

https://dspace.org/

Analysing Test-Driven Development Adherence in Open-Source Projects Using Test-to-Code Traceability Links

Kirui, Gerald Kipruto

2024
University of the Witwatersrand, Johannesburg
https://hdl.handle.net/10539/45426
Kirui, Gerald Kipruto. (2024). Analysing Test-Driven Development Adherence in Open-Source Projects Using Test-to-Code Traceability Links [Masters dissertation, University of the Witwatersrand, Johannesburg]. WIReDSpace. https://hdl.handle.net/10539/45426

Downloaded from a DSpace repository. https://dspace.org/

Analysing Test-Driven Development Adherence in Open-Source Projects Using Test-to-Code Traceability Links

Gerald K. Kirui

A dissertation submitted to the Faculty of Engineering and the Built Environment, University of the Witwatersrand, Johannesburg, in fulfilment of the requirements for the degree of Master of Science in Engineering.

Johannesburg, August 2024

Declaration

I declare that this dissertation is my own, unaided work, except where otherwise acknowledged. It is being submitted for the degree of Master of Science in Engineering to the University of the Witwatersrand, Johannesburg. It has not been submitted before for any degree or examination to any other university.

Signed this 31st day of July 2024



Gerald K. Kirui

Abstract

Several studies have been conducted to determine the impact of Test-Driven Development (TDD) on software quality. Many studies utilise test-to-code traceability tools and strategies to detect TDD adherence as part of their methodology. However, most test-to-code traceability tools rely on filename-based matching algorithms, which suffer from low recall; therefore, most TDD detection methods are not accurate. This study aims to assess the effectiveness of a statement-based matching algorithm over a filename-based one and whether it can be used to detect the TDD adherence of a software project. The filename-based and statement-based matching algorithms are implemented in Python. To evaluate these algorithms, 500 tests from sixteen Java projects (encompassing frameworks, libraries, and tools for data processing, testing, and web services) are used. These projects range in size from 5,115 lines of code (LoC) to 378,167 LoC. This evaluation helps in understanding the performance of the algorithms through their weighted F1-scores. A mathematical function is created from first principles to correlate method coverage and time deltas with the test-with-development (TWD) adherence of a project. Thereafter, 100 Java projects are used to demonstrate the utility of this function. The results show that the statement-based matching algorithm, with a weighted F1-score of 0.771 and a 95% confidence interval of [0.741, 0.786], is more accurate than the filename-based matching algorithm, with a weighted F1-score of 0.218 and a 95% confidence interval of [0.193, 0.246]. Additionally, the results show that the relative TWD adherence of eleven projects is found to be highly correlated with TDD scores from surveys ($r_s = 0.723$ and p-value $= 0.012$), and the method coverage of eighteen projects is found to be highly correlated with code coverage obtained from Codecov ($r_s = 0.744$ and p-value $= 0.0003$). The literature review reveals that no studies have explored the correlation between TDD adherence and the development step size in software projects. An investigation is conducted using a statement-based test-to-code traceability tool and relative TWD adherence values to determine if TDD projects have smaller development steps compared to non-TDD projects. The study finds no significant correlation between median production method churn and relative TWD adherence of projects in all but one out of eight cases.

Acknowledgements

I wish to extend my sincere thanks to my supervisor, Dr. Stephen Levitt, for his invaluable guidance, support, and leadership throughout my research journey. His expertise and mentorship have been pivotal to the successful completion of my work.

I would like to thank my parents, Prof. Joseph Kirui and Mrs. Pauline Kirui, for their support and encouragement throughout my journey in the field of software engineering. Their belief in my abilities and constant love were crucial to reaching my goals. Additionally, I extend my heartfelt gratitude to Winnie, Caleb, and Basetsana for their support and companionship. Their presence has been a source of comfort and motivation, for which I am deeply appreciative.

Finally, my sincere appreciation goes to Dariel for the financial support and enabling the pursuit and completion of this research.

Contents

Declaration i

Abstract ii

Acknowledgements iii

Contents iv

List of Figures ix

List of Tables xi

List of Symbols xiii

Nomenclature xv

1 Introduction 1

1.1 Research Area and Research Questions 2

1.2 Research Overview 3

2 Literature Review 5

2.1 Traceability Links 5

2.1.1 Strict Naming Conventions 7

2.1.2 Lenient Naming Conventions 7
2.1.3 Filename Similarity 7
2.1.4 Last Call Before Assert 8
2.1.5 Lexical Analysis 8
2.1.6 Term Frequency–Inverse Document Frequency (TFIDF) . . . 9
2.2 Test-Driven Development 9
2.2.1 Surveys 9
2.2.2 Code Coverage 10
2.2.3 Finite-State Machines 11
2.2.4 Co-evolution of Tests and Production Code 12
2.3 Conclusion 13

3 Research Context and Objectives **14**

3.1 Problem Statement 14
3.1.1 Traceability Links 15
3.1.2 TDD Adherence 15
3.1.3 Development Step Size 18
3.2 Motivation 19
3.3 Scope 20
3.4 Conclusion 20

4 Matching Tests to Production Files **22**

4.1 Filename-Based Matching Algorithm 22
4.1.1 Distinguishing between Tests and Production Files 22

4.1.2 Method of Operation 23

4.2 Statement-Based Matching Algorithm 25

4.2.1 Distinguishing between Tests and Production Files 25

4.2.2 Block 2: Extract Assert Statements 26

4.2.3 Block 3: Split Fluent Assertion Methods in Assert Statements 27

4.2.4 Block 4: Identify Arguments that are Directly Tested 28

4.2.5 Block 5: Locate Directly and Indirectly Tested Objects . . . 28

4.2.6 Block 6: Extract Production Class Names from Declaration Statements 30

4.2.7 A Comprehensive Example using Apache Commons-Math . . 30

4.3 Performance Analysis 32

4.3.1 Performance Metrics 33

4.3.2 Program Architecture 34

4.4 Bootstrap Confidence Interval 41

4.5 Conclusion 41

5 Test-With Development Adherence Metric 43

5.1 Introduction 43

5.2 Absolute TWD Adherence 44

5.2.1 Mapping Tests to Production Files 45

5.2.2 Collection of Data from Commits 47

5.2.3 Time Deltas 50

5.2.4 Test Code Committed Prior to Production Code 52

5.2.5 Method Coverage 54
5.2.6 Evaluating Absolute TWD Adherence 56
5.3 Relative TWD Adherence 65
5.3.1 K-means Clustering 65
5.3.2 Evaluating Relative TWD Adherence 66
5.4 Development Step-Size 70
5.4.1 Production Method Churn 70
5.4.2 The Relationship Between Production Method Churn and TWD Adherence 72
5.5 Conclusion 74

6 Results **75**

6.1 File Matching Algorithms 75
6.2 TWD Adherence 79
6.2.1 Ranking Projects by TWD Adherence 80
6.3 Development Step Size 83
6.3.1 Validation of the Method Coverage System 84
6.3.2 Validation of the TWD Adherence Evaluation System 86
6.4 Processing Time 87
6.5 Threats to Validity 89
6.5.1 Internal Validity 89
6.5.2 External Validity 90
6.5.3 Construct Validity 90

6.6 Conclusion 91

7 Conclusion **93**

7.1 Datasets Used 94

7.2 Future Work 95

References **96**

A Technical Aspects of Absolute TWD Adherence System **104**

B Comprehensive List of Projects Analysed **106**

C SAICSIT 2023 Conference Paper **115**

List of Figures

2.1 A flow diagram of the basic steps of TDD. 11

4.1 Determining the similarity between “annotationsalidator” and “annotationvalidatorfactory”. 23

4.2 Flow diagram of the filename-based matching algorithm. 24

4.3 Flow diagram of the statement-based matching algorithm. 26

4.4 A directed acyclic graph. 30

4.5 Architecture diagram of the performance assessing system. 34

4.6 The output of the logger via the command-line. 40

5.1 The core components of the TWD adherence evaluation system. . . . 45

5.2 The mapping technique between tests and production files. 45

5.3 A UML diagram of the commit and file data structure. 48

5.4 A representation of the time deltas of the Shapes project. 50

5.5 Calculating time deltas between all modified tests and each method they test. 52

5.6 Getting the time deltas between a single test file and each method it tests 52

5.7 The distribution of the percentage of test-first pairs in 100 Java projects. 54

5.8 The distribution of median time deltas in 100 Java projects 58

5.9 A representation of the time deltas and method churn of the Shapes project. 71

5.10 Calculating the churn for all the tested production methods of a project. 73

5.11 Getting the churn for a single production file's tested methods. . . . 73

6.1 F1-scores for the statement-based and filename-based matching algorithms given n test samples from different Java projects. 76

6.2 Difference between the upper bound and lower bound for the 95% confidence interval of the weighted F1-score for the matching algorithms 78

6.3 The elbow plot of distortion vs no. of clusters for the given data. . . 80

6.4 The scatter plot of relative TWD adherence for 100 Java projects. . 81

6.5 A CDF plot of the time deltas between assertions and production methods in SIG-Fasten. 81

6.6 A CDF plot of the time deltas between assertions and production methods in ArgoUML. 82

List of Tables

4.1 TP, FP and FN values 38

4.2 Precision, Recall and F1-score values 39

5.1 Output of the time delta subsystem 51

5.2 The legend for the symbols used in Figure 5.5 and Figure 5.6. 51

5.3 Example of obtaining relative TWD adherence. 67

5.4 Calculation of the within-cluster TWD adherence 68

5.5 Calculating the relative TWD adherence of projects. 69

5.6 Comparison with other normalisation techniques. 69

5.7 Output of the time delta subsystem 72

5.8 The legend for the symbols used in Figure 5.10 and Figure 5.11 . . . 72

6.1 Weighted performance metrics for the matching algorithms 76

6.2 Upper bound and lower bound values for the weighted F1-score for the matching algorithms for 10 test samples and 500 test samples. . 79

6.3 Project Selection Criteria. 79

6.4 Aspects of SIG-Fasten and ArgoUML 81

6.5 The correlation between the median method churn and relative TWD adherence for various ranges of assertions. 83

6.6 The correlation between the actual code coverage from Codecov and the method coverage calculated by the system in this study. 85

6.7 TDD scores from surveys and the relative TWD adherence values from the experiment. 87

6.8 Processing time for ten Java projects 88

6.9 Computer specifications 88

B.1 Comprehensive list of projects analysed in this study (1 - 16) 107

B.2 Comprehensive list of projects analysed in this study (17 - 30) . . . 108

B.3 Comprehensive list of projects analysed in this study (31 - 44) . . . 109

B.4 Comprehensive list of projects analysed in this study (45 - 58) . . . 110

B.5 Comprehensive list of projects analysed in this study (59 - 70) . . . 111

B.6 Comprehensive list of projects analysed in this study (71 - 82) . . . 112

B.7 Comprehensive list of projects analysed in this study (83 - 94) . . . 113

B.8 Comprehensive list of projects analysed in this study (95 - 100) . . . 114

List of Symbols

The principal symbols used in this dissertation are summarised below, and the first equation in which each symbol appears is given.

r	Similarity Ratio, *Equation (4.1)*
M	Sum of Matching Characters, *Equation (4.1)*
T	Sum of String Lengths, *Equation (4.1)*
ρ	Precision, *Equation (4.3)*
λ	Recall, *Equation (4.4)*
ψ	F1-score, *Equation (4.5)*
$\bar{\rho}$	Weighted Precision, *Equation (4.6)*
ω	No. of Test Files, *Equation (4.6)*
$\bar{\lambda}$	Weighted Recall, *Equation (4.7)*
$\bar{\psi}$	Weighted F1-score, *Equation (4.8)*
R_{TF}	Ratio of Test-First Pairs, *Equation (5.1)*
N_{TF}	Number of Test-First Pairs, *Equation (5.1)*
N_{TW}	Number of Test-With Pairs, *Equation (5.1)*
N_{TL}	Number of Test-Last Pairs, *Equation (5.1)*
τ	TWD Adherence of a Project, *Equation (5.10)*
δ	Time Delta, *Equation (5.10)*

ζ	Method Coverage, *Equation (5.10)*
$\vec{\Delta}$	Matrix of Time Deltas, *Equation (5.10)*
$\vec{\Gamma}$	Matrix of Method Coverage, *Equation (5.10)*
τ'	Within-Cluster TWD Adherence, *Equation (5.16)*
τ_p	Maximum TWD Adherence, *Equation (5.16)*
n_p	Project With the Maximum TWD Adherence, *Equation (5.16)*
τ_r	Relative TWD Adherence, *Equation (5.17)*

Nomenclature

TDD Test-Driven Development

TWD Test-With Development

TFIDF Term Frequency–Inverse Document Frequency

ITLD Iterative Test–Last Development

TLD Test–Last Development

FSM Finite State Machine

VCS Version Control System

TP True Positive

FP False Positive

FN False Negative

WCSS Within-Cluster Sum of Squares

Chapter 1

Introduction

Test-Driven Development (TDD) is a software development technique that was introduced by Kent Beck [1]. TDD requires a developer to write tests for a feature before any production code is developed. Since no production code exists at the point when tests are first written, the test must fail. Thereafter, just enough production code that makes the test pass must be written. Lastly, the code must be refactored; this involves modifying the structure of the code while ensuring that tests still pass. This forms a single TDD cycle. A software project developed using TDD goes through several of these cycles in an iterative manner.

TDD promises several internal and external benefits, including greater maintainability and a lower defect rate, most of which are centred around software quality [2]. Software quality is an important aspect of software engineering, ranking as the second sub-field with the highest number of articles published [3]. For this reason, numerous studies have been conducted to verify the impact of TDD on software quality in order to verify these claims. Many of these studies are empirical, which means that they rely on real-world data, observations, and/or experiments (as opposed to speculative, anecdotal, or theoretical evidence) [4]. However, research findings into the impact of TDD on software quality have been contradictory or inconclusive [5].

In order to conduct empirical TDD research, it is essential to identify which projects have been developed using TDD. This sometimes involves matching production files with their respective test files, a process more formally known as establishing test-to-code traceability links. Identifying TDD projects also involves capturing the order in which test code and production code were written in a project. Because of the uncertainty in the exact order in which test code and production code were written upon analysing a Git project, a variant of TDD known as test-with development

(TWD) is also considered in research. TWD is defined as the co-evolution of tests and production code modifications that are independent of the order in which tests and production code were written. Since TDD emphasises writing just enough production code to make a test pass, it is important to understand whether TDD is associated with smaller commits. However, few studies have conducted research into the relationship between TDD and the median development step size in a software project.

There are several reasons for the contradictory or inconclusive nature of TDD research findings, but the common shortcoming among studies is the use of a simple filename-based technique to establish traceability links between test and production files, as it fails to accurately establish links between test files and the production code that is tested [6]. Another shortcoming is the poor framework that is used to classify the extent to which TDD was adhered to during the development of a software project, which includes surveys and test coverage.

A static statement-based technique is proposed, developed, and analysed in order to establish traceability links between test files and production files in a more accurate manner, and a method to determine the TWD adherence of a project is presented to overcome the previously mentioned shortcomings.

Using the TWD adherence metric, the effect of TWD on a project's development step size is investigated, as there has been little research conducted on it.

1.1 Research Area and Research Questions

This study examines the performance of techniques that are used to establish traceability links between test files and production files and introduces a novel technique that performs better than the commonly used filename-based technique. In addition, this study investigates the methods used to determine the TDD adherence of a software project and introduces a novel mathematical function that can be used in conjunction with existing methods to determine the TWD adherence of software projects quantitatively. Finally, the median development step size in software projects is investigated to determine whether projects that have high TWD adherence are associated with smaller commits.

The research questions are as follows:

RQ1: Can test code be matched more accurately to production code using a statement-level approach rather than the commonly-used filename-based technique?

RQ2: How can such a matching algorithm be used to measure the TDD adherence of a project?

RQ3: Do developers make smaller commits when practising TDD?

1.2 Research Overview

The remainder of the dissertation is structured as follows:

Chapter 2 (Literature Review) presents a review of the relevant literature. Section 2.1 discusses existing techniques that are used to establish traceability links between test files and production files. Section 2.2 discusses common methods used to classify software projects according to their TDD adherence.

Chapter 3 (Research Context and Objectives) presents the problem statement, motivation, scope of work, and research questions for the study.

Chapter 4 (Matching Tests to Production Files) presents the design of the filename-based and statement-based matching algorithms. Emphasis is placed on the sub-systems that make up the statement-based matching algorithm. In addition, a system to measure the performance of both algorithms is presented. Lastly, a system to quantify the confidence in the reported performance of both matching algorithms is shown.

Chapter 5 (Test-With Development Adherence Metric) presents the system to determine the TWD adherence of individual and multiple software projects. Firstly, the mathematical function that is used to determine the TWD adherence of a single project, formally termed "absolute TWD adherence", is presented. Thereafter, the system used to benchmark software projects against each other according to their individual TWD adherence values to produce a comparative value, formally termed "relative TWD adherence", is presented. Finally, a system to determine the median development step size of a software project is presented.

Chapter 6 (Results) presents the results of the study and a critical analysis of the results. In addition, the research questions are answered, and threats to the validity of the study are discussed.

Chapter 7 (Conclusion) presents the conclusion of the dissertation. A summary of the findings is presented, an overview of the purposes of different groups of projects is provided, and ideas for future work are discussed.

Additional supporting material may be found in the appendices of the dissertation. These appendices are presented as follows:

Appendix A presents the technical aspects of the TWD adherence system.

Appendix B provides a comprehensive list of the 100 open-source software projects analyzed in this research. This includes the project name, repository URL, latest commit hash during analysis, and a brief description of each project.

Appendix C contains the conference paper presented at the SAICSIT 2023 conference and published in the proceedings [7], which won the award for the best computer science paper.

Chapter 2

Literature Review

This chapter presents a literature review of studies concerned with test-to-code traceability links and TDD. Static, dynamic, and composite traceability tools are outlined, and their advantages and drawbacks are discussed. Popular methods that are used in studies to determine the TDD adherence of projects are also discussed, which include surveys, code coverage, finite-state machines, and the co-evolution of test code and production code.

2.1 Traceability Links

Traceability links are a representation of the relationships between software artefacts [8]. In this study, tools that establish *test-to-code* traceability links, which are a representation of the set of production files that are covered by a given test file, are investigated. For instance, such a tool could pair a test class called `AccountTest.java` with a production class called `Account.java`. Traceability links are important in various aspects of the software-development life-cycle, including project management and fault location [9]. They are also important in the research methodology of software engineering studies, especially in TDD studies, since such studies rely on matching test files to their corresponding production files [10].

There are several techniques that have been developed to establish test-to-code traceability links in a software project. They are subdivided into static, dynamic, and composite techniques. Static test-to-code traceability tools do not require a software project to be compiled in order to establish traceability links since the source code of files, not bytecode, is analysed [11]. In contrast, dynamic test-to-code traceability tools require a software project to be compiled for the establishment of traceability

links by accessing the call stack [11]. Composite test-to-code traceability tools utilise more than one static or dynamic technique to establish traceability links.

Static test-to-code traceability tools have the advantage of ease of use and application [12]: Normally, software projects need to be configured properly in order to run successfully. However, the incorrect or improper setup of various components, settings, or parameters within the software system often occurs. This is known as a misconfiguration [13]. A misconfiguration would affect the build process of a software project and thus hinder the use of a dynamic analysis approach. Moreover, it is challenging to use dynamic code-traceability tools on several software projects (in the case of empirical research), as the configuration would need to be done for each software project. If dynamic tools were used in a study, it would limit the sample size of the projects. For these reasons, static tools are advantageous: they do not require projects to be built (as they analyse code artefacts parsed as text), and they can be applied to a much larger sample size of software projects compared to dynamic code-traceability tools.

The downside of using static-code traceability tools is that they fail to capture several dynamic aspects of programming, including polymorphism, method dispatch, dynamic binding, and reflection [14]. This may limit their *recall*, which refers to the fraction of relevant instances (tested production classes) that have been correctly identified out of the total number of relevant instances (all the tested production classes in the test file) [4]. Moreover, since dynamic test-to-code traceability tools rely on the analysis of bytecode generated by the compiler or on accessing the call stack, they may be more reliable than static tools. Another point in favour of dynamic test-to-code traceability tools is that they analyse multiple layers of the call stack to establish traceability links between production classes and test classes, i.e., function calls that are made when a test is executed are recorded and linked to the production classes in which they are defined [15]. This is unlike static tools, which commonly limit the establishment of traceability links to the public interface of production classes.

Composite traceability tools aim to address the shortcomings of using a single tool by mitigating the negative aspects and capitalising on the positive aspects of each technique [16].

There have been several static and dynamic test-to-code-traceability techniques that have been developed. Popular test-to-code-traceability techniques in the literature are discussed below.

2.1.1 Strict Naming Conventions

JUnit recommends that a test filename be named after the production file that it tests by prepending or appending the string "test" to the filename [17]. For example, given a production file `Engine.java`, the corresponding test file could be named `TestEngine.java` or `EngineTest.java`. A technique that depends on strict naming conventions takes advantage of this pattern, as it establishes a link between a test file and a production file if they differ exactly by the case-insenstive string "test". Such a technique offers very high precision; it is very likely that a production file is indeed tested by a test file if a link is established between them. In addition, such a technique is computationally inexpensive; establishing links in a project using this technique would be completed relatively quickly compared to other techniques. However, the downside is that it offers very low recall; it establishes links for unit tests only [6] and thus fails to capture links for test files that cover multiple production files, such as integration tests. TDD studies that rely on this matching technique include [18, 19, 20].

2.1.2 Lenient Naming Conventions

To overcome the low recall of the technique based on strict naming conventions, the lenient naming convention technique relaxes the matching condition to match a test file to a production file. A match is established if the test filename contains the production filename in it. Recall is improved as more relevant results are retrieved. For instance, consider a test file and a production file called `MultipleScreensTest.java` and `Screen.java`. The two files would be matched since "screen" is a substring of the string "multiplescreens". However, the improved recall comes at the cost of lower precision: consider a production file called `Car.java` and a test file called `CarpetTest.java`. From the naming of the files, it is unlikely that the two should be associated at all, but the lenient naming convention technique would establish a link between these two files since "carpet" contains the string "car". Studies that rely on this matching technique include [21].

2.1.3 Filename Similarity

Unlike techniques based on naming conventions that are boolean (a production filename that is analysed by a boolean technique is either a part of a test filename or it is not), techniques based on filename similarity return a ratio that represents

how similar a test filename is to a production filename. Such techniques include the Ratcliff-Obershelp algorithm [22], which matches two strings by considering the longest common substring between them and the length of the shorter string, and the Levenshtein distance, which matches two strings by analysing the minimum number of editions it would take to transform one string to another [23]. Such techniques return a number between 0 and 1 that represents the similarity. Hence, to match strings using these techniques, a cut-off value is required. For example, one can specify a cut-off value above which a match is established, e.g., 80%. The higher the cut-off value, the lower the leniency, resulting in higher precision and lower recall. The lower the cut-off value, the greater the leniency, resulting in lower precision and higher recall. This property is favourable as the matching can be altered if unsatisfactory matching results are obtained for a given project. TDD studies that use this matching technique include [24].

2.1.4 Last Call Before Assert

Unlike the previously discussed techniques that analyse only filenames to establish traceability links, the last-call-before-assert technique analyses the source code of a test file parsed as text. It relies on the assumption that the last function call before an assertion corresponds to the production class that is being tested. The downside of such a technique is that it is not guaranteed (and may rarely be the case) that the object or function involved in the last call before the assert statement is from a production class. The object or function may simply be associated with a third-party class. The upside is that it accurately identifies production classes that are tested if objects are created from the intended production class immediately before the assert statement, since said object will be passed as an argument in the assert statement (and thus be explicitly tested). Studies that use this technique include [6, 25, 26, 27].

2.1.5 Lexical Analysis

Lexical analysis is a technique used to analyse the vocabulary used in production code and test code to establish traceability links between them. It extracts natural language and assigns a high likelihood of association between a test file and a production file if they share several keywords in common. Since lexical analysis considers the actual words that are used in a test file, it can be more accurate than using a filename-based approach because it reduces the number of false positives in the matches. However, since it cannot infer the context in which a word is used in a

test file, it can also incorrectly establish a match. Studies that use this technique include [6, 28, 29].

2.1.6 Term Frequency–Inverse Document Frequency (TFIDF)

TFIDF is a dynamic technique that operates on the assumption that if an object of a production class is instantiated more than other objects in a test file, then it is likely that the test file tests functionality from that production class, thereby establishing a link. It is calculated by multiplying the term frequency of a string by the inverse document frequency of a string. The term frequency of a string is the number of times the string appears in a document. The inverse document frequency of a string is the logarithm of the number of files that contain the string. Since TFDIF is dynamic, it may perform better than the filename-based technique in scenarios where tested production classes cannot be inferred from filenames. However, TFDIF requires the execution of code, which limits its use on large numbers of files. Studies that use this technique include [26, 30, 31, 32].

2.2 Test-Driven Development

In his book on Extreme Programming [1], Kent Beck introduced TDD, which is a software development practice in which test code is written prior to production code in an incremental and iterative manner with the sole purpose of delivering quality code. TDD is characterised by writing tests before production code, short iterations, and frequent refactoring. Software engineering researchers have conducted several studies to investigate the positive claims put forward by TDD proponents. Such claims include the improvement of refactoring, superior code quality, 100% code coverage, the generation of documentation, and the reduction of defect rates [33]. The methodologies used by researchers to quantify the TDD adherence of projects include the following: surveys, code coverage, finite-state machines, and the co-evolution of test code and production code.

2.2.1 Surveys

In some controlled studies, surveys are used to assess the extent to which TDD practices were followed during the development of a software product.

It is common practice for software development teams to define a methodology through which to produce software artefacts [34]. Developers adhere to the defined methodology throughout the lifecycle of the software project; hence, they have first-hand experience with the details of the methodology. That is, if a third party attempted to define the details of the methodology by solely analysing the software artefacts, it would be difficult to do so without the insights of those within the project. Surveys take advantage of the first-hand account of developers by preparing a list of questions pertaining to the processes used to develop software. The list is presented to developers, who then (ideally) respond with a detailed and honest account of the processes used when developing software. In the context of TDD research, such surveys include questions related to whether using TDD is enforced during software development.

The advantage of using surveys to evaluate the use of TDD in a software project is that it is relatively simple to evaluate whether or not a particular project was developed using TDD; survey participants can easily recall their experience in a project to determine whether they used TDD. This simplifies the data analysis phase of TDD research since complex analysis is not required to determine whether TDD was used in the development of a project. The disadvantage is that surveys are prone to bias and projects cannot be compared objectively; developers may exaggerate the extent to which they used TDD, especially if they know that using TDD is considered a good practice compared to iterative test-last development (ITLD) [35]. In addition, since developers can only account for their experience in a particular project, the results of the survey cannot be used to compare different projects with one another accurately due to variables between projects that may not be accounted for. TDD studies that use surveys include [36, 37, 38, 39].

2.2.2 Code Coverage

In some studies, the code coverage of a software project is used as a proxy for the extent to which TDD was used to develop the software project.

A core tenet of TDD is that no single line of source code should be written before its corresponding test code is written. This implies that the code coverage of a software project that was developed using TDD should, *ideally*, be 100%. However, in practice, it is challenging to achieve 100% code coverage. Therefore, in some studies, the approach is to use high code coverage as an indication of the use of TDD, while low code coverage is used as an indication of the lack of use of TDD.

The advantage of using code coverage as a proxy for the extent to which TDD was used in a project is that it is a simple and objective metric to obtain for a project. Popular tools like Codecov [40] and Coveralls [41] readily publish the code coverage information of several popular open-source projects, and such information can be obtained via an API with ease. The code coverage values of multiple projects can be compared against one another. The disadvantage is that high code coverage does not necessarily imply that TDD was employed, even though TDD often leads to high code coverage. While it is true that the strict application of TDD in a software project results in high code coverage, it is not always true that a project that is found to have high code coverage was necessarily developed using TDD; it could have been developed using ITLD. For this reason, code coverage is usually used in conjunction with other analysis techniques to determine the extent to which TDD was used in a project. TDD studies that include code coverage in the analysis of the TDD adherence of projects include [24, 18].

2.2.3 Finite-State Machines

Some TDD studies make use of tools that are implementations of finite-state machines to determine the extent to which TDD was used in a software project. A finite-state machine (FSM) is a computational model that consists of a limited set of states, transitions between those states, and actions and is used to model sequential logic, control flows, or protocols [42]. Consider Figure 2.1 below, which shows the main steps associated with TDD as described by Beck [1]: It is clear that TDD consists of a finite set of states with transitions between those states. For this reason, a FSM can be used to model the application of TDD in a project that uses a version control system (VCS).

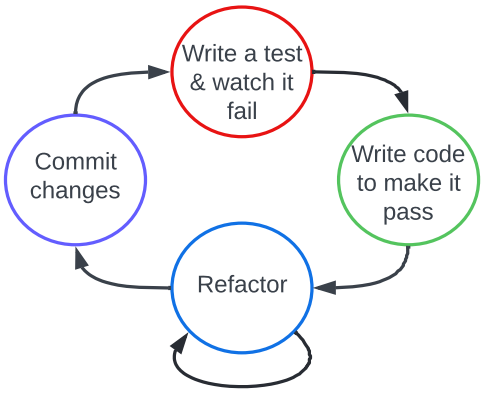


Figure 2.1: A flow diagram of the basic steps of TDD.

The advantage of using a FSM to determine the extent to which TDD was used is that it is a rigorous and accurate method; it can be used to determine whether a single cycle was completed using TDD since states and transitions are clearly defined. One can then determine the proportion of cycles that constitute TDD cycles in order to determine the extent to which TDD was used in a project. The disadvantage of using FSMs is that they do not take into account the temporal aspect of TDD, only the transition between states. As a result, a cycle is considered a TDD cycle if it only follows the red-green-refactor transition, even if a significant amount of time passes between individual states. TDD studies that use FSMs include [43, 44, 45].

2.2.4 Co-evolution of Tests and Production Code

Analysing the co-evolution of tests and production code is another strategy used to determine the TDD adherence of a software project. It is concerned with analysing commit behaviour, code artefacts, and the relationships between them.

In Test-Driven Development (TDD), the process begins by writing a failing test, followed by developing sufficient production code to pass the test. After ensuring that both the test and production code work correctly, the next step is to refactor both. The cycle concludes with committing the updated code. As discussed earlier, FSMs rely on the distinct states and the transitions between them in order to detect a single cycle of TDD. However, in practice, especially upon analysing code artefacts from a VCS, the distinct states and transitions between TDD may not be apparent. This is due to the fact that test code and production code can be committed in an order that does not correspond to the order in which they were actually written. The co-evolution of test code and production code analyses how frequently test code and production code are added or modified together while taking into account the temporal relationship between them. This allows researchers to approximate whether TDD was used when writing a particular set of test and production code, since shorter commit time differences between test code and production code are associated with TDD.

The advantage of analysing the co-evolution between test code and production code to represent the extent to which TDD was applied in a project is that it is useful when code artefacts are analysed retroactively; several TDD studies analyse open-source projects. If one does not use surveys or tools that track the states of TDD (like some FSMs do), then data pertaining to the use of TDD is impossible to obtain. Hence, analysing the co-evolution of test code and production code is a practical way to

approximate the use of TDD in a project. The disadvantage is that it is a relaxed definition of TDD, and therefore it does not accurately capture the use of TDD in projects as depicted in Figure 2.1. TDD studies that are based on analysing the co-evolution of test code and production code include [18, 24, 46].

2.3 Conclusion

A review of the literature on traceability links and TDD has been presented. Test-to-code traceability tools are subdivided into static, dynamic, and composite tools. Static tools can be applied to a much larger sample size of projects than dynamic tools, while dynamic tools offer high performance. Composite tools combine static or dynamic tools to improve performance. Analysis of naming conventions, filename similarity, source code analysis, and lexical analysis are common static traceability techniques, while term frequency-inverse document frequency is a common dynamic traceability technique. Several TDD studies seek to investigate the effect of TDD on code quality, and common techniques used to determine the extent to which TDD was applied by a participant or group include surveys, code coverage, FSMs, and the co-evolution of test code and production code.

