 1 otherwise 0.

If t is the test t and m is the method:

$$score(t, m) = \begin{cases} 1 & if\ m\ is\ called\ before\ assert \\ 0 & otherwise \end{cases}$$

3.2.1.7. Tarantula

Tarantula is used to find the suspiciousness of a test as described in the Section 2. If the value is higher, the higher the probability that the code is faulty. If c is the code entity, the suspiciousness can be defined as follows:

$$suspiciouness(c) = \frac{\frac{failedTests\ (c)}{totalFailedTests}}{\frac{passedTests\ (c)}{totalPassedTests} + \frac{failedTests(c)}{totalFailedTests}}$$

To calculate the score for the traceability link, we assume that all the test is passed except the one that is under consideration. This heuristic is used to identify the method under test. If T is the set of all tests, and the method m tested by t is the

method:

$$score(t,m) = \frac{1}{\frac{|\{t \in T: m \in t\}| - 1}{|T| - 1} + 1}$$

3.2.1.8. Term Frequency-Inverse Document Frequency (TF-IDF)

TF-IDF is used to get the frequency of term inside the document and the documents repository. If the term has high frequency inside the document and low frequency in rest of the documents, its significance is higher with respect to that document. The same approach is applied her while creating the traceability link. The method name is considered as the term and all tests are considered as the documents. If a method has high frequency in test and not in other tests, it is possible that test is testing that specific method.

If t is the test t and m is the method:

$$score(t,m) = tf(t,m).\ idf(m)$$

If M is the set of all methods and T is the set of all tests, then:

$$tf(t,m) = \ln\left(1 + \frac{1}{|\{m \in M: m \in t\}|}\right)$$

$$idf(m) = \ln\left(1 + \frac{1}{|\{t \in T: m \in t\}|}\right)$$

3.2.2. Static Techniques

The static techniques that are used to get information are described as follow.

3.2.2.1. Static Naming Convention (NC)

Static name convention now includes another condition. Instead of just comparing the names of the methods, it also compares the name of test class and class-under-test. The comparison is done after removing "*test"* from the test name and test class name. If the name of the test and test class is the same as the method name and class name, the link is established.

If n_t is the name of test t , n_{tc} in test class name (after removing test from it), n_m is

the name of method and n_{mc} is the name of class under test, then:

$$score(t,m) = \begin{cases} 1 & if\ n_t\ equals\ n_m\ \text{^}\ n_{tc}\ equals\ n_{mc} \\ 0 & otherwise \end{cases}$$

3.2.2.2. Static Naming Convention Contains (NCC)

Static name convention contains now includes another condition. Instead of just comparing the names of the methods, it also compares the name of test class and class-under-test. The comparison is done after removing "test" from the test method name and test class name. If the name of the test and test class contains the method name and class name, the links are established.

If n_t is the name of test t , n_{tc} in test class name (after removing test from it), n_m is the name of method and n_{mc} is the name of class under test, then:

$$score(t,m) = \begin{cases} 1 & if\ n_m\ \ substring\ of\ n_t\ \text{^}\ n_{mc}\ \ substring\ of\ n_{tc} \\ 0 & otherwise \end{cases}$$

3.2.2.3. Static Longest Common Subsequence Both (LCS-B)

LCS-B is used similar to dynamic approach. The score is calculated the same way.

3.2.2.4. Static Longest Common Subsequence Unit (LCS-U)

LCS-U is used similar to dynamic approach. The score is calculated the same way.

3.2.2.5. Static Levenshtein Distance

Levenshtein distance is used similar to dynamic approach. The score is calculated the same way.

3.2.3. Score Scaling

In our approach following are the two score scaling techniques are applied:

3.2.3.1. Call Depth Discounting

The method that is closer to the test in the call stack is a method that is under test. We have included the call depth discounting factor that discounts the test-to-method pair based on the distance between them.

If t is the test t and m is the method:

$$score_d(t,m) = \ score(t,m) \ . \gamma^{(dist(t,m)-1)} \quad \text{where } \gamma \in [0,1]$$

If a method is directly called from the test then $dist(t,m)$ becomes 1, the discount factor becomes 0 because of subtracting one of it. The discount factor does not need to be applied in this case.

3.2.3.2. Normalization

The concept of normalization is very useful in the statistical and classification problems, especially in machine learning approaches. And it clearly improves the accuracy of a classification problem [48]. As the score is used to rank and predict the true traceability link. But the score can vary in different techniques. In order to

Technique	Score	Normalized	Threshold - Method Level	Threshold - Class Level
NC	0 or 1	-	-	-
NCC	0 or 1	-	-	-
LCS-B	[0, 1]	Yes	0.30	0.30
LCS-U	[0, 1]	Yes	0.75	0.25
Levenshtein	[0, 1]	Yes	0.30	0.25
LCBA	0 or 1	-	-	-
Tarantula	[0, 1]	Yes	0.65	0.99
TF-IDF	[0, 1]	Yes	0.35	0.20
Static NC	0 or 1	-	-	-
Static NCC	0 or 1	-	-	-
Static LCS-B	[0, 1]	Yes	0.30	0.25
Static LCS-U	[0, 1]	Yes	0.75	0.25
Static Levenshtein	[0, 1]	Yes	0.30	0.25
Combined	[0, 1]	Yes	0.35	0.20

Table 1 - Score Range, Normalization and Thresholds

maintain the score on a single scale, we have used the normalization approach to scale the score between 0 and 1.

If t is the test t and m is the method:

$$score_n(t,m) = \frac{score_d(t,m) - \min\left(\{score_d(t,m) \mid m \in t\}\right)}{\min\left(\{score_d(t,m) \mid m \in t\}\right) - \ \max\left(\{score_d(t,m) \mid m \in t\}\right)}$$

Then thresholds τ are applied on the values to make the score either 0 or 1 where 1 describes the true traceability link.

The Table 1 shows the normalization and threshold values applied on the approaches.

