Enhancing Automated Software Refactoring via Simultaneous Testing, Dependency Analysis, and Examining Multi-level Software Quality Concerns

by

Jeffrey J. Yackley

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer and Information Science)
in the University of Michigan-Dearborn
2022

Doctoral Committee:

Professor Marouane Kessentini, Co-Chair
Professor Bruce R. Maxim, Co-Chair
Assistant Professor Abdallah Chehade
Assistant Professor Foyzul Hassan
Associate Professor Zhiwei Xu

ISELab

Intelligent Software Engineering

© Jeffrey J. Yackley 2022

All Rights Reserved

For my parents, because of you “I am on my way; I can go the distance..."

ACKNOWLEDGEMENTS

It has been a challenging five years to bring this dissertation to fruition and I would like to thank everyone who made it possible and supported me on this amazing journey.

First, I would like to thank my co-advisors Profs. Marouane Kessentini and Bruce Maxim who spent countless hours guiding me through innumerable late nights, manuscript edits, and conference presentations. Your expertise and insight pushed me to be a better researcher and your encouragement kept me on track even through a global pandemic.

I would also like to thank Bruce for taking me on as his Padawan. You have been a fantastic mentor and role-model from the start of my undergraduate program in computer and information science so many years ago all through my graduate program. I've been honored to learn from you these past ten years and I know I wouldn't be half the instructor I am today if it wasn't for your advice and example as a brilliant, caring, student-focused, and fair professor.

Many thanks to my fellow ISE Lab members, in particular, Thiago Ferreira and Rafi Almhana for your friendship and support these many years.

To my parents, Sandra Yackley-Anderson and Richard Anderson, your faith in me, support, and love has been irreplaceable. I'm only able to write this dissertation because you never gave up on me and pushed me to succeed. I love and appreciate you both tremendously.

Next, I would like to thank my partner, Mariah Bauman, for putting up with the many sleepless nights and the long hours spent studying, writing, and coding while I

finished the work for this dissertation. Your love and support throughout this process has meant more to me than you will ever know.

Lastly, but by no means least, thank you to the CIS Department at U of M - Dearborn for providing the support to complete this thesis. In addition, thank you to our wonderful staff, particularly Kimberly LaPere and Allison Kerry for helping to keep things running smoothly and guiding me through all of the administrative hurdles of the program.

TABLE OF CONTENTS

DEDICATION ii

ACKNOWLEDGEMENTS iii

LIST OF FIGURES x

LIST OF TABLES xii

LIST OF ABBREVIATIONS xiii

ABSTRACT xvi

CHAPTER

I. Introduction 1

1.1 Research Context 1

1.2 Problem Statement 3

1.3 Contributions 5

1.3.1 Contribution 1: Simultaneous Refactoring and Regression Testing 5

1.3.2 Contribution 2: Detecting and Understanding Collections of Refactoring Recommendations 8

1.3.3 Contribution 3: Investigating the Relationships between Architecture and Code Anti-patterns Using Random Forest and Grid Search 12

1.4 Publication List 14

1.5 Organization of the Dissertation 15

II. State of the Art 16

2.1 Introduction 16

2.2 Software Refactoring 16

2.3 Automated Software Refactoring Recommendation Tools . . . 17

2.4 Software Refactoring Dependencies 18

2.5 Regression Testing 20

2.6 Software Anti-patterns 21

2.6.1 Architecture Anti-patterns 22

2.6.2 Code Anomalies 23

III. Simultaneous Refactoring and Regression Testing 25

3.1 A Motivating Example 25

3.2 Approach Overview 26

3.2.1 Simultasking 27

3.2.2 Adaptation 29

3.2.2.1 Unified Search Space Representation . . 30

3.2.2.2 Fitness Functions 33

3.2.2.3 Evolutionary Operators 35

3.3 Validation 37

3.3.1 Research Questions 37

3.3.1.1 RQ1-A: Quality Improvement 37

3.3.1.2 RQ1-B: Refactoring Meaningfulness . . 37

3.3.1.3 RQ2-A: Synergy between Regression Testing and Refactoring to Support Software Maintenance in Practice 38

3.3.1.4 RQ2-B: Testing Effort Reduction and Refactoring Coverage . . . 38
3.3.1.5 Study Context . . . 39
3.3.2 Data Collection . . . 40
3.3.2.1 RQ1 - Refactoring . . . 40
3.3.2.2 RQ2 - Regression Testing . . . 43
3.3.3 Experimental Settings and Data Analysis . . . 43
3.3.4 Results . . . 45
3.3.4.1 RQ1-A: Quality Improvement . . . 45
3.3.4.2 RQ1-B: Refactoring Meaningfulness . . . 46
3.3.4.3 RQ2-A and RQ2-B: Test Case Selection Coverage and Effort Reduction . . . 48
3.4 Threats to Validity . . . 50
3.5 Conclusion . . . 50
IV. Dependent or Not: Detecting and Understanding Collections of Refactorings . . . 52
4.1 A Motivating Example . . . 52
4.2 Refactoring Dependency Theory . . . 54
4.2.1 Definitions . . . 55
4.2.1.1 Refactoring Pre- and Post-Conditions . . . 56
4.2.2 Algorithm for Detecting Refactoring Dependencies . . . 57
4.2.3 DPRef . . . 60
4.3 Empirical Study . . . 62
4.3.1 Research Questions . . . 62
4.3.1.1 RQ1: Precision . . . 63
4.3.1.2 RQ2: Relation . . . 65

4.3.1.3 RQ3: Improvement 66
4.3.2 Experimental Settings 66
4.3.3 Results and Discussion 69
4.3.3.1 Results for RQ1 69
4.3.3.2 Results for RQ2 72
4.3.3.3 Results for RQ3 75
4.4 Threats to Validity 77
4.5 Implications and Future Work 78
4.5.1 Refactoring Pattern Extraction 79
4.5.2 Refactoring Collaborations Between Developers . . . 80
4.5.3 Change Operator in Search-based Refactoring . . . 80
4.5.4 Interactive Refactoring Tool Support 80
4.6 Conclusion 80
V. Investigating the Relationships between Architecture and Code Anti-patterns Using Random Forest and Grid Search 82
5.1 A Motivating Example 82
5.2 Approach Overview 85
5.3 Experimental Design 88
5.3.1 Research Questions 88
5.3.2 Experimental Setup and Formulae 88
5.3.2.1 Metrics for RQ1 88
5.3.2.2 Metrics for RQ2 89
5.3.2.3 Metrics for RQ3 90
5.3.2.4 Data Collection 91
5.3.2.5 Parameter Tuning 92

5.4 Experimental Results 93
5.4.1 Results for RQ1 93
5.4.2 Results for RQ2 98
5.4.3 Results for RQ3 102
5.5 Threats to Validity 105
5.6 Conclusion 106
VI. Conclusion 107
6.1 Summary 107
6.2 Future Work 108
BIBLIOGRAPHY 111

LIST OF FIGURES

Figure

1.1	Overview of the contributions of this thesis.	5
1.2	Example output of a refactoring recommendation tool: JDeodorant.	10
3.1	An example of decoding a random-key chromosome into domain-specific representations.	32
3.2	A summary of the 2-task environment.	33
3.3	An illustration of the assortative mating using the adapted crossover for knowledge transfer.	36
3.4	Median percentage of fixed code smells (NF) over 30 runs at the 95% confidence level ($\alpha < 5\%$).	46
3.5	Median quality gain (G) over 30 runs at the 95% confidence level ($\alpha < 5\%$).	46
3.6	The outcomes of the industrial validation on the JDI system by 6 developers during 5 days.	48
3.7	Median effort to test the refactorings (EF) over 30 runs at the 95% confidence level ($\alpha < 5\%$).	49
4.1	A simplified solution of 6 refactorings for the JFreeChart project.	53
4.2	Execution of Algorithm 1 on the example of Figure 4.1.	60
4.3	DPRef, a web-tool for detecting refactoring dependencies.	61
4.4	DPRef showing detected refactoring dependencies for JFreeChart.	61
4.5	Number of refactorings per non-trivial graph in each data set.	68
4.6	Box-plots of refactoring dependency correctness for the 9,595 projects.	70

4.7 Participant Survey. 71

4.8 Distribution of refactorings in trivial versus non-trivial graphs based on the 9,595 projects. 73

4.9 Size of non-trivial refactoring graphs in the 9,595 projects. 74

4.10 Distribution of the refactoring types among non-trivial graphs for the 9,595 projects. 74

4.11 The number of graphs that improved the quality metrics. 76

4.12 Rate of quality improvement (%) for the refactoring graphs per metric. 76

5.1 Visualization of anti-patterns and code anomalies for the BioInfo Project. 85

5.2 An overview of our approach. 86

5.3 Our web-app showing the relationships between code anomalies and architecture anti-patterns for the Opencsv project. 87

5.4 Predicting **Cyclic Dependencies** architecture-level anti-patterns from class-level code anomalies. 94

5.5 Predicting **God Package** architecture-level anti-patterns from class-level code anomalies. 95

5.6 Predicting **SAP Breaker** architecture-level anti-patterns from class-level code anomalies. 96

5.7 Each bar represents the number of occurrences of architecture-level anti-patterns that exist for each set of code anomalies. For example, the second bar indicates for almost 500 of the input data (a set of 16 code anomalies) two architecture-level anti-patterns exist at the same time. 97

5.8 Each bar represents how many of the data points have that specific architecture anti-pattern independent from other architecture anti-patterns that may exist in those data points. 98

5.9 Distribution of code anomalies per architecture anti-pattern type. . 101

LIST OF TABLES

Table

2.1 A brief description of different code anomalies used in our study. . . 24

3.1 Statistics of the studied systems. 39

3.2 Participants involved in RQ1-B. 43

3.3 RQ1-B: Would you apply the proposed refactorings? 47

4.1 Refactoring types and their pre- and post-condition rules. 58

4.2 QMOOD quality metrics. 63

4.3 Statistics of the subject projects. 67

4.4 Selected Participants. 68

5.1 Anti-patterns and code anomalies detected from Apache Cassandra. 84

5.2 Parameter Tuning Results 92

5.3 Evaluation Results 93

5.4 Detection rules for three architecture-level anti-patterns. 97

5.5 Most important features for the classification of the **SAP Breaker** Anti-pattern 98

5.6 Most important features for the classification of the **Cyclic Dependencies** Anti-pattern 99

5.7 Most important features for the classification of the **God Package** Anti-pattern 99

5.8 Results from the manual validation of the classifier. 103

LIST OF ABBREVIATIONS

AOP Aspect-Oriented Programming

BC Blob Class

BO Blob Operation

CBO Coupling Between Objects

DC Data Class

DCP Data Clumps

DH Distorted Hierarchy

DOE Design of Experiments

EAs Evolutionary Algorithms

ED External Duplication

EF Median Effort to Test Refactorings

FE Feature Envy

GC God Class

IEEE Institute of Electrical and Electronics Engineers

IC Intensive Coupling

ID Internal Duplication

LOC Lines of Code

MC Message Chain

MCr Manual Correctness

MFEA Multi-Factorial Evolutionary Algorithm

MNF Maximum Number of Features

MO-MFO Multi-Objective-Multi-Factorial Optimization

MOT Mono-task Multi-objective Regression Technique

MSL Minimum Sample Leaf Size

NASA National Aeronautics and Space Administration

NCCS Number of Corrected Code Smells

NDCS Number of Detected Code Smells

NF Number of Fixed Code Smells

NHTSA National Highway Traffic Safety Administration

NSGA-II Non-dominated Sorting Genetic Algorithm-II

NT Number of Trees in the Forest

NTCL Number of Times Code Anomaly Appeared

NTR Non-Trivial Rate

QMOOD Quality Model for Object-Oriented Design

RC Rate of Correctness

RPB Refused Parent Bequest

SAP Stable Abstractions Principle

SC Schizophrenic Code

SCAM Source Code Analysis and Manipulation

SD Sibling Duplication

SDLC Software Development Life-cycle

SDMPC Standard Deviation of Methods Per Class

SS Shotgun Surgery

TB Tradition Breaker

TCs Test Cases

TNTS Total Number of Test Samples

TQI Total Quality Index

TSE Transactions on Software Engineering

UML Unified Modeling Language

ABSTRACT

Software development is a messy process filled with an assortment of widely varying practices, procedures, and activities. Software refactoring, the process by which code is restructured without changing its external behavior, is one of many such activities squeezed in to tight deadlines and frantic work schedules. Refactoring is often discussed as an isolated process, yet developers most often perform floss-refactoring where refactoring operations are performed along-side some other software development activity, such as the addition of a new feature or to fix a bug, in order to support the work for that activity. This has created a niche for automated tools to assist developers with these time-intensive tasks and the high cognitive cost associated with the practice of constant task-switching.

The various automated tools created to support refactoring currently rely on combinations of quality metrics and detected code anti-patterns, often called code smells or code anomalies, to find opportunities to improve the code. Yet, the tools do not use more task-specific knowledge that exists to further improve both the refactoring recommendations and the associated tasks. Additionally, these automated tools frequently produce an overwhelming number of refactoring recommendations which take the form of multiple solutions which are themselves comprised of hundreds of individual refactoring operations to be performed on the software project. The overwhelming nature of these recommendation options and competing quality objectives makes refactoring code quite challenging. To further add to this already onerous effort, these refactoring recommendations are presented to developers with little concern for how these refactorings are linked together or need to be applied to the code

creating a truly herculean task.

To address these challenges, we present the following contributions:

1. We designed a simultasking[1], search-based algorithm that unifies the software development tasks of refactoring and regression testing which are currently treated in isolation, yet inherently linked. Developers frequently switch between these two activities, using regression testing to identify unwanted behavior changes introduced while refactoring, and applying refactorings on identified buggy code fragments. This overlap between both developer activity and engineering material, the source code, makes it an ideal target for potential knowledge transfer. The salient feature of the algorithm therefore is a unified and generic solution representation scheme for both problems which serves as a common platform for knowledge transfer between them. We implemented and evaluated the simultasking approach on six open-source systems and one industrial project. Our study features quantitative and qualitative analysis performed with professional and student developers. The results show that our simultasking approach provides advantages over current state-of-the-art, mono-task techniques which treat refactoring and regression testing separately.

2. We developed a theory for reasoning about collections of refactorings through defining an ordering dependency relation among refactorings and organizing collections of refactorings as a set of refactoring graphs. We then created an algorithm, based on our theory, to identify refactoring dependencies and we further illustrate these concepts with a web-tool for visualizing such refactoring dependencies and refactoring graphs. Our validation results demonstrate that 43% of the 1,457,873 recommended refactorings from 9,595 open-source projects that we studied are part of dependent refactoring graphs. Furthermore, refac-

[1]Simultaneous Tasking: Executing two or more tasks at the same time without weakening of any of the tasks.

torings are not only commonly involved in dependent relations, but also when applied dependent refactoring graphs improve all of the quality attribute metrics in our experiments more than individual refactorings.

3. We investigated high-level architecture anti-patterns and their correlation or lack-thereof of with code smells as we noted that software quality can be significantly impacted by architecture deterioration. This decay commonly results in unexpected side effects, an inability to support new features, and an overall reduction in software performance. Currently, there is limited knowledge of when code smells lead to architectural problems, or how the latter can cause code smells to propagate. Determining which anti-pattern should be refactored and how is never a pure technical problem in practice. High-level refactoring decisions have to consider trade-offs between code quality, available resources, project schedule, time-to-market, and management support. However, most existing anti-pattern detection tools focus on the code-level, and thus generate hundreds of code anomalies per project without linking them to architecture anti-patterns or indicating to developers where to start refactoring. We analyzed the occurrences of 3 types of anti-patterns and 16 code smells on 113 projects to determine: (1) What properties and types of code smells can serve as indicators of architectural problems? (2) Are there common anti-patterns? (3) Which types of high-level architecture anti-patterns are correlated with code smells, and under what circumstances? The answers to these questions will help developers understand where abstraction is needed, where refactoring is needed, and to what extent code smells can be prioritized and classified. Thus, we formulated the problem as a multi-class classification task and applied Random Forest along with a 5-fold, cross validation, Grid Search. We also conducted a survey with 34 developers to manually check the ranking and classification of code smells based on their impact and severity on architecture quality.