Chapter 3

Research Context and Objectives

This chapter presents the problem statement, aim, scope, and research questions of this study. Popular static test-to-code traceability tools that appear in existing studies suffer from low recall due to their reliance on filename-based approaches. Furthermore, current techniques for determining TDD adherence are not reliable, often relying on subjective surveys. Lastly, TDD studies have not sufficiently investigated the association between TDD adherence and the median development step size. This study addresses these issues by analysing Java projects on GitHub, introducing a more accurate test-to-code traceability tool, and exploring the correlation between TDD adherence and development step size.

3.1 Problem Statement

In software engineering research, accurately assessing the adherence to Test-Driven Development (TDD) and establishing effective traceability links between test and production files are critical for deriving valid conclusions. This chapter explores the existing methodologies used to establish these traceability links and assess TDD adherence, highlighting the limitations and proposing the need for more robust approaches. By examining the challenges in these areas, this section aims to provide a comprehensive understanding of the current state of research and identify gaps that need to be addressed for improving the accuracy and reliability of empirical studies in software engineering. Additionally, understanding the relationship between TDD adherence and development step size can shed light on how TDD influences software quality, offering further insights into the impact of this development practice.

3.1.1 Traceability Links

Zaidman et al. [18] used the strict filename-based approach to match a single test file to a single production file. Borle et al. [24] matched a test file to a production file using the filename similarity approach (implemented using Python's `difflib.SequenceMatcher` [47]) with a threshold of 80%. Hanhela [19] matched one test file to only one production using the strict filename-based approach, i.e., the test file's name starts with the name of the production file and ends with the word "Test". Wang et al. [20] paired a production class with its corresponding test class using the strict filename-based approach by checking if the test filename differed from the production filename only by the string "Test" in the test filename. Van Rompaey et al. [6] reported that linking a test case to its corresponding production file(s) using a filename-based approach can yield high precision and recall, depending on project guidelines, whereas White et al. [26] reported that the filename-based approaches yielded low recall.

The majority of studies that rely on establishing traceability links between test and production files use a filename-based approach [18, 24, 19, 21, 20, 6, 26]. As a result, many test files are left unpaired with the production files that they actually test, as in the case of unit tests. In the case of integration tests, not all of the relevant links between a test file and all of the production files that are actually covered are established. Hence, the commonly used matching approach suffers from low recall, which ultimately affects the validity of the findings of such studies. In addition, studies that involve dynamic approaches (which may be more accurate than the traditional filename-based approach) are limited in terms of the number of projects that can be analysed [6, 26], which affects the generalisation of such studies. This leads to the first research question: "Can test code be matched more accurately to production code using a statement-level approach rather than the commonly used filename-based technique?"

3.1.2 TDD Adherence

Karac et al. [36] requested the students that participated in their controlled study to rate on a scale from 0 to 100 how closely they adhered to TDD practices during the experiment. In a study conducted by Buffardi et al. [37], one of the questions in the survey requested that the participants rate on a five-point scale how consistent they were in applying TDD. Hu et al. [38] requested participants in the TDD group that formed part of their study to describe the extent to which they followed TDD.

Upon analysing the responses to the survey, the authors reported that only 54% of the participants in the TDD group claimed to follow TDD strictly. Furthermore, the authors admitted that they did not provide the same survey question to the non-TDD group to find out how closely they followed TDD and only assumed that the group did not follow TDD based on the development steps of a single participant. Sanchez et al. [39] found that of the eleven participants that responded to their survey, three wrote tests before production code, seven wrote tests together with production code, and one wrote tests after half of the production code was finalized. The authors note that the participants did not follow TDD strictly, which brings into question the validity of the findings of their study since the positive results may also be attributed to ITLD.

The use of surveys to determine the TDD adherence of projects is susceptible to positive response bias. Positive response bias is the tendency of a survey participant to exaggerate a behaviour if it is considered socially desirable [35]. In the context of the surveys used in TDD studies, positive response bias would be apparent when participants reported complying with TDD more closely than they actually did. For this reason, the over-reliance on surveys to determine the TDD adherence of a software project can result in findings that lack validity. Hence, there is a need for an objective approach to assess the TDD adherence of software projects. An objective method can be used in conjunction with surveys to increase the validity of the conclusions drawn in TDD studies that involve the analysis of projects.

Borle et al. [24] used class coverage as a proxy for the TDD adherence of a software project. They also used the maximum time difference between the commit timestamp between a test file and the production file it covers for all files in a project to determine the TDD adherence of a project. A project wherein greater than 90% of the production classes are covered and wherein all pairs of test files and production files were included in the same commit was considered to adhere to TDD fully. However, it is possible to achieve a class coverage greater than 90% using ITLD as well, which nullifies the use of code coverage alone as a proxy for TDD adherence [48]. In addition, finding the maximum difference in the commit timestamps between tests and production files in a project to assess TDD adherence assumes that projects developed using TDD will have every single test-and-production file pair existing in the same commit. Moreover, reducing the difference between all the commit timestamps between tests and production files to a single number using a simple maximum function in order to represent TDD adherence is not representative of TDD since TDD involves a series of incremental and iterative changes to test code and production code, which implies a set of time differences instead of only one value [1].

Hilton et al. [44] used a FSM to assess the TDD adherence of a project by referring to the red-green-refactor steps that form part of TDD. While FSMs are effective in recognising the different phases of TDD, they do not assess the frequency and consistency of commits because they focus only on the sequence of key events that are associated with TDD.

Fucci et al. [49] posited that a higher percentage of test-first cycles is associated with greater TDD adherence. However, in the context of analysing software projects after the fact, the exact sequence of writing test code in relation to production code may not be straightforward; test code could be written before production code and then committed together with it. A more inclusive approach would allow for variation in the sequence between test code and production code based on the difference in commit timestamps since the true sequence cannot be inferred with confidence.

Zaidman et al. [18] stated that TDD can be observed as the simultaneous committing of test code and production code in the same commit. However, this rules out changes between tests and production code with a non-zero time difference between their respective commits, which means that such a method does not capture the frequency and consistency of commits, which is a crucial aspect to consider.

Lubsen et al. [46] defined association rules for tests and production files within the same commit to assess TDD adherence. Such an approach disregards potential associations between tests and production files that are in different commits, which limits the accuracy of the approach: all associations between tests and production files are defined in the same commit, and thus the difference in commit timestamps is always zero. Since TDD is time-dependent, this approach may not accurately assess the TDD adherence of a software project.

Hence, there is a need for a quantitative approach that can be applied to code artefacts that, unlike using code coverage alone, effectively tells apart a project that was developed using TDD from a project that was developed using ITLD. There is a need for a quantitative approach that assesses the commit behaviour apparent in a software project in order to determine its TDD adherence in a more representative manner. In addition, there is a need to include an analysis of the timing of commits (i.e., how frequently and consistently commits are made) during the assessment of the TDD adherence of projects. Lastly, there is a need for a quantitative approach that takes into account the variability in the true sequence between test code and production code when assessing the TDD adherence of a software project after the fact. This leads to the second research question: “How can such a matching algorithm

be used to measure the TDD adherence of a project?"

3.1.3 Development Step Size

Janzen et al. [50] investigated the impact of test-first development on code size, complexity, coupling, and cohesion. One case study and five quasi-controlled experiments were conducted, with a total of twelve participants and fifteen participants in the test-first group and the test-last group, respectively. They found that test-first developers wrote smaller methods on average than their test-last counterparts. They also found that in most of the studies, test-first developers wrote fewer methods per class than their test-last counterparts. The authors stated that this suggests that test-first developers write smaller, simpler classes and methods than test-last developers.

George [51] conducted TDD experiments with pair programmers to determine its effect on code quality. Student and professional developers were placed into a TDD group and a non-TDD group that took a waterfall software development approach. Among other findings, the author found that the TDD group wrote fewer lines of code and fewer operations.

Madeyski et al. [52] investigated the impact of TDD on productivity. As part of their methodology, they requested a single experienced participant to design a web-based system. They measured productivity in three ways: the implementation time per user story, the lines of code written per user story, and the lines of code written per hour. They found that the participant's productivity was higher using TDD compared to TLD if the development process was divided into two distinct phases. Vu et al. [53] conducted an experiment with fourteen undergraduate students in a software engineering capstone project that spanned a year to determine the effect of TDD on productivity, internal and external quality, and the programmers' perception of the methodology. The authors found that the TLD group had higher productivity than the TDD group, since the TLD team took 14.09 hours on average to implement a feature, compared to 23.38 hours per feature and 22.30 hours per feature on average for the two TDD teams. Panvcur et al. [54] conducted an experiment with undergraduate students over a period of two years to determine the effect of TDD on productivity, external quality, complexity, code coverage, and fault-finding capabilities. They measured productivity as the number of successfully implemented user stories per hour. They found that the effect of TDD on productivity is small and positive, but not statistically significant.

Although many studies have examined the impact of TDD on the final product size and participant productivity, there is a lack of research on its effect on the median development step size throughout the development process. TDD emphasises writing tests and production code in an iterative manner and in rapid succession; this involves writing the bare minimum code to make a test pass, after which the production code is refactored and changes are committed [1]. One would expect that a project that was developed with TDD would have a smaller median development step size compared to one that was developed with TLD if other confounding variables were controlled. Hence, it is a valuable addition to the body of knowledge of TDD research to investigate the correlation between the TDD adherence of projects and the median value of their respective development step sizes. This leads to the third research question: "Do developers make smaller commits when practising TDD?"

3.2 Motivation

Several TDD researchers take it for granted that the filename-based approach is accurate as a method to establish traceability links between test files and production files. This assumption may add to the inconclusive and often contradictory nature of the findings made by TDD researchers. The accurate establishment of traceability links between test files and production files is important for the validity of conclusions drawn by TDD researchers. This study aims to introduce a static test-to-code traceability tool that is more accurate than the commonly used filename-based matching algorithm. By introducing a more accurate approach for establishing test-to-code traceability links, TDD research will benefit due to more reliable matching results. This will help to more reliably establish the efficacy of TDD in relation to code quality.

Many TDD studies rely on surveys to determine the TDD adherence of a software project, while others rely on techniques that analyse code artefacts. Surveys are notorious for their low response rate and for being unreliable in certain cases, while techniques based on analysing code artefacts may not do so accurately. In addition, most existing techniques do not reveal the extent to which TDD was applied in a software project; they are binary in output. This results in the miscategorization of projects. This study aims to introduce a new mathematical function that analyses the differences in commit timestamps and the method coverage between test code and production code to produce a numerical value that represents a project's TDD adherence. TDD research will benefit from this, as projects can be compared in large

numbers to analyse the effect of TDD on factors such as code quality, among others.

Most TDD studies that involve an investigation into the relationship between TDD and an aspect of a software product related to size (such as productivity) focus more on the analysis of the final product and less on the analysis of trends associated with the incremental changes of the project throughout its development, such as the median development step size in the project. The aim of this study is to conduct an investigation to determine whether the application of TDD is associated with smaller development steps by applying the new TWD adherence metric mentioned earlier. TDD research will benefit from an investigation into the relationship between the TDD adherence of a project and the median development step size in the project because it will provide insights into the commit behaviour of TDD practitioners, thereby confirming or refuting the claim that TDD is necessarily associated with granular, incremental changes to production code.

3.3 Scope

This study introduces a static test-to-code traceability tool. The test frameworks considered in this study are JUnit [55] and TestNG [56], and the projects analysed in this study are open-source software projects that are hosted on GitHub. The full list of projects analysed in this study can be found in Appendix B. The study introduces a mathematical function that maps the commit timestamps and method coverage between assertions and the production methods they test to a single numerical value that represents a quantity related to the TDD adherence of said project. The mathematical function is used to develop a system to benchmark Java software projects against each other according to the quantity related to TDD adherence. Lastly, an investigation is conducted to determine whether projects that display greater TDD adherence are associated with smaller development steps.

3.4 Conclusion

The problem statement, aim, scope, research questions and objectives of this study have been presented. Most existing test-to-code traceability tools used in TDD studies suffer from low recall, techniques used to determine the TDD adherence of projects are not reliable, and TDD studies have not investigated the association between the TDD adherence of a project and the median development step size. This

study aims to address these concerns by introducing a novel test-to-code traceability tool, defining a new mathematical function that calculates a quantity related to TDD adherence, and determining the association between the median value of a project's development step size and the extent to which TDD was used in its development. The study analyses open-source Java projects on GitHub.

Chapter 4

Matching Tests to Production Files

This chapter presents the design and implementation of the filename-based and statement-based matching algorithms. The filename-based matching algorithm is implemented using the Ratcliff-Obsershelp algorithm, which operates on filenames, while the statement-based matching algorithm comprises six stages that apply unique operations to test code. A system to assess the performance of the matching algorithms is introduced; it calculates the bootstrap confidence interval of the weighted F1-score of both of the matching algorithms.

4.1 Filename-Based Matching Algorithm

The filename-based matching algorithm that is implemented in this section is based on the Ratcliff-Obsershelp algorithm [22] as discussed in Subsection 2.1.3. This approach is chosen because it offers a better recall and F1-score compared to the other filename-based matching algorithms that are based on naming conventions. This will ensure a fair comparison between the performance of the filename-based and the statement-based matching algorithms.

4.1.1 Distinguishing between Tests and Production Files

Before establishing matches, two sub-sets are created from the set of all files in a project, namely the set of test files and the set of production files. It accomplishes this by analysing whether the string “test” appears in a filename. The match is

case-insensitive and position-insensitive. Listing 4.1 below shows a Python function that achieves this.

```
def isTestFile(filename):
    if "test" in filename.lower():
        return True
    return False
```

Listing 4.1: A function that checks whether a file is a test file.

4.1.2 Method of Operation

The method takes two arguments as input, namely, the name of the file that is assumed to be a test file and a list of all production filenames in the project. The output is a set of the names of tested production files. A match is established between a test file and a production file if there is at least 80% similarity in their filenames. The similarity ratio r between two strings A and B is determined by the Ratcliff-Obsershelp algorithm and is given by the following formula:

$$r = 2 \cdot \frac{M}{T} \tag{4.1}$$

Where M is the sum of individual characters in A and B that match in the same order, and T is the sum of the lengths of strings A and B. The function `difflib.SequenceMatcher` from the Python standard library is used to calculate the similarity ratio between two strings. For example, consider a JUnit4 [57] test file called `AnnotationsValidatorTest` and a JUnit4 production file called `AnnotationValidatorFactory`. All characters are converted to lowercase, and the string "test" is removed from the test filename before checking similarity. Figure 4.1 below illustrates how a match can be established between the two files.

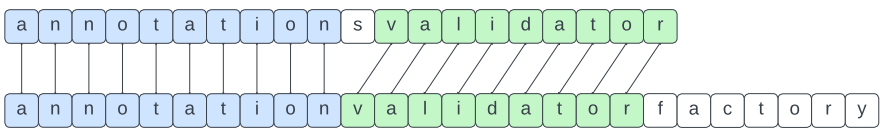


Figure 4.1: Determining the similarity between "annotationsalidator" and "annotationvalidatorfactory".

The sum of the lengths of "annotationsalidator" and "annotationvalidatorfactory" equals 46. Hence, $T = 46$. In addition, there are 19 individual characters between

"annotationsalidator" and "annotationvalidatorfactory" that match in the same order, as shown by the lines connecting the blue and green blocks. Hence, $M = 19$. Therefore, the similarity ratio r is given by:

$$\begin{aligned} r &= 2 \cdot \frac{M}{T} \\ &= 2 \cdot \frac{19}{46} \\ &= 0.82 \end{aligned} \tag{4.2}$$

Since 0.82 is greater than the threshold of 0.8, a match is established between `AnnotationsValidatorTest` and `AnnotationValidatorFactory`.

The process described above represents matching between a single test file and a single production file. Figure 4.2 below illustrates how the process described above is used to establish every possible match between a test file and all the production files in a project.

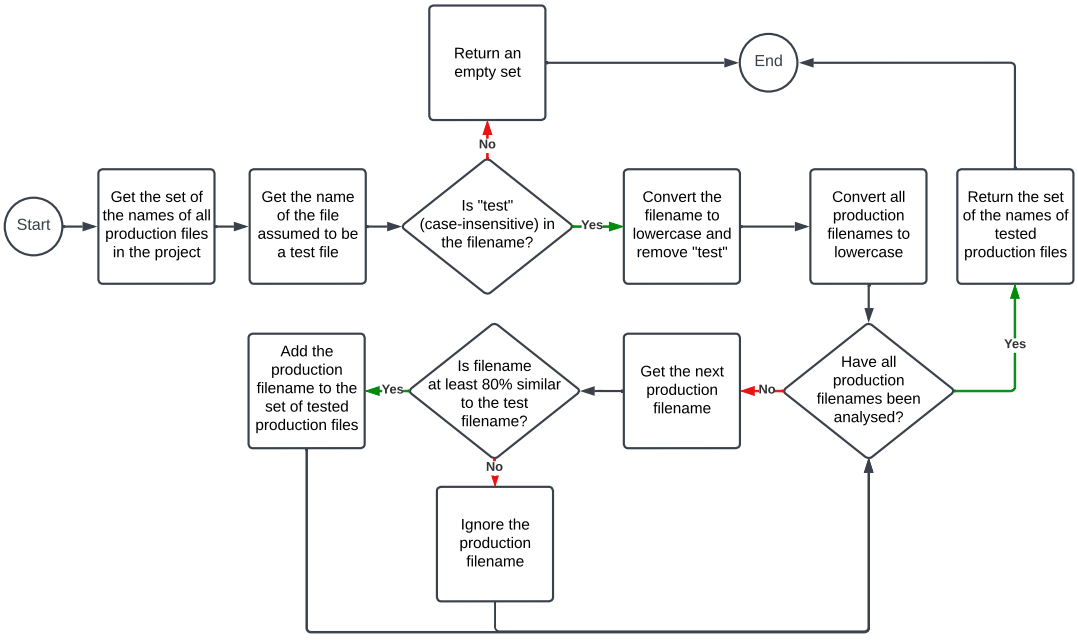


Figure 4.2: Flow diagram of the filename-based matching algorithm.

For example, given a test file `AnnotationsValidatorTest` from JUnit4, the set in Listing 4.2 is retrieved after applying the algorithm represented by Figure 4.2.

```
[
    "AnnotationValidator",
    "AnnotationValidatorFactory",
    "AnnotationsValidator"
]
```

Listing 4.2: Sample output of the filename-based matching algorithm

4.2 Statement-Based Matching Algorithm

Whereas the filename-based matching algorithm matches production files and test files based on their filenames, the statement-based matching algorithm analyses source code line-by-line to match files.

4.2.1 Distinguishing between Tests and Production Files

A complete list of production files and test files in a project is sourced in order to match files. The tool used for this procedure is PyDriller [58]. To differentiate between a test file and a production file, two conditions are evaluated: the first is whether at least one assert statement is included in the source code, and the second is whether at least one import from JUnit or TestNG is made. JUnit and TestNG are testing frameworks that are designed to facilitate writing and executing tests for software applications, particularly in the context of unit testing. Zerouali et al. [59] established that JUnit and TestNG are the most popular testing frameworks. Hence, only JUnit and TestNG are considered in the regular expression associated with the second condition. The regular expressions used to evaluate the two conditions are shown below.

```
assertionRegex = r"assert(Equals|That|ThatThrownBy|Same|NotSame|
    NotNull|Null|True|ArrayEquals|Throws|False|Timeout)\s*\("
```

Listing 4.3: A regular expression that matches an assertion in source code.

```
importRegex = r"import\s*.*;"
```

Listing 4.4: A regular expression that matches imports in source code.

The set of all files that have ever been created in a Java project can be divided into two subsets, namely production files and test files, using the regular expressions in Listing 4.3 and 4.4. A test file is one that has at least one match for the regular

expression in Listing 4.3 and at least one import from JUnit or TestNG from the complete list of imports obtained using the regular expression in Listing 4.4. It is assumed that a JUnit import has the substring "junit" and a TestNG import has the substring "testng". Otherwise, it is a production file. Figure 4.3 below describes the main procedures that make up the statement-based matching algorithm.

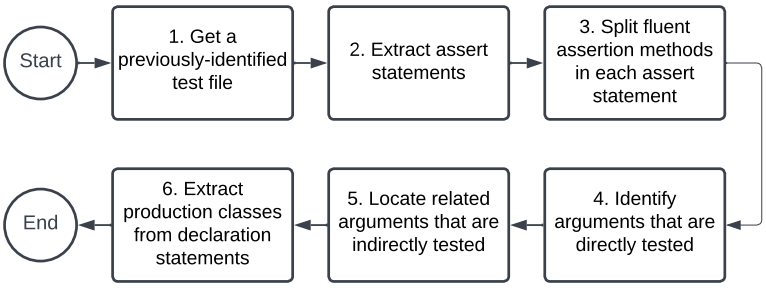


Figure 4.3: Flow diagram of the statement-based matching algorithm.

4.2.2 Block 2: Extract Assert Statements

Block 2 takes test code as input and returns a list of assert statements used in it. Assertion methods usually take object references as arguments, and these objects could be direct instantiations of production classes or instantiations of third-party classes. A third-party class is one that is provided by a third-party library or framework, which is a piece of software that is created and maintained by a separate organization or individual. In some cases, assertion methods take static classes as arguments. Therefore, block 2 isolates parts of the test code that have a clear link to the production classes being tested.

Consider the test code in Listing 4.5, which creates two 2×2 matrices, adds them, and then compares the result to the expected matrix. The `getNorm()` method is used to compute the Frobenius norm of the difference between the `expectedMatrix` and the `result` matrices, and the test asserts that this norm is close to zero within a tolerance of 10^{-9}. Given this test code, block 2 will return the output as shown in Listing 4.6.

```
@Test
public void testAddRealMatrices() {
    double[][] dataA = { {1, 2}, {3, 4} };
    double[][] dataB = { {5, 6}, {7, 8} };
    double[][] expectedData = { {6, 8}, {10, 12} };

    RealMatrix matrixA = MatrixUtils.createRealMatrix(dataA);
    RealMatrix matrixB = MatrixUtils.createRealMatrix(dataB);
    RealMatrix expectedMatrix = MatrixUtils.createRealMatrix(
    expectedData);
    RealMatrix result = matrixA.add(matrixB);

    assertEquals(0, expectedMatrix.subtract(result).getNorm(), 1e-9);
}
```

Listing 4.5: Testing the `RealMatrix` class from Apache Commons-Math.

```
[
   "assertEquals(0, expectedMatrix.subtract(result).getNorm(), 1e-9);"
]
```

Listing 4.6: The output of block 2 given Listing 4.5 as input.

4.2.3 Block 3: Split Fluent Assertion Methods in Assert Statements

A fluent assertion method is a method that facilitates the chaining of multiple assertions together such that the entire assert statement is more human-readable and expressive [60]. Block 3 is responsible for splitting the assertion methods with the goal of isolating the arguments in each method. It takes a list of assert statements obtained from block 2 as input and returns the split fluent assertion methods. Listing 4.7 is an example of an assert statement that uses fluent assertion methods. When it is passed to block 3, the output is given by Listing 4.8.

```
[
   "assertThat(actualList)
       .isNotEmpty()
       .hasSize(5)
       .isEqualTo(expectedList);"
]
```

Listing 4.7: An assert statement that uses fluent assertion methods.

```
[
    "assertThat(actualList)",
    "isNotEmpty()",
    "hasSize(5)",
    "isEqualTo(expectedList)"
]
```

Listing 4.8: The output of block 3 given Listing 4.7 as input.

Notice that the system isolates the chained methods from the assert statement by splitting the assert statement by the dot operator into separate entities.

4.2.4 Block 4: Identify Arguments that are Directly Tested

Block 4 identifies the arguments that are passed as parameters to assertion methods. It accepts the individual assertion methods invoked in an assert statement (the output of block 3) and returns the arguments passed to these methods. Some of the arguments could be instantiated by production classes, while others could simply be instantiated by third-party classes. For example, given Listing 4.8, the output of block 4 is shown in Listing 4.9. Notice that the object names `actualList` and `expectedList` are extracted since they are directly passed to the assert statement as arguments. Since `5` is only a value and does not reference any object, it is discarded.

```
[
    "actualList",
    "expectedList"
]
```

Listing 4.9: The output of block 4 given Listing 4.8 as input.

4.2.5 Block 5: Locate Directly and Indirectly Tested Objects

Block 5 is responsible for identifying *and* locating the positions of references to objects within the code that depend on the objects that are directly tested in assert statements. By taking into account the names of objects passed directly to assert statements (the output of block 4), it conducts a comprehensive analysis of the test code to determine all relevant relationships between objects. To achieve this, block 5 identifies the relationship between different objects in the code, as one object could be the return value of another object.

Block 5 takes two parameters as input and returns a single output. The two parameters are the test file parsed as text and all the object references that are directly tested. The output is a set of production filenames that includes the very same directly tested object references that were passed as a parameter, as well as all the object references they depend on. This is achieved by splicing the declaration statements associated with each object reference and then extracting the object from which it was instantiated, if it exists. Note that for each newly discovered object, there may exist yet another object from which it is instantiated. Hence, this process is executed recursively until all the object references (both directly and indirectly tested) are identified. Hence, the relationship between these objects can be represented by a directed acyclic graph where a node represents an object reference and an edge represents a declaration statement. Block 5 applies the breadth-first search algorithm to the directed acyclic graph [61] to determine all objects that are indirectly tested in a test file.

Consider Listing 4.10 for an example of the action of block 5. Note that Line 7 in Listing 4.10 represents the continuation of the test in the same fashion, i.e., the creation of `objectC`, `objectD`, etc. The relationship between the objects is represented by a directed acyclic graph, as shown in Figure 4.4 below. Notice that although `objectZ` is the only object that is directly tested, all other objects that it depends on for its creation are also returned by block 5. This is because the test would have failed if there were any issues during the creation of the other objects. Hence, the other objects are considered to be tested, albeit indirectly.

```
1  @Test
2  public void someTestFunction() {
3      ProductionClassA objectA = new ProductionClassA();
4
5      ProductionClassB objectB = objectA.someFunctionA(1, someParameterA
       );
6
7      // Test continues ...
8
9      ProductionClassZ objectZ = objectY.someFunctionY(someParameterX);
10
11     assertNotNull(objectZ);
12 }
```

Listing 4.10: A test function where the object being tested is dependent on other objects.

```
[
   "objectA", "objectB", ..., "objectY", "objectZ"
]
```

Listing 4.11: The output of block 5 given Listing 4.10 and “objectZ” as input.



Figure 4.4: A directed acyclic graph.

4.2.6 Block 6: Extract Production Class Names from Declaration Statements

Block 6 is responsible for extracting production class names from declaration statements. Before the extraction, it identifies the lines of code that contain the declaration statements for all the names of objects that are directly or indirectly tested in an assert statement. Block 6 takes as input a collection of tested objects (the output of block 5) and a complete set of production classes in a project. It returns a subset of the production classes that appear in at least one declaration statement. To achieve this, block 6 gets the positions of the directly and indirectly tested objects in the parsed test file from block 5. Then, for each position, a part of the parsed test code is extracted via string slicing. The extracted substring is the declaration statement corresponding to the object whose position was provided. The start of each substring is the last newline character (“\n”) before the given position, and the end of each substring is the first semicolon (“;”) after the given position. Lastly, each substring is analysed to check for the presence of production class names. Note that this process works for both regular classes and static classes.

4.2.7 A Comprehensive Example using Apache Commons-Math

Consider the `testFirstDerivative` method shown in Listing 4.12: This test method evaluates the correctness of a numerical differentiation implementation using Apache Commons Math [62]. The test verifies the correctness of a method that calculates the first derivative of the sine function, which is the cosine function, at a particular x-value.

Initially, an anonymous implementation of the `UnivariateFunction` interface is created with the `value` method returning the sine of x. This function is then assigned to the variable `f`. Next, the `Differentiator.differentiate(f)` method is invoked to compute an approximation of the first derivative of the function `f`, which is then assigned to the variable `df`. A test point $x = 0.5$ is chosen, and the value of the approximated first derivative `df` is computed at this point, resulting in the value `actual`. In parallel, the expected value of the first derivative (cosine of x) is calculated using `FastMath.cos(x)` and is assigned to the variable `expected`. Finally, the test function employs an assert statement to validate the correctness of the numerical differentiation by comparing `expected` and `actual` within a tolerance of 10^{-10}. Listings 4.13, 4.14 and 4.15 show the output of each block from 2 to 6 upon analysing this test code. The positions of object references identified by block 5 are indicated in yellow in Listing 4.12.

```
@Test
public void testFirstDerivative() {
    final UnivariateFunction f = new UnivariateFunction() {
        public double value(double x) {
            return FastMath.sin(x);
        }
    };
    final UnivariateFunction df = Differentiator.differentiate(f);
    final double x = 0.5;
    final double actual = df.value(x);
    final double expected = FastMath.cos(x);
    Assert.assertEquals(expected, actual, 1e-10);
}
```

Listing 4.12: Testing the behaviour of classes from Apache Commons-Math.

```
["assertEquals(expected, actual, 1e-10);"] // block 2
["assertEquals(expected, actual, 1e-10)"] // block 3
["(expected, actual, 1e-10)"] // block 4
["actual", "expected"] // block 5
["actual", "expected", "df"] // block 6 - all object references
[390, 299, 400, 338, 308, 218] // block 6 - positions of all object
    references
```

Listing 4.13: The output of blocks 2 - 5.

```
[
   "final UnivariateFunction df = Differentiator.differentiate(f);",
   "final double expected = FastMath.cos(x);",
   "final double actual = df.value(x);"
]
```

Listing 4.14: The declaration statements extracted by block 6.

```
[
    "UnivariateFunction",
    "Differentiator",
    "FastMath"
]
```

Listing 4.15: The production classes identified by block 6.

Consider `UnivariateFunction` in the output in Listing 4.15: The test function `testFirstDerivative` creates an instance of an anonymous class that implements the `UnivariateFunction` interface, and an instance of this class is tested as part of the test function, which is why the `UnivariateFunction` item (which is actually an interface) appears in the output. In addition, notice that the object `f` is not considered to be tested. This is because none of its functions are being invoked directly in the test; only the static class `Differentiator` is considered to be tested in the context of finding the first derivative of `f`. Finally, notice that the statement-based matching algorithm does not consider the name of the file within which the test function `testFirstDerivative` lies and that the production classes in the output (Listing 4.15) bear no similarity to each other. This is in contrast to the filename-based matching algorithm, which would require knowledge of the filename and would return production classes whose names are similar.

4.3 Performance Analysis

Both the statement-based and filename-based matching algorithms are classification algorithms. Therefore, their performance will be assessed by their precision, recall, and F1-score. The subsections below describe the methodology used to assess and compare the performance of both algorithms to answer **RQ1**, namely, "Can test code be matched more accurately to production code using a statement-level approach?"