3.3. Link Prediction

The links are constructed with the help of prediction. Two types of link prediction is done; class level and method level. The techniques are first applied to the test and methods, and then it is applied on the test classes and tested classes.

3.3.1. Prediction at Method Level

The individual traceability techniques are executed at the method level and score are calculated. A matrix is formed with the result/score of each technique. If T is the set of all tests and M is the set of all methods, the matrix can be defined as:

$$M \in \mathbb{R}^{|T| \times |M|}$$

Each element of the matrix (M_{tm}) represents a score for the test to method pair $(t, m) \in (T \times M)$.

Then each matrix is normalized and combined to get another matrix. Threshold (τ) is then applied on all of the matrix for each technique (Table 1). The score above the threshold is considered as 1 i.e. it represents a true traceability link. The traceability link can be constructed as:

$$TM = \{(t, m) \in T\ X\ M \mid M_{tm} \geq \tau\}$$

3.3.2. Prediction at Class Level

Just as the method level prediction, same steps are applied for the class level. A matrix is formed with the result/score of each technique. If TC presents the set containing all test classes and MC is the set containing all classes under test, the matrix can be defined as:

$$C \in \mathbb{R}^{|TC| \times |MC|}$$

Each element of the matrix ($C_{c_t c_m}$) represents a score for the test class-class pair $(c_t, c_m) \in (TC \times MC)$.

Then each matrix is normalized and combined to get another matrix. Threshold (τ) is then applied on all of the matrix for each technique (Table 1). The score above the threshold is considered as 1 i.e. it represents a true traceability link. The traceability link can be constructed as:

$$TC = \{(c_t, c_m) \in (TC \times MC) \mid C_{c_t c_m} \geq \tau \}$$

3.4. Implementation

The developed prototype tool is compatible with the C# (.NET 4.7, .NET Core 3.1, and .NET 5,6) applications that use the NUnit testing framework as their backbone for unit testing. There are a few other testing frameworks such as XUnit and MS Test are also, but the tool is focused on NUnit tests.

The tool uses both information and dynamic information to rank the traceability links. The static information is collected simply from the Assembly/Dynamic Link Library (.dll) file for the Test Project. Assembly is the collection of types (e.g. Test classes and methods) that are built to form a logical unit. These are the building block of .NET Applications [51]. The tool has an Assembly Analyzer module that parses the Test project assembly using the concept of Reflection [52] and gets the information regarding test class, test method, and the methods that are called inside the test method.

The dynamic information is collected through execution call traces. To collect the call traces, there is no direct way of gather this information. This information is collected through the concept of Aspect Oriented Programming (AOP). It is a concept of applying common routines to whole application e.g. Logging or exception handling [53]. PostSharp [55] is a utility that provides an easy way of intercepting the methods. we have used this utility for instrumentation to attach some extra information at each test method calls. It uses the concept reflection to add extra information at runtime. Whenever a test method is starts, the dynamic call traces are written to the output as show in the Figure 3.