CHAPTER I

Introduction

1.1 Research Context

In 1999, Stephen R. Schach wrote in his textbook, *Software Engineering*, that software maintenance and evolution accounted for 67% of the total Software Development Life-cycle (SDLC) costs of a software project [1]. Twenty-two years later in 2021, a software quality consulting firm, Galorath[1], estimated that number to be typically around 75% of the total cost of ownership where 50% was consumed for corrective, adaptive, and perfective maintenance while the remaining 50% was due to the costs of software enhancements [2]. While the drastic increase in maintenance and evolution costs are multi-factorial and outside the scope of this thesis, one reason is the sheer number of tasks that encompass this phase of the SDLC. These various tasks include: fixing software bugs, migration, software refactoring, software testing, software change management, and version control among many others [1, 3]. When one looks at the the long list of the various tasks involved in software maintenance and evolution it can easily be seen how daunting it is to accomplish all of these tasks with high quality under the often intractable production deadlines software developers find themselves facing every day.

From 2000 to 2010, Toyota[2] automobiles suffered from strange cases of unintended

[1] https://galorath.com/
[2] https://www.toyota.com/

acceleration which resulted in the deaths of 89 people and 57 injuries over the decade [4]. The resulting investigation by the United States Department of Transportation National Highway Traffic Safety Administration (NHTSA) and the National Aeronautics and Space Administration (NASA) eventually concluded that excessive use of global variables contributed to the creation of Spaghetti Code [4]. Spaghetti Code [5] is a well-known anti-pattern in software development where the flow or logic of the code has become disorganized and usually incomprehensible to follow just as the noodles of spaghetti wind through each other in to a messy (even if tasty) clump on a plate. An anti-pattern is a common, but ultimately poor solution to a common issue in software development [3].

One of the identified root contributors to the cause of the Spaghetti Code as identified by an expert witness in the trial, Philip Koopman from Carnegie Mellon University, was excessive use of global variables. Ideally there should be none in the code, but the actual number present was between 9,000-11,500 representing approximately 82% of the total variables in the code [4]. Additionally as stated by Koopman, the throttle code was found to have a cyclomatic complexity metric score of 146 whereas a score of 50 would represent code that was not testable which further demonstrated the presence of the Spaghetti Code anti-pattern [4]. This amongst other problems with the code resulted in a settlement for $1.6 billion dollars in damages from a class action lawsuit against Toyota [4].

As can be see in the Toyota case study, software quality issues can result in significant and dire consequences not just in dollars, but also human lives. To further highlight this point, consider the field of medical device manufacturing where recalls have significantly increased since 2018 with the leading contributor to the recalls being caused by various software quality issues [6]. This begs the question: how then can companies and their developers deal with such staggering software quality issues?

Software refactoring, one of the many tasks of the SDLC phase of software main-

tenance and evolution, is a critical process which is able to improve software quality as well as helping to enable other maintenance tasks such as defect correction and feature addition. Software refactoring is the process of changing the internal structure of the software without changing its external behavior [7]. The field of refactoring was pioneered by Ralph Johnson and David Notkin separately in the late 1980s [8]. However, it was Martin Fowler who most recently rejuvenated and popularized refactoring in 1999 [7].

Anti-patterns such as Spaghetti Code in addition to other quality metrics such as the Quality Model for Object-Oriented Design (QMOOD) can provide indications of problem areas in the code that need restructuring and are thus good candidates for refactoring [7, 9, 10, 11]. Software refactoring in its purest form is performed by applying one of the more than sixty, small, atomic operations to code to accomplish some pre-defined restructuring of the code [7, 12, 13]. For example, the Move Field refactoring operation will move a single field from class 1 to class 2. However, the way researches write about software refactoring is often at odds with how refactoring is actually performed by software developers. Most commonly, developers perform floss-refactoring which is where they refactor in order to accomplish some other goal such as to help fix a bug and therefore do not perform refactoring as a distinct, atomic operation although we can still detect the results of the isolated refactoring as if it was executed on the code as a distinct operation [13].

1.2 Problem Statement

Currently, there are numerous manual and automated software refactoring tools [14, 15, 16, 17, 18, 19]. Several of these tools[3] exist as either integrated features or as plug-ins in development environments. Yet, they do not support developers

[3]https://visualstudio.microsoft.com/, https://www.eclipse.org/, https://www.jetbrains.com/idea/, https://www.sonarqube.org/

in switching between tasks and treat refactoring as another, separate activity that developers can use at their disposal when working on their project code. Further, there is no transfer of knowledge between tasks even though the various maintenance tasks overlap with software refactoring such as correcting defects, adding features, or testing.

Software refactoring tools, particularly automated tools, frequently generate an overwhelming number of refactoring operation recommendations. A tool might produce multiple solutions each with hundreds to thousands of refactorings. As developers are already facing a time crunch, it is little surprise that developers do not favor using the current tools to support refactoring [13]. It is hard to trust recommendations when one is unable to evaluate each suggestion on the code. Further, there are hidden dependencies between refactorings in the recommended list output from the various tools. As developers may only choose to implement some of the refactorings in the recommendation list, the act of picking and choosing further complicates applying the refactorings as a refactoring may rely on another refactoring.

In terms of software refactoring, a popular target to find code that needs to be refactored are anti-patterns. While there is a good understanding and collection of well-known anti-patterns at the code-level, there is currently a poor understanding of how these code-level anti-patterns impact the architecture of a software project. An understanding of which code-level anti-patterns lead to architecture issues and if refactoring the underlying code-level anti-patterns resolves higher-level problems in a project would lead to creating better software refactoring tools that are able to more quickly find the most severe problems that are contributing to larger issues and generate more useful recommendations for resolving those issues.

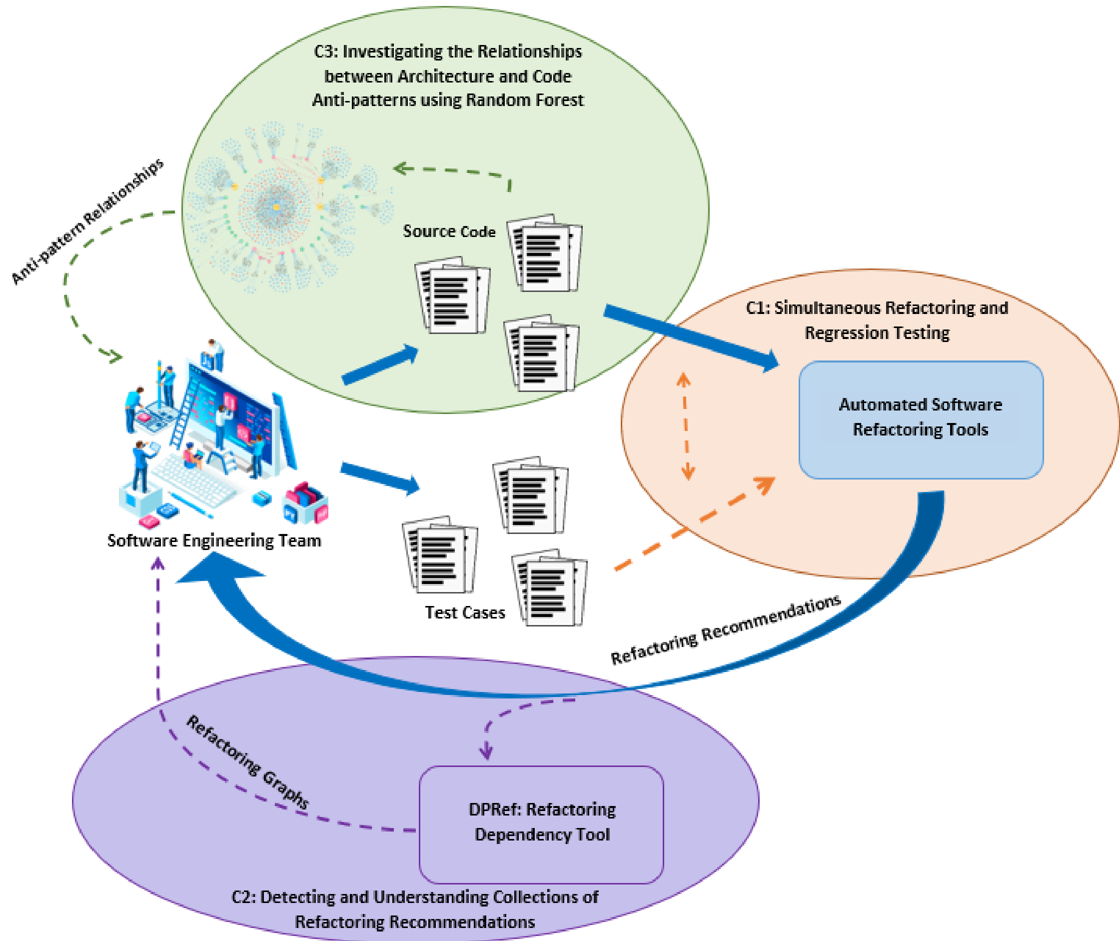

Figure 1.1: Overview of the contributions of this thesis.

1.3 Contributions

To address the issues mentioned in Subsection 1.2, we offer the following solutions organized into the three contributions as shown in Fig. 1.1 [4].

1.3.1 Contribution 1: Simultaneous Refactoring and Regression Testing

Current software development processes are expensive, laborious, and error prone [20], with software developers frequently switching between various tasks [21]. For instance, they commonly shift back and forth between fixing critical bugs and refactoring the related code. While such a multitasking behavior can optimize the allocation

[4]Free clip art used in the creation of Fig. 1.1 provided by clipartkey.com and clipart-library.com.

of human resources, its cognitive cost can be high and exacerbated by the lack of automated support tools.

Various techniques have been proposed to automate refactoring, including the detection of refactoring opportunities such as anti-patterns [9, 22, 23] and refactoring strategies to fix them [24, 25]. Most of the approaches rely on static and dynamic code analysis to identify refactoring opportunities that maximize quality improvement. Similarly, many techniques address regression testing to select or prioritize test cases that maximize the coverage of code changes [26, 27]. In contribution 1, we focus on test case selection as a regression testing technique. While refactoring and regression testing might look like two disjointed activities, they are implicitly connected. There are at least three reasons for this: (1) Since refactoring is not supposed to change system behavior, developers should perform regression testing on the refactored code to ensure that no bugs were introduced; (2) both activities go through a first step of identifying the code component(s) to refactor and/or to test; (3) developers are usually interested in refactoring buggy code to improve its maintainability; therefore, they aim to reduce the likelihood of introducing new bugs while working on it [9, 28].

Only a few studies are proposed in the literature to use such implicit connections [28, 29, 30]. The main goal of these regression testing approaches is to reduce the number of test cases to run by differentiating between regular code changes and refactorings. Thus, the regression testing process will not run test cases on methods where only refactorings, *e.g.* rename method, were applied since they are assuming that the applied refactorings will not change the behavior. However, none of these techniques addressed the possibility of treating refactoring and regression testing simultaneously. As a result, refactorings may not be recommended for relevant buggy code components and selected test cases may not cover relevant poor-quality classes.

Contribution 1 remedies that gap by proposing a unified and generic solution representation scheme for both refactoring and regression testing to serve as common

platform for knowledge transfer between them. The resulting simultasking search allows for the involvement of multiple tasks at once with each one contributing a unique factor influencing the evolution of a *single* population of individuals. In this contribution, each of the problems of refactoring and regression testing corresponds to an optimization task, and each one contributes to our simultasking model with its own objectives which are intended to be complementary to those of the other problem.

We implemented and evaluated the proposed simultasking approach on six open-source systems and one industrial project by: (1) assessing the quality improvement brought by the refactorings recommended by our approach as compared to state-of-the-art refactoring tools [11, 31]; (2) asking 25 developers to evaluate the meaningfulness of the refactorings generated by our approach and by a competitive mono-task approach [11] using the same fitness functions we employ to identify refactorings (thus, having as the only difference with our approach the *mono-task vs multi-task focus*); and (3) assessing the effectiveness of the test case selection output of our approach as compared to baseline techniques composed by a bi-objective greedy algorithm [32] and a mono-task testing approach using the same fitness functions for testing of simultasking. We found based on manual evaluation and surveys with developers that, on average, the simultasking approach performs better than the state-of-the-art techniques, with 89% of recommended refactorings classified as meaningful by developers and real bugs identified via regression testing in an industrial setting.

The better performance of our simultasking approach in comparison to mono-task algorithms may be explained by two observations from our experiments: (1) relevant refactoring opportunities can be recommended in parts of the code that are reached not thanks to the fitness functions dedicated to the refactoring task (removing code anti-patterns), but due to the need for covering them in the test case selection process; and (2) bugs can be discovered because of the knowledge transferred from

the refactoring task that promotes the selection of test cases aimed at covering the refactored components. The 2-stage approach (refactoring then regression testing) did not outperform the simultasking approach. In fact, simultasking enables the knowledge transfer in two ways during the optimization process while the 2-stage approach is limited to only: (1) one way of knowledge transfer (from refactoring to regression testing), and (2) the final refactoring solution rather than the variety of refactoring solutions generated during the optimization process.

This work has been accepted and presented at the Institute of Electrical and Electronics Engineers (IEEE) 19th International Working Conference on Source Code Analysis and Manipulation (SCAM) [33]

1.3.2 Contribution 2: Detecting and Understanding Collections of Refactoring Recommendations

Even for the most competent organizations, building and maintaining high performing software applications with high-quality code is a challenging and expensive endeavor [34]. Working in a fast-paced environment that demands frequent releases across several products and deployment environments, developers are often forced to compromise high-quality standards in favor of meeting deadlines [35]. Lack of robust automated tools results in buggy and poor quality software that causes financial losses, high maintenance costs, increased fault-proneness, and delayed or canceled projects [36]. Software refactoring is widely recognized as an effective approach for maintaining high quality software by restructuring existing code without changing its external behavior [37].

Resolving code smells and broader code quality improvements across a project often requires applying multiple refactorings, which has lead to the creation of tools that recommend extensive collections of refactorings to developers [38]. These collections are presented to users as a sequence with an implied strict ordering among

the refactorings. In practice, many of these orderings are not significant while others retain significant meaning. Yet, these tools treat each refactoring in the sequence in isolation. For instance, Cinnéide *et al.* [39] investigate the impact of only individual refactorings on quality attribute metrics, such as using the Move Method refactoring operation to reduce the coupling of a class, without studying the impact of a sequence of refactorings. Bibiano *et al.* [40] evaluates the relationships between refactoring types (*e.g.* Move Class, Extract Interface) and code smells. However, the study is based on the assumption that refactorings are only related if applied to the same code location, *e.g.* a class. However, most refactoring types modify multiple code fragments *e.g.* Move Method modifies two classes, the source and target of the move operation.

Further, a common practice among developers is to manually apply these generated sequence of refactorings from tools [38, 40, 41]. This is additionally complicated by the practice of developers picking and choosing which, if any, of the recommendations they will implement in their code. Figure 1.2 shows an example of the refactoring recommendations of JDeodorant [42]. Similarly to other existing automated refactoring tools, the dependencies between the refactorings are not revealed. Thus, this leaves the challenging task of interpreting the sequence of refactoring recommendations to the developers who lack a theory to tell the difference between when the orderings are not significant and when they retain significant meaning. This makes refactoring recommendations generated by tools harder to understand than is necessary especially as the critical refactoring dependency information is not integrated in refactoring recommendation tools and highlights a gap in the existing research.

To close this gap, in contribution 2 we describe a theory for reasoning about collections of refactorings through a definition of ordering dependencies among refactorings and an algorithm for identifying these dependencies. We aim to improve the accuracy of refactoring recommendation tools by detecting refactoring dependencies,

Prob... Java... Decl... Search Con... Debug Call ... Dupli...

Refactoring Type	Source Entity	Target Class
Move Method	SmellDetector.SourceModel.Resolver::...	SmellDetector.SourceMod...
Move Method	SmellDetector.SourceModel.Resolver::...	SmellDetector.SourceMod...
Move Method	SmellDetector.smells.designSmells.Ab...	SmellDetector.metrics.Typ...
Move Method	SmellDetector.smells.ThresholdsParse...	SmellDetector.smells.Thres...
Move Method	SmellDetector.smells.implementation...	SmellDetector.SourceMod...
Move Method	SmellDetector.SourceModel.Resolver::...	SmellDetector.SourceMod...