4.3.1 Performance Metrics

A true positive (TP) production file selection occurs when a production file is tested by a test file and the algorithm identifies it accurately. A false positive (FP) production file selection occurs when a production file is not actually tested by a test file but the algorithm reports that it is tested. Lastly, a false negative (FN) production file selection occurs when a production file is tested by a test file but the algorithm fails to identify it. Using TP, FP, and FN values, the precision (ρ), recall (λ), and F1-score (ψ) of a classification algorithm can be expressed mathematically as follows:

$$\rho = \frac{TP}{TP + FP} \tag{4.3}$$

$$\lambda = \frac{TP}{TP + FN} \tag{4.4}$$

$$\psi = 2 \cdot \frac{\rho \cdot \lambda}{\rho + \lambda} \tag{4.5}$$

A set of test files should be sampled to assess the performance of the matching algorithms. Each test file should be manually analysed, and the production files that are considered to be tested should be recorded. The TP, FP, and FN values corresponding to a single test file are obtained as follows: if the set `A` represents the truly tested production files and the set `B` represents the production files identified by a matching algorithm, then TP = `A` $\cap$ `B`, FP = `B` $\setminus$ `A`, and FN = `A` $\setminus$ `B`. To obtain a balanced representation of the performance of the matching algorithms, multiple projects should be analyzed. Naturally, this results in more than one value for precision, recall, and F1-score (one set of values for each project). Hence, the weighted precision, weighted recall, and weighted F1-score should be used to represent the overall performance of the algorithms, where the weight is determined by the number of test files in each project. If there are n projects with ω_i tests in the i^{th} project, then the weighted precision $\bar{\rho}$, weighted recall $\bar{\lambda}$, and weighted F1-score $\bar{\psi}$ are given by the following expressions:

$$\bar{\rho} = \frac{\sum_{i=1}^{n} \omega_i \cdot \rho_i}{\sum_{i=1}^{n} \omega_i} \tag{4.6}$$

$$\bar{\lambda} = \frac{\sum_{i=1}^{n} \omega_i \cdot \lambda_i}{\sum_{i=1}^{n} \omega_i} \tag{4.7}$$

$$\bar{\psi} = \frac{\sum_{i=1}^{n} \omega_i \cdot \psi_i}{\sum_{i=1}^{n} \omega_i} \tag{4.8}$$

4.3.2 Program Architecture

This subsection describes the main subsystems that are designed to assess the performance of the matching algorithms. Figure 4.5 is an architecture diagram of the parts that make up the validation system discussed in this section.

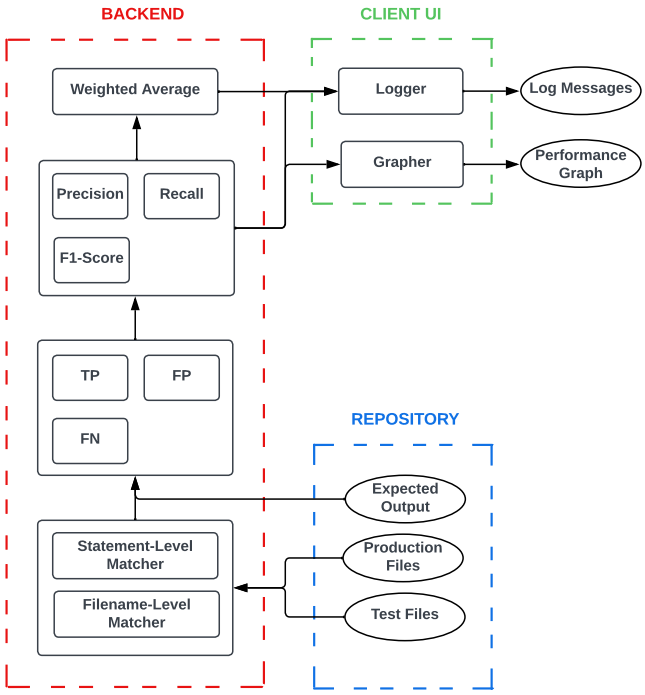


Figure 4.5: Architecture diagram of the performance assessing system.

Sampling and Manual Analysis of Files

The sampling subsystem provides a repository of files that are used to assess the performance of the matching algorithm. When this subsystem is executed once, it

takes the name of a project for which a set of test files is required as input. The output is the source code of a random test file (with the ".txt" extension) and an empty set (with the ".json" extension). This setup allows one to *manually* analyse a test file and to *manually* type the names of tested production files in the JSON file. The entries typed in the JSON file are the true values for the random test file. For example, given "swagger-core" as input, the subsystem could provide Listing 4.16 (AnnotationsUtilsTest from Swagger-Core [63]) and Listing 4.17 (an empty JSON file) below as output. Some lines in Listing 4.16 were omitted due to the length of the test file. Listing 4.16 is a random test file selected by the sampling subsystem; if the subsystem is executed again, it will provide a different test file.

```
// code omitted

public class AnnotationsUtilsTest {
    // code omitted

    @Test(dataProvider = "expectedSchemaFromTypes")
    public void resolveSchemaFromType(Class<?> aClass, Map<String, Object> expected) {
        Schema schema = AnnotationsUtils.resolveSchemaFromType(aClass, new Components(), null);

        Assert.assertEquals(schema.getType(), expected.get("type"));
        Assert.assertEquals(schema.getFormat(), expected.get("format"));
        Assert.assertEquals(schema.get$ref(), expected.get("$ref"));
    }

    // code omitted

    @Test(dataProvider = "expectedSchemaFromTypeAndFormat")
    public void getSchema(String methodName, Map<String, Object> expected) throws NoSuchMethodException {
        final Method method = getClass().getDeclaredMethod(methodName);
        Content annotationContent = method.getAnnotation(ApiResponse.class).content()[0];
        Optional<? extends Schema> schema = AnnotationsUtils.getSchema(annotationContent, new Components(), null);

        Assert.assertTrue(schema.isPresent());
        Assert.assertEquals(schema.get().getType(), expected.get("type"));
        Assert.assertEquals(schema.get().getFormat(), expected.get("format"));
        Assert.assertEquals(schema.get().get$ref(), expected.get("$ref"));
    }
    // code omitted
}
```

Listing 4.16: A random test file sampled from Swagger-Core.

```
[

]
```

Listing 4.17: An empty JSON file.

Consider Listing 4.17: an empty JSON file is generated so that one can manually type the names of production files that AnnotationsUtilsTest tests. For example, Listing 4.18 below shows the names of production files that are considered to be tested by `AnnotationsUtilsTest` after manual analysis. Notice that this set represents the true values corresponding to the test file. This set is referred to as the "expected output" because it is what is expected for the matching algorithms to return if they were to operate accurately.

```
[
    "AnnotationsUtils",
    "Content",
    "Schema",
    "Components"
]
```

Listing 4.18: The manually-populated JSON file (expected output).

Execution of Matching Algorithms

Once a reasonable number of test files have been sampled and analysed manually, the matching algorithms are executed with their respective inputs. In the case of the filename-based matching algorithm, the input is a single test filename and a set of all production files in the project in question. In the case of the statement-based matching algorithm, the input is the test code and a set of all production files in the project in question. The output of both matching algorithms is the set of production files labelled as being tested. For example, given AnnotationsUtils.txt in Listing 4.16 and a set of all production files that exist in Swagger-Core, the output of the statement-based matching algorithm is shown in Listing 4.19 below.

```
[
    "Components",
    "Schema",
    "AnnotationsUtils"
]
```

Listing 4.19: The output of the statement-based matching algorithm given Listing 4.16 as input.

Similarly, Listing 4.20 below is the output of the filename-based matching algorithm given the string "AnnotationsUtilsTest" and a set of all production files that exist in Swagger-Core.

```
[
    "AnnotationsUtils"
]
```

Listing 4.20: The output of the filename-based matching algorithm given Listing 4.16 as input.

Confusion Matrix Calculator

The confusion matrix calculator is responsible for comparing the output of the matching algorithms to the expected output in order to obtain TP, FP, and FN values. Proceeding with the example of AnnotationsUtilsTest (Listing 4.16), Table 4.1 below shows the corresponding TP, FP, and FN values for both matching algorithms.

Table 4.1: TP, FP and FN values

Classifier	**TP**	**FP**	**FN**
Statement-Based	3	0	1
Filename-Based	1	0	3

Precision, Recall and F1-score Calculator

The precision, recall, and F1-score calculator is responsible for calculating precision, recall, and F1-score given the TP, FP, and FN values. Precision is calculated using Equation (4.3), recall is calculated using Equation (4.4), and F1-score is calculated using Equation (4.5). Continuing from the previous example, the precision of the statement-based matching algorithm is 1, and its recall is 0.75. Therefore, its F1-score is 0.857. On the other hand, the precision of the filename-based matching algorithm is 1, and its recall is 0.25. Therefore, its F1-score is 0.4. Note that these values are obtained from a single test file for a single project. The performance analysis of both matching algorithms will involve more than one project, each having more than one test. Hence, the precision, recall, and F1-score will be calculated after analysing *all* the sampled test files for each project.

Weighted Average Calculator

The weighted average calculator is responsible for mapping the precision, recall, and F1-score values of many projects to a *single* precision, recall, and F1-score value, which is representative of the overall performance of the statement-based matching algorithm or the filename-based matching algorithm. For example, assume that three projects are analysed: Swagger-Core [63], Docker-Java [64], and Apache Commons-Math [62]. Twenty test files are analysed for Swagger-Core, thirty test files are analysed for Docker-java, and fifteen test files are analysed for Apache Commons-Math. Hence, $\omega_1 = 20$, $\omega_2 = 30$, and $\omega_3 = 15$. Furthermore, assume that the values in Table 4.2 below are the precision, recall, and F1-score values of the statement-based matching algorithm executed with this data as input:

Table 4.2: Precision, Recall and F1-score values

Project	**Test Files**	**Precision**	**Recall**	**F1-score**
Swagger-Core	20	0.91	0.72	0.8
Docker-Java	30	0.93	0.67	0.78
Commons-Math	15	0.95	0.76	0.84

In this case, $\bar{\rho} = 0.93$, $\bar{\lambda} = 0.71$, and $\bar{\psi} = 0.8$ using Equations (4.6), (4.7), and (4.8), respectively. These values represent the overall performance of the statement-based matching algorithm given these three projects.

Logger

The purpose of the logger is to log the progress of the performance analysis process via the command line. The input is a group of performance metrics and information from the performance analysis system. The output is an organised set of log messages. The log messages are organised into three groups:

- The performance of both classifiers given the latest test file sampled for a particular project.
- The performance of both classifiers given all the sampled test files for a particular project.
- The performance of both classifiers given all the sampled test files for all sampled projects.

Figure 4.6 is an example of the output of the logger on the command-line interface. Notice the three groups of log messages as discussed above: First, the logger shows the output of both classifiers given `XMLGregorianCalendarTest` from Swagger-Core. This makes it easy to manually compare the output of both classifiers to the expected output stored in the JSON file that was last modified. Next, the logger shows the precision, recall, and F1-score of both classifiers given all the sampled tests for Swagger-Core. Finally, the logger shows the weighted precision, weighted recall, and weighted F1-score of both classifiers given all the sampled tests for all sampled projects.

```
C:\Users\gerald.kirui\Software-Masters\Tools>py fileAnalyserValidator.py
---------------- Single Test File Analysis (XMLGregorianCalendarTest.java from SWAGGER-CORE)
Statement-Level Matcher : {'Schema', 'ModelConverters'}
Filename-Level Matcher  : {}
---------------- Single Project Analysis : (SWAGGER-CORE) ----------------------------
                          Precision     Recall     F1-Score
Statement-Level Matcher :  1.0000        0.8605     0.9250
Filename-Level Matcher  :  0.1111        0.0233     0.0385
(Test files analysed : 14)
---------------- Multi-Project Analysis  ----------------------------
                          Weighted Precision     Weighted Recall     Weighted F1-Score
Statement-Level Matcher :  0.8416                0.7116              0.7711
Filename-Level Matcher  :  0.4947                0.1401              0.2183
(Test files analysed : 500, Projects analysed : 16)
```

Figure 4.6: The output of the logger via the command-line.

Grapher

The grapher is responsible for plotting the F1-score values of several projects as a bar graph. The F1-score values of both the statement-based matching algorithm and the filename-based matching algorithm are plotted as separate bars for the same project. This provides a visual representation of the performance difference between the two matching algorithms.

4.4 Bootstrap Confidence Interval

Up to this point, a method to calculate the weighted F1-score of a matching algorithm has been provided. However, the weighted F1-score alone does not provide any information on the confidence in the performance measured as a function of the number of test files used. This means that, without a method to determine the confidence level in the weighted F1-score for a given sample size of test files, one cannot be certain of the weighted F1-score that is reported.

Consider a naive approach to establish confidence in the weighted F1-score: one can collect a very large sample of test files and test the matching algorithm with it. In this case, the measured weighted F1-score will approach the true weighted F1-score of the system. However, this is infeasible since practical experiments usually have time and budget constraints. A better approach is to find the bootstrap confidence interval for the weighted F1-score calculated from a sample of test files. There are three main steps for determining the confidence interval of a metric [65]:

1. Create a distribution based on K bootstrapped samples for the statistic u.
2. Sort the values in ascending order.
3. The lower bound of the 100(1 - 2α) percent confidence interval is the K α^{th} value, and the upper bound is the K $(1-\alpha)^{th}$ value in the sorted distribution.

K represents the number of pseudo-samples that are generated from the original sample, u represents the statistic of interest (e.g., mean), and α represents a variable that determines the confidence level (e.g., if α equals 0.025, then the confidence level is 95%). In this study, $u \equiv$ weighted F1-score and $\alpha = 0.025$.

4.5 Conclusion

The design and implementation of the filename-based and statement-based matching algorithms, as well as the performance assessment system, have been presented. The filename-based matching algorithm is designed using the function `difflib.SequenceMatcher` from the Python standard library, which is based on the Ratcliff-Obsershelp algorithm. The statement-based matching algorithm comprises six stages, each of which applies static operations to the test code parsed as text. The performance assessment procedure involves sampling test files, analysing and recording the tested production

files, executing the matching algorithms on the sampled test files, and calculating the weighted F1-score. Auxiliary parts of the performance assessment system include logging and graphing. The bootstrap confidence interval was proposed to determine the true weighted F1-score of the matching algorithms by creating and analysing the distribution of pseudo-samples based on the original sample obtained.

Chapter 5

Test-With Development Adherence Metric

This chapter presents the definitions of the mathematical functions that evaluate the absolute and relative test-with development (TWD) adherence of projects, as well as the strategy to determine the median development step-size of a project. The absolute TWD adherence mathematical function operates on time deltas and method coverage between added or modified assertions and the production methods they test to obtain the desired output, while the relative TWD adherence mathematical function clusters, transforms, and normalises a set of absolute TWD adherence values to obtain the desired output. The system that evaluates the median development step-size of a project considers the series of changes to production methods that are covered by an added or modified assertion.

5.1 Introduction

Since a test can cover more than one production class, one or more production files are associated with a test file. TDD emphasizes writing tests before their corresponding production code; the fact that tests are committed immediately before production code does not guarantee that they were actually written before production code. Due to uncertainty in the exact order in which tests and production code were written, the approach taken in this study is to relax the definition of TDD and instead consider test-with development (TWD), which is defined as the co-evolution of tests and production code modifications that is independent of the order in which tests and production code were written, similar to the definition proposed by Bannerman [66].

This is corroborated by Zaidman et al. [18], who found that test-driven development is translated as a simultaneous commit of a production source file alongside its unit test. In practice, it is unlikely to find projects where tests and production code are always committed at the same time throughout their development. The application of TWD between test code and its corresponding production code is evidenced by their being committed at the same time or with a small time difference between them, i.e., the definition of TDD is relaxed so that TDD practice is analyzed on a spectrum. Recall that the order of committing test code and production code does not guarantee the time when test code or production code was written, since test code may be written before production code or at the same time and may be committed in a different order.

5.2 Absolute TWD Adherence

Absolute TWD adherence is a metric that represents the degree to which TWD was applied during the development of a software project. Such a metric is necessary because it provides insight into the level of test coverage and the overall quality of the project. By measuring TWD adoption, teams can assess whether they are using a practice that can lead to more robust, maintainable, and testable code. Additionally, such a metric can be used to identify areas for improvement and track progress over time. Figure 2.1 in Chapter 2 depicts the main steps associated with TDD as described by Beck [1].

To justify the use of TWD as a useful measure for TDD, consider one case raised by Beller et al. [45] where TDD in the strict sense may vary from the recognition of TDD in practice, namely compilation errors. Compilation errors may affect the strict application of TDD during the test creation phase of TDD when the class or method that is asserted against does not exist yet. To have an executing but failing test, developers would need to mix in the modification or creation of production code. This deviation from the strict separation between test and production code may affect how TDD is recognised in a VCS.

The approach of considering TWD as a useful measure for TDD makes up for the deviation between the practical and strict application of TDD as follows: the time differences allow for the assessment of the temporal relationship between a test assertion and the production code that it covers. The subsections below outline how Figure 2.1 in Chapter 2 can be used to detect how close to the ideal of TDD a software project was developed, in order to answer **RQ2**: “How can such a matching

algorithm be used to measure the TDD adherence of a project?" Figure 5.1 describes a high-level overview of the core components of the system.

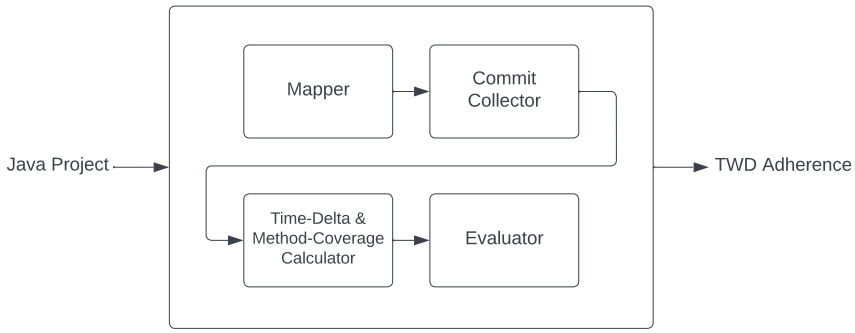


Figure 5.1: The core components of the TWD adherence evaluation system.

The mapper groups all the production files that each test file covers; the commit collector organises data from commits into a suitable data structure for manipulation; the time delta calculator transforms the data structure into a list of time deltas; and the evaluator transforms a list of time deltas into a single value that represents the absolute TWD adherence of the software project.

5.2.1 Mapping Tests to Production Files

The implementation of mapping tests to production files is discussed in Section 4.2 in Chapter 4. Figure 5.2 below shows the mapping technique between tests and production files for a project. The output is a collection of all the test files, including the production files that each of them has ever tested during the lifecycle of a project, even if a production class is deleted from the project.

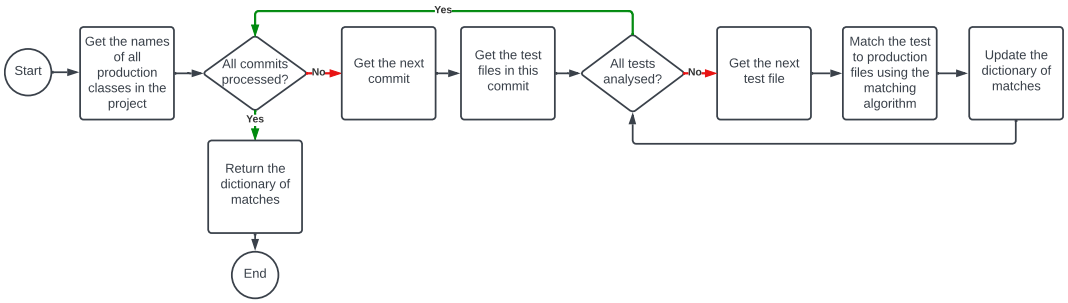


Figure 5.2: The mapping technique between tests and production files.

As a demonstration of the mapper in action, suppose there is a Java project called Shapes. The Shapes project allows a user to get the perimeter of a rectangle or a circle.

To make sure that the intended functionality of the project is achieved, the project also includes tests. The production code for getting the perimeter of a rectangle is contained in `Rectangle.java`, as shown in Listing 5.1. It has a constructor to specify the length and breadth of the rectangle and one method, `getPerimeter()`, which returns the perimeter of a rectangle.

```
1  public class Rectangle {
2      private double length;
3      private double breadth;
4
5      public Rectangle(double length, double breadth) {
6          this.length = length;
7          this.breadth = breadth;
8      }
9
10     public double getPerimeter() {
11         return 2 * (this.length + this.breadth);
12     }
13 }
```

Listing 5.1: Rectangle.java

Like `Rectangle.java`, `Circle.java` implements the logic for getting the circumference of a circle, as shown in Listing 5.2. It has a constructor to specify the radius of the circle and one method, `getCircumference()`, which returns the circumference of a circle. Notice that the method is implemented incorrectly. This is intentional, as it is part of the explanation of getting the time difference between an assertion and the production method it tests, henceforth referred to as a *time delta*.

```
1  import java.lang.Math.*;
2
3  public class Circle {
4      private double radius;
5
6      public Circle(double radius) {
7          this.radius = radius;
8      }
9
10     public double getCircumference() {
11         return Math.PI * this.radius; // Initially incorrect
12     }
13 }
```

Listing 5.2: Circle.java

Lastly, the tests for the two classes are implemented in `ShapesGeometryTest.java`. This test file has two methods, namely `testPerimeter` and `testCircumference`, both of which verify the behaviour of objects created by `Rectangle.java` and `Circle.java`. Listing 5.3 below shows the implementation of `ShapesGeometryTest.java`.

```
1  import org.junit.*;
2  import java.lang.Math.*;
3
4  public class ShapesGeometryTest {
5      @Test
6      public void testPerimeter(){
7          double length = 4;
8          double breadth = 2;
9          Rectangle rectangle = new Rectangle(length, breadth);
10         double perimeter = rectangle.getPerimeter();
11         double expectedPerimeter = 2 * (length + breadth);
12         assertEquals(perimeter, expectedPerimeter);
13     }
14
15     @Test
16     public void testCircumference(){
17         double radius = 5;
18         Circle circle = new Circle(radius);
19         double circumference = circle.getCircumference();
20         double expectedCircumference = 2 * Math.PI * 5;
21         assertEquals(circumference, expectedCircumference);
22     }
23 }
```

Listing 5.3: ShapesGeometryTest.java

It is clear that both `Rectangle.java` and `Circle.java` are tested by `ShapesGeometry-Test.java`. Hence, when this project is passed as input into the mapping system, it is expected that it will match `ShapesGeometryTest.java` to both `Rectangle.java` and `Circle.java`. See Appendix A, Figure A.1 for a technical representation of the match results.

5.2.2 Collection of Data from Commits

Recall that the goal is to determine the TWD adherence of a software project. Up to this point, the only information obtained is a collection of the production files that are covered by each test in the project. To determine TWD adherence, commit behaviour should be analysed as well. Hence, the purpose of collecting data from

commits is to convert a Git repository of a Java project into a suitable data structure that will ultimately facilitate the evaluation of the TWD adherence metric. As a description of the data structure, consider an example of one way that the Shapes project could have been developed:

1. In the first commit, the developer creates the `Rectangle.java` file with the `Rectangle` class, including the `getPerimeter` method.
2. Next, the developer creates a test file named `ShapesGeometryTest.java`. The developer also creates a test method called `testPerimeter` that tests the production method created in the first commit. Suppose the test is committed 76 seconds after the first commit.
3. Thereafter, the developer creates another production class called `Circle`. In this class, the developer writes the implementation for getting the circumference of a circle. Initially, the implementation is logically incorrect. Suppose this commit is made 103 seconds after the second commit.
4. After realising the flawed implementation, the developer proceeds to correct the mistake. Suppose this commit is made 79 seconds after the third commit.
5. Lastly, the developer returns to `ShapesGeometryTest.java` to extend it by including a test method for the `getCircumference` method. Suppose this commit is made 625 seconds after the fourth commit.

Notice that in this project, each commit has changes to either a production file or a test file. In general, a project typically contains multiple commits, each of which may have changes to multiple production or test files or none at all. The UML diagram below shows a representation of this data structure. Note that this data structure and the Commit object are not the same as Git's data structure; this data structure is derived from a Git commit.

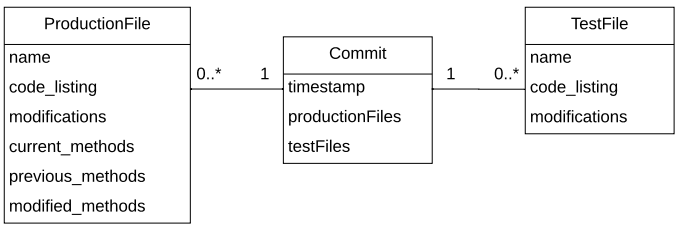


Figure 5.3: A UML diagram of the commit and file data structure.

The `timestamp` field stores the date and time at which a commit was made. It is from this field that the temporal aspect of commit behaviour is made available to the system. This field is critical for the calculation of the TWD adherence of a project. The `name` field stores the name of a production file or test file that is part of a commit. Consider the second commit in the example above: only a single test file is included. Therefore, the `Commit` object corresponding to the second commit will have only one `TestFile` object, whose name is `ShapesGeometryTest`. The `code_listing` field stores the source code of a Java file parsed as text. Consider the fourth commit, wherein the `getCircumference` method of the `Circle.java` class is corrected. In this case, the source code in Listing 5.4 will be parsed as text and assigned to the `code_listing` field.

```
1  import java.lang.Math.*;
2
3  public class Circle {
4      private double radius;
5
6      public Circle(double radius) {
7          this.radius = radius;
8      }
9
10     public double getCircumference() {
11         return 2 * Math.PI * this.radius; // corrected
12     }
13 }
```

Listing 5.4: Circle.java (corrected).

Notice that the only difference between Listing 5.2 and 5.4 is Line 11. This fact will help with an appreciation of the `modifications` field, which is discussed in Section 5.4, as it is closely associated with the production method churn system. The `current_methods` field stores a list of the names of methods that exist in a production file at the time when a commit was made. The `previous_methods` field stores a list of the names of methods that existed in a production file prior to the time when a commit was made. Some of these methods may not be present in the `current_methods` since a deletion may have occurred. The `modified_methods` field stores a list of the names of methods whose implementations were modified at the time when a commit was made.

5.2.3 Time Deltas

The purpose of the time delta calculator in Figure 5.1 is to convert the data structure discussed previously to a list of time deltas. It finds the time difference (in seconds) between an added or modified assertion's commit timestamp and the timestamp of the latest version of the tested production method that was committed previously. It is assumed that the tests pass.

The following example demonstrates how the time deltas are obtained. Consider Figure 5.4 below, which is a representation of the commit history of the Shapes project.

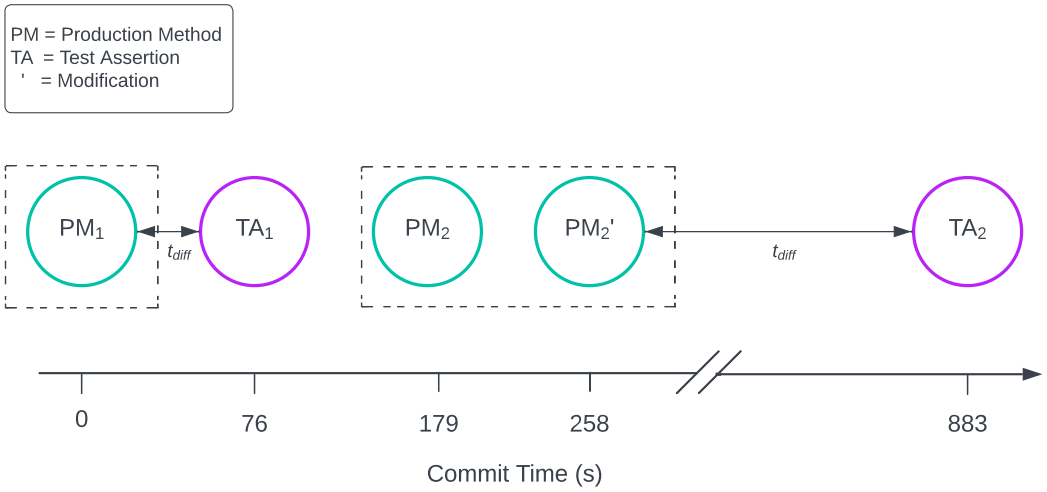


Figure 5.4: A representation of the time deltas of the Shapes project.

PM_1 refers to the production method `getPerimeter` of the `Rectangle` class, and TA_1 refers to the test assertion in `testPerimeter` of the `ShapesGeometryTest` class. PM_2 refers to the incorrectly implemented production method `getCircumference` of the `Circle` class, while PM_2' refers to the correctly implemented production method `getCircumference` of the `Circle` class. Lastly, TA_2 refers to the test assertion in `testCircumference` of the `ShapesGeometryTest` class.

The approach taken is to match the test assertion with the most recent version of the corresponding production method. Therefore, from Figure 5.4, it is clear that a total of two time deltas are associated with the commit history; the first one is the time delta between PM_1 and TA_1, and the second one is the time delta between PM_2' and TA_2. The two time deltas are expected to be 76 s and 625 s, respectively. Table 5.1 shows the expected output of the subsystem. Appendix A Listing A.2

shows a technical representation of the output of the time deltas subsystem when it is presented with the data structure obtained from the Shapes project.

Table 5.1: Output of the time delta subsystem

Production Method	**Test Method**	**Time Delta**
PM_1	TA_1	76
PM_2	TA_2	N/A
PM_2'	TA_2	625

Table 5.2 shows the legend for the symbols used in the diagrams. Figure 5.5 and Figure 5.6 below show how the system processes the data structure described previously to calculate the time deltas.

Table 5.2: The legend for the symbols used in Figure 5.5 and Figure 5.6.

Symbol	**Description**
C_1	The commit that contains changes to the test file currently being analyzed
PF_T	The production file that is covered by the test file in C_1
PM_T	The method in PF_T that is covered by the test file in C_1
C_2	A commit that was done before C_1 (C_2 can also be the same as C_1)
PF	A production file whose changes were committed in C_2
PM_{list}	The list of all methods in PF that were modified and included in C_2

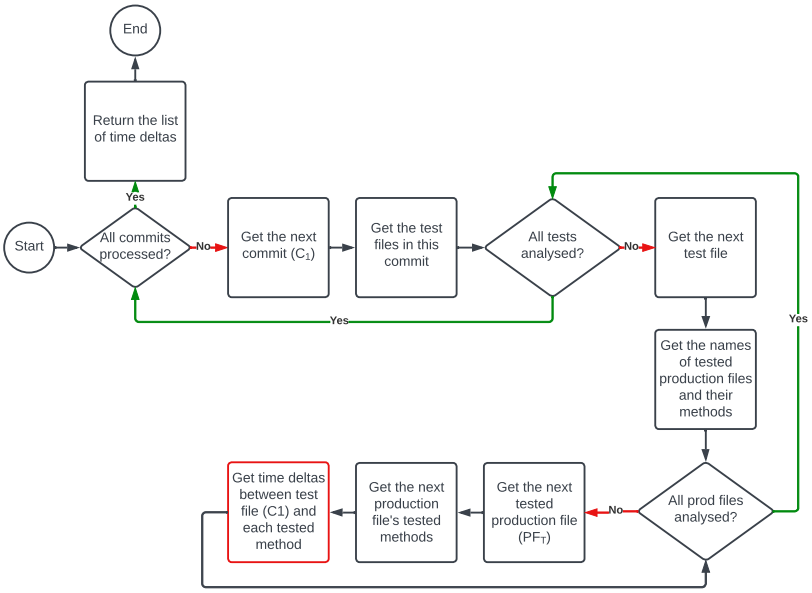


Figure 5.5: Calculating time deltas between all modified tests and each method they test.