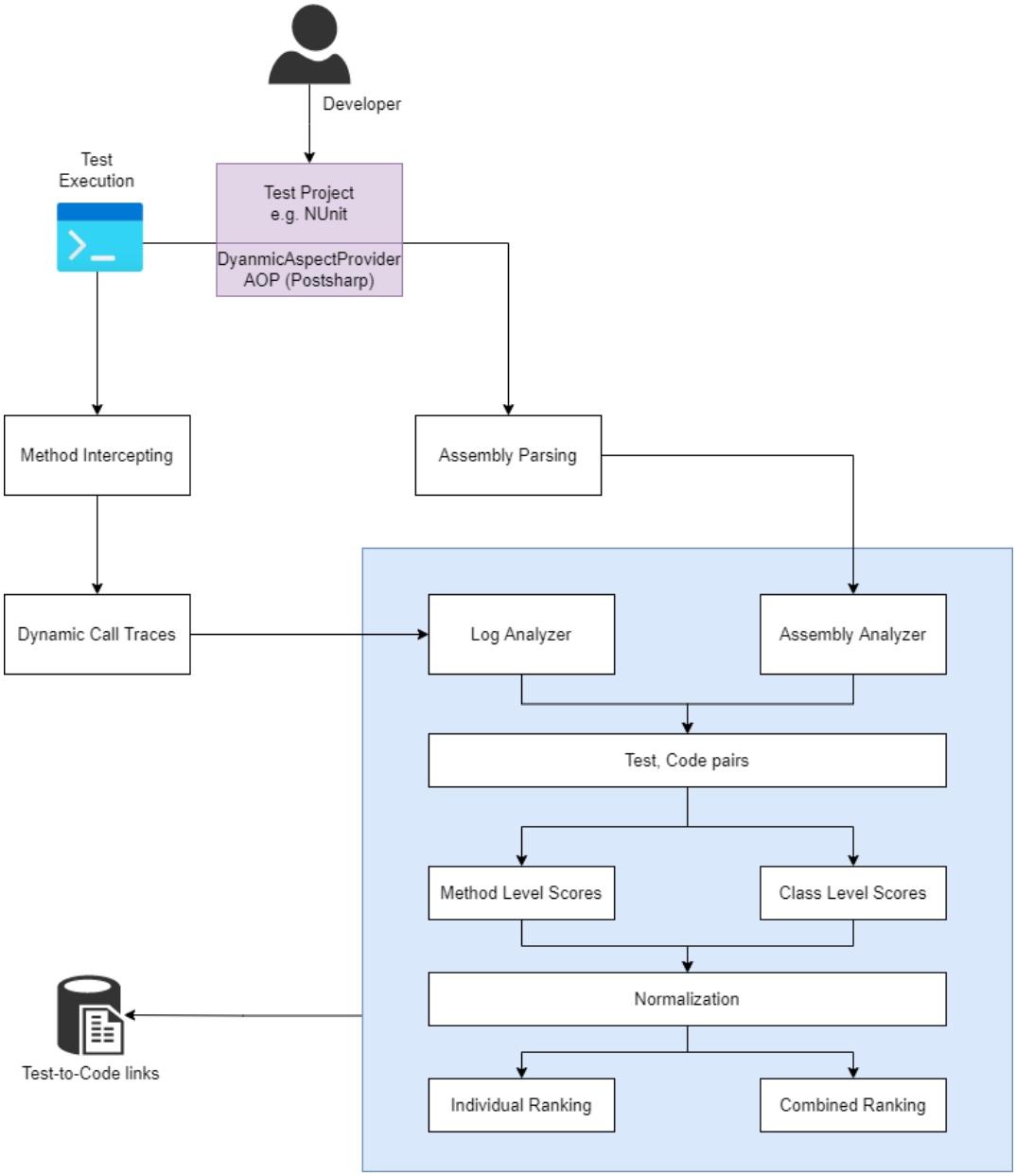

Figure 2 - Implementation of Tracer Tool

These output traces are collected in the log file. The log data represent the raw data that needs to be converted into meaningful information. The logs are then parsed to collect the method hits made from the unit test. The information is parsed and saved in the form of a JSON file for both dynamic and static information as shown in Figure 4. This information is then used to form the candidate test to code links.

First, the score at the method level is calculated. The scores are calculated for the individual technique for each test-method to method pair. The scores are scaled using normalization because some of the techniques do not have a range between 0 and 1. The score is then combined by taking averaging of all scores for a pair and normalized again to form a

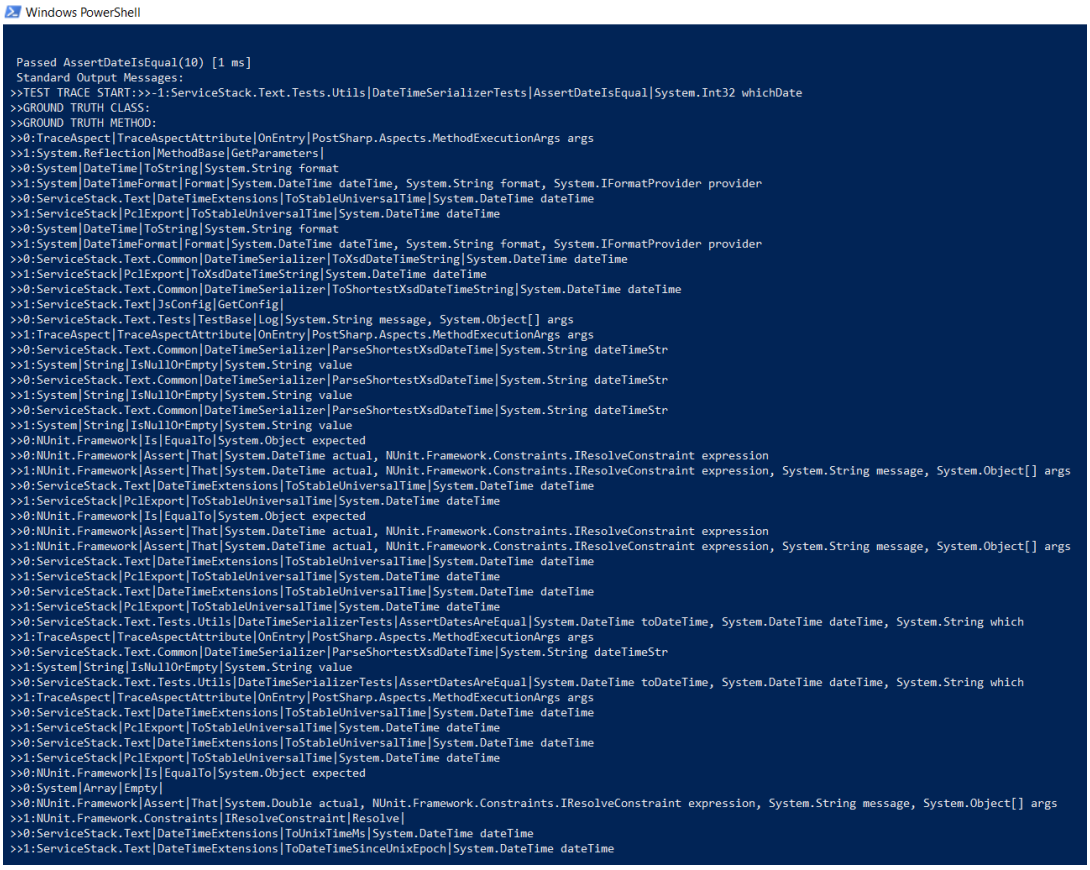

Figure 3 - Dynamic Call Traces Output

combined score. The same process is then repeated for the class level. The score for each test-class to class pair is calculated, and normalized. Then the scores are combined and normalized again. For each pair, a threshold technique is applied that does not give discreet values. Based on that, the candidate link is ranked whether it is a true traceability link or not.

