

UNIVERSIDADE FEDERAL DE JUIZ DE FORA
INSTITUTO DE CIÊNCIAS EXATAS
PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Heleno de Souza Campos Junior

**A framework for test case prioritization in the
continuous software engineering**

Juiz de Fora

2018

UNIVERSIDADE FEDERAL DE JUIZ DE FORA
INSTITUTO DE CIÊNCIAS EXATAS
PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Heleno de Souza Campos Junior

**A framework for test case prioritization in the
continuous software engineering**

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação, do Instituto de Ciências Exatas da Universidade Federal de Juiz de Fora como requisito parcial para obtenção do título de Mestre em Ciência da Computação.

Orientador: Marco Antônio Pereira Araújo

Juiz de Fora

2018

Ficha catalográfica elaborada através do programa de geração automática da Biblioteca Universitária da UFJF, com os dados fornecidos pelo(a) autor(a)

Campos Junior, Heleno de Souza.

A framework for test case prioritization in the continuous software engineering / Heleno de Souza Campos Junior. -- 2018.

129 f.

Orientador: Marco Antônio Pereira Araújo

Dissertação (mestrado acadêmico) - Universidade Federal de Juiz de Fora, Instituto de Ciências Exatas. Programa de Pós Graduação em Ciência da Computação, 2018.

1. Engenharia de Software. 2. Teste de Software. 3. Integração contínua. 4. Manutenção de software. I. Araújo, Marco Antônio Pereira, orient. II. Título.

Heleno de Souza Campos Junior

**A framework for test case prioritization in the continuous
software engineering**

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação, do Instituto de Ciências Exatas da Universidade Federal de Juiz de Fora como requisito parcial para obtenção do título de Mestre em Ciência da Computação.

Aprovada em 19 de Setembro de 2018.

BANCA EXAMINADORA

Prof. D.Sc. Marco Antônio Pereira Araújo - Orientador
Universidade Federal de Juiz de Fora

Prof. D.Sc. Rodrigo Oliveira Spínola
Universidade Salvador

Prof. D.Sc. Victor Ströele de Andrade Menezes
Universidade Federal de Juiz de Fora

ACKNOWLEDGMENTS

I'd like to thank everybody that contributed to this work.

CAPES.

UFJF.

Professors of PGCC.

Reviewers of SBES and WTDSOFT 2017 conferences.

Colleagues from NEnC, our lab.

Colleagues and friends from IF Sudeste MG, my undergrad institute.

Reviewers of this work for their contribution, Rodrigo Spínola and Victor Ströele.

In special, I would like to thank my advisor Marco Antônio, for not measuring efforts when needed and always providing me the best opportunities. Also, for his belief in my work and allowance to follow the path I wanted.

Camila Paiva, for sharing ideas and always supporting my work.

Maria Luiza Falci, my partner in life, for always being by my side, supporting me in every possible aspect.

Finally, my family, for providing all the necessary environment and support that allowed me to continue studying.

"Wyrd bið ful aræd"

Anonymous author

The Wanderer

The Exeter Book

RESUMO

Testes de regressão são executados após cada mudança no software. Em ambientes de desenvolvimento de software que adotam práticas da Engenharia de Software Contínua, como a Integração contínua, por exemplo, software é modificado, e testado diversas vezes em curtos prazos. Cada execução dos testes pode levar horas para terminar, gerando atraso em relação à descoberta de falhas no projeto. Para prevenir esse atraso, técnicas de otimização são utilizadas. Uma delas é a priorização de casos de testes (TCP). Nessa técnica, a execução dos testes é reordenada de acordo com um objetivo, que normalmente é a detecção de falhas. Dessa forma, testes que têm maior probabilidade de falhas são executados primeiro. Um problema com essa abordagem é que existem diversas técnicas na literatura, mas pouca evidência em relação ao seu uso. Além disso, quase não existe infra estrutura para apoiar a adoção dessas técnicas no contexto industrial. O objetivo deste trabalho é planejar e implementar um framework que permita o uso, experimentação e implementação de técnicas de TCP. Esperamos que isso ajude praticantes a adotar essas técnicas no contexto industrial, principalmente da engenharia de software contínua. Esperamos também que a criação dessa infra estrutura ajude pesquisadores a executar mais estudos experimentais sobre a eficiência do uso dessas técnicas. Para mostrar a viabilidade do framework proposto, é executado um estudo experimental com 16 técnicas de priorização diferentes, executadas em um total de 22 versões de 2 projetos open source. Os resultados coletados sugerem que o uso das técnicas de priorização resultam em retornos mais rápidos em relação à existência de falhas nos projetos, possivelmente resultando em ciclos mais rápidos de desenvolvimento.

Palavras-chave: Manutenção de Software. Engenharia de Software. Teste de Software. Integração Contínua.

ABSTRACT

Regression tests are executed after every change in software. In a software development environment that adopts Continuous Software Engineering practices such as Continuous Integration, software is changed, built and tested many times in a short period. Each execution can take hours to finish, delaying feedback about failures to the developer. To prevent this, regression test optimization techniques are used. One such technique is test case prioritization (TCP), which reorder the execution of the test cases according to some goal. The most common goal is fault detection, in which test cases are ordered so that those that have higher probability of detecting faults are executed first. One problem with this approach is that there are lots of different available techniques in the literature, but the amount of evidence of its use is low. Furthermore, there is almost no infrastructure support to adopt those techniques at the industry context. The goal of this work is to design and implement a framework that allows the use, experimentation and implementation of TCP techniques. We hope that this will help practitioners on adopting these techniques at the industry context, more specifically, in the continuous software engineering environment. We also hope that creating this infrastructure will encourage researchers on performing more empirical studies regarding test case prioritization techniques effectiveness. In order to show the feasibility of the proposed framework, we perform an empirical study with 16 different TCP techniques executed on a total of 22 versions of 2 different open source projects. Results suggest that using those TCP techniques result in faster feedback about the existence of failures in the projects, possibly resulting in shorter development cycles.

Keywords: Software Maintenance. Software Engineering. Software Testing. Continuous Integration.

LIST OF FIGURES

4.1	Typical continuous integration process.	41
4.2	Optimus framework architecture.	42
4.3	Example software project.	45
4.4	TCP techniques API for Optimus Framework.	79
5.1	Optimus Framework implemented modules.	81
5.2	Sequence diagram of test process within Optimus Framework.	82
5.3	Fault seeding process in Optimus Framework.	85
5.4	Historical analyzer relational database model.	86
5.5	Example configuration of Optimus Framework.	89
5.6	Example run of Optimus Framework.	89
5.7	Information of example run of Optimus Framework.	92
5.8	Generated summary report for example run of Optimus Framework.	93
5.9	Generated raw report for example run of Optimus Framework.	93
5.10	Execution order in raw report for example run of Optimus Framework.	93
6.1	Example raw report generated from experiment run.	99
6.2	Example of Optimus Framework configuration to run the experiment.	100
6.3	Boxplot of APFD values obtained for CoreNLP project.	101
6.4	Normality test for T0 from CoreNLP.	103
6.5	Normality test for T1-C from CoreNLP.	105
6.6	Normality test for T1-M from CoreNLP.	106
6.7	Homoscedasticity test for T1-C and T1-M from CoreNLP.	107
6.8	Boxplot of APFD values obtained for Jackson-databind project.	109
6.9	Normality test for T0-C from Jackson-databind.	111
6.10	Normality test for T1-C from Jackson-databind.	111
6.11	Homoscedasticity test for T0-C and T1-C from Jackson-databind project.	112
6.12	Normality test for T1-C from Jackson-databind.	115
6.13	Normality test for T1-M from Jackson-databind.	116
6.14	Homoscedasticity test for T1-C and T1-M from Jackson-databind.	116

LIST OF TABLES

2.1	Rank of authors per number of publications.	22
2.2	Papers amount per publication venue.	23
2.3	Most investigated TCP approaches.	24
2.4	Test coverage matrix.	25
2.5	Most used TCP effectiveness metrics.	27
2.6	Tools used for each different activity in selected TCP primary studies.	28
2.7	Summary of best APFD achieved on each application used by selected papers on their empirical evaluations.	30
2.8	Possible APFD factors.	30
2.9	Summary of our relevant factors findings.	31
2.10	Amount of papers queried.	32
2.11	Literature evidence of TCP use in industrial settings.	34
3.1	Comparison of requirements among related work.	39
4.1	Variations of Total and Additional coverage techniques.	46
4.2	Variations of ART coverage technique.	47
4.3	Coverage matrix for the example project.	48
4.4	Total coverage for the example application.	49
4.5	Additional coverage for example application, first iteration.	52
4.6	Additional coverage for example application, second iteration.	52
4.7	Additional coverage for example application, third iteration.	53
4.8	Additional coverage for example application, fourth iteration.	53
4.9	Additional coverage for example application, fifth iteration.	55
4.10	Coverage strings for test cases.	55
4.11	Intersection calculation for Jaccard.	55
4.12	Union calculation for Jaccard.	56
4.13	Jaccard distances to ordered set, first iteration	56
4.14	Jaccard distances to ordered set, second iteration	57
4.15	Jaccard distances to ordered set, third iteration	57

4.16	Variations for history-based TCP techniques.	58
4.17	Historical results matrix.	58
4.18	Amount of historical failures per test case.	60
4.19	Recent failures score of test cases.	62
4.20	Variations of modification-based TCP techniques.	62
4.21	Total diff score for test cases of the example application.	66
4.22	Additional diff score for test cases of the example application, first iteration.	67
4.23	Additional diff score for test cases of the example application, second iteration.	67
4.24	Additional diff score for test cases of the example application, third iteration.	67
4.25	Additional diff score for test cases of the example application, fourth iteration.	68
4.26	Variations of similarity-based techniques.	68
4.27	Execution profile for test cases.	68
4.28	Frequency profile for test cases.	69
4.29	Ordered sequence for test cases.	69
4.30	Levenshtein distance value between test cases.	72
4.31	Minimum distances between test cases and ordered set, first iteration.	72
4.32	Minimum distances between test cases and ordered set, second iteration.	74
4.33	Minimum distances between test cases and ordered set, third iteration.	74
4.34	Test cases clusters, first iteration.	74
4.35	Test cases clusters distances, first iteration.	75
4.36	Test cases clusters, second iteration.	75
4.37	Clusters frequency profile, second iteration.	75
4.38	Clusters ordered sequences, second iteration.	76
4.39	Test cases clusters distances, second iteration.	76
4.40	Test cases clusters, third iteration.	76
4.41	Clusters frequency profile, third iteration.	77
4.42	Clusters ordered sequences, third iteration.	77
4.43	Test cases clusters distances, third iteration.	78
4.44	Test cases clusters, fourth iteration.	78
5.1	Allowed configurations for Optimus Framework.	90
6.1	Experiment projects information.	97

6.2	TCP techniques used in the experiment.	97
6.3	Comparisons for RQ1 of CoreNLP experiment.	102
6.4	Hypotheses test results for RQ1 of CoreNLP.	104
6.5	Comparisons for RQ2 of CoreNLP experiment.	105
6.6	Normality and homoscedasticity analyses for RQ2 of CoreNLP.	108
6.7	Hypothesis tests results for RQ2 of CoreNLP.	108
6.8	Comparisons for RQ1 of Jackson-databind experiment.	110
6.9	Normality and homoscedasticity analyses for RQ1 of Jackson-databind.	113
6.10	Hypothesis tests results for RQ1 of Jackson-databind.	113
6.11	Comparisons for RQ2 of Jackson-databind experiment.	114
6.12	Hypothesis tests results for RQ2 of Jackson-databind.	117

LIST OF ACRONYMS

TCP Test Case Prioritization

MQ Mapping Question

RQ Research Question

APFDAverage Percentage of Faults Detected

SIR Software Infrastructure Repository

LOC Lines Of Code

TSL Test Script Language

RTORegression Testing Optimization

API Application Programming Interface

ART Adaptive Random Testing

FOS Farthest-first Ordered Sequence

GOS Greed-aided-clustering Ordered Sequence

CI Continuous Integration

PDF Portable Document Format

GQM Goal Question Metric

JDK Java Development Kit

KLOC Thousands of Lines Of Code

CONTENTS

1	INTRODUCTION	15
1.1	RESEARCH QUESTIONS	16
1.2	GOALS	16
1.3	RESEARCH METHODOLOGY	17
1.4	OUTLINE	17
2	BACKGROUND	18
2.1	INTRODUCTION	18
2.2	CONTINUOUS SOFTWARE ENGINEERING	18
2.3	SOFTWARE TESTING	19
2.4	REGRESSION TESTING	19
2.5	REGRESSION TESTING OPTIMIZATION	20
2.6	TEST CASE PRIORITIZATION	20
2.7	SYSTEMATIC REVIEW AND MAPPING OF THE LITERATURE	21
2.7.1	Systematic mapping results	22
2.7.1.1	Coverage-based TCP techniques	24
2.7.1.2	History-based techniques	25
2.7.1.3	Modification-based techniques	26
2.7.1.4	Similarity-based techniques	26
2.8	SYSTEMATIC REVIEW RESULTS	29
2.9	INDUSTRIAL ADOPTION	31
2.10	FINAL CONSIDERATIONS	32
3	RELATED WORKS	35
3.1	TCP USAGE	35
3.2	TCP IN CONTINUOUS SOFTWARE ENGINEERING	36
3.3	TCP USAGE AND CONTINUOUS SOFTWARE ENGINEERING	36
3.4	EXISTING APPROACHES COMPARISON	37
3.5	FINAL CONSIDERATIONS	39

4	OPTIMUS FRAMEWORK DESIGN	40
4.1	INTRODUCTION	40
4.2	REQUIREMENTS	40
4.3	ARTIFACTS REPOSITORY	41
4.4	EXPERIMENTS SUPPORT	41
4.4.1	TCP Effectiveness Analyzer	42
4.4.2	Reports generator	43
4.5	ANALYZERS	43
4.5.1	Coverage analyzer	43
4.5.2	Historical data analyzer	43
4.5.3	Execution trace analyzer	44
4.5.4	Modifications analyzer	44
4.6	TCP TECHNIQUES	44
4.6.1	Coverage-based	44
4.6.1.1	Total coverage	48
4.6.1.2	Additional coverage	48
4.6.1.3	Adaptive Random Testing	52
4.6.2	History-based	57
4.6.2.1	Most failures first	58
4.6.2.2	Recent failures first	60
4.6.3	Modification-based	60
4.6.3.1	Total diff coverage	63
4.6.3.2	Additional diff coverage	63
4.6.4	Similarity-based	66
4.6.4.1	Fartherst-first ordered sequence	69
4.6.4.2	Greed-aided-clustering ordered sequence	72
4.7	NEW TECHNIQUES IMPLEMENTATION	77
4.8	FINAL CONSIDERATIONS	78
5	OPTIMUS FRAMEWORK IMPLEMENTATION	80
5.1	OPTIMUS-COMMON	80
5.2	OPTIMUS-FRAMEWORK	80
5.3	OPTIMUS-TEST	81

5.4	OSS FAULTS FINDER	82
5.5	FAULT INJECTION PLUGIN	82
5.6	OPTIMUS COVERAGE ANALYZER	84
5.7	OPTIMUS HISTORICAL ANALYZER	85
5.8	OPTIMUS MODIFICATION ANALYZER	86
5.9	OPTIMUS EXECUTION TRACE ANALYZER	87
5.10	OPTIMUS FRAMEWORK USAGE	87
5.11	FINAL CONSIDERATIONS	91
6	EVALUATION.....	94
6.1	INTRODUCTION	94
6.2	EXPERIMENTAL STUDY	94
6.2.1	Objects of analysis	95
6.2.2	Variables	96
6.2.3	Experiment setup	98
6.2.4	Data and analysis	99
6.2.4.1	CoreNLP	99
6.2.4.2	Jackson-databind	109
6.2.5	Threats to validity	118
6.3	DISCUSSION AND LESSONS LEARNED	118
6.4	FINAL CONSIDERATIONS	119
7	CONCLUSION	121
7.1	RESEARCH LIMITATIONS	122
7.2	FUTURE WORKS	122
	REFERENCES	123

1 INTRODUCTION

Regression tests are executed after software modification in order to ensure that previously developed software parts are working and that newly developed source code behaves like it is supposed to. To cope with market's necessity of rapid software deliveries, software development companies have been widely adopting agile practices (FITZGERALD; STOL, 2017).

However, adopting those practices only at the development and operational level usually is not enough. It is also necessary to integrate the business strategy level in order to achieve a continuous improvement of the development processes. This integration is named Continuous Software Engineering (FITZGERALD; STOL, 2017).

Practices of Continuous Software Engineering include Continuous Integration, which is concerned with merging every modification made on the software into the main development branch. In a rapid development environment, a big amount of software updates is delivered in a short time and each modification of the software implies the execution of regression tests. Considering that the baseline approach of regression testing is to execute every test case of the software, this scenario can result in a bottleneck in the development process, since developers need to wait the completion of the regression test execution in order to receive feedback about possible failures. Examples available in the literature report regression test suites that take around 1000 machine hours to finish execution (DO et al., 2010).

Regression tests optimization techniques are often used to solve the problem. They are divided into three categories (YOO; HARMAN, 2012), minimization or reduction, selection and prioritization. While minimization and selection techniques focus on a subset of test cases to be executed, prioritization techniques still execute all test cases, but in a specific order. For this reason, prioritization techniques are sometimes more reliable and cost-effective (DO et al., 2010).

Test case prioritization (TCP) techniques modify the execution order of the test cases according to some criterion. One common criterion is fault detection, where the goal of such techniques is to detect the maximum amount of faults by executing the minimum amount of test cases. In this way, if, for some reason, the execution of the test cases is

interrupted before finishing, a high amount of the faults will have already been detected. Different approaches to prioritize test cases are used by those techniques. Some examples include historical failure data, test coverage and software modifications (YOO; HARMAN, 2012).

It is shown in this work that current challenges faced by researchers and practitioners interested in using TCP techniques are motivated by the high amount of different available techniques in the literature, which use different input information that can not be trivially obtained. These inputs are not always available in software projects, requiring the setup of additional tools to gather and record them.

Furthermore, it is also shown that there is a low amount of empirical evidence supporting the use of TCP techniques, which may hinder its potential benefits for different contexts when practitioners need to choose an approach to employ in industrial applications.

1.1 RESEARCH QUESTIONS

To further explore the TCP research area, this work aim at analyzing the literature to gather evidences of the above-mentioned problems. Based on these evidences, to propose and evaluate a solution to support the TCP usage in software projects. The following questions were formulated to guide the research:

- Q1: How is TCP being used in industry and literature?
- Q2: How to create a framework to support TCP usage and experimentation?
- Q3: How does the resulting framework support practitioners and researchers?

1.2 GOALS

The main goal of this work is to propose a framework to support the use and experimentation of TCP techniques. This goal can be divided in specific goals, according to the research questions. Those specific goals are:

- G1: to establish a systematic mechanism to analyze how TCP is being used in the literature.

- G2: to develop a framework able to execute TCP techniques and measure its effectiveness.
- G3: to evaluate TCP techniques on open source projects using the developed framework.

1.3 RESEARCH METHODOLOGY

Methodology followed in this work includes the planning, execution and reporting of a systematic literature review and mapping, following Kitchenham (2007) guidelines, in order to gather knowledge about the studied topic. Based on results of the review and mapping, the main proposal is designed and implemented. The resulting product is then subjected to experimental studies, following guidelines provided in Wohlin et al. (2012). Generated evidence from all the steps are used to answer the research questions.

1.4 OUTLINE

The remainder of this work is organized as follows: Chapter 2 describes theoretical foundations about software testing, test case prioritization and continuous software engineering. A systematic literature review and mapping is also presented. In Chapter 3 studies related to this work are discussed. Based on the foundations and findings from Chapter 2 and gaps identified from related works, a framework is designed in Chapter 4. Chapter 5 describes the implementation of such framework, which is evaluated through experimental studies in Chapter 6 and finally in Chapter 7, final considerations, research limitations and future works are presented.

2 BACKGROUND

2.1 INTRODUCTION

In this chapter, the main concepts that will be discussed and used in this work are explained. Initially, we introduce the concepts of Continuous Software Engineering, which is the development environment where test case prioritization is analyzed in this work. Then we discuss the use of software testing in this scenario, more specifically, test case prioritization during regression testing.

A systematic literature review and mapping is reported, discussing the main aspects of the state-of-the-art regarding this topic. Then a structured search is presented, aiming to report the state-of-practice of the topic in industry.

2.2 CONTINUOUS SOFTWARE ENGINEERING

Software development scenario is going through many changes in the last years. Part of these changes is to cope with market's necessity of rapid software deliveries. Due to that, software development companies have been adopting agile practices. However, adopting practices that only change the development or operational level of companies is usually not enough. Fitzgerald and Stol (2017) argue that a holistic view over the processes of a company is needed in order to achieve a continuous improvement cycle. This is named Continuous Software Engineering, which includes different continuous activities related to development, operations and business strategy that combined, result in continuous improvement of a company's processes (FITZGERALD; STOL, 2017).

Among the activities in this scenario, Continuous Integration is concerned with merging development branches as developers commit them to the source code repository, instead of keeping different active branches simultaneously. In order to make continuous integration happen, build automation is essential to minimize human failures. Besides automation, developers' culture change is also necessary, since the responsibility of having every commit integrated into the main branch is critical to the success of the process (MEYER, 2014).

Evolving from continuous integration, continuous delivery ensures that the software is

ready to be delivered at any given time (CHEN, 2015). To make this happen, a pipeline is used, which contains steps that a software modification has to go through in order to be ready to be delivered. This delivery is not always to the production environment. The stage at which a company is able to automatically deliver to the production environment is named continuous deployment. To reach this stage, all development activities need to be well executed and coordinated. Normally, these continuous practices are adopted incrementally.

A software process that is within these development related activities is software testing. The continuous activity related to software testing is often referred to as continuous testing. In this activity, feedback from the execution of software tests is used to improve the next executions, aiming to achieve a continuous improvement of the testing process (FITZGERALD; STOL, 2017).

2.3 SOFTWARE TESTING

The process of verifying that the software does not have errors is called software testing (MYERS et al., 2011). This process has been approached in different ways. For example, black-box strategies group testing techniques that do not require access or analysis of the source code. In the other hand, white-box strategies are concerned with the structure of the program, and thus, require access to it.

White-box strategies include unit testing, the process of testing the correctness of the smallest parts of a program, called units, and integration testing, which is concerned with testing the interaction of different parts of a program.

A common misconception about testing definitions often occurs. According to Ammann and Offutt (2016), a software fault is a static defect in the software. The manifestation of faults is called a software error. A software failure is an incorrect behavior according to the expectations with the software. It is common to find the term software bug referring to all of these three definitions.

2.4 REGRESSION TESTING

Modifications are performed in the software to correct defects, to add functionalities, to adapt it to different environments or even to maintain and evolve its structure aim-

ing to facilitate future modifications. Those activities are named software maintenance (PFLEEGER; ATLEE, 2009). For every maintenance task, there is a high probability that new defects are added, which are called regressions, implying that the software lost quality (DO, 2016). Then, the main objective of regression tests is to ensure that after maintenance is performed into the software, no regressions occurred.

A traditional regression test suite is composed of all previously developed tests for the software. The approach of using this traditional test suite is named Retest-All, being the default practice nowadays (DO, 2016). The problem with this approach is that the regression test execution can be very costly in terms of computational resources, requiring long time to finish. The continuous integration process aggravates even more this, because the software is regression-tested many times during the day. To solve this problem, regression test optimization techniques are used.

2.5 REGRESSION TESTING OPTIMIZATION

Regression test optimization techniques can be divided into three types. Minimization or Reduction techniques aim at reducing the size of the test suite by excluding redundant test cases. Selection techniques aim at selecting a specific group of test cases from the test suite to be executed. Prioritization techniques aim at reordering the execution sequence of the test cases, such that those test cases that might reveal a failure are executed first (DO, 2016).

While minimization and selection techniques focus on a sub-set of test cases to be executed, prioritization techniques still execute all test cases, but in a specific order. For this reason, prioritization techniques are sometimes more reliable and cost-effective (DO et al., 2010).

2.6 TEST CASE PRIORITIZATION

Test case prioritization (TCP) problem is defined by Rothermel et al. (2001).

Definition 1. Given a test suite T , a permutation set of T named PT and f as a function from PT to the real numbers.

Find $T' \in PT$ such that $(\forall T'')(T'' \in PT)(T'' \neq T')[f(T') \geq f(T'')]$.

In this way, according to their definition, PT is the set of all possible orderings of T and f is a function that when applied to any ordering yields an award value for that ordering.

In simpler words, TCP techniques reorder the execution order of the test cases according to some criterion. One common criterion is fault detection, where the goal of the technique is to detect the maximum amount of faults by executing the minimum amount of test cases. In this way, if, for some reason, the execution of the test cases is interrupted before finishing, a high amount or all faults have already been detected. Different approaches to prioritize test cases are used by those techniques. Some examples include historical failure data, test coverage, requirements and system models (YOO; HARMAN, 2012).

As an example, a simple technique that relies on test coverage information, order test cases according to the descending amount of source code elements that each test case cover. The assumption is that test cases that cover a higher amount of source code have a greater probability of revealing a fault. In this way, it should be executed first than others.

2.7 SYSTEMATIC REVIEW AND MAPPING OF THE LITERATURE

Systematic Literature Mapping (SLM) is a type of secondary study, with the goal of surveying the literature to map knowledge of a particular topic. Systematic Literature Review (SLR) has a more specific goal, aiming to investigate research questions more deeply (KITCHENHAM, 2007).

Based on Kitchenham (2007) guidelines, we conducted a systematic literature review and mapping to give an overview of the research area of test case prioritization. We aimed to answer the following mapping (MQ) and review (RQ) questions:

- MQ1: Which are the most active researchers on TCP techniques?
- MQ2: Which publication venues have more primary studies on TCP?
- MQ3: Which are the most investigated TCP approaches?
- MQ4: What kinds of evaluation metrics are most frequently used in primary TCP studies?

- MQ5: What is the existing infrastructure to support the different activities for TCP evaluations?
- RQ1: What are the existing empirical evidences for TCP techniques?
 - RQ1.1: Which techniques achieve the best results in terms of effectiveness for common applications (i.e. applications that appear in more than one different study)?
 - RQ1.2: Which study context factors can affect effectiveness results obtained by TCP techniques?
 - RQ1.3: How TCP effectiveness results vary according to the used granularity? (i.e. the scale used for a TCP technique input data)

For brevity reasons, we present only the main results and discussions of this systematic literature review and mapping. For a complete description of the process, refer to Junior et al. (2017).

A total of 1563 papers were analyzed. After going through all the steps of the selection process, 90 papers remained, which were considered to answer the systematic mapping questions. Applying quality assessment criteria further refined these 90 papers and 13 remained. Those were considered to answer the systematic review questions.

2.7.1 SYSTEMATIC MAPPING RESULTS

Tables 2.1 and 2.2 summarize results for MQ1 and MQ2.

Table 2.1: Rank of authors per number of publications.

Author	Papers	Author	Papers
Gregg Rothermel	17	Hong Mei	4
Hyunsook Do	8	Sebastian Elbaum	4
Dan Hao	7	Siavash Mirabab	4
Lu Zhang	7	Zheng Li	4
Lingming Zhang	6	Bo Jiang	4
Ladan Tahvildari	5	W. K. Chan	4
Mark Harman	5	Luay Tahat	4
Bogdan Korel	5	George Koutsogiannakis	4
Zhenyu Chen	4		

A total of 64 different TCP techniques are proposed in the selected studies. These techniques can be divided into categories according to different features (CATAL; MISHRA,

Table 2.2: Papers amount per publication venue.

Publication venue	Papers
IEEE Transactions on Software Engineering	9
Lecture Notes in Computer Science	6
International Conference on Software Maintenance	5
International Computer Software and Applications Conference	4
International Conference on Software Engineering	4
International Symposium on Software Testing and Analysis	4
Software Testing, Verification and Reliability	3
International Symposium on Software Reliability Engineering	3
International Conference on Software Quality, Reliability and Security	3
International Conference on Software Testing, Verification and Validation	3
International Symposium on Foundations of Software Engineering	3
International Journal of Software Engineering and Knowledge Engineering	3
Journal of Systems and Software	3
Software Quality Journal	3
Others	34

2012; SINGH et al., 2012). These features include, for example, the input data that they need or on which assumptions they are based on.

Considering the input data required for proposed techniques in literature, selected studies describe techniques that use source code, binary form of the source code, call graphs, requirements/specifications, the whole system being tested, fault matrix, time budget, test suite, coverage information, source code change information, test input data, test execution history, execution trace and different software quality metrics.

Considering TCP techniques base approach, a previous secondary study by Singh et al. (2012) identified 6 different types of approaches. We identified 6 more. They are genetic based, modification based, coverage based, history based, fault based, similarity based, requirements based, model based, oracle based, fault diagnosis based, search based and program structure based techniques. As MQ3 is concerned with the most frequently used approaches, their usage among selected studies is listed in Table 2.3. Note that some techniques can be considered in more than one category approach. Example of this is the Bayesian Network technique proposed by Mirarab and Tahvildari (2008), which uses information about source code change and coverage and thus, can be considered as coverage based and modification based.

Table 2.3: Most investigated TCP approaches.

Approach	Amount of techniques
Coverage based	31
History based	12
Modification based	8
Similarity based	8
Genetic based	7
Model based	5
Requirements based	4
Search based	4
Fault based	2
Program structure based	2
Oracle based	1
Fault diagnosis based	1

2.7.1.1 Coverage-based TCP techniques

The assumption for coverage-based TCP techniques is that higher test coverage means better fault revealing ability. In this way, this kind of techniques prioritize test cases that cover more elements of the source code.

These kinds of techniques require that the software being tested be analyzed to collect the coverage information. This analysis can be dynamic, that is, it is collected on the fly, as test cases are executed or static, when source code is statically analyzed.

Two different approaches using test case coverage information can be found in literature. They are the total coverage and additional coverage approaches.

Techniques that use the total coverage approach, order test cases according to the sum of source code elements they cover. When there is a tie between two or more test cases, one of them is chosen randomly. Proposed by Rothermel et al. (1999), it is currently the most investigated TCP approach.

The additional coverage approach was proposed in the same paper as the total coverage (ROTHERMEL et al., 1999). Their difference is that the additional approach considers which source code elements are already covered by test cases already ordered during the ordering process. Thus, if a test case cannot provide additional coverage to the already ordered test cases, it is not considered to be important to be run first. When there is no test case that can increase the coverage of ordered test cases, then the information of already covered source code elements is emptied so that the remaining test cases can also be included in the ordered test cases. In this approach, ties are also resolved randomly.

Table 2.4: Test coverage matrix.

	E1	E2	E3	E4	E5
T1	1	0	1	0	0
T2	0	1	1	1	1
T3	1	1	1	0	0
T4	0	1	0	0	0

To illustrate how the above approaches work, consider the coverage matrix in Table 2.4. In this example, each row represents a test case (T) of the program being tested and each column represents an element (E) of the source code. This element can be a method, a statement or a branch. A cell value of 1 means that the test case covers the element represented by that cell and a value of 0 means that it does not.