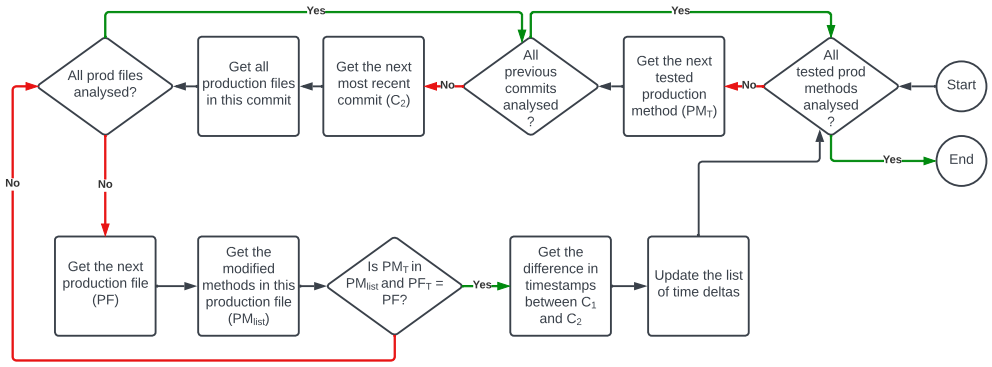


Figure 5.6: Getting the time deltas between a single test file and each method it tests

5.2.4 Test Code Committed Prior to Production Code

This subsection demonstrates the findings obtained when the percentage of test-first assertion-production method pairs in 100 Java projects were analyzed. The details of the list of projects can be found in Appendix B. This is done to substantiate the approach taken in this study to consider only test-with and test-last pairs when finding time deltas.

There are four possibilities that can occur when the statement-based matching algorithm attempts to establish a link between an assertion and a production method: a match is established where the assertion was committed strictly before the production

method (test-first pair); at the same time as the production method (test-with); after the production method (test-last); or none of the above (i.e., a match was unsuccessful due to limitations in the matching algorithm).

Let N_{TL} represent the number of successful test-last matches, N_{TW} represent the number of successful test-with matches, N_{TF} represent the number of successful test-first matches, and n_e represent the number of unsuccessful matches. For each project, these variables were evaluated as follows:

Firstly, for each assertion in the commit history of a project, the system assumes that the corresponding production method that is covered exists in a previous commit or in the same commit. Hence, the system analyses every commit in the range [0, k], where k is the index of the commit containing the assertion. This represents $(N_{TL} + N_{TW})$. If a match is not established in this range, the system assumes that the production method exists in a later commit. Hence, it proceeds to analyse all commits in the range [$k + 1$, m], where m is the index of the latest commit. This represents N_{TF}. If a match is still not established at this point, the system assumes that a detection error has occurred; therefore, it updates the value n_e. Detection errors are excluded from the calculation. The ratio of test-first pairs R_{TF} is evaluated by the following equation:

$$R_{TF} = \frac{N_{TF}}{(N_{TF} + N_{TW} + N_{TL})} \tag{5.1}$$

Consider the boxplot in Figure 5.7 below which was obtained by finding the ratio of test-first pairs in 100 Java projects using Equation (5.1). The median percentage of assertion-production method pairs that are strictly test-first in nature was found to be only 3.56%. This finding is corroborated by Hanhela [19], who found that only 5% of the test-production file pairs were test-first in nature. Therefore, the vast majority of tests are committed at the same time or after the completion of the production code they test. For this reason, the time delta subsystem will consider changes to production methods that were committed earlier than their corresponding test or at the same time.

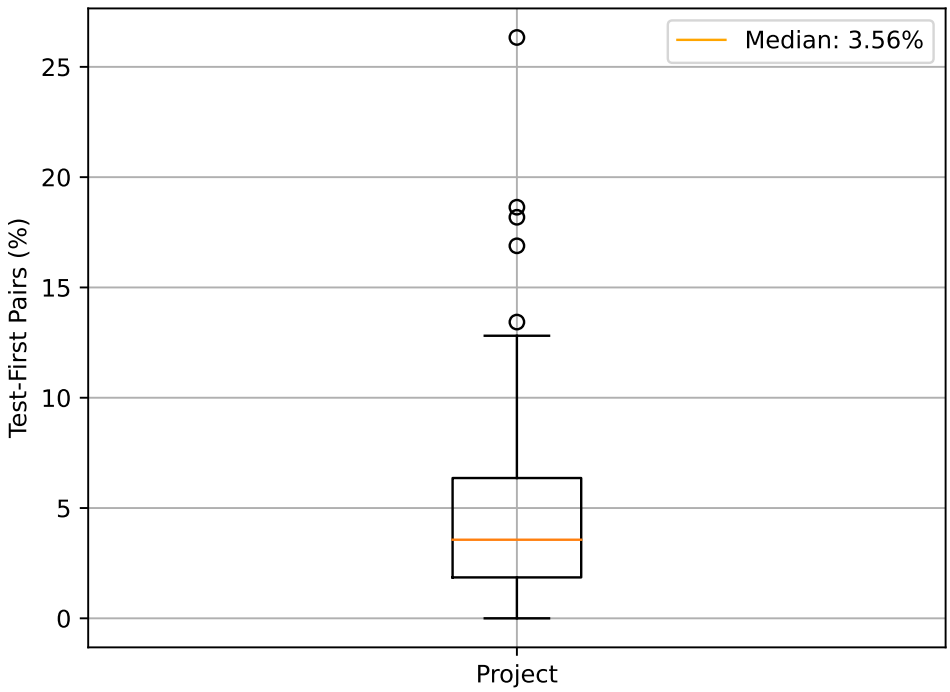


Figure 5.7: The distribution of the percentage of test-first pairs in 100 Java projects.

5.2.5 Method Coverage

Small time deltas are not the only key indicator of proper application of TWD. Another key indicator is test coverage. A strict application of TDD would mean that every line of production code that is written is covered by some test code. Since a static approach is taken in this study, method coverage instead of line or branch coverage is considered, as it is easier to detect without the need to build and configure each project and execute the test suite.

The following procedure is used to determine the method coverage contributed by an assertion in relation to a production class: First, the production methods of a production class that are directly or indirectly tested by an assertion are recorded. A directly-tested production method is a public method of a production class that is invoked in a test file. An indirectly-tested production method is a public or private method of a production class that is invoked by a directly-tested production method. The directly and indirectly tested production methods form set A. Secondly, all the production methods that exist in a production class at the time when an assertion is written are recorded. These methods form set B. The method coverage is the ratio of the number of elements in set A to the number of elements in set B. Hence, the method coverage ζ of an assertion is given by Equation 5.2 below:

$$\zeta = \frac{|A|}{|B|} \tag{5.2}$$

Constructors are ignored since they will always be covered in terms of method coverage since an object will always need to be created, and thus it is always assumed to be working properly. For example, consider a class called `BankAccount` as shown in Listing 5.5, which represents a basic model for a bank account. It contains attributes and methods to manage the account's balance. The test code in Listing 5.6 tests the functionality of the `BankAccount` class. In particular, it verifies the behaviour of the `deposit` and `getBalance` methods. If `BankAccount` and `TestBankAccount` are committed in a project, the method coverage of the assert statement on line 10 in Listing 5.6 will be calculated as follows: The methods in the `BankAccount` class that are directly tested by the assertion are `deposit` and `getBalance`. Moreover, since the private method `addToBalance` is invoked by `deposit`, it is considered indirectly tested by the assertion. Hence, all the methods (except the constructor) in `BankAccount` are either directly or indirectly tested. Therefore, the method coverage of the assertion with respect to `BankAccount` is 100%, even though `addToBalance` is not explicitly tested.

```java
public class BankAccount {
    private double balance;

    public BankAccount() {
        balance = 0.0;
    }

    public void deposit(double amount) {
        if (amount > 0) {
            addToBalance(amount);
        }
    }

    private void addToBalance(double amount) {
        balance += amount;
    }

    public double getBalance() {
        return balance;
    }
}
```

Listing 5.5: BankAccount.java

```java
import org.junit.Test;
import static org.junit.Assert.assertEquals;

public class TestBankAccount {
    @Test
    public void testDeposit() {
        BankAccount account = new BankAccount();
        account.deposit(100.0);
        double balance = account.getBalance();
        assertEquals(100.0, balance, 0.01);
    }
}
```

Listing 5.6: TestBankAccount.java

5.2.6 Evaluating Absolute TWD Adherence

The purpose of this subsystem is to reduce a list of time deltas and method coverage to a single value, referred to as absolute TWD adherence. This represents how closely the development of a project adhered to TWD. Let the mathematical function that maps time deltas and method coverage to absolute TWD adherence be represented

by $\boldsymbol{f}$. There are certain properties of $\boldsymbol{f}$ that must hold true if $\boldsymbol{f}$ is to adequately represent absolute TWD adherence. These will be discussed, and a reason for each property will be given with reference to the practical application of TDD. Recall that a project has multiple tests, and a test has multiple assertions. Hence, $\boldsymbol{f}$ must adequately assess the TWD adherence first at the assertion level, then at the test file level to finally assess the TWD adherence of an entire project.

TWD Adherence of a Single Assertion-Production Method Pair

Suppose there are two projects: Project A and Project B. Developers of Project A include tests alongside production code changes in the same commit and aim to achieve high coverage of their production code. Developers of Project B, on the other hand, only commit tests that correspond to changes to production code after several days with low code coverage. The smaller the time difference between a test and a given set of changes to production code and the higher the method coverage, the more the programming style resembles TDD (as depicted in Figure 2.1).

Consider a single assertion and a single production method that it tests directly. It stands to reason that the higher the adherence to TWD, the smaller the difference in timestamps between the assertion and the production method. Since the median percentage of test-first pairs was found to be only 3.56%, test-first pairs are ignored. Hence, the difference will always be positive. This suggests a monotonically decreasing relationship between the TWD adherence of a *single* assertion-production method pair and the increasing difference between their timestamps. Let δ represent the time difference between a single assertion and the production method it tests. Then, the function $\boldsymbol{h}(\delta)$ that represents the TWD adherence of the assertion-production method pair must exhibit the following behaviour for it to be monotonically decreasing with respect to δ:

$$\boldsymbol{h}(\delta_1) \leq \boldsymbol{h}(\delta_2) \text{ if } \delta_1 \geq \delta_2 \tag{5.3}$$

The following definition for $\boldsymbol{h}(\delta)$ is proposed:

$$\boldsymbol{h}(\delta) = e^{-\delta} \qquad \delta \geq 0 \tag{5.4}$$

Where δ is measured in days. A unit of days was chosen because the median time deltas of 100 Java projects were found to range from 0 to 647.41 days, with a median

of 123.76 days, as shown in Figure 5.8 below. Using seconds as the unit would result in a very large range of values, which might not provide a meaningful representation of the data. The precision of seconds might be too fine-grained for analysis, thus making it difficult to interpret the results and draw meaningful conclusions. On the other hand, using weeks as the unit might lead to a loss of detail, as it would aggregate the time deltas into larger intervals. This could potentially hide important variations within the data and make it harder to analyse specific patterns or trends.

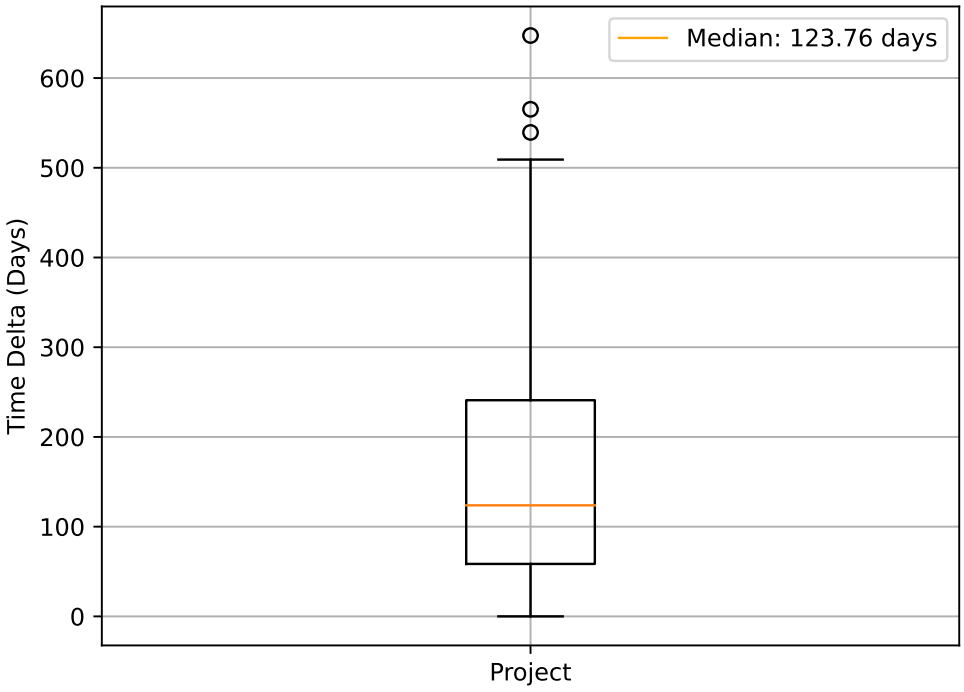


Figure 5.8: The distribution of median time deltas in 100 Java projects

The definition of $\boldsymbol{h}(\delta)$ as described in equation 5.4 was chosen because of its behaviour when $\delta = 0$ and when $\delta \to \infty$, because it is neither indeterminate nor unbounded in both cases. Consider the following example to demonstrate the monotonicity of $\boldsymbol{h}(\delta)$: Suppose, in one instance, the difference in timestamps between an assertion and a production method is 60 minutes. Suppose, in another instance, it is 30 minutes. Converted to days, $\delta_1 = 1/24$ and $\delta_2 = 1/48$. Applying the function $\boldsymbol{h}$, the result is $\boldsymbol{h}(\delta_1) = 0.961$ and $\boldsymbol{h}(\delta_2) = 0.984$. Since $\boldsymbol{h}(\delta_1) \leq \boldsymbol{h}(\delta_2)$ and $\delta_1 \geq \delta_2$, $\boldsymbol{h}$ is indeed monotonically decreasing with respect to δ.

TWD Adherence of Multiple Changes between a Single Test File and a Set of Production Files

Suppose there are two projects: Project A and Project B. Developers of Project A aim to achieve high method coverage when writing tests, whereas developers of Project B employ very little testing.

With these assumptions, there should be a larger number of unique assertions in Project A than in Project B; hence, Project A should achieve a higher absolute TWD adherence value than Project B (all else being equal). A unique assertion is one that is added or modified in a commit. However, a large number of assertions is not the only factor that should be considered, since development techniques such as TLD may also yield a large number of assertions if the whole project is considered. One key difference between TWD and TLD is the temporal aspect of committing test code and the corresponding production code, i.e., in TWD, production code is committed together with tests or is closely preceded or succeeded by tests, which implies a small time difference between them. On the other hand, in TLD, tests are committed long after changes to production code are committed. Therefore, apart from the total number of assertions in a project, the frequency of changes and the temporal relationship between tests and production code are also considered.

Naturally, a project that applies TWD should have more test-production code changes that have time deltas that are closer to zero than one that applies TLD, which means that the former should have a higher absolute TWD adherence than the latter (all else being equal). In other words, a TWD project should *ideally* exhibit a pattern characterized by only time deltas that are equal to zero when analyzing timestamps between test code commits and production code commits, whereas a TLD project should ideally exhibit clearly defined stages of testing and code development, resulting in larger time deltas.

Another important aspect is the code coverage contribution made by the assertions in a test file. Consider two projects: Project C and Project D. Project C has one test file with one test method that contains one assertion in the same commit. The test file covers one production file with one method completely. Project D, in contrast, has two test files, each with one assertion method in the same commit. However, the two test files cover only one production method each in a production file that has ten production methods in total. If one considers only the number of assertions made in a project, then one would conclude that Project D adheres to TWD more than Project C, which is flawed since Project C has a method coverage of 100% whereas

Project D has a method coverage of only 20%.

In light of this, consider the following as a definition for a mathematical function that assesses the TWD adherence of a set of changes between assertions and production methods for a single test file: Let $\vec{\delta} = [\delta_1, \delta_2, \delta_3, ..., \delta_n]$ represent a list of time deltas between a set of changes of assertions and each of the production methods they test, and let $\vec{\zeta} = [\zeta_1, \zeta_2, \zeta_3, ..., \zeta_n]$ represent the method coverage of each assertion in relation to the respective production classes of each production method. For every additional assertion-production method pair, TWD adherence must strictly increase. This relationship suggests a monotonically increasing behaviour. Hence, the function $\boldsymbol{g}(\vec{\delta}, \vec{\zeta})$ that represents the TWD adherence of the series of changes between assertion-production method pairs must exhibit the following behaviour for it to be monotonically increasing:

$$\boldsymbol{g}(\vec{\delta}_1, \vec{\zeta}_1) \geq \boldsymbol{g}(\vec{\delta}_2, \vec{\zeta}_2) \text{ if } \vec{\delta}_1 \supseteq \vec{\delta}_2 \text{ and } \vec{\zeta}_1 \supseteq \vec{\zeta}_2 \tag{5.5}$$

The following definition for $\boldsymbol{g}(\vec{\delta}, \vec{\zeta})$ is proposed:

$$\begin{aligned} \boldsymbol{g}(\vec{\delta}, \vec{\zeta}) &= \sum_{i=1}^{|\vec{\delta}|} \zeta_i \, \boldsymbol{h}(\delta_i) \\ &= \sum_{i=1}^{|\vec{\delta}|} \zeta_i \, e^{-\delta_i} \end{aligned} \tag{5.6}$$

Where $|\vec{\delta}|$ is the number of elements in the vector $\vec{\delta}$, which corresponds to the time deltas between changes to assertions in a *single* test file and the production methods they test, and $|\vec{\zeta}|$ is the number of elements in the vector $\vec{\zeta}$ that corresponds to the method coverage of each assertion on the production class of the production method that it tests. Note that $|\vec{\delta}| = |\vec{\zeta}|$. Consider the following example to demonstrate the behaviour of $\boldsymbol{g}(\vec{\delta}, \vec{\zeta})$: Suppose test file A has the following time deltas in days between the production methods it tests: $\vec{\delta_A} = [1/24, 1/48, 1/36, 1/12]$ and suppose the method coverage that corresponds to this is $\vec{\zeta_A} = [1, 1, 1, 1]$, where each element represents the method coverage as a percentage. In addition, suppose another test file B has the following time deltas in days between the production methods it tests: $\vec{\delta_B} = [1/24, 1/48, 1/36, 1/12, 1/15]$ and suppose the method coverage that corresponds to this is $\vec{\zeta_B} = [1, 1, 1, 1, 1]$. Applying $\boldsymbol{g}$ to the two arrays, the result is $\boldsymbol{g}(\vec{\delta}_A, \vec{\zeta}_A) = 0.174$ and $\boldsymbol{g}(\vec{\delta}_B, \vec{\zeta}_B) = 0.233$. Notice that $\vec{\delta_B} \supseteq \vec{\delta_A}$, $\vec{\zeta_B} \supseteq \vec{\zeta_A}$ and $\boldsymbol{g}(\vec{\delta}_B, \vec{\zeta}_B) \geq \boldsymbol{g}(\vec{\delta}_A, \vec{\zeta}_A)$. Hence, $\boldsymbol{g}(\vec{\delta}, \vec{\zeta})$ is monotonically increasing.

TWD Adherence of an Entire Project

Thus far, a definition has been provided for the function $\boldsymbol{g}(\vec{\delta},\ \vec{\zeta})$ that assesses the TWD adherence of a set of changes between assertions of a single test file and the production methods they test. However, a project consists of multiple test files, each with their own assertions, which in turn have their own method coverage. The TWD adherence of an entire project can be considered an aggregation of the TWD adherence of each test file, taking into account the time deltas and method coverage of each assertion within it. Therefore, the mathematical function $\boldsymbol{f}$ that evaluates the TWD adherence of a project can be defined as follows: Let $\vec{\Delta} = [\vec{\delta}_1,\ \vec{\delta}_2,\ \vec{\delta}_3,\ ...,\ \vec{\delta}_n]$ represent a matrix of time deltas between assertions and the production methods they test, for every test file in every commit of a project, and let $\vec{\Gamma} = [\vec{\zeta}_1,\ \vec{\zeta}_2,\ \vec{\zeta}_3,\ ...,\ \vec{\zeta}_n]$ represent a matrix of the method coverage of each assertion on the production class it tests. Since TWD involves writing test code and production code in close succession with a small time difference between their commits or within the same commit, TWD adherence at the project level should strictly increase with every assertion that is added or modified and included in a commit. This suggests a monotonically increasing behaviour of the function that evaluates the TWD adherence of a project. Therefore, the function $\boldsymbol{f}(\vec{\Delta},\ \vec{\Gamma})$ should have the following property:

$$\boldsymbol{f}(\vec{\Delta}_1, \vec{\Gamma}_1) \geq \boldsymbol{f}(\vec{\Delta}_2, \vec{\Gamma}_2) \text{ if } \vec{\Delta}_1 \supseteq \vec{\Delta}_2 \text{ and } \vec{\Gamma}_1 \supseteq \vec{\Gamma}_2 \tag{5.7}$$

Equation 5.7 can be understood as follows: whenever a test is modified (and committed), the TWD adherence of the project will be maintained or improved compared to its previous state, depending on how close the corresponding production code was committed with respect to the test as well as the method coverage contribution of the added test. This implies that closer commit time proximity and higher method coverage between the creation of tests and the corresponding production code modifications indicate a higher degree of TWD adherence. When the creation of tests and the associated code changes occur in close succession and are better synchronized, it suggests a more rigorous application of TWD. Time proximity refers to the closeness in the commit timestamps between the creation of a test and the corresponding production code modifications, while synchronisation means that the test and the production code changes are kept in sync throughout the development cycle.

The following definition for $\boldsymbol{f}(\vec{\Delta},\ \vec{\Gamma})$ is proposed:

$$
\begin{aligned}
\boldsymbol{f}(\vec{\Delta}, \vec{\Gamma}) &= \sum_{i=1}^{|\vec{\Delta}|} \boldsymbol{g}(\vec{\delta}_i, \vec{\zeta}_i) \\
&= \sum_{i=1}^{|\vec{\Delta}|} \sum_{j=1}^{|\vec{\delta}_i|} \zeta_{ij}\ \boldsymbol{h}(\delta_{ij}) \\
&= \sum_{i=1}^{|\vec{\Delta}|} \sum_{j=1}^{|\vec{\delta}_i|} \zeta_{ij}\ e^{-\delta_{ij}}
\end{aligned}
\tag{5.8}
$$

Where $|\vec{\Delta}|$ is the number of vectors in the matrix $\vec{\Delta}$, and $|\vec{\Gamma}|$ is the number of vectors in the matrix $\vec{\Gamma}$. The inner summation corresponds to the TWD adherence of a single test file that has been modified, created, and committed, while the outer summation corresponds to the TWD adherence of all the modified or created test files in a project throughout its development. For similar reasons as with $\boldsymbol{g}$, $\boldsymbol{f}$ is also monotonically increasing. This property is useful for absolute TWD adherence since any number of software projects can be compared to one another (this is discussed in Section 5.3). Expression 5.9 below demonstrates this property mathematically.

$$
\boldsymbol{f} \in [0, \infty) \tag{5.9}
$$

A suitable definition for the function $\boldsymbol{f}$ that evaluates the TWD adherence τ of a project given a matrix of time deltas $\vec{\Delta}$ and a matrix of method coverage values $\vec{\Gamma}$ has thus been provided:

$$
\begin{aligned}
\tau &\equiv \boldsymbol{f}(\vec{\Delta}, \vec{\Gamma}) \\
&= \sum_{i=1}^{|\vec{\Delta}|} \sum_{j=1}^{|\vec{\delta}_i|} \zeta_{ij}\ e^{-\delta_{ij}}
\end{aligned}
\tag{5.10}
$$

It is important to note that Equation 5.10 is independent of the number of commits in a project; it depends only on the number of added or modified assertions, the time difference between each assertion and the production methods that are tested, and the method coverage of each assertion in relation to the tested production class. Proceeding with the example of the Shapes project: Recall that there are five commits, but only commit number 2 and commit number 5 contained changes to a single test file. Hence, $|\vec{\Delta}|$ equals 2. The addition of the test file in commit number 2 contributed 100% of the total method coverage of the production file since

constructors are ignored during method coverage analysis. Similarly, the changes to the test file in commit number 5 contributed 100% of the total method coverage of `Circle.java`. Hence, $\zeta_{11} = 1$ and $\zeta_{21} = 1$. Moreover, recall that a single assertion was added in both cases. Hence, $|\vec{\delta}_1|$ equals 1 and $|\vec{\delta}_2|$ equals 1. In addition, from Figure 5.4, the time deltas δ_{11} and δ_{21} are 76 s and 625 s, respectively. Converted to days, the values are 0.00087963 and 0.0072338, respectively. Hence, the absolute TWD adherence of the Shapes project is evaluated using Equation 5.10 as follows:

$$\begin{aligned}
\tau &= \sum_{i=1}^{|\vec{\Delta}|} \sum_{j=1}^{|\vec{\delta}_i|} \zeta_{ij} e^{-\delta_{ij}} \\
&= \sum_{i=1}^{2} \sum_{j=1}^{|\vec{\delta}_i|} \zeta_{ij}\ e^{-\delta_{ij}} \\
&= \sum_{j=1}^{|\vec{\delta}_1|} \zeta_{1j}\ e^{-\delta_{1j}} + \sum_{j=1}^{|\vec{\delta}_2|} \zeta_{2j}\ e^{-\delta_{2j}} \\
&= \sum_{j=1}^{1} \zeta_{1j}\ e^{-\delta_{1j}} + \sum_{j=1}^{1} \zeta_{2j}\ e^{-\delta_{2j}} \\
&= \zeta_{11}\ e^{-\delta_{11}} + \zeta_{21}\ e^{-\delta_{21}} \\
&= 1 \cdot e^{-0.00087963} + 1 \cdot e^{-0.0072338} \\
&= 1.992
\end{aligned} \tag{5.11}$$

Recall that the range of absolute TWD adherence is infinite, as expressed by Expression 5.9. Hence, it is possible for the Shapes project to obtain a higher value than this, depending on the future commit behaviour evident in the Shapes project.

To gain an appreciation of the utility of this function, suppose the Shapes project was developed such that tests and production code changes were committed at the same time. In this case, δ_{11} and δ_{21} would both equal 0. Hence, the absolute TWD adherence of the project would be calculated as follows by replacing the relevant variables in the solution in 5.11:

$$\begin{aligned}
\tau &= \zeta_{11}\ e^{-\delta_{11}} + \zeta_{21}\ e^{-\delta_{21}} \\
&= 1 \cdot e^{-(0)} + 1 \cdot e^{-(0)} \\
&= 2
\end{aligned} \tag{5.12}$$

Notice that the function assigns a greater value to projects that demonstrate better

adherence to TWD, which is the desired outcome. Note that the value 2 is the theoretical maximum for two assertion-production pairs that achieve 100% of the method coverage in their respective production files. Again, suppose that the Shapes project was developed so that the test code was committed a week later than the production code. In this case, δ_{11} and δ_{21} would both equal 7 days. Therefore, the absolute TWD adherence of the Shapes project would be calculated as follows:

$$\begin{aligned} \tau &= \zeta_{11} \cdot e^{-\delta_{11}} + \zeta_{21} \cdot e^{-\delta_{21}} \\ &= 1 \cdot e^{-(7)} + 1 \cdot e^{-(7)} \\ &= 0.002 \end{aligned} \tag{5.13}$$

Notice that the function assigns a smaller value to projects that demonstrate inadequate application of TWD. In addition, suppose that the Shapes project did not have any tests at all. In this case, $|\vec{\Delta}| = 0$ since no assertions have been added or modified. Therefore, the absolute TWD adherence of the Shapes project would be calculated as follows:

$$\begin{aligned} \tau &= \sum_{i=1}^{|\vec{\Delta}|} \sum_{j=1}^{|\vec{\delta}_i|} \zeta_{ij} \; e^{-\delta_{ij}} \\ &= \sum_{i=1}^{0} \sum_{j=1}^{|\vec{\delta}_i|} \zeta_{ij} \; e^{-\delta_{ij}} \\ &= 0 \end{aligned} \tag{5.14}$$

Notice that the function assigns the minimum TWD adherence value of zero to projects that display no application of testing whatsoever: The value 0 is the theoretical minimum for this function. Lastly, suppose that `Circle.java` was committed with two function implementations in the first commit (`getCircumference` and `getArea`) instead of only one, and suppose that `Rectangle.java` was also committed with two function implementations in the third commit (`getPerimeter` and `getArea`) instead of only one. In this case, the test file `ShapesGeometryTest.java` would cover only 50% of the production methods in `Circle.java`, namely `getCircumference`. Likewise, `ShapesGeometryTest.java` would cover only 50% of the production methods in `Rectangle.java`, namely `getPerimeter`. Hence, $\zeta_{11} = 0.5$ and $\zeta_{21} = 0.5$. Keeping the same time deltas as in 5.11, the absolute TWD adherence in this case would be calculated as follows:

$$\begin{aligned}\tau &= \zeta_{11}\, e^{-\delta_{11}} + \zeta_{21}\, e^{-\delta_{21}} \\ &= 0.5 \cdot e^{-0.00087963} + 0.5 \cdot e^{-0.0072338} \\ &= 0.996\end{aligned} \tag{5.15}$$

Notice that the absolute TWD adherence is smaller in the case where production classes are only partially covered (0.966) compared to the case where the production classes are covered completely (1.992), even though both cases have the same time deltas. Therefore, the function assigns a greater TWD adherence value to projects that achieve higher method coverage than those that achieve less coverage, even tests are committed with the same frequency.

5.3 Relative TWD Adherence

On its own, absolute TWD adherence is biased towards larger projects since they are simply more likely to have more assertions than smaller projects. Hence, there is not much utility in considering the absolute TWD adherence of a project in isolation. However, controlled studies often compare and contrast an independent variable (e.g., code quality) of more than one software project after controlling for the dependent variable (e.g., the extent to which TDD was applied during the development of the projects). Hence, there is a need to compare the extent to which projects adhered to TWD during their development. For this reason, it is necessary to devise a strategy to benchmark the TWD adherence of multiple projects while ensuring a fair comparison between them. The value obtained upon transforming a project's absolute TWD adherence for the purpose of benchmarking is henceforth referred to as its *relative TWD adherence*, which represents how closely the development of a software project adhered to TWD in relation to another project with a similar number of assertions.

5.3.1 K-means Clustering

Recall that Equation (5.10) depends on the absolute number of added or modified assertions throughout the development of a project: For every additional assertion that is added or modified, absolute TWD adherence strictly increases. It does not depend on the size of a project or on the number of commits created throughout its development. If the absolute TWD adherence values of projects are compared directly without any normalisation or clustering, projects with fewer assertions would

be disadvantaged. Hence, it is desirable to cluster projects with a similar number of assertions. For this reason, the number of assertions in a project is chosen as the variable with which to cluster projects into distinct groups using K-means clustering [67].

Determining the Optimal Number of Clusters

Before the K-means algorithm can be applied to group projects according to the number of added or modified assertions, the optimal number of clusters is first determined using the elbow method [68]. The elbow method is a common technique used in machine learning to determine the optimal number of clusters for K-means clustering. The method is called "elbow" because the plot of the number of clusters versus the within-cluster sum of squares (WCSS) resembles an arm with a clear elbow point.

The elbow method works by first fitting the K-means clustering model to the data for a range of cluster values. Then, for each model, the WCSS is calculated, which represents the sum of the squared distances between the observations and their cluster centroid. A plot is then created with the number of clusters on the x-axis and the WCSS on the y-axis. The optimal number of clusters corresponds to the point at which the WCSS begins to level off and the improvement in clustering performance starts to diminish. This indicates that the additional clusters do not explain much more of the variance in the data.

As stated, the purpose of clustering is to group projects together based on their total number of assertions in order to perform a fair TWD adherence comparison.

5.3.2 Evaluating Relative TWD Adherence

The relative TWD adherence τ_r of a project with n assertions is evaluated from a cluster of projects using the equations below,

$$\tau' = \left(\frac{1}{\tau_p} \cdot \frac{n_p}{n}\right) \cdot \tau \tag{5.16}$$

$$\tau_r = \frac{\tau'}{\tau'_{max}} \tag{5.17}$$

Where τ' is the project's within-cluster TWD adherence in relation to the pivot project, τ is the project's absolute TWD adherence, τ_p is the highest absolute TWD adherence in the cluster, n_p is the number of assertions in the project with the maximum absolute TWD adherence in the cluster, and τ'_{max} is the highest within-cluster TWD adherence. Equation 5.16 is applied to each project via the following steps:

- Get the cluster of projects that a given project is a part of.
- Get the number of assertions n_p and the absolute TWD adherence τ_p of the project with the maximum absolute TWD adherence in the cluster. This project is referred to as the *pivot* project.
- Transform the absolute TWD adherence τ of the given project to the within-cluster TWD adherence τ' using Equation (5.16).