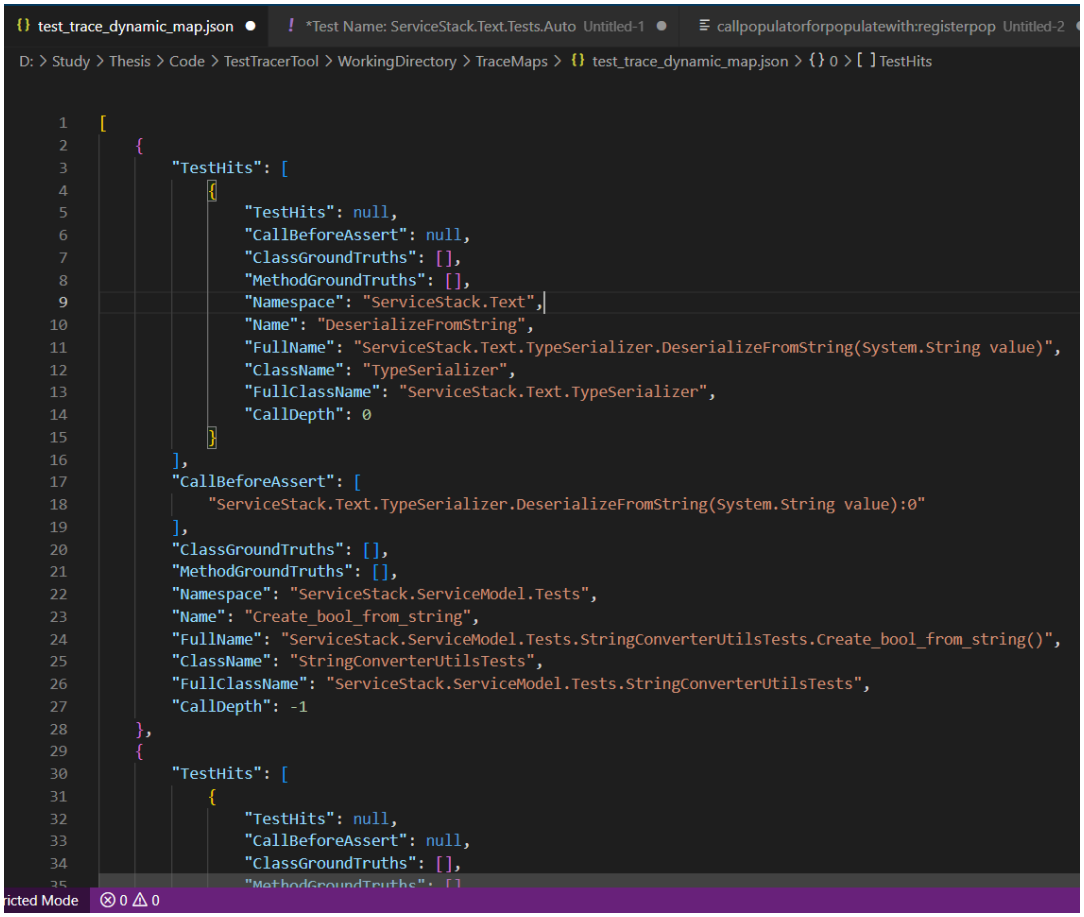

Figure 4 - JSON map for test method hits

CHAPTER 4

EVALUATION

The evaluation of the tool is done on the open source projects after defining the ground truths.

4.1. Subjects

For the evaluation purpose, two open source projects are selected that have unit tests written with the NUnit testing framework.

- ServiceStack.Text (https://github.com/ServiceStack/ServiceStack.Text)
- Aeron.NET (https://github.com/AdaptiveConsulting/Aeron.NET)

The source code for both of these projects is available at the GitHub platform. Both of the projects are using different naming conventions for the test class names and test method names.

4.2. Ground Truths

The ground truth is developed for both of the subjects to measure the quality of the techniques used in the tool. There was no existing ground truth available. For this purpose, a team of two developers who work in the well known software houses and have an experience around 4-5 years helped in establishing the ground truth.

For the ground truth, the tool exposes an attribute/annotation built in C#. This attribute can be applied on both levels i.e. class and method. For class level, the attribute is applied on the class level. The attribute takes an argument of the fully qualified names for the classes under test i.e. classes which are tested by the test class. Similarly, the attribute is then applied on the methods. For method it takes an argument of fully qualified names for methods under test. In this way, the ground truth is established for both class and method levels. The ground truth annotation can be seen in Figure 5.

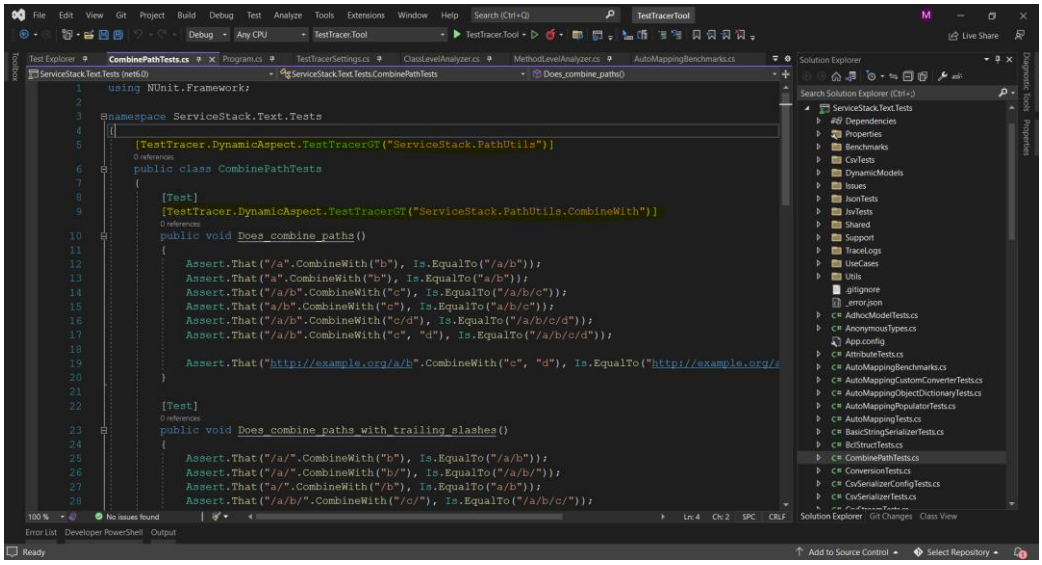

Figure 5 – Ground truth at Method level and Class level attribute

4.3. Measures

For the evaluation purposes, we have selected one of the basic measures: precision, recall, accuracy and F1 score. These measures are one of the basic evaluation measures for the classification problems. Precision tells us the ratio of all true positives out of all positives, and Recall tell us the ratio of all true positive out of all predictions. Accuracy tell us the ratio of how accurate the model is at prediction [56].

F1 Score is a similar metric that maintain the ratio of both precision and recall at optimal. It is a harmonic mean of precision and recall. Instead of maintaining precision and recall separately, it is easier to maintain this one metric that maximizes both of these metrics [56].