Figure 1.2: Example output of a refactoring recommendation tool: JDeodorant.

which allows the developers to efficiently interact with such refactoring recommendation tools. Additionally, we define refactoring recommendations as sets of refactoring graphs rather than as refactoring sequences. We illustrate these concepts with a web-tool for visualizing refactoring dependencies and refactoring graphs.

Refactorings, when formalized, have clear pre-conditions defining the circumstances in which they can be applied and post-conditions defining the effects of applying them. For instance, one of the pre-conditions of a Move Method refactoring is that the method exists in the class from which it will be moved and one of its post-conditions is that the method must exist in the target class afterward. Therefore, a refactoring dependency exists when any post-condition of one refactoring matches any pre-condition of another refactoring, *e.g.* the method exists in the relevant class. These linked refactorings then can be organized into groups based upon their dependencies. We represent these groups as directed acyclic graphs, where the nodes are the refactorings and the edges are the dependencies. This approach offers three main benefits: (1) developers can quickly and intuitively understand the dependencies among refactorings that constrain recommendations, *e.g.* which refactorings must be done together; (2) developers can more easily compare recommendations by focusing on essential, rather than accidental, differences; and (3) tool builders can easily inte-

grate new features to detect invalid refactorings and improve their recommendation algorithms.

We validated our proposed theory based on 1,457,873 refactorings recommended for 9,595 Java projects publicly available on GitHub[5]. We considered 14 types of refactorings that are most commonly used in practice [14, 43]. We also developed a web-tool, DPRef, that implements the proposed ordering dependency detection algorithm. It transforms a refactoring sequence recommended by existing refactoring tools [42, 44] into refactoring graphs based on the detected dependencies. We conducted experiments to evaluate the correctness of the detected dependencies, discover what portion of refactorings in recommendations are actually dependent rather than independent, and estimate the potential impact of dependent refactorings on several quality attribute metrics. Finally, we conducted a human validation study with 27 developers to manually evaluate the correctness of the detected dependencies and their relevance.

Our implemented algorithm achieved 100% in correctly detecting all dependencies between refactorings and identifying invalid refactorings. Furthermore, our findings demonstrate that 43% of the 1,457,873 recommended refactorings are part of dependent refactoring graphs. This finding confirms that refactorings are commonly involved in dependent relations and cannot be applied truly independently. Furthermore, dependent refactorings improve all six QMOOD quality attribute metrics [45] in our experiments better than independent refactorings. The manual validation of the refactorings by 27 developers shows that all the identified dependencies are correct for a sample of 233 refactorings after applying them directly on the code of 61 open source projects based on the order proposed by DPRef. The post-study survey with the developers confirmed the relevance of detecting the dependencies to help them understand the sequence of recommended refactorings.

[5]https://github.com/

We also provide a **replication package**[6] that includes the refactoring dependency detection tool and necessary data for our large scale validation. The replication package will enable researchers and tool builders to integrate the refactoring dependency feature into existing refactoring recommendation and detection tools and further investigate the relationships among refactorings. This work is currently under review at the IEEE Journal Transactions on Software Engineering (TSE) and our web-tool is available on-line[7] [46].

1.3.3 Contribution 3: Investigating the Relationships between Architecture and Code Anti-patterns Using Random Forest and Grid Search

Architecture decay causes significant consequences for the quality of a software project. This decay can cause numerous, unexpected side effects, an inability to support new features, and an overall reduction in software performance and reliability among many other negative consequences [47, 48]. As projects evolve, developers frequently postpone improving the quality of their systems in the rush to deliver a new release until a crisis happens [49]. When that postponement occurs, the project accrues a technical debt which is the cost of choosing a simpler and therefore quicker approach instead of more thorough, quality-based, and therefore more time-consuming approach [3]. This accrual of technical debt is the foundation of architectural decay and eventually a terminally broken system architecture with significant quality loss [50, 51, 52]. It is critical to identify and fix these quality issues as early as possible at different levels of abstraction.

One of the main challenges is that even though quality issues often occur at multiple levels, existing studies focus mainly on the code-level such as code smells [22, 53, 54, 55]. However, recent studies [14, 56] show that 92% of interviewed developers emphasized that they had never performed architecture refactoring in isolation from

[6]https://sites.google.com/view/refactoringdependency
[7]https://iselab-dpref.herokuapp.com

code refactoring. Currently, it is not clear if, when, and how anti-patterns at different levels influence each other.

While detecting the co-occurrence of code anomalies is well supported by different tools [11, 13, 57, 58, 59, 60, 61, 62, 63, 64], they do not specify where to start or how they depend on each other. A large number of code smells are detected without grouping them together based on their possible relationships with architecture anti-patterns. Some recent studies [22, 53, 54, 55, 65] investigated the relationships between code smells and several quality aspects of the software (*e.g.* maintainability and comprehension), and conceptually characterized interactions between code anomalies. However, the relationships between code anomalies and architecture anti-patterns, such as cyclic dependencies, are rarely investigated. Empirical studies mainly address how individual occurrences of code anomalies emerge during software evolution [66] or impact quality attributes [48, 67, 68, 69].

There is limited knowledge of if and when code anomalies lead to architectural problems, or how the latter can cause code anomalies to propagate. Given the existence of multiple names of software problems at different levels, such as anti-patterns [70], bad smells [7, 71], hotspot pattern [72], and technical debt [51, 73, 74], in this contribution, we uniformly label architecture problems as *anti-patterns* and code-level quality issues as *code anomalies*.

The objectives of this contribution are to: (1) reveal the correlation or lack of correlation between architecture anti-patterns and code anomalies, (2) determine the impacts of anti-patterns and make them explicit to help programmers select the ones that need to be refactored, and (3) integrate these learned relationships into a refactoring opportunities detection tool. This knowledge will guide future research on avoiding, detecting, and fixing them with (semi) automated support. We formulated the problem of identifying relationships between architecture anti-patterns and code anomalies as a multi-class classification task and applied Random Forest along

with 5-fold cross validation Grid Search to extract the best association rules. In our adaptation, the code anomalies are the features of the predictive model (*i.e.*, internal nodes of the decision tree) and architecture-level anti-patterns are the classes to be predicted (*i.e.*, leaves in the decision tree).

We used a set of 16 code anomalies and 3 architecture-level anti-patterns in 113 open-source Java projects for generating the association rules using existing tools for code anomalies and architecture anti-pattern detection. We selected these tools due to the low false positive results when evaluated on open-source projects. The accuracy of the best tree is 87%. We have achieved the highest precision of 91%, highest recall of 90% and highest F_1-score of 75%. We have conducted a survey with 34 developers to manually check the ranking and classification of code anomalies based on their impact and severity on the architectures' quality. The participants manually confirmed the accuracy of the generated association rules and the benefits of grouping code anomalies based on their impact on the architecture quality after they used our tool with the integrated association rules.

This work is currently under review at the open-access journal, IEEE Access [75].

1.4 Publication List

- Jeffrey J. Yackley, Marouane Kessentini, Gabriele Bavota, Vahid Alizadeh, and Bruce R. Maxim, "Simultaneous Refactoring and Regression Testing," *In Proceedings of the 2019 IEEE 19th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, Cleveland, OH, USA, 30 September-1 October 2019, pp.216-227, DOI: 10.1109/SCAM.2019.00032. Acceptance Rate: 24%

- Thiago Ferreira, James Ivers, Jeffrey J. Yackley, Marouane Kessentini, Ipek Ozkaya, and Khouloud Gaaloul, "Dependent or Not: Detecting and Under-

standing Collections of Refactorings", Under review at *IEEE Transaction on Software Engineering (TSE)*, 2022. Impact Factor: 7.77

- Jeffrey J. Yackley, Somayeh Molaei, Marouane Kessentini, and Bruce R. Maxim, "Investigating the Relationships between Architecture and Code Anti-patterns Using Random Forest and Grid Search," Under review at *IEEE Access*, 2022. Impact Factor: 3.37

1.5 Organization of the Dissertation

This thesis is organized as follows: Chapter II introduces the current state of the art including the background and related works necessary to contextualize and understand the contributions to the thesis and their place in the wider body of research. Chapter III describes the first contribution on unifying two, related software maintenance tasks, software refactoring and regression testing, through a simultasking algorithm to simultaneously improve both tasks. Chapter IV chronicles the second contribution on a theory of refactoring ordering dependencies, an algorithm to detect them, and a web-tool to organize refactorings into refactoring graphs to highlight the hidden dependencies between refactorings in a recommended solution. Chapter V highlights the third contribution that investigates the relationships between architecture-level and code-level anti-patterns using machine learning. Lastly in Chapter VI, a summary of the work in this thesis is presented with a brief discussion of future research directions.

CHAPTER II

State of the Art

2.1 Introduction

In this chapter, we cover the background information to understand and contextualize the work in the contributions of this thesis as well as providing an overview of the closest related work from existing studies. The chapter organizes this discussion in to the following topics: Subsection 2.2 on software refactoring, Subsection 2.3 on automated refactoring tools, Subsection 2.4 on refactoring dependencies, Subsection 2.5 on regression testing, and Subsection 2.6 on anti-patterns.

2.2 Software Refactoring

Software refactoring is defined as the process by which the internal structure of code is modified without changing the external behavior of the code [7]. Catalogs like Fowler's [7] have identified many types of refactorings, *e.g.* Move Method, Extract Class, Pull Up Field, each of which is a semantics preserving code transformation that improves code structure. Developers routinely apply such refactorings in their day-to-day work with modern development environments providing limited support for applying selected refactorings as directed by a developer.

2.3 Automated Software Refactoring Recommendation Tools

There has been both industry and research interest in developing automated and semi-automated refactoring tools to support developers [76]. One representative example is JDeodorant, the tool proposed by Tsantalis and Chatzigeorgiou [42]. JDeodorant and similar recommendation tools [14, 15, 16, 17, 18, 19] generate recommendations as sequences of refactoring instances. The experiments described in contribution 2 take refactoring recommendations as input. Thus, our discussion in this section focuses on this category of studies. We point the interested reader to the survey by Bavota *et al.* [77] for an overview of approaches supporting code refactoring recommendations. In another refactoring recommendation tool, O'Keeffe and Cinnéide [78] formulate refactoring tasks as a multi-objective search problem to generate alternative designs by applying a sequence of refactoring operations. Such a search is guided by a quality evaluation function based on eleven object-oriented design metrics that reflect refactoring goals. Harman and Tratt [17] were the first to introduce the concept of Pareto optimality to search-based refactoring. They used it to combine two metrics, namely Coupling Between Objects (CBO) and Standard Deviation of Methods Per Class (SDMPC), into a fitness function and showed its superior performance as compared to a mono-objective technique [17]. The two aforementioned studies [17, 78] paved the way for several search-based approaches aimed at recommending refactorings [15, 44, 79, 80, 81, 82].

A representative example of these search-based refactoring techniques is the work by Ouni *et al.* [44], who propose a multi-criteria code refactoring approach aimed at optimizing contrasting objectives: (1) minimizing the number of code smells; (2) minimizing the refactoring cost (*i.e.*, the number of recommended refactorings); (3) preserving the design semantics (meaning considering textual information embedded in code identifiers and comments in the refactoring recommendation); and (4) maximizing the consistency with code changes performed over the system's change

history. In contribution 2, we use the refactoring recommendations generated by this tool based on (1) its superior performance compared to the state of the art [44]; (2) the large number of supported refactoring types, and (3) its being publicly available. Contribution 2 of this thesis is not generating refactoring recommendations. Any refactoring recommendation approach can be used to generate the refactorings, the input of our proposed approach, if they support some or all of the refactoring types summarized in Table 4.1.

2.4 Software Refactoring Dependencies

Chavez *et al.* [83] investigated how refactoring types affect five quality attributes based on the version history of twenty-three open-source projects. They found that 94% of refactorings are applied to code with at least one low quality attribute value, with 65% of refactorings improving attributes and 35% of all refactorings being neutral on the system. Similarly, Cinnéide *et al.* [39] studied the impact of individual refactorings on quality attributes, such as using the Move Method refactoring operation to reduce the coupling of a class. None of these studies considered the impact of a sequence of refactorings on the quality attributes.

Murphy-Hill *et al.* [41] investigated refactoring tool usage through both sampling developers' code and manually checking if their refactorings were performed with tool support and looked at 240,000 tool-assisted refactorings to find assumptions on how developers, in general, refactor code. They ultimately concluded that refactoring tools are rarely used by developers in practice with 90% of refactorings being performed manually, and that 40% of refactorings occur in batches.

Bibiano *et al.* [40] analyzed batch refactoring characteristics and their effects on code smells in open and closed-source projects and concluded that 57% of batches/ patterns are simple compositions of only two types of refactoring operations. They highlight the lack of tool support to automatically detect refactoring dependencies as

a barrier. However, this study is based on the assumption that refactorings are only related if applied to the same code location, which often is not the case for types of refactorings that modify multiple code fragments.

Mens *et al.* [84, 85] define and detect mutual exclusions, sequential dependencies, and asymmetric conflicts between refactorings. These studies analyze dependencies at the model-level working with the Unified Modeling Language (UML) and they use graph transformation techniques to detect invalid refactorings. The detection of conflicts between refactorings at the model-level (UML) is based on a set of rules, a matrix where the lines and columns are model refactoring types, that are manually defined. The type of refactorings is different and simplified compared to code-level ones. Furthermore, the authors were looking for mutually exclusive UML refactorings rather than detecting dependencies.

Liu *et al.* [86, 87] propose a conflict-aware scheduling approach, which schedules refactorings according to the conflict matrix of refactorings and effects of each individual refactoring using a multi-objective optimisation model. In this work, the authors focused on identifying the best schedule to apply refactorings where the conflicts are defined based on which code smells are to be fixed. Thus, the same refactorings fixing different code smells or applied in the same locations are grouped together. The notion of dependencies is defined in contribution 2 in a different way than Liu *et al.* [86] where they are more about the conflicts between the refactoring themselves and not their goals.

Sousa *et al.* [88] identify and analyze composite refactorings within and across commits from the commit history of 48 GitHub software projects. The concept of defining dependencies in this work is different than contribution 2 where the dependency is about grouping the refactorings applied within the same commits/locations.

Overall, existing studies do not provide a rigorous definition of ordering dependencies among refactorings. They mainly define what might be better considering

similarity relations, such as a collection of refactorings that have similar effects, *e.g.* fixing a code smell, or similar context, *e.g.* applied by the same developer or to the same code location [89, 90].

2.5 Regression Testing

Pressman and Maxim define regression testing as "the process of verifying that software that was previously developed and tested still performs the same way after it has been changed" [3]. Regression testing often involves running incredibly large test suites with hundreds if not thousands of test cases depending on the size of the system being tested. This has lead to the development of many techniques which have been proposed with the goal of reducing the regression testing effort, mostly via test minimization/selection [26, 27, 32, 91, 92, 93, 94, 95, 96, 97, 98] and prioritization [32, 99, 100, 101, 102, 103, 104, 105, 106].

Yoo and Harman [32, 95] present a multi-objective approach supporting test case selection. They use different search algorithms to look for solutions aimed at optimizing two and three contrasting objectives. In the two-objective approach, they aim at maximizing statement coverage while minimizing the computational cost of test execution. Those are also the fitness functions that we optimize in our approach in contribution 1. In the three-objective version, they also try to maximize the past faults detection. Panichella *et al.* [27] built on top of these works showing how to improve the performance of multi-objective genetic algorithms by injecting diversity during the generation of the solutions.

Contribution 1 is complementary to the previously mentioned related works. Indeed, our goal is not to present the most effective search-based technique for test case selection, but rather to show the potential benefits of exploiting a simultasking search-based algorithm to solve simultaneously strongly related search problems such as refactoring and regression test case selection.