Finally, the relative TWD adherence τ_r of every project is obtained using Equation (5.17). Consider the following example that demonstrates how the relative TWD adherence of each project in the same cluster is obtained. Assume that there are five projects with the following number of assertions and absolute TWD adherence:

Table 5.3: Example of obtaining relative TWD adherence.

Project	n	τ
A	80	503
B	95	334
C	75	600
D	63	254
E	60	550

Upon inspecting Table 5.3, it is clear that Project C has the highest absolute TWD adherence. This makes it the pivot project with regard to obtaining the within-cluster TWD adherence of all the projects. Hence, $n_p = 75$ and $\tau_p = 600$. Notice that the ratio of the number of assertions in the pivot project (n_p) to the number of assertions in a given project (n) is used in Equation (5.16) to obtain the within-cluster TWD adherence of each project τ'. The significance of this is to adjust for the difference in the number of assertions between projects in the same cluster; it is assumed that projects with fewer assertions than the pivot project would have an amplified absolute TWD adherence if they had the same number of assertions as the pivot

project, and projects with more assertions than the pivot project would have an attenuated absolute TWD adherence if they had the same number of assertions as the pivot project. Equation (5.16) is used to obtain the fourth column in Table 5.4 below, which represents the within-cluster TWD adherence. The within-cluster TWD adherence τ' of a project essentially represents how much better or how much worse a project's absolute TWD adherence *would have been* in relation to the pivot project if the same number of assertions were used (thus controlling the impact of the number of assertions on the absolute TWD adherence, further assisting with fair comparison). It is used to reduce the effect that the number of assertions has on the comparison of absolute TWD adherence between projects.

Table 5.4: Calculation of the within-cluster TWD adherence

Project	n	τ	τ'
A	80	503	0.786
B	95	334	0.439
C	75	600	1
D	63	254	0.504
E	60	550	1.146

Notice that Project E has the highest within-cluster TWD adherence τ', as shown in Table 5.4. It has essentially become the new pivot project; if Project C (the original pivot project) and Project E had the same number of assertions, Project E would likely outperform Project C in terms of absolute TWD adherence (assuming that the time deltas and method coverage trend in Project E stay the same).

Since the goal is to transform the absolute TWD adherence of a project to a value in the range of 0 to 1 that represents its relative TWD adherence, a normalization of the within-cluster TWD adherence values is required, which is a simple division of all the within-cluster TWD adherence values by the maximum value. Hence, Equation (5.17) is used to obtain the fifth column in Table 5.5 below, which represents the relative TWD adherence of each project.

There are other popular normalisation methods that could have been used, namely min-max normalisation and sigmoid normalisation [69]. The normalisation method chosen in this study, namely Equation (5.17), is compared to these popular methods, which have the following expressions:

Table 5.5: Calculating the relative TWD adherence of projects.

Project	n	τ	τ'	τ_r
A	80	503	0.786	0.686
B	95	334	0.439	0.383
C	75	600	1	0.873
D	63	254	0.504	0.434
E	60	550	1.146	1

$$\tau_{r_{min-max}} = \frac{\tau' - \tau'_{min}}{\tau'_{max} - \tau'_{min}} \tag{5.18}$$

$$\tau_{r_{sigmoid}} = \frac{1}{1 + e^{-\tau'}} \tag{5.19}$$

Consider the output of each of these normalisation techniques on the same example, as shown in Table 5.6.

Table 5.6: Comparison with other normalisation techniques.

Project	n	τ	τ'	τ_r	$\tau_{r_{min-max}}$	$\tau_{r_{sigmoid}}$
A	80	503	0.786	0.686	0.491	0.687
B	95	334	0.439	0.383	0	0.608
C	75	600	1	0.873	0.793	0.731
D	63	254	0.504	0.434	0.092	0.623
E	60	550	1.146	1	1	0.759

Notice that the min-max normalisation assigned a relative TWD adherence value of 0 to Project B, even though Project B achieved a substantial absolute TWD adherence score of 334. This suggests that the min-max normalisation technique does not accurately represent the TWD adherence of projects in relation to one another. Moreover, notice that the sigmoid normalisation Equation 5.19 did not assign a value of 1 to any project at all, and the relative TWD adherence values all lie within the range [0.6, 0.8]. This is not desirable since it computes a similar relative TWD adherence for projects that may vary greatly in absolute TWD adherence. Hence, the sigmoid normalisation technique, too, does not accurately represent the TWD adherence of projects in relation to one another. Therefore, Equation (5.17) was chosen to evaluate the relative TWD adherence of projects.

5.4 Development Step-Size

Proponents of TDD claim that TDD should be practiced by making small, iterative changes to production code, where each change is accompanied by a test. The previous sections dealt with detecting the use of TWD from time deltas and method coverage. This section focuses on determining the median size of production code changes covered by an added or modified assertion, which is useful to answer **RQ3**, namely: "Do developers work in smaller steps when practicing TDD?"

5.4.1 Production Method Churn

Production method churn refers to how much or how little a production method changes during the development of a software project. The subsystem responsible for measuring production method churn depends on the data structure discussed in Subsection 5.2.2. The system that evaluates production method churn converts the data structure to a list of production method churn values. Production method churn in the context of this document is defined as the sum of the number of added lines and the number of deleted lines in a production method that is covered by a single assertion. Refer to Figure 5.3 for the following discussion of the `modifications` field: The `modifications` field stores the line number on which an addition or deletion occurred as well as the literal lines of code that were added or deleted.

Consider the fourth commit of the Shapes project as outlined in Subsection 5.2.2: there is only one `ProductionFile` object that exists in this commit. The value stored in its `name` field is "Circle". The `Circle` class's `getCircumference` method was modified in a single line to get from Listing 5.2 to 5.4. Hence, the `modifications` dictionary should reference only a single added line and a single deleted line. Starting with the deleted line, the actual line number is `11` of Listing 5.2. As for the added line, the actual line number is also 11 in Listing 5.4. Listing A.3 in Appendix A shows a technical representation of the `modifications` field. Proceeding with the example of the Shapes project, consider Figure 5.9 below.

Notice that Figure 5.9 is similar to Figure 5.4 except that Figure 5.9 now includes method churn data. As outlined before, PM_1 refers to the production method `getPerimeter` of the `Rectangle` class, and TA_1 refers to the test assertion in `testPerimeter` of the `ShapesGeometryTest` class. PM_2 refers to the intentionally incorrect production method `getCircumference` of the `Circle` class, while PM_2' refers to the corrected production method `getCircumference` of the `Circle` class. Lastly,

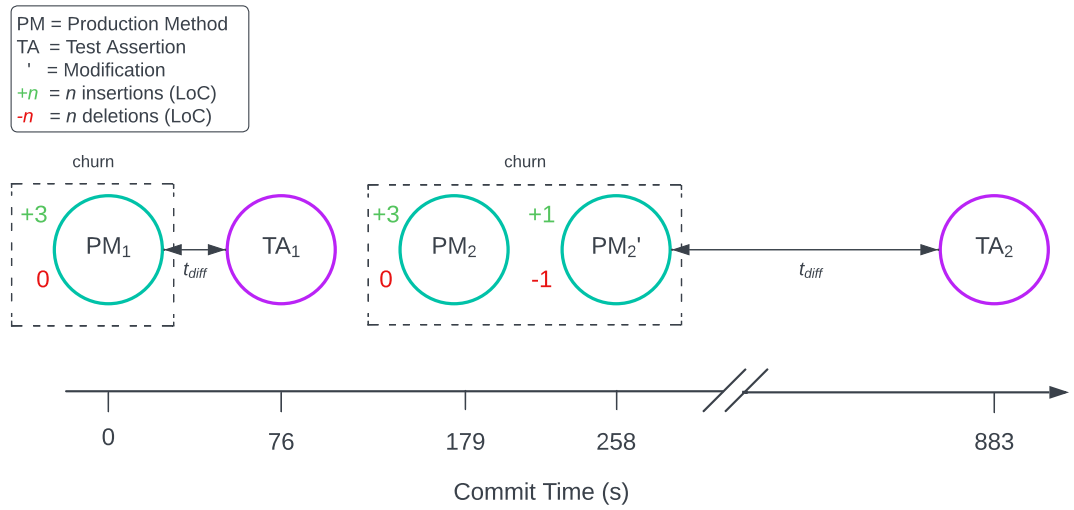


Figure 5.9: A representation of the time deltas and method churn of the Shapes project.

TA_2 refers to the test assertion in `testCircumference` of the `ShapesGeometryTest` class.

The total production method churn equals the sum of the individual production method churn values up to the point at which production method changes are tested by an assertion. Consider Figure 5.9: PM_1 was newly created; thereafter, an assertion (TA_1) was created in `ShapesGeometryTest.java`: the addition of PM_1 corresponds to a single instance of production method churn. The total production method churn (boxed into the first dashed rectangle) equals the individual production method churn of PM_1 in this case since PM_1 is immediately followed by a test. Three lines of code were added, and no lines were deleted during the initial creation of PM_1. Hence, the production method churn in this case equals 3.

Consider the dashed rectangular box on the right in Figure 5.9: this represents the second instance of production method churn. PM_2 (having a total size of 3 LoC) was newly created in the third commit; this means that three lines were added and zero lines were deleted in the creation of PM_2. Hence, the production method churn associated with PM_2 equals 3. However, PM_2 was modified in the fourth commit, resulting in PM_2'. This modification involved changing a single line of PM_2. This means that one line was added and one line was deleted. Hence, the production method churn associated with PM_2' is 2. Since these changes are tested in the fifth commit (TA_2), the total production method churn (corresponding to the changes in the dashed rectangle on the right) is the sum of the individual production method churns of PM_2 and PM_2', which equals 5.

Upon analysis of the commit history of the Shapes project in Figure 5.9, it is apparent that there are two instances of production method churn (since a total of *two* series of production method changes were followed by corresponding tests). The two values are 3 and 5. Table 5.7 below shows the output of the system that evaluates production method churn, while Table 5.8 shows the legend for the symbols used in the diagrams in Figure 5.10 and Figure 5.11. Figure A.4 in Appendix A shows a technical representation of the output of the system that evaluates production method churn after it analyses the Shapes project.

Table 5.7: Output of the time delta subsystem

Production Method	**Test Method**	**Individual Method Churn**	**Method Churn**
PM1	TA1	3	3
PM2	TA2	3	
PM2'	TA2	2	5

Figure 5.10 and Figure 5.11 below demonstrate how the system that evaluates production method churn collects data from a given Java project.

Table 5.8: The legend for the symbols used in Figure 5.10 and Figure 5.11

Symbol	**Description**
TF	The test file currently being analyzed
C_1	The commit that contains changes to TF
PF_T	The production file that is covered by TF
PM_T	The method in PF_T that is covered by TF
m	The overall method churn
C_2	A commit that was done before C_1 (C_2 can also be the same as C_1)
PF	A production file whose changes were committed in C_2
PM_{list}	The list of all methods in PF that were modified and included in C_2

5.4.2 The Relationship Between Production Method Churn and TWD Adherence

To answer **RQ3**, Spearman's rank order correlation coefficient and p-value [70] of the median production method churn and the relative TWD adherence are evaluated for each cluster of projects. The median is preferred to the mean because the median is a

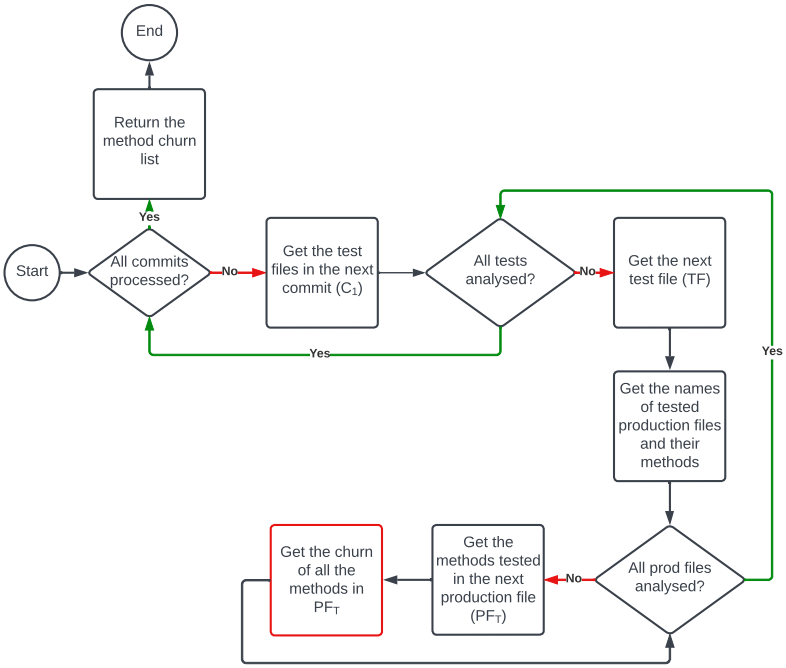


Figure 5.10: Calculating the churn for all the tested production methods of a project.

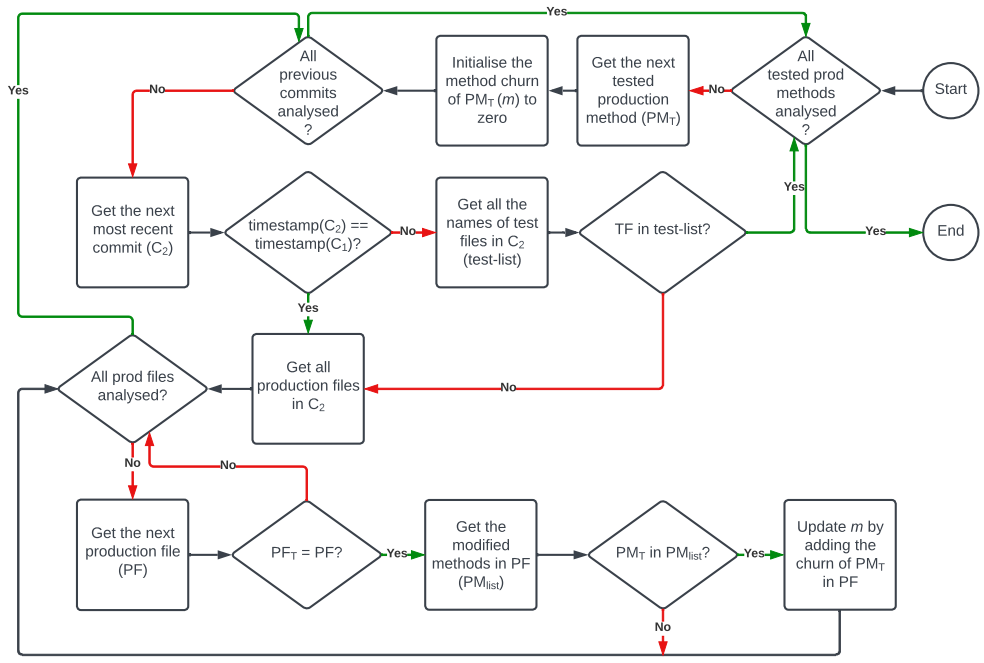


Figure 5.11: Getting the churn for a single production file's tested methods.

more robust measure of central tendency that is less sensitive to outliers in the data, which translates to a better representation of the overall production method churn of a given project [71]. If developers work in smaller steps, then for each cluster, one would expect to see a strong and negative correlation between relative TWD adherence and median production method churn.

5.5 Conclusion

Definitions for the mathematical functions that evaluate the absolute and relative TWD adherence of projects have been presented, and a strategy to determine the median development step size has been devised. The mathematical function that evaluates the absolute TWD adherence of a project takes into account the time deltas between added or modified assertions and the production methods they test, as well as the method coverage of each added or modified assertion on the production class it tests. The relative TWD adherence of a project is obtained by clustering projects according to the number of assertions they contain and by transforming and normalising its absolute TWD adherence depending on the projects in its cluster. The median development step size is found by taking the median of the production method churn of the series of production code changes that are covered by an added or modified assertion throughout the project.

Chapter 6

Results

This chapter presents the results of the study and the discussion of the answers to the research questions. The statement-based matching algorithm has a weighted F1-score of 0.771 with a 95% confidence interval of [0.741, 0.786], while the filename-based matching algorithm has a weighted F1-score of 0.218 with a 95% confidence interval of [0.193, 0.246]. The study found a strong, positive, and statistically significant correlation between the system's method coverage and Codecov's code coverage for eighteen Java projects (r_s = 0.744, p-value = 0.0003). In addition, the study found a strong, positive, and statistically significant correlation between the relative TWD adherence and Codecov's code coverage for eleven Java projects (r_s = 0.723, p-value = 0.012). It was concluded that there is no correlation between relative TWD adherence and the median production method churn.

6.1 File Matching Algorithms

The performance assessment system described in Section 4 was used to answer **RQ1**, namely, "Can test code be matched more accurately to production code using a statement-level approach?" A total of 500 tests from sixteen Java projects were used in the performance assessment. These projects ranged in size from 5,115 lines of code (LoC) to 378,167 LoC and encompassed a variety of types, including frameworks, libraries, and tools for data processing, testing, and web services. Further details about these projects can be found in Appendix B. Figure 6.1 shows the performance of the two matching algorithms on the 500 test files. The numbers at the top of each bar in Figure 6.1 indicate the number of test files sampled for each project.

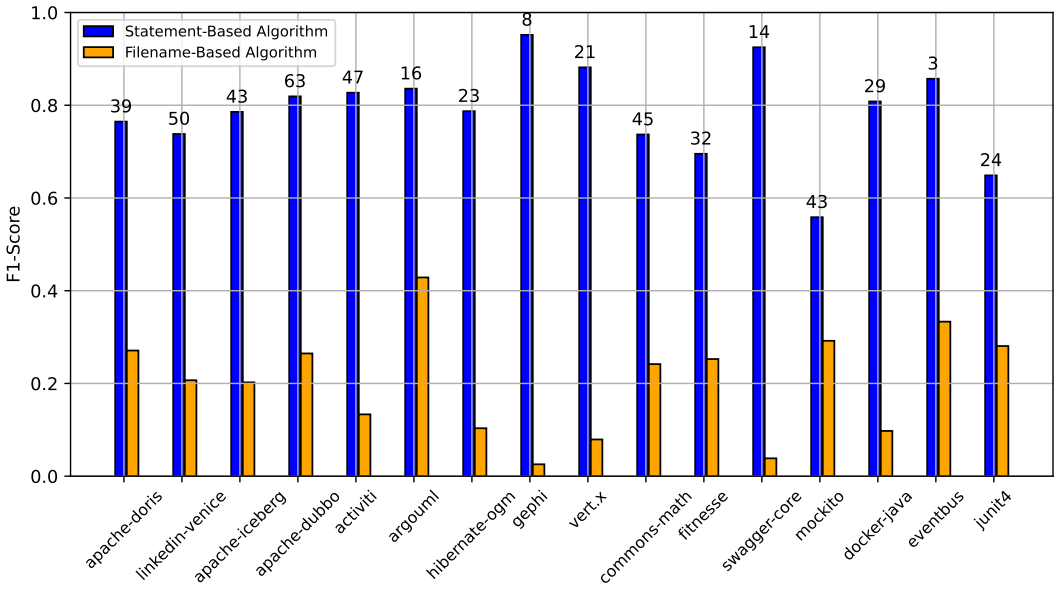


Figure 6.1: F1-scores for the statement-based and filename-based matching algorithms given n test samples from different Java projects.

Table 6.1 below shows the weighted performance metrics for the file matching algorithms on the 16 Java projects.

Table 6.1: Weighted performance metrics for the matching algorithms

Matching Algorithm	**Weighted Precision**	**Weighted Recall**	**Weighted F1-Score**	**Weighted Accuracy**
Statement-Based	0.842	0.712	0.771	0.783
Filename-Based	0.495	0.140	0.218	0.380
Performance Improvement	+0.347	+0.572	+0.553	+0.403

From Figure 6.1 and Table 6.1, it is clear that the statement-based matching algorithm outperforms the filename-based matching algorithm. The statement-based matching algorithm's weighted precision (84.2%) is 34.7 percentage points higher than the filename-based matching algorithm's weighted precision (49.5%). Practically, this means that the statement-based matching algorithm has a 34.7% higher likelihood of correctly identifying tested production files while minimising the inclusion of untested production files. Similarly, the statement-based matching algorithm's weighted recall (71.2%) exceeds the filename-based matching algorithm's weighted recall (14.0%) by 57.2 percentage points. This indicates that the statement-based matching algorithm has a 57.2% higher likelihood of capturing tested production

files, significantly reducing the chances of missing such files compared to the filename-based matching algorithm. Regarding the F1-score, the statement-based matching algorithm's weighted F1-score (77.1%) is 55.3 percentage points higher than the filename-based matching algorithm's weighted F1-score (21.8%). This suggests that the statement-based matching algorithm has a 55.3% better balance between precision and recall, making it more effective at accurately identifying tested production files while minimising both false positives and false negatives. In practical terms, these percentage differences imply that the statement-based matching algorithm is far more effective at matching test files to production files, resulting in more accurate and reliable matches.

Consider the performance of the algorithms on Gephi in Figure 6.1: The extreme performance difference between the algorithms may be due to the sampled tests being integration tests, as each test covers an average of ten production files; hence, the chosen test filename does not match any one particular production file. Since the statement-based matching algorithm analyses source code instead of filenames, its performance is not affected in the same way that the filename-based matching algorithm is. Notice the relatively poor performance of the statement-based matching algorithm with respect to Mockito. Mockito, as a mocking framework, is designed to simplify the process of creating mocks and stubs for unit testing [72], making it highly likely that its own tests rely on the features it provides, such as dependency injection, mocking, and stubbing. Mockito's own tests primarily consist of unit tests that focus on testing individual components in isolation, using mocks and stubs to replace external dependencies. Although some integration tests might be employed to ensure the proper functioning of components when interacting with each other, the primary focus is on unit testing with mocks and stubs, allowing the Mockito development team to ensure the framework's reliability and correctness while maintaining a fast and efficient testing process. Since the statement-based matching algorithm's method of operation does not detect the use of mocks or stubs, this may explain its relatively poor performance with respect to Mockito.

Figure 6.2 below shows the difference in the upper bound and lower bound of the 95% confidence interval of the weighted F1-score for both matching algorithms. Table 6.2 shows the upper bound and lower bound weighted F1-score values for $n = 10$ and $n = 500$.

Pseudo-sample sizes ranging from 1 to 500 were considered for the plot. For a pseudo-sample size of 10, the interval for the statement-based matching algorithm is [0.618, 0.929], with a large difference of 0.311. This means that there is a 95% chance

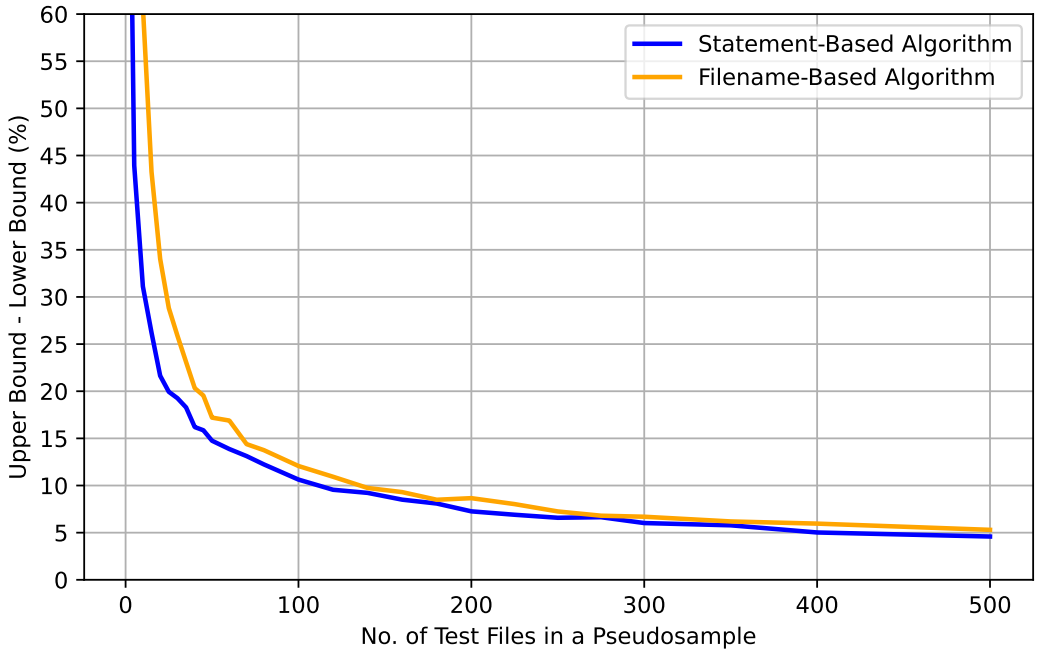


Figure 6.2: Difference between the upper bound and lower bound for the 95% confidence interval of the weighted F1-score for the matching algorithms

that the range [0.618, 0.929] contains the true performance of the statement-based matching algorithm when using a sample size of 10. The interval for the filename-based matching algorithm's weighted F1-score is [0.170, 0.778], with a large difference of 0.608. Similarly, this means that there is a 95% chance that the range [0.170, 0.778] contains the true performance of the filename-based matching algorithm when using a sample size of 10. Notice the large confidence interval for both matching algorithms when using a sample size of 10. The practical meaning of this is that using a small number of test files offers too wide a range for the confidence in the true performance of the matching algorithms, making it unreliable for drawing conclusions.

However, at a pseudo-sample size of 500, the 95% confidence interval of the statement-based matching algorithm is [0.741, 0.786] and that of the filename-based matching algorithm is [0.193, 0.246]. Regardless of the specific sample size used, the statement-based matching algorithm is more likely to provide more accurate matches between test files and production files due to its more favourable weighted F1-score confidence interval. Moreover, the differences between the respective lower bounds (0.548) and the upper bounds (0.540) of the matching algorithms indicate a substantial performance gap in favour of the statement-based matching algorithm. The consistent superiority of the statement-based matching algorithm's confidence intervals suggests that it is more favourable to use for matching test files to production files.

Table 6.2: Upper bound and lower bound values for the weighted F1-score for the matching algorithms for 10 test samples and 500 test samples.

Matching Algorithm	$\bar{\psi}_{upper}$ **($n = 10$)**	$\bar{\psi}_{lower}$ **($n = 10$)**	$\bar{\psi}_{upper}$ **($n = 500$)**	$\bar{\psi}_{lower}$ **($n = 500$)**
Statement-Based	0.929	0.618	0.786	0.741
Filename-Based	0.778	0.170	0.246	0.193

6.2 TWD Adherence

A total of 100 Java projects were cloned from GitHub. Table 6.3 shows the selection criteria of the projects. The perils that were considered during project selection in this study were outlined by Kalliamvakou et al. [73]. Further details about these projects can be found in Appendix B. Of the 100 Java projects, eighteen were used to validate the method coverage system by comparing the method coverage calculated by the system for each project to each project's actual test coverage as calculated by Codecov, a popular dynamic code coverage tool [74]. The names of the eighteen projects and the reasons for their selection are provided in Sub-section 6.3.1. In addition, eleven projects were used to validate the TWD adherence system because they were part of the survey done by Hilton et al. [75] and Zaidman et al. [18], thus allowing for a direct comparison of the results obtained through the findings from this study. The names of the projects as well as the reasons for their selection are provided in Sub-section 6.3.2. The projects were processed using the statement-based matching algorithm, the absolute TWD adherence evaluator, and the relative TWD adherence evaluator in order to answer **RQ2**, namely: "How can such a matching algorithm be used to measure the TDD adherence of a project?"

Table 6.3: Project Selection Criteria.

Perils	**Selection Criteria**
Most projects have very few commits.	Select at least 10 projects per cluster, each with 2,000 - 35,000 commits.
Most projects are inactive.	Select projects with at least 100 stars, with the last commit within the last year.
A large portion of repositories are not for software development.	Select projects using Java as the main programming language.
Two-thirds of projects are personal.	Select projects with more than one committer.

6.2.1 Ranking Projects by TWD Adherence

The number of assertions in the 100 Java projects that were sampled ranged from 31 to 17659. Firstly, the absolute TWD adherence values of each project were obtained according to Equation (5.10). Thereafter, the optimal number of clusters was determined using the elbow method in order to obtain the relative TDD adherence values. Figure 6.3 below shows the elbow plot for the given data.

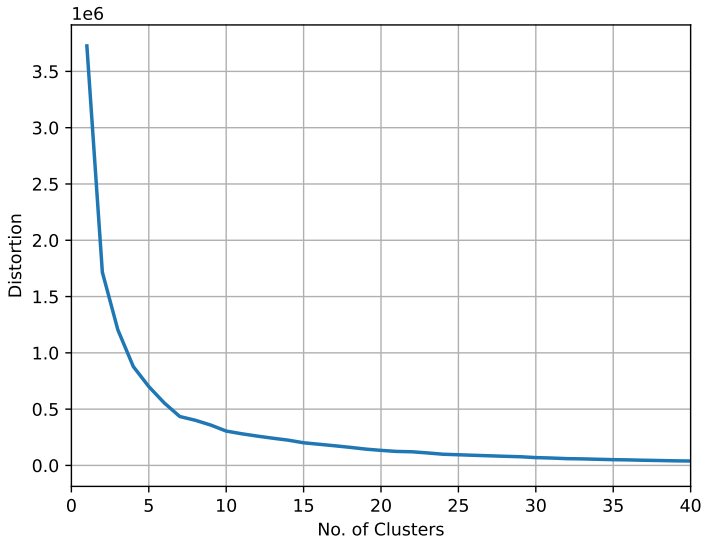


Figure 6.3: The elbow plot of distortion vs no. of clusters for the given data.

By close inspection of Figure 6.3, it can be observed that the graph starts to flatten at a cluster number of 7. Hence, a cluster number of 7 is chosen as the most optimal for this dataset. Lastly, the K-means algorithm was used to cluster projects into 7 groups according to the number of unique assertions in each project. The relative TWD adherence for each project within a given cluster was obtained according to Equation (5.17). Figure 6.4 below shows the relative TWD adherence of the 100 projects in 7 clusters.

Consider SIG-Fasten and ArgoUML in the second (red) cluster. Table 6.4, Figure 6.5, and Figure 6.6 below show aspects of both projects that are important for the discussion to follow.