Precision, Recall, Accuracy and F1 Score can be calculated as follow:

$$Precision = \frac{TP}{TP + FP}$$

$$Recall = \frac{TP}{TP + FN}$$

$$Accuracy = \frac{TP + TN}{Total}$$

$$F1\ Score = 2 \times \frac{Precision \times Recall}{Precision + Recall}$$

4.4. Results

The tool outputs the results for each test to code candidate pair and rank it as a true traceability link. Based on the scores, thresholds are applied and each candidate

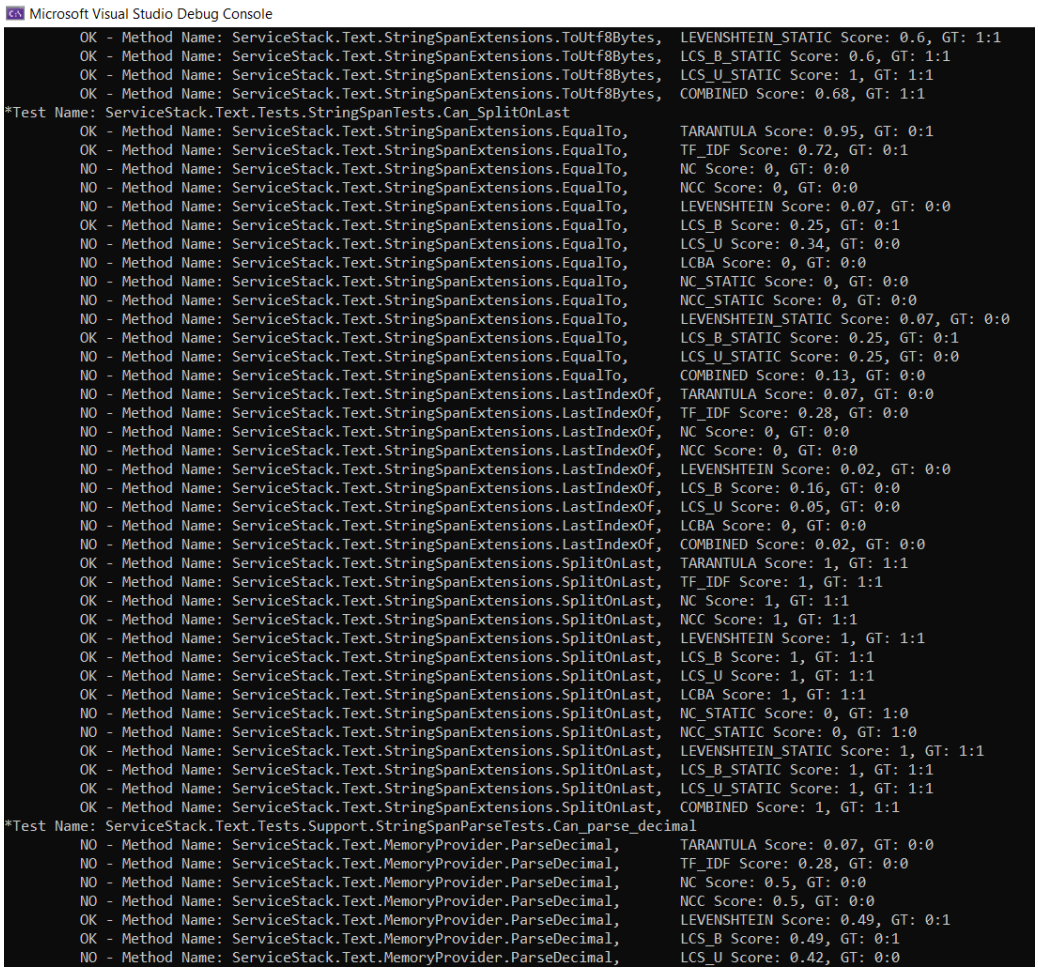

Figure 6 - Candidate Pair Ranking

pair is ranked telling us whether a pair represents the true traceability link or not. The output of these can be seen in the Figure 5.

4.4.1. Prediction at Method Level

Technique	Precision	Recall	Accuracy	F1Score	Efficiency
NC	**100**	12	57	22	100
NCC	95	45	72	61	98
Levenshtein	88	72	82	79	90
LCS B	83	75	80	79	86
LCS U	94	75	80	**79**	86
Tarantula	63	92	70	75	48
TF IDF	56	70	59	62	48
LCBA	76	65	73	70	81
NC Static	**100**	3	39	5	100
NCC Static	**100**	3	39	5	100
Levenshtein Static	93	76	81	**84**	91
LCS B Static	88	78	80	**83**	82
LCS U Static	94	78	83	**85**	91
Combined	**97**	**78**	**88**	**86**	**98**

Table 2 - ServiceStack.Text - Method level metrics

Technique	Precision	Recall	Accuracy	F1Score	Efficiency
NC	**0**	0	50	0	100
NCC	86	23	60	36	96
Levenshtein	100	54	77	70	100
LCS B	100	42	71	59	100
LCS U	95	69	83	**80**	96
Tarantula	74	100	83	**85**	65
TF IDF	62	100	69	76	38
LCBA	54	58	54	56	50
NC Static	0	0	26	0	100
NCC Static	0	0	26	0	100
Levenshtein Static	100	54	66	70	100
LCS B Static	100	31	49	47	100
LCS U Static	95	69	74	**80**	89
Combined	**78**	**86**	**85**	**86**	**73**

Table 3 - Aeron.NET - Method level metrics

The method level score for ServiceStack.Text and Aeron.NET are shown in Table 2 and 3 respectively. NCC seems to perform better at the method level than NC.

Both NC and NCC both are better choices where proper naming conventions are followed. NC performs poorly when the method names are long and descriptive. For example, NC precision and F1 Score is 0 for Aeron.NET as shown in Table 3. In this project, there are no methods found that have the same name as test method names. LCS-U and LCS-U static also perform well in the method level scoring due to their better F1 Score. LCBA does not perform well for both of the projects. The reason for this is, the last call before asserts are usually the helper methods. It is not always the same method that is under test.