2.6 Software Anti-patterns

Palomba *et al.* defines anti-patterns as “poor solutions to recurring design problems" noting that “They occur in object-oriented systems when developers introduce these bad practices while designing and implementing their systems in the rush to deliver a new release" [107]. This means that anti-patterns are not bugs in the software since they implement the correct functionality. The implementation is simply flawed from a non-functional requirements perspective (*e.g.* bad design, performance, *etc.*).

In [108], Lin *et al.* present Refactoring Navigator, an interactive architectural refactoring recommendation tool. Their approach takes as input some code implementation as a starting point and a desired high-level design as the target before iterating refactoring recommendations to fix the detected anti-patterns while allowing the user to provide feedback on the refactoring recommendations [108]. Moha *et al.* [22] proposed DECOR, a method to specify and detect code anomalies.

Abbes *et al.* in [54] performed an empirical study to investigate whether the occurrence of two anti-patterns, Blob and Spaghetti code, actually affects the understandability of systems by developers during comprehension and maintenance tasks. Abbes *et al.* concluded that developers could work around code with one anti-pattern, but code with multiple anti-patterns significantly decreased a developer's performance [54]. Yamashita and Moonen [53] empirically investigated 12 code smells and analyzed their interactions in relation to software maintenance problems using a team of professional developers. They noticed a strong coupling effect between smelly code and co-located smells that affected maintainability [53]. With similar purpose, [55], quantified the affect of 12 code-level anomalies on the maintenance effort.

In [48], Oizumi *et al.* discuss that code anomalies have a cumulative effect to realize a design problem. They argue that each code-level anomaly alone may represent only a partial embodiment of a design problem and that groups, they call agglomerations, provide enough information to locate/predict design issues [48].

An analysis of the relationship between design flaws and software defects was proposed in [67]. D'Ambros *et al.* investigated this relationship by analyzing the frequency of design flaws in six open-source software systems and based on that analysis investigated the correlation of flaws with post-release defects. [66] conducted an empirical study by investigating the change history of 200 open-source projects looking to find evidence of when and how code smells are introduced in to the code. Tufano *et al.* conclude from their study four main lessons: (1) code elements are affected by smells since their origination, (2) code elements that become smelly as a result of maintenance and evolution have different metric trends from unaffected code elements, (3) refactoring introduced smells in 400 cases, and (4) high workload and deadline pressure are responsible for making developers more likely to introduce smells than a new developer to the code [66].

However, none of the above studies analyzed how code anomalies and architecture anti-patterns relate together and how this relationship can help developers to have a better understanding of which code anomalies are currently impacting architecture anti-patterns. To make our terminology for contribution 3 precise, we provide the following definitions for architecture anti-patterns and code anomalies based on the related work.

2.6.1 Architecture Anti-patterns

In this section, we provide the definitions for the three architecture anti-patterns that we investigated in contribution 3. We based our definitions off of the work of Mannan *et al.* [109] who refined definitions for some of the most common code smells which includes: Cyclic Dependencies, Stable Abstractions Principle (SAP) Breaker, and God Class. Our definitions abstract from the code-level to capture the behavior of the smell at the higher architecture-level where we consider groups of related classes called packages.

Cyclic Dependencies: There exists a cyclic dependency similarly defined by Mannan *et al.* for classes [109] between two or more packages in the system. For example, package A, is dependent upon package B which is dependent on package C which is dependent on package A leading to a cycle of dependencies.

SAP Breaker: The Stable Abstraction Breaker anti-pattern is a package which has an inverse relationship between its abstractness and its use by other packages following the stable abstraction principle [109]. Stated another way, the more one package is used by another package the more abstract the package should be.

God Package: Similar to the concept of a God Class [109] which aggregates the responsibilities of multiple classes, the God Package is a package which aggregates the responsibilities of multiple different packages. This results in an overly complex package that hoards behaviors and data that would be otherwise better organized in to separate packages.

2.6.2 Code Anomalies

As shown by Nucci *et al.* , code anomalies have been well studied in the literature with works focusing on: (1) their introduction, (2) evolution, (3) effect on program comprehension, and (4) the ability of developers to find and correct them [110]. Moreover, Nucci *et al.* also write of several code smell detection tools that have been proposed in the literature [110]. Different studies have been focused on

different subsets of code smells. One can find a survey on different code smells and their detection techniques in [111, 112]. In contribution 3, we detected a set of sixteen code-level smells, which we refer to as code anomalies throughout the contribution to avoid confusion with architecture anti-patterns, for 113 different Java open-source projects. The revised and simplified definitions of these code smells are summarized in Table 2.1 and are based on the work of Mannan *et al.* [109].

Table 2.1: A brief description of different code anomalies used in our study.

Code Anomalies	**Definition**
Blob Class (BC)	A large and complicated class that is challenging to maintain; highly probable it exhibits strong coupling.
Blob Operation (BO)	A long and overly complicated method which incorporates too much of the behavior of its class.
God Class (GC)	A large class that incorporates the behavior and data of multiple other classes; possess low cohesion.
Data Class (DC)	A class that consists primarily of data members without methods that meaningfully interact with the data; usually also lacks appropriate encapsulation of the data.
Schizophrenic Code (SC)	A class that models two or more different design abstractions that should be separate from each other.
Refused Parent Bequest (RPB)	A child class that fails to make use of the inherited members from its parent class.
Tradition Breaker (TB)	A child class that provides significant functionality that is unrelated to its parent class functionality.
Distorted Hierarchy (DH)	An inheritance hierarchy that is approximately six or more classes deep.
External Duplication (ED)	A module(s) of a system replicate(s) functionality already found in another unassociated module(s).
Sibling Duplication (SD)	Occurs when two or more classes at the same inheritance hierarchy level duplicate each other's behavior.
Internal Duplication (ID)	A class or module that duplicates parts of its own code.
Data Clumps (DCP)	Occurs when the same data is repetitively passed as individual parameters in a system instead of being encapsulated and passed as an object.
Intensive Coupling (IC)	Occurs when a method is tightly coupled to numerous other functionality throughout the system.
Message Chain (MC)	Occurs when an extended progression of method calls is used to retrieve or produce data.
Shotgun Surgery (SS)	Occurs when code throughout the system in numerous places must be altered to allow modification or addition of a feature.
Feature Envy (FE)	A method that requires the data members of other classes more so than its own class indicating the method has been coded in the incorrect location.

CHAPTER III

Simultaneous Refactoring and Regression Testing

In this chapter we present contribution 1: simultaneous refactoring and regression testing. In this contribution, we begin with Section 3.1 that demonstrates an authentic and concrete instance that supports the simultasking approach for refactoring and regression testing. In Section 3.2, we present an overview of our approach before covering the validation of the approach in Section 3.3. We discuss the threats to validity in Section 3.4. Finally, we conclude with a summary of our work and mention potential future research directions in Section 3.5.

3.1 A Motivating Example

To illustrate the motivation for this work, we present an example from the Aspect-J[1] project, an open-source, Aspect-Oriented Programming (AOP) extension to the Java programming language. For the releases between 03/2002 and 01/2014, Aspect-J had 593 bug reports documented in *Bugzilla*[2].

One of the Aspect-J classes exhibiting a high fault proneness is *AjcTask*, with 28 bug fixes over several releases. The analysis of the bug reports shows that seven of these bugs were introduced as the result of 63 refactorings performed by developers.

[1]https://www.eclipse.org/aspectj/
[2]https://www.bugzilla.org/

For example, bug reports 463472[3] and 4996393[4] discuss bugs introduced after applying *Rename Method* refactorings to the *AjcTask* class. We also found that the average number of design defects (code smells) affecting *AjcTask* was between 9 and 14 across the multiple releases that we analyzed. Those numbers were obtained by running a code smell detection tool [14] and manually validating the candidate smell instances it produced. By using the approach in Ouni *et al.* [11], we found four possible refactoring solutions to fix the identified code smells. Those solutions comprise different refactorings and require different sets of test cases for the regression testing of the refactored code. The shortest refactoring solution was a sequence of 16 refactorings, 37 test cases, and took 46 minutes for regression testing (running the test cases); the longest solution included 52 refactorings, 128 test cases, and took 232 minutes of run time. The running times were obtained with an execution platform featuring an Intel i7 and 8GB of RAM. This simple example shows that, as already observed in the literature: (1) refactoring may introduce bugs [113] and (2) code smells may relate to high class fault proneness [69, 114]. These observations motivate our approach for supporting refactoring and regression testing as a simultasking search problem.

3.2 Approach Overview

We begin by outlining the similarities between refactoring and regression testing. The first activity tries to find refactorings that *cover* detected design defects, whereas the second tries to find test cases to *cover* code changes. In both cases, *high problem coverage* is required to achieve good solutions. Furthermore, both the regression test cases and refactorings need to be selected. For refactoring, we seek to select refactorings to maximize the number of design defects that are fixed while minimizing the refactoring cost (minimizing the number of refactoring operations). For regression

[3] https://bugs.eclipse.org/bugs/show_bug.cgi?id=46347
[4] https://bugs.eclipse.org/bugs/show_bug.cgi?id=499639

testing, we seek to select the test cases to maximize the number of detected bugs while minimizing the testing effort (minimizing the number of test cases to execute). Finally, the selection of a refactoring solution by developers may depend on the number of test cases needed to test it, since they seek to improve code quality while minimizing the risk of introducing bugs and testing cost. Thus, the decision-making process needs to consider both problems simultaneously, and the knowledge to solve one problem may be useful to solve the other.

The new search approach involves multiple tasks synergistically contributing their individual influences to the evolution of a population of individuals, based on shared interests.

3.2.1 Simultasking

There are two key insights behind evolutionary multitasking. Firstly, given a population of solutions in a unified search space (encompassing more than a single task), it is intuitively more probable for a randomly generated or genetically modified solution to be competent for at least one task. In our case, the random selection of methods from the source code can be relevant for testing or refactoring or both. Secondly, as the search progresses, it may happen that the high performing genetic building blocks (or schema) created for a particular task turn out to also be useful for another task. Thus, the test cases selection block of the unified solution may identify relevant buggy methods that can be also transferred to the second refactoring block. In this case, the simultasking algorithm can converge faster to find good refactoring recommendations when the regression testing task is solved. As argued before, the proposed simultasking approach allows such mutually beneficial genetic building blocks to be exchanged spontaneously, thereby leading to accelerated convergence towards the solution(s).

To compare the candidate solutions during simultasking, we first define the prop-

erties to describe every individual p_i in a population P, with $i \in \{1,2, \ldots, |P|\}$. As detailed in the next section, every individual is encoded into a unified space Y that encompasses the individual task spaces $X1$,$X2$,...,X_K, and can be translated into a task-specific solution with respect to any of the K optimization tasks. We are limited to two tasks ($K = 2$).

Definition 1 (Factorial Rank): The factorial rank r_i^j of p_i for task T_j is the p_i's index in the list of population members sorted in decreasing order of preference w.r.t. T_j.

Definition 2 (Skill Factor): The skill factor τ_i of p_i is the one task, among all other tasks in a K-factorial environment, with which the individual is associated. If p_i is evaluated for all tasks, then $\tau_i = argmin_j \ \{r_i^j\}$, where $j \in \{1,2,\ldots,K\}$.

Definition 3 (Scalar Fitness): The scalar fitness of p_i in the simultasking environment is given by $\phi_i = 1/r^i_{\tau_i}$. P_t is the current population at generation t, and C_t is the generated offspring of P_t.

The algorithm largely follows a standard evolutionary procedure based on the Non-dominated Sorting Genetic Algorithm-II (NSGA-II) [115], with the addition of a new entity, the skill factor (τ). The NSGA-II component is mainly used to find a trade-off between conflicting objectives. The skill factor is seen as the computational representation of an individual's cultural background. The skill factor is passed on

through generations by the simple process of *imitation*, which is perhaps the most prevalent form of vertical cultural transmission. In other words, an offspring randomly copies the skill factor of any one of its parents, with the scope for fruitful genetic exchange occurring when parents with different skill factors undergo genetic recombination to create an offspring. For the case of our tasks, one solution can have a high skill factor for the refactoring task while another solution can have a high skill factor for the regression testing task. Thus, the genetic recombination will take the best block from each solution and combine them to generate better solutions solving both tasks. We will generate new solutions either combining both blocks of these two solutions (taking the best from every solution) or transferring the knowledge between the blocks of the same solution (transferring the best from the block having the higher skill factor). For instance, the methods recommended for refactoring located in a block with high refactoring skill factor will be transferred to the block of regression testing of the same solution to be covered by test cases. While the foundations of our approach are based on the multitasking algorithm, we introduced several adaptations to the original version based on the characteristics of the addressed regression testing and refactoring problems as described in the next section.

3.2.2 Adaptation

The input of our approach consists of the source code of the evaluated system and its previous commits (code changes), a set of code smell detection rules [14], and a set of test cases from previous releases. The output is a set of Pareto front solutions. Each solution is composed from a set of recommended refactorings and another of selected test cases to execute. Four objectives are used to evaluate solutions: (1) minimize the number of code smells, (2) minimize the number of recommended refactorings (the refactoring cost), (3) maximize the code coverage, and (4) minimize the number of selected test cases and the run-time of the execution order of selected test cases.

While we have four objectives, we have still selected NSGA-II since the knowledge transfer between the blocks of the solutions using the skill factor helped to reduce the size of the Pareto front comparing to a regular NSGA-II algorithm. Thus, we did not observe a larger number of solutions in the Pareto front despite the use of four objectives. The supported refactoring operations are: Extract Class, Extract Super Class, Extract Sub Class, Encapsulate Field, Decrease/Increase Field Security, Decrease/Increase Method Security, Extract Interface, Inline Class, Extract Method, Move Field, Move Method, Push Down Field, Push Down Method, Pull Up Field, Pull Up Method, and Move Class. In the following subsections, we describe the different adaptation steps, especially the generic solution representation to unify the search space of our two tasks to be optimized.

3.2.2.1 Unified Search Space Representation

The unified solution space, encoding and decoding steps are key ingredients of the simultasking adaptation. In our case, we have two optimization tasks to perform simultaneously. Given D_j the dimensionality of the jth task, we define a unified search space with dimensionality $D_{multitask}$ equal to $max_j\{D_j\}$. During the population initialization step, every individual is thus endowed with a vector y of $D_{multitask}$ random variables that constitutes its chromosome (its complete genetic material). Essentially, the ith dimension of the unified search space is represented by a random key y_i, and the fixed range represents the box-constraint of the unified space. When addressing a task T_j (refactoring or regression testing), we simply refer to the first D_j random keys of the chromosome. This encoding technique, in place of simply concatenating the variables of each optimization task to form a giant chromosome of $D_1 + D_2$ elements, encourages the discovery and implicit transfer of useful genetic material from one task to another in an efficient manner. In other words, refactoring and regression testing are unified into one solution representation integrating the dimensions of both

tasks. A dimension of the first task is a refactoring operation while the dimension of the second task is a selected test case. Both tasks have in common the methods (the knowledge to transfer) that need to be either refactored and/or tested.

Both tasks of refactoring and test cases selection are discrete problems, and every test case and/or refactoring is applicable to the methods of the system to evaluate, either in part or in whole. For example, a test case may cover a method in the code or a refactoring can be applied to it. Thus, the overlap of both phenotypes is implicitly represented by the methods to be covered by test cases or to be refactored and the random keys can be decoded in a straightforward manner. The ith random key of an individual is viewed as a method assigned to test case(s) and refactoring(s). Thus, that assigned method needs to be refactored and covered by the test cases. To assign test cases/refactorings to methods, the random keys are simply sorted in ascending order, with each random key corresponding to a method. Then, test cases/refactorings are assigned to methods according to their position in the sorted list. When a class-level refactoring is assigned to a method, we select the class containing that method. It is possible to use different levels of granularity than the method level (such as the statement level) for the solutions representation. We decided to use the method level to ensure a reasonable analysis/execution cost.