Considering the total coverage approach, the total coverage of each test would be: T1 = 2, T2 = 4, T3 = 3 and T4 = 1. Thus, one possible execution order for this setting would be T2, T3, T1 and T4. Now considering the additional coverage approach, the test case that covers the biggest amount of source code would be chosen first, which is T2 and the list for already covered code would contain E2, E3, E4 and E5. This means that the next chosen test need to cover the biggest amount of not yet covered source code. Since there is a tie between T1 and T3, each covering 1 not yet covered element (E1), one of them is chosen randomly. Consider that T1 is chosen. Now, since it is not possible to cover additional source code elements with the not yet chosen test cases, the covered source code list is emptied and the test case which covers the most elements of not yet covered source code is chosen (T3) and then the remaining test case T4. One possible final execution order for this setting would be T2, T1, T3 and T4.

2.7.1.2 History-based techniques

History-based techniques use test cases historical data to determine the order for the current execution. The underlying assumption of these kinds of techniques is that test cases that failed in the past tend to also fail in the future.

A simple approach in this category is to order test cases according to the descending amount of times they failed in previous execution. Kim and Porter (2002) propose a more sophisticated approach, in which the ordering of each test case is based on three features from past executions: frequency of execution, fault revealing ability and source code coverage. In an empirical evaluation, they found that historical data might be useful

to increase the effectiveness of regression testing.

Marijan and Liaaen (2016) investigate how the amount of historical data used in these kinds of prioritization techniques impacts on its fault detection effectiveness. They define history time window as being the amount of previous test case execution data to be considered in the process. They found that the variation of this time window impacts on the history-based TCP techniques effectiveness.

2.7.1.3 Modification-based techniques

Modification-based techniques use source code change information as input to prioritize test cases. The underlying assumption is that modifying the source code can lead to the injection of new faults. In this way, prioritizing changed source code might reveal new faults first. Do et al. (2006) investigate the use of a technique in this category. The technique combines coverage and modification information to order the test cases. In the total diff method coverage prioritization technique, test cases are sorted by the total descending amount of coverage of changed methods between two versions of a program. On the other hand, the additional diff method coverage prioritization technique uses additional coverage to already ordered tests to sort them.

Mirarab and Tahvildari (2008) use modification information as one of the inputs for a Bayesian Network aiming at ordering test cases. (EGHBALI; TAHVILDARI, 2016) evaluate the technique. They found that the Bayesian Network and Additional Bayesian Network approach are outperformed by the simple total coverage technique in almost all experimented software projects.

2.7.1.4 Similarity-based techniques

Similarity-based TCP techniques use test case similarity metrics values as input. The underlying idea is to order test cases according to their (dis) similarities so that firstly executed test cases are the more diverse possible. In this way, they can possible discover faults that are different between them.

Research in this category typically lies on finding a similarity metric that is more suitable to be used. For example, (WANG et al., 2015) propose a similarity-based TCP technique and perform an experiment comparing 5 different similarity measures: Jaccard Index, Gower-Legendre, Soka-Sneath, Euclidian Distance, Cosine Similarity and

Table 2.5: Most used TCP effectiveness metrics.

Metric	Amount of studies
APFD (Average Percentage of Faults Detected)	63
APFDc (Cost-cognizant Average Percentage of Faults Detected)	8
f-measure	6
Relative Position	5
APSC (Average Percentage of Statement Coverage)	4

Proportional Distance. As results, they found that the Euclidian Distance yields the best effectiveness with their proposed approach and is comparable to the coverage-based techniques.

A total of 24 different TCP effectiveness metrics are used by selected studies in our systematic literature mapping. However, many of them are only used in one or two studies. A rank of most used metrics is shown in Table 2.5, answering MQ4.

The most commonly used TCP effectiveness metric is the Average Percentage of Faults Detected (APFD), proposed by Rothermel et al. (2001). Its values range from 0 to 1, where higher values mean faster fault detection rates. APFD values are calculated using Equation 2.1.

$$APFD = 1 - \frac{TF_1 + TF_2 + \dots + TF_m}{nm} + \frac{1}{2n} \quad (2.1)$$

In Equation 2.1, n is the amount of test cases, m is the amount of faults and TF_i represents the execution order of the first test case that detects the fault i . Its value ranges from 0 to 1, where the closest to 1, the higher effectiveness. It is worth noting that an APFD value of 1 is hard to achieve because it would mean that all faults are detected by the first executed test case. It is thus often necessary to calculate the theoretical optimal APFD value for each execution scenario when reporting an APFD measure for a TCP technique, since it can vary according to the amount of faults and test cases.

Authors use different tools to support their activities during TCP techniques execution and evaluation. We collected data about used tools for each activity in selected studies, which are depicted in Table 2.6, answering MQ5. A large amount of tools is used for source code coverage information gathering and source code mutation. This concentration can possibly be explained by the fact that the vast majority of TCP techniques are coverage based, as shown in Table 2.3.

Source code mutation tools are used by researchers to seed artificial faults into software.

Table 2.6: Tools used for each different activity in selected TCP primary studies.

Activity	Tools amount	Tools name
Code coverage information	12	CodeCover, gcov, Emma, Cobertura, JaCoCo, Cantata++, mma, Sofya, xSuds, LDRA Testbed, ATAC, Fault Tracer
Source code mutation	11	PIT, MuJava, Proteum, MutGen, Javalanche, Jumble, Major Mutation Framework, Jester, Sofya, Zoltar
Prioritization tool/framework	4	xSuds, Apros, MOTCP+, MOTCP
Source code metrics information	4	SLOCCount, ckjm, CLOC, SWT-Metrics
Execution trace information	4	valgrind, daikon, AspectJ, gcov
Bytecode manipulation/analysis	4	FaultTracer tool, ASM, Galileo, Sofya
Change analysis	3	sandmark, diff, Celadon
Linear integer programming solver: GUROBI optimization, ILOG CPLEX; test scripts generation/creation: TSL, AutoBlackTest; test execution information: time, ant; software repository: SIR; artifacts traceability: Traceclipse; multi-objective optimization framework: Jmetal; analysis of dynamic binary code: Vulcan; Bayesian Network framework: Smile Library; fault localization: Zoltar; test analysis: xSuds; fault history: LDRA Testbed; clustering: Matlab; information retrieval framework: Indri toolkit; control flow graph information: Aristotle; refactoring diff: ref-finder.		

In this way, they can measure the effectiveness of TCP techniques. It is also notable a huge amount of studies that use the Software Infrastructure Repository (SIR) to find artifacts to be used in their evaluation. In fact, it is the only cited repository that we found among the analyzed studies.

2.8 SYSTEMATIC REVIEW RESULTS

Our systematic literature review questions are mainly concerned with empirical results obtained by using TCP techniques reported by selected studies. Moreover, we are also interested on investigating relationships among factors that affect the effectiveness of these techniques.

In order to answer RQ1.1, APFD values were collected from selected studies experiments. After that, they were classified according to the software applications from where they were collected. Based on that, we highlighted the biggest APFD values for each application investigated by these studies, in order to find which techniques achieved those values. Results for this analysis are listed in Table 2.7. A complete list of collected APFD values is available in Junior et al. (2017).

Different factors can affect these results and one should not exclusively use this table to infer the best TCP technique. Instead, we provide different quantitative analysis over data collected during this review, aiming to identify relevant factors that can affect the results obtained by using TCP techniques.

Possible APFD factors candidates considered in this review are motivated by Do and Rothermel (2006) study. They make a qualitative analysis of results obtained by the use of TCP techniques on five different studies. They consider as possible factors: program size, indicated by Lines of Code (LOC) metric; test case source, i.e., who developed the test cases; test case type, number of faults in each version of the tested program and the type of these faults. In addition to the factors analyzed by them, we also considered type of seeded faults, TCP techniques and test case granularity. Each factor and its values for selected studies are described in Table 2.8.

For all analysis, the dependent variable is the APFD result obtained from selected studies. Usually the process followed by studies to obtain APFD values involves executing regression test suites from different applications, using different TCP techniques. For this reason, usually, each study reports a number of different APFD values. All of the analysis

Table 2.7: Summary of best APFD achieved on each application used by selected papers on their empirical evaluations.

Application	Technique	APFD
XML-Security	ART-st	0.970
Jdepend	add-cov	0.902
Checkstyle	t-block-cov	0.684
tcas	a-block-cov	0.840
schedule2	a-block-cov	0.720
schedule	a-block-cov	0.720
tot_info	t-block-cov	0.758
print_tokens	t-block-cov	0.808
print_tokens2	t-block-cov	0.761
replace	t-stmt-cov	0.966
space	t-stmt-cov	0.997
ant	a-stmt-cov	0.954
jmeter	a-branch-cov	0.883
jtopas	a-branch-cov	0.970
Altitude Switch (ASW)	t-branch-cov	0.766
Wheel Brake System (WBS)	t-branch-cov and a-fn-cov	0.821
Flight Guidance System (FGS)	a-fn-cov	0.801
NoiseGen	a-fn-cov	0.742
Galileo	BNA-block	0.940
NanoXML	a-fn-cov	0.945

Techniques summary: **ART-st**: statement ART; **add-cov**: additional coverage; **t-block-cov**: total block coverage; **a-block-cov**: additional block coverage; **t-stmt-cov**: total statement coverage; **a-stmt-cov**: additional statement coverage; **a-branch-cov**: additional branch coverage; **t-branch-cov**: total branch coverage; **BNA-block**: additional bayesian network with block coverage; **a-fn-cov**: additional function coverage

Table 2.8: Possible APFD factors.

Factor	Values (treatments)
Faults type	Seeded / real
Average number of faults per version	Continuous values
Test case source	Provided / generated
Test case type	JUnit / TSL
Program size (LOC)	Continuous values
Faults seeding type	Manual / mutation
TCP technique coverage granularity	Branch / statement / block / method and function
Test case granularity	Method level / class level

Table 2.9: Summary of our relevant factors findings.

Factor	P-value	Effect on APFD
Fault type	0.000	Significant
Average amount of faults per version	0.009	Significant
Test case source	0.032	Significant
Test case type	0.689	Not significant
LOC	0.958, 0.708	Not significant
Seeded faults type	0.000	Significant
TCP technique granularity	0.000	Significant
Test granularity	0.014	Significant

process was performed using the statistical tool Minitab 17. Table 2.9 summarizes the results found. For detailed process, check Junior et al. (2017).

We found that 6 out of 8 candidate factors achieved a significant result. When compared to Do and Rothermel (2006) qualitative analysis, we could confirm all 3 factors that they considered relevant to the APFD result. They are fault type, average amount of faults per version and test case source. These results can be used to guide future research in the topic. For example, to develop TCP techniques that execute test cases at the method level, achieving better effectiveness.

2.9 INDUSTRIAL ADOPTION

The systematic review and mapping presented in Section 2.7 does not focus on usage of TCP in industry. For this reason, to strengthen the understanding of the practice of TCP in industry, we perform a structured search in literature with the goal of finding evidence about the use of TCP techniques in the industrial setting, i.e. in projects being developed by real companies.

Research question for this search is: at what level TCP is used in industry?

The strategy to answer this question is to query digital libraries using a Boolean string composed of keywords related to our goal. The main reason for choosing this strategy is to prevent the retrieval of papers that are not peer-reviewed.

Search string: (Test or testing) AND (prioritization OR prioritisation) AND (adoption OR industry OR experience OR case OR report). It was used in four different digital libraries: IEEE Xplore, ACM digital Library, Science Direct and Scopus. The amount of papers returned by each one is described in Table 2.10.

In total, the digital libraries returned 1581 papers. Each of these papers was analyzed

Table 2.10: Amount of papers queried.

Digital library	Amount of papers
IEEE	398
ACM	393
Science Direct	51
Scopus	739
Total	1581

based on its title, abstract and keywords to check if they could help answering our research question. Papers that were clearly related to our search were fully read. In the end of this process, 11 papers were selected. Their references are listed in Table 2.11.

Most (9) of the selected papers only report the usage of TCP techniques in empirical studies, using software objects from industrial partners, like Microsoft Carlson et al. (2011), Sony Ericsson Engström et al. (2011), Westermo Research and Development AB Strandberg et al. (2016), Salesforce.com Busjaeger and Xie (2016) and Cisco Wang et al. (2016). These studies do not report the use of TCP techniques after the studies were performed.

Only 2 of 11 selected papers report usage of TCP techniques in the daily routine of real companies. Czerwonka et al. (2011) report the use of a tool named CRANE in Microsoft. This tool provides data that may be useful during Windows products maintenance and integrates with a TCP tool, named Echellon, which is internal and proprietary to Microsoft. Prioritization is performed based on coverage criteria. They also report that the use of this tool brought benefits related to costs and efficiency. Strandberg et al. (2016) report a case study in a real industrial environment with the company Westermo Research and Development AB. They apply the use of an automation and TCP tool, which was later integrated into the company development process.

Analyzed evidence from literature suggests that the available tools and frameworks are specific for some cases, like an empirical study or environment and TCP might not yet be ready to be used in industrial settings.

2.10 FINAL CONSIDERATIONS

In this chapter, we aimed to explain the main concepts that will be used throughout this work. We also aimed to answer our first research question: “How is TCP being

used in industry and literature?”. This is done through the systematic literature review and mapping, discussed in Section 2.7, answering how TCP is being used in literature and through a structured search of the literature for industrial reports of TCP usage, in Section 2.9.

All techniques that were considered in our review analysis are coverage-based. This indicates the necessity for more quality empirical studies regarding different TCP techniques. There are plenty of empirical studies, as demonstrated by the number of accepted studies in the mapping (90), but when our quality assessment was applied, only thirteen studies remained. Furthermore, the amount of rejected papers in the mapping and review suggests that authors are proposing many TCP techniques but they are not empirically evaluating them. This fact can be a problem. Since there is a low amount of empirical studies, practitioners and researchers do not have a reliable amount of empirical evidence to choose a suitable TCP technique for their needs.

Regarding TCP usage in industry, our structured search indicates that TCP is not mature enough yet and is not being widely used. We found only two reports of industrial usage. The majority of our findings show the use of industrial software projects in empirical studies but no further usage in the day-to-day development process.

Findings from this chapter indicate the necessity of actions seeking to mature the research in TCP techniques in order to help practitioners on adopting its approaches in industry. An initial step for this might be to provide reliable and easy ways to use and experiment TCP techniques.

Table 2.11: Literature evidence of TCP use in industrial settings.

Ref.	Title	Year	Industrial partner	Adopt TCP?
(YANG et al., 2017)	An Industrial Study of Natural Language Processing Based Test Case Prioritization	2017	Everyday Network Co. Ltd.	X
(PAREJO et al., 2016)	Multi-objective test case prioritization in highly configurable systems: A case study	2016	Name not reported	X
(STRANDBERG et al., 2016)	Experience Report: Automated System Level Regression Test Prioritization Using Multiple Factors	2016	Westermo Research and Development AB	✓
(BUSJAEGER; XIE, 2016)	Learning for test prioritization: an industrial case study	2016	Salesforce.com	X
(WANG et al., 2016)	Enhancing Test Case Prioritization in an Industrial Setting with Resource Awareness and Multi-objective Search	2016	Cisco Norway	X
(MARLIAN et al., 2013)	Test Case Prioritization for Continuous Regression Testing: An Industrial Case Study	2013	Name not reported	X
(NARDO et al., 2013)	Coverage-Based Test Case Prioritisation: An Industrial Case Study	2013	Name not reported	X
(SRIKANTH; COHEN, 2011)	Regression testing in Software as a Service: An industrial case study	2011	Name not reported	X
(CZERWONKA et al., 2011)	CRANE: Failure Prediction, Change Analysis and Test Prioritization in Practice – Experiences from Windows	2011	Microsoft	✓
(CARLSON et al., 2011)	A clustering approach to improving test case prioritization: An industrial case study	2011	Microsoft	X
(ENGSTRÖM et al., 2011)	Improving Regression Testing Transparency and Efficiency with History-Based Prioritization – An Industrial Case Study	2011	Sony Ericsson Mobile Communications	X

3 RELATED WORKS

Related works are separated into three categories. The first one is for works that allow the use of TCP techniques. The second one is for works that allow the TCP usage and can be integrated into the continuous integration process. Finally, the third one is for works that describe TCP approaches for continuous software engineering but did not describe any means to execute it, like a tool or framework.

3.1 TCP USAGE

In this subsection, works that describe stand-alone tools that can execute TCP techniques are discussed.

Do et al. (2006) discuss the use of a JUnit framework extension together with a byte-code analyzer tool, named Galileo. This extension is then used in an empirical study to evaluate implemented TCP techniques effectiveness. Implemented techniques include coverage and modification-based approaches.

Kauffman and Kapfhammer (2012) propose two tools that are part of a test case prioritization/reduction framework. According to authors, the goal of using this framework is to support empirical studies on these types of techniques and also to enable practitioners to use them in their daily development tasks. The framework includes search-based and coverage-based TCP techniques. The framework is also able to measure produced outputs for prioritization and reduction approaches and offer visualizations so researchers can analyze experiment results. It integrates with JUnit and Cobertura¹. Although it seems promising for usage in this work, it has not been updated in the last six years.

Alves et al. (2016) use an open source TCP tool, called PriorJ, which allows the use of TCP techniques on Java language applications that use JUnit test framework. Among its offered TCP techniques, there are coverage-based ones, using source code coverage at method and statement level, a change-based technique and a refactoring-based approach. It does not offer any kind of integration with build automation tools, being a standalone tool, which can hinder its usage on a Continuous Integration environment.

¹<https://github.com/cobertura/cobertura>

(SÁNCHEZ; SEGURA, 2017) propose a tool named SmarTest, which is a test prioritization tool for accelerating the detection of faults in Drupal environment. It prioritizes test cases based on the number of commits made in the code, or based on the tests that failed in last executions. Authors do not discuss any means of integrating this tool into a continuous integration process.

3.2 TCP IN CONTINUOUS SOFTWARE ENGINEERING

In this sub-section, works that discuss the use of TCP techniques in continuous software engineering environment are discussed.

Elbaum et al. (2014) propose TCP techniques that are modeled specifically for continuous integration environments. The techniques are based on the notion of time window for testing and are based on historical execution data. The techniques are experimented on test data from Google and results show that the proposed techniques can reduce delay to detect faults during the test process.

Marijan and Liaaen (2016) also investigate the use of time windows for prioritizing test cases in continuous regression testing. However, they use different type of historical test data than Elbaum et al. (2014). They investigate the effect of varying the amount of historical data to be considered when prioritizing test cases using historical-based approaches on a project.

3.3 TCP USAGE AND CONTINUOUS SOFTWARE ENGINEERING

In this section, we discuss studies that intersect the previous two sub-sections. That is, works that propose or discuss the use of a TCP tool or framework that can be used with automated build tools and thus integrated into continuous software engineering activities, such as continuous integration.

Plewnia (2015) proposes a Regression Test Optimization (RTO) platform named Lazzer that eases the implementation of RTO techniques and their usage on existing software. It considers two different types of RTO approaches, namely selection and prioritization techniques. It is not currently publicly available. Implemented TCP techniques are history-based. It offers a Maven integration, which can allow its usage on continuous integration processes. The platform is only evaluated qualitatively and has not been used in any TCP

experiment.

Öhlin (2017) proposes a framework to evaluate TCP approaches effectiveness. It is used to evaluate approaches with test data from Spotify Company. Approaches include history-based, modification-based and machine learning-based. Author found as results that the approaches that performed better were history-based. He did not, however, integrate the developed framework into the CI process of the company.

Spieker et al. (2017) propose an automatic reinforcement learning method to prioritize and select test cases in a continuous integration environment named Retecs. The technique is based on historical information about test executions.

Test Load Balancer² (TLB) is a tool that allows the partitioning of test cases in equal parts to be run in parallel. Moreover, it allows the use of a TCP technique, executing test cases that failed in the previous build first or allowing developers to implement their own prioritization technique. TLB supports integration with automated build tools, like Maven or Ant and also support different programming languages, like Java and Ruby.

Clover – Atlassian³ is a test management framework. It offers coverage information and tests optimization. Among the optimizations, the prioritization uses information about failed tests in previous run to determine the order of the next run. Tests are ordered according to coverage of modified code and ascending by invocation run time.

Maven Surefire⁴ is a Maven plugin to execute test cases during the build process of projects using custom configurations, like parallel processing or inclusion of external tests. It also generates reports about executed tests in different formats. Moreover, it offers the option to reorder the tests to be executed, executing those that failed on previous run first.

3.4 EXISTING APPROACHES COMPARISON

In the previous Chapter, the following problems were identified in literature.

1. Low amount of empirical studies.
2. Low amount of industry usage evidence.
3. Diversity of TCP techniques input data.

²<http://test-load-balancer.github.io/>

³<https://www.atlassian.com/software/clover>

⁴<http://maven.apache.org/surefire/maven-surefire-plugin/>

4. Possibility of improving TCP rate of fault detection effectiveness using method test granularity.

Taking into account the necessity of automation of builds in the continuous software engineering environment and the problems listed above, we derive requirements that a solution must have to address the problems. The first requirement is that the solution must integrate into the automated build process. This can possibly be achieved with automated build tools plugins.

The second requirement is derived from the low amount of empirical studies in literature. We conjecture that this might be due to the difficulties associated with performing an empirical study in this topic. To address this, we believe that a solution needs to provide an API so that researchers can implement the TCP techniques that they need to experiment. Furthermore, the solution must also provide means of measuring the effectiveness of implemented TCP techniques.

Another problem that possibly hinder the use of TCP is that different TCP techniques need different project data information, like test coverage, historical execution and modification information. In this way, another derived requirement is that a solution needs to provide these types of data.

To be able to further investigate the use of test-method granularity, a solution must provide a mechanism to achieve this.

Finally, we conjecture that the low amount of industry usage evidence might be due to the combination of most of the discussed problems. Providing a solution that can execute different TCP techniques, which use different project data information, at the test-method granularity and able to provide measurements of its effectiveness, can possibly encourage researchers on performing and reporting more empirical studies. More empirical evidence may enable practitioners to choose adequate techniques for industrial usage. In this sense, a good solution might also need to be publicly available.

Requirements:

1. Build cycle integration
2. API to implement new TCP techniques
3. TCP techniques effectiveness measurement
4. Provision of project data

5. Test execution at the method granularity
6. Publicly available

Based on those requirements, we present a comparison in Table 3.1 to reason about how related works address them.

Table 3.1: Comparison of requirements among related work.

Solution	Requirements					
	1	2	3	4	5	6
(DO et al., 2006)	<i>X</i>	<i>X</i>	<i>X</i>	Coverage	✓	<i>X</i>
(KAUFFMAN; KAPFHAMMER, 2012)	<i>X</i>	<i>X</i>	✓	Coverage	<i>X</i>	✓
(ALVES et al., 2016) - PriorJ	<i>X</i>	✓	✓	Coverage, Refactoring, Modifications	<i>X</i>	✓
(SÁNCHEZ; SEGURA, 2017) - SmarTest	<i>X</i>	<i>X</i>	<i>X</i>	Coverage, Historical	N/A	✓
(PLEWNIA, 2015)	Maven	✓	<i>X</i>	Historical	<i>X</i>	<i>X</i>
(ÖHLIN, 2017)	<i>X</i>	✓	✓	Historical, Modifications	N/A	<i>X</i>
(SPIEKER et al., 2017)	<i>X</i>	<i>X</i>	✓	Historical	N/A	✓
Test Load Balancer (TLB)	Ant, Buildr	✓	<i>X</i>	Historical	<i>X</i>	✓
Clover – Atlassian	Maven, Ant	<i>X</i>	<i>X</i>	Coverage, Historical	<i>X</i>	✓
Maven Surefire	Maven	<i>X</i>	<i>X</i>	Historical	<i>X</i>	✓

3.5 FINAL CONSIDERATIONS

In this chapter we reasoned about existing works and how they address problems identified in the last Chapter. As can be observed in Table 3.1, existing solutions do not address all the requirements to solve the problems. For this reason, in the next Chapter we propose Optimus Framework. It is a test framework aiming at providing means for executing and experimenting different TCP techniques.

4 OPTIMUS FRAMEWORK DESIGN

4.1 INTRODUCTION

In this chapter we design a framework aiming to solve the problems discussed in the last sections. The main goal of the framework is to allow the execution, implementation and experimentation of TCP techniques in a continuous integration environment.

4.2 REQUIREMENTS

As discussed in the last Chapter, requirements for this proposal were derived from problems identified in the literature and not met by related works.

- R1: to allow the usage of such framework in the continuous integration environment, it needs to provide integration with automated build tools.
- R2: to allow the inclusion of new TCP techniques, the framework must provide an API.
- R3: to support the conduction of experiments with TCP techniques, the framework must provide means of measuring TCP techniques effectiveness.
- R4: to support the development of new techniques, the framework must provide different project data to be used as input. In order to maximize the diversity of possible techniques, the most used approaches need to be addressed. In this case, coverage-based, history-based, modification-based and similarity-based, according to our systematic literature mapping.
- R5: to allow investigation of different test granularities, the framework must allow the execution of test cases at the method and class granularity.
- R6: to encourage the adoption of TCP techniques, the framework must be publicly available and easy to be used.

Regression testing is one of the activities performed during the project build. The process is outlined in Figure 4.1 to provide a better understanding of how it happens in a continuous integration environment.

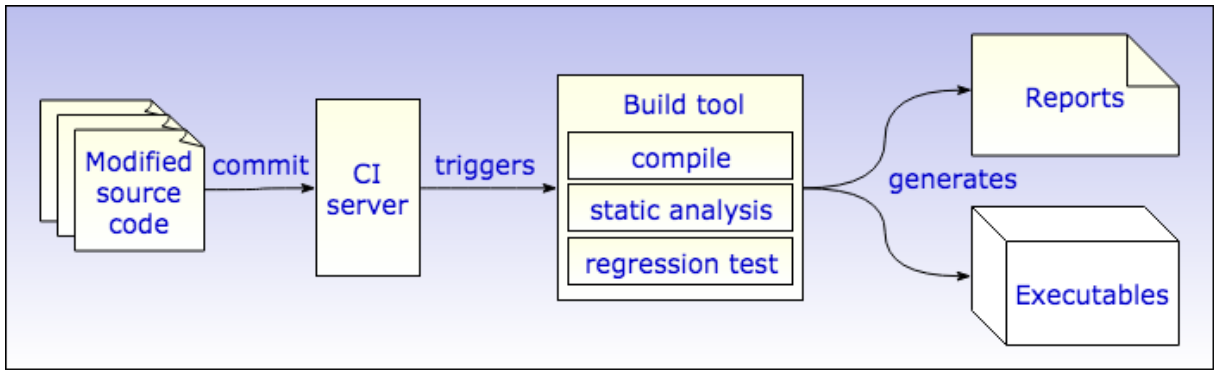


Figure 4.1: Typical continuous integration process.

Based on this process, test case prioritization is performed during the regression test execution, which is executed during the software build.

Considering that the focus of the framework is to meet the needs of both industry and researchers, it must provide two different means of executing TCP techniques. One, aiming at practitioners from industry, is to simply execute a TCP technique during the build process of a project. The other, aiming at researchers from academia, is to execute different TCP techniques and configurations over a software project, aiming to measure and compare their effectiveness.

In this way, we propose the framework depicted in Figure 4.2. Each of its components is described next.

The framework interfaces with an automated build tool. It must be called during the regression testing phase of the build by the build tool, execute and then return the control to the build tool.

4.3 ARTIFACTS REPOSITORY

This module is responsible for storing data and providing access to configuration of the framework and artifacts of the project, generated during its execution. Example: test case results, experiments reports, coverage data.

4.4 EXPERIMENTS SUPPORT

Two modules in the architecture are dedicated to support the experimentation of TCP techniques: TCP effectiveness analyzer and Reports generator.

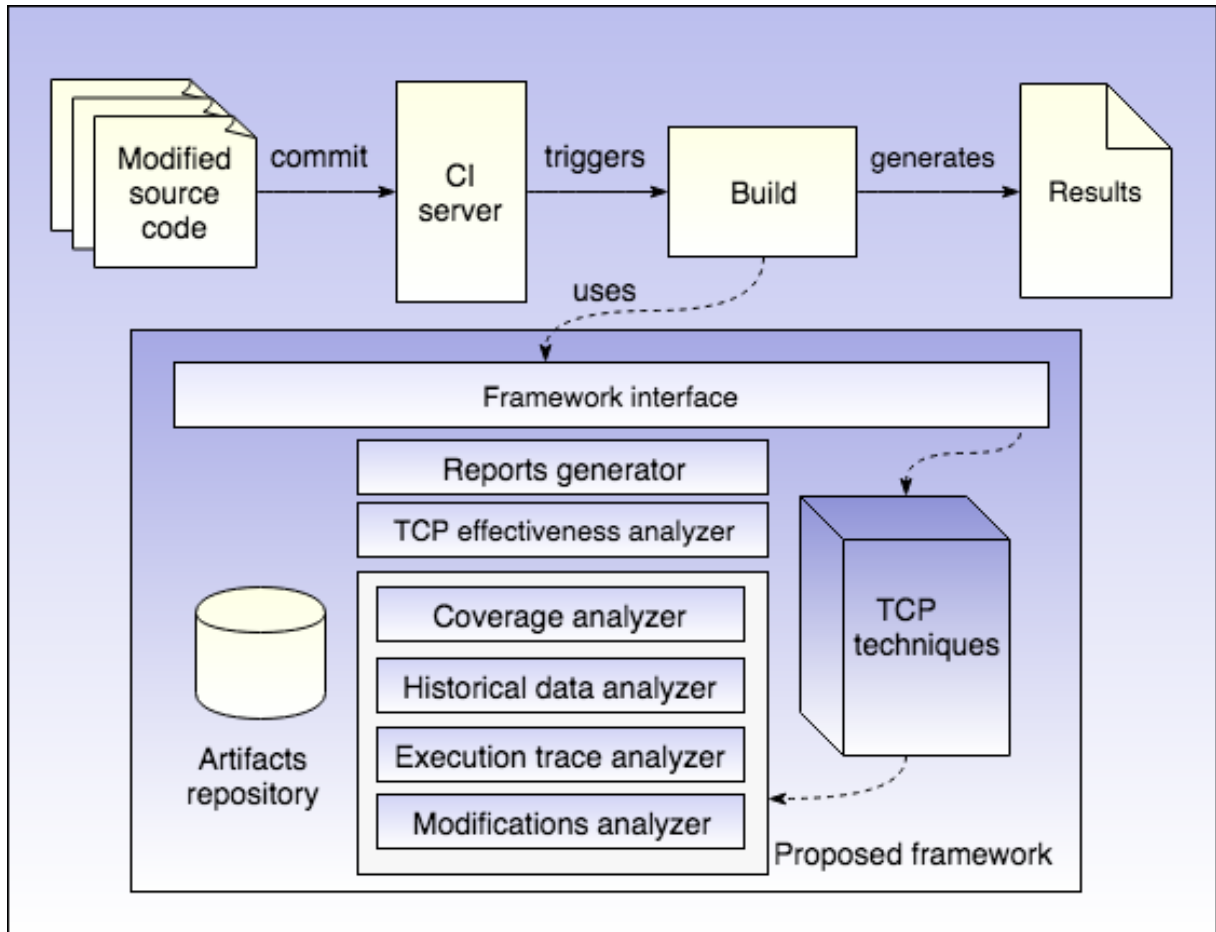


Figure 4.2: Optimus framework architecture.