If we see at the combined score for both projects, it seems to be performing well. It provides a better F1 Score and better accuracy than each of the individual techniques.

4.4.2. Prediction at Class Level

The class level score for ServiceStack.Text and Aeron.NET are shown in the Table 4 and 5 respectively. NCC seems to perform better at the class level than NC. Both NC and NCC both are better choices where proper naming conventions are followed. NC performs poorly when the method names are long and descriptive. But at class level, mostly the test class names have almost the same names as the class under test. LCS-B performs better at class level then the LCS-U because it has better F1 Score and accuracy. Similarly, LCS-U static seems to be performing better than LCS-B static.

LCBA does not perform well for both of the projects. The reason for this is, last call before asserts are usually the helper methods. It is not always the same method which is under test.

Technique	Precision	Recall	Accuracy	F1Score	Efficiency
NC	**100**	25	59	40	100
NCC	**100**	31	62	48	100
Levenshtein	92	69	79	79	92
LCS B	100	75	86	**86**	100
LCS U	86	75	79	**80**	85
Tarantula	83	62	72	71	85
TF IDF	92	69	79	79	92
LCBA	69	69	66	69	62
NC Static	**100**	29	47	44	100
NCC Static	**100**	36	53	53	100
Levenshtein Static	91	71	74	80	80
LCS B Static	100	64	74	**78**	100
LCS U Static	100	71	79	**83**	100
Combined	**88**	**88**	**86**	**88**	**85**

Table 4 - ServiceStack.Text – Class level metrics

Technique	Precision	Recall	Accuracy	F1Score	Efficiency
NC	**100**	50	77	67	100
NCC	**100**	50	77	67	100
Levenshtein	90	90	91	90	92
LCS B	83	100	91	**95**	92
LCS U	83	100	91	**91**	83
Tarantula	57	80	64	67	50
TF IDF	90	90	91	90	92
LCBA	60	90	68	72	50
NC Static	**100**	56	75	71	100
NCC Static	**100**	56	75	71	100
Levenshtein Static	89	89	88	89	86
LCS B Static	90	100	94	**95**	86
LCS U Static	90	100	94	**95**	86
Combined	**91**	**100**	**95**	**95**	**92**

Table 5 - Aeron.Net - Class level metrics

If we see at the combined score for both projects, it seems to be performing well. It provides better F1 Score and better accuracy than each of the individual techniques.

CHAPTER 5

CONCLUSION AND FUTURE WORK

5.1. Conclusion

Test to code traceability links is really helpful in maintaining the code artifacts. It provides traceability information when the code is changed at production. It tells which test needs to be updated when a code is changed. Not only that, it provides an efficient selection of tests that needs to be run when regression or integration testing is done. But developing these links manually and maintaining them is a difficult task for the developers.

The prototype tool presented in this research helps in developing these links automatically. It reduces the burden on the developers by doing it automatically. It creates the link based on the traditional techniques. The tool uses a combined approach that gives a better result in terms of better accuracy and f1 score as discussed and presented in Chapter 4.

The tool provides traceability links between two types. 1) test method-to-method traceability links 2) test class-to-class traceability. The tool provides more effective results at both class and method levels than the individual technique as shown in Chapter 4.

5.2. Contribution

.NET (C#) is among the most used and trending programming languages. Many enterprise applications are being developed using the new framework such as .Net Core and .NET 5,6 as it provides the benefits of cross-platform and performance improvements.

There are no tools available that can test and code artifacts automatically for these projects. The tools that are available work with Java programming languages. A couple of tools are available for C# for this purpose, but the code needs to be updated for this purpose. A developer needs to insert code inside the classes and

methods to generate the traceability links. Which again, puts the burden on the developers and it is again manual work.

The tool presented does not require any code changes inside the code. It simply uses the concept of Aspect Oriented Programming (AOP), to inject the code at runtime and collect the call traces. Then based on these traces, the tool automatically develops the traceability links.

5.3. Future Work

The prototype tool presented in this research works with the NUnit testing frameworks. There are a few other testing frameworks that are also popular and adopted by the developers as well. These testing frameworks include the XUnit and MS Test. The tool needs to be generalized to cater to these two frameworks as well. The basics of these testing frameworks are the same. So, there is a need for these frameworks to be incorporated into the tool as well.

The tool does not clearly distinguish between interface/abstract and concrete implementations. Some work is required at this end as well. The tool needs to filter out these abstract implementations as well.

References

[1] Sandler, C., Badgett, T., & Myers, G. (2013). *The art of software testing*. Hoboken, N.J.: Wiley. doi: 10.1002/9781119202486

[2] Garousi, V., & Zhi, J. (2013). A survey of software testing practices in Canada. *Journal Of Systems And Software*, *86*(5), 1354-1376. doi: 10.1016/j.jss.2012.12.051

[3] Abhijit A. Sawant, Pranit H. Bari, P. M. Chawan (2012). Software Testing Techniques and Strategies. *International Journal of Engineering Research and Applications*, 2248-9622.