SIG-Fasten has the highest relative TWD adherence value (1), while ArgoUML has the lowest relative TWD adherence value (0.143) in the red cluster. This is corroborated by the results shown in Table 6.4, Figure 6.5, and Figure 6.6: About 27% of added or modified assertions are committed at the same time as the production

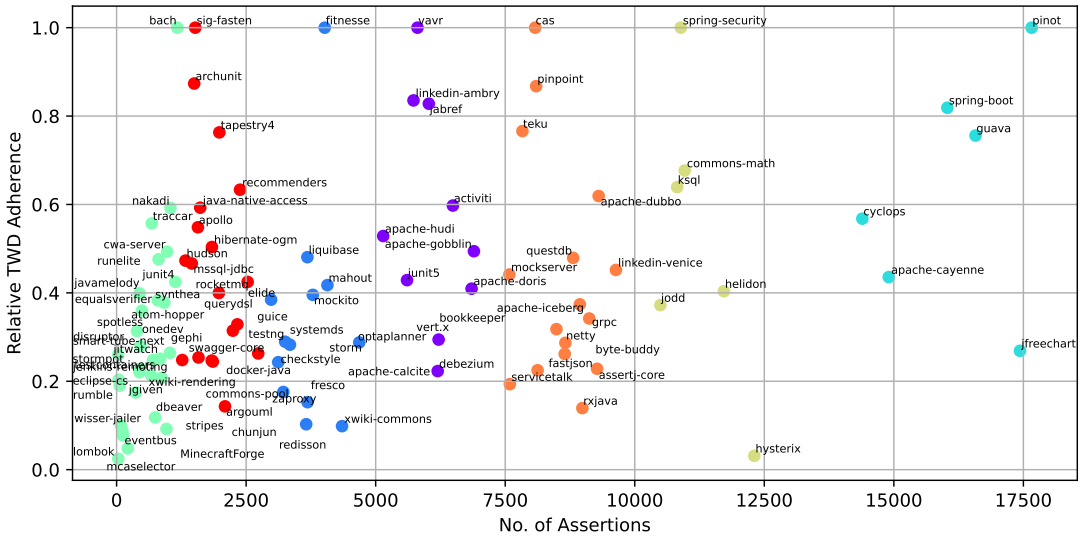


Figure 6.4: The scatter plot of relative TWD adherence for 100 Java projects.

Table 6.4: Aspects of SIG-Fasten and ArgoUML

Project	**Relative TWD Adherence**	**Size (LoC)**	**Unique Assertions**	**Matches**
SIG-Fasten	1	52267	1518	3383
ArgoUML	0.143	335705	2092	2588

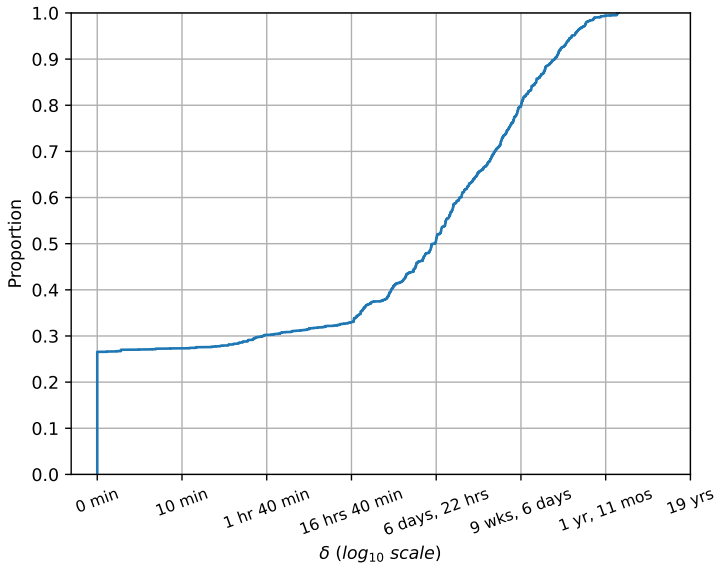


Figure 6.5: A CDF plot of the time deltas between assertions and production methods in SIG-Fasten.

methods they test in SIG-Fasten, compared to only 4% in ArgoUML. In addition, 50% of added or modified assertions are committed within 7 days after their corresponding

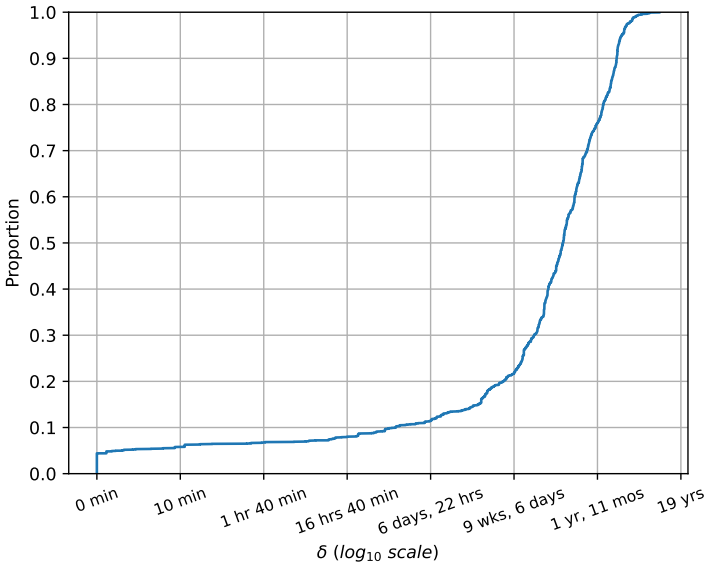


Figure 6.6: A CDF plot of the time deltas between assertions and production methods in ArgoUML.

production methods in SIG-Fasten, but in the case of ArgoUML, 50% of the added or modified assertions are committed within 7 months after their corresponding production methods. Note: The value of 7 months was found by reading the value at the midpoint between "9 wks, 6 days" (10^5 minutes) and "1 yr, 11 mos" (10^6 minutes). The midpoint is equal to $10^{5.5}$, which is equivalent to 316227 minutes or roughly 7 months. This suggests that tests co-evolve with production code in SIG-Fasten since smaller time deltas are observed, which is indicative of TDD [18]. As shown in Table 6.4, SIG-Fasten has a total production code size of 52267 LoC, whereas ArgoUML has a substantially larger total production code size of 335705 LoC. Moreover, while SIG-Fasten and ArgoUML have a comparable number of unique assertions (1518 and 2092, respectively), more modifications to assertions have been made in SIG-Fasten throughout its development (3383) compared to ArgoUML (2588). This suggests that developers of SIG-Fasten refactor and update their tests together with their production code more frequently than developers of ArgoUML, even though SIG-Fasten is much smaller than ArgoUML, which is indicative of TDD [1, 18]. These findings are further corroborated by Zaidman et al. [18], who found that developers of SIG were encouraged to use TDD, while developers in ArgoUML mentioned that using TDD is not a habit [18].

6.3 Development Step Size

The median production method churn for each of the 100 projects in their respective clusters was calculated in order to answer **RQ3**, namely, "Do developers work in smaller steps when practicing TDD?" In particular, the correlation between the median production method churn and the relative TWD adherence of each project in a cluster was evaluated. Table 6.5 below shows the correlation between the median method churn and relative TWD adherence for different ranges of assertions.

Table 6.5: The correlation between the median method churn and relative TWD adherence for various ranges of assertions.

No. of assertions	**Correlation Between Median Method Churn and Relative TWD Adherence**
31 - 1171	r_s = -0.340, p-val: 0.066
1267 - 2733	r_s = -0.140, p-val: 0.568
2982 - 4683	r_s = -0.105, p-val: 0.734
5145 - 6896	r_s = -0.091, p-val: 0.803
7580 - 9637	r_s = -0.044, p-val: 0.873
10495 - 12307	r_s = 0.460, p-val: 0.358
14392 - 17659	r_s = 0.972, p-val: 0.001

As shown in Table 6.5, the range $[14392 - 17659]$ is associated with a statistically significant correlation between the median method churn and relative TWD adherence; the correlation is strong and positive. For the rest of the clusters, however, the correlation is insignificant, which suggests that any correlation that is observed is likely due to chance. Hence, it is concluded that developers do not work in smaller steps when practicing TDD. The reason for this observation may be that there are other confounding variables that are not taken into account when grouping projects, such as the project complexity and the skill level of the developers in the team, to name a few. In other words, whether or not small steps are taken in a project is not solely due to whether or not TDD is practiced by developers of the project; it may also be affected by factors such as the complexity of the project or the work experience of developers in the project.

6.3.1 Validation of the Method Coverage System

The null hypothesis H_0 is that there is no correlation between the method coverage calculated by the system and the test coverage calculated by Codecov for the eighteen projects that were sampled. The Java projects used for validating method coverage are as follows: Apache Dubbo, Apache Gobblin, Apache Hudi, Apollo, Cas, Apache Commons-Math, Apache Commons-Pool, Jabref, Jodd, JUnit5, Linkedin-Ambry, Mockito, MSSQL-JDBC, Apache Pinot, Pinpoint, Apache RocketMQ, Synthea, and Vavr. Further details, including repository links to these projects, can be found in Appendix B. These projects were selected because they were the only ones out of the 100 projects that were available on Codecov with valid test coverage values. The test coverage values from Codecov were found by searching for each project manually on the website by specifying each project's repository on GitHub and then looking for the particular project's name in the search results. For example, the test coverage values for Apache projects were found by placing "apache" in the URL "https://app.codecov.io/gh/" to result in "https://app.codecov.io/gh/apache". The method coverage of each project was calculated by analysing the latest version of each production class in the project and evaluating the percentage of tested methods. Table 6.6 shows the coverage data from Codecov and the system developed in this study for each project. Spearman's rank-order correlation was evaluated to determine the correlation between the actual code coverage and the calculated method coverage. It was found that $r_s = 0.744$ and p-value $= 0.0003$. This suggests a very strong and positive correlation between the code coverage and the calculated method coverage. In addition, since p-value $\ll 0.05$, the correlation is significant, which means that there is less than 5% chance that the correlation is due to chance. Thus, the null hypothesis H_0 is rejected, and it is concluded that the method coverage is highly correlated with the actual code coverage of the project.

However, notice that although the correlation is very strong, positive, and statistically significant, the actual values vary substantially. For instance, the actual code coverage of Vavr is 92.74%, but the calculated method coverage is only 47.43%. A possible reason for this lies in the difference in the actual quantities measured by method coverage and test coverage: dynamic code coverage tools such as Codecov analyse the call stack in much more depth, and they also evaluate statement coverage and branch coverage. As a result, such tools are able to identify the exact lines of code and branches in production code that are tested by a given assertion, which is part of the reason that they take long to process [76]. However, the method coverage tool developed in this study can only analyse up to one level deep in the call stack. This means that it can only identify the public methods covered by an assertion and any

Table 6.6: The correlation between the actual code coverage from Codecov and the method coverage calculated by the system in this study.

Java Projects	**Actual Code Coverage (%)**	**Calculated Method Coverage (%)**
Apache Dubbo	69.51	20.98
Apache Gobblin	47.64	14.29
Apache Hudi	51.49	16.76
Apollo	49.72	16.84
Cas	91.56	19.21
Apache Commons-Math	86.55	35.35
Apache Commons-Pool	83.03	20.98
Jabref	41.95	14.91
Jodd	52.21	16.4
JUnit5	75.81	15.79
LinkedIn Ambry	70.19	37.39
Mockito	85.53	18.21
MSSQL-JDBC	54.52	17.68
Apache Pinot	61.4	13.81
Pinpoint	38.2	10.88
Apache RocketMQ	43.27	17.07
Synthea	77.17	23.17
Vavr	92.74	47.43

other methods that are invoked within the public method in question. Moreover, the method coverage tool does not analyse statement coverage nor branch coverage; it keeps track of *whether or not* a method has been directly or indirectly covered by an assertion. For these reasons, the method coverage tool developed in this study will identify only a fraction of the true coverage achieved in a project. It is important to note that this issue has a limited impact on the TWD adherence system because, in the context of this study, the primary goal of the method coverage evaluator is to identify the ratio of methods in a production class that are covered by a test, not the overall coverage of a project. Nonetheless, the importance of the strong, positive, and statistically significant correlation is to demonstrate that the method coverage tool developed in this study works as expected.

6.3.2 Validation of the TWD Adherence Evaluation System

The null hypothesis H_0 is that there is no correlation between TDD scores obtained from surveys and the relative TWD adherence value generated for a given project. The Java projects used for validation are as follows: Apache Mahout, Apache Commons-Math, FitNesse, Hudson, JUnit4, JUnit5, XWiki-Commons, and XWiki-Rendering (obtained by Hilton [75]), as well as SIG-Fasten, ArgoUML, and Checkstyle (obtained by Zaidman et al. [18]). Further details, including repository links to these projects, can be found in Appendix B. These projects were selected on the basis that developers from the projects were interviewed or asked to complete a survey in two studies, namely the studies conducted by Hilton [75] and Zaidman et al. [18], which provided a means to compare and validate the TWD adherence system using peer-reviewed results. Table 6.7 shows the correlation between the TDD scores obtained from the surveys and the relative TWD adherence values generated experimentally. It is important to note that the survey done in [18] did not explicitly provide a TDD score out of 5; however, it mentioned the following key points:

> "Test driven development [at SIG] is encouraged...," "Changes to the production code [at SIG] may only be committed together with the modified corresponding test code," "...both developers [at Checkstyle] indicate that they try to write a failing test documenting the bug, before fixing it and making the test pass," "Test-driven development [at ArgoUML] is not a habit."

In addition, the authors of [18] discovered that Checkstyle appeared to have a more phased application of TDD upon analysing the project. Lastly, since the industrial software project analysed in [18] was not made available, the most popular open-source software project from SIG, namely Fasten [77], was cloned instead. Hence, Fasten was assigned a score of 5, Checkstyle was assigned a score of 3, and ArgoUML was assigned a score of 1.

Upon determining the Spearman rank order correlation between the TDD scores from research and the relative TWD values obtained experimentally, the following was discovered: $r_s = 0.723$ and p-value $= 0.012$. The coefficient r_s is strong and positive. This suggests that a project that is assigned a higher TDD score by manual analysis is also assigned a higher relative TWD value by the system. Conversely, a project that is assigned a lower TDD score by manual analysis is also assigned a lower relative TWD value by the system. In addition, the p-value is less than 0.05, which

Table 6.7: TDD scores from surveys and the relative TWD adherence values from the experiment.

Project	**Survey**	**Experiment**
Apache-Mahout	4	0.418
Commons-Math	4	0.394
FitNesse	5	1.000
Hudson	5	0.334
JUnit4	5	0.613
JUnit5	5	0.261
XWiki-Commons	2	0.098
XWiki-Rendering	2	0.308
SIG-Fasten	5*	0.706
ArgoUML	1*	0.101
Checkstyle	3*	0.186

means that the correlation is statistically significant; there is less than 5% likelihood that the correlation observed between the TDD scores and the corresponding relative TWD values is due to chance. Hence, there is sufficient evidence to reject the null hypothesis, and thus it is concluded that the relative TWD adherence is highly correlated with the TDD scores obtained from surveys.

6.4 Processing Time

Ten Java projects were sampled at every 10th percentile of unique assertions from the 100 Java projects that were processed in Subsection 6.2.1 to demonstrate how long it takes for projects to be processed. For each of the ten projects, all the scripts that were written for this study were executed, including the statement-based test-to-code traceability tool and the TDD adherence tool, among others. Table 6.8 below shows the time taken to process each of the projects, and Table 6.9 shows the specifications of the computer used to process the data. Further details on the the ten sampled projects that appear in Table 6.8 can be found in Appendix B.

It takes between seven minutes and nine hours to process a single project, with most projects taking less than two-and-a-half hours to be fully processed. The processing time of the statement-based matching algorithm is longer than that of the filename-based matching algorithm, but it comes with the advantage of higher

Table 6.8: Processing time for ten Java projects

Project	**Assertions**	**Processing Time (hh:mm:ss)**
Spotless	394	0:09:32
Equals Verifier	794	0:07:14
Bach	1171	0:49:04
Hibernate-OGM	1840	0:27:39
Query DSL	2982	2:21:18
Apache Mahout	4069	0:50:41
Activiti	6490	3:19:34
Byte-Buddy	8648	1:53:17
KSQL	10819	8:45:20
Apache Pinot	17659	5:13:00

Table 6.9: Computer specifications

Specification	**Detail**
Processor	12th Gen Intel(R) Core(TM) i5-12600H 2.70 GHz
Memory	32.0 GB (4800MT/s)
Storage	NVMe PC SN740 NVMe WD 256 GB
Operating System	Windows 11 Pro

accuracy. It is important to note that the focus of this research is on evaluating the performance of the matching algorithms and demonstrating how the TWD adherence metric can be used on projects, not optimizing the performance of the programs used in this research; therefore, the figures presented in Table 6.8 are meant to provide an estimation of the processing times on a desktop system. Shorter processing times can be achieved through better hardware and making use of concurrency where appropriate.

6.5 Threats to Validity

6.5.1 Internal Validity

Internal validity refers to the degree to which the research is able to establish a causal relationship between the independent variable and the dependent variable while mitigating the effect of covert variables [78].

It was demonstrated in Subsection 6.1 that there is no correlation between the performance of the matching algorithms and project size or the number of assertions in a project. However, the performance of the matching algorithms can still be affected by factors such as the application of different testing styles or frameworks and the sample size of test files. The effect of these covert variables on the overall performance was mitigated by calculating the bootstrap confidence interval for the weighted F1-scores of the two matching algorithms.

The manual analysis of test files to determine the production files that are tested may have introduced human error. This could include overlooking a production file that is tested, incorrectly identifying a production file that is not tested, or misunderstanding the code in the test file. To reduce the manual labour and the likelihood of human error, a script was executed every time a new test file was sampled. The script would list all the production classes that appear anywhere in the test code of that particular test file. This allowed one to peruse through only a subset of all the production files that exist in the project, thereby reducing the number of production class names to look for manually in the test code.

The major variables believed to affect TWD adherence, namely time deltas between added or modified assertions and the production methods they test, as well as the method coverage of each added or modified assertion, were included in the mathematical expression given by Equation (5.10). A potential covert variable, namely the overall testing effort of projects, was controlled by grouping projects together based on the number of assertions that exist in each project before comparing the TWD adherence of projects. However, other potential covert variables, such as developer experience and team culture (which may have an effect on the TWD adherence of a project), were not controlled.

It was concluded in Section 6.3 that there is no correlation between development step sizes and the TWD adherence of projects, as most of the clusters had a statistically insignificant correlation between the two variables. However, it is possible that other

variables play a role in the development step size, such as project complexity [79]. Controlling these variables may help uncover a statistically significant correlation.

6.5.2 External Validity

External validity refers to the generalizability of the research findings to other contexts or populations [78].

In Chapter 4, a specific type of filename-based matching algorithm was implemented; the source code for the implementation was not provided by any existing study. As a result, it is possible that the performance reported in this study may not be generalizable to other studies. This was mitigated by implementing the filename-based matching algorithm using the Ratcliff-Obershelp algorithm from Python's standard library, as it is able to find multiple production classes with similar filenames in a way that can be replicated.

The performance of the statement-based matching algorithm may not be generalizable to projects that use testing frameworks whose syntax is substantially different from that of JUnit or projects that rely heavily on mocks or stubs for testing purposes. This was mitigated by including projects with varying testing styles in the performance analysis stage and by calculating the bootstrap confidence interval; this allowed the sample space of test files to have a more general representation so that the performance is not biased towards projects that employ simple unit tests with JUnit. Moreover, since JUnit and TestNG are known to be the most popular testing frameworks [59], this threat is dismissed.

It was found in Subsection 6.3.2 that the correlation between the relative TWD adherence values of eleven projects and the TDD scores of the same projects is strong, positive, and significant. However, due to the small sample size used for validation, it is possible that the result may not hold when the system is validated against a much larger sample size of projects. More data in the form of surveys is required to increase confidence in the claim that the mathematical functions defined for TWD adherence are a suitable way to determine the TDD adherence of software projects.

6.5.3 Construct Validity

Construct validity refers to the degree to which the operational definitions of the variables in the study accurately reflect the theoretical concepts they are intended to

measure. It is concerned with examining whether the quantities that were intended to be measured are actually measured and interpreted correctly [78].

Construct validity was ensured during the implementation of the TWD adherence system by drawing a clear distinction between TWD and TDD. In addition, the TWD adherence system was validated against eleven projects from two studies that obtained survey data from the developers of the projects. Lastly, the method coverage subsystem that is part of the TWD adherence system was validated against eighteen Java projects using test coverage data from Codecov. Both validations yielded satisfactory results; the correlation in both cases was strong, positive, and statistically significant. It is important to note that TDD follows a strict red-green-refactor cycle, whereas the order of writing tests and production does not matter in the case of a TWD cycle. This constitutes a limitation of the TWD adherence tool developed in this study, as it suggests that it cannot be used to detect the strict application of TDD in a project. Rather, the value of the TWD adherence tool is that it serves as an additional mechanism to indicate the use of TDD in a project. The tool can be combined with other mechanisms, such as surveys, to determine the TDD adherence of projects.

Construct validity was ensured during the implementation of the tool that evaluates the median development step size in a project by validating the underlying tools it depends on and by constructing a number of sanity tests. The median development step size of a project is intended to measure the median production method churn (the sum of changes to a production method up to the point at which it is tested) for all methods in a project. Such a measurement depends on the performance of the TWD adherence system and the test-to-code traceability tool, both of which were validated. In addition, sanity tests (one of which was included in the discussion in Section 5.4) were used to ensure that the tool operated as expected. However, important exceptions that were not considered in the calculation of the development step size are the complexity introduced during merge conflicts, since the actual development steps will be misrepresented.

6.6 Conclusion

Results for the file matching algorithms, the TWD adherence assessment system, and the development step size assessment system have been presented, and answers to the research questions have been provided. The statement-based matching algorithm has a weighted F1-score of 0.771 with a 95% confidence interval of [0.741, 0.786], while

the filename-based matching algorithm has a weighted F1-score of 0.218 with a 95% confidence interval of [0.193, 0.246]. The performance of both matching algorithms was found to be independent of project size or the number of assertions in a project. The study revealed a strong and positive correlation between the system's method coverage and Codecov's code coverage across eighteen projects ($r_s = 0.744$, p-value $= 0.0003$). Similarly, there was a strong and positive correlation between relative TWD adherence and survey scores from researchers across eleven projects ($r_s = 0.723$, p-value $= 0.012$). It was concluded that there is no correlation between the median production method churn and the relative TWD adherence of projects, as conflicting correlations were observed in different project clusters.

Chapter 7

Conclusion

This study analyses TDD adherence in several open-source software projects using a statement-based tool that establishes test-to-code traceability links.

Research is conducted into the techniques used to establish test-to-code traceability links that are used in several studies. They are subdivided into static, dynamic, and composite techniques. The filename-based technique is found to be the most popular in existing studies, but it suffers from low recall. A statement-based matching algorithm is proposed and implemented to address the performance issues of the filename-based matching algorithm. The weighted F1-score of the statement-based and filename-based matching algorithms is calculated using 500 test files from sixteen Java projects, and the bootstrap confidence interval of the weighted F1-score is calculated to represent the overall performance of both matching algorithms.

A mathematical function is defined from first principles to quantify the extent to which TWD was applied in a project. The function maps the time deltas and method coverage between an added or modified assertion and the production method it covers to a numerical value that represents the absolute TWD adherence of a project. Furthermore, a mathematical expression is defined to transform a project's absolute TWD adherence to its relative TWD adherence based on the number of unique assertions it has relative to other projects, and the K-means algorithm is used to cluster projects according to the number of assertions they contain. A total of 100 Java projects are used to demonstrate the ranking of projects by their relative TWD adherence.

A method to calculate the development step size of a project is proposed in order to discover whether projects that adhere to TDD necessarily develop code in smaller steps. The median development step size is measured by calculating the median

production method churn that corresponds to an added or modified assertion up to the last point at which that piece of production code is tested. The correlation between the relative TWD adherence of projects in a single cluster and the median production method churn for each project is measured to determine whether projects that adhere to TWD are associated with smaller development step sizes.

The key findings of the research questions are as follows:

RQ1 - Accurate matching: The statement-based matching algorithm, with a weighted F1-score of 0.771 and a 95% confidence interval of [0.741, 0.786], is found to be significantly more accurate than the filename-based matching algorithm, with a weighted F1-score of 0.218 and a 95% confidence interval of [0.193, 0.246].

RQ2 - TDD adherence: The relative TWD adherence values of eleven projects are found to be highly correlated with TDD scores from surveys ($r_s = 0.723$ and p-value $= 0.012$). In addition, the method coverage of eighteen projects is found to be highly correlated with code coverage obtained from Codecov ($r_s = 0.744$ and p-value $= 0.0003$).

RQ3 - Development step size: There is no significant correlation between the median production method churn and the relative TWD adherence of projects in all but one out of eight cases. For the cluster of projects with a number of assertions in the range [14392, 17659], the correlation is very strong and significant ($r_s = 0.972$, p-val $= 0.001$). This is contrary to the hypothesis that projects with higher TWD adherence have smaller development step sizes [80]. Only the cluster of projects with a number of assertions in the range $[14392 - 17659]$ has a statistically significant and positive correlation ($r_s = 0.972$, p-val: 0.001).

7.1 Datasets Used

In the dissertation, different groups of Java projects are analysed for specific purposes. Chapter 4 discusses three projects to illustrate how the performance of the matching algorithms can be assessed. Chapter 6.1 focuses on sixteen projects from which 500 test files are extracted to evaluate the performance of the matching algorithms. Chapter 6.2 features eleven projects used to validate the TWD adherence system, sourced from two separate surveys. Additionally, eighteen projects are employed to validate the method coverage system by comparing the system-calculated method coverage with actual test coverage. Lastly, the total sample of 100 projects represents

the overall benchmark used to evaluate and compare the relative TWD adherence values across all projects.

7.2 Future Work

While it has been shown that the statement-based matching algorithm performs much better than the filename-based matching algorithm, its performance still depends on the programming style and testing frameworks used in a project, as well as whether or not mocks or stubs are used in tests. Future work could entail the use of a large language model to establish test-to-code traceability links since large language models excel at capturing complex relationships between variables, class names, and overall code semantics [81]. In particular, the use of OpenAI's API and GPT-4.0 can be considered for this task.

Several studies have investigated the relationship between the TDD adherence of projects and aspects related to code quality and complexity. This research has introduced a new TWD adherence metric; therefore, it would be beneficial to revisit existing studies to determine the correlation between code quality and complexity and the new TWD adherence metric.

Only one variable is used to cluster projects in order to determine relative TWD adherence. Future work could entail using more than one variable to cluster projects before determining relative TWD adherence. It is recommended that McCabe's cyclomatic complexity be included as another variable for clustering, since the complexity of a project may have an effect on the testing applied to the project.

References

[1] K. Beck. *Extreme Programming Explained*. Addison-Wesley Professional, 1999.

[2] B. Pedroso, R. Jacobi, and M. Pimenta. "TDD Effects: Are We Measuring the Right Things?" In *Agile Processes in Software Engineering and Extreme Programming: 11th International Conference, XP 2010, Trondheim, Norway, June 1-4, 2010. Proceedings 11*, pp. 393–394. Springer, 2010.

[3] L. Zhang, J.-H. Tian, J. Jiang, Y.-J. Liu, M.-Y. Pu, and T. Yue. "Empirical Research in Software Engineering — a Literature Survey." *Journal of Computer Science and Technology*, vol. 33, pp. 876–899, 2018.

[4] C. M. Callahan and T. R. Moon. "Sorting the Wheat from the Chaff: What Makes for Good Evidence of Effectiveness in the Literature in Gifted Education?" *Gifted Child Quarterly*, vol. 51, no. 4, pp. 305–319, 2007.

[5] A. Santos, S. Vegas, O. Dieste, F. Uyaguari, A. Tosun, D. Fucci, B. Turhan, G. Scanniello, S. Romano, I. Karac, et al. "A Family of Experiments on Test-Driven Development." *Empirical Software Engineering*, vol. 26, pp. 1–53, 2021.

[6] B. Van Rompaey and S. Demeyer. "Establishing Traceability Links Between Unit Test Cases and Units Under Test." In *2009 13th European Conference on Software Maintenance and Reengineering*, pp. 209–218. IEEE, 2009.

[7] G. K. Kirui and S. P. Levitt. "Matching Production and Test Files: A Comparison of Filename and Statement-Based Approaches." In *Annual Conference of South African Institute of Computer Scientists and Information Technologists*, pp. 3–18. Springer, 2023.

[8] P. Maeder, M. Riebisch, and I. Philippow. "Traceability for Managing Evolutionary Change." In *SEDE*, pp. 1–8. 2006.

[9] K. Jaber, B. Sharif, and C. Liu. "A Study on the Effect of Traceability Links in Software Maintenance." *IEEE Access*, vol. 1, pp. 726–741, 2013.

[10] J. H. Hayes, A. Dekhtyar, and D. S. Janzen. "Towards Traceable Test-Driven Development." In *2009 ICSE Workshop on Traceability in Emerging Forms of Software Engineering*, pp. 26–30. IEEE, 2009.

[11] S. Demeyer, B. Verhaeghe, A. Etien, N. Anquetil, and S. Ducasse. "Evaluating the Efficiency of Continuous Testing during Test-Driven Development." In *2018 IEEE Workshop on Validation, Analysis and Evolution of Software Tests (VST)*, pp. 21–25. IEEE, 2018.

[12] H. Ogasawara, M. Aizawa, and A. Yamada. "Experiences with program static analysis." In *Proceedings Fifth International Software Metrics Symposium. Metrics (Cat. No. 98TB100262)*, pp. 109–112. IEEE, 1998.

[13] Z. Yin, X. Ma, J. Zheng, Y. Zhou, L. N. Bairavasundaram, and S. Pasupathy. "An Empirical Study on Configuration Errors in Commercial and Open Source Systems." In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pp. 159–172. 2011.

[14] A. Gosain and G. Sharma. "A Survey of Dynamic Program Analysis Techniques and Tools." In *Proceedings of the 3rd International Conference on Frontiers of Intelligent Computing: Theory and Applications (FICTA) 2014: Volume 1*, pp. 113–122. Springer, 2015.

[15] P. Bogren and I. Kristola. "Exploring the use of call stack depth limits to reduce regression testing costs." Final Project DT133G, Mid Sweden University, Mid Sweden University, 2021. Main field of study: Computer Engineering BA (C).

[16] M. Mongiovì, G. Giannone, A. Fornaia, G. Pappalardo, and E. Tramontana. "Combining static and dynamic data flow analysis: a hybrid approach for detecting data leaks in Java applications." In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, pp. 1573–1579. 2015.

[17] JUnit Team. "JUnit FAQ.", 2023. URL https://junit.org/junit4/faq.html#running_15/.

[18] A. Zaidman, B. Rompaey, A. Deursen, and S. Demeyer. "Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining." *Empirical Software Engineering*, vol. 16, pp. 325–364, 04 2011.

[19] J. Hanhela. *Usage of test-driven development in open source projects*. Master's thesis, J. Hanhela, 2015.

[20] S. Wang, M. Wen, Y. Liu, Y. Wang, and R. Wu. "Understanding and Facilitating the Co-Evolution of Production and Test Code." In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 272–283. IEEE, 2021.

[21] P. S. Kochhar, T. F. Bissyandé, D. Lo, and L. Jiang. "Adoption of Software Testing in Open Source Projects A Preliminary Study on 50,000 Projects." In *2013 17th European Conference on Software Maintenance and Reengineering*, pp. 353–356. IEEE, 2013.

[22] J. W. Ratcliff and D. E. Metzener. "Pattern-matching-the gestalt approach." *Dr Dobbs Journal*, vol. 13, no. 7, p. 46, 1988.

[23] S. Zhang, Y. Hu, and G. Bian. "Research on String Similarity Algorithm based on Levenshtein Distance." In *2017 IEEE 2nd Advanced Information Technology, Electronic and Automation Control Conference (IAEAC)*, pp. 2247–2251. IEEE, 2017.

[24] N. Borle, M. Feghhi, E. Stroulia, R. Grenier, and A. Hindle. "Analyzing the Effects of Test Driven Development in GitHub." In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pp. 1062–1062. IEEE, 2018.

[25] A. Qusef, R. Oliveto, and A. De Lucia. "Recovering Traceability Links between Unit Tests and Classes Under Test: An Improved Method." In *2010 IEEE International Conference on Software Maintenance*, pp. 1–10. IEEE, 2010.

[26] R. White, J. Krinke, and R. Tan. "Establishing Multilevel Test-to-Code Traceability Links." In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pp. 861–872. 2020.

[27] A. Qusef, G. Bavota, R. Oliveto, A. D. Lucia, and D. Binkley. "Evaluating test-to-code traceability recovery methods through controlled experiments." *Journal of Software: Evolution and Process*, vol. 25, no. 11, pp. 1167–1191, 2013.