The simplified illustrative example depicted in Figure 3.1 summarizes the cross-domain decoding procedures described heretofore. For our 2-factorial problem, we have $D_{multitask}(testcases, refactorings) = max\{3, 4\} = 4$. Therefore, we randomly generate a 4-dimensional chromosome with the following random key representation: (0.23, 0.12, 0.42, 0.11). This 4-dimensional chromosome represents the unified solution representation. Then, we randomly select a set of n test cases for the sequence of methods corresponding to three keys (0.23, 0.12, 0.42) and a random set of m refactorings applied to the methods corresponding to all four keys (0.23, 0.12, 0.42, 0.11). Of course, the methods' precedence is important, since it may impact the

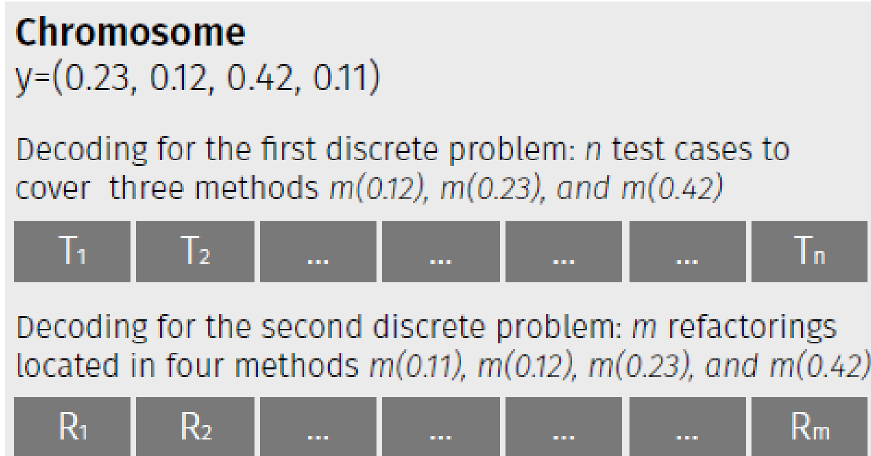

Figure 3.1: An example of decoding a random-key chromosome into domain-specific representations.

evaluation functions. In case there is no exact match between a key and one of the methods, we select the method with the closest Euclidean distance to the key. Please note that this is a simplified example, but in practice the size of both tasks is very different and large. The reader can refer to the following paper for more details about the random-key representation [116]. For our adaptation, the goal is to find a way to represent the engineering material (methods of the source code) as random keys that can be assigned to the different tasks of regression testing and refactoring in a generic manner. Then, these keys can be used for the knowledge transfer between tasks based on the skill factor scores.

Figure 3.2 shows the utility of exploiting knowledge overlap in evolutionary simultasking. In particular, the common knowledge is expected to be primarily contained in the intersecting region in phenotype space that corresponds to the three common methods of our example for the refactoring and regression testing tasks. As explained previously, the exchange of methods between the refactoring and regression testing blocks can happen within the same solution or between solutions using the guided change operators to recombine the blocks.

It is not possible to just simply add additional regression testing objectives to a

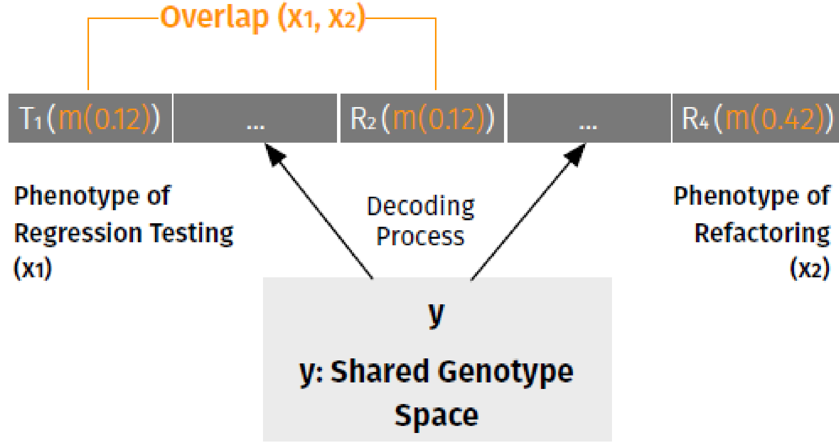

Figure 3.2: A summary of the 2-task environment.

multi-objective refactoring algorithm without unifying the two solutions of refactoring and regression testing (a key contribution of multi-tasking). In fact, it is not possible to evaluate a set of test cases based on refactorings where the solution representation is just a set of test cases. Furthermore, our multi-tasking adaptation proposes also to use novel change operators to ensure the knowledge transfer between both refactoring and regression testing as explained later (another key contribution).

3.2.2.2 Fitness Functions

As already stated, we have four objective functions to consider in our problem formulation. It is possible to aggregate the two fitness functions of each task to reduce the number of non-dominated solutions and the computational cost. However, we preferred to use four fitness functions since we found them conflicting and the algorithm still provides good solutions with reasonable computational cost. Furthermore, the use of the skill factors along with the fitness functions reduced the number of non-dominated solutions due the intentional overlap between these solutions.

The first two fitness functions can be stated as follows:

$$\begin{cases} f_1(X,S) = \frac{NCCS(X,S)}{NDCS(S)} \\ f_2(X,S) = length(X) \end{cases} \tag{3.1}$$

$$Subject\ to\ x = \{x_1, \ldots, x_n\} \in X$$

where X is the set of all refactoring sequences of software system S, x_i is the ith refactoring in the sequence X, $NCCS(X,S)$ is the Number of Corrected Code Smells (NCCS) after applying the refactoring solution X to system S, $NDCS(S)$ is the Number of Detected Code Smells (NDCS) prior to the application of solution X to S. The code smells are detected on the system before and after refactoring using the rules defined in [14]. Thus, f_1 defines the ratio of the number of corrected code smells per all known smells in the system (to maximize), and f_2 represents the number of operations to apply (to minimize).

The two remaining fitness functions can be stated as follows:

$$\begin{cases} f_3(T,S) = (1 - \frac{TS_1+TS_2+\ldots+TS_M}{N \times M}) + (1 - \frac{TC_1+TC_2+\ldots+TC_M}{N \times M}) + \frac{1}{N} \\ f_4(N) = |T' \cap \bar{T}| + \sum_{i=0}^{N.length} ET_i \end{cases} \tag{3.2}$$

$$Subject\ to\ TS = (TS_1, \ldots, TS_n) \in T\ and\ TC = (TC_1, \ldots, TC_n) \in T$$

where T is the test suite of all N test cases of system S, TS_i indicates a test case covering statement i, while TC_i, similar to TS_i, represents a test case that covers a **changed** statement i; M is the number of changed statements and can be easily calculated by exploring the code elements involved in each refactoring operation that belongs to the solution and the regular developers' code changes found in recent commits; ET_i is the run-time of test case i; T' represents the subset of the selected test cases to execute; f_3 aggregates our coverage criteria (to maximize) and f_4 denotes

the number of selected test cases (to minimize) and measures the runtime of the execution order of the selected test cases (to minimize). We did not aggregate the functions since the multifactorial algorithm supports the use of multiple objectives.

3.2.2.3 Evolutionary Operators

As with most Evolutionary Algorithms (EAs), the Multi-Factorial Evolutionary Algorithm (MFEA) uses crossover and mutation as its core genetic mechanisms and the key for the knowledge transfer between tasks. Following the principles of assortative mating in the bio-cultural models of multifactorial inheritance, crossover is favored to occur between parents belonging to the same cultural background (between solutions associated to the same task in our case). The goal of the crossover is to ensure the knowledge transfer between two solutions where each solution is good in at least solving one task. The crossover will generate new solutions by taking the best from each block (having a high skill factor) of best solutions that are selected for each task separately. Thus, this operator will keep the blocks of the solution having high scalar fitness values of each task in the new generated solutions and replacing the methods of the blocks with low scalar fitness for each task by the methods of the block with high scalar fitness (either between the same or different tasks). The mutation operator will introduce changes to the methods of each block based on the skill factor of each block. For instance, some of the methods of the block having a high refactoring skill factor will be replacing the methods of a block having a low regression skill factor or a block of another solution with a low refactoring skill factor. Figure 3.3 illustrates an example of using the crossover to take the best from each solution for each task (S1) and transferring the engineering material (methods) from the regression testing task to the refactoring task (S2).

The constructive scalar fitness allows the evolution of these solutions within the same search space. This process encourages the discovery and implicit transfer of

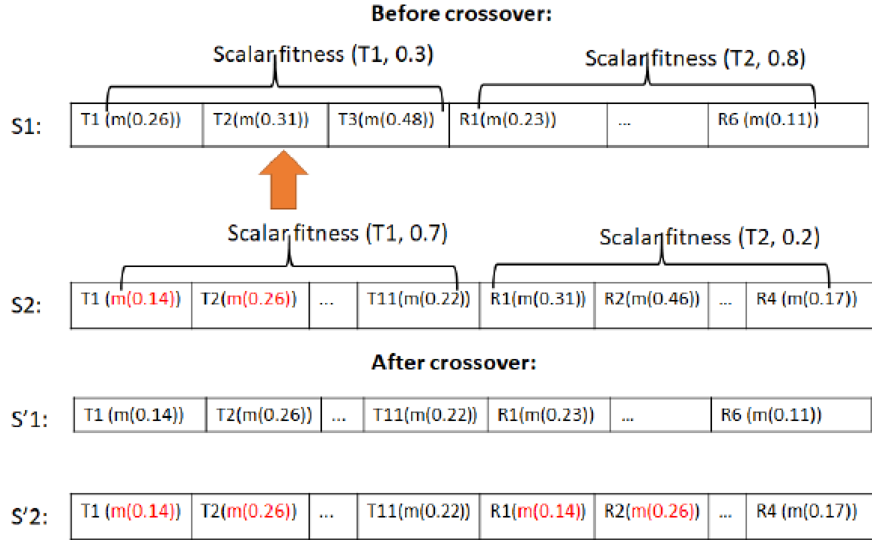

Figure 3.3: An illustration of the assortative mating using the adapted crossover for knowledge transfer.

useful genetic material from one task to another. The use of genetic operators allows the exchange of genetic building blocks corresponding to more than one optimization task when creating offspring solutions. The assortative selection is one of the most important components of the simultasking algorithm. Of course, the use of the different change operators is combined with a set of pre/post-conditions to check the correctness of the generated solutions after applying the operators. For example, if the algorithm recommends to move method m_1 from class C_1 to class C_2, but a method having the same signature as m_1 already exists in C_2, then it will be considered as an invalid refactoring operation. The list of refactoring pre/post conditions can be found in [117].

3.3 Validation

3.3.1 Research Questions

3.3.1.1 RQ1-A: Quality Improvement

To what extent can our approach improve the quality of software systems as compared to mono-task refactoring techniques?

In RQ1-A we use code smells [7] and internal quality attributes [118] as proxies to assess the quality improvement brought by the refactoring operations generated by Multi-Objective-Multi-Factorial Optimization (MO-MFO). We compare its performance with two, state-of-the-art, mono-task refactoring techniques: Ouni *et al.* [11] and JDeodorant [31]. Ouni *et al.* [11] proposed a mono-task multi-objective refactoring formulation based on NSGA-II using the two fitness functions of the refactoring task described in this paper. JDeodorant [31] is an Eclipse[5] plugin able to detect code smells and automatically recommend refactorings to fix them. JDeodorant is not based on the use of heuristic search. As JDeodorant supports a lower number of refactoring types with respect to the ones we considered, we restrict our comparison with it to these refactorings.

3.3.1.2 RQ1-B: Refactoring Meaningfulness

Are the refactoring recommendations produced by MO-MFO meaningful from a developer's point of view? How do they compare with those generated by a mono-task technique?

Using anti-patterns and internal quality indicators as proxies for code quality (as we do in RQ1-A) has strong limitations. For this reason, in RQ1-B we survey 25 developers asking for their opinion about the meaningfulness of the refactorings recommended by our technique and by the mono-task competitive techniques [11]. In

[5] https://www.eclipse.org/

RQ1-B we do not compare with JDeodorant since the mono-task refactoring technique of Ouni *et al.* outperformed JDeodorant based on the systems considered in our experiments [11]. The main substantial difference between MO-MFO and the approach by Ouni *et al.* [11] is indeed the multi-task perspective of MO-MFO. This allows us to verify whether multi-task permits MO-MFO to identify meaningful refactorings missed by the mono-task approach.

3.3.1.3 RQ2-A: Synergy between Regression Testing and Refactoring to Support Software Maintenance in Practice

To what extent can MO-MFO support the simultaneous selection of relevant test cases for both refactoring and regular code changes while still finding relevant refactorings in a real world scenario?

We integrated a beta version of MO-MFO into a previously licensed refactoring tool, and asked one of our industrial partners to use it for a limited period of 5 business days (with 6 developers involved). During this period, we checked the ability of MO-MFO to select relevant test cases by identifying real bugs for both refactoring and regular code changes introduced by the programmers during their daily activities. We have also evaluated the ability of MO-MFO to recommend relevant refactorings during that period.

3.3.1.4 RQ2-B: Testing Effort Reduction and Refactoring Coverage

What is the effectiveness of our approach in maximizing the coverage of the recommended refactorings and introduced code changes, while reducing the number of selected test cases?

Here we focus on the coverage of the statements changed due to the implemented refactoring operations. Clearly, only analyzing the coverage of the selected test cases does not tell the whole story about the usefulness of a regression testing technique.

Indeed, a trivial solution to maximize code coverage would be to select the whole test suite. However, this would not reduce the testing cost. For this reason in RQ2-B, we consider both the code coverage ensured by the selected test cases as well as the percentage of selected test cases as proxies to evaluate the test case selection effectiveness. We compare MO-MFO with two baselines. The first one, is a Mono-task Multi-objective Regression Technique (MOT) exploiting exactly the same fitness functions as the two we use in MO-MFO to support regression testing. The second is the approach proposed by Yoo and Harmon [32] based on a greedy bi-objective algorithm to maximize the coverage and reduce the cost. In the industry validation study, we have also evaluated the ability of our approach in detecting real bugs by using the recommended test cases.

3.3.1.5 Study Context

The *context* of our study is represented by the seven systems in Table 3.1. We selected these seven systems for our validation because they range from medium to large size projects and have been actively developed over the past 10 years. JDI[6] is an industrial project for which 6 of the developers involved in the maintenance of JDI agreed to take part in RQ1-B and RQ2-A.

Table 3.1: Statistics of the studied systems.

System	Release	# Classes	# Smells	KLOC	# Test Cases	Method Coverage
Xerces-J	v2.7.0	991	91	240	2218	47%
JHotDraw	v7.5.1	585	25	21	663	53%
JFreeChart	v1.0.18	521	72	170	2217	72%
GanttProject	v1.11.1	245	49	41	842	61%
JDI	v5.8	638	88	247	2647	68%
Apache Ant	v1.8.2	1191	112	255	2703	62%
Rhino	v1.7.5	305	69	42	973	71%

Table 3.1 provides information about the size of the subject systems (in terms of number of classes and KLOC), number of code smells affecting them as detected with the rules defined in [14], number of JUnit[7] test cases they have, and the method cov-

[6]Company anonymized due to request for confidentiality.
[7]https://junit.org/junit5/

erage ensured by these tests. For instance, JFreeChart[8] has 2,217 test cases resulting in a ratio of ∼4.2 test methods per class. The initial test case method coverage varies from one project to another. Apache Ant[9] has 2,703 tests that cover 62% of its 2691 methods while Xerces-J[10] achieves 47% coverage with 2,218 test cases. With such a number of test cases, the compilation and execution of the complete test suite may require substantial time (almost 90 minutes for Apache Ant with an i7 Processor and 8GB RAM) due to factors such as external dependencies and test case timeouts.

3.3.2 Data Collection

We present the data collection and analysis process grouped by research question category: refactoring or regression testing.