4.4.1 TCP EFFECTIVENESS ANALYZER

This module is responsible for measuring the effectiveness of TCP techniques and helping with the conduction of experiments. It includes four sub-modules: fault seeder, faults finder, test execution simulator and automation of experiments.

Fault seeder: to allow the experimentation of TCP techniques, the project in which they are executed must contain faults. According to Andrews et al. (2005), in absence of real faults, mutation faults can be used instead. The aim of this sub-module is to seed faults into projects using mutation of the source code.

Faults finder: according to Paterson et al. (2018), using mutation faults for measuring TCP techniques effectiveness is not always representative of real faults. For this reason, this sub-module is responsible for searching and downloading versions of open source projects that contain real failing tests.

Test execution simulator: executing the regression tests incur in executing all test cases of a project. This task may require hours to finish, making it difficult to measure

the effectiveness of many TCP techniques and compare them. For this reason, this sub-module is responsible for accelerating this process by executing the test cases once to gather the necessary data and then simulate the execution for the TCP techniques.

Automation of experiments: executing experiments to measure and compare the effectiveness of different TCP techniques require the manual trigger of the regression tests for each desired configuration. To support this laborious task, this sub-module is responsible for automating this process, by setting all the configurations to be run and executing each of them.

4.4.2 REPORTS GENERATOR

This module is responsible for generating reports after the execution of experiments using the framework. Those reports must contain detailed data about the execution of each test case and each TCP technique. Furthermore, a summary report must also be generated, aggregating the results of different TCP techniques to allow its comparison.

4.5 ANALYZERS

Analyzers are responsible for extracting and providing project data as input for the TCP techniques. Types of project data included are motivated by the most used approaches in literature, according to our systematic mapping.

4.5.1 COVERAGE ANALYZER

This module is responsible for collecting and analyzing coverage data. This coverage information is relative to the source code covered by test cases. In this way, this module must provide per-test coverage information. Moreover, it should also be provided in different granularities of source code elements, like statements, branches and methods.

4.5.2 HISTORICAL DATA ANALYZER

This module is responsible for collecting historical data about test cases execution. In this way, after the execution of each test case, generated information must be stored to be used in the future. Thus, this module should also provide access for stored information.

4.5.3 EXECUTION TRACE ANALYZER

This module is responsible for collecting execution traces of test cases. Execution traces are used by similarity-based TCP techniques. An execution trace differs from coverage information in the sense that the latter informs if each source code element was executed by each test case, while the former informs how many times each source code element was executed by each test case. Besides collecting such information, this module must also provide access to it.

4.5.4 MODIFICATIONS ANALYZER

This module is responsible for managing, collecting and providing access to source code modification information between versions of a project. Such information includes, for example, source code elements that were modified and amount of lines modified.

4.6 TCP TECHNIQUES

This module provides a set of TCP techniques that can be used in the framework. New techniques can be developed and added at any time. A total of 9 TCP techniques were selected, based on our systematic review and mapping of the literature, to be included in the first version of the framework. They are described next.

To illustrate each technique example, consider a simple software project designed as depicted in the UML class diagram in Figure 4.3. Source code classes are represented in blue and test classes are represented in green.

In the example, there are 3 classes, with a test class for each one and a test class that tests the integration between ClassA and ClassB. Moreover, arrows in the diagram indicate the dependency between them. For example, ClassA depends on ClassB, because there is an attribute of type ClassB.

4.6.1 COVERAGE-BASED

As it was already described, coverage-based TCP techniques use coverage information to determine the order of the test cases. Three coverage-based techniques are included in our framework. They are total coverage and additional coverage, which were already described in Section 2.7.1.1 and Adaptive Random Testing (ART). Considering planned

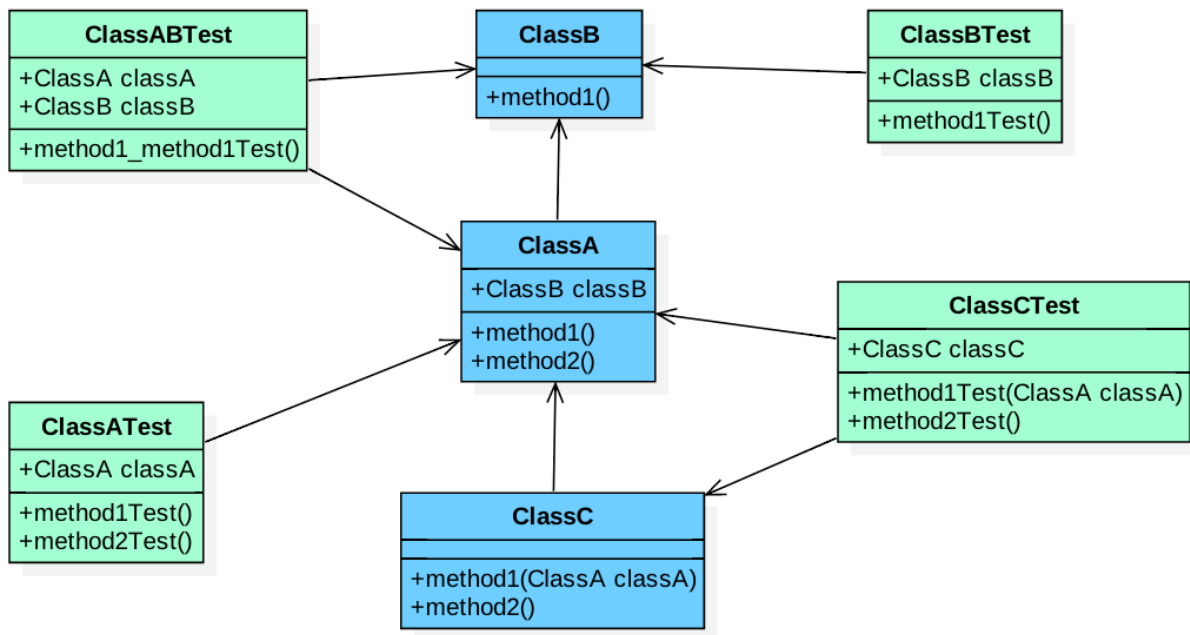


Figure 4.3: Example software project.

coverage granularities to be included in the framework, those techniques are included in 3 different versions (statement, branch and method); 2 different versions considering the test execution granularity (method-level and class-level) and 3 different versions considering their base approach (Total, Additional or ART). ART technique is also available with 3 different versions regarding its selection function (Max, Min and Avg). In this way, the framework provides 6 variations of total coverage, 6 variations of additional coverage and 18 variations of ART, totaling 30 different options of coverage-based TCP techniques, as listed in Table 4.1 and Table 4.2. Their base approaches are described next and examples are given considering the method test-level, method coverage granularity and the Min selection function for ART.

For demonstration purposes of coverage-based approaches, a coverage matrix is displayed in Table 4.3, relative to the example project depicted in Figure 4.3. In Table 4.3, the cell value “1” means that the method in that column is covered by the test case in its row. A value of 0 means the opposite. For example, method1 of the ClassB is covered by test cases #3 and #6.

Table 4.1: Variations of Total and Additional coverage techniques.

Name	Technique	Coverage granularity	Test granularity
Total statement coverage at test method level	Total coverage	Statement	Method-level
Total branch coverage at test method level		Branch	
Total method coverage at test method level		Method	
Total statement coverage at test class level		Statement	Class-level
Total branch coverage at test class level		Branch	
Total method coverage at test class level		Method	
Additional statement coverage at test method level	Additional coverage	Statement	Method-level
Additional branch coverage at test method level		Branch	
Additional method coverage at test method level		Method	
Additional statement coverage at test class level		Statement	Class-level
Additional branch coverage at test class level		Branch	
Additional method coverage at test class level		Method	

Table 4.2: Variations of ART coverage technique.

Name	Coverage granularity	Test granularity	Selection function
ART Max statement at test method level	Statement	Method-Level	Max
ART Max branch at test method level	Branch		
ART Max method at test method level	Method		
ART Max statement at test class level	Statement	Class-Level	
ART Max branch at test class level	Branch		
ART Max method at test class level	Method		
ART Min statement at test method level	Statement	Method-Level	Min
ART Min branch at test method level	Branch		
ART Min method at test method level	Method		
ART Min statement at test class level	Statement	Class-Level	
ART Min branch at test class level	Branch		
ART Min method at test class level	Method		
ART Avg statement at test method level	Statement	Method-Level	Avg
ART Avg branch at test method level	Branch		
ART Avg method at test method level	Method		
ART Avg statement at test class level	Statement	Class-Level	
ART Avg branch at test class level	Branch		
ART Avg method at test class level	Method		

Table 4.3: Coverage matrix for the example project.

#	Tests cases	Source Code				
		ClassA		ClassB	ClassC	
		method1	method2	method1	method1	method2
1	ClassATest. method1Test	1	1	0	0	0
2	ClassATest. method2Test	0	1	0	0	0
3	ClassBTest. method1Test	0	0	1	0	0
4	ClassCTest. method1Test	1	1	0	1	0
5	ClassCTest. method2Test	0	0	0	0	1
6	ClassABTest. method1_method1Test	1	0	1	0	0

4.6.1.1 Total coverage

The total coverage approach, described in Algorithm 1, order test cases according to the total amount of source code elements covered. If a tie occurs between two or more test cases that cover the same amount of source code elements, one of them is chosen randomly.

Considering the coverage matrix displayed in Table 4.3, the total coverage for each test case is displayed in Table 4.4.

According to Algorithm 1 and the total coverage for each test case displayed in Table 4.4, firstly test case #4 would be chosen, since it covers 3 source code elements, which is the biggest amount among the test cases. Then, a tie occurs between test cases #1 and #6, with each of them covering 2 source code elements. Thus, one of them is chosen randomly. Suppose that #6 is chosen. In the next iteration #1 is chosen, and then a tie occurs again between test cases #2, #3 and #5. The process repeats until all test cases are ordered. One possible final order for this approach in this scenario is: #4, #6, #1, #3, #2 and #5.

4.6.1.2 Additional coverage

The additional coverage approach is similar to the total one. The main difference is that a list of already covered source code elements by already ordered test cases is maintained and updated after each iteration. This is done to allow the calculation of the additional coverage that a test case can yield to the already ordered test cases if it is chosen. In this

Algorithm 1: Total coverage base algorithm

Input: Test coverage matrix C , test cases set T
Output: Ordered test cases set T'
 $T' \leftarrow \emptyset$;
while T not empty **do**
 $MaxT \leftarrow \emptyset$;
 $MaxC \leftarrow 0$;
 foreach t in T **do**
 $totalCoverage \leftarrow \text{getTotalCoverage}(t, C)$;
 if $totalCoverage > MaxC$ **then**
 $MaxT \leftarrow \emptyset$
 end if
 if $totalCoverage \geq MaxC$ **then**
 Add t to $MaxT$;
 $MaxC \leftarrow totalCoverage$;
 end if
 end foreach
 $chosenT \leftarrow \text{random}(MaxT)$;
 Add $chosenT$ to T' ;
 Remove $chosenT$ from T ;
end while
Function $\text{getTotalCoverage}(t, C)$:
 $totalCoverage \leftarrow 0$;
 foreach covered element of t in C **do**
 $totalCoverage \leftarrow totalCoverage + 1$;
 end foreach
return $totalCoverage$

Table 4.4: Total coverage for the example application.

#	Tests cases	Total coverage
1	ClassATest.method1Test	2
2	ClassATest.method2Test	1
3	ClassBTest.method1Test	1
4	ClassCTest.method1Test	3
5	ClassCTest.method2Test	1
6	ClassABTest.method1_method1Test	2

way, it is expected that test cases executed firstly cover the greatest amount of source code elements possible for that test suite. If a tie occurs between the additional coverage of two or more test cases, one of them is chosen randomly. Furthermore, if no test cases can provide additional coverage to the already ordered set, the already covered elements list is emptied so that the process can start over. The base algorithm for this approach is displayed in Algorithm 2.

Considering the coverage matrix displayed in Table 4.3, the initial additional coverage for each test case is displayed in Table 4.5. In the first iteration, this information is always the same for total and additional approaches.

According to Algorithm 2, the first chosen test case would be #4, since it yields the greatest amount of additional coverage to the ordered set of test cases, which is empty, since this is the first iteration. After adding test case #4 to the ordered test cases set, the covered elements list contains: ClassA.method1, ClassA.method2 and ClassC.method1. The additional coverage for each test case is recalculated and displayed in Table 4.6.

Since there is a tie between the additional coverage of test cases #3, #5 and #6, one of them is chosen randomly. Suppose that #5 is chosen. Now, the covered elements list contains: ClassA.method1, ClassA.method2, ClassC.method1 and ClassC.method2. The additional coverage for each test case is recalculated and displayed in Table 4.7.

Since there is a tie between test cases #3 and #6, one of them is chosen randomly. Suppose that #3 is chosen. The covered elements list contains: ClassA.method1, ClassA.method2, ClassC.method1, ClassC.method2 and ClassB.method1. The ordered test cases set contains: #4, #5 and #3. The additional coverage for each test case is recalculated and displayed in Table 4.8.

Considering that there is no additional source code element that can be covered by not yet ordered tests, the covered elements list is emptied. The process restarts and additional coverage is recalculated as displayed in Table 4.9.

Since there is a tie between test cases #1 and #6, one of them is chosen randomly. Suppose that #6 is chosen. The covered elements list contains: ClassA.method1 and ClassB.method1. The ordered test cases set contains: #4, #5, #3 and #6. The process continues until all test cases have been ordered. One possible final order for this scenario would be: #4, #5, #3, #6, #2 and #1.

Algorithm 2: Additional coverage base algorithm

Input: Test coverage matrix C , test cases set T

Output: Ordered test cases set T'

$L \leftarrow \emptyset$; ▷ temporary set of covered elements
 $T' \leftarrow \emptyset$;

while T not empty **do**

$MaxAdditionalT \leftarrow \emptyset$;

$MaxAdditionalC \leftarrow 0$;

foreach t in T **do**

$AdditionalC \leftarrow \text{getTotalCoverage}(t, C, L)$;

if $AdditionalC > MaxAdditionalC$ **then**

$MaxAdditionalT \leftarrow \emptyset$

end if

if $AdditionalC \geq MaxAdditionalC$ **then**

Add t to $MaxAdditionalT$;

$MaxAdditionalC \leftarrow AdditionalC$;

end if

end foreach

if $MaxAdditionalC = 0$ **then**

$L \leftarrow \emptyset$;

else

$t \leftarrow \text{random}(MaxAdditionalT)$;

Add covered elements by t to L ;

Add t to T' ;

Remove t from T ;

end if

end while

Function $\text{getAdditionalCoverage}(t, C)$:

$additionalCoverage \leftarrow 0$;

foreach covered element of t in C **do**

$additionalCoverage \leftarrow additionalCoverage + 1$;

end foreach

return $additionalCoverage$

Table 4.5: Additional coverage for example application, first iteration.

#	Tests cases	Additional coverage
1	ClassATest.method1Test	2
2	ClassATest.method2Test	1
3	ClassBTest.method1Test	1
4	ClassCTest.method1Test	3
5	ClassCTest.method2Test	1
6	ClassABTest.method1_method1Test	2

Table 4.6: Additional coverage for example application, second iteration.

#	Tests cases	Additional coverage
1	ClassATest.method1Test	0
2	ClassATest.method2Test	0
3	ClassBTest.method1Test	1
5	ClassCTest.method2Test	1
6	ClassABTest.method1_method1Test	1

4.6.1.3 Adaptive Random Testing

The Adaptive Random Testing (ART) technique also uses coverage information to execute. As its name suggests, it is based on randomness. According to Jiang et al. (2009), the main idea is to spread the distribution of test cases as evenly as possible across the input domain, which in this case is the coverage of source code elements.

The algorithm is based on two main steps. One is the generate procedure, which builds a candidate set of test cases to be selected. Test cases are randomly added to this set as long as they can increase the test coverage of the whole set, in other words, as long as their additional coverage is not 0. In this way, if a test case that does not add coverage to the set is drawn, the algorithm continues to the next step.

The goal of the next step is to select one test case from the candidate set to be included in the final ordered set. Test cases are selected based on their distance from the ordered set. Distance is measured using the Jaccard Distance metric with coverage data as input. Three different functions can be used in this step to select a test case based on their distance. They are minimum, average or maximum distances values. If, for example, the minimum distance is used, then the test case selected will have the greatest minimum distance to the ordered set of test cases. The choice of a distance function is a research topic itself. Algorithm 3 describes ART approach in detail.

Table 4.7: Additional coverage for example application, third iteration.

#	Tests cases	Additional coverage
1	ClassATest.method1Test	0
2	ClassATest.method2Test	0
3	ClassBTest.method1Test	1
6	ClassABTest.method1_method1Test	1

Table 4.8: Additional coverage for example application, fourth iteration.

#	Tests cases	Additional coverage
1	ClassATest.method1Test	0
2	ClassATest.method2Test	0
6	ClassABTest.method1_method1Test	0

Definition 2. Jaccard distance is a dissimilarity index based on the original Jaccard index proposed by Jaccard (1901). It can be used to measure the dissimilarity between two sets. In this work, it measures the dissimilarity between two test cases, according to their coverage information. To represent the coverage information of a test case, a Boolean string is constructed with as many characters as there are coverable source code elements in the program. A character value of “1” in the string means that the corresponding source code element is covered by that test case and a value of 0 means otherwise. The distance value is given by Equation 4.1. Bigger the value, bigger the dissimilarity between A and B.

$$J(A, B) = 1 - \frac{|A \cap B|}{|A \cup B|} \quad (4.1)$$

Considering the coverage information from Table 4.3, an example of the execution of this technique is described following. Consider also that the selection function is the Minimum one.

The first step consists of randomly generating a candidate set of test cases. Suppose that the first test case drawn is #5. The next one is #3, which has an additional coverage score of 1. Suppose that the next one drawn is #6, which also has an additional coverage score of 1, considering the candidate set. The next one is #4, with an additional score of 2. Any next test case will have an additional coverage score of 0, since all source code elements are already covered by the current candidate set. In this way, we will use this candidate set (#3, #4, #5 and #6) in the next step.

Algorithm 3: ART base algorithm

Input: Test coverage matrix C , test cases set T

Output: Ordered test cases set T'

$T' \leftarrow \emptyset$;

while T not empty **do**

$candidateSet \leftarrow \text{generate}(T, C)$;

$t' \leftarrow \text{select}(candidateSet)$;

 Add t' to T' ;

 Remove t' from T ;

end while

Function $\text{generate}(T, C)$:

$candidateSet \leftarrow \emptyset$;

$additionalC \leftarrow 1$;

$t \leftarrow \text{random}(T)$;

while $additionalC \neq 0$ **do**

 Add t to $candidateSet$;

$t \leftarrow \text{random}(T)$;

$additionalC \leftarrow$ get additional coverage for t in $candidateSet$, using C ;

end while

return $candidateSet$

Function $\text{select}(candidateSet, T')$:

$maxDistanceT \leftarrow \emptyset$;

$maxDistance \leftarrow -1$;

foreach cs in CS **do**

$distance \leftarrow \text{selectFunction}(cs, T')$; \triangleright returns the Maximum, Average
 or Minimum distance from T'

if $distance > maxDistance$ **then**

$maxDistanceT \leftarrow cs$;

$maxDistance \leftarrow distance$;

end if

end foreach

return $maxDistanceT$

Table 4.9: Additional coverage for example application, fifth iteration.

#	Tests cases	Additional coverage
1	ClassATest.method1Test	2
2	ClassATest.method2Test	1
6	ClassABTest.method1_method1Test	2

The next step selects one test case from the candidate set, according to their distance from the ordered test case set. Since our selection criterion is the Minimum distance function, the test case with the greatest minimum distance will be selected.

Jaccard distance is calculated considering the coverage string of the test cases. Table 4.10 displays the coverage string for each candidate set test case.

Table 4.10: Coverage strings for test cases.

Test case #	Coverage string
#3	0,0,1,0,0
#4	1,1,0,1,0
#5	0,0,0,0,1
#6	1,0,1,0,0

Considering that the ordered test case set is still empty, all the calculations will consider B from Equation 4.1 as 00000. Thus, for example, to calculate the distance for test case #3 (00100), the Equation 4.2 calculation is followed.

$$J(00100, 00000) = 1 - \frac{|00100 \cap 00000|}{|00100 \cup 00000|} \quad (4.2)$$

To calculate intersection (first line of the fraction), we perform a character by character comparison, using the AND operator, counting how many characters are both of value 1 and have the same index in the string, like Table 4.11 displays.

Table 4.11: Intersection calculation for Jaccard.

A	0	0	1	0	0
B	0	0	0	0	0
Result	x	x	x	x	x

In the case of Table 4.11, the result of the intersection is 0. To calculate the union (second line of the fraction), a comparison is made to check whether one or the other character in the same index from the string is of value 1, as shown in Table 4.12.

Table 4.12: Union calculation for Jaccard.

A	0	0	1	0	0
B	0	0	0	0	0
Result	x	x	✓	x	x

In the case of Table 4.12, the result of the union is 1. Now, the calculation is performed, dividing the result of the intersection by the result of the union, in this case, $1-0/1$, which is equal to 1. Thus, the distance from test case #3 to an empty test case is 1. After all calculations, Table 4.13 displays Jaccard distance values obtained for each test case.

Table 4.13: Jaccard distances to ordered set, first iteration

Candidate set elements	Distance to ordered set
#3	1
#4	1
#5	1
#6	1

Since all distance values are 1, the first one in the iteration will be selected. In this case, #3 is the first test case added to the ordered set. Now the execution goes back to the first step, to generate a new candidate set. Suppose that test case #1 is drawn. Then, test case #2, which does not add any coverage to the candidate set. In this way, the generate procedure stops and continues to the next step. The candidate set contains only test case #1.

In the next step, the Jaccard distance for test case #1 and the ordered set, which contains only test case #3, is calculated. Equation 4.3 shows how the calculation proceeds.

$$J(11000, 00100) = 1 - \frac{|11000 \cap 00100|}{|11000 \cup 00100|} = 1 - \frac{0}{3} = 1 \quad (4.3)$$

Since the candidate set only has one element, it is added to the ordered set of test cases, which now contains: #3 and #1. Going back to the generate procedure, suppose that test case #4 is drawn. The next one drawn is #6, which yields one additional coverage score to the set, and then #2, which does not add any coverage. Thus, the candidate set considered for the next step contains test cases #4 and #6. Distance values are calculated to each test case from the ordered set, as displayed in Table 4.14.

Minimum values are 0.34 for candidate test case #4 and 0.5 for candidate test case #6. The greatest minimum value is 0.5 and thus, test case #6 is selected and included to

Table 4.14: Jaccard distances to ordered set, second iteration

Candidate set elements	Distance to ordered set	
	#3	#1
#4	1	0.34
#6	0.5	0.67

the ordered set, which now contains test cases: #3, #1 and #6.

Back to the candidate set generation, suppose that test case #4 is drawn and then #2. Since #2 does not add any coverage to the candidate set, only #4 is considered for the next step. Considering that the candidate set only has one value, it is added to the ordered test cases set, which now contains: #3, #1, #6 and #4.

Going back to the generation step, suppose that #2 is drawn. After that, only #5 is left to be drawn. The candidate set contains #2 and #5. Distances of the candidate set elements to the ordered set elements are calculated as displayed in Table 4.15.

Table 4.15: Jaccard distances to ordered set, third iteration

Candidate set elements	Distance to ordered set			
	#3	#1	#6	#4
#2	1	0.5	1	0.67
#5	1	1	1	1

Minimum values are 0.5 for test case #2 and 1 for test case #5. The greatest minimum value between the candidate set and the ordered set is 1, for test case #5. In this way, it is added to the ordered set. Considering that only #2 remains as not yet ordered, it is finally added to the ordered set of test cases. The final test cases order for this technique would be #3, #1, #6, #4, #5 and #2.

4.6.2 HISTORY-BASED

History-based techniques as its name suggests, use historical test cases information as input for calculating the order of execution in the current build. Two history-based are included in our framework. They are most failures first and recent failures first. They can be used at the method-level and at the class-level test granularity. Thus, there are 4 different variations of history-based techniques available, as listed in Table 4.16.

For demonstration purposes of history-based approaches, a historical results matrix is displayed in Table 4.17, relative to the example project depicted in Figure 4.3. In Table 4.17, a cell value “1” means that the referred test case represented by the row has

Table 4.16: Variations for history-based TCP techniques.

Name	Test granularity
Most failures first at test method level	Method-level
Recent failures first at test method level	
Most failures first at test class level	Class-level
Recent failures first at test class level	

successfully been executed in the past execution represented by the column. A value of “0” means that the test case has failed in that execution. For example, test case #2 has failed in executions 1 and 3. Executions are counted chronologically. This means that execution 1 happened before 2, which happened before 3 and so on. The most recent execution in this example is execution 5.

Table 4.17: Historical results matrix.

#	Test case	Past execution results				
		1	2	3	4	5
1	ClassATest.method1Test	1	1	1	1	1
2	ClassATest.method2Test	0	1	0	1	1
3	ClassBTest.method1Test	1	0	1	1	1
4	ClassCTest.method1Test	1	1	1	1	0
5	ClassCTest.method2Test	1	1	0	1	1
6	ClassABTest.method1_method1Test	1	1	1	0	0

4.6.2.1 Most failures first

This technique is based on the total amount of failures that each test case has revealed in past executions. It is based on Algorithm 4. If a tie occurs between test cases, one of them is chosen randomly.

As an example execution of this technique, consider the historical results matrix in Table 4.17. The amount of failures per test case is represented in Table 4.18.

Since there is a tie between test cases #2 and #6, with 2 historical failures each, one of them is chosen randomly. Suppose that #6 is chosen. One possible execution order would be #6 and #2. Now there is another tie between test cases #3, #4 and #5. One possible execution order would be #6, #2, #4, #5 and #3. Since test case #1 has not failed yet, it is ordered for the last position. One possible final execution order for this scenario, using the most failures first technique would be: #6, #2, #4, #5, #3 and #1.

Algorithm 4: Most failures first algorithm

Input: Test cases execution results history matrix H , test cases set T

Output: Ordered test cases set T'

$T' \leftarrow \emptyset$;

while T not empty **do**

$MaxT \leftarrow \emptyset$;

$MaxH \leftarrow 0$;

foreach t in T **do**

$failuresAmount \leftarrow \text{getFailuresAmount}(t, H)$;

if $failuresAmount > MaxH$ **then**

$MaxT \leftarrow \emptyset$;

end if

if $failuresAmount \geq MaxH$ **then**

 Add t to $MaxT$;

$MaxH \leftarrow failuresAmount$;

end if

end foreach

$chosenT \leftarrow \text{random}(MaxT)$;

 Add $chosenT$ to T' ;

 Remove $chosenT$ from T ;

end while

Function $\text{getFailuresAmount}(t, H)$:

$failuresAmount \leftarrow 0$;

foreach r in H_t **do**

if $H_{tr} = 0$ **then**

$failuresAmount \leftarrow failuresAmount + 1$;

end if

end foreach

return $failuresAmount$

Table 4.18: Amount of historical failures per test case.

#	Test case	Amount of failures
1	ClassATest.method1Test	0
2	ClassATest.method2Test	2
3	ClassBTest.method1Test	1
4	ClassCTest.method1Test	1
5	ClassCTest.method2Test	1
6	ClassABTest.method1_method1Test	2

4.6.2.2 Recent failures first

This technique is based on the freshness of failures for each test case. If a test case failed recently, it receives a bigger weight when calculating its score than if it failed a long time ago. This means that recently failed test cases have bigger priority. Ties are resolved randomly.

The score for each test case is based on the execution results history, which contains all executions for each test case and its result. If a test case failed in the $n - th$ execution and $n - 1 - th$ execution, its score is $n + (n - 1)$, for example. Test cases are then ordered according to the descending value of scores. The score is calculated according to Equation 4.4.

$$scoreT_i = \sum_{j=1}^{exec_i} freshness_{ij}, \text{ where } freshness_{ij} = \begin{cases} j, & \text{if } H_{ij} = 0 \\ 0, & \text{if } H_{ij} = 1 \end{cases} \quad (4.4)$$

Where i is the $i - th$ test case, j is the $j - th$ execution of the test case, H_{ij} is the result of the $j - th$ execution of the $i - th$ test case in the execution results history matrix. $exec$ is the total of executions for the test case. Algorithm 5 details how this technique works.

As an example of this technique, consider the historical results matrix in Table 4.17. The calculated score for each test case is represented in Table 4.19.

Considering the scores displayed in Table 4.19, the execution order in this scenario, using this technique would be: #6, #4, #2, #5, #3 and #1.

4.6.3 MODIFICATION-BASED

Modification-based TCP techniques use source code modification information to determine the order that test cases will be executed. We include two approaches in this category, which are also based on the total and additional coverage approach, described in Section

Algorithm 5: Recent failures first algorithm

Input: Test cases execution results history matrix H , test cases set T

Output: Ordered test cases set T'

$T' \leftarrow \emptyset$;

while T not empty **do**

$MaxT \leftarrow \emptyset$;

$MaxScore \leftarrow 0$;

foreach t in T **do**

$score \leftarrow \text{getScore}(t, H)$;

if $score > MaxScore$ **then**

$MaxT \leftarrow \emptyset$;

end if

if $score \geq MaxScore$ **then**

 Add t to $MaxT$;

$MaxScore \leftarrow score$;

end if

end foreach

$chosenT \leftarrow \text{random}(MaxT)$;

 Add $chosenT$ to T' ;

 Remove $chosenT$ from T ;

end while

Function $\text{getScore}(t, H)$:

$score \leftarrow 0$;

foreach j in H_t **do**

if $H_{tj} = 0$ **then**

$score \leftarrow score + j$;

end if

end foreach

return $score$

Table 4.19: Recent failures score of test cases.

#	Test case	Score
1	ClassATest.method1Test	0
2	ClassATest.method2Test	4
3	ClassBTest.method1Test	2
4	ClassCTest.method1Test	5
5	ClassCTest.method2Test	3
6	ClassABTest.method1_method1Test	9

4.6.1. Included approaches are named total diff coverage and additional diff coverage. Two consecutive versions of the project being tested are considered in these approaches. This means that the source code of the current version of the project is compared with the source code of the previous version.

The two approaches in this category are offered in different versions, regarding its test and source code granularity. Two variations are included regarding the test granularity, method-level and class-level. Regarding source code granularity, method and class are available as option. In this way, there are 8 (2 x 2 x 2) modification-based techniques included, which are listed in Table 4.20.