[28] A. Kicsi, V. Csuvik, and L. Vidács. "Large Scale Evaluation of Natural Language Processing Based Test-to-Code Traceability Approaches." *IEEE Access*, vol. 9, pp. 79089–79104, 2021.

[29] A. Mahmoud and N. Niu. "Source Code Indexing for Automated Tracing." In *Proceedings of the 6th International Workshop on Traceability in Emerging Forms of Software Engineering*, pp. 3–9. 2011.

[30] V. Bauer, T. Völke, and S. Eder. "Comparing TF-IDF and LSI as IR technique in an Approach for Detecting Semantic Re-implementations in Source Code." Research report, Technical University of Munich, Institute for Informatics, Munich, Germany, 2015.

[31] M. Gaudencio, A. Dantas, and D. D. Guerrero. "Can Computers Compare Student Code Solutions as Well as Teachers?" In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*, pp. 21–26. 2014.

[32] Z. Zhang, Y. Lei, J. Xu, X. Mao, and X. Chang. "TFIDF-FL: Localizing Faults Using Term Frequency-Inverse Document Frequency and Deep Learning." *IEICE TRANSACTIONS on Information and Systems*, vol. 102, no. 9, pp. 1860–1864, 2019.

[33] K. Beck. *Test-Driven Development: By Example*. Addison-Wesley Professional, 2003.

[34] B. C. Hardgrave, F. D. Davis, and C. K. Riemenschneider. "Investigating Determinants of Software Developers' Intentions to Follow Methodologies." *Journal of Management Information Systems*, pp. 123–151, 2003.

[35] N. C. Schaeffer, J. A. Seltzer, and M. Klawitter. "Estimating sponse and Response Bias: Resident and sident Parents' Reports About Child Support." *Sociological Methods & Research*, vol. 20, no. 1, pp. 30–59, 1991.

[36] I. Karac, B. Turhan, and N. Juristo. "A Controlled Experiment with Novice Developers on the Impact of Task Description Granularity on Software Quality in Test-Driven Development." *IEEE Transactions on Software Engineering*, vol. 47, no. 7, pp. 1315–1330, 2019.

[37] K. Buffardi and S. H. Edwards. "Exploring Influences on Student Adherence to Test-Driven Development." In *Proceedings of the 17th ACM Annual Conference on Innovation and Technology in Computer Science Education*, pp. 105–110. 2012.

[38] Z. Hu, Y. Song, and E. F. Gehringer. "A Test-Driven Approach to Improving Student Contributions to Open-Source Projects." In *2019 IEEE Frontiers in Education Conference (FIE)*, pp. 1–9. IEEE, 2019.

[39] J. C. Sanchez, L. Williams, and E. M. Maximilien. "On the Sustained Use of a Test-Driven Development Practice at IBM." In *Agile 2007 (AGILE 2007)*, pp. 5–14. IEEE, 2007.

[40] Codecov Team. "Codecov." https://codecov.io/. Accessed: 2024-07-21.

[41] Coveralls Team. "Coveralls." https://coveralls.io/. Accessed: 2024-07-21.

[42] J. C. Monteiro and A. L. Oliveira. "Finite State Machine Decomposition For Low Power." In *Proceedings of the 35th Annual Design Automation Conference*, pp. 758–763. 1998.

[43] H. Kou, P. M. Johnson, and H. Erdogmus. "Operational definition and automated inference of test-driven development with Zorro." *Automated Software Engineering*, vol. 17, pp. 57–85, 2010.

[44] M. Hilton, N. Nelson, H. McDonald, S. McDonald, R. Metoyer, and D. Dig. "TDDViz: Using Software Changes to Understand Conformance to Test Driven Development." In *Agile Processes, in Software Engineering, and Extreme Programming: 17th International Conference, XP 2016, Edinburgh, UK, May 24-27, 2016, Proceedings 17*, pp. 53–65. Springer International Publishing, 2016.

[45] M. Beller, G. Gousios, A. Panichella, and A. Zaidman. "When, How, and Why Developers (Do Not) Test in Their IDEs." In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pp. 179–190. 2015.

[46] Z. Lubsen, A. Zaidman, and M. Pinzger. "Using Association Rules to Study the Co-evolution of Production & Test Code." In *2009 6th IEEE International Working Conference on Mining Software Repositories*, pp. 151–154. IEEE, 2009.

[47] Python Software Foundation. "difflib – Helpers for computing deltas." Python 3 Documentation. Retrieved from https://docs.python.org/3/library/difflib.html.

[48] A. Čaušević, D. Sundmark, and S. Punnekkat. "Test Case Quality in Test Driven Development: A Study Design and a Pilot Experiment." In *16th International Conference on Evaluation & Assessment in Software Engineering (EASE 2012)*, pp. 223–227. IET, 2012.

[49] D. Fucci, H. Erdogmus, B. Turhan, M. Oivo, and N. Juristo. "A Dissection of the Test-Driven Development Process: Does It Really Matter to Test-First or to Test-Last?" *IEEE Transactions on Software Engineering*, vol. 43, no. 7, pp. 597–614, 2016.

[50] D. Janzen and H. Saiedian. "Does Test-Driven Development Really Improve Software Design Quality?" *IEEE Software*, vol. 25, no. 2, pp. 77–84, 2008.

[51] B. George. *Analysis and Quantification of Test Driven Development Approach*. Master's thesis, North Carolina State University, 2002.

[52] L. Madeyski and Ł. Szała. "The Impact of Test-Driven Development on Software Development Productivity — An Empirical Study." In *European Conference on Software Process Improvement*, pp. 200–211. Springer, 2007.

[53] J. H. Vu, N. Frojd, C. Shenkel-Therolf, and D. S. Janzen. "Evaluating Test-Driven Development in an Industry-sponsored Capstone Project." In *2009 Sixth International Conference on Information Technology: New Generations*, pp. 229–234. IEEE, 2009.

[54] M. Pančur and M. Ciglarič. "Impact of test-driven development on productivity, code and tests: A controlled experiment." *Information and Software Technology*, vol. 53, no. 6, pp. 557–573, 2011.

[55] JUnit Team. "JUnit 5." https://junit.org/junit5/. Accessed: 2024-07-21.

[56] TestNG Team. "TestNG." https://https://testng.org/. Accessed: 2024-07-21.

[57] JUnit Team. "JUnit4." https://github.com/junit-team/junit4. Commit: cc7c5005. Accessed: 2024-07-21.

[58] D. Spadini, M. Aniche, and A. Bacchelli. "PyDriller: Python Framework for Mining Software Repositories." In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 908–911. 2018.

[59] A. Zerouali and T. Mens. "Analyzing the Evolution of Testing Library Usage in Open Source Java Projects." In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 417–421. IEEE, 2017.

[60] M. Leotta, M. Cerioli, D. Olianas, and F. Ricca. "Fluent vs Basic Assertions in Java: An Empirical Study." In *2018 11th International Conference on the Quality of Information and Communications Technology (QUATIC)*, pp. 184–192. IEEE, 2018.

[61] I. Baidari and A. Hanagawadimath. "Traversing directed cyclic and acyclic graphs using modified BFS algorithm." In *2014 Science and Information Conference*, pp. 175–181. IEEE, 2014.

[62] Apache Commons Math Team. "Apache Commons Math." https://github.com/apache/commons-math. Accessed: 2024-07-22.

[63] Swagger API Team. "Swagger Core GitHub Repository." https://github.com/swagger-api/swagger-core, accessed 2022.

[64] Docker Java Team. "Docker Java." https://github.com/docker-java/docker-java. Accessed: 2024-07-22.

[65] P. R. Cohen. *Empirical Methods for Artificial Intelligence*, vol. 139. MIT press Cambridge, MA, 1995.

[66] S. Bannerman and A. Martin. "A multiple comparative study of test-with development product changes and their effects on team speed and product quality." *Empirical Software Engineering*, vol. 16, no. 2, pp. 177–210, 2011.

[67] K. P. Sinaga and M.-S. Yang. "Unsupervised K-Means Clustering Algorithm." *IEEE Access*, vol. 8, pp. 80716–80727, 2020.

[68] M. Syakur, B. Khotimah, E. Rochman, and B. D. Satoto. "Integration K-Means Clustering Method and Elbow Method For Identification of The Best Customer Profile Cluster." In *IOP Conference Series: Materials Science and Engineering*, vol. 336, p. 012017. IOP Publishing, 2018.

[69] S. Bhanja and A. Das. "Impact of Data Normalization on Deep Neural Network for Time Series Forecasting." 2019. URL https://arxiv.org/abs/1812.05519.

[70] J. Hauke and T. Kossowski. "Comparison of Values of Pearson's and Spearman's Correlation Coefficients on the Same Sets of Data." *Quaestiones Geographicae*, vol. 30, no. 2, pp. 87–93, 2011.

[71] R. de Nijs and T. L. Klausen. "On the Expected Difference between Mean and Median." *Electronic Journal of Applied Statistical Analysis*, vol. 6, no. 1, pp. 110–117, 2013.

[72] D. Spadini, M. Aniche, M. Bruntink, and A. Bacchelli. "To Mock or Not To Mock? An Empirical Study on Mocking Practices." In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pp. 402–412. IEEE, 2017.

[73] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian. "The Promises and Perils of Mining GitHub." In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pp. 92–101. 2014.

[74] Codecov. "Codecov About Page.", 2023. URL https://about.codecov.io/.

[75] R. Hilton. "Quantitatively Evaluating Test-Driven Development by Applying Object-Oriented Quality Metrics to Open Source Projects." *PDF]. Available: http://www. nomachetejuggling. com/files/tdd_thesis. pdf, M. Sc thesis, Department of Computer & Information Sciences, Regis University, CO*, 2009.

[76] A. M. Smith, J. Geiger, G. M. Kapfhammer, and M. L. Soffa. "Test Suite Reduction and Prioritization with Call Trees." In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*, pp. 539–540. 2007.

[77] Software Improvement Group. "SIG-Fasten GitHub Repository." https://github.com/software-improvement-group-research/fasten, accessed 2022.

[78] S. Easterbrook, J. Singer, M.-A. Storey, and D. Damian. "Selecting Empirical Methods for Software Engineering Research." *Guide to Advanced Empirical Software Engineering*, pp. 285–311, 2008.

[79] D. Ståhl, A. Martini, and T. Mårtensson. "Big Bangs and Small Pops: On Critical Cyclomatic Complexity and Developer Integration Behavior." In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pp. 81–90. IEEE, 2019.

[80] G. Hill and Made Tech. "Transcript of 'Test Driven Development, with Geepaw Hill'." https://www.madetech.com/podcast/episode-15-geepaw-hill/transcript/, 2024. Podcast transcript.

[81] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. Grundy, and H. Wang. "Large language models for software engineering: A systematic literature review." *arXiv preprint arXiv:2308.10620*, 2023.

Appendix A

Technical Aspects of Absolute TWD Adherence System

The following listings are associated with the illustrative project discussed in Chapter 5. These listings represent the outputs of various core components of the system that evaluates the test-with development (TWD) adherence of a project. They are intended to inform the reader of the technical aspects of the system, providing a deeper understanding of the procedures it performs. These listings are abstracted from the main dissertation and can be referenced as needed.

```
{
    "ShapesGeometryTest": [
        "Rectangle",
        "Circle"
    ]
}
```

Listing A.1: Output of the mapping system given the Shapes project

```
{
  "time-deltas": [
    76.0,
    625.0
  ],
  "count": 2,
  "mean": 350.5,
  "median": 350.5
}
```

Listing A.2: Output of the time deltas subsystem given the data structure of the shapes project

The `modifications` field stores a dictionary where the key is either “added” or “deleted”, and the value is a list of tuples. The first element of the tuple is an integer which stores the line number on which an addition or deletion has occurred. The second element of the tuple is a string which stores the literal line of code that was added or deleted.

```
{
  "added": [
    [
      11,
      "return 2 * Math . PI * this . radius ; // corrected"
    ]
  ],
  "deleted": [
    [
      11,
      "return Math.PI * this.radius; // Initially incorrect"
    ]
  ]
}
```

Listing A.3: Representation of the value stored in the `modifications` field.

```
{
  "elements": [
    3,
    5
  ],
  "count": 2,
  "mean": 4.0,
  "median": 4.0
}
```

Listing A.4: Output of the production method churn system given the data structure of the Shapes project.

Appendix B

Comprehensive List of Projects Analysed

The following appendix provides a comprehensive list of 100 open-source software projects hosted on GitHub that were analysed in this study. The inclusion of this detailed list is crucial for ensuring the reproducibility and integrity of the research outcomes. Each entry in the table includes the project name, the repository URL, the latest commit hash during the analysis, and a brief description of the project.

This appendix serves as an essential resource for researchers and practitioners who wish to replicate or extend the study. By providing specific commit hashes, the analysis can be precisely reproduced, facilitating further validation and comparison of the findings. The descriptions offer a quick reference to understand the scope and functionality of each project, making it easier to identify relevant projects for future research.

In summary, this appendix not only enhances the transparency and reproducibility of the study but also provides a valuable dataset for the broader research community interested in open-source software analysis and test-to-code traceability.

Table B.1: Comprehensive list of projects analysed in this study (1 - 16)

No.	Project	Repository URL	Commit Hash	Description
1	Activiti	https://github.com/Activiti/Activiti	851d55befe	A workflow and Business Process Management (BPM) Platform
2	Apache Calcite	https://github.com/apache/calcite	040e85a92	A dynamic data management framework.
3	Apache Cayenne	https://github.com/apache/cayenne	983c51cd1	An open source persistence framework
4	Apache Doris	https://github.com/apache/doris	477b28efa	A high performance and unified analytics database.
5	Apache Dubbo	https://github.com/apache/dubbo	9cdda8207	An RPC and microservice framework.
6	Apache Gobblin	https://github.com/apache/gobblin	22e984ba7	A distributed data integration framework.
7	Apache Hudi	https://github.com/apache/hudi	0bc3eb8aab	A transactional data lake platform
8	Apache Iceberg	https://github.com/apache/iceberg	5688d59f6	A high-performance format for huge analytic tables.
9	Apollo	https://github.com/apolloconfig/apollo	a9579f177	A reliable configuration management system.
10	Archunit	https://github.com/TNG/ArchUnit	cdc0582d	A Java architecture test library
11	Argo-UML	https://github.com/argouml-tigris-org/argouml	3ddc7f94d8	A UML diagramming application.
12	AssertJ-Core	https://github.com/assertj/assertj	1d02a978b	A library providing easy to use rich typed assertions
13	Atom-Hopper	https://github.com/rackerlabs/atom-hopper	03a3f475	A framework for processing Atom formatted events.
14	Bach	https://github.com/sormuras/bach	740481fe	A lightweight Java build tool
15	Book-keeper	https://github.com/apache/bookkeeper	1666d820a3	A low latency storage service for append-only workloads
16	Byte-Buddy	https://github.com/raphw/byte-buddy	ab9cae77dd	A code generation library for creating Java classes

Table B.2: Comprehensive list of projects analysed in this study (17 - 30)

No.	Project	Repository URL	Commit Hash	Description
17	Cas	https://github.com/apereo/cas	ff899aa0cb	Identity & Single Sign On
18	Checks-style	https://github.com/checkstyle/checkstyle	40bc1d0f1	A tool for checking Java source code for adherence to a Code Standard
19	Chunjun	https://github.com/DTStack/chunjun	c09a4c139	A data integration framework
20	Commons Math	https://github.com/apache/commons-math	e85dd5999	A library of lightweight, mathematics components
21	Commons Pool	https://github.com/apache/commons-pool	708f2e92	An object-pooling API library
22	CWA Server	https://github.com/corona-warn-app/cwa-server	5b96bac0	Corona-Warn-App for Germany based on the exposure notification API
23	Cyclops	https://github.com/aol/cyclops	11cedae18	A platform for writing applications in Java 8.
24	DBeaver	https://github.com/dbeaver/dbeaver	2e1a4eda8b	A free universal database tool and SQL client
25	Debezium	https://github.com/debezium/debezium	7a27e0db2	A data streaming platform for change data capture
26	Disruptor	https://github.com/LMAX-Exchange/disruptor	eaca68c	A high performance inter-thread messaging library
27	Docker-Java	https://github.com/docker-java/docker-java	8253a999	A platform to manage virtualized application containers
28	Eclipse-CS	https://github.com/checkstyle/eclipse-cs	f339bd57	Integrates the Java code auditor into Eclipse IDE
29	Elide	https://github.com/yahoo/elide	a92efb877	A library to setup a GraphQL or JSON API web service
30	Equals Verifier	https://github.com/jqno/equalsverifier	4d7e2388	Verify the contract for equals and hashCode methods in a class

Table B.3: Comprehensive list of projects analysed in this study (31 - 44)

No.	Project	Repository URL	Commit Hash	Description
31	Eventbus	https://github.com/greenrobot/EventBus	842a4a3	Simplifies communication between components
32	FastJSON	https://github.com/alibaba/fastjson	7abc84fc2	Converts Java Objects into their JSON representation
33	FitNesse	https://github.com/unclebob/fitnesse	df3e687b6	An acceptance testing framework and wiki
34	Fresco	https://github.com/facebook/fresco	df720bd4e	An Android library for managing images
35	Gephi	https://github.com/gephi/gephi	f27ae4fc8	A platform for visualizing and manipulating graphs
36	GRPC	https://github.com/grpc/grpc-java	47ddfa4f2	A framework for building performant RPC services.
37	Guava	https://github.com/google/guava	48b02ce53c	Google core libraries for Java
38	Guice	https://github.com/google/guice	d6997b0ad	A dependency injection framework for Java
39	Helidon	https://github.com/helidon-io/helidon	893239cb1	Java libraries for writing microservices
40	Hibernate OGM	https://github.com/hibernate/hibernate-ogm	7773e663c	Domain model persistence for NoSQL datastores
41	Hudson	https://github.com/hudson/hudson-2.x	82286f089	Monitors the execution of repeated jobs
42	Hystrix	https://github.com/Netflix/Hystrix	3cb21589	A latency and fault tolerance library
43	JabRef	https://github.com/JabRef/jabref	b33ebfb4a	Graphical app for managing BibTeX databases
44	Java-Native Access	https://github.com/java-native-access/jna	673079f3a	Provides Java programs easy access to native shared libraries

Table B.4: Comprehensive list of projects analysed in this study (45 - 58)

No.	Project	Repository URL	Commit Hash	Description
45	Java-Melody	https://github.com/javamelody/javamelody	6e90f932	Monitors Java apps in QA and production environments.
46	Jenkins Remoting	https://github.com/jenkinsci/remoting	be63e462	Implements communication in Jenkins automation server
47	JFree Chart	https://github.com/jfree/jfreechart	b62b85d0	A 2D chart library for Java applications
48	JGiven	https://github.com/TNG/JGiven	6bac970c	Behaviour Driven Development (BDD) tool
49	Jit-Watch	https://github.com/AdoptOpenJDK/jitwatch	a289525	A log analyser / visualiser for Java HotSpot JIT compiler.
50	Jodd	https://github.com/oblac/jodd	647898309	Set of micro-frameworks and developer-friendly tools
51	JUnit4	https://github.com/junit-team/junit4	cc7c5005	A programmer-oriented testing framework for Java
52	JUnit5	https://github.com/junit-team/junit5	4a3bc288dd	A programmer-friendly testing framework for Java
53	KSQL	https://github.com/confluentinc/ksql	9515e770d6	A database for building stream processing apps
54	Linked-In Ambry	https://github.com/linkedin/ambry	0a4f63d38	A distributed object store
55	Linked-In Venice	https://github.com/linkedin/venice	92475897e	A derived data storage platform
56	Liqui-base	https://github.com/liquibase/liquibase	9d6d14455	Track and deploy database schema changes
57	Lombok	https://github.com/projectlombok/lombok	731bb185	A library for reducing boilerplate code
58	Apache Mahout	https://github.com/apache/mahout	6588ded69	Library for interfacing with quantum computing backends

Table B.5: Comprehensive list of projects analysed in this study (59 - 70)

No.	Project	Repository URL	Commit Hash	Description
59	MCA Selector	https://github.com/Querz/mcaselector	763e6b4	A tool to select chunks in Minecraft worlds for deletion/export.
60	Mine-Craft Forge	https://github.com/MinecraftForge/MinecraftForge	4011c3fb0	A set of tools and modifications to the Minecraft base game
61	Mockito	https://github.com/mockito/mockito	02d63565a	A mocking framework for unit tests written in Java
62	Mock-Server	https://github.com/mock-server/mockserver	829340399	Enables mocking of any system integrated with HTTP/HTTPS
63	MSSQL JDBC	https://github.com/microsoft/mssql-jdbc	5ed44517	A Type 4 JDBC driver that provides database connectivity
64	Nakadi	https://github.com/zalando/nakadi	42ceff5cd	An event bus that implements a RESTful API abstraction
65	Netty	https://github.com/netty/netty	7d540fc39c	An event-driven asynchronous network application framework
66	Onedev	https://github.com/theonedev/onedev	5262e7b9f	Self-hosted Git server with CI/CD and issue tracking
67	Opta-Planner	https://github.com/kiegroup/optaplanner	ebea550a93	A fast, easy-to-use, open source AI constraint solver for software developers
68	Apache Pinot	https://github.com/apache/pinot	6582750ea6	A realtime distributed OLAP datastore
69	Pin-Point	https://github.com/pinpoint-apm/pinpoint	e4404f2b98	An APM tool for large-scale distributed systems.
70	Query DSL	https://github.com/querydsl/querydsl	cd608f04e	Enables the construction of type-safe SQL-like queries

Table B.6: Comprehensive list of projects analysed in this study (71 - 82)

No.	Project	Repository URL	Commit Hash	Description
71	QuestDB	https://github.com/questdb/questdb	274b2e250	A time-series database for fast ingest and SQL queries
72	Recommenders	https://github.com/eclipse-archived/recommenders	60d8b8194	Code completion and developer productivity—tools for Eclipse
73	Redisson	https://github.com/redisson/redisson	f9c00cbf0	A Redis Java client and Real-Time Data Platform.
74	Apache Rocket MQ	https://github.com/apache/rocketmq	d5b4d8431	A cloud native messaging and streaming platform
75	Rumble	https://github.com/RumbleDB/rumble	274c2f046	A query engine for querying large-scale JSON datasets.
76	Runelite	https://github.com/runelite/runelite	73116f35f	An enhanced open-source Old School RuneScape client.
77	RxJava	https://github.com/ReactiveX/RxJava	6bdacb446	Composes asynchronous and event-based programs
78	Service Talk	https://github.com/apple/servicetalk	670cc9350	A networking framework that evolves with your app
79	SIG Fasten	https://github.com/software-improvement-group-research/fasten	789cb6d1	Analyses package dependency networks at the call graph level
80	Smart Tube	https://github.com/yuliskov/SmartTube	e1cd4dda8	An advanced player for set-top boxes and tvs running Android OS
81	Spotless	https://github.com/diffplug/spotless	8b249638	A code formatter for enforcing consistent style in codebases.
82	Spring-Boot	https://github.com/spring-projects/spring-boot	c83ab9bbc5	A framework for building stand-alone, production-grade Spring applications.

Table B.7: Comprehensive list of projects analysed in this study (83 - 94)

No.	Project	Repository URL	Commit Hash	Description
83	Spring-Security	https://github.com/spring-projects/spring-security	9cb668aec2	A framework for securing Spring applications.
84	Apache Storm	https://github.com/apache/storm	a8fb509f7	A distributed real-time computation system
85	Storm-Pot	https://github.com/chrisvest/stormpot	7af4a8c	A fast object pool for the JVM
86	Stripes	https://github.com/StripesFramework/stripes	5c795d45	A web application framework for Java
87	Swagger Core	https://github.com/swagger-api/swagger-core	55471d099	A Java implementation of the OpenAPI Specification
88	Synthea	https://github.com/synthetichealth/synthea	2fd1e964	A synthetic patient population simulator
89	Apache System DS	https://github.com/apache/systemds	c44bcca81	An open source ML system for the end-to-end data science lifecycle
90	Apache Tapestry4	https://github.com/apache/tapestry4	d1b8ade7e	A framework for building dynamic web applications.
91	Teku	https://github.com/Consensys/teku	182f5601f	An open-source Ethereum consensus client
92	Test Containers	https://github.com/testcontainers/testcontainers-java	263282b5	A library for integration tests with disposable containers.
93	TestNG	https://github.com/testng-team/testng	f1363151	A testing framework supporting data-driven tests.
94	Traccar	https://github.com/traccar/traccar	009e19b44	An open source GPS tracking system

Table B.8: Comprehensive list of projects analysed in this study (95 - 100)

No.	Project	Repository URL	Commit Hash	Description
95	Vavr	https://github.com/vavr-io/vavr	48a3193bc	An object-functional language extension to Java 8
96	Vert.x	https://github.com/eclipse-vertx/vert.x	9f6467871	A tool-kit for building reactive applications on the JVM
97	Wisser Jailer	https://github.com/Wisser/Jailer	82aa8cdd	A tool for database subsetting and relational data browsing.
98	XWiki Commons	https://github.com/xwiki/xwiki-commons	f7e1141ca	Libraries usable by projects side the XWiki ecosystem
99	XWiki Rendering	https://github.com/xwiki/xwiki-rendering	a42f19dc6	Converts input text into various output formats
100	Zaproxy	https://github.com/zaproxy/zaproxy	e2a4b593e	A free security tool for finding web application vulnerabilities

Appendix C

SAICSIT 2023 Conference Paper

The following paper was presented and published at the conference of the South African Institute of Computer Scientists and Technologists (SAICSIT) in 2023, where it was awarded the best computer science paper of SAICSIT 2023. This paper addresses the first research question of this dissertation: "Can test code be matched more accurately to production code using a statement-level approach?" It encompasses content from Chapter 4 and the results from Section 6.1 in Chapter 6.

The paper provides a literature review of popular methods to establish test-to-code traceability links and defines a problem statement. It then presents high-level implementations of the filename-based and statement-based matching algorithms. Additionally, it outlines a method for assessing and comparing the performance of these two algorithms using 500 test files from sixteen Java projects.

The findings indicate that the statement-based matching algorithm, with a 95% confidence interval of [0.6815, 0.7347], significantly outperforms the filename-based matching algorithm, which has a 95% confidence interval of [0.1931, 0.2459].



Matching Production and Test Files: A Comparison of Filename and Statement-Based Approaches

Gerald Kipruto Kirui and Stephen Phillip Levitt( )

University of the Witwatersrand, Johannesburg, South Africa
stephen.levitt@wits.ac.za

Abstract. Accurate matching of test and production files plays a critical role in the analysis and evaluation of Test-Driven Development (TDD). However, current approaches often yield unsatisfactory results due to their reliance on filename-based matching. The purpose of this study is to compare the performance of a statement-based matching algorithm with the traditional filename-based approach. A comprehensive evaluation was conducted using 500 tests from 16 open-source Java projects, wherein the weighted F1-scores of both methods were assessed. Subsequently, the 95% confidence intervals were determined using a pseudosample size of 500. The statement-based approach achieved a 95% confidence interval of [0.6815, 0.7347], while the filename-based method had a notably lower interval of [0.1931, 0.2459]. These results demonstrate the superior performance of the statement-based matching algorithm, providing a more accurate and reliable solution for matching test and production files in TDD research. In conclusion, the statement-based matching algorithm significantly outperforms the filename-based method, which will benefit TDD research by offering a more accurate method of matching production files to test files.

Keywords: Traceability Links · Assertion Analysis · F1-Score

1 Introduction

Test-Driven Development or Test-Driven Design (TDD) is a software development practice in which automated tests are written prior to production code in an incremental, iterative manner with the sole purpose of delivering quality code [4].

Researchers who are interested in software testing are often interested in matching test code files with the corresponding production code files that are being tested, since TDD requires tests to be written at a certain point in time with respect to production code.

In the case of empirical TDD studies, this matching allows researchers to use file commit timestamps to determine when tests are written with respect to the production code being tested. Code coverage as an indicator of TDD

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2023
A. Gerber and M. Coetzee (Eds.): SAICSIT 2023, CCIS 1878, pp. 3–18, 2023.
https://doi.org/10.1007/978-3-031-39652-6_1

is insufficient, as it includes no timing information regarding when tests are written. Most researchers take it for granted that matching a test file to its target production files by comparing filenames is reasonably accurate. In this paper, we investigate the accuracy of such a method and propose an alternative, more accurate approach.

Section 2 discusses related work and outlines the problem statement. Sections 3 and 4 describe the methods of operation of the filename-based and statement-based matching algorithms, respectively. Sections 5 and 6 explain metrics to analyze the performance of the algorithms and strategies to quantify the confidence level in the performance measured, respectively. Section 7 provides and discusses the results. Lastly, Sect. 8 concludes the paper.

2 Background

2.1 Related Work

Zaidman et al. [15] investigated the co-evolution of production code and test code using three projects (two open-source projects and one industrial project). They analyzed commit behavior, the relative growth of production code and test code, and test coverage evolution over time. The authors concluded that TDD can be detected as the simultaneous committing of production code and associated test code in a VCS.

Borle et al. [1] compared the productivity and code quality of GitHub projects that use TDD to varying degrees. They created a TDD or TDD-like set (experimental) and a non-TDD set (control). The degree of TDD variation was allowed in two ways: the difference in commit timestamps between production code and test code, and class coverage. The authors analyzed the commit velocity of test-production file pairs and found that only 0.8% of Java projects strictly practice TDD and that there is no observable benefit from using TDD.

Kochhar et al. [5] conducted a study to determine the presence or absence of test cases in 50,000 GitHub projects. They found that 57.34% of projects did not have any tests, and 90% of projects had fewer than 100 tests. Projects with tests were larger in size than those without, with a weak positive correlation between the size of a project and the number of tests. There was also a negative correlation between the size of a project and the number of tests per LoC.

Hanhela [3] investigated the use of the test-first approach in open-source projects. The author found that only 5% of test-production file pairs were "test-first", while 61% were "test-with" and 34% were "test-last". The majority of projects (74%) had at least one test file, and 52% had at least one test-production file pair committed either before or with the production file. The author concluded that since the majority of test files are committed with their corresponding production files, they were likely written around the same time, and the test-first approach is not widely used in open-source projects.

Wang et al. [13] investigated the co-evolution of production code and test code. They analyzed 975 open-source Java projects to determine the factors that affect whether test code should be updated. From the results, they proposed a

machine-learning-based approach, SITAR, to predict the test code that requires an update when its corresponding production code is modified. After evaluating SITAR on 20 popular Java projects, they found that it obtained an average precision and recall of up to 81.4% and 76.1%, respectively, in a within-project setting.

Van Rompaey et al. [12] evaluated six strategies that resolve traceability between production code and test cases, namely naming convention, fixture element types, static call graph, last call before assert, lexical analysis, and co-evolution. To determine the baseline for traceability, they requested developers to manually select units under test that link to a given set of test cases from three projects, namely JPacman, ArgoUML, and Mondrian. The performance of each strategy was then determined via applicability and accuracy by comparing the output to the baseline. They concluded that the strategy based on naming convention yielded high precision and recall, although they note that it may be project-dependent. The authors recommended combining the strategies to improve traceability resolution.

White et al. [14] presented TCTracker, an approach to establish test-to-code traceability at the method-level and at the class-level. TCTracker combines multiple static and dynamic techniques to address the shortcomings of using a single technique for test-to-code traceability. They evaluated TCTracker on a dataset of traceability links that was manually curated from four open-source projects. They found that TCTracker performs better than any individual technique in establishing traceability links between tests and production code.