3.3.2.1 RQ1 - Refactoring

To address **RQ1-A**, we calculated the *Number of Fixed Code Smells (NF)* as the percentage of code smells fixed by the refactoring solutions generated by the three considered approaches, over the total number of code smells affecting the subject systems. The detection of code smells before/after applying a refactoring solution was performed with the rules defined in [14]. The considered code smells are *Blob Class (BC), Feature Envy (FE), Data Class (DC), Spaghetti Code (SC), Functional Decomposition (FD), and Shotgun Surgery (SS).*

Since the concept of a code smell is very subjective (different developers may have different opinions on whether a code component is smelly or not) [119], we also use more objective metrics to assess the quality of the refactorings generated by the experimental approaches. We adopted the G metric based on *QMOOD* [118] that estimates the quality improvement of the system by comparing the quality be-

[8]https://www.jfree.org/jfreechart/
[9]https://ant.apache.org/
[10]https://xerces.apache.org/xerces-j/

fore/after refactoring independently from the number of fixed design defects. Four quality factors are considered by *QMOOD*: reusability, flexibility, understandability and effectiveness. All of them are formalized using a set of quality metrics. Hence, the total gain in quality G for each of the considered *QMOOD* quality attributes q_i before and after refactoring can be estimated as:

$$G = \frac{\sum_{i=1}^{4} G_{q_i}}{4} \quad where \quad G_{q_i} = q'_i - q_i \tag{3.3}$$

where q'_i and q_i represent the value of the quality attribute i respectively after and before refactoring.

To answer **RQ1-B** we asked 25 developers to evaluate the meaningfulness of the refactorings recommended by MO-MFO (multi-task) and by the approach of Ouni *et al.* [11] (mono-task) on the seven subject systems. Before explaining the study design for RQ1-B, it is important to remember that both the experimental techniques generate output sequences of refactoring operations that make sense when considered together rather than when looking at them in isolation. However, it is not an option to ask a developer to assess the meaningfulness of all the refactoring operations (potentially hundreds) generated for a given system. For this reason, we started by filtering, for each system, the sequences of refactoring operations impacting (1) a single subsystem and (2) no more than ten classes. Then, from these sequences of refactorings, we randomly selected two sequences per system per treatment (meaning, four sequences per system, two generated by MO-MFO and two by [11]).

Each participant was then asked to assess the meaningfulness of four sequences of refactoring operations: two generated by MO-MFO and two by [11]. Since on six of the seven systems (all but JDI) we involved external developers (professional developers who did not take part in the development of the subject system), we made sure that each participant only evaluated refactoring sequences recommended by the

two competitive techniques on one specific system (*e.g.* JHotDraw[11]). The rationale for such a choice is that an external developer would need time to acquire a system's knowledge by inspecting its code, and we did not want participants to comprehend the code from four different systems, since this would introduce a strong tiring effect in our study. The six developers of the JDI project evaluated the refactoring sequences generated for that system, since here we wanted to exploit their experience as original developers of the system. They used MO-MFO, as a beta version tool, during a period of 5 days instead of a refactoring tool that we licensed to their company in the past. Our industrial partner was motivated to try out MO-MFO since they are interested in estimating the refactoring cost in terms of testing especially with relevant refactoring suggestions. They also expressed a concern about the lack of regression testing support in existing refactoring tools which make their developers reluctant to apply refactorings.

To support such a complex experimental design, we built a Java[12] Web-app that automatically assigns the refactoring sequences to be evaluated to the developers. The web-app showed each participant one sequence of refactoring operations on a single page, providing the developer with (1) the list of refactorings (Move Method m_i to class C_j , then Push Down Field f_k to subclass C_j, *etc.*), (2) the code of the classes impacted by the sequence of refactorings, and (3) the complete code of the subject system of the refactorings. The web page showing the refactoring sequence asked participants the question, *Would you apply the proposed refactorings?*, with a choice between *no* (the refactoring sequence is not meaningful), *maybe* (the refactoring sequence is meaningful, but the quality improvement it brings does not justify changing the code), or *yes* (the refactoring sequence is meaningful and should be implemented). Moreover, participants were allowed to leave a comment justifying their assessment (this was optional).

[11]https://www.randelshofer.ch/oop/jhotdraw/
[12]https://www.java.com/en/

Table 3.2: Participants involved in RQ1-B.

System	# Participants	Avg. Prog. Exp.	Avg. Java Exp.	Avg. Refact. Exp.(1-5)
Xerces-J	4	11	9	4.0 (high)
JHotDraw	4	10	7	3.0 (medium)
JFreeChart	4	10	7	3.3 (medium)
GanttProject	4	9	8	3.5 (high)
JDI	6	14	12	4.5 (very high)
Apache Ant	3	9	7	3.7 (high)

Table 3.2 shows the participants involved in our study and how they were distributed in the evaluation of the refactoring sequences generated on the seven systems. For the 5 day industrial validation, we integrated a routine in our MO-MFO tool to record all the actions of the 6 developers including the number of applied and rejected refactorings, number of selected test cases, the introduced code changes, commit messages, and the identified bugs by the test cases.

3.3.2.2 RQ2 - Regression Testing

To answer **RQ2-A**, we evaluated the number of identified bugs for both refactorings and regular code changes by the selected test cases. We have also calculated the percentage of selected test cases to cover both the code changes and refactorings. We calculated, as well, the number of applied and rejected refactorings by each developer.

We answer **RQ2-B** by computing the *Median Effort to Test Refactorings (EF)* metric, aimed at evaluating the ability of the simultasking approach to generate refactoring solutions with a minimum testing effort and defined as the ratio of *selected Test Cases (TCs)* to cover the maximum number of code components impacted by refactoring operations over the number of *TCs* in the original test suite:

$$EF = \frac{\#\ TCs\ in\ reduced\ test\ suite\ covering\ as\ much\ as\ possible\ all\ refactorings}{\#\ TCs\ in\ original\ test\ suite} \times 100 \tag{3.4}$$

3.3.3 Experimental Settings and Data Analysis

For each algorithm and for each system, we performed a set of experiments using several population sizes: 50, 100, 200, and 300. The upper and lower bounds on

the chromosome length were set to 10 and 350, respectively. The stopping criterion was set to 10,000 fitness evaluations for all algorithms to ensure fairness. Since the offspring performance is measured by the skill factor that it imitates, the *rmp* value was set to 0.5 to increase the probability of applying the crossover between parents regardless of their skill factor to increase the diversity of the solutions.

We note that the mono-objective deterministic refactoring approach JDeodroant only provides one refactoring solution, while the other algorithms generate sets of non-dominated solutions. To make meaningful comparisons, we selected the best solution for the simultasking and multi-objective algorithms using a knee-point strategy. The knee point corresponds to the solution with the maximal trade-off between the different objectives. Thus, we selected the knee point from the Pareto approximation having the median hyper-volume IHV value.

The stochastic nature of the deployed algorithms required 30 independent simulation runs for each problem instance [120], and the obtained results were statistically analyzed with the Wilcoxon rank sum test with a 95% confidence level ($\alpha = 0.05$). We used the Vargha-Delaney A measure [120] which is a non-parametric effect size measure. In our context, given the different performance metrics (*NF*, G), the A statistic measures the probability that running an algorithm B1 yields better performance than running another algorithm B2. If the two algorithms are equivalent, then A = 0.5.

Concerning RQ1-B, we report the percentage of refactoring sequences assessed with a *no*, *maybe*, or *yes* by developers for each treatment (MO-MFO and [11]) and system. Then, we discuss interesting comments left by developers when justifying their assessment.

3.3.4 Results

3.3.4.1 RQ1-A: Quality Improvement

Figure 3.4 and Figure 3.5 provide the percentage of fixed code smells (*NF*) and the quality gain (*G*) based on the *QMOOD* model, respectively. The average *NF* on the seven systems is 86% with peaks of ∼90% for JFreeChart and GanttProject[13]. The recommended refactorings also improved the *G* metric values (Figure 3.5) of the seven systems. The average quality gain for the JFreeChart system was the highest among the seven systems with 0.38. The improvement in the quality gain shows that the recommended refactorings help to optimize different quality metrics. In addition, the performance of MO-MFO is superior as compared to the competitive refactoring techniques [11, 31], even though the difference in terms of fixed code smells is not that marked (Figure 3.4). This latter result is also due to the fact that MO-MFO does not only recommend refactoring operations aimed at removing code smells but, thanks to the knowledge transfer from the regression testing task, it also focuses on refactoring classes not affected by code smells but covered by the test cases. For example, in a manual investigation of the refactorings recommended by MO-MFO for JFreeChart, we found that 17 of the impacted classes do not exhibit any criticality as indicated by code quality proxies such as metrics and code smells. This is the case of the *EncoderUtil* class, only recommended for refactoring by MO-MFO, and not by the other algorithms. This indicates that the knowledge transfer between refactoring and regression testing is likely the reason for the refactorings applied to *EncoderUtil*.

The statistical tests confirm that for the different evaluation metrics: (1) on small software projects (GanttProject and Rhino[14]), our approach outperforms the competitive techniques with an A value > 0.91; (2) on medium and large software projects (JDI, Apache Ant, Xerces-J, JHotDraw, and JFreeChart), it achieves higher *NF*/*G*

[13] https://www.ganttproject.biz/
[14] https://mozilla.github.io/rhino/

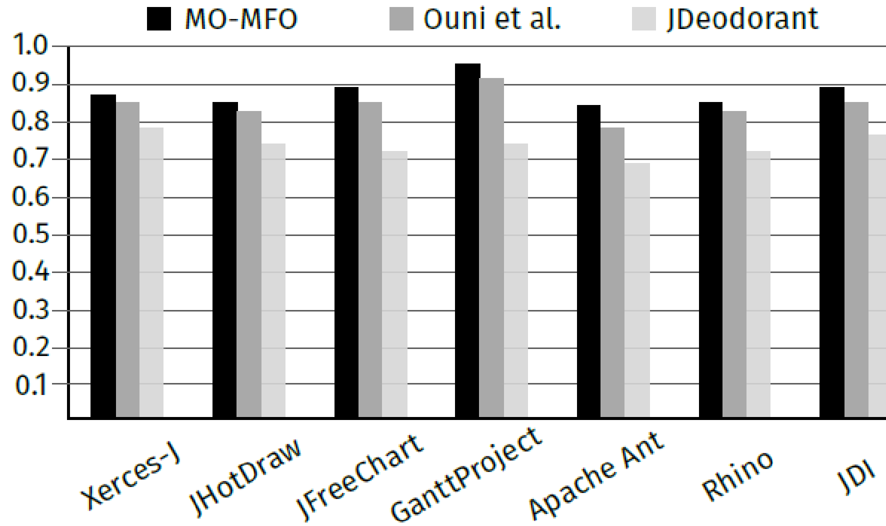

Figure 3.4: Median percentage of fixed code smells (NF) over 30 runs at the 95% confidence level ($\alpha < 5\%$).

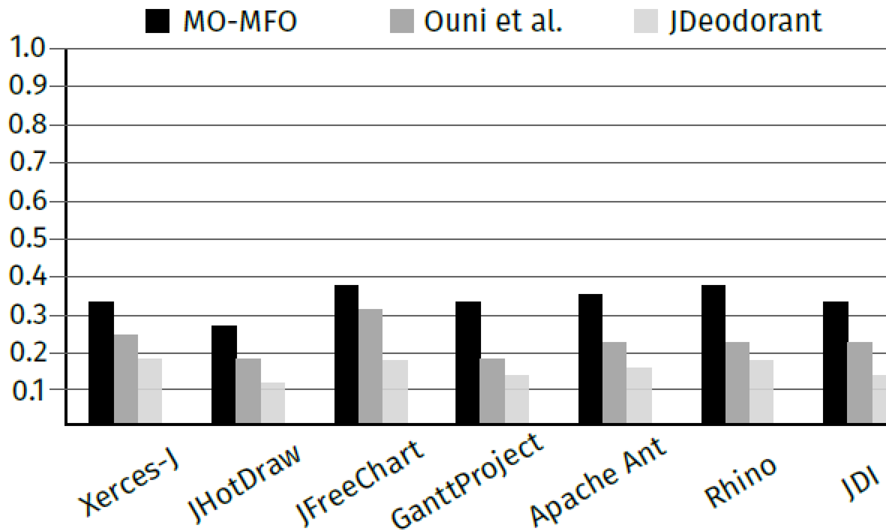

Figure 3.5: Median quality gain (G) over 30 runs at the 95% confidence level ($\alpha < 5\%$).

w.r.t. the other algorithms with an A value > 0.86.

3.3.4.2 RQ1-B: Refactoring Meaningfulness

Table 3.3 summarizes the manual refactoring evaluation results obtained from the 25 participants. Note that there is a slight deviation between the total number

of refactorings evaluated by the two approaches (136 *vs.* 144) since, as explained in Section 3.3, we did not consider the data analysis for the evaluations in which participants spent less than 60 seconds to assess the meaningfulness of the refactoring sequence under analysis.

Table 3.3: RQ1-B: Would you apply the proposed refactorings?

Approach	**No**	**Maybe**	**Yes**
MO-MFO	15/136 (11%)	18/136 (13%)	103/136 (76%)
Ouni *et al.* [11]	27/144 (19%)	49/144 (34%)	68/144 (47%)

The unification of regression testing and refactoring improved the relevance of the recommended refactorings compared to the mono-task multi-objective approach. Indeed, while the percentage of meaningful recommendations (the sum of the *maybe* and *yes* answers) is similar between the two approaches (89% for MO-MFO and 81% for Ouni *et al.*), the percentage of refactorings that participants believe must be applied (*yes* answers) is significantly higher for MO-MFO (76% vs 47%). Looking at the comments left by participants when justifying their assessment, four out of the six original developers of the JDI system highlighted in their comments for three refactoring sequences that they found the refactorings relevant because it is improving the modularity of a buggy class that they frequently modify. For example, one of the developers wrote in a comment:

> "That's a relevant one, I spent days fixing one of the bugs located there so I like this extract class and move method. It may probably take me less time in the future to fix future bugs in that class after your recommendation to split it and move some methods out of it".

We found this comment as important qualitative evidence of the value of unifying refactoring and regression testing especially that (1) this class was not affected by any code smell, and (2) the comment comes from an original developer of the industrial system.

3.3.4.3 RQ2-A and RQ2-B: Test Case Selection Coverage and Effort Reduction

Figure 3.6 summarizes the results of deploying our simulatsking tool during 5 business days to our industrial partner. The six developers used the tool as part of their daily programming activities instead of a previously licensed refactoring tool. The tool was deployed as a web app that connects automatically to a private GitHub repository whenever a number of code changes are introduced by the developers to simultaneously check for refactorings and select test cases for regression testing. The total number of bugs successfully detected by MO-MFO was 22. Six of these bugs were related to the refactorings applied by the developers as a consequence of the approach's recommendations, and the remaining sixteen were related to recent code changes. The total number of statements modified by the developers during the 5 days are 227 which corresponds to 31 methods. The number of selected test cases by MO-MFO during this period was 72 test cases out of 2,647. The developers accepted 37 refactorings recommended by our tool and rejected 8.

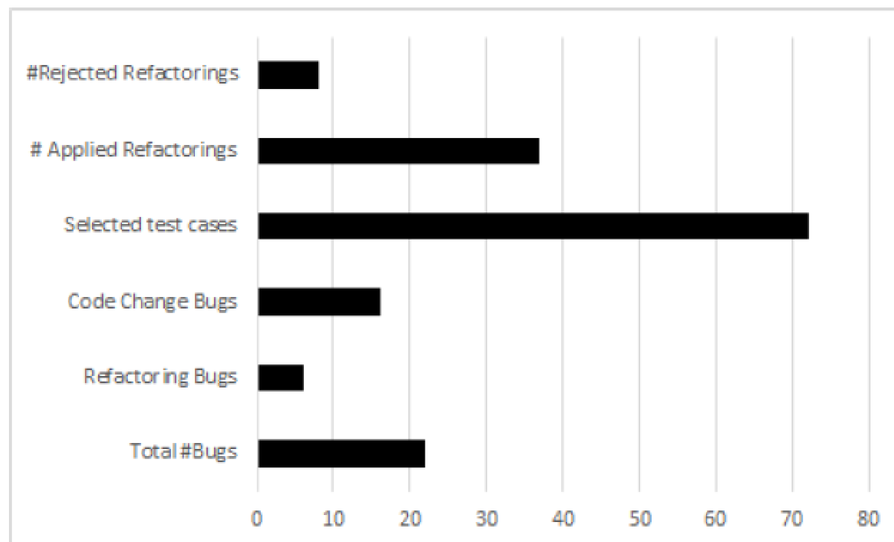

Figure 3.6: The outcomes of the industrial validation on the JDI system by 6 developers during 5 days.