Table 4.20: Variations of modification-based TCP techniques.

Name	Base approach	Source code granularity	Test granularity
Total method diff coverage at test method level	Total coverage	Method	Method-level
Total class diff coverage at test method level		Class	
Total method diff coverage at test method level	Total coverage	Method	Class-level
Total class diff coverage at test method level		Class	
Additional method diff coverage at test method level	Additional coverage	Method	Method-level
Additional class diff coverage at test method level		Class	
Additional method diff coverage at test method level	Additional coverage	Method	Class-level
Additional class diff coverage at test method level		Class	

For demonstration purposes of included modification-based approaches, consider the coverage matrix displayed in Table 4.3. Furthermore, consider that the following methods were modified since the last version of the project: ClassA.method2 and ClassC.method1.

4.6.3.1 Total diff coverage

This technique is based on the total coverage of source code elements that were changed since the last version. In this way, coverage information is also used. Test cases are ordered according to the descending amount of total modified elements covered. Ties are resolved randomly. Algorithm 6 details the approach.

As an example of this technique, consider the coverage matrix in Table 4.3 and the modified elements `ClassA.method2` and `ClassC.method1`. The diff score for each test case is displayed in Table 4.21.

Based on the scores from Table 4.21, one possible final order for the test cases is: #4, #2, #1, #3, #6 and #5.

4.6.3.2 Additional diff coverage

The additional diff coverage technique is based on the additional coverage technique, with the addition of the modification information. In this way, tests are ordered considering the additional coverage of modified elements provided to already ordered tests. If all modified elements have already been covered by already ordered tests, ordering continues with the traditional additional coverage algorithm. Ties are resolved randomly. Algorithm 7 details the approach.

Considering the coverage matrix displayed in Table 4.3 and the modified elements `ClassA.method2` and `ClassC.method1`, the initial additional diff coverage for each test case is displayed in Table 4.22.

According to Algorithm 7, the first chosen test case would be #4, since it yields the greatest amount of additional coverage of modified elements to the ordered set of test cases, which is empty, since this is the first iteration. After adding test case #4 to the ordered test cases set, the covered elements list contains: `ClassA.method1`, `ClassA.method2` and `ClassC.method1`. Since all modified elements have been already covered by #4, execution continues considering the traditional additional coverage algorithm. The additional diff coverage for each test case is recalculated and displayed in Table 4.23.

Since there is a tie between the additional coverage of test cases #3 and #5, one of them is chosen randomly. Suppose that #3 is chosen. Now, the covered elements list contains: `ClassA.method1`, `ClassA.method2`, `ClassB.method1` and `ClassC.method1`. The additional coverage for each test case is recalculated and displayed in Table 4.24.

Algorithm 6: Total diff coverage algorithm

Input: Test cases coverage matrix C , test cases set T , modified source code elements M

Output: Ordered test cases set T'

$T' \leftarrow \emptyset$;

while T not empty **do**

$MaxT \leftarrow \emptyset$;

$MaxC \leftarrow 0$;

foreach t in T **do**

$totalCoverage \leftarrow \text{getTotalDiffCoverage}(t, C, M)$;

if $totalCoverage > MaxC$ **then**

$MaxT \leftarrow \emptyset$;

end if

if $totalCoverage \geq MaxC$ **then**

 Add t to $MaxT$;

$MaxC \leftarrow totalCoverage$;

end if

end foreach

$chosenT \leftarrow \text{random}(MaxT)$;

 Add $chosenT$ to T' ;

 Remove $chosenT$ from T ;

end while

Function $\text{getTotalDiffCoverage}(t, C, M)$:

$totalDiffCoverage \leftarrow 0$;

foreach covered element e of t in C **do**

if $e \in M$ **then**

$totalDiffCoverage \leftarrow totalDiffCoverage + 1$;

end if

end foreach

return $totalDiffCoverage$

Algorithm 7: Additional diff coverage algorithm

Input: Test cases coverage matrix C , test cases set T , modified source code elements M

Output: Ordered test cases set T'

$T' \leftarrow \emptyset$;

$L \leftarrow \emptyset$;

while T not empty **do**

$MaxAdditionalT \leftarrow \emptyset$;

$MaxAdditionalC \leftarrow 0$;

foreach t in T **do**

$additionalC \leftarrow \text{getAdditionalDiffCoverage}(t, C, M, L)$;

if $additionalC > MaxAdditionalC$ **then**

$MaxAdditionalT \leftarrow \emptyset$;

end if

if $additionalC \geq MaxAdditionalC$ **then**

 Add t to $MaxAdditionalT$;

$MaxAdditionalC \leftarrow additionalC$;

end if

end foreach

if $MaxAdditionalC = 0$ **then**

$L \leftarrow \emptyset$;

else

$t \leftarrow \text{random}(MaxAdditionalT)$;

 Add covered elements by t to L ;

 Add t to T' ;

 Remove t from T ;

 Remove covered elements by t from M ;

end if

end while

Function $\text{getAdditionalDiffCoverage}(t, C, M, L)$:

$additionalDiffC \leftarrow 0$;

foreach covered element e of t in C **do**

if M is not empty **then**

if $e \notin L$ and $e \in M$ **then**

$additionalDiffC \leftarrow additionalDiffC + 1$;

end if

else

if $e \notin L$ **then**

$additionalDiffC \leftarrow additionalDiffC + 1$;

end if

end if

end foreach

return $additionalDiffC$

Table 4.21: Total diff score for test cases of the example application.

#	Test case	Score
1	ClassATest.method1Test	1
2	ClassATest.method2Test	1
3	ClassBTest.method1Test	0
4	ClassCTest.method1Test	2
5	ClassCTest.method2Test	0
6	ClassABTest.method1_method1Test	0

Test case #5 has additional diff coverage of 1, while the remaining test cases have 0. In this way, it is chosen as the next test. The ordered test case set contains: #4, #3 and #5. The covered elements list contains: ClassA.method1, ClassA.method2, ClassB.method1, ClassC.method1 and ClassC.method2. Considering that all elements of the source code are already been covered by the already ordered tests, the coverage list is emptied and execution continues. The additional coverage for each test case is recalculated and displayed in Table 4.25.

Since there is a tie between test cases #1 and #6, one of them is chosen randomly. Suppose that #1 is chosen. The covered elements list contains: ClassA.method1 and ClassA.method2. The ordered test case set contains: #4, #3, #5 and #1. The process continues until all test cases have been ordered. One final order for this scenario would be: #4, #3, #5, #1, #2 and #6.

4.6.4 SIMILARITY-BASED

Similarity-based TCP techniques use information about test cases to calculate their similarity. We include two approaches in this category. They are the Farthest-first Ordered Sequence (FOS) and Greed-aided-clustering Ordered Sequence (GOS), both proposed by Fang et al. (2013). The two approaches in this category are offered in different versions, regarding the granularity of the test cases, as displayed in Table 4.26.

These techniques use the notion of ordered sequence of program elements to calculate the similarities between test cases.

Execution profile is the summary of source code elements executed by a test case. Based on execution profiles, these techniques use the frequency profile, which contains the counting of how many times each element was executed by each test case. Furthermore, an ordered sequence is the ordered frequency profile of a test case. In this way, consider

Table 4.22: Additional diff score for test cases of the example application, first iteration.

#	Tests cases	Additional diff coverage
1	ClassATest.method1Test	1
2	ClassATest.method2Test	1
3	ClassBTest.method1Test	0
4	ClassCTest.method1Test	2
5	ClassCTest.method2Test	0
6	ClassABTest.method1_method1Test	0

Table 4.23: Additional diff score for test cases of the example application, second iteration.

#	Tests cases	Additional diff coverage
1	ClassATest.method1Test	0
2	ClassATest.method2Test	0
3	ClassBTest.method1Test	1
5	ClassCTest.method2Test	1
6	ClassABTest.method1_method1Test	0

Table 4.24: Additional diff score for test cases of the example application, third iteration.

#	Tests cases	Additional diff coverage
1	ClassATest.method1Test	0
2	ClassATest.method2Test	0
5	ClassCTest.method2Test	1
6	ClassABTest.method1_method1Test	0

Table 4.25: Additional diff score for test cases of the example application, fourth iteration.

#	Tests cases	Additional diff coverage
1	ClassATest.method1Test	2
2	ClassATest.method2Test	1
6	ClassABTest.method1_method1Test	2

Table 4.26: Variations of similarity-based techniques.

Name	Test granularity
Farthest-first Ordered Sequence	Method-level
Greed-aided-clustering Ordered Sequence	
Farthest-first Ordered Sequence	Class-level
Greed-aided-clustering Ordered Sequence	

the example software depicted in Figure 4.3.

Using the coverage information, represented in Table 4.3, it is possible to derive the execution profile of `ClassATest.method1Test`, for example.

The execution profile can be represented using a binary string, containing as much characters as there are source code elements in the program being tested. For this example scenario, the string representing the execution profile of the test cases would have 5 characters. Each character represents the execution (“1”) or not (“0”) of the source code element, as displayed in Table 4.27.

Table 4.27: Execution profile for test cases.

#	Test case	Execution profile
1	ClassATest.method1Test	{1, 1, 0, 0, 0}
2	ClassATest.method2Test	{0, 1, 0, 0, 0}
3	ClassBTest.method1Test	{0, 0, 1, 0, 0}
4	ClassCTest.method1Test	{1, 1, 0, 1, 0}
5	ClassCTest.method2Test	{0, 0, 0, 0, 1}
6	ClassABTest.method1_method1Test	{1, 0, 1, 0, 0}

Based on these execution profiles, the frequency profile can also be represented as a string, where each character represents how many times that source code element was executed by that test case. Table 4.28 display frequency profile values created to support our examples. To clarify, as an example, test case #1 executes the first source code element 2 times and the second source code element 1 time.

Having the frequency profile for each test case, the ordered sequence is derived by simply ordering the characters for the frequency profile string and displaying their original

Table 4.28: Frequency profile for test cases.

#	Test case	Frequency profile
1	ClassATest.method1Test	{2, 1, 0, 0, 0}
2	ClassATest.method2Test	{0, 3, 0, 0, 0}
3	ClassBTest.method1Test	{0, 0, 1, 0, 0}
4	ClassCTest.method1Test	{5, 2, 0, 1, 0}
5	ClassCTest.method2Test	{0, 0, 0, 0, 1}
6	ClassABTest.method1_method1Test	{2, 0, 2, 0, 0}

index from the frequency profile, as displayed in Table 4.29. These ordered sequences will be used to display examples of each technique in this category in the next sections. Consider for example, the test case #4, where the frequency profile is 5, 2, 0, 1, 0. The value for each index in the set is: 1: 5, 2: 2, 3: 0, 4: 1, 5:0. In this way, after ordering the frequency profile for this test case, the resulting (index) sequence is: 3, 5, 4, 2 and 1.

Table 4.29: Ordered sequence for test cases.

#	Test case	Ordered sequence
1	ClassATest.method1Test	{3, 4, 5, 1, 2}
2	ClassATest.method2Test	{1, 3, 4, 5, 2}
3	ClassBTest.method1Test	{1, 2, 4, 5, 3}
4	ClassCTest.method1Test	{3, 5, 4, 2, 1}
5	ClassCTest.method2Test	{1, 2, 3, 4, 5}
6	ClassABTest.method1_method1Test	{2, 4, 5, 1, 3}

4.6.4.1 Fartherst-first ordered sequence

The Fartherst-first Ordered Sequence (FOS) technique as its name suggests, uses the ordered sequence information of test cases as input to order them. Its goal is to give priority to test cases that are the most diverse between themselves, so that by executing the test set in that order, a diversity of source code elements are covered by the tests.

The algorithm for this technique works by firstly selecting a test case that can yield the greatest code coverage. Then, next test cases are selected based on two distances, measured by Levenshtein Distance, which is explained in Definition 3. One is the distance between two test cases. The other is related to the distance between a test case and the set of already ordered test cases. To calculate the latter, first all the distances between the candidate test case and already ordered test cases are calculated. Then, the minimum distance is used as representative for the distance between the candidate test case and the

ordered set. Three different strategies can be used for this distance, namely, the minimum, the average and the maximum distance. We chose the minimum distance based on the results of an empirical study conducted by Fang et al. (2013). The algorithm for this technique is displayed in Algorithm 8.

Definition 3. Levenshtein distance is a metric proposed by Levenshtein (1965), which is used to determine the minimum number of operations needed to transform one string into another. Allowed operations are insertion, deletion and substitution. This metric is used in this work to represent similarity between two strings.

Considering the ordered sequence for the test cases of our example scenario, displayed in Table 4.29, an execution of this technique would occur like the following.

The first test case to be selected would be the one with the greatest total coverage, in this case, test case #4. From now on, it is needed to consider the Levenshtein distances between the ordered set of test cases and all other not yet ordered test cases. The distances between each test case ordered sequence is displayed in Table 4.30. These will be considered in the following iterations of the technique.

The ordered test case set contains test case #4. The greatest minimum distance between a test case from the not yet ordered set and the ordered test is 5, provided by test cases #5 and #6. In this case, two test cases yields this maximum distance. The one that appears first is selected, since no tie resolving mechanism is defined in the original algorithm. In this way, test case #5 is added to the ordered test case set, which now contains test cases #4 and #5. The minimum distances for the next iteration are displayed in Table 4.31. Those were extracted from Table 4.30.

The greatest minimum distance is 4. Test cases that have this value are #1, #2 and #3. The one that appears first is selected, which is #1. Now the ordered test set contains #4, #5 and #1. The minimum distances for the next iteration are displayed in Table 4.32.

The greatest minimum distance now is still 4. Test cases with this value are #2 and #3. The one that appears first is selected, which is #2. Now the ordered test set contains #4, #5, #1 and #2. The minimum distances for the next iteration are displayed in Table 4.33.

The greatest minimum distance now is still 4. Test case with this value is #3, which is added to the ordered test set. Now the ordered test set contains #4, #5, #1, #2 and #3.

Algorithm 8: Farthest-first ordered sequence base algorithm

Input: Test coverage matrix C , test cases set T , test cases ordered sequence set O

Output: Ordered test cases set T'

$T' \leftarrow \emptyset$;

$t \leftarrow$ test with the highest total coverage in C ;

Add t to T' ;

Remove t from T ;

while T not empty **do**

$t \leftarrow$ selectNextTestCase(T, T', O);

 Add t to T' ;

 Remove t from T ;

end while

Function selectNextTestCase(T, T', O):

$maxDistanceT \leftarrow \emptyset$;

$maxDistance \leftarrow -1$;

foreach t in T **do**

$minDistance \leftarrow$ a big value;

foreach t' in T' **do**

$orderedSequenceT \leftarrow O_t$;

$orderedSequenceT' \leftarrow O_{t'}$;

$distance \leftarrow$ levenshtein($orderedSequenceT, orderedSequenceT'$);

if $distance < minDistance$ **then**

$minDistance \leftarrow distance$;

end if

end foreach

if $minDistance > maxDistance$ **then**

$maxDistance \leftarrow minDistance$;

$maxDistanceT \leftarrow t$;

end if

end foreach

return $maxDistanceT$

Table 4.30: Levenshtein distance value between test cases.

Test cases number	#1	#2	#3	#4	#5	#6
#1	0	4	5	4	5	2
#2	4	0	2	4	4	5
#3	5	2	0	4	3	4
#4	4	4	4	0	5	5
#5	5	4	3	5	0	5
#6	2	5	4	5	5	0

Table 4.31: Minimum distances between test cases and ordered set, first iteration.

Test case	Minimum distance value	Minimum distance test case
#4	4	#1, #2, #3
#5	3	# 3

Since only one test case remains in the not yet ordered test set, it is added to the ordered set, completing the execution with the final test case order for this technique being #4, #5, #1, #2, #3 and #6.

4.6.4.2 Greed-aided-clustering ordered sequence

The Greed-aided-clustering Ordered Sequence (GOS) technique also uses ordered sequence information of test cases as input to order them. It combines the strategies of the additional coverage strategy and clustering of test cases. The algorithm takes as input the amount of clusters to be used, ordered sequence information for the test cases, test coverage information and the test suite to be prioritized. It works by creating single test cases clusters initially. Then clusters are merged with another cluster that has the minimum distance among all others to the current cluster. Distance is calculated using Levenshtein Distance, between the ordered sequences of a pair of clusters. This process is repeated until the target amount of clusters, which is given as input to the algorithm, is hit. Once this happens, each cluster is individually prioritized using the additional coverage algorithm. The next step is to iteratively select one test case from each cluster to be included in the final prioritized order until all test cases have been included. Algorithm 9 details the approach.

Considering the ordered sequence for the test cases of our example scenario, displayed in Table 4.29, an execution of this technique would occur like the following.

Initially, six test case clusters are created, each one containing one single test case, as

Algorithm 9: Greed-aided-clustering ordered sequence base algorithm

Input: Test coverage matrix C , test cases set T , test cases ordered sequence set O , amount of clusters n

Output: Ordered test cases set T'

$T' \leftarrow \emptyset$;

$K \leftarrow |T|$ single test case clusters;

while $|K| \leq n$ **do**

 | Find a pair of clusters with minimum Levenshtein distance, using O ;

 | Merge the pair of clusters;

 | Remove both clusters from K ;

 | Add the merged cluster to K ;

end while

foreach k *in* K **do**

 | Prioritize k test cases using additional coverage and C ;

end foreach

$i \leftarrow 0$;

while $|K| > 0$ **do**

 | $t' \leftarrow$ first test case from K_i ;

 | Remove t' from K_i ;

if $|K_i| = 0$ **then**

 | Remove K_i from K ;

end if

 | Add t' to T' ;

$i \leftarrow i + 1$;

if $i \geq |K|$ **then**

 | $i \leftarrow 0$;

end if

end while

Table 4.32: Minimum distances between test cases and ordered set, second iteration.

Test case	Minimum distance value	Minimum distance test case
#4	4	#2, #3
#5	3	#3
#1	2	#6

Table 4.33: Minimum distances between test cases and ordered set, third iteration.

Test case	Minimum distance value	Minimum distance test case
#4	4	#3
#5	3	#3
#1	2	#6
#2	2	#3

displayed in Table 4.34.

Table 4.34: Test cases clusters, first iteration.

Cluster	Test cases
1	#1
2	#2
3	#3
4	#4
5	#5
6	#6

The second step of the algorithm is to merge clusters until the target amount of clusters, provided as input, is hit. In our case, suppose that our target amount of clusters is 3.

To merge clusters, it is necessary to calculate the distance between them, considering the ordered sequence information. Considering that clusters only have one test case, the distance between clusters is the same as the distance between the test cases, as displayed in Table 4.35.

There are two pairs of clusters with the minimum distance (2) between them: #1, #6 and #2, #3. As in the Farthest-first Ordered Sequence algorithm, no tie resolving mechanism is used in the original algorithm. In this case, the first pair found by the algorithm is chosen to be merged. Thus, clusters #1 and #6 are merged into one cluster. The updated clusters and their test cases are displayed in Table 4.36.

Considering that there is a cluster (#1) with more than one test case, it is needed

Table 4.35: Test cases clusters distances, first iteration.

Clusters distance	#1	#2	#3	#4	#5	#6
#1	0	4	5	4	5	2
#2	4	0	2	4	4	5
#3	5	2	0	4	3	4
#4	4	4	4	0	5	5
#5	5	4	3	5	0	5
#6	2	5	4	5	5	0

Table 4.36: Test cases clusters, second iteration.

Cluster	Test cases
1	#1, #6
2	#2
3	#3
4	#4
5	#5

to merge the ordered sequence of both test cases (#1 and #6), to represent the ordered sequence of the whole cluster.

To generate the new ordered sequence information, it is needed to add together the frequency profiles of test cases that are in the same cluster. Thus, considering the original frequency profiles, as displayed in Table 4.28, the updated frequency profile for each cluster is displayed in Table 4.37.

Table 4.37: Clusters frequency profile, second iteration.

Cluster #	Test cases #	Frequency profile
#1	#1, #6	{4, 1, 2, 0, 0}
#2	#2	{0, 3, 0, 0, 0}
#3	#3	{0, 0, 1, 0, 0}
#4	#4	{5, 2, 0, 1, 0}
#5	#5	{0, 0, 0, 0, 1}

Now that we have the updated frequency profile for each cluster, it is possible to generate the ordered sequence for them, as displayed in Table 4.38.

The updated Levenshtein distances between each pair of clusters, considering their ordered sequences, is displayed in Table 4.39.

Considering the updated distances between each pair of clusters, the minimum distance is 2, between clusters #3 and #2. In this way, both clusters are merged into one. The

Table 4.38: Clusters ordered sequences, second iteration.

Cluster #	Test cases #	Ordered sequence
#1	#1, #6	{4, 5, 2, 3, 5}
#2	#2	{1, 3, 4, 5, 2}
#3	#3	{1, 2, 4, 5, 3}
#4	#4	{3, 5, 4, 2, 1}
#5	#5	{1, 2, 3, 4, 5}

Table 4.39: Test cases clusters distances, second iteration.

Clusters distance	#1	#2	#3	#4	#5
#1	0	5	5	4	4
#2	5	0	2	4	4
#3	5	2	0	4	3
#4	4	4	4	0	5
#5	4	4	3	5	0

updated clusters and their test cases are displayed in Table 4.40.

Table 4.40: Test cases clusters, third iteration.

Cluster	Test cases
1	#1, #6
2	#2, #3
3	#4
4	#5

Now the frequency profiles of each cluster, as displayed in Table 4.41, are used to recalculate their ordered sequences, displayed in Table 4.42.

The updated Levenshtein distances between each pair of cluster, considering their ordered sequences, is displayed in Table 4.43.

Considering the updated distances between each pair of clusters, the minimum distance is 4, between clusters #1 and #3. In this way, both clusters are merged into one. The updated clusters and their test cases are displayed in Table 4.44.

Considering that the target amount of clusters is hit, which is 3, the execution continues to the next step. In this step, test cases are prioritized within each cluster, considering their coverage and using the additional coverage algorithm. The coverage information is displayed in Table 4.3.

For cluster #1, the test case that yields the greatest coverage needs to be chosen as the first one. In this case, #4 is chosen. The additional coverage score for test cases #1

Table 4.41: Clusters frequency profile, third iteration.

Cluster #	Test cases #	Frequency profile
#1	#1, #6	{4, 1, 2, 0, 0}
#2	#2, #3	{0, 3, 1, 0, 0}
#3	#4	{5, 2, 0, 1, 0}
#4	#5	{0, 0, 0, 0, 1}

Table 4.42: Clusters ordered sequences, third iteration.

Cluster #	Test cases #	Ordered sequence
#1	#1, #6	{4, 5, 2, 3 1}
#2	#2, #3	{1, 4, 5, 3, 2}
#3	#4	{3, 5, 4, 2, 1}
#4	#5	{1, 2, 3, 4, 5}

and #6 are recalculated. #1 gives 0 additional coverage and #6 gives 1. In this way, test case #6 is chosen. Thus, the order for cluster #1 is: #4, #6 and #1.

For cluster #2, both test cases (#2 and #3) yield the same amount of coverage, which is 1. In this way, one is chosen randomly. One possible order for cluster #2 is: #3 and #2.

For cluster #3, there is only one test case (#5). In this way, there is no need to apply the additional coverage algorithm to order it.

The next step of GOS algorithm is to iteratively select one test case from each ordered cluster to be added to the final ordered test case set. In this way, the final order for our example scenario, using the GOS algorithm is: #4, #3, #5, #6, #2 and #1.

4.7 NEW TECHNIQUES IMPLEMENTATION

Considering the requirements of the designed modules, the framework should be extensible, providing ways of implementing new TCP techniques. This extensibility may help researchers willing to conduct experiments with TCP techniques, in the sense that they may need to implement their new TCP approaches. With this scenario in mind, we developed an API that all TCP techniques in the architecture must adhere.

We reduce TCP techniques to two base strategies, which we call Default and Additional, respectively.

In the Default approach, all test cases can be directly compared to each other, ac-

Table 4.43: Test cases clusters distances, third iteration.

Clusters distance	#1	#2	#3	#4
#1	0	5	4	4
#2	5	0	5	4
#3	4	5	0	5
#4	4	4	5	0

Table 4.44: Test cases clusters, fourth iteration.

Cluster	Test cases
1	#1, #6, #4
2	#2, #3
3	#5

ording to some criterion. For example, using the total amount of source code elements covered. Based on these comparisons, a set of test cases can be ordered before execution.

In the Additional approach, test cases cannot be compared to each other, because to determine the order, it is needed to analyze the already ordered test cases. For example, using the additional amount of source code elements covered, it is needed to calculate the source code elements of already ordered test cases, before determining the order of the next test case.

Considering the explained two approaches and examples, TCP techniques must follow the design depicted Figure 4.4, implementing the interface Orderer and if using the Default strategy, implementing also the Comparator interface. In this example, TotalCoverage and AdditionalCoverage are two implemented TCP techniques.

4.8 FINAL CONSIDERATIONS

In this chapter we discussed the design of the proposed framework to support TCP techniques usage and experimentation. It was designed according to problems identified in the literature. This Chapter answers our second research question "How to create a framework to support TCP usage and experimentation?". In the next chapter, the implementation of the framework will be discussed.

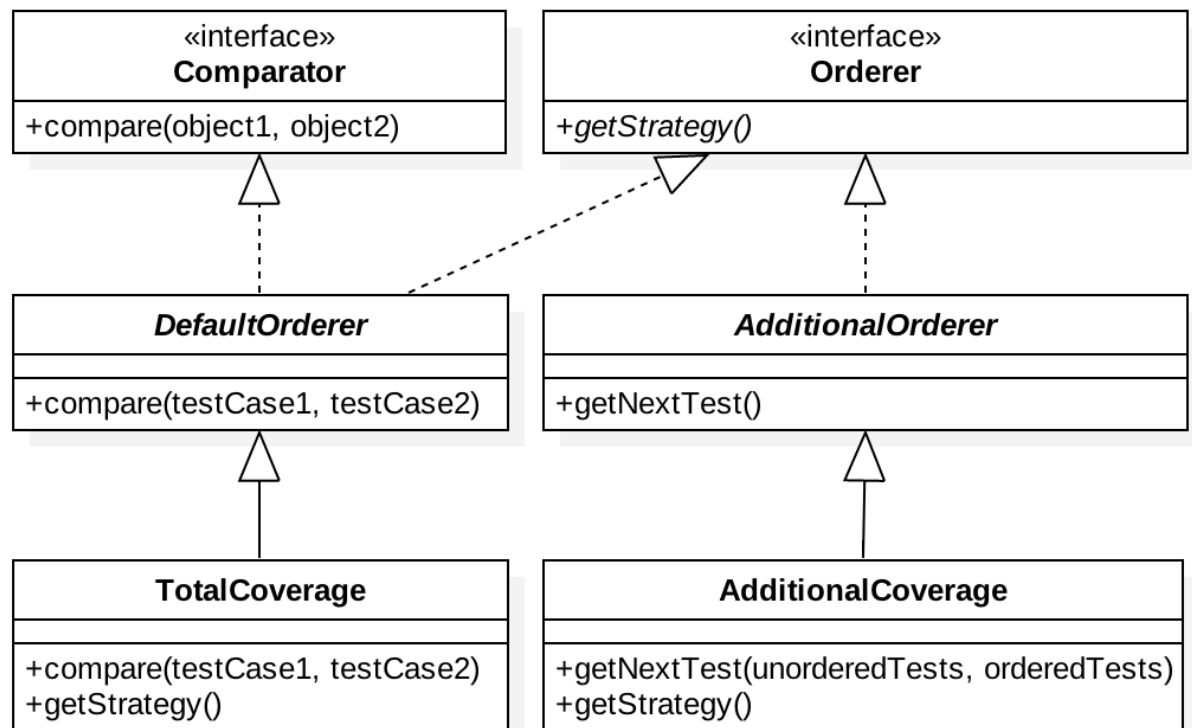


Figure 4.4: TCP techniques API for Optimus Framework.

5 OPTIMUS FRAMEWORK IMPLEMENTATION

Based on the design described in the last Chapter, we implemented the architecture.

Considering the requirement of integration with an automated build tool, Maven was used. In this way, as the Maven framework itself uses the Java language, we also focused on it.

The test phase of a Maven build life cycle is conducted by the Maven Surefire plugin. Maven Surefire allow its functionalities extension through test providers. For this reason, we implemented a Maven Surefire test provider, which is responsible for providing test cases to be executed by the JUnit framework.

The JUnit framework is the most used test framework for Java programming language. A typical JUnit test is composed of a test class with test methods. Test classes can be grouped in test suites. Normally, all test methods of a test class are executed one after another. In this way, by simply using the default JUnit framework, it is not possible to execute test methods from different classes arbitrarily, as requested by our requirements. However, as the test provider is responsible for providing test cases to the JUnit framework, we can provide independent test methods to be executed, fulfilling this requirement.

The architecture implementation was divided in 8 different modules, depicted in Figure 5.1, according to the attributions of each one, which will be described next.

5.1 OPTIMUS-COMMON

The main goal of this module is to provide access to source code that is common to other modules, including domain classes that model our environment. Moreover, it also provides the feature to extract information from tested projects, like the location of each source code class and its methods, for example.