[4] Tosun, A., Ahmed, M., Turhan, B., & Juristo, N. (2018). On the effectiveness of unit tests in test-driven development. *Proceedings Of The 2018 International Conference On Software And System Process*. doi: 10.1145/3202710.3203153

[5] Cleland-Huang, J., Gotel, O., Huffman Hayes, J., Mäder, P., & Zisman, A. (2014). Software traceability: trends and future directions. *Future Of Software Engineering Proceedings*. doi: 10.1145/2593882.2593891

[6] Gotel, O., Cleland-Huang, J., Hayes, J., Zisman, A., Egyed, A., & Grünbacher, P. et al. (2011). Traceability Fundamentals. *Software And Systems Traceability*, 3-22. doi: 10.1007/978-1-4471-2239-5_1

[7] Königs, S., Beier, G., Figge, A., & Stark, R. (2012). Traceability in Systems Engineering – Review of industrial practices, state-of-the-art technologies and new research solutions. *Advanced Engineering Informatics*, *26*(4), 924-940. doi: 10.1016/j.aei.2012.08.002

[8] Wiederseiner, C., Garousi, V., & Smith, M. (2011). Tool Support for Automated Traceability of Test/Code Artifacts in Embedded Software Systems. *2011IEEE 10Th International Conference On Trust, Security And Privacy In Computing And Communications*. doi: 10.1109/trustcom.2011.151

[9] Qusef, A. (2013). Test-to-code traceability: Why and how?. *2013 IEEE Jordan Conference On Applied Electrical Engineering And Computing Technologies (AEECT)*. doi: 10.1109/aeect.2013.6716450

[10] Krajewski, R. (2022). The State Of C# Development In 2022. Retrieved 14 July 2022, from https://www.ideamotive.co/blog/the-state-of-csharp-development

[11] Saltali, I. (2022). What is new in DotNET 5.0. Retrieved 14 July 2022, from https://www.kloia.com/blog/what-is-new-in-.net-5.0

[12] Lander, R. (2022). Announcing .NET 6 -- The Fastest .NET Yet. Retrieved 14 July 2022, from https://devblogs.microsoft.com/dotnet/announcing-net-6/#performance

[13] .NET customers showcase | See what devs are building. (2022). Retrieved 14 July 2022, from https://dotnet.microsoft.com/en-us/platform/customers

[14] Ridgway, J. and McCusker, S. and Pead, D. (2004) 'Literature review of e-assessment.', Project Report. Futurelab, Bristol. https://dro.dur.ac.uk/1929/

[15] Runeson, P. (2006). A survey of unit testing practices. *IEEE Software, 23*(4), 22-29. doi: 10.1109/ms.2006.91

[16] Daka, E., & Fraser, G. (2014). A Survey on Unit Testing Practices and Problems. *2014 IEEE 25Th International Symposium On Software Reliability Engineering*. doi: 10.1109/issre.2014.11

[17] A., A., Akour, M., Alazzam, I., & Hanandeh, F. (2016). Regression Test-Selection Technique Using Component Model Based Modification: Code to Test Traceability. *International Journal Of Advanced Computer Science And Applications, 7*(4). doi: 10.14569/ijacsa.2016.070411

[18] Vidacs, L., & Pinzger, M. (2018). Co-evolution analysis of production and test code by learning association rules of changes. *2018 IEEE Workshop On Machine Learning Techniques For Software Quality Evaluation (Maltesque)*. doi: 10.1109/maltesque.2018.8368456

[19] Kicsi, A., Vidács, L., & Gyimóthy, T. (2020). TestRoutes. *Proceedings Of The 17Th International Conference On Mining Software Repositories*. doi: 10.1145/3379597.3387488

[20] Elsner, D., Hauer, F., Pretschner, A., & Reimer, S. (2021). Empirically evaluating readily available information for regression test optimization in continuous integration. *Proceedings Of The 30Th ACM SIGSOFT International Symposium On Software Testing And Analysis*. doi: 10.1145/3460319.3464834

[21] Parizi, R., Lee, S., & Dabbagh, M. (2014). Achievements and Challenges in State-of-the-Art Software Traceability Between Test and Code Artifacts. *IEEE Transactions On Reliability, 63*(4), 913-926. doi: 10.1109/tr.2014.2338254

[22] Rompaey, B., & Demeyer, S. (2009). Establishing Traceability Links between Unit Test Cases and Units under Test. *2009 13Th European Conference On Software Maintenance And Reengineering*. doi: 10.1109/csmr.2009.39

[23] Csuvik, V., Kicsi, A., & Vidács, L. (2019). Evaluation of Textual Similarity Techniques in Code Level Traceability. *Computational Science And Its*

Applications – ICCSA 2019, 529-543. doi: 10.1007/978-3-030-24305-0_40

[24] Yan, Q., Li, Y., Wu, Y., & Zhou, J. (2021). DFlow : A Data Flow Analysis Tool for C/C++. *IEEJ Transactions On Electrical And Electronic Engineering*, *16*(12), 1635-1641. doi: 10.1002/tee.23467

[25] Ma'ayan, D. (2018). The quality of junit tests. *Proceedings Of The 1St International Workshop On Software Qualities And Their Dependencies*. doi: 10.1145/3194095.3194102

[26] Jones, J., & Harrold, M. (2005). Empirical evaluation of the tarantula automatic fault-localization technique. *Proceedings Of The 20Th IEEE/ACM International Conference On Automated Software Engineering - ASE '05*. doi: 10.1145/1101908.1101949

[27] Christian, H., Agus, M., & Suhartono, D. (2016). Single Document Automatic Text Summarization using Term Frequency-Inverse Document Frequency (TF-IDF). *Comtech: Computer, Mathematics And Engineering Applications*, *7*(4), 285. doi: 10.21512/comtech.v7i4.3746

[28] Kicsi, A., Vidács, L., & Gyimóthy, T. (2020). TestRoutes. *Proceedings Of The 17Th International Conference On Mining Software Repositories*. doi: 10.1145/3379597.3387488

[29] Meimandi Parizi, R., Kasem, A., & Abdullah, A. (2015). Towards Gamification in Software Traceability: Between Test and Code Artifacts. *Proceedings Of The 10Th International Conference On Software Engineering And Applications*. doi: 10.5220/0005555503930400

[30] Gergely, T., Balogh, G., Horváth, F., Vancsics, B., Beszédes, Á., & Gyimóthy, T. (2018). Differences between a static and a dynamic test-to-code traceability recovery method. *Software Quality Journal*, *27*(2), 797-822. doi: 10.1007/s11219-018-9430-x

[31] Gergely, T., Balogh, G., Horváth, F., Vancsics, B., Beszédes, Á., & Gyimóthy, T. (2018). Analysis of Static and Dynamic Test-to-code Traceability Information. *Acta Cybernetica*, *23*(3), 903-919. doi: 10.14232/actacyb.23.3.2018.11