Overall, the achieved results confirm the effectiveness of our approach to enable both tasks simultaneously. We found that 24 out of the 37 accepted refactorings impact the same classes where the bugs were located. The achieved results confirm the basic intuition behind this work, showing that buggy files might be in need of refactoring even if their code quality as assessed by code smells/quality metrics is not problematic. The six developers also confirmed that they feel more comfortable in applying refactorings due to the integrated support for regression testing. This may explain the reason why a good number of recommended refactorings were applied.

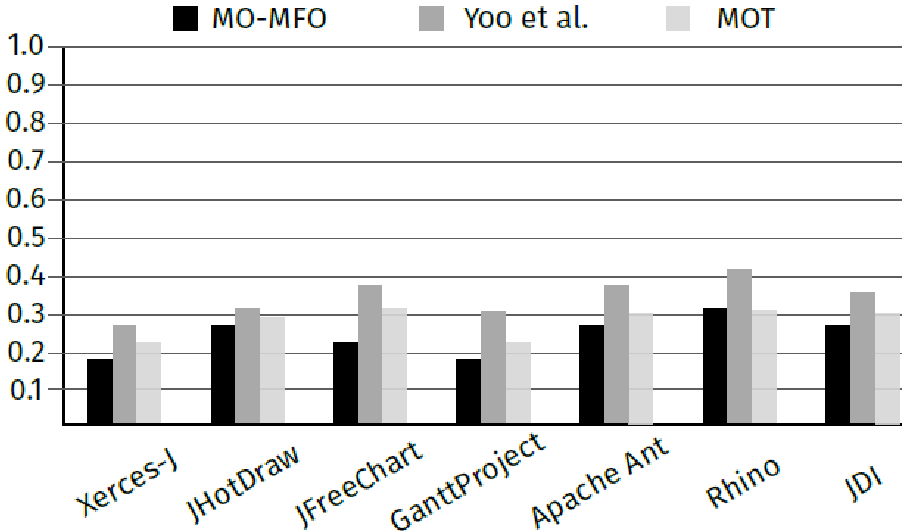

Figure 3.7: Median effort to test the refactorings (EF) over 30 runs at the 95% confidence level ($\alpha < 5\%$).

Finally, Figure 3.7 confirms the effectiveness of the simultasking technique in terms of selecting relevant test cases to cover the introduced refactorings. Most of the components impacted by the refactorings recommended by the simultasking approach were covered by the selected test cases, while requiring an effort/number of test cases (EF) representing, on average 24% of the original test suite size. When comparing MO-MFO to both the greedy regression testing approach [32] and multi-objective mono-task one, MOT, the test cases selected by MO-MFO cover the expected refactorings with similar effectiveness, but with a lower number of test cases (EF) as shown

in Figure 3.7.

3.4 Threats to Validity

Our simultasking formulation treats the two tasks of refactoring and test case selection with the same importance, but developers may have different priorities when working on these two tasks simultaneously. Another internal threat is related to the used detection rules of code smells that may identify false positives and miss false negatives.

Construct validity is concerned with the relationship between theory and what is observed. To evaluate the results of our approach, we selected solutions at the knee point when comparing with other techniques, but developers may select a different solution based on their preferences in order to give different weights to the objectives when selecting the best refactoring or test case solution.

External validity refers to the generalize-ability of our findings. We performed our experiments on six open-source systems belonging to different domains, and one industrial project, by involving 25 participants in the evaluations of the refactoring operations. However, we cannot assert that our results can be generalized to other applications, and to other developers. Furthermore, we can extend the manual validation of suggested refatorings to include a larger number of recommendations. Future replications of this study are necessary to confirm our findings.

3.5 Conclusion

We presented a first attempt to unify two different software engineering problems using the tasks of refactoring and regression testing as a case study. The salient feature of the proposed simultasking approach is that it incorporates a unified solution representation scheme, which serves as a common platform for knowledge transfer,

hence allowing each task to influence the solution of the other. To evaluate the effectiveness of our technique, we applied it to six open-source projects and one industrial project comparing it with state-of-the-art approaches. Our results show promising evidence on the usefulness of the simultasking approach.

Future work will involve validating our technique with additional refactoring types, test cases, programming languages, code smell types, and additional tasks (*e.g.* the next release problem).

CHAPTER IV

Dependent or Not: Detecting and Understanding Collections of Refactorings

In this chapter we present contribution 2: detecting and understanding collections of refactorings. In this contribution, we begin with Section 4.1 that demonstrates an genuine instance of links between refactorings for a popular open-source project. In Section 4.2, we discuss our refactoring dependency theory and algorithm for detecting dependencies in refactoring recommendation lists. In Section 4.3, we present our empirical study and validation of our technique. We discuss the threats to validity in Section 4.4. Next, we highlight the implications of the contribution and propose directions of future work in Section 4.5. Finally, we conclude in Section 4.6 with a summary of our work.

4.1 A Motivating Example

The key to applying refactorings successfully is the decision of which refactorings to apply and where to apply them. In essence, this requires a developer to instantiate a refactoring by supplying the parameters that allow a type of refactoring to be unambiguously applied to code. For example, to instantiate a Move Method refactoring operation, a developer must supply parameters that indicate which method to move

and where to move it. Throughout this contribution, when we talk about refactorings and dependencies among refactorings, we are referring to refactoring instances.

While refactoring recommendations generated by tools to mimic this activity are typically represented as sequences, not all orderings in these sequences are significant. That is, the same code could be generated by two solutions that contain the same refactorings, but simply each solution applies them in a different order. This is because while many refactorings are independent of one another, other refactorings are dependent on each other such that removing or reordering a refactoring from a solution could make other refactorings invalid. With the current growth of interactive tools to support refactoring [14, 18], developers are offered solutions that contain dozens to hundreds of refactorings and the option to selectively apply elements of a solution. Without a theory for reasoning about refactoring dependencies, developers can inadvertently make decisions, *e.g.* ignoring part of a solution, that result in failure (code cannot be successfully refactored) and enter a tedious trial and error loop. Making refactoring dependencies visible improves developers' understanding of how refactorings work together and allows them to make sound inferences regarding their application.

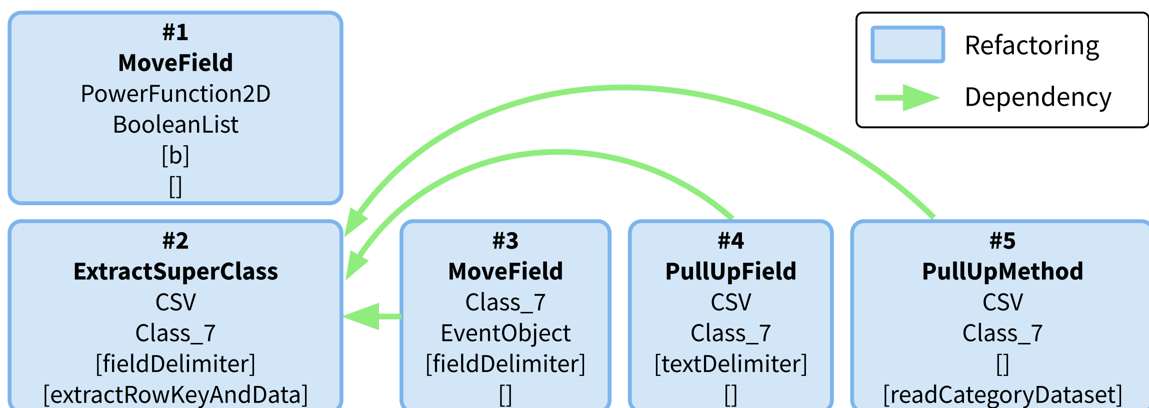

Figure 4.1: A simplified solution of 6 refactorings for the JFreeChart project.

Figure 4.1 shows a simplified example of a solution composed of 5 refactorings to be applied to the JFreeChart project. Three of the refactorings (#3, #4, #5) depend

on another refactoring (#2) because the Extract Super Class refactoring (#2) creates a new class (Class_7), on which refactorings #3, #4, and #5 operate. If the new class is not created first, then refactorings #3, #4, and #5 will fail. Thus, there exists an ordering dependency from each of #3, #4, and #5 to #2. Refactoring #1, however, has distinct parameters, indicating that it operates on different code elements, thus it has no ordering dependencies on any others in this solution. Presenting these dependencies to a developer clarifies the options that the developer has to refactor their code. For example, the developer could choose not to apply any refactoring except for refactoring #2 without consequences; if the developer chooses not to apply refactoring #2, then refactorings #3, #4, and #5 cannot be applied either. Detecting ordering dependency relationships among refactorings is essential to more effectively applying refactorings.

4.2 Refactoring Dependency Theory

The refactoring dependency theory for reasoning about collections of refactorings is built upon two concepts. The first is the definition of an ordering dependency relation among refactorings in a collection of refactorings. Pre- and post-conditions for refactoring types are used to detect refactoring dependencies. The second is the organization of a collection of refactorings as a set of refactoring graphs. Together, these concepts improve our ability to understand the meaning of collections of refactorings, allowable operations on them, and their composition in practice.

In this section, we describe the elements of our theory, the algorithm for detecting refactoring dependencies, and an associated web-tool that implements this detection algorithm.

4.2.1 Definitions

Our proposed dependency relation captures an ordering dependency between pairs of refactoring instances. Specifically, an ***ordering dependency*** ($rf_2 \mapsto rf_1$) between two refactoring instances (rf_1 and rf_2) exists when rf_2 can only be successfully applied after rf_1 has been applied. That is, rf_1 makes a change to code that is necessary in order to apply rf_2. This condition can be evaluated based on the combination of pre- and post-conditions of the types of refactorings involved and the parameters of each refactoring instance. For example, to apply Move Method (a type of refactoring) to move method m_1 from class c_1 to class c_2 (m_1, c_1, and c_2 being the parameters of the refactoring instance), several pre-conditions must hold (*e.g.* m_1, c_1, and c_2 must all exist and m_1 must be defined on c_1). The pre- and post-conditions of each type of refactoring will be described in the next sub-section.

Building on this ordering dependency definition, we organize collections of refactorings as sets of refactoring graphs rather than as sequences of refactorings. A ***refactoring graph*** is a weakly connected directed acyclic graph composed of refactoring instance vertices and ordering dependency edges. Using the ordering dependencies as the basis for forming refactoring graphs (Algorithm 1) results in a set of graphs with the following traits:

- Each refactoring instance is an element of exactly one refactoring graph.
- Some graphs contain a single refactoring instance because that refactoring is truly independent of all others. We call these ***trivial graphs*** comprised of a single node of a refactoring instance.
- The remaining graphs contain multiple refactoring instances, each of which is part of one or more dependencies. We call these ***non-trivial graphs***.
- Each refactoring graph is independent of every other graph in the solution.

Refactoring recommendations typically comprise a collection of compatible refactorings, and as such positive dependencies are more relevant to common use cases. The idea of negative refactorings would be more applicable if a recommendation contained mutually exclusive advice, *e.g.* three Move Method refactorings that move the same method to three different locations. This is not the common use case, but this work could be easily adapted. The essence of identifying a refactoring that precluded (or invalidated) another could be performed using the same pre- and post-conditions, but with a modification to check for differences rather than commonalities, *e.g.* refactoring #1's post-condition moves the location of a method to class A and refactoring #2's pre-condition requires that same method to reside in class B. It may require additional work to consider the initial state of a program, but the same principles would likely apply.

4.2.1.1 Refactoring Pre- and Post-Conditions

As our approach for detecting ordering dependencies relies on the pre- and post-conditions of specific types of refactorings, we began with validated conditions in the current literature [37, 44, 78, 89] related to 14 types of refactorings. We selected the refactoring types summarized in Table 4.1 since they were those most frequently used in practice based on existing studies [41, 43, 83] and since our work focuses more on complex/composite refactoring operations, which have more complex/sophisticated pre/post-conditions. The pre- and post-conditions sets published in current literature were extensively validated for correctness and completeness [37, 44, 78, 89]. The complete list of updated pre- and post-conditions for the 14 supported types of refactorings are organized in Table 4.1. In this table, the post-conditions that are presented are only those that represent a change. This is important for the dependency detection algorithm, allowing it to efficiently identify changes that enable pre-conditions of dependent refactorings. Also, the functions used for describing the

pre- and post-conditions and their meanings are defined in [44].

4.2.2 Algorithm for Detecting Refactoring Dependencies

Algorithm 1 describes the process for detecting refactoring ordering dependencies. These dependencies are detected based on comparisons between pre- and post-conditions of refactoring instances. The proposed algorithm takes a list of refactoring instances as input and generates a set of refactoring graphs as output.

Algorithm 1: Dependency Detection Algorithm.

```
Input: refactoring solution C = {r_1, r_2, r_3, ..., r_n}
Output: forest of refactoring graphs F = {f_1, f_2, f_3, ..., f_m}
V ← ∅, E ← ∅;
foreach r_i ∈ C do
    V ← V ∪ r_i;
    P ← post_conditions(r_i);
    foreach r_j ∈ C | j > i do
        Q ← pre_conditions(r_j);
        M ← P ∩ Q;
        if |M| ≠ 0 then
            E ← E ∪ {r_j, r_i};
G ← (V, E);
F ← partition(G);
return F;
```

Lines 1 and 2 initialize the lists of refactoring instances (nodes of the graph, V) and refactoring dependencies (edges of the graph, E). Then, the post-conditions of each refactoring instance of the solution C (collection of refactorings) are evaluated for matching with the remaining refactoring instances in C (Lines 3–13). Specifically, the algorithm looks for any match between predicates of pre- and post-conditions from Table 4.1. That is, if any predicate of the post-condition of one refactoring (any element of P) matches any predicate of the pre-condition of another refactoring (any element of Q), then a dependency has been detected and an edge is added to the graph between those refactorings (Lines 5–10). We repeat this process until all the refactorings have been visited. Then, Lines 14 identifies the different trivial and

Table 4.1: Refactoring types and their pre- and post-condition rules.