5.2 OPTIMUS-FRAMEWORK

This module implements a Maven plugin, which is executed during the regression test phase of the Maven life cycle. It is also a wrapper to use our framework. In this way, the

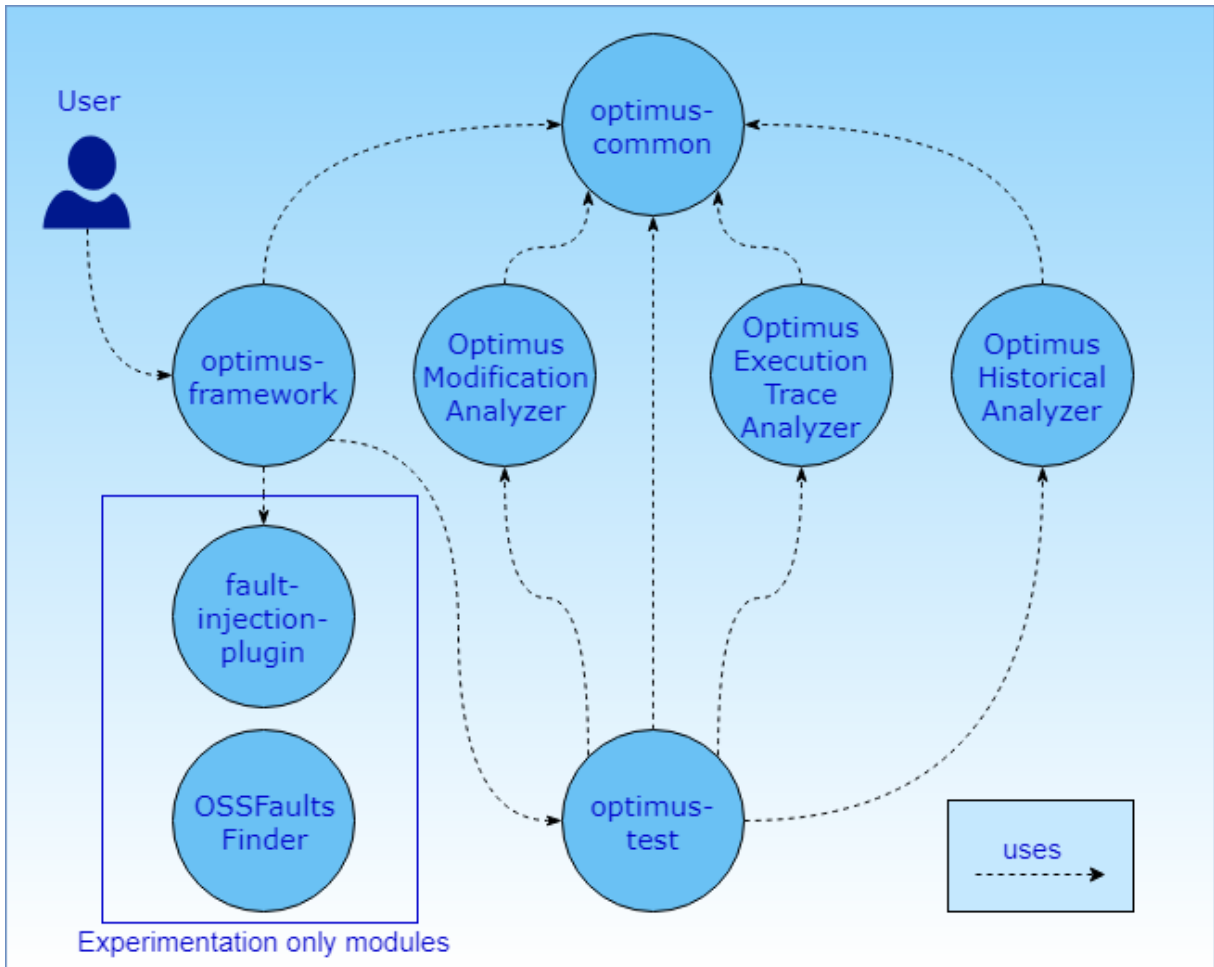


Figure 5.1: Optimus Framework implemented modules.

user only needs to add this dependency to his project in order to use our provided features. Another responsibility of this module is to orchestrate the execution of the prioritization, by calling the `optimus-test` provider, described next.

5.3 OPTIMUS-TEST

This module implements the Maven Surefire provider. It is responsible for providing test cases to be executed by the JUnit framework.

Considering that it provides the test cases to be executed, TCP techniques are applied in this module. In this way, a prioritization technique is applied to order the test case set before it is sent to the JUnit framework.

Figure 5.2 depicts the testing process. It starts in Maven Surefire. In this step, tests of the project are discovered and the entire testing environment is prepared. Maven Surefire then send the discovered test cases to a provider. In this case, it calls our provider,

OptimusProvider. Our provider then calls the TestsSorter to sort the test cases received. TestsSorter calls a TCP technique, which implements the Orderer interface. The ordered tests are returned to our provider, which calls the JUnitExecutor to provide JUnit the set of ordered tests to be executed. JUnit calls all the JUnitListeners to broadcast test execution messages. When all tests are executed, control is given back to our provider, which reports to MavenSurefire that the execution is over.

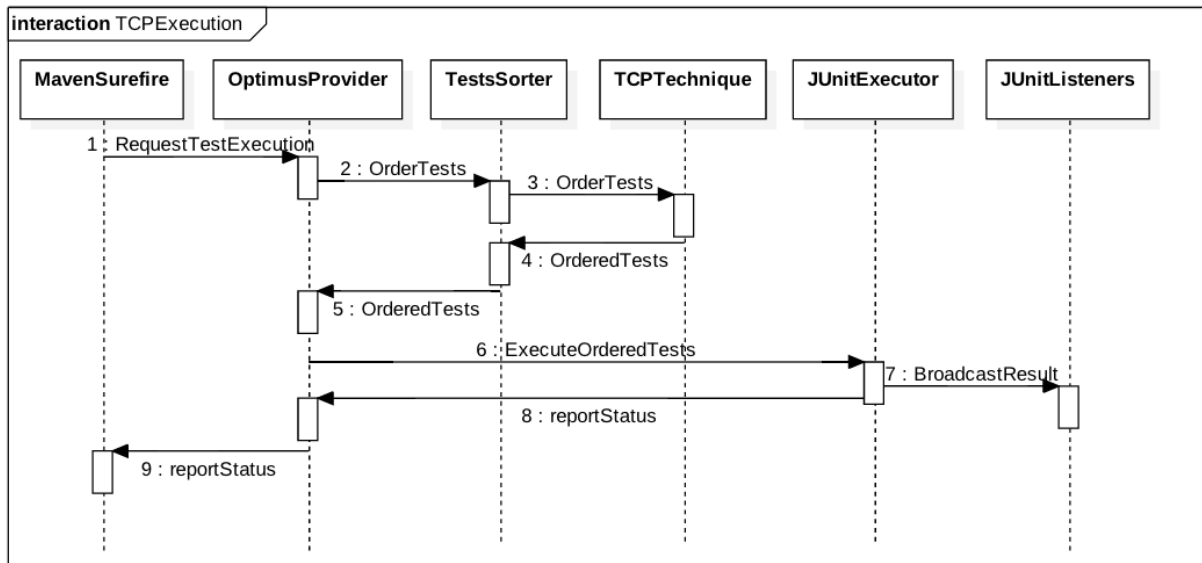


Figure 5.2: Sequence diagram of test process within Optimus Framework.

5.4 OSS FAULTS FINDER

This module is currently independent from the others, meaning that the user must execute it manually. However, it can be used to support the experimentation process of TCP techniques by downloading open source project versions that contain failing tests.

This module makes use of the Travis CI API to query for open source projects historical data. If it finds a build that contains failing tests, it automatically downloads the source code from the Github repository. These downloaded source code can be used to measure and compare the effectiveness of TCP techniques through experiments.

5.5 FAULT INJECTION PLUGIN

This module is responsible for injecting (seeding) mutation faults into the source code of a project. The mutated source code can then be used during the experimentation process.

We use the PIT Mutation Test tool to generate mutated versions of the source code of the project. If the tests of a project can detect a mutation, we say that the mutation can be killed. In this way, the mutated source code can be used as a faulty version of the project.

Different mutation operators can be applied by PIT. We use the seven default ones listed below.

- Conditional Boundary Mutator: this mutation operator swaps expression conditions in the source code. As an example, a condition `if(x >= y)` will be changed to `if(x > y)`.
- Increments mutator: this mutation operator acts on increments and decrements in the source code. An increment becomes a decrement and vice versa. As an example, `x++` becomes `x--`.
- Invert negatives mutator: this mutation operator inverts negative numbers/variables in the source code, changing them to positive. As an example, `-x` becomes `x`.
- Math mutator: this mutation operator swaps math operators in the source code, replacing them with an opposite operator. For example, `x=y*2` becomes `x=y/2`.
- Negate conditional mutator: this mutation operator negates conditional expressions in the source code. As an example, `if(a == b)` becomes `if(a != b)`.
- Return values mutator: this mutation operator acts on return values of methods of the source code. As an example, a method that returns an integer value of 0 is changed to return 1.
- Void method calls mutator: this mutation operator removes method calls from the structure of other methods. In this way, if there is a method call that does not have a return value, when mutated, it will be removed from the execution flow.

Equivalence of mutants is a problem that happens with mutation testing, when generated mutants are equivalent and not effective. This is not a problem in our case because we are not interested on measuring the effectiveness of the test suite itself, but rather on creating faulty versions of the source code.

To prevent inserting mutants that override themselves, we limited the addition of one mutant per method of classes of the project being seeded. Moreover, PIT mutation tool perform the mutation process on the bytecode generated by the compiler. In this way, we use a decompiler to generate the source code text for mutations.

The process of fault seeding with this module is depicted in Figure 5.3 and explained below.

- Starting from the original source code classes, PIT runs and generates mutations, which are contained in Java bytecode files. One file is generated for each generated mutation.
- Mutations that can be killed, i.e. that can be detected by existing test cases, are selected and the remaining are discarded.
- For each generated bytecode file that contains a killed mutation:
 - Bytecode is transformed into Java source code text. In our case, we are using Procyon Decompiler¹, version 4.0.0.Final.
 - If no mutation was already added to the method of this mutation, merge this mutation to the source code of the seeded project.
- Generate a copy of the original project, containing the seeded mutation faults.

5.6 OPTIMUS COVERAGE ANALYZER

This module is responsible for collecting and providing project coverage information as input for TCP techniques. Coverage is collected using the JaCoCo² coverage agent during run-time. In this way, every time the project is built, for every test case executed, coverage information is collected and recorded in the artifacts repository.

Coverage information provided by this analyzer is relative to the previous build of the software. This is done to prevent the double execution of the test suite, since the test cases need to be executed with the updated software if updated coverage information is needed.

¹<https://bitbucket.org/mstrobels/procyon/wiki/Home>

²<https://github.com/jacoco/jacoco>

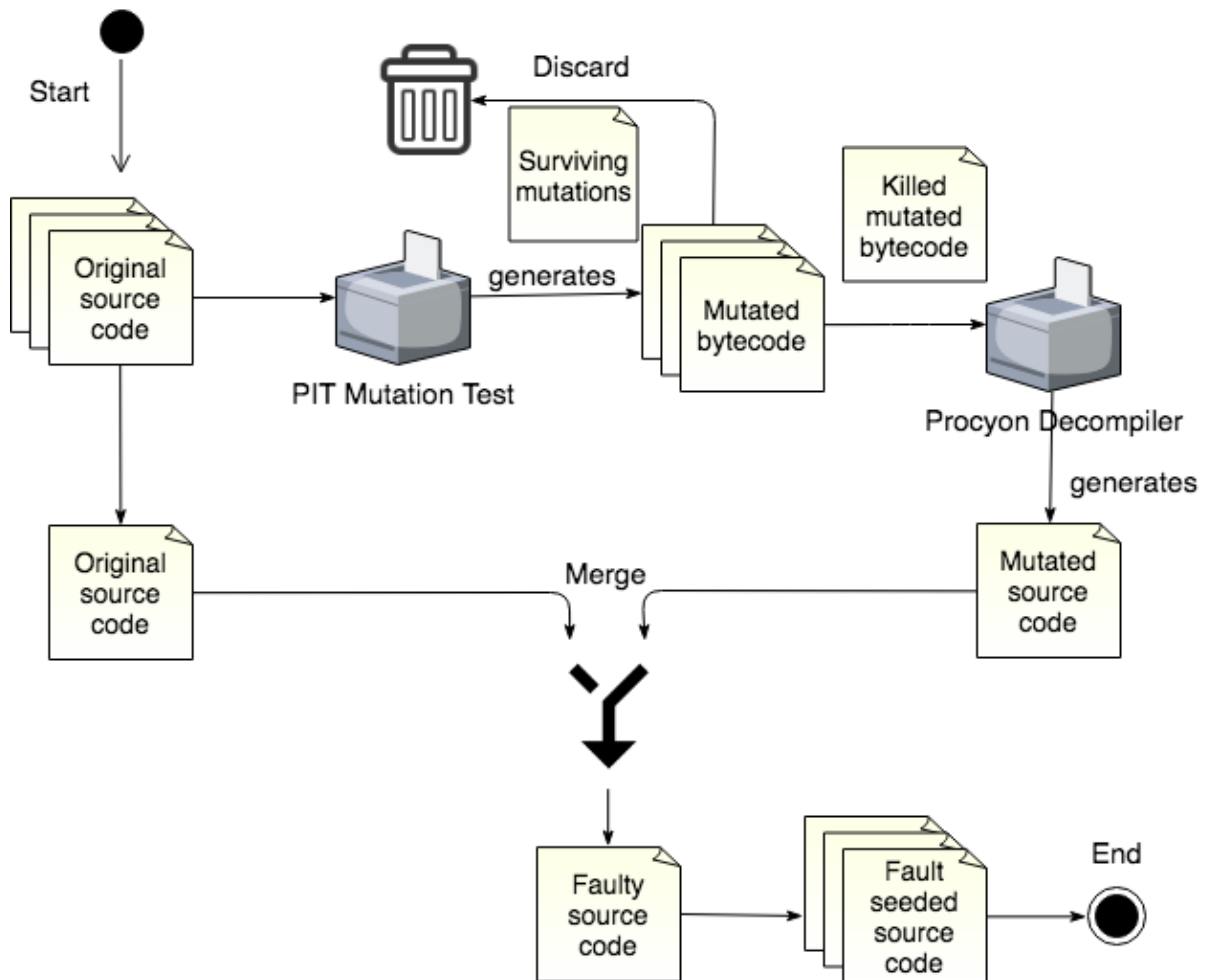


Figure 5.3: Fault seeding process in Optimus Framework.

Collected coverage is provided in three different source code granularities: statement (line), branch and method.

5.7 OPTIMUS HISTORICAL ANALYZER

This module is responsible for storing and providing historical information about test cases execution. To simplify the implementation and use of the framework, a self-contained relational database is used, in our case SQLite version 3.2.1. In this way, no additional tools need to be installed to use the framework, like a database server.

Considering history-based TCP techniques, historical information that is needed as input include test case result (failed or passed) and execution time. Moreover, it is important to be able to query how many times and when a test case was executed along the execution history. Thus, the relational model depicted in Figure 5.4 is used to provide and store all the necessary information in the database.

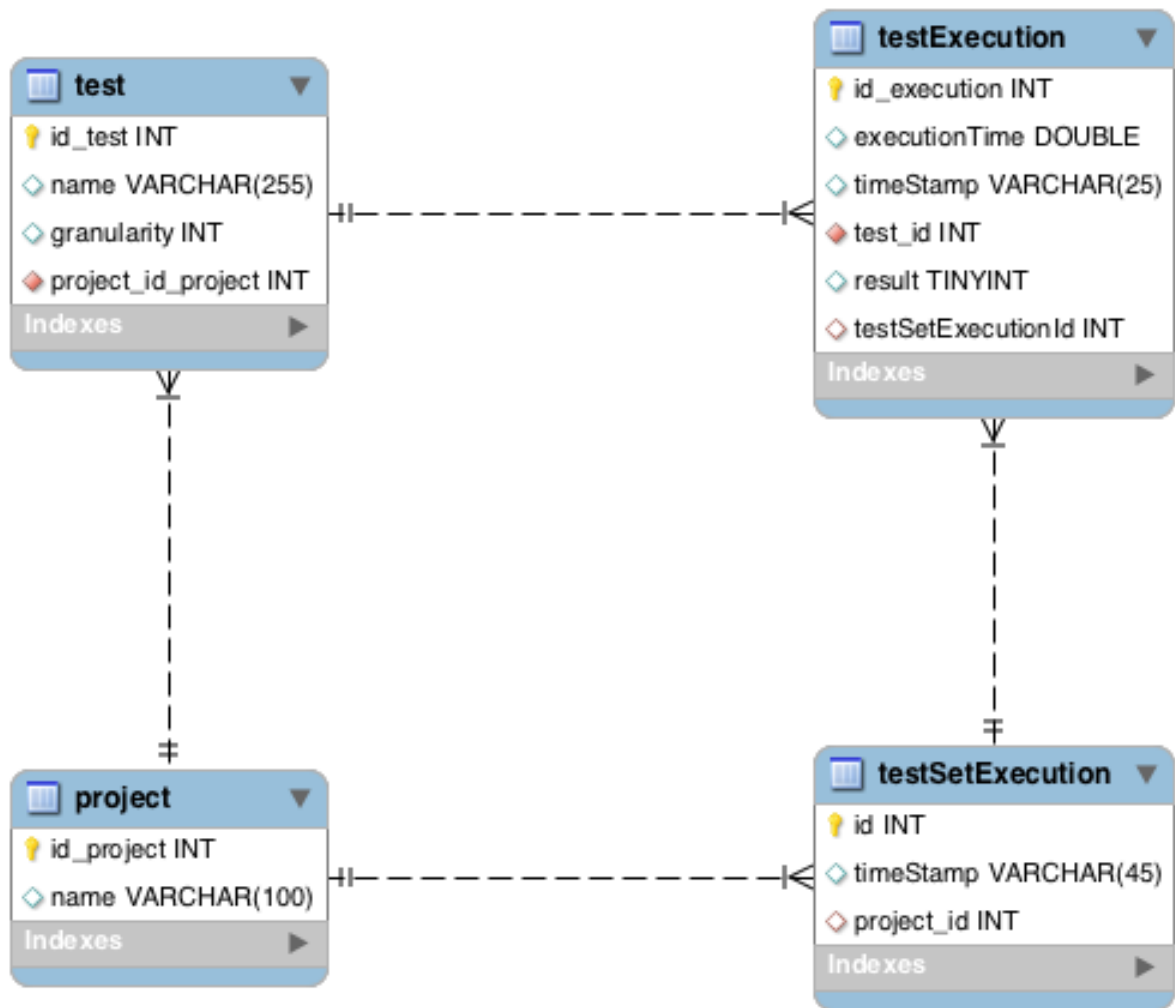


Figure 5.4: Historical analyzer relational database model.

To collect the data, a JUnit listener is used in the `optimus-test` module. The listener is called after the execution of each test case and thus, the result and details are recorded in the database.

5.8 OPTIMUS MODIFICATION ANALYZER

This module is responsible for maintaining and analyzing modifications between versions of a project. It must be able to provide information like how many lines of a method from a class were changed since the last build or which methods were changed.

To allow the collection of this type of information, if used, the module stores a full copy of the source code after building the project. This copy is used to compare with the current version of the project. The copy is updated after each build and the comparison is

done using Diff Utils³ library, which can provide the difference between two versions of a text, in our case, of every class of the project. We extract methods using the JavaParser⁴ library.

5.9 OPTIMUS EXECUTION TRACE ANALYZER

This module is responsible for providing execution trace information. Execution trace refers to how many times each source code element was executed by each test case.

Different granularities can be used for execution trace, like in coverage information. We provide this information at the line granularity.

To collect such information, we use the Cobertura tool. It is a coverage tool that provides information on how many times each line was executed. The problem is that it does not allow collecting this information for each test case, providing only a report in the end of the tests execution, containing aggregated data. For this reason, we created a JUnit listener in the `optimus-test` module that is used to intermediate the process of collection. After the execution of each test case, Cobertura data is retrieved and stored.

5.10 OPTIMUS FRAMEWORK USAGE

In order to use the framework, the user must download and locally install the project, which is available in a open source repository⁵ on Github.

After installing the framework, the user must setup his project Maven configuration file (`pom.xml`). It is necessary to add the framework dependency information, so that during the build of the project, Maven can look up for the framework binaries and include it in the user project classpath. The configuration is added to the `<build>` and `<plugin>` groups of the configuration file, like the example in Listing 5.1. This example shows the default information needed to execute the framework using the total coverage TCP technique, using the statement coverage granularity and the class-level test execution granularity, which is set by default. This simple configuration is enough for industrial use. A practitioner intending to use the framework, only has to choose a TCP technique and set it in the configuration. After that, all builds of his project will execute using the

³<https://github.com/wumpz/java-diff-utils>

⁴<https://github.com/javaparser/javaparser>

⁵<https://github.com/helenocampos/optimus>

order given by the chosen technique.

Listing 5.1: Example configuration for Optimus Framework.

```

<build>
  <plugins>
    <plugin>
      <groupId>io.github.helenocampos</groupId>
      <artifactId>optimus-framework</artifactId>
      <version>1.0.1</version>
      <executions>
        <execution>
          <goals>
            <goal>experiment</goal>
          </goals>
        </execution>
      </executions>
      <configuration>
        <prioritization>total statement coverage</
          prioritization>
      </configuration>
    </plugin>
  </plugins>
</build>

```

There are two types of execution of the framework: prioritization and experiment. Those are set in the `<goal>` tag.

Prioritization execution mode simply executes the test cases of the project using a single TCP technique.

Experiment execution mode can execute more than one TCP technique and compare them by calculating the effectiveness as measured by the APFD metric. Furthermore, report files are generated to allow the user to compare the techniques.

Within the experiment execution mode, there are three types of experiment that can be used. They are mutation, versions and local.

Mutation experiment mode uses the fault-injection-plugin in order to create a faulty version of the project being built and tested. After that, chosen TCP techniques are used to execute the test cases in the calculated order. The order in which the seeded faults are detected by the test cases is recorded, to allow the calculation of the APFD metric. In the end of the process, collected data is printed to a report file.

Local experiment mode assumes that faults already exist in the project being built and tested. Test cases are executed in the order calculated by the TCP techniques chosen by the user and a report file is generated in the end of the process, showing the APFD that each technique achieved during the execution.

Versions experiment is similar to the local one. However, it also assumes that there are different versions of the project, each one in a separated folder, which should be experimented with the TCP techniques chosen by the user. In this way, the reports generated in the end of the process are aggregated to allow the comparison of the TCP techniques across different versions of the project.

Different configurations can be included inside the `<configuration>` tag. A list of allowed configurations is displayed in Table 5.1.

To illustrate the usage of the framework, an example configuration is displayed in Figure 5.5. In this configuration, the framework will execute in the experiment mutation mode, executing the TCP techniques default (no prioritization), random and total statement coverage, at the method test level. The random technique will execute 5 times and executions will be simulated.

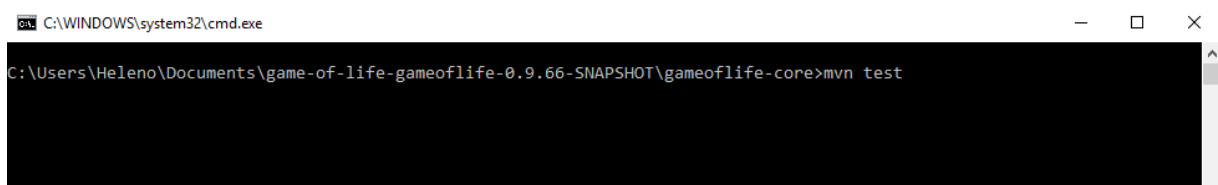
```

<plugin>
  <groupId>io.github.helenocampos</groupId>
  <artifactId>optimus-framework</artifactId>
  <version>1.0.1</version>
  <executions>
    <execution>
      <goals>
        <goal>experiment</goal>
      </goals>
    </execution>
  </executions>
  <configuration>
    <prioritizationTechniques>
      <technique>default</technique>
      <technique>random</technique>
      <technique>total statement coverage</technique>
    </prioritizationTechniques>
    <granularity>method</granularity>
    <experimentType>mutation</experimentType>
    <randomRepeat>5</randomRepeat>
    <simulateExecution>true</simulateExecution>
    <experimentOutputDirectory>C:\Users\Heleno\Documents\Experiments</experimentOutputDirectory>
  </configuration>
</plugin>

```

Figure 5.5: Example configuration of Optimus Framework.

A project with this configuration is run with the “mvn test” command, as displayed in Figure 5.6.



```

C:\WINDOWS\system32\cmd.exe
C:\Users\Heleno\Documents\game-of-life-gameoflife-0.9.66-SNAPSHOT\gameoflife-core>mvn test

```

Figure 5.6: Example run of Optimus Framework.

The framework executes, displaying some basic information about which technique

Table 5.1: Allowed configurations for Optimus Framework.

Tag	Description	Values
<prioritization>	Defines a single TCP technique to be used.	TCP techniques name. E.g. total statement coverage.
<prioritizationTechniques>	Defines a list of TCP techniques to be used. Each element of the list must be between <technique>tag.	E.g. <technique>total statement coverage</technique> <technique>random</technique>
<reports>	Defines a list of reports to be generated. Each element of the list must be between <report>tag.	Available options are “summary” and “raw” reports. If none is defined, by default both are generated
<granularity>	Defines the granularity in which the test cases will be executed.	Available options are “method” or “class”.
<dbPath>	Defines the path where statistics will be stored. These statistics are used by history-based TCP techniques.	A path to a .db file (the file is created automatically at the specified path.) E.g. C:\myproject\mydb.db
<experimentOutputDirectory>	Defines the path to save mutated copies of the project being tested.	Path to a directory. E.g. C:\experiments
<experimentType>	Specifies the type of the experiment to be run.	Allowed values are: “mutation”, “versions” or “local”.
<printLogs>	Enables the printing of logs during the execution of the framework.	False (default) or true.
<clustersAmount>	Specifies the amount of clusters to be used with GOS TCP technique.	A numerical value. E.g. 2.
<backupSourceCode>	Enables the backup of the source code of the project between versions. If enabled, the src folder of the current version will be stored to be used in the next version. This is used by Modification-based techniques.	False (default) or true.
<backupPath>	Specifies the path where the backup of the source code is to be stored.	Path to a directory. E.g. C:\myproject\bkp
<simulateExecution>	Enables the simulation of the execution of TCP techniques.	False (default) or true.
<versionsFolder>	Specifies the path to the root folder where versions of the project can be found. This is used exclusively by the versions experiment execution mode.	Path to a directory. E.g. C:\myproject\versions
<executionTimes>	Specifies how many times the experiment is going to run. This is used by the mutation experiment execution mode. Every run generates a new copy of the project, with (potentially) new mutations.	A numerical value. Default value is 1.
<randomRepeat>	Specifies how many times the TCP technique “random” is going to run.	A numerical value. Default value is 1.

were executed and when, as displayed in Figure 5.7.

In the end of the process, reports are generated. Figure 5.8 shows the summary report, containing aggregated information and the comparison between the techniques executed in the experiment. This report is generated using the Portable Document Format (PDF) format.

Figure 5.9 shows the raw report, containing all collected information during the techniques execution. This report is generated using the `xlsx` format, which can be opened using the Microsoft Excel program.

This report also includes sheets containing the test cases execution order for each executed technique, as displayed in Figure 5.10, for the default (no prioritization) technique in this example.

5.11 FINAL CONSIDERATIONS

In this chapter, we discussed the implementation and usage of the proposed framework to support TCP techniques usage and experimentation. In the next chapter, the implemented framework will be evaluated through experiments with real software.

```

C:\WINDOWS\system32\cmd.exe
> MEMORY_ERROR 0 NOT_STARTED 0 STARTED 0 RUN_ERROR 0
> NO_COVERAGE 0
-----
[OPTIMUS]Injecting faults
-----
[OPTIMUS]Collecting coverage data
-----
[OPTIMUS]Executing tests with default prioritization technique. Started at: 15:04:47
-----
[OPTIMUS]Execution took 10.264 seconds.
-----
[OPTIMUS]Executing tests with default prioritization technique. Started at: 15:04:57
-----
[OPTIMUS]Execution took 15.25 seconds.
-----
[OPTIMUS]Executing tests with random prioritization technique. Started at: 15:05:12
-----
[OPTIMUS]Execution took 10.373 seconds.
-----
[OPTIMUS]Executing tests with random prioritization technique. Started at: 15:05:23
-----
[OPTIMUS]Execution took 9.744 seconds.
-----
[OPTIMUS]Executing tests with random prioritization technique. Started at: 15:05:33
-----
[OPTIMUS]Execution took 9.071 seconds.
-----
[OPTIMUS]Executing tests with random prioritization technique. Started at: 15:05:42
-----
[OPTIMUS]Execution took 8.761 seconds.
-----
[OPTIMUS]Executing tests with random prioritization technique. Started at: 15:05:50
-----
[OPTIMUS]Execution took 8.582 seconds.
-----
[OPTIMUS]Executing tests with total statement coverage prioritization technique. Started at: 15:05:59
-----
[OPTIMUS]Execution took 10.145 seconds.
-----
[OPTIMUS]Generating reports
-----

```

Figure 5.7: Information of example run of Optimus Framework.

Prioritization Experiment Report Summary

Generated at: 15:06:10 07/27/2018

Experiment context

Amount of test cases in the experimented software: 49

Granularity of the test cases in the experimented software: method

Project: gameoflife-core						
Technique	Executions	Min APFD	Mean APFD	Median APFD	Max APFD	Standard Deviation
total statement coverage	1	0.553	0.553	0.553	0.553	0
random	5	0.492	0.51	0.511	0.536	0.018
default	1	0.481	0.481	0.481	0.481	0

Figure 5.8: Generated summary report for example run of Optimus Framework.

Report for gameoflife-core build. Generated at: 15:06:12 07/27/2018

Project	Prioritization technique	APFD	Failures	Executed tests	Test granularity	Execution time (seconds)	Timestamp	Execution order
gameoflife	default	0.48054105	43	49	method	0.252	15:05:03 07/27/2018	Order
gameoflife	random	0.49193165	43	49	method	0.318	15:05:19 07/27/2018	Order
gameoflife	random	0.49335548	43	49	method	0.287	15:05:28 07/27/2018	Order
gameoflife	random	0.53607024	43	49	method	0.262	15:05:38 07/27/2018	Order
gameoflife	random	0.51566207	43	49	method	0.288	15:05:47 07/27/2018	Order
gameoflife	random	0.51091595	43	49	method	0.28	15:05:55 07/27/2018	Order
gameoflife	total statement coverage	0.55268153	43	49	method	0.495	15:06:05 07/27/2018	Order

Figure 5.9: Generated raw report for example run of Optimus Framework.

Test logs for default prioritization technique

[Back to Execution data](#)

Order	Test name	Test passed?	Execution time (seconds)
1	com.wakaleo.gameoflife.domain.WhenYouCreateACell.aDeadCellShouldBePrintedAsADot	TRUE	0,007
2	com.wakaleo.gameoflife.domain.WhenYouCreateACell.aDeadCellShouldBeRepresentedByADot	TRUE	0
3	com.wakaleo.gameoflife.domain.WhenYouCreateACell.aDeadCellSymbolShouldBeADot	FALSE	0,005
4	com.wakaleo.gameoflife.domain.WhenYouCreateACell.aLiveCellShouldBePrintedAsAnAsterisk	TRUE	0
5	com.wakaleo.gameoflife.domain.WhenYouCreateACell.aLiveCellShouldBeRepresentedByAnAsterisk	TRUE	0
6	com.wakaleo.gameoflife.domain.WhenYouCreateACell.aLiveCellSymbolShouldBeAnAsterisk	FALSE	0,001
7	com.wakaleo.gameoflife.domain.WhenYouCreateAGrid.ModifyingTheGridContentsAsAnArrayShouldNotModifyTheOriginalContents	FALSE	0
8	com.wakaleo.gameoflife.domain.WhenYouCreateAGrid.aNewGridShouldBeEmpty	FALSE	0,001

Figure 5.10: Execution order in raw report for example run of Optimus Framework.

6 EVALUATION

6.1 INTRODUCTION

In this chapter, the proposed framework is evaluated through an empirical experiment, according to Wohlin et al. (2012) guidelines. The main goal of this evaluation is to demonstrate that the framework can be used to support the experimentation of TCP techniques, generating evidence about their effectiveness on real software.

6.2 EXPERIMENTAL STUDY

In order to achieve the goal of the evaluation, an experimental study comparing TCP techniques on open source software, using the proposed framework is performed.