[32] Qusef, A., Bavota, G., Oliveto, R., Lucia, A., & Binkley, D. (2012). Evaluating test-to-code traceability recovery methods through controlled experiments. *Journal Of Software: Evolution And Process*, *25*(11), 1167-1191. doi: 10.1002/smr.1573

[33] Aljawabrah, N., & Qusef, A. (2019). TCTracVis. Proceedings Of The Second International Conference On Data Science, E-Learning And Information Systems - DATA '19. doi: 10.1145/3368691.3368735

[34] (2022). TCTracer: Establishing test-to-code traceability links using dynamic and static techniques. Empirical Software Engineering, 27(3). doi: 10.1007/s10664-021-10079-1

[35] Rubasinghe, I., Meedeniya, D., & Perera, I. (2018). Automated Inter-artefact Traceability Establishment for DevOps Practice. *2018 IEEE/ACIS 17Th International Conference On Computer And Information Science (ICIS)*. doi: 10.1109/icis.2018.8466414

[36] Ghafari, M., Ghezzi, C., & Rubinov, K. (2015). Automatically identifying focal methods under test in unit test cases. *2015 IEEE 15Th International Working Conference On Source Code Analysis And Manipulation (SCAM)*. doi: 10.1109/scam.2015.7335402

[37] Rafati, A., Lee, S., Parizi, R., & Zamani, S. (2015). A test-to-code traceability method using .NET custom attributes. *Proceedings Of The 2015 Conference On Research In Adaptive And Convergent Systems*. doi: 10.1145/2811411.2811553

[38] Aljawabrah, N., Gergely, T., Misra, S., & Fernandez-Sanz, L. (2021). Automated Recovery and Visualization of Test-to-Code Traceability (TCT) Links: An Evaluation. *IEEE Access, 9*, 40111-40123. doi: 10.1109/access.2021.3063158

[39] Aung, T., Huo, H., & Sui, Y. (2019). Interactive Traceability Links Visualization using Hierarchical Trace Map. *2019 IEEE International Conference On Software Maintenance And Evolution (ICSME)*. doi: 10.1109/icsme.2019.00059

[40] Csuvik, V., Kicsi, A., & Vidacs, L. (2019). Source Code Level Word Embeddings in Aiding Semantic Test-to-Code Traceability. *2019 IEEE/ACM 10Th International Symposium On Software And Systems Traceability (SST)*. doi: 10.1109/sst.2019.00016

[41] Liu, Y., Lin, J., Zeng, Q., Jiang, M., & Cleland-Huang, J. (2020). Towards Semantically Guided Traceability. *2020 IEEE 28Th International Requirements Engineering Conference (RE)*. doi: 10.1109/re48521.2020.00043

[42] Laghari, G., Dahri, K., & Demeyer, S. (2018). Comparing Spectrum Based Fault Localisation Against Test-to-Code Traceability Links. *2018 International Conference On Frontiers Of Information Technology (FIT)*. doi: 10.1109/fit.2018.00034

[43] White, R., & Krinke, J. (2018). TestNMT: function-to-test neural machine translation. *Proceedings Of The 4Th ACM SIGSOFT International Workshop On NLP For Software Engineering*. doi: 10.1145/3283812.3283823

[44] Qusef, A., Bavota, G., Oliveto, R., De Lucia, A., & Binkley, D. (2014). Recovering test-to-code traceability using slicing and textual analysis. *Journal Of Systems And Software*, *88*, 147-168. doi: 10.1016/j.jss.2013.10.019

[45] Gergely, T., Balogh, G., Horváth, F., Vancsics, B., Beszédes, Á., & Gyimóthy, T. (2018). Differences between a static and a dynamic test-to-code traceability recovery method. *Software Quality Journal*, *27*(2), 797-822. doi: 10.1007/s11219-018-9430-x

[46] Yujian, L., & Bo, L. (2007). A Normalized Levenshtein Distance Metric. *IEEE Transactions On Pattern Analysis And Machine Intelligence*, *29*(6), 1091-1095. doi: 10.1109/tpami.2007.1078

[47] András Kicsi, László Tóth, and László Vidács. (2018). Exploring the benefits of utilizing conceptual information in test-to-code traceability. Proceedings of The 6Th International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering, 8–14. doi:10.1145/3194104.3194106

[48] Jayalakshmi, T., & Santhakumaran, A. (2011). Statistical Normalization and Back Propagationfor Classification. *International Journal Of Computer Theory And Engineering*, 89-93. doi: 10.7763/ijcte.2011.v3.288

[49] Kicsi, A., Csuvik, V., & Vidacs, L. (2021). Large Scale Evaluation of Natural Language Processing Based Test-to-Code Traceability Approaches. IEEE Access, 9, 79089-79104. doi: 10.1109/access.2021.3083923

[50] Jane Cleland-Huang, Orlena Gotel, and Andrea Zisman. 2012. Software and Systems Traceability. Springer Publishing Company, Incorporated.

[51] Assemblies in .NET. (2022). Retrieved 9 August 2022, from https://docs.microsoft.com/en-us/dotnet/standard/assembly/

[52] Reflection (C#). (2022). Retrieved 9 August 2022, from https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/reflection

[53] Clarke, J. (2009). Platform-Level Defenses. *SQL Injection Attacks And Defense*, 377-413. doi: 10.1016/b978-1-59749-424-3.00009-8

[54] Groves, M. (2013). The Fundamentals of AOP. *AOP in .NET: Practical Aspect-Oriented Programming*, Shelter Island: Manning.

[55] AOP in .NET | Aspect-Oriented Programming – PostSharp. (2022). Retrieved 9 August 2022, from https://www.postsharp.net/aop.net

[56] Precision vs Recall. (2022). Retrieved 9 August 2022, from https://medium.com/@shrutisaxena0617/precision-vs-recall-386cf9f89488