2.2 Problem Statement

Zaidman et al. [15] used the similarity of filenames to match a single test file to a single production file. Borle et al. [1] used a regular expression that matches imports from testing frameworks to check if a repository includes test files. They matched one test file to only one production file by checking if their filenames differed by the case-insensitive string "test". Kochhar et al. [5] identified test files by evaluating whether a file's name contains the string "test". Hanhela [3] matched one test file to only one production file according to the similarity in their filenames, i.e., the test file's name starts with the name of the production file and ends with the word "Test". Wang et al. [13] paired a production class with its corresponding test class by checking if the test filename differed from the production filename only by the string "Test" in the test filename. Van Rompaey et al. [12] established that linking a test case to its corresponding production file(s) by checking if they differ by the string "Test" can yield high precision and recall, depending on project guidelines. White et al. [14] noted that the techniques based on naming conventions yielded low recall.

Many test files are left unpaired when undertaking a purely name-based approach when matching test files and production files. For instance, suppose a project has three production files called `Circle.java`, `Rectangle.java`, and `Triangle.java` and one test file called `PerimeterTest.java`, which tests the implementation of the perimeter function in each of the production classes. If

the traditional filename-based matching algorithm were used to match the three production files to the test file, it would fail since the filenames are not defined similarly in order to explicitly link `PerimeterTest.java` to the production files. A statement-based matching algorithm would overcome this problem as it would not depend on the filenames but rather on the classes being instantiated in the test file. Moreover, the strategy used by most researchers to match tests to production files does not take into account integration tests because integration tests involve testing multiple software units as a combined entity [11] and it is not clear which unit the integration test should be named after.

This paper will introduce a strategy for matching a test file to its target files at statement-level granularity by linking assert statements in test code to the public methods of production classes that are tested by the assert statements.

3 Matching Files via Filenames

The filename-based algorithm checks if a filename contains the string "test" to determine if it is a test file. It checks if the production filename is at least 80% similar to the test filename after all filenames are converted to lowercase and "test" is removed from the test filename. Python's `difflib.SequenceMatcher` is used to match strings given a threshold and is based on the Ratcliff-Obershelp algorithm [7]. The similarity ratio is given by `2 * M/T` where `M` is the sum of the matches and `T` is the sum of the lengths of the original strings. Refer to Fig. 1: The production file `AnnotationValidatorFactory` would be matched to the test file `AnnotationsValidatorTest` because the similarity ratio is equal to 2 * (10 + 9)/(20 + 26) which is equal to 0.82.

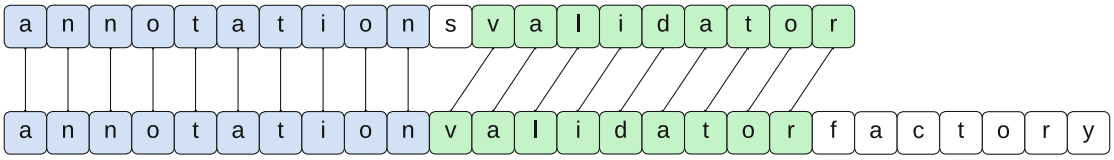


Fig. 1. Similarity between `annotationsalidator` and `annotationvalidatorfactory`

4 Matching Files via Assertions

The statement-based matching algorithm differs from the filename-based matching algorithm in that it analyzes source code line-by-line to match files, rather than matching files based on their filenames.

4.1 Distinguishing Between Tests and Production Files

PyDriller [8] is used to obtain a full list of production and test files in a project. To label a file as a test file, two factors are considered: whether the source code

includes at least one assert statement and whether there is at least one import from JUnit or TestNG, since they are the most popular testing frameworks [16]. The first regular expression in Listing 1.1 matches the assert statements used by the test frameworks. The second regular expression captures all the imports used in a file; the resultant list must be filtered by checking if the string "`junit`" or "`testng`" is part of the import, which would indicate that testing is performed in the file in question.

```
r"assert(Equals|That|ThatThrownBy|Same|NotSame|NotNull|Null
    |True|ArrayEquals|Throws|False|Timeout)\s*\("
r"import\s*.*;"
```

Listing 1.1. Regular expressions to determine whether a file is a test file

A file is considered a test file only if it has at least one match result for each of the regular expressions in Listing 1.1. Otherwise, it is considered a production file. The key subsystems utilized by the statement-based matching algorithm are depicted in Fig. 2 below.

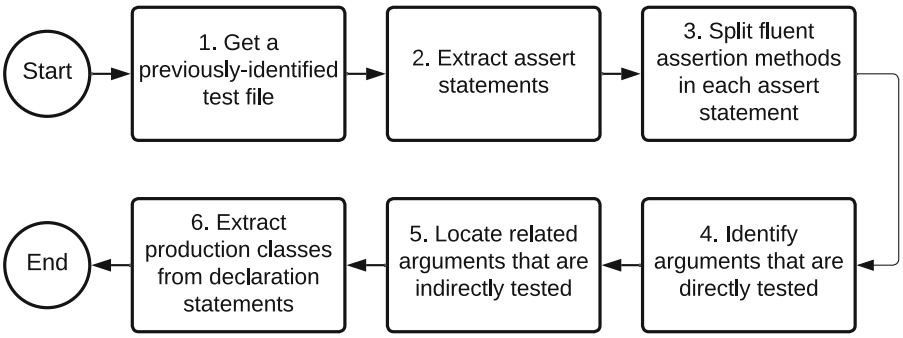


Fig. 2. Flow diagram of the statement-based matching algorithm.

4.2 Block 2: Extract Assert Statements

Block 2 takes test code as input and outputs a list of assert statements used in the test code. This block serves to isolate the parts of the test code that have a direct link to the production classes being tested. Assertion methods take objects of classes as arguments, and these could be direct instantiations of production classes or instantiations of third-party classes. A third-party class is a class that is provided by a third-party library or framework, which is a piece of software that is created and maintained by a separate organization or individual. Consider the test function in Listing 1.2, which verifies the behaviour of the add function from the Complex class from Apache Commons Math [10]. Given this test function, block 2 will return the array in Listing 1.3.

```java
@Test
public void testAddComplexNumbers() {
    Complex a = new Complex(3, 4);
    Complex b = new Complex(1, -2);
    Complex expectedResult = new Complex(4, 2);
    Complex result = a.add(b);

    assertEquals(expectedResult, result);
}
```

Listing 1.2. Testing the Complex class from Apache Commons-Math

```
[
    "assertEquals(expectedResult, result);"
]
```

Listing 1.3. Output of block 1 given Listing 1.2

4.3 Block 3: Split Fluent Assertion Methods in Assert Statements

A fluent assertion method is a method that allows one to chain multiple assertions together such that the entire assert statement is more human-readable and expressive [6]. The functions `isNotEmpty`, `hasSize` and `contains` in the assert statement in Listing 1.4 are examples of fluent assertion methods. Block 3 is responsible for splitting the assertion methods with the goal of isolating the arguments in each assertion method. It takes a list of assert statements obtained from block 2 as input and returns the split fluent assertion methods. The element in the array in Listing 1.4 is an example of an assert statement that uses fluent assertion methods. When it is passed to block 3, the output is given by Listing 1.5.

```
[
    "assertThat(numbers).isNotEmpty().hasSize(5).contains(4)
    ;"
]
```

Listing 1.4. An example of an assert statement that uses fluent assertion methods

```
[
    "assertThat(numbers)",
    "isNotEmpty()",
    "hasSize(5)",
    "contains(4)"
]
```

Listing 1.5. The output of block 3 given Listing 1.4 as input

Notice that the block isolates each fluent assertion method from the assert statement by splitting them into separate elements.

4.4 Block 4: Identify Directly-Tested Arguments

Block 4 identifies the arguments used in the split assertion methods. It takes the split assertion methods of an assert statement as input and returns the arguments within these methods because some of the arguments may be objects instantiated by production classes, thus providing a clear link between the test file and a production file. The array of elements in Listing 1.6 is an example of an input to block 4. The corresponding output is given by Listing 1.7. Notice that only the arguments that are valid object names (`firstList` and `secondList`) are returned, whereas primitive data types and methods invoked by objects (such as `9` and `getSize` respectively) are discarded.

```
[
    "assertThat(firstList)",
    "isNotEmpty()",
    "hasSize(secondList.getSize())",
    "contains(9)"
]
```

Listing 1.6. An example of split assertion methods and their arguments

```
[
    "firstList",
    "secondList"
]
```

Listing 1.7. Output of block 4 given Listing 1.6 as input

4.5 Block 5: Locate Directly and Indirectly Tested Objects

Block 5 is responsible for identifying *and* locating the positions and names of object references within the code that have dependencies on the object references that are directly tested in assert statements. By taking into account the names of objects passed directly into assert statements (the output of block 4), it conducts a comprehensive analysis of the test code to determine all relevant relationships between objects. To achieve this, block 5 pinpoints the connections between different objects in the code, as one object could be the return value of another object.

Block 5 takes two parameters as input and returns one entity. The two parameters are the test file parsed as text and all the object references that are directly tested. The output is a set of the very same directly tested object references as well as all the object references they depend on. This is achieved by splicing the declaration statements associated with each object reference and then extracting the object from which it was instantiated, if it exists. Note that for each newly discovered object, there may exist yet another object from which it is instantiated. Hence, this subsystem is executed recursively until all the object references (both directly and indirectly tested) are identified. Consider Listing

1.8 for an example of the action of block 5. Note that Line 7 in Listing 1.8 represents the continuation of the test in the same fashion, i.e., the creation of `objectC`, `objectD`, etc.

```
1  @Test
2  public void someTestFunction() {
3      ProductionClassA objectA = new ProductionClassA();
4
5      ProductionClassB objectB = objectA.someFunctionA(1,
       someParameterA);
6
7      // Test continues ...
8
9      ProductionClassZ objectZ = objectY.someFunctionY(
       someParameterX);
10
11     assertNotNull(objectZ);
12 }
```

Listing 1.8. A test function where the object being tested is dependent on other objects

```
1 [
2    "objectA", "objectB", ..., "objectY", "objectZ"
3 ]
```

Listing 1.9. Output of block 5 given Listing 1.8 and "objectZ" as input

Notice that although `objectZ` is the only object that is directly tested, all other objects that it depends on for its creation are also returned by block 5. This is because the test would have failed if there were any issues during the creation of the other objects. Hence, the other objects are considered to be tested, albeit indirectly.

4.6 Block 6: Extract Production Classes from Declaration Statements

Block 6 is responsible for extracting the production classes from the declaration statements. Before the extraction, it identifies the lines of code that contain the declaration statements for all the names of objects that are directly or indirectly tested in an assert statement. Block 6 takes as input a collection of tested objects (the output of block 5) and a complete set of production classes in a project. It returns a subset of the production classes that appear in at least one declaration statement.

To achieve this, block 6 gets the positions of the directly and indirectly tested objects in the parsed test file from block 5. Then, for each position, a part of the parsed test code is extracted via string slicing. The extracted substring is the declaration statement corresponding to the object whose position was provided. The start of each substring is the last newline character ("\n") before the given

position, and the end of each substring is the first semi-colon (";") after the given position. Lastly, each substring is analyzed to check for the presence of production class names.

Consider Listing 1.10: the `testPolynomialFunction` method examines the behavior of two distinct polynomial function classes, `PolynomialFunction` and `PolynomialFunctionLagrangeForm` from Apache Commons Math. Initially, a `PolynomialFunction` object named `func` is instantiated using an array of coefficients $\{1, -2, 1\}$, representing the polynomial $f(x) = (x-1)^2$. The method computes the value of this polynomial at $x = 2.0$, storing the result in the variable `resultOne`. Next, a `PolynomialFunctionLagrangeForm` object called `lagrange` is created, utilizing arrays of x-coordinates $\{-1, 0, 1\}$ and corresponding y-coordinates $\{2, 1, 0\}$. The method then calculates the value of the Lagrange polynomial at $x = 2.0$ and assigns it to the variable `resultTwo`. Finally, the test function employs assert statements to validate the correctness of both polynomial evaluations. Listings 1.11, 1.12 and 1.13 show the output of each block from 2 to 6 upon analyzing this test code. The positions of object names identified by block 5 are indicated in yellow in Listing 1.10.

```
1  public void testPolynomialFunction() {
2      double[] coeffs = {1, -2, 1};
3      PolynomialFunction func = new PolynomialFunction(
       coeffs);
4      double resultOne = func.value(2.0);
5
6      double[] x = {-1, 0, 1};
7      double[] y = {2, 1, 0};
8      PolynomialFunctionLagrangeForm lagrange = new
       PolynomialFunctionLagrangeForm(x, y);
9      double resultTwo = lagrange.value(2.0);
10
11     assertThat(resultOne).isEqualTo(1.0)
12                          .and(resultTwo).isEqualTo(-1.0);
13 }
```

Listing 1.10. Testing the behaviour of classes from Apache Commons-Math

```
1  ["assertThat(resultOne).isEqualTo(1.0).and(resultTwo).
       isEqualTo(-1.0);"] // block 2
2  ["assertThat(resultOne)", "isEqualTo(1.0)", "and(resultTwo)
       ", "isEqualTo(-1.0)"] // block 3
3  ["resultOne", "resultTwo"] // block 4
4  ["resultOne", "resultTwo", "func", "lagrange"] // block 5 -
       all object names associated with the test
5  [96, 268, 332, 433, 146, 344, 381, 158] // block 5 -
       corresponding positions of the object names
```

Listing 1.11. The output of block 2 - 5

```
[
    "double resultOne = func.value(2.0);",
    "PolynomialFunction func = new
    PolynomialFunction(coeffs);",
    "double resultTwo = lagrange.value(2.0);",
    "PolynomialFunctionLagrangeForm lagrange = new
    PolynomialFunctionLagrangeForm(x, y);"
]
```

Listing 1.12. Declaration statements extracted by block 6

```
[
     "PolynomialFunction",
     "PolynomialFunctionLagrangeForm"
]
```

Listing 1.13. Production classes identified by block 6

5 Performance Metrics: Precision, Recall and F1-Score

A true positive (TP) production file selection is when a production class is accurately identified by the algorithm in a test file, while a false positive (FP) production file selection is when the algorithm falsely reports that a production class is tested. A false negative (FN) production file selection is when a production class is tested by a test file, but the algorithm fails to identify it. The performance of the file matching algorithms on a single project is determined using precision (ρ), recall (λ) and F1-score (ψ) according to the expressions below:

$$\rho = \frac{TP}{TP + FP} \tag{1}$$

$$\lambda = \frac{TP}{TP + FN} \tag{2}$$

$$\psi = 2 \cdot \frac{\rho \cdot \lambda}{\rho + \lambda} \tag{3}$$

The ground truth (a set of production files that are truly tested) for a single test file was determined by manually analyzing each file and recording the tested production classes. The manual analysis was done by a single individual; a script that randomly samples one test file per execution was used. After every execution, the script also provided a subset of production filenames that appear in the test file (that may or may not be tested). The individual then manually looked through the code to determine which production files from the subset were actually tested by the test file. The TP, FP, and FN values were evaluated by running a script that compared the output of the algorithms to the ground truth. For a more general indication of the performance of the file

matching algorithms, multiple projects were analyzed. Hence, the weighted precision, weighted recall, and weighted F1-score were used to represent the overall performance, where the weight is determined by the number of test files in each project. If there are n projects with ω_i tests in the i^{th} project, then the weighted precision ($\bar{\rho}$), weighted recall ($\bar{\lambda}$), and weighted F1-score ($\bar{\psi}$) are given by the following expressions:

$$\bar{\rho} = \frac{\sum_{i=1}^{n} \omega_i \cdot \rho_i}{\sum_{i=1}^{n} \omega_i} \tag{4}$$

$$\bar{\lambda} = \frac{\sum_{i=1}^{n} \omega_i \cdot \lambda_i}{\sum_{i=1}^{n} \omega_i} \tag{5}$$

$$\bar{\psi} = \frac{\sum_{i=1}^{n} \omega_i \cdot \psi_i}{\sum_{i=1}^{n} \omega_i} \tag{6}$$

6 Bootstrap Confidence Intervals

The weighted F1-score was chosen to represent the overall performance of the matching algorithms. The greater the number of test files used to evaluate the performance of the matching algorithms, the greater the confidence level in the weighted F1-score, but it is impossible to collect an infinite number of test files. Hence, the bootstrap confidence interval for the weighted F1-score was chosen to quantify the level of certainty in the performance of the matching algorithms when a given sample size is used. It is a method for estimating the certainty in the weighted F1-score due to sampling variation by resampling the dataset with replacement and computing the weighted F1-score for each bootstrap sample. The confidence interval provides a range of likely values for the true population value of the weighted F1-score. There are three main steps for determining the confidence interval of a metric [2]:

1. Create a distribution based on K bootstrapped samples for the statistic u.
2. Sort the values in ascending order.
3. The lower bound of the 100 $(1-2\alpha)$ percent confidence interval is the K α^{th} value, and the upper bound is the K $(1-\alpha)^{th}$ value in the sorted distribution.

K represents the number of pseudosamples that are generated from the original sample, u represents the statistic of interest (e.g., mean), and α represents a variable that determines the confidence level (e.g., if α equals 0.025, then the confidence level is 95%). Bootstrap confidence intervals are valuable in this study as they inform the reader of the confidence in the reported performance of the algorithm.

7 Results and Discussion

The performance of the statement-based and filename-based matching algorithms was evaluated using 500 tests from sixteen open-source Java projects. Figure 3 shows the F1-scores of the two matching algorithms for each Java project. Table 1 shows the weighted performance metrics of the matching algorithms. Table 2 shows a summary of the statistics related to the F1-score of the Java projects.

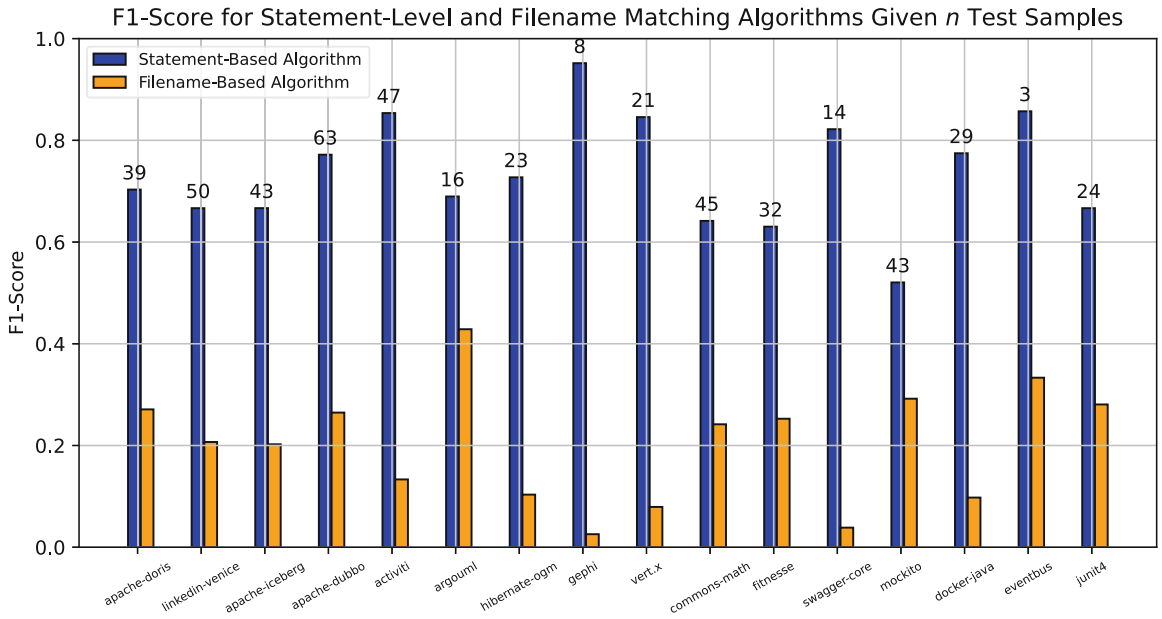


Fig. 3. F1-scores for the statement-based and filename-based matching algorithms given n test samples from different Java projects.

Table 1. Weighted performance metrics for the matching algorithms.

Matching Algorithm	Weighted Precision ($\bar{\rho}$)	Weighted Recall ($\bar{\lambda}$)	Weighted F1-Score ($\bar{\psi}$)	Weighted Accuracy
Statement-Based	0.8667	0.6114	0.7170	0.7427
Filename-Based	0.4947	0.1401	0.2180	0.3803
Performance Improvement	+0.3720	+0.4713	+0.4990	+0.3624

The statement-based matching algorithm's weighted precision (86.67%) is 37.2% points higher than the filename-based matching algorithm's weighted precision (49.47%). Practically, this means that the statement-based matching algorithm has a 37.2% higher likelihood of correctly identifying tested production files while minimizing the inclusion of untested production files. Similarly, the statement-based matching algorithm's weighted recall (61.14%) exceeds

Table 2. Summary of statistics across the 16 Java Projects

Matching Algorithm	Min	Max	Variance
Statement-Based	0.5206	0.9517	0.0121
Filename-Based	0.0256	0.4286	0.0129

the filename-based matching algorithm's weighted recall (14.01%) by 47.13% points. This indicates that the statement-based matching algorithm has a 47.13% higher likelihood of capturing tested production files, significantly reducing the chances of missing such files compared to the filename-based matching algorithm. Regarding the F1-score, the statement-based matching algorithm's weighted F1-score (71.70%) is 49.9% points higher than the filename-based matching algorithm's weighted F1-score (21.80%). This suggests that the statement-based matching algorithm has a 49.9% better balance between precision and recall, making it more effective at accurately identifying tested production files while minimizing both false positives and false negatives. In practical terms, these percentage differences imply that the statement-based matching algorithm is far more effective at matching test files to production files, resulting in more accurate and reliable outcomes if it is used for TDD research. However, it incurs higher spatial and time complexity, requiring more computer memory and operations, leading to a longer processing time.

Consider the performance of the algorithms on the project Gephi: the extreme performance difference between the algorithms may be due to the sampled tests being integration tests, as each test tests an average of ten production files; hence, the chosen test filename does not match any one particular production file. Since the statement-based matching algorithms analyzes source code instead of filenames, its performance is not affected in the way that the filename-based matching algorithm is affected. Notice the relatively poor performance of the statement-based matching algorithm with respect to Mockito. Mockito, as a mocking framework, is designed to simplify the process of creating mocks and stubs for unit testing [9], making it highly likely that its own tests rely on the features it provides, such as dependency injection, mocking, and stubbing. Mockito's own tests primarily consist of unit tests that focus on testing individual components in isolation, using mocks and stubs to replace external dependencies. Although some integration tests might be employed to ensure proper functioning of components when interacting with each other, the primary focus is on unit testing with mocks and stubs, allowing the Mockito development team to ensure the framework's reliability and correctness while maintaining a fast and efficient testing process. Since the statement-based matching algorithm's method of operation does not detect the use of mocks or stubs, this may explain its relatively poor performance with respect to Mockito.

The difference between the upper bound and lower bound values (expressed as a percentage) for the 95% confidence interval of the weighted F1-score for both matching algorithms is displayed in Fig. 4. Table 3 shows upper bound and lower bound values using a sample size of 10 and 500.

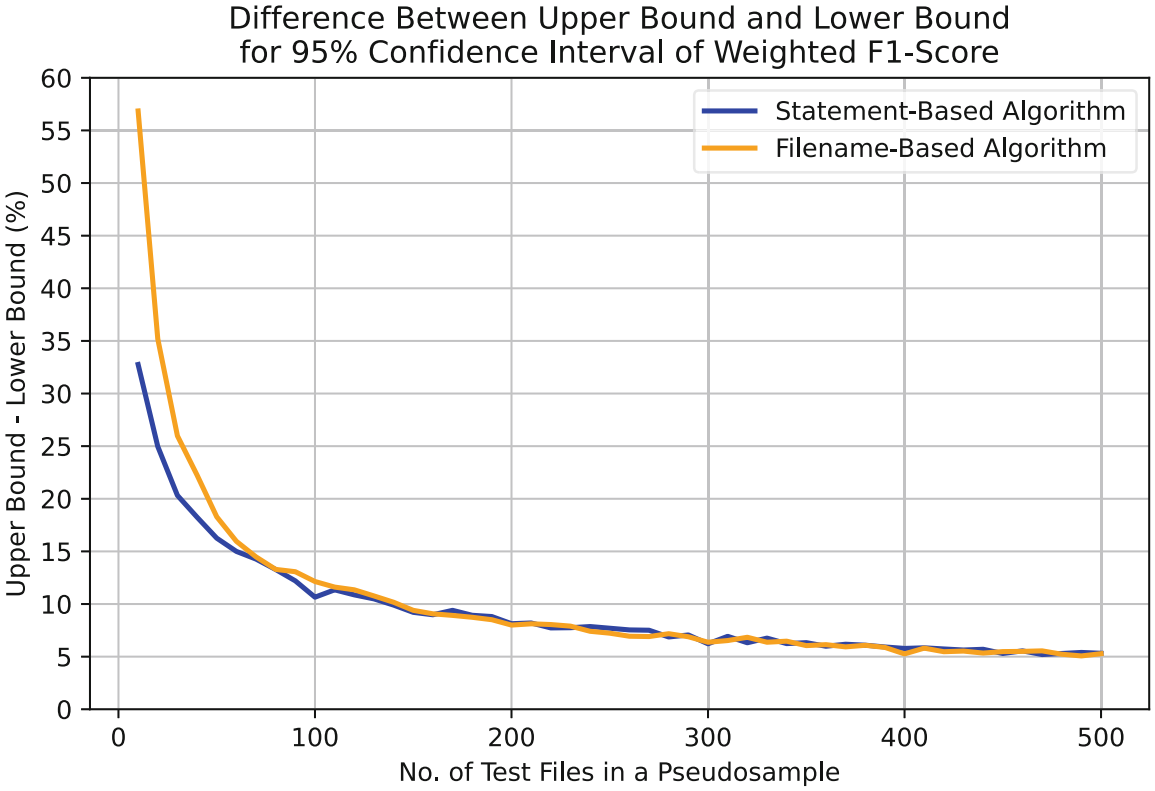


Fig. 4. Difference between the upper bound and lower bound for the 95% confidence interval of the weighted F1-score for the matching algorithms

Table 3. Upper bound and lower bound values for the weighted F1-score for the matching algorithms for 10 test samples and 500 test samples

Matching Algorithm	$\bar{\psi}_{upper}$ $(n = 10)$	$\bar{\psi}_{lower}$ $(n = 10)$	$\bar{\psi}_{upper}$ $(n = 500)$	$\bar{\psi}_{lower}$ $(n = 500)$
Statement-Based	0.8952	0.5676	0.7347	0.6815
Filename-Based	0.7501	0.1816	0.2459	0.1931

Pseudosample sizes ranging from 10 to 500 in increments of 10 were used to generate the plot. At a pseudosample size of 500, the 95% confidence interval of the statement-based matching algorithm is [0.6815, 0.7347] and that of the filename-based matching algorithm is [0.1931, 0.2459]. This suggests that, regardless of the specific sample size used, the statement-based matching algorithm is more likely to provide more accurate matches between test files and production files due to its higher weighted F1-score confidence interval. Moreover, the differences in the lower bounds (0.4884) and the upper bounds (0.4888) indicate a substantial performance gap in favor of the statement-based matching algorithm. This consistent superiority of the statement-based matching algorithm's confidence intervals suggests that it is more favourable to use for matching test files to production files in most scenarios.

7.1 Threats to Validity

This study may have limitations due to a sample size of 500 tests from sixteen projects, although this was mitigated by the use of the bootstrap confidence interval. The implementation of the filename-based algorithm from prior literature could not be obtained; there could be variations in the performance between the algorithm implemented for this study and the algorithm in prior research. The manual analysis of test files to determine which production classes are tested may have also introduced human error. To reduce the manual labour and the chance of human error in determining the production classes that are tested, a script was used to simply list the production classes that appear anywhere in the test code. This reduced the likelihood of incorrect classification of tested production classes. The statement-based matching algorithm may be affected when applied to new datasets that rely heavily on mocks or stubs, as it was not designed to consider the presence of mocks or stubs. The external validity of the findings may be constrained because of the use of open-source Java projects since software projects written in different languages or in different contexts may have different testing or programming styles. TDD involves writing tests prior to production code, and the statement-based matching algorithm can be used to detect the use of TDD. However, other aspects, such as the time difference between committing tests and production files, need to be taken into account to accurately detect the use of TDD. Hence, the statement-based matching algorithm alone may not accurately assess whether TDD is being practiced since it analyzes only one aspect of TDD.

8 Conclusion

This paper has proposed a statement-based matching algorithm. It consists of six blocks: block 1 gets a previously-identified test file; block 2 extracts assert statements; block 3 splits fluent assertions; block 4 identifies directly tested arguments; block 5 identifies directly and indirectly tested objects; and block 6 extracts production classes from declaration statements. The statement-based matching algorithm was shown to be more accurate, as it has a weighted F1-score of 0.7170 and a 95% confidence interval of [0.6815, 0.7347], whereas the filename-based matching algorithm has a weighted F1-score of 0.2183 and a 95% confidence interval of [0.1931, 0.2459]. The improved accuracy will benefit future TDD research as it can be used as a more accurate way to check if a project has used TDD, increasing the validity of the conclusions drawn by researchers. Future work will extend the statement-based matching algorithm to detect static classes in addition to instance classes, thereby improving recall.

References

1. Borle, N., Feghhi, M., Stroulia, E., Grenier, R., Hindle, A.: Analyzing the effects of test driven development in GitHub. In: 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE), pp. 1062–1062. IEEE (2018)
2. Cohen, P.R.: Empirical Methods for Artificial Intelligence, vol. 139. MIT Press, Cambridge (1995)
3. Hanhela, J.: Usage of test-driven development in open source projects. Ph.D. thesis, Master's thesis Jari Hanhela Spring (2015)
4. Janzen, D., Saiedian, H.: Test-driven development concepts, taxonomy, and future direction. Computer **38**(9), 43–50 (2005)
5. Kochhar, P.S., Bissyandé, T.F., Lo, D., Jiang, L.: Adoption of software testing in open source projects-a preliminary study on 50,000 projects. In: 2013 17th European Conference on Software Maintenance and Reengineering, pp. 353–356 (2013). https://doi.org/10.1109/CSMR.2013.48
6. Leotta, M., Cerioli, M., Olianas, D., Ricca, F.: Fluent vs basic assertions in java: an empirical study. In: 2018 11th International Conference on the Quality of Information and Communications Technology (QUATIC), pp. 184–192. IEEE (2018)
7. Ratcliff, J.W., Metzener, D.E.: Pattern-matching-the gestalt approach. Dr Dobbs J. **13**(7), 46 (1988)
8. Spadini, D., Aniche, M., Bacchelli, A.: Pydriller: python framework for mining software repositories. In: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 908–911 (2018)
9. Spadini, D., Aniche, M., Bruntink, M., Bacchelli, A.: To mock or not to mock? an empirical study on mocking practices. In: 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR), pp. 402–412. IEEE (2017)
10. The Apache Software Foundation: Apache Commons Math GitHub Repository. https://github.com/apache/commons-math. Accessed 2022
11. Thomas, D., Hunt, A.: Mock objects. IEEE Softw. **19**(3), 22–24 (2002). https://doi.org/10.1109/MS.2002.1003449
12. Van Rompaey, B., Demeyer, S.: Establishing traceability links between unit test cases and units under test. In: 2009 13th European Conference on Software Maintenance and Reengineering, pp. 209–218. IEEE (2009)
13. Wang, S., Wen, M., Liu, Y., Wang, Y., Wu, R.: Understanding and facilitating the co-evolution of production and test code. In: 2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 272–283. IEEE (2021)
14. White, R., Krinke, J., Tan, R.: Establishing multilevel test-to-code traceability links. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, pp. 861–872 (2020)
15. Zaidman, A., Rompaey, B., Deursen, A., Demeyer, S.: Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining. Empirical Softw. Eng. **16**, 325–364 (2011). https://doi.org/10.1007/s10664-010-9143-7
16. Zerouali, A., Mens, T.: Analyzing the evolution of testing library usage in open source java projects. In: 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 417–421. IEEE (2017)