The goal of the experiment is defined according to the Goal Question Metric (GQM) template, proposed by Caldiera and Rombach (1994). The GQM is used to guide the experiment definition, by first defining a goal, then deriving research questions that should be answered to achieve the goal. To answer the questions, metrics are used. In this way, the goal is to **analyze** open source software test suite executions **for the purpose of** evaluation, **with respect to** the rate of faults detection using TCP techniques with Optimus Framework, **from the viewpoint** of researchers.

Based on the experiment goal, the research questions for this experiment are derived as follows.

RQ1: does the use of Optimus Framework implemented TCP techniques improve the rate of faults detection on open source Java projects?

This question aims to investigate the impact in the build process if the original developers of the experimented projects had used a TCP technique.

RQ2: can test suite execution granularity impact the effectiveness of Optimus Framework implemented techniques on open source Java projects?

This question is motivated by Do et al. (2006) study, which experimentally evaluated the same question on 4 Java projects and also by the findings of our systematic literature review, which suggests that TCP techniques executed at the method test granularity achieve higher effectiveness than those executed at the class granularity. Based on this,

we want to find if this is also valid for different open source Java projects.

A series of experimental activities are needed to answer these research questions. These activities are not trivial to be executed manually, requiring the execution of test cases from software projects using different configurations. In this way, we believe that if we find answers for these research questions, we have evidence to answer the third question of this work, which aim to demonstrate how our proposed framework support practitioners and researchers.

6.2.1 OBJECTS OF ANALYSIS

Considering that we are interested on investigating open source Java projects and the use of the proposed framework by researchers and practitioners, we used the following approach to choose representative projects to be used in the experiment.

- We used Github website, since it is the most famous open source platform.
- Since we are interested in Java language projects, we used the website search tool to query for projects written in Java.
- The list of projects returned by the query was ordered by amount of stars, showing the most famous projects in the platform.
- Based on this list, we searched for projects that meet the following requirements.
 - Is not an example/tutorial project.
 - Written in Java language.
 - Use Maven dependency manager (identified by projects that have a file named pom.xml).
 - Have JUnit test cases.
 - Has a public continuous integration environment setup on TravisCI.
 - Has at least one build that failed due to test failures.
 - Does not contain Maven sub modules, since sub modules are required to be compiled in a specific order and thus it would not be possible to reschedule test execution order from sub modules within a project.

Following the retrieved list of projects from Github and applying above requirements, CoreNLP and Jackson-databind projects were selected. Within the selected projects, we selected builds in the build history that were not completed due to test failures. The build is also required to be reproducible on a local machine without any further configuration, like download of external libraries, models or older versions of the Java Development Kit (JDK). Successive builds that had the same failures were not selected since they would have the same result.

CoreNLP is a project from Stanford that provides a set of natural language analysis tools written in Java language, according to their homepage¹. By the moment that this experiment was performed, the public continuous integration (CI) server² of the project contained a total of 2803 builds, while the Github repository sums a total of 14673 commits. For this experiment, the OSSFaultsFinder found 11 faulty builds on the build history repository. Details about each build are described in Table 6.1.

Jackson-databind is a sub project from the Jackson project, which is a fast and compliant streaming JSON parser and writer, according to their homepage³. Jackson-databind contains general-purpose data-binding functionality for the project. When this experiment was performed, its CI server⁴ contained about 2869 builds, and its Github repository⁵, a total of 4918 commits. OSSFaultsFinder found 54 faulty builds in the build history, however, when those builds were downloaded and reproduced on a local machine, only 15 failed. From those 15 faulty builds, it was observed that 4 of them were identical to another build and thus they were removed from the experiment, in order to prevent duplicated data. A total of 11 builds of this project were used in this experiment. Its details are described in Table 6.1.

6.2.2 VARIABLES

Independent variables for this experiment are prioritization techniques implemented in the Optimus Framework, whose treatment values are listed in Table 6.2 and described in Section 4.6; and test suite granularity, whose treatment values are class level and method level.

¹<https://github.com/stanfordnlp/CoreNLP>

²<https://travis-ci.org/stanfordnlp/CoreNLP/builds>

³<http://fasterxml.com/projects.html>

⁴<https://travis-ci.org/FasterXML/jackson-databind/builds>

⁵<https://github.com/FasterXML/jackson-databind>

Table 6.1: Experiment projects information.

Project	Build number	Failures amount	Source code under test		JUnit test suite	
			# Classes	KLOC	#Test classes	#Test methods
CoreNLP	#55	58	1907	729.7	164	825
	#60	18	1907	729.7	164	825
	#65	17	1907	729.8	164	825
	#89	18	1938	734.4	165	830
	#110	19	1938	725.9	165	830
	#218	42	1949	743.2	166	840
	#277	17	1957	737.9	165	827
	#430	57	1968	742	166	836
	#432	21	1968	741.9	166	836
	#449	58	1968	742	166	836
	#450	21	1968	742	166	836
Jackson-databind	#1568	3	769	154.8	16	1620
	#1650	3	771	155.7	17	1641
	#1983	3	786	157.9	18	1698
	#2142	3	869	172.3	18	1979
	#2402	2	808	160.8	18	1754
	#2717	3	934	174.2	17	2140
	#2718	3	934	174.3	17	2141
	#2720	3	934	174.3	17	2141
	#2724	3	934	174.3	17	2142
	#2745	5	911	178.9	19	2109
	#2824	3	919	180.1	19	2130

Table 6.2: TCP techniques used in the experiment.

Acronym	Technique name
T0	Traditional (no prioritization)
T1	Random
T2	Theoretical optimal
T3	Total statement coverage
T4	Total branch coverage
T5	Total method coverage
T6	Additional statement coverage
T7	Additional branch coverage
T8	Additional method coverage
T9	Total diff method coverage
T10	Additional diff method coverage
T11	Total diff class coverage
T12	Additional diff class coverage
T13	Maxmin branch ART
T14	Maxmin statement ART
T15	Maxmin method ART
T16	Most failures first
T17	Recent failures first

TCP techniques for this experiment were chosen based on their viability of execution for selected projects. For instance, implemented similarity-based techniques were left out of the experiment because they are too costly to execute on big projects. We tried to use the Farthest-first Ordered Sequence technique on one build of CoreNLP, for example, and after 12 hours, the technique was still calculating the final test cases order. According to Wu et al. (2012), who proposed the ordered sequence techniques, it can be fairly expensive to employ and may not reduce the overall regression testing cost. ART techniques are also costly. According to Jiang et al. (2009), the time complexity of the algorithm is $O(m^3n)$ on the worst case, where m is the amount of test cases and n is the total amount of source code elements of the project. For this reason, despite the availability of 18 variations of ART in the proposed framework, we choose only 6 (T13, T14 and T15 at the method and class test granularity) of them to be included in the experiment. This choice was based on results found by Jiang et al. (2009) in an empirical experiment.

The dependent variable is the rate of fault detection achieved by the execution of the test cases using each of the treatment values on each of the object of analysis (a build from the selected open source projects).

The rate of fault detection is measured by the Average Percentage of Faults Detected (APFD) value. This is the most common metric used to measure effectiveness of TCP techniques. Its value is calculated according to Equation 2.1.

6.2.3 EXPERIMENT SETUP

Faulty builds of the selected projects were found and downloaded using the OSSFaults-Finder module of the Optimus Framework.

With all the faulty versions of the selected project downloaded, the framework was executed in the versions experiment mode, iterating over each version of the projects and executing the selected TCP techniques for each configuration.

Selected TCP techniques for this experiment are listed in Table 6.2. The traditional "technique" (T0), named "default" in the framework, represents the natural execution order of test cases, as done by default when executing the test suite. The random TCP technique (T1) was executed 100 times to generate a mean APFD for each build, since each execution produces a different test case order and thus a different APFD value. Each configuration execution in Optimus Framework generates a report file, like the one

Report for jackson-databind build. Generated at: 03:56:42 08/19/2018								
Project	Prioritization technique	APFD	Failures	Executed tests	Test granularity	Execution time (seconds)	Timestamp	Execution order
jackson-databind1568	default	0.5649176	3	1620	class	7.303	20:40:52 08/	Order
jackson-databind1568	optimal	0.9976337	3	1620	class	3.618	20:40:56 08/	Order
jackson-databind1568	total statement coverage	0.5332304	3	1620	class	3.311	20:41:00 08/	Order
jackson-databind1568	total branch coverage	0.5332304	3	1620	class	3.81	20:41:04 08/	Order
jackson-databind1568	total method coverage	0.5332304	3	1620	class	3.296	20:41:08 08/	Order
jackson-databind1568	additional statement coverage	0.5284979	3	1620	class	4.93	20:41:13 08/	Order
jackson-databind1568	additional branch coverage	0.4593621	3	1620	class	4.463	20:41:18 08/	Order
jackson-databind1568	additional method coverage	0.4075102	3	1620	class	4.195	20:41:23 08/	Order
jackson-databind1568	total diff method coverage	0.5332304	3	1620	class	3.36	20:41:27 08/	Order
jackson-databind1568	additional diff method coverage	0.4554526	3	1620	class	3.088	20:41:31 08/	Order
jackson-databind1568	total diff class coverage	0.5332304	3	1620	class	2.64	20:41:34 08/	Order
jackson-databind1568	additional diff class coverage	0.3346707	3	1620	class	3.155	20:41:38 08/	Order
jackson-databind1568	maxmin branch ART	0.5618312	3	1620	class	296.565	20:46:35 08/	Order
jackson-databind1568	maxmin statement ART	0.5562757	3	1620	class	321.043	20:51:57 08/	Order
jackson-databind1568	maxmin method ART	0.5585390	3	1620	class	291.31	20:56:49 08/	Order
jackson-databind1568	most failures first	0.5649176	3	1620	class	2.596	20:56:52 08/	Order
jackson-databind1568	recent failures first	0.5649176	3	1620	class	2.322	20:56:56 08/	Order
jackson-databind1568	random	0.4208847	3	1620	class	2.329	20:56:59 08/	Order

Figure 6.1: Example raw report generated from experiment run.

depicted in Figure 6.1, containing the APFD value and execution order for each executed TCP technique and project version.

The optimal technique (T2) is the theoretical optimal APFD that could be achieved by a TCP technique in a specific run, considering the amount of faults and test cases.

The framework configuration to run the experiment was similar to the one depicted in Figure 6.2. Four configuration files were needed to run the entire experiment. The differences between them were the test granularity and the versions folder for the subject project.

6.2.4 DATA AND ANALYSIS

Since there are two different projects as objects of analysis, they will be analyzed individually regarding the research questions. All analyses were performed using Minitab⁶ 17 statistical tool.

6.2.4.1 CoreNLP

To summarize collected data, box plots are depicted in Figure 6.3 for all executed TCP techniques for CoreNLP project. In techniques names, the letter C represents the tech-

⁶<http://www.minitab.com>

```

<plugin>
  <groupId>io.github.helenocampos</groupId>
  <artifactId>optimus-framework</artifactId>
  <version>1.0.1</version>
  <executions>
    <execution>
      <goals>
        <goal>experiment</goal>
      </goals>
    </execution>
  </executions>
  <configuration>
    <prioritizationTechniques>
      <technique>default</technique>
      <technique>optimal</technique>
      <technique>total statement coverage</technique>
      <technique>total branch coverage</technique>
      <technique>total method coverage</technique>
      <technique>additional statement coverage</technique>
      <technique>additional branch coverage</technique>
      <technique>additional method coverage</technique>
      <technique>total diff method coverage</technique>
      <technique>additional diff method coverage</technique>
      <technique>total diff class coverage</technique>
      <technique>additional diff class coverage</technique>
      <technique>maxmin branch ART</technique>
      <technique>maxmin statement ART</technique>
      <technique>maxmin method ART</technique>
      <technique>most failures first</technique>
      <technique>recent failures first</technique>
      <technique>random</technique>
    </prioritizationTechniques>
    <granularity>method</granularity>
    <experimentType>versions</experimentType>
    <printLogs>false</printLogs>
    <randomRepeat>100</randomRepeat>
    <backupSourceCode>true</backupSourceCode>
    <backupPath>C:\Users\Heleno\Documents\dissertation\coreNLP_method\bkp</backupPath>
    <dbPath>C:\Users\Heleno\Documents\dissertation\coreNLP_method\db</dbPath>
    <simulateExecution>true</simulateExecution>
    <versionsFolder>C:\Users\Heleno\Documents\dissertation\coreNLP_method</versionsFolder>
  </configuration>
</plugin>

```

Figure 6.2: Example of Optimus Framework configuration to run the experiment.

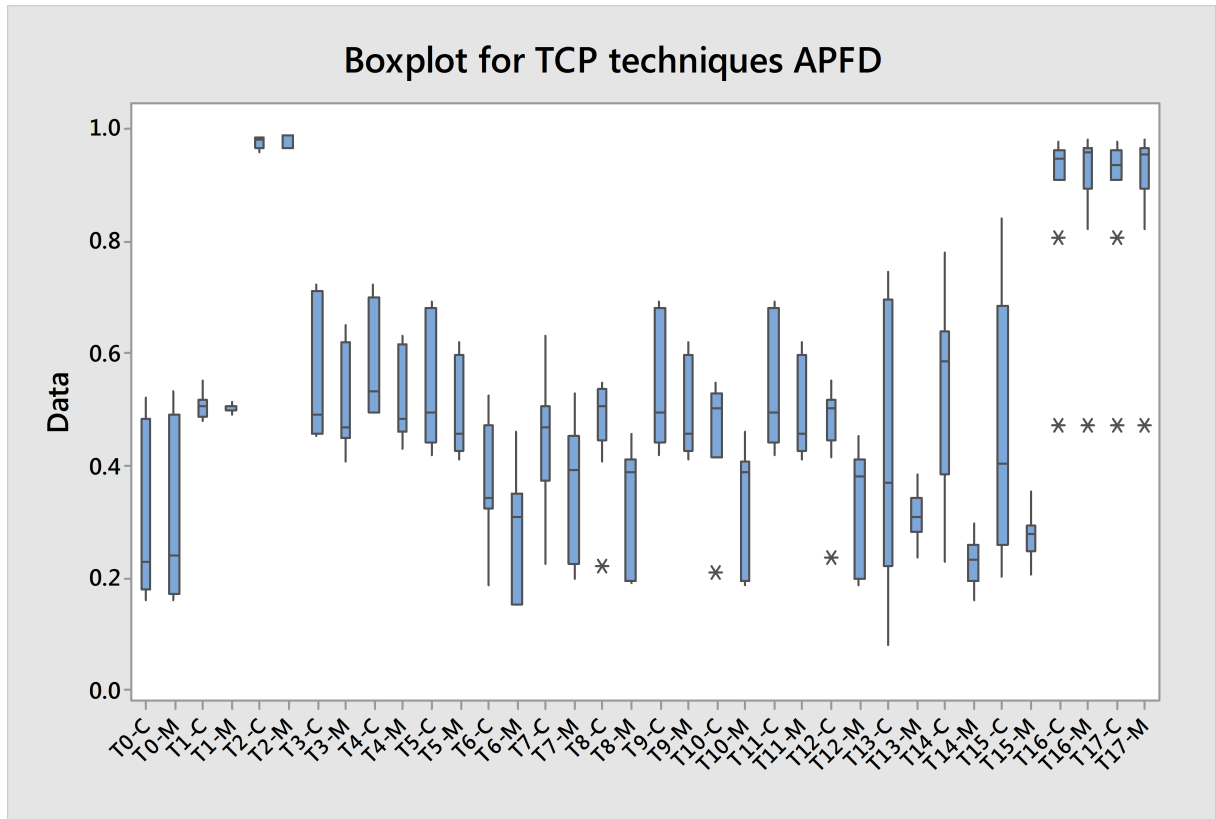


Figure 6.3: Boxplot of APFD values obtained for CoreNLP project.

nique executed at the class test level and M at the method test level.

Research question 1 is concerned with comparing the use of different TCP techniques against the non-use, which is the control technique T0-C in Table 6.2. Thus, the design of this analysis is a series of sixteen sub-experiments with one factor, which is the use or not of a TCP technique, and two treatments, whose values are APFD obtained using T0-C (fixed) and using the remaining techniques displayed in Table 6.2 (T1-C and T3-C through T17-C). Sub-experiment comparisons are listed in Table 6.3.

For this analysis it is used 11 builds of the CoreNLP software project with 15 different test case orderings from TCP techniques T3 through T17 and 100 orderings from T1 for each build. This results in a total of 1276 different APFD values (trials) collected for this research question.

All normality distribution checking of the samples were done with Shapiro-Wilk test, since for each sample analyzed we have 11 data points (builds) and that this test is recommended when samples have less than 30 data points. Furthermore, checking the normality of the fixed treatment (T0-C), a p-value of less than 0.010 is obtained. As it is less than the significance level of 0.05, the sample can be characterized as non-normal.

Table 6.3: Comparisons for RQ1 of CoreNLP experiment.

#	Comparison
1	T0-C x T1-C
2	T0-C x T3-C
3	T0-C x T4-C
4	T0-C x T5-C
5	T0-C x T6-C
6	T0-C x T7-C
7	T0-C x T8-C
8	T0-C x T9-C
9	T0-C x T10-C
10	T0-C x T11-C
11	T0-C x T12-C
12	T0-C x T13-C
13	T0-C x T14-C
14	T0-C x T15-C
15	T0-C x T16-C
16	T0-C x T17-C

Since it is not normal, it is necessary to use a non-parametric hypothesis test in all following analysis for this research question. The non-parametric test recommended for this type of design is Mann-Whitney test. The output of the normality test is depicted in Figure 6.4, which is produced by the statistical tool.

Sub-experiment #1 compares APFD values obtained when no TCP technique (T0-C) was used against when the random TCP technique (T1-C) was used. Hypotheses are defined as:

- H_0 : Medians obtained using T0-C are equal to medians obtained using T1-C.
- H_1 : Medians obtained using T0-C are significantly different to medians obtained using T1-C.

Hypothesis test output is displayed in Listing 6.1. Obtained p-value is 0.0031, which is less than the significance level of 0.05. This indicates the rejection of H_0 and suggests that the samples from the two groups are significantly different.

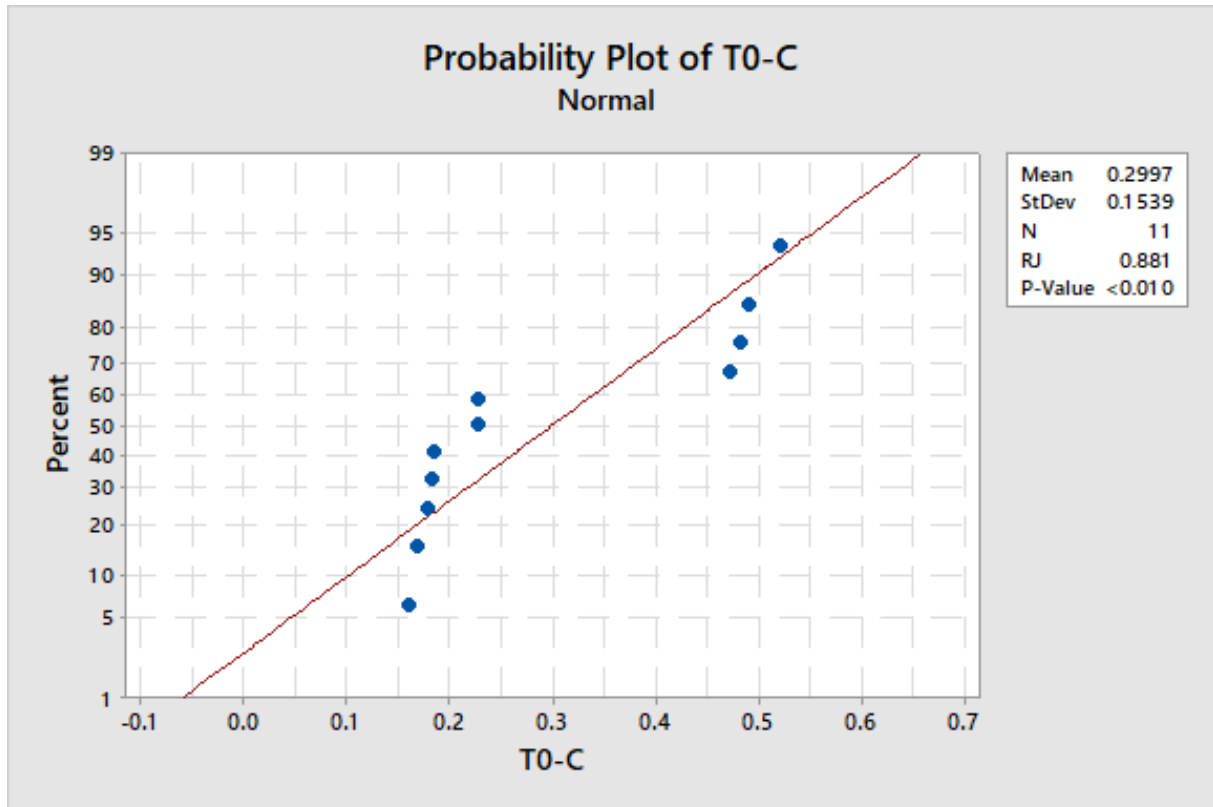


Figure 6.4: Normality test for T0 from CoreNLP.

Listing 6.1: Hypothesis test result for sub-experiment #1 of RQ1 from CoreNLP.

Mann–Whitney Test and CI: T0–C, T1–C

	N	Median
T0–C	11	0.2283
T1–C	11	0.5067

Point estimate for $\eta_1 - \eta_2$ is -0.2892

95.1 Percent CI for $\eta_1 - \eta_2$ is $(-0.3267, -0.0269)$

W = 81.0

Test of $\eta_1 = \eta_2$ vs $\eta_1 \neq \eta_2$ is significant at 0.0031

To prevent repetitive text for each sub-experiment, remaining sub-experiment analyses are summarized in Table 6.4. All sub-experiments used Mann-Whitney hypothesis test, since the fixed treatment distribution is not normal.

81% of the comparisons result in a TCP technique achieving statistically better effectiveness than the non-use of a TCP technique, as shown by highlighted rows in Table 6.4, for RQ1 analysis of CoreNLP project. The remaining 19% of the comparisons resulted in statistically equivalent effectiveness, despite the fact that the medians were higher for TCP techniques when compared to the non-use of a TCP technique.

Table 6.4: Hypotheses test results for RQ1 of CoreNLP.

#	Comparison	Hypothesis test result	APFD comparison result
1	T0-C x T1-C	T0-C \neq T1-C p-value (0.0031)	T0-C (0.2283) < T1-C (0.5067)
2	T0-C x T3-C	T0-C \neq T3-C p-value (0.0151)	T0-C (0.2283) < T3-C (0.4895)
3	T0-C x T4-C	T0-C \neq T4-C p-value (0.0003)	T0-C (0.2283) < T4-C (0.5319)
4	T0-C x T5-C	T0-C \neq T5-C p-value (0.0126)	T0-C (0.2283) < T5-C (0.4944)
5	T0-C x T6-C	T0-C = T6-C p-value (0.2122)	T0-C (0.2283) = T6-C (0.3425)
6	T0-C x T7-C	T0-C = T7-C p-value (0.1007)	T0-C (0.2283) = T7-C (0.4677)
7	T0-C x T8-C	T0-C \neq T8-C p-value (0.0104)	T0-C (0.2283) < T8-C (0.5058)
8	T0-C x T9-C	T0-C \neq T9-C p-value (0.0126)	T0-C (0.2283) < T9-C (0.4944)
9	T0-C x T10-C	T0-C \neq T10-C p-value (0.0215)	T0-C (0.2283) < T10-C (0.5011)
10	T0-C x T11-C	T0-C \neq T11-C p-value (0.0126)	T0-C (0.2283) < T11-C (0.4944)
11	T0-C x T12-C	T0-C \neq T12-C p-value (0.0058)	T0-C (0.2283) < T12-C (0.5021)
12	T0-C x T13-C	T0-C = T13-C p-value (0.2122)	T0-C (0.2283) = T13-C (0.3692)
13	T0-C x T14-C	T0-C \neq T14-C p-value (0.0031)	T0-C (0.2283) < T14-C (0.5845)
14	T0-C x T15-C	T0-C \neq T15-C p-value (0.0256)	T0-C (0.2283) < T15-C (0.4023)
15	T0-C x T16-C	T0-C \neq T16-C p-value (0.0002)	T0-C (0.2283) < T16-C (0.9465)
16	T0-C x T17-C	T0-C \neq T17-C p-value (0.0002)	T0-C (0.2283) < T17-C (0.9362)

Research question 2 is concerned with comparing TCP techniques effectiveness when executed at the method and class test level. Thus, the design of this analysis is a series of sixteen sub-experiments with one factor, which is the granularity of the tests, and two treatments, whose values are method and class level.

In this analysis, the same project builds are subjected to the same intervention (TCP technique) for each of the treatments. For this reason, we say that the analysis is paired.

For this analysis it is used 11 builds of the CoreNLP software with 15 different test case orderings from TCP techniques T3 through T17 and 100 orderings from T1 for each build. Furthermore, for each ordering we have two different test granularities being used. This results in a total of 2530 different APFD values (trials) collected for this research question. Sub-experiments are listed in Table 6.5.

As in RQ1 analysis, the first sub-experiment data analysis is described in details and the remaining are summarized.

Sub-experiment #1 compares T1 executed at the class level test granularity (T1-C) and at the method level granularity (T1-M). Using Shapiro-Wilk to test normality for each sample, results in a p-value > 0.100 for T1-C, as depicted in Figure 6.5 and for T1-M, as depicted in Figure 6.6. As both p-values are greater than the established significance level of 0.05, it suggests that the samples are normally distributed.

Homoscedasticity of the samples are checked, since they both have normal distribu-

Table 6.5: Comparisons for RQ2 of CoreNLP experiment.

#	Control x intervention
1	T1-C x T1-M
2	T3-C x T3-M
3	T4-C x T4-M
4	T5-C x T5-M
5	T6-C x T6-M
6	T7-C x T7-M
7	T8-C x T8-M
8	T9-C x T9-M
9	T10-C x T10-M
10	T11-C x T11-M
11	T12-C x T12-M
12	T13-C x T13-M
13	T14-C x T14-M
14	T15-C x T15-M
15	T16-C x T16-M
16	T17-C x T17-M

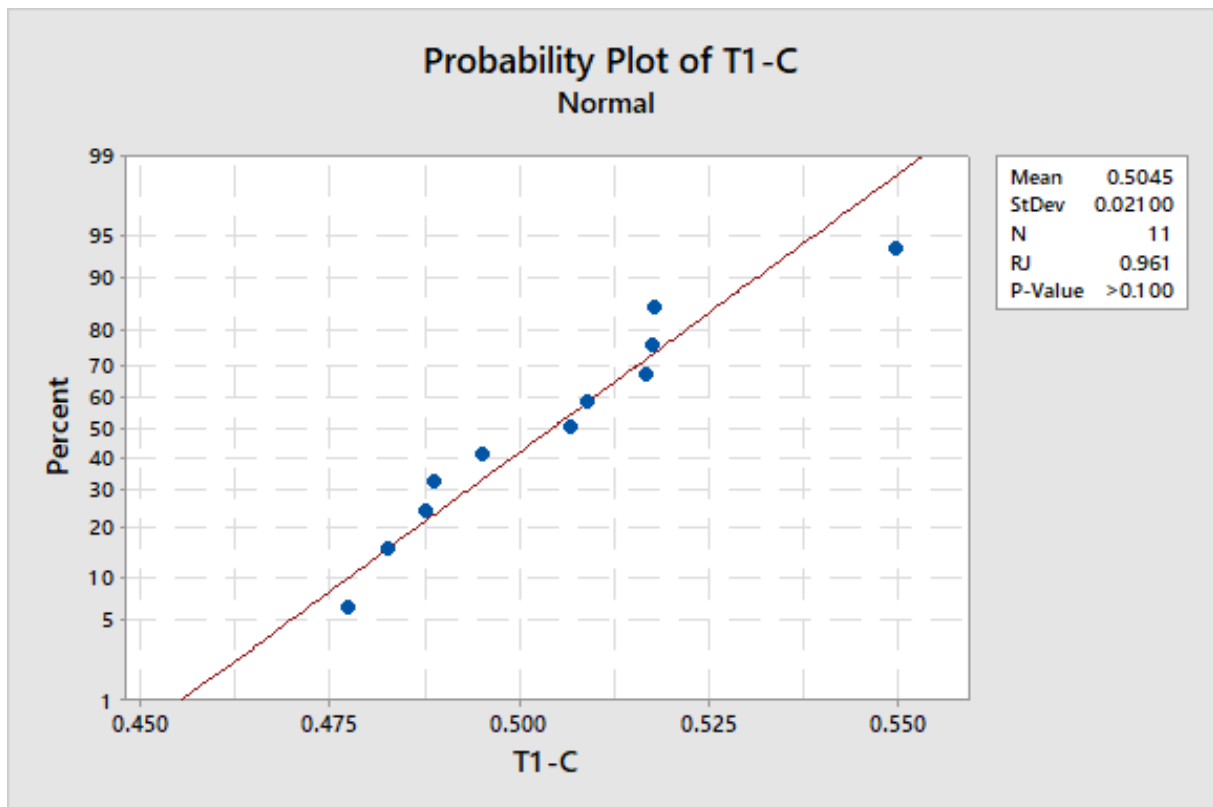


Figure 6.5: Normality test for T1-C from CoreNLP.

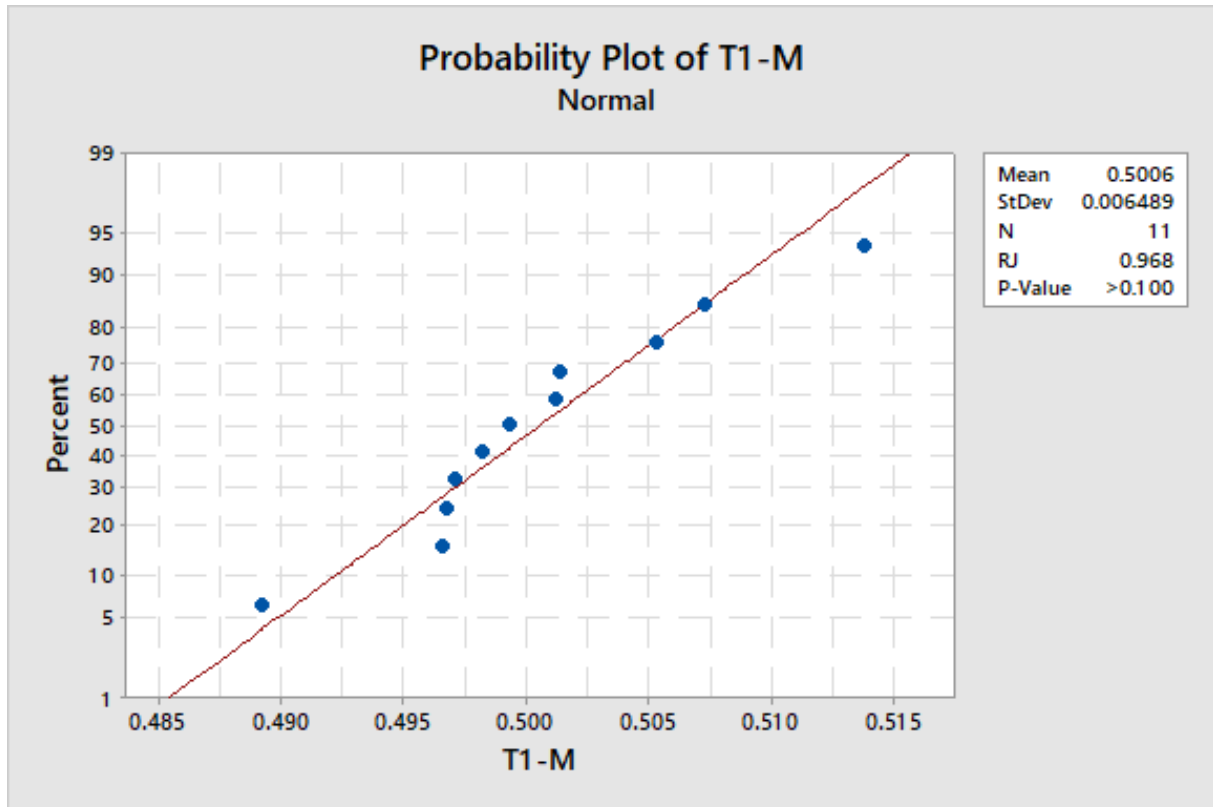


Figure 6.6: Normality test for T1-M from CoreNLP.

tions, using Levene's test, depicted in Figure 6.7. A p-value of 0.008 is obtained, which is less than the significance level of 0.05, indicating that the samples are not homoscedastic.

As the samples are normal but not homoscedastic, a non-parametric hypothesis test must be used. Since we have a paired design of 1 factor and 2 treatments, Wilcoxon signed-rank test is used, which is the non-parametric alternative for Paired T-Test. Hypotheses for this analysis are defined as:

- H_0 : APFD median obtained using T1-C is equal to the median obtained using T1-M.
- H_1 : APFD median obtained using T1-C is significantly different from the median obtained using T1-M.

The hypothesis test output from Minitab is displayed in Listing 6.2. The p-value obtained is 0.563, which is bigger than the significance level of 0.05. This result indicates that the null hypothesis H_0 cannot be rejected and thus suggests that the medians obtained using T1-C and T1-M are statistically equivalent.

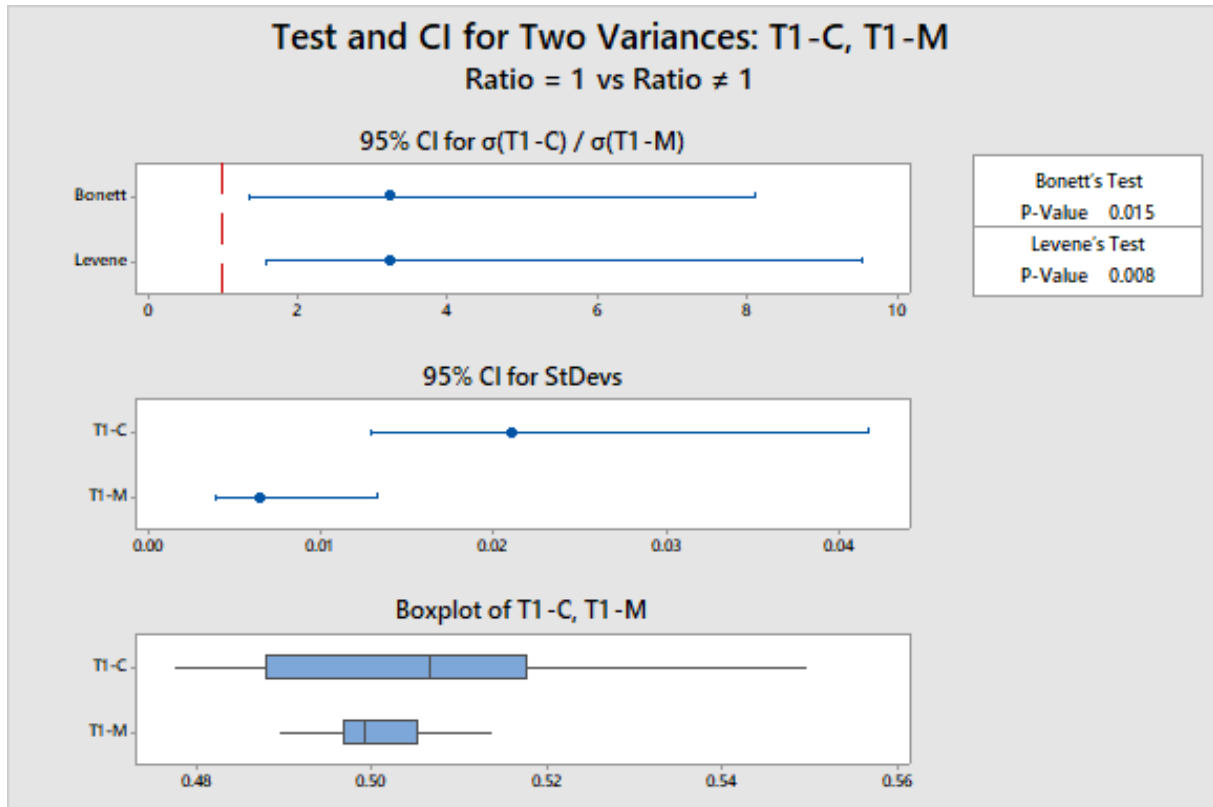


Figure 6.7: Homoscedasticity test for T1-C and T1-M from CoreNLP.

Listing 6.2: Hypothesis test result for sub-experiment #1 of RQ2 from CoreNLP.

Wilcoxon Signed Rank Test: (T1-C) - (T1-M)

Test of median = 0.000000 versus median \neq 0.000000

	N	N for Test	Wilcoxon Statistic	P	Estimated Median
(T1-C) - (T1-M)	11	11	40.0	0.563	0.002923

To prevent repetitive text for each sub-experiment, analyses of normality, homoscedasticity and hypothesis test used for each sub-experiment is listed in Table 6.6. Hypothesis test results are displayed in Table 6.7.

According to Table 6.7, no comparison resulted in better effectiveness for a TCP technique executed at the method granularity when compared to it executed at the class granularity. 38% of the comparisons resulted in statistically equivalent effectiveness for both granularities and the remaining 62% performed better at the class than at the method granularity.

Table 6.6: Normality and homoscedasticity analyses for RQ2 of CoreNLP.

#	Comparison	Normality	Homoscedasticity	Hypothesis test
1	T1-CxT1-M	✓	Not homoscedastic (p-value = 0.008)	Wilcoxon Signed Rank
2	T3-CxT3-M	T3-C not normal (p-value <0.01)	-	Wilcoxon Signed Rank
3	T4-CxT4-M	T4-C not normal (p-value = 0.028)	-	Wilcoxon Signed Rank
4	T5-C x T5-M	T5-M not normal (p-value = 0.035)	-	Wilcoxon Signed Rank
5	T6-C x T6-M	✓	✓	Paired T-Test
6	T7-C x T7-M	✓	✓	Paired T-Test
7	T8-C x T8-M	T8-C not normal (p-value <0.01)	-	Wilcoxon Signed Rank
8	T9-C x T9-M	T9-M not normal (p-value = 0.035)	-	Wilcoxon Signed Rank
9	T10-C x T10-M	T10-C not normal (p-value <0.01)	-	Wilcoxon Signed Rank
10	T11-C x T11-M	T11-M not normal (p-value = 0.035)	-	Wilcoxon Signed Rank
11	T12-C x T12-M	T12-C not normal (p-value <0.01)	-	Wilcoxon Signed Rank
12	T13-C x T13-M	✓	Not homoscedastic (p-value = 0.002)	Wilcoxon Signed Rank
13	T14-C x T14-M	✓	Not homoscedastic (p-value = 0.031)	Wilcoxon Signed Rank
14	T15-C x T15-M	✓	Not homoscedastic (p-value = 0.000)	Wilcoxon Signed Rank
15	T16-C x T16-M	T16-C not normal (p-value <0.01)	-	Wilcoxon Signed Rank
16	T17-C x T17-M	T17-C not normal (p-value <0.01)	-	Wilcoxon Signed Rank

Table 6.7: Hypothesis tests results for RQ2 of CoreNLP.

#	Comparison	Hypothesis test result	APFD comparison result
1	T1-CxT1-M	T1-C = T1-M p-value (0.563)	(0.50669) T1-C = T1-M (0.49933)
2	T3-CxT3-M	T3-C \neq T3-M p-value (0.029)	(0.4895) T3-C >T3-M (0.466)
3	T4-CxT4-M	T4-C \neq T4-M p-value (0.009)	(0.5319) T4-C >T4-M (0.4814)
4	T5-C x T5-M	T5-C \neq T5-M p-value (0.018)	(0.4944) T5-C >T5-M (0.4556)
5	T6-C x T6-M	T6-C = T6-M p-value (0.064)	(0,366) T6-C = T6-M (0,2884)
6	T7-C x T7-M	T7-C \neq T7-M p-value (0.000)	(0,4508) T7-C >T7-M (0,361)
7	T8-C x T8-M	T8-C \neq T8-M p-value (0.004)	(0.5058) T8-C >T8-M (0.3882)
8	T9-C x T9-M	T9-C \neq T9-M p-value (0.018)	(0.4944) T9-C >T9-M (0.4556)
9	T10-C x T10-M	T10-C \neq T10-M p-value (0.004)	(0.5011) T10-C >T10-M (0.3879)
10	T11-C x T11-M	T11-C \neq T11-M p-value (0.018)	(0.4944) T11-C >T11-M (0.4556)
11	T12-C x T12-M	T12-C \neq T12-M p-value (0.004)	(0.5021) T12-C >T12-M (0.3813)
12	T13-C x T13-M	T13-C = T13-M p-value (0.398)	(0.3692) T13-C = T13-M (0.3095)
13	T14-C x T14-M	T14-C \neq T14-M p-value (0.005)	(0.5845) T14-C >T14-M (0.2314)
14	T15-C x T15-M	T15-C = T15-M p-value (0.083)	(0.4023) T15-C = T15-M (0.2785)
15	T16-C x T16-M	T16-C = T16-M p-value (0.308)	(0.9465) T16-C = T16-M (0.9564)
16	T17-C x T17-M	T17-C = T17-M p-value (0.308)	(0.9362) T17-C = T17-M (0.9539)

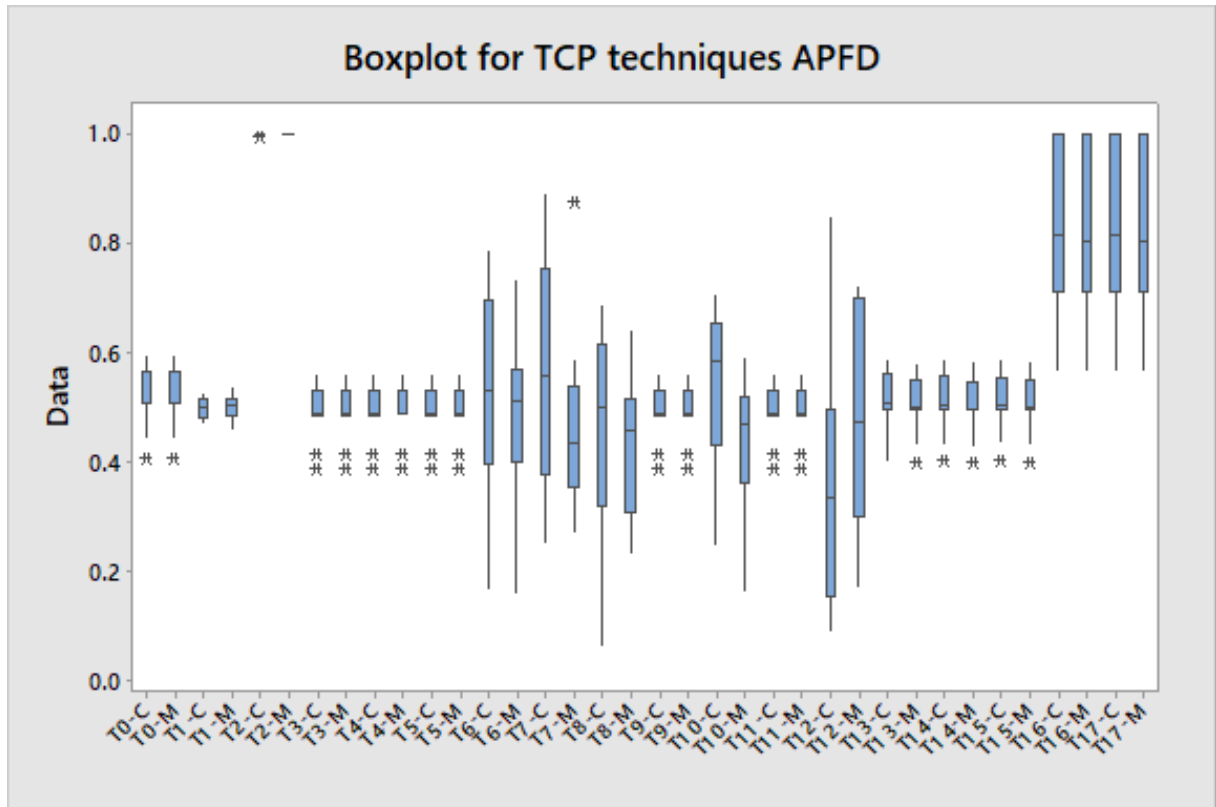


Figure 6.8: Boxplot of APFD values obtained for Jackson-databind project.

6.2.4.2 Jackson-databind

To summarize collected data, box plots are depicted in Figure 6.8 for all executed TCP techniques for Jackson-databind project.

Research question 1 is concerned with comparing the use of different TCP techniques against the non-use, which is the control technique T0-C in Table 6.2. Thus, the design of this analysis is a series of sixteen sub-experiments with one factor, which is the use or not of a TCP technique, and two treatments, whose values are APFD obtained using T0-C (fixed) and using the remaining techniques displayed in Table 6.2 (T1-C and T3-C through T17-C). Sub-experiment comparisons are listed in Table 6.8.

For this analysis it is used 11 builds of the Jackson-databind software with 15 different test case orderings from TCP techniques T3 through T17 and 100 orderings from T1 for each build. This results in a total of 1276 different APFD values (trials) collected for this research question.

All normality distribution checking of the samples were done with Shapiro-Wilk test, since for each sample analyzed we have 11 data points (builds) and that this test is recommended when samples have less than 30 data points.

Table 6.8: Comparisons for RQ1 of Jackson-databind experiment.

#	Comparison
1	T0-C x T1-C
2	T0-C x T3-C
3	T0-C x T4-C
4	T0-C x T5-C
5	T0-C x T6-C
6	T0-C x T7-C
7	T0-C x T8-C
8	T0-C x T9-C
9	T0-C x T10-C
10	T0-C x T11-C
11	T0-C x T12-C
12	T0-C x T13-C
13	T0-C x T14-C
14	T0-C x T15-C
15	T0-C x T16-C
16	T0-C x T17-C

The normality test of the fixed treatment (T0-C) results in a p-value > 0.1 . As it is greater than the significance level of 0.05, the sample can be characterized as normal. The output of the normality test is depicted in Figure 6.9, which is produced by the statistical tool.

Sub-experiment #1 compares APFD values obtained when no TCP technique (T0-C) was used against when the random TCP technique (T1-C) was used.

The normality test of T1-C results in a p-value > 0.1 . As it is greater than the significance level of 0.05, the sample can be characterized as normal. The output of the normality test is depicted in Figure 6.10.

Since both T0-C and T1-C have normal distribution, the homoscedasticity are also checked. Levene's test is used for this purpose and the result is displayed in Figure 6.11. The resulting p-value of 0.02, which is less than the significance level of 0.05, suggests that the sample is not homoscedastic.

Considering that both variables have normal distribution but are not homoscedastic, a non-parametric hypothesis test must be used. Since the design of this analysis is 1 factor and 2 treatments, Mann-Whitney test is chosen. Thus, hypotheses are defined as:

- H_0 : Medians obtained using T0-C are equal to medians obtained using T1-C.
- H_1 : Medians obtained using T0-C are significantly different to medians obtained

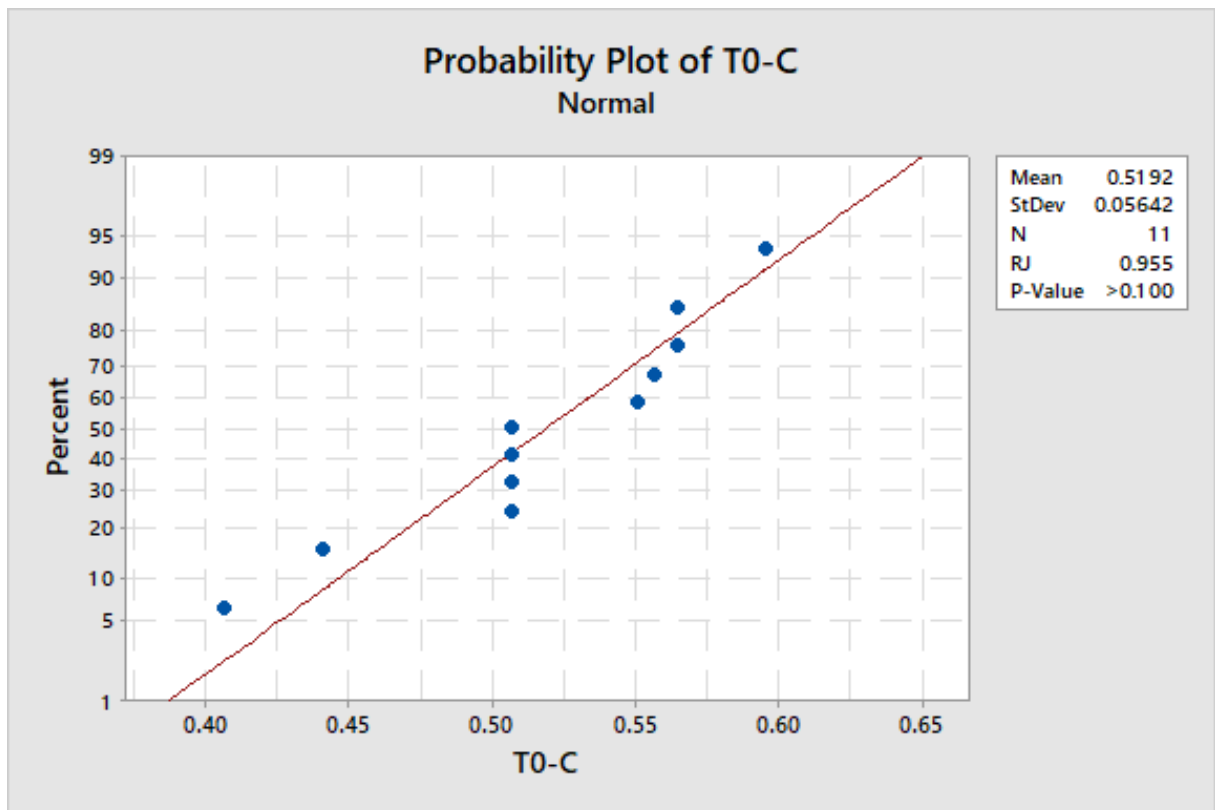


Figure 6.9: Normality test for T0-C from Jackson-databind.

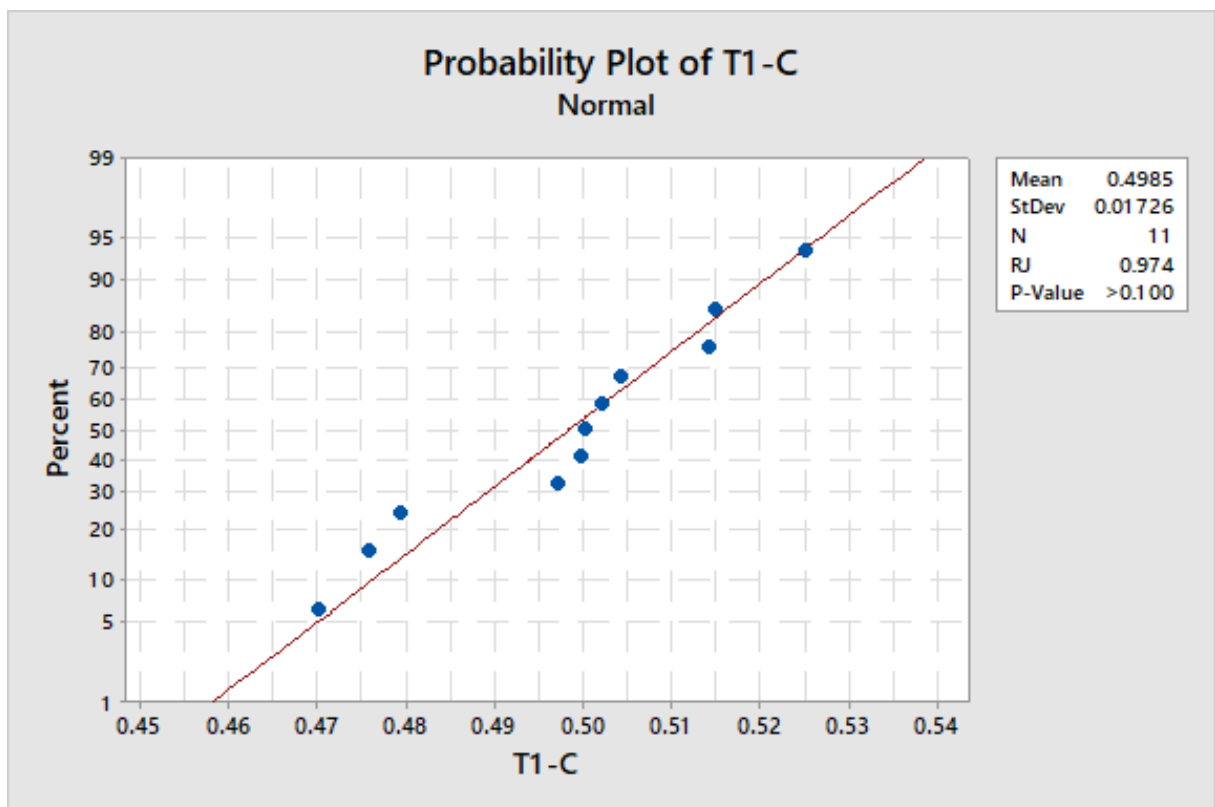


Figure 6.10: Normality test for T1-C from Jackson-databind.

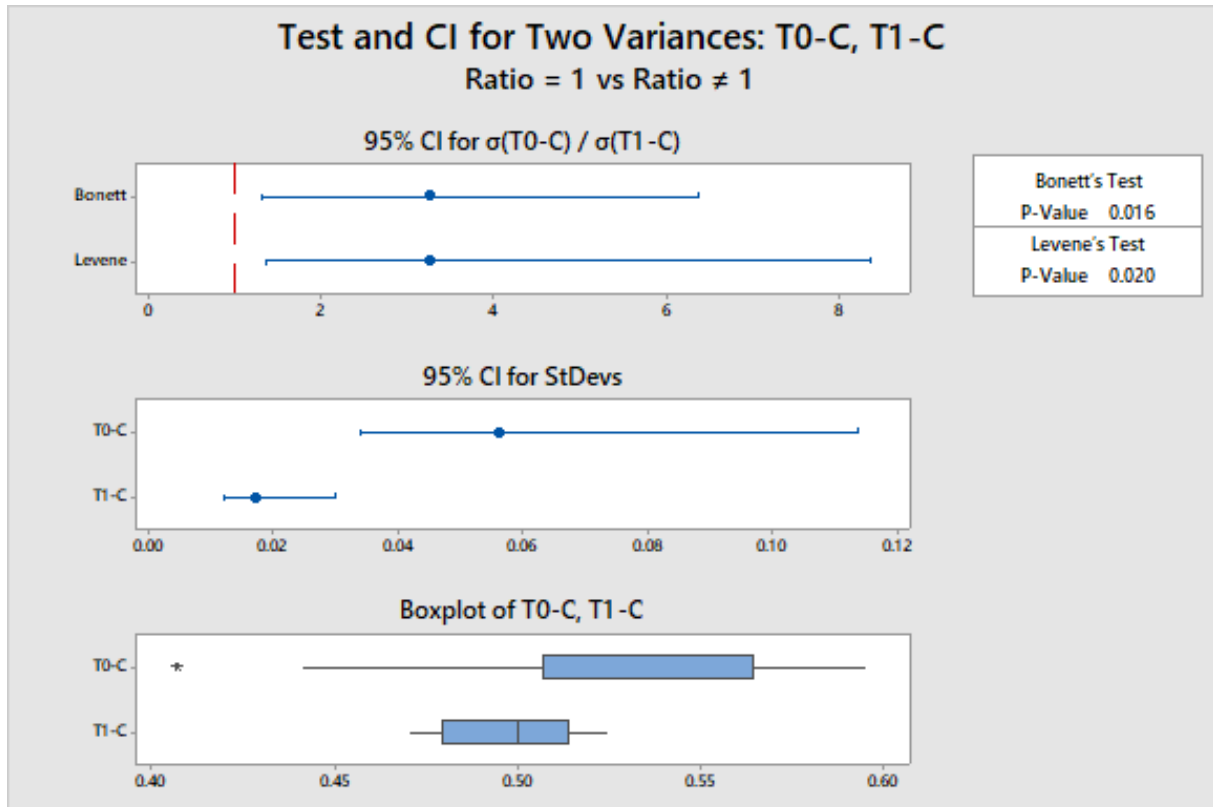


Figure 6.11: Homoscedasticity test for T0-C and T1-C from Jackson-databind project.

using T1-C.

Hypothesis test output is displayed in Listing 6.3. Obtained p-value is 0.0878, which is greater than the significance level of 0.05. This result indicates that H_0 cannot be rejected and suggests that the medians are statistically equivalent.

Listing 6.3: Hypothesis test result for sub-experiment #1 of RQ1 from Jackson-databind.

Mann-Whitney Test and CI: T0-C, T1-C

	N	Median
T0-C	11	0.50739
T1-C	11	0.50030

Point estimate for $\eta_1 - \eta_2$ is 0.03134
 95.1 Percent CI for $\eta_1 - \eta_2$ is (-0.00745, 0.06282)
 W = 153.0
 Test of $\eta_1 = \eta_2$ vs $\eta_1 \neq \eta_2$ is significant at 0.0878
 The test is significant at 0.0877 (adjusted for ties)

To prevent repetitive text for each sub-experiment, analyses of normality, homoscedasticity and hypothesis test used for each sub-experiment is listed in Table 6.9. Hypothesis test results are displayed in Table 6.10.

13% of the comparisons result in a TCP technique achieving statistically better ef-

Table 6.9: Normality and homoscedasticity analyses for RQ1 of Jackson-databind.

#	Comparison	Normality	Homoscedasticity	Hypothesis test
1	T0-C x T1-C	✓	Not homoscedastic (p-value = 0.02)	Mann-Whitney
2	T0-C x T3-C	✓	✓	Test-T
3	T0-C x T4-C	✓	✓	Test-T
4	T0-C x T5-C	✓	✓	Test-T
5	T0-C x T6-C	✓	Not homoscedastic (p-value = 0.01)	Mann-Whitney
6	T0-C x T7-C	✓	Not homoscedastic (p-value = 0)	Mann-Whitney
7	T0-C x T8-C	✓	Not homoscedastic (p-value = 0.014)	Mann-Whitney
8	T0-C x T9-C	✓	✓	Test-T
9	T0-C x T10-C	✓	✓	Test-T
10	T0-C x T11-C	✓	✓	Test-T
11	T0-C x T12-C	✓	Not homoscedastic (p-value = 0.006)	Mann-Whitney
12	T0-C x T13-C	✓	✓	Test-T
13	T0-C x T14-C	✓	✓	Test-T
14	T0-C x T15-C	✓	✓	Test-T
15	T0-C x T16-C	✓	Not homoscedastic (p-value = 0.008)	Mann-Whitney
16	T0-C x T17-C	✓	Not homoscedastic (p-value = 0.008)	Mann-Whitney

Table 6.10: Hypothesis tests results for RQ1 of Jackson-databind.

#	Comparison	Hypothesis test result	APFD comparison result
1	T0-C x T1-C	T0-C = T1-C p-value (0.0877)	T0-C (0.5074) = T1-C (0.5003)
2	T0-C x T3-C	T0-C = T3-C p-value (0.249)	T0-C (0.5192) = T3-C (0.4917)
3	T0-C x T4-C	T0-C = T4-C p-value (0.249)	T0-C (0.5192) = T4-C (0.4917)
4	T0-C x T5-C	T0-C = T5-C p-value (0.249)	T0-C (0.5192) = T5-C (0.4917)
5	T0-C x T6-C	T0-C = T6-C p-value (1.000)	T0-C (0.5074) = T6-C (0.5285)
6	T0-C x T7-C	T0-C = T7-C p-value (0.7427)	T0-C (0.5074) = T7-C (0.5567)
7	T0-C x T8-C	T0-C = T8-C p-value (0.7426)	T0-C (0.5074) = T8-C (0.4974)
8	T0-C x T9-C	T0-C = T9-C p-value (0.249)	T0-C (0.5192) = T9-C (0.4917)
9	T0-C x T10-C	T0-C = T10-C p-value (0.877)	T0-C (0.5192) = T10-C (0.527)
10	T0-C x T11-C	T0-C = T11-C p-value (0.249)	T0-C (0.5192) = T11-C (0.4917)
11	T0-C x T12-C	T0-C \neq T12-C p-value (0.0215)	T0-C (0.5074) > T12-C (0.3347)
12	T0-C x T13-C	T0-C = T13-C p-value (0.752)	T0-C (0.5192) = T13-C (0.5114)
13	T0-C x T14-C	T0-C = T14-C p-value (0.725)	T0-C (0.5192) = T14-C (0.5107)
14	T0-C x T15-C	T0-C = T15-C p-value (0.754)	T0-C (0.5192) = T15-C (0.5116)
15	T0-C x T16-C	T0-C \neq T16-C p-value (0.0002)	T0-C (0.5074) < T16-C (0.8148)
16	T0-C x T17-C	T0-C \neq T17-C p-value (0.0002)	T0-C (0.5074) < T17-C (0.8148)

Table 6.11: Comparisons for RQ2 of Jackson-databind experiment.

#	Control x intervention
1	T1-C x T1-M
2	T3-C x T3-M
3	T4-C x T4-M
4	T5-C x T5-M
5	T6-C x T6-M
6	T7-C x T7-M
7	T8-C x T8-M
8	T9-C x T9-M
9	T10-C x T10-M
10	T11-C x T11-M
11	T12-C x T12-M
12	T13-C x T13-M
13	T14-C x T14-M
14	T15-C x T15-M
15	T16-C x T16-M
16	T17-C x T17-M

fectiveness than the non-use of a TCP technique, as shown by highlighted rows in Table 6.10, for RQ1 analysis of Jackson-databind project. 81% resulted in statistically equivalent effectiveness and the remaining 6% resulted in better effectiveness for the non-use of a TCP technique compared to the use of a TCP technique.

Research question 2 is concerned with comparing TCP techniques effectiveness when executed at the method and class test level. Thus, the design of this analysis is a series of sixteen sub-experiments with one factor, which is the granularity of the tests, and two treatments, whose values are method and class level.

In this analysis, the same project builds are subjected to the same intervention (TCP technique) for each of the treatments. For this reason, we say that the analysis is paired.

For this analysis it is used 11 builds of the Jackson-databind software with 15 different test case orderings from TCP techniques T3 through T17 and 100 orderings from T1 for each build. Furthermore, for each ordering we have two different test granularities being used. This results in a total of 2530 different APFD values (trials) collected for this research question. Sub-experiments are listed in Table 6.11.

As in RQ1 analysis, the first sub-experiment data analysis is described in details and the remaining are summarized.

Sub-experiment #1 compares T1 executed at the class level test granularity (T1-C)

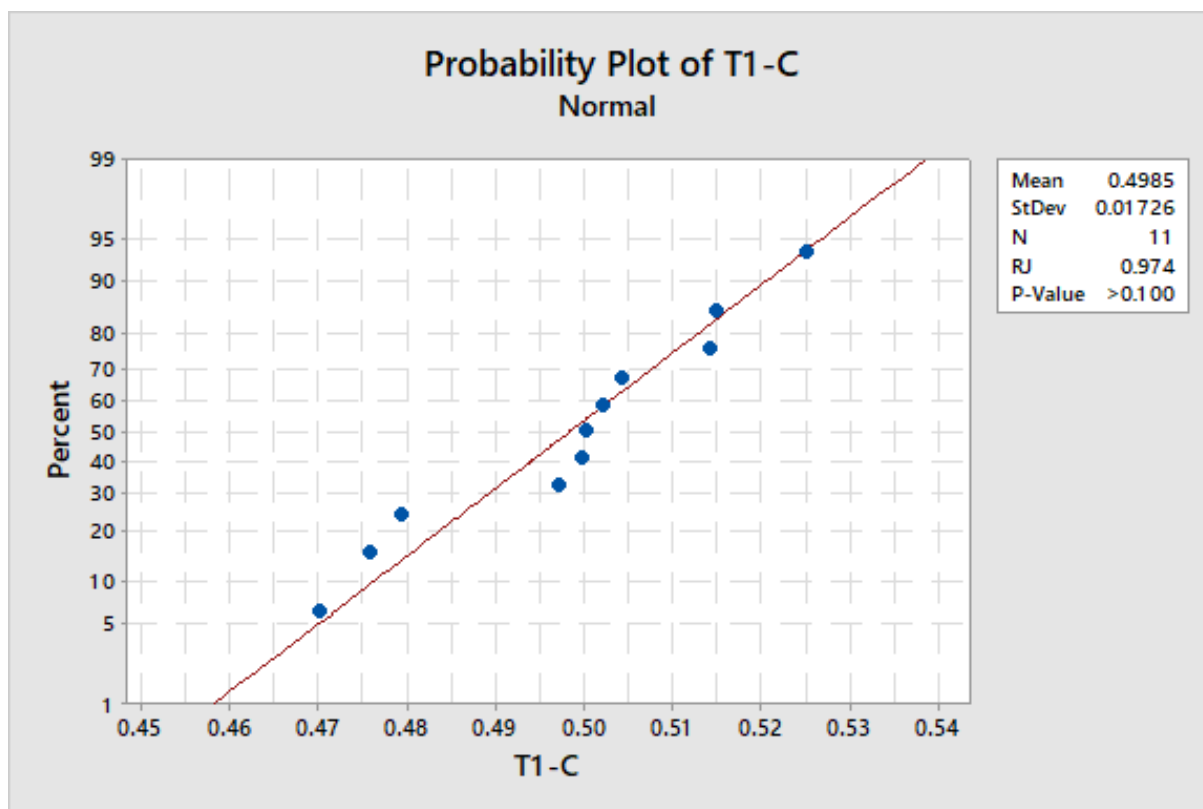


Figure 6.12: Normality test for T1-C from Jackson-databind.

and at the method level granularity (T1-M). Using Shapiro-Wilk to test normality for each sample, results in a p-value > 0.100 for T1-C, as depicted in Figure 6.12 and for T1-M, as depicted in Figure 6.13. As both p-values are greater than the established significance level of 0.05, it suggests that the samples are normally distributed.

Homoscedasticity of the samples are also checked, using Levene's test, depicted in Figure 6.14. A p-value of 0.633 is obtained, which is greater than the significance level of 0.05, indicating that the samples are homoscedastic.

As the samples are normal and homoscedastic, a parametric hypothesis test can be used. Since we have a paired design of 1 factor and 2 treatments, Paired T-test is used. Hypotheses for this analysis are defined as:

- H_0 : APFD mean obtained using T1-C is equal to the mean obtained using T1-M.
- H_1 : APFD mean obtained using T1-C is significantly different from the mean obtained using T1-M.

The hypothesis test output from Minitab is displayed in Listing 6.4. The p-value obtained is 0.847, which is bigger than the significance level of 0.05. This result indicates that the

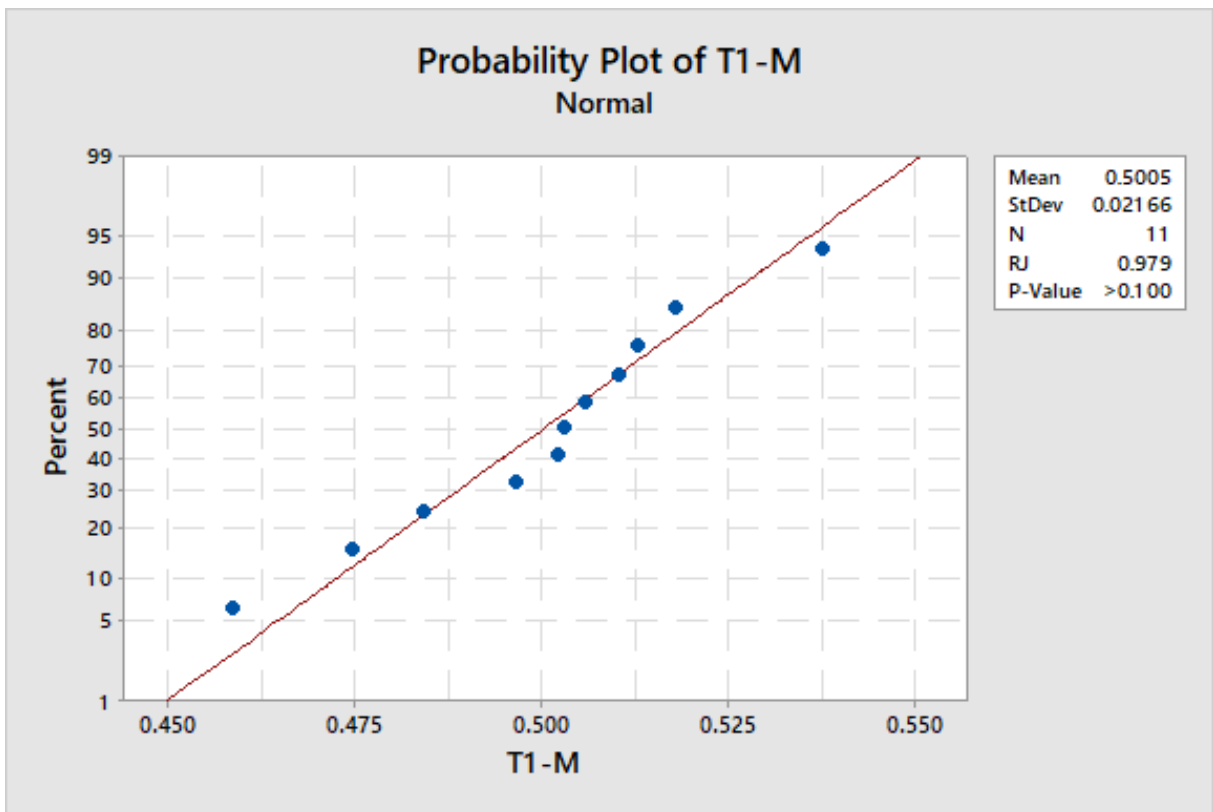


Figure 6.13: Normality test for T1-M from Jackson-databind.

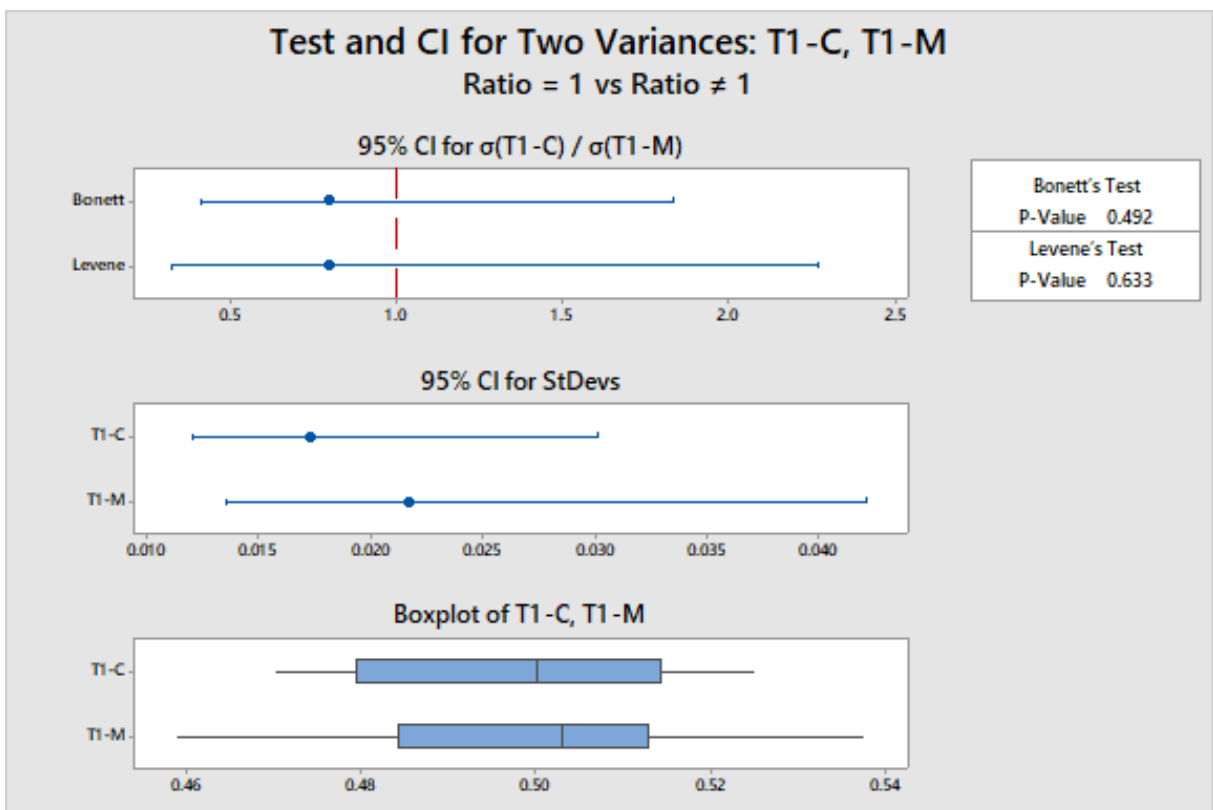


Figure 6.14: Homoscedasticity test for T1-C and T1-M from Jackson-databind.

Table 6.12: Hypothesis tests results for RQ2 of Jackson-databind.

#	Comparison	Hypothesis test result	APFD comparison result
1	T1-C x T1-M	T1-C = T1-M p-value (0.847)	(0.4985) T1-C = 1-M (0.5005)
2	T3-C x T3-M	T3-C = T3-M p-value (0.817)	(0.4917) T3-C = 3-M (0.4917)
3	T4-C x T4-M	T4-C = T4-M p-value (0.064)	(0.4917) T4-C = 4-M (0.4920)
4	T5-C x T5-M	T5-C = T5-M p-value (0.817)	(0.4917) T5-C = 5-M (0.4917)
5	T6-C x T6-M	T6-C = T6-M p-value (0.414)	(0.5160) T6-C = 6-M (0.4712)
6	T7-C x T7-M	T7-C = T7-M p-value (0.117)	(0.5744) T7-C = 7-M (0.4684)
7	T8-C x T8-M	T8-C = T8-M p-value (0.674)	(0.4542) T8-C = 8-M (0.4284)
8	T9-C x T9-M	T9-C = T9-M p-value (0.817)	(0.4917) T9-C = 9-M (0.4917)
9	T10-C x T10-M	T10-C = T10-M p-value (0.145)	(0.5269) T10-C > 10-M (0.4376)
10	T11-C x T11-M	T11-C = T11-M p-value (0.817)	(0.4917) T11-C = 11-M (0.4917)
11	T12-C x T12-M	T12-C = T12-M p-value (0.322)	(0.3773) T12-C = 12-M (0.4689)
12	T13-C x T13-M	T13-C = T13-M p-value (0.058)	(0.5114) T13-C = 13-M (0.5066)
13	T14-C x T14-M	T14-C \neq T14-M p-value (0.004)	(0.5107) T14-C > 14-M (0.5064)
14	T15-C x T15-M	T15-C \neq T15-M p-value (0.000)	(0.5116) T15-C > 15-M (0.5066)
15	T16-C x T16-M	T16-C = T16-M p-value (0.364)	(0.8281) T16-C = 16-M (0.8111)
16	T17-C x T17-M	T17-C = T17-M p-value (0.364)	(0.8279) T17-C = 17-M (0.8111)

null hypothesis H_0 cannot be rejected and thus suggests that the means obtained using T1-C and T1-M are statistically equivalent.

Listing 6.4: Hypothesis test result for sub-experiment #1 of RQ2 from Jackson-databind.

Paired T for T1-C - T1-M				
	N	Mean	StDev	SE Mean
T1-C	11	0.49853	0.01726	0.00520
T1-M	11	0.50050	0.02166	0.00653
Difference	11	-0.00197	0.03298	0.00994

95% CI for mean difference: (-0.02413, 0.02019)
T-Test of mean difference = 0 (vs \neq 0): T-Value = -0.20 P-Value = 0.847

To prevent repetitive text for each sub-experiment, remaining sub-experiment analyses are summarized in Table 6.12. In this analysis, all samples are normal and homoscedastic and thus, Paired T-Test was used for all sub-experiments.

According to Table 6.12, no comparison resulted in better effectiveness for a TCP technique executed at the method granularity when compared to it executed at the class granularity. 81% of the comparisons resulted in statistically equivalent effectiveness for both granularities and the remaining 19% performed better at the class than at the method granularity.

6.2.5 THREATS TO VALIDITY

Some factors can affect the validity of the results obtained in this experiment. They are reported according to the guidelines provided by Wohlin et al. (2012).

External validity is concerned with the generalization of the results from this study. It is worth of note that the chosen open source projects might not represent all open source projects. To mitigate this problem, we choose two projects among the list of most famous projects on Github that met our experiment requirements. However, more investigation is needed in order to generalize the results.

Internal validity is concerned with factors that can affect the independent variables and are not known when the experiment is performed. In this case, the proposed framework used during the experimentation process might contain bugs that were not yet discovered and that could affect the outcomes of the experiment. To mitigate this, we extensively tested the framework on smaller projects, so we could debug and check if the results were valid.

6.3 DISCUSSION AND LESSONS LEARNED

Regarding research question 1, for CoreNLP, 13 out of 16 experimented techniques performed better than the non-use of a TCP technique. Three techniques (T6-C, T7-C and T13-C) did not perform better than non-use of a TCP technique. Despite the fact that they achieve a higher APFD median than the control T0-C, the hypothesis tests suggest that the difference between them is not statistically significant.

For Jackson-databind project, only two TCP techniques performed better than the non-use of a TCP technique. They are the most failures first (T16-C) and recent failures first (T17-C). Both are history-based. Additional diff class coverage (T12-C) performed worse than the non-use of a TCP technique. All other techniques performed statistically similar to the default approach.

These results suggest that if the developers of these open source projects had used a history-based TCP technique at the class test granularity, they would discover faults in their project earlier than they actually did, when not using test case prioritization. Furthermore, it also suggests that some of the TCP techniques may not be worth of use for these projects, since they perform similar to not using them.

Regarding research question 2, for both projects, the results obtained indicate that executing those TCP techniques at the test method granularity do not improve the rate of fault detection, opposed to what was found in our systematic literature review. This result strengthens findings from Do et al. (2006). In their experimental study, they found that executing test cases at the test method level for a Java project did not improve the rate of fault detection, comparing to test cases at the test class level.

Despite the results we found, only two projects were experimented, and they might not represent all open source projects. In this sense, new studies are needed to investigate this question. During the execution of this study, some lessons were learned and we think that would be useful to report as other researchers may face the same challenges.

One lesson is that prioritizing test cases at the method level is only useful for industry if the project to be tested has unit tests that are truly unit tests. That is, they are independent from each other in the sense that, executing them at a specific order should not change their behavior and make them fail. We observed that this phenomenon happened during our experiment in the CoreNLP project. Different executions from the same build of the project presented different amount of faults. This can potentially impact in the final effectiveness of a TCP technique and report faults that do not really exist in the source code.

Another lesson is regarding the use of a test framework to support our empirical experiment. Before actually designing and building this framework, performing an empirical experiment was much harder and laborious. We would need, for example, when we extracted data to answer our first research question for CoreNLP, where 1276 APFD values were generated, to manually trigger the build of the project 1276 times, each time with a different configuration. With our proposed framework, we executed 4 times to generate all APFD 5060 values for this experiment. Internally the framework automatize almost every task in this process, like executing a test suite using a TCP technique, identifying faults, calculating APFD values and generating reports.

6.4 FINAL CONSIDERATIONS

In this chapter we demonstrate the support that the framework provides to the execution and experimentation process of TCP techniques through an experiment. This answers the third research question of this work, "How does the resulting framework support

practitioners and researchers?”.

We investigate through an empirical study whether or not implemented TCP techniques improve the rate of fault detection on two open source projects and furthermore, if test execution granularity impacts on obtained APFD from those techniques.

As results we found that some of the techniques could improve the rate of fault detection on the projects, and overall, the two implemented history-based TCP techniques yielded the better improvements. We also found that executing those TCP techniques at the test method granularity level may not be worth for those projects, since they do not perform better than when executed at the class test level.

7 CONCLUSION

In this work we highlighted the benefits of using TCP techniques in the continuous software engineering environment, where processes are constantly optimized.

To answer our research question regarding how TCP is being used in industry and literature, we performed a systematic literature review and mapping and a structured search of the literature for TCP industrial usage evidence. We found that there is a lack of automated tools to support both research and industrial use of such techniques. These evidence suggest that TCP is still not mature enough in literature to allow practitioners to choose TCP techniques based on experimental evidence.

Aiming to support TCP research and usage, our second research question that guided this work is concerned with how to create a framework to support both TCP usage and experimentation. The literature has been surveyed for related works and how they implement features that ease this process. After that, we designed and implemented a framework to support the execution, experimentation and implementation of TCP techniques.

The last research question of this work aim to answer how the resulting framework support practitioners and researchers working with TCP. To answer that, an experiment was performed using two open source projects from Github. In this study, 16 TCP techniques were compared in terms of rate of faults detection to the baseline approach of regression testing, which consists of not using a TCP technique. Results from the empirical study suggest that using those TCP techniques result in faster feedback about the existence of failures in the projects build, possibly resulting in shorter development cycles. They also suggest that executing those TCP techniques at the method test level, that is, executing one test method per time as opposed to the traditional approach of executing entire test classes, are not worth for these projects, since they do not improve the rate of fault detection. Furthermore, the execution of the experiment was heavily supported by the automation the framework provides to the process, by executing different TCP techniques with different configurations and measuring their effectiveness.

All research questions and objectives stated in Chapter 1 were visited throughout the conduction of this work. We hope that the contributions will help both researchers and

practitioners on further understanding the benefits of using TCP in continuous software engineering environment.

7.1 RESEARCH LIMITATIONS

The following limitations were identified in this work:

- Optimus Framework was designed to work with projects that use the Maven build tool. For this reason, it may not be useful for users that have projects with different build tools.
- If researchers develop TCP techniques that rely on project information different from those offered by Optimus Framework analyzers, they will need to develop such mechanism to capture them. However, this might not be hard, since the project is divided into modules.
- Some techniques, such as Farthest-first Ordered Sequence and Greed-aided-clustering Ordered Sequence implemented in the framework were not included in the empirical study for viability reasons. However, they were tested on smaller projects outside the empirical study.

7.2 FUTURE WORKS

Future works include experimenting different projects to check if the results are also valid for them. In addition, it may be useful to replicate existent studies from literature using the Optimus Framework to validate if results are similar. In order to do that, it is necessary to adapt the projects used in other studies to be supported by Optimus Framework, since they do not use Maven.

Another important point regarding the support for experimentation is the implementation of support for multi-module Maven projects, since big open source projects use this feature. This might be a big step to allow bigger experimental studies and a step towards more understanding of impacts of using TCP techniques on different kinds of projects. Lastly, another future work is to evaluate the framework in terms of performance, measuring the cost-benefit of using different TCP techniques using our approach.

REFERENCES

- ALVES, E. L. G.; MACHADO, P. D. de L.; MASSONI, T.; KIM, M. Prioritizing test cases for early detection of refactoring faults. **Softw. Test., Verif. Reliab.**, v. 26, p. 402–426, 2016.
- AMMANN, P.; OFFUTT, J. **Introduction to software testing**, 2016.
- ANDREWS, J. H.; BRIAND, L. C.; LABICHE, Y. Is mutation an appropriate tool for testing experiments? In: **ACM. Proceedings of the 27th international conference on Software engineering**, 2005. p. 402–411.
- BUSJAEGER, B.; XIE, T. Learning for test prioritization: an industrial case study. In: **ACM. Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering**, 2016. p. 975–980.
- CALDIERA, V. R. B.-G.; ROMBACH, H. D. Goal question metric paradigm. **Encyclopedia of software engineering**, v. 1, p. 528–532, 1994.
- CARLSON, R.; DO, H.; DENTON, A. M. A clustering approach to improving test case prioritization: An industrial case study. **2011 27th IEEE International Conference on Software Maintenance (ICSM)**, p. 382–391, 2011.
- CATAL, C.; MISHRA, D. Test case prioritization: a systematic mapping study. **Software Quality Journal**, v. 21, p. 445–478, 2012.
- CHEN, L. Continuous delivery: Huge benefits, but challenges too. **IEEE Software**, v. 32, p. 50–54, 2015.
- CZERWONKA, J.; DAS, R.; NAGAPPAN, N.; TARVO, A.; TETEREV, A. Crane: Failure prediction, change analysis and test prioritization in practice – experiences from windows. **2011 Fourth IEEE International Conference on Software Testing, Verification and Validation**, p. 357–366, 2011.
- DO, H. Recent advances in regression testing techniques. In: **Advances in Computers**, 2016. v. 103, p. 53–77.

- DO, H.; MIRARAB, S.; TAHVILDARI, L.; ROTHERMEL, G. The effects of time constraints on test case prioritization: A series of controlled experiments. **IEEE Transactions on Software Engineering**, IEEE, n. 5, p. 593–617, 2010.
- DO, H.; ROTHERMEL, G. On the use of mutation faults in empirical assessments of test case prioritization techniques. **IEEE Transactions on Software Engineering**, v. 32, p. 733–752, 2006.
- DO, H.; ROTHERMEL, G.; KINNEER, A. Prioritizing junit test cases: An empirical assessment and cost-benefits analysis. **Empirical Software Engineering**, v. 11, p. 33–70, 2006.
- EGHBALI, S.; TAHVILDARI, L. Test case prioritization using lexicographical ordering. **IEEE Transactions on Software Engineering**, v. 42, p. 1178–1195, 2016.
- ELBAUM, S. G.; ROTHERMEL, G.; PENIX, J. Techniques for improving regression testing in continuous integration development environments. In: **SIGSOFT FSE**, 2014.
- ENGSTRÖM, E.; RUNESON, P.; LJUNG, A. Improving regression testing transparency and efficiency with history-based prioritization – an industrial case study. **2011 Fourth IEEE International Conference on Software Testing, Verification and Validation**, p. 367–376, 2011.
- FANG, C.; CHEN, Z.; WU, K.; ZHAO, Z. Similarity-based test case prioritization using ordered sequences of program entities. **Software Quality Journal**, v. 22, p. 335–361, 2013.
- FITZGERALD, B.; STOL, K.-J. Continuous software engineering: A roadmap and agenda. **Journal of Systems and Software**, Elsevier, v. 123, p. 176–189, 2017.
- JACCARD, P. Étude comparative de la distribution florale dans une portion des alpes et des jura. **Bull Soc Vaudoise Sci Nat**, v. 37, p. 547–579, 1901.
- JIANG, B.; ZHANG, Z.; CHAN, W. K.; TSE, T. H. Adaptive random test case prioritization. **2009 IEEE/ACM International Conference on Automated Software Engineering**, p. 233–244, 2009.

- JUNIOR, H. de S. C.; ARAÚJO, M. A. P.; DAVID, J. M. N.; BRAGA, R. M. M.; CAMPOS, F.; STRÖELE, V. Test case prioritization: a systematic review and mapping of the literature. In: **SBES**, 2017.
- KAUFFMAN, J. M.; KAPFHAMMER, G. M. A framework to support research in and encourage industrial adoption of regression testing techniques. **2012 IEEE Fifth International Conference on Software Testing, Verification and Validation**, p. 907–908, 2012.
- KIM, J.-M.; PORTER, A. A. A history-based test prioritization technique for regression testing in resource constrained environments. In: **ICSE**, 2002.
- KITCHENHAM, B. **Guidelines for performing systematic literature reviews in software engineering**, 2007.
- LEVENSHTAIN, V. Binary codes capable of correcting spurious insertions and deletion of ones. **Problems of information Transmission**, v. 1, n. 1, p. 8–17, 1965.
- MARIJAN, D.; GOTLIEB, A.; SEN, S. Test case prioritization for continuous regression testing: An industrial case study. **2013 IEEE International Conference on Software Maintenance**, p. 540–543, 2013.
- MARIJAN, D.; LIAAEN, M. Effect of time window on the performance of continuous regression testing. **2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)**, p. 568–571, 2016.
- MEYER, M. Continuous integration and its tools. **IEEE Software**, v. 31, p. 14–16, 2014.
- MIRARAB, S.; TAHVILDARI, L. An empirical study on bayesian network-based approach for test case prioritization. **2008 1st International Conference on Software Testing, Verification, and Validation**, p. 278–287, 2008.
- MYERS, G. J.; SANDLER, C.; BADGETT, T. **The art of software testing**, 2011.
- NARDO, D. D.; ALSHAHWAN, N.; BRIAND, L. C.; LABICHE, Y. Coverage-based test case prioritisation: An industrial case study. **2013 IEEE Sixth International Conference on Software Testing, Verification and Validation**, p. 302–311, 2013.

- ÖHLIN, P. **Prioritizing Tests with Spotify's Test & Build Data using History-based, Modification-based & Machine Learning Approaches**. Tese (Doutorado) — Linköping University, Sweden, S-581 83 Linköping, Sweden, 2017. Master Thesis.
- PAREJO, J. A.; SÁNCHEZ, A. B.; SEGURA, S.; CORTÉS, A. R.; LOPEZ-HERREJON, R. E.; EGYED, A. Multi-objective test case prioritization in highly configurable systems: A case study. **Journal of Systems and Software**, v. 122, p. 287–310, 2016.
- PATERSON, D.; KAPFHAMMER, G. M.; FRASER, G.; MCMINN, P. Using controlled numbers of real faults and mutants to empirically evaluate coverage-based test case prioritization. In: IEEE. **Proceedings of the International Workshop on Automation of Software Test (AST 2018)**, 2018.
- PFLEEGER, S. L.; ATLEE, J. M. (Ed.). **Software engineering - theory and practice (4. ed.)**, 2009.
- PLEWNIA, C. **A Framework for Regression Test Prioritization and Selection**. Tese (Doutorado) — RWTH Aachen University, Germany, Aachen, Germany, 2015. Master Thesis.
- ROTHERMEL, G.; UNTCH, R. H.; CHU, C.; HARROLD, M. J. Test case prioritization: An empirical study. In: **ICSM**, 1999.
- ROTHERMEL, G.; UNTCH, R. H.; CHU, C.; HARROLD, M. J. Prioritizing test cases for regression testing. **IEEE Transactions on software engineering**, IEEE, v. 27, n. 10, p. 929–948, 2001.
- SÁNCHEZ, A. B.; SEGURA, S. Smartest: A test case prioritization tool for drupal. In: **SPLC**, 2017.
- SINGH, Y.; KAUR, A.; SURI, B.; SINGHAL, S. Systematic literature review on regression test prioritization techniques. **Informatica (Slovenia)**, v. 36, p. 379–408, 2012.
- SPIEKER, H.; GOTLIEB, A.; MARIJAN, D.; MOSSIGE, M. Reinforcement learning for automatic test case prioritization and selection in continuous integration. In: ACM. **Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis**, 2017. p. 12–22.

- SRIKANTH, H.; COHEN, M. B. Regression testing in software as a service: An industrial case study. **2011 27th IEEE International Conference on Software Maintenance (ICSM)**, p. 372–381, 2011.
- STRANDBERG, P. E.; SUNDMARK, D.; AFZAL, W.; OSTRAND, T. J.; WEYUKER, E. J. Experience report: Automated system level regression test prioritization using multiple factors. **2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)**, p. 12–23, 2016.
- WANG, R.; JIANG, S.; CHEN, D. Similarity-based regression test case prioritization. In: **SEKE**, 2015.
- WANG, S.; ALI, S.; YUE, T.; BAKKELI, O.; LIAAEN, M. Enhancing test case prioritization in an industrial setting with resource awareness and multi-objective search. **2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)**, p. 182–191, 2016.
- WOHLIN, C.; RUNESON, P.; HÖST, M.; OHLSSON, M. C.; REGNELL, B.; WESSLÉN, A. **Experimentation in software engineering**, 2012.
- WU, K.; FANG, C.; CHEN, Z.; ZHAO, Z. Test case prioritization incorporating ordered sequence of program elements. **2012 7th International Workshop on Automation of Software Test (AST)**, p. 124–130, 2012.
- YANG, Y.; HUANG, X.; HAO, X.; LIU, Z.; CHEN, Z. An industrial study of natural language processing based test case prioritization. **2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)**, p. 548–549, 2017.
- YOO, S.; HARMAN, M. Regression testing minimization, selection and prioritization: a survey. **Software Testing, Verification and Reliability**, Wiley Online Library, v. 22, n. 2, p. 67–120, 2012.