Refactoring	Cond.	Rules
Move Method	Pre	exist(c1, c2, m) && NOT(inheritanceHierarchy(c1, c2)) && defines(c1, m) && NOT(defines(c2, m))
(c1, c2, m)	Post	NOT(defines(c1, m)) && defines(c2, m)
Move Field	Pre	exist(c1, c2, f) && NOT(inheritanceHierarchy(c1, c2)) && defines(c1, f) && NOT(defines(c2, f))
(c1, c2, f)	Post	NOT(defines(c1, f)) && defines(c2, f)
Pull Up Field	Pre	exist(c1, c2, f) && isSuperClassOf(c2, c1) && defines(c1, f) && NOT(defines(c2, f))
(c1, c2, f)	Post	defines(c2, f) && NOT(defines(c1, f))
Pull Up Method	Pre	exist(c1, c2, m) && isSuperClassOf(c2, c1) && defines(c1, m) && NOT(defines(c2, m))
(c1, c2, m)	Post	defines(c2, m) && NOT(defines(c1, m))
Push Down Field	Pre	exist(c1, c2, f) && isSuperClassOf(c1, c2) && defines(c1, f) && NOT(defines(c2, f))
(c1, c2, f)	Post	defines(c2, f) && NOT(defines(c1, f))
Push Down Method	Pre	exist(c1, c2, m) && isSuperClassOf(c1, c2) && defines(c1, m) && NOT(defines(c2, m))
(c1, c2, m)	Post	defines(c2, m) && NOT(defines(c1, m))
Inline Class	Pre	exist(c1, c2) && ∀e ∈ elements: defines(c2, e) && NOT(defines(c1, e))
(c1, c2, {elements})	Post	exist(c1) && NOT(exist(c2)) && ∀e ∈ elements: defines(c1, e)
Extract Class	Pre	exist(c1) && NOT(exist(c2)) && methods(c1) > 2 && ∀e ∈ elements: defines(c1, e)
(c1, c2, {elements})	Post	exist(c2) && ∀e ∈ elements: defines(c2, e) && NOT(defines(c1, e))
Extract Super Class	Pre	exist(c1) && NOT(exist(c2)) && methods(c1) > 2 && ∀e ∈ elements: defines(c1, e)
(c1, c2, {elements})	Post	exist(c2) && isSuperClassOf(c2, c1) && ∀e ∈ elements: defines(c2, e) && NOT(defines(c1, e))
Extract Sub Class	Pre	exist(c1) && NOT(exist(c2)) && methods(c1) > 2 && ∀e ∈ elements: defines(c1, e)
(c1, c2, {elements})	Post	exist(c2) && isSuperClassOf(c1, c2) && ∀e ∈ elements: defines(c2, e) && NOT(defines(c1, e))
Encapsulate Field	Pre	exist(c1) && defines(c1, f) && NOT(defines(c1, set_f)) && NOT(defines(c1, get_f))
(c1, f)	Post	defines(c1, set_f) && defines(c1, get_f)
Decrease Field Security (c1, f, newSecLvl)	Pre	exist(c1) && defines(c1, f) && ((newSecLvl == “public” ⟹ (NOT(isPublic(f)) && isProtected(f))) && (newSecLvl == “protected” ⟹ (NOT(isProtected(f)) && NOT(isPublic(f)) && isPrivate(f))))
	Post	(newSecLvl == “public” ⟹ isPublic(f)) || (newSecLvl == “protected” ⟹ isProtected(f))
Decrease Method Security (c1, m, newSecLvl)	Pre	exist(c1) && defines(c1, m) && ((newSecLvl == “public” ⟹ (NOT(isPublic(m)) && isProtected(m))) && (newSecLvl == “protected” ⟹ (NOT(isProtected(m)) && NOT(isPublic(m)) && isPrivate(m))))
	Post	(newSecLvl == “public” ⟹ isPublic(m)) && (newSecLvl == “protected” ⟹ isProtected(m))
Increase Field Security (c1, f, newSecLvl)	Pre	exist(c1) && defines(c1, f) && ((newSecLvl == “private” ⟹ (NOT(isPrivate(f)) && isProtected(f))) && (newSecLvl == “protected” ⟹ (NOT(isProtected(f)) && NOT(isPrivate(f)) && isPublic(f))))
	Post	(newSecLvl == “private” ⟹ isPrivate(f)) && (newSecLvl == “protected” ⟹ isProtected(f))
Increase Method Security (c1, m, newSecLvl)	Pre	exist(c1) && defines(c1, m) && ((newSecLvl == “private” ⟹ (NOT(isPrivate(m)) && isProtected(m))) && (newSecLvl == “protected” ⟹ (NOT(isProtected(m)) && NOT(isPrivate(m)) && isPublic(m))))
	Post	(newSecLvl == “private” ⟹ isPrivate(m)) && (newSecLvl == “protected” ⟹ isProtected(m))

non-trivial graphs that are formed based on the detected dependencies.

To illustrate Algorithm 1, consider the motivating example described in Figure 4.1. This example contains six refactoring instances:

```
#1  ExtractClass(OverwriteDataSet;Class_6;[x]; [addChangeListener])

#2  MoveField(PowerFunction2D;BooleanList;[b];[])

#3  ExtractSuperClass(CSV;Class_7;[fieldDelimiter]; [extractRowKeyAndData])

#4  MoveField(Class_7;EventObject;[fieldDelimiter];[])

#5  PullUpField(CSV;Class_7;[textDelimiter];[])

#6  PullUpMethod(CSV;Class_7;[];[readCategoryDataset])
```

Figure 4.2 illustrates the application of the algorithm to this example. In Step 1, the post-conditions of refactoring #1 are compared to the pre-conditions of all remaining refactorings. Since there is no match among pre- and post-conditions for any of them, no dependency is added. Next, the post-conditions of refactoring #2 are compared in Step 2 to the pre-conditions of the next refactorings in the sequence (#3 - #6). Again, no match is found. In Step 3, the post-conditions of refactoring #3 are compared to pre-conditions of refactorings #4, #5, and #6. For each, a match *exist(Class_7)* is found. Thus, three dependencies are added, from each of #4, #5, and #6 to #3. The algorithm continues, but no additional matches are found. Thus, Algorithm 1 transforms this sequence into three refactoring graphs (two trivial graphs and one non-trivial graph that includes four refactorings).

When the developers interact with the tool by modifying or rejecting some of the refactorings then the dependencies detection algorithm is re-executed to check the impact of those changes on the graphs. Thus, all the graphs will be updated

Figure 4.2: Execution of Algorithm 1 on the example of Figure 4.1.

instantly during those interactions.

4.2.3 DPRef

To validate our definitions and algorithm and to make our refactoring dependency detection approach available to the community, we implemented DPRef[1], a free and open-source web-platform that allows users to provide a sequence of refactorings to be applied as input and generates a set of refactoring graphs as output. The home-page of the DPRef web-tool and a sample input list of refactoring operations for the

[1] https://iselab-dpref.herokuapp.com/

JFreeChart project is demonstrated in Figure 4.3.

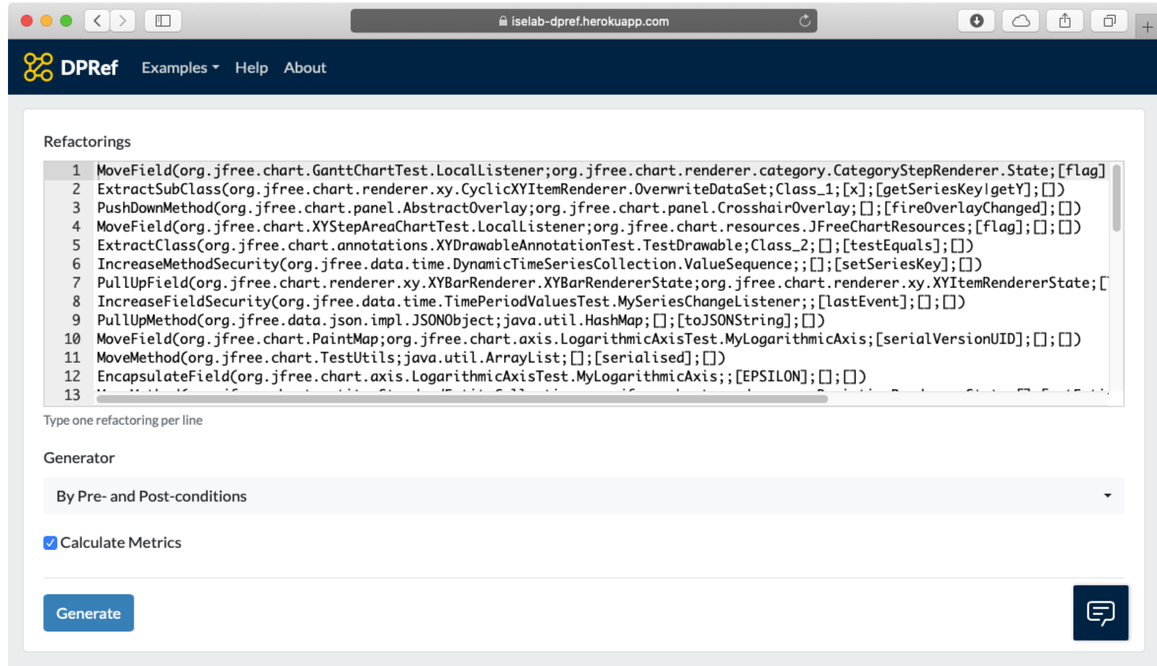

Figure 4.3: DPRef, a web-tool for detecting refactoring dependencies.

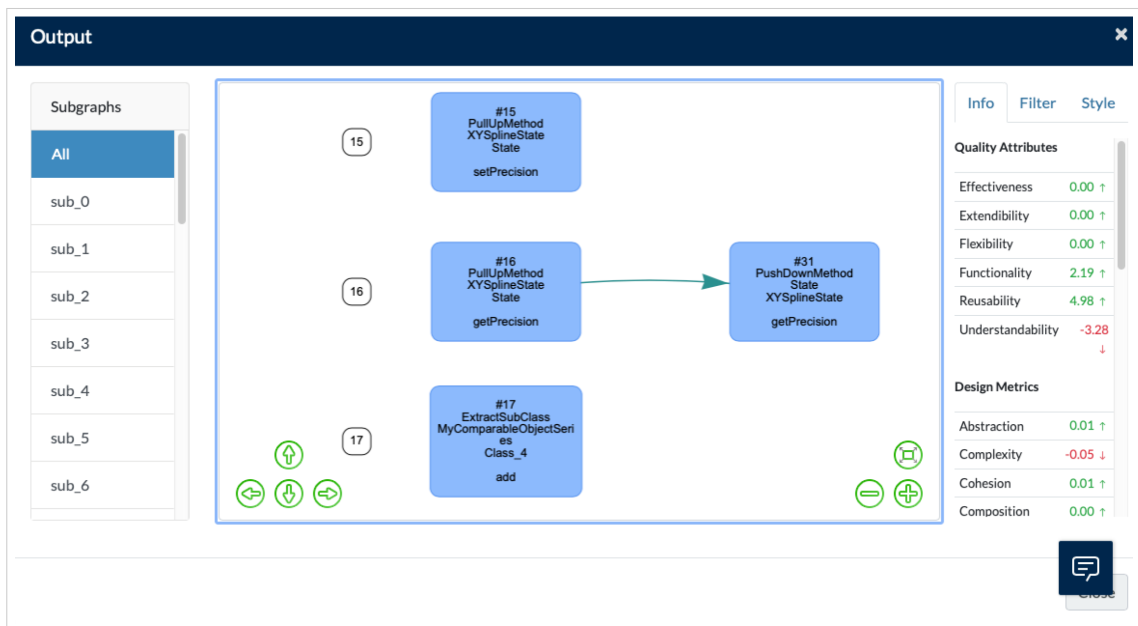

Figure 4.4: DPRef showing detected refactoring dependencies for JFreeChart.

Figure 4.4 shows DPRef's output, which includes the generated set of refactoring graphs and associated information like the impact on quality attributes for each graph

or individual refactoring. Our tool automatically applies the refactoring graphs on the source code. Then, the quality metrics are calculated on the code after applying those refactorings. Developers are then able to change the generated refactoring graphs and apply filters to show, for instance, just the graphs that significantly improve specific quality attributes. DPRef also includes an Eclipse plug-in to execute any refactorings selected by the user on the actual source code. The tool is available as part of our replication package.

4.3 Empirical Study

In this section, we present our research questions, validation methodology, experimental setup, and discuss our findings.

4.3.1 Research Questions

The following research questions guide the evaluation of our refactoring dependency theory:

RQ1: *(Precision)* To what extent are the detected refactoring ordering dependencies correct?

RQ2: *(Relation)* To what extent are refactorings dependent?

RQ3: *(Improvement)* To what extent do non-trivial refactoring graphs improve quality attributes compared to trivial refactoring graphs, *i.e.*, independent refactorings?

We collected data from 9,595 open-source repositories to evaluate the correctness of the detected ordering dependency relationships among refactorings. For each

project, we executed existing refactoring recommendation tools [42, 44] to find relevant refactorings to be applied on the code of those projects. In this study, we use the refactoring recommendations generated by those tools based on their superior performance compared to the state of the art with over 90% of precision, recall, and manual correctness based on large, open-source, and industry projects; the large number of supported refactoring types; and them being publicly available. We describe the parameter settings used by such refactoring recommendation tools in the next section. Finally, we detected dependencies among refactorings and generated refactoring graphs based on Algorithm 1 for all generated refactoring solutions to understand if and how refactorings are applied together rather than in isolation. Finally, we studied and compared the impact of refactoring graphs on different well-defined quality attributes, based on the QMOOD model [45] detailed in Table 4.2.

Table 4.2: QMOOD quality metrics.

QMOOD Metrics	**Definition / Computation**
Reusability	$-0.25 * Coupling + 0.25 * Cohesion + 0.5 * Messaging + 0.5 * DesignSize$
Flexibility	$0.25 * Encapsulation - 0.25 * Coupling + 0.5 * Composition + 0.5 * Polymorphism$
Understandability	$-0.33 * Abstraction + 0.33 * Encapsulation - 0.33 * Coupling + 0.33 * Cohesion - 0.33 * Polymorphism - 0.33 * Complexity - 0.33 * DesignSize$
Functionality	$0.12 * Cohesion + 0.22 * Polymorphism + 0.22 * Messaging + 0.22 * DesignSize + 0.22 * Hierarchies$
Extendibility	$0.5 * Abstraction - 0.5 * Coupling + 0.5 * Inheritance + 0.5 * Polymorphism$
Effectiveness	$0.2 * Abstraction + 0.2 * Encapsulation + 0.2 * Composition + 0.2 * Inheritance + 0.2 * Polymorphism$

4.3.1.1 RQ1: Precision

To answer **RQ1**, we use two methods: automated and manual correctness. We used these complementary methods because the manual evaluation can be error-prone, time-consuming, and not scalable while the automated evaluation may lack insights

and feedback from developers.

For automated correctness, we removed refactorings from *valid* non-trivial graphs and determined whether the removal invalidated the graph. We define a valid graph as a refactoring graph for which all pre-conditions of all refactorings hold (Table 4.1). As defined earlier, if an ordering dependency exists between two refactorings (a head refactoring depends on a tail refactoring), then the head refactoring can only be successfully applied after the tail; removing any tail refactoring should then invalidate at least one pre-condition of a head refactoring. As such, our specific test was to remove one tail refactoring from each valid non-trivial graph with more than two refactorings and then count the number of valid and invalid non-trivial graphs. To answer **RQ1**, we calculated the Rate of Correctness (RC) after removing one tail refactoring from all non-trivial graphs as follows:

$$RC = \frac{\text{\# of Invalid Non-trivial Graphs}}{\text{\# of Non-trivial Graphs}} \tag{4.1}$$

Our assertion is that the refactoring dependency detection algorithm is correct if all non-trivial graphs become invalid if at least one tail refactoring is removed from each, as above.

For the manual evaluation, we asked 27 full-time developers to manually check the correctness of 50 valid non-trivial graphs totaling 233 refactorings from 5 open-source projects[2][3][4][5][6] that contained at least 5K Lines of Code (LOC) and involved significant refactorings in the last 2 years. The graphs are a sequence of refactorings and some of them are large in size. We made sure during the sampling process to use the following criteria to avoid any bias in the selected refactorings for the manual validation: refactoring types, project size, project domain, and locations of

[2]https://github.com/phunware/maas-ads-android-sdk
[3]https://github.com/solita/query-utils
[4]https://github.com/forge/roaster
[5]https://github.com/goobi/goobi-ugh
[6]https://github.com/kongchen/swagger-maven-example

the refactorings (files).

The participants were asked to use our tool to identify refactoring dependencies, assess the correctness of those dependencies, and apply and compile the refactorings. When checking correctness, developers were asked to evaluate each refactoring independently to determine whether each identified dependency was necessary and whether there were any missing dependencies. The participants looked to the generated dependencies graph of the refactorings using our visual representation in the DPRef tool. Then, they reviewed the code before and after applying any selected refactorings. In case of any doubt, the participants could select the refactorings from the graph and they would be automatically executed on the code. Conflicting refactorings may generate errors in the code which may confirm the missing dependencies. In this case, the refactoring is considered as invalid in this exercise. Otherwise, a refactoring for which the set of dependencies was both correct and complete is considered as a valid refactoring. We define a Manual Correctness (MCr) score as:

$$MCr = \frac{\#\text{ of Valid Refactorings}}{\#\text{ of Evaluated Refactorings}} \tag{4.2}$$

4.3.1.2 RQ2: Relation

To answer **RQ2**, we calculated the number of dependencies (edges) and graphs (trivial and non-trivial) for all projects. We also counted the number of refactorings in non-trivial graphs and the most frequently occurring refactoring types in them, as well as the Non-Trivial Rate (NTR) defined as follows:

$$NTR = \frac{\#\text{ of Refactorings in Non-trivial Graphs}}{\#\text{ of Refactorings}} \tag{4.3}$$

These evaluation metrics allow us to understand the extent of refactoring dependencies. Furthermore, we can evaluate the refactoring types that are less commonly applied in isolation and also understand the complexity of the non-trivial graphs

based on their sizes.

4.3.1.3 RQ3: Improvement

To answer **RQ3**, we consider all the trivial and non-trivial graphs to evaluate their impact on the quality attributes and the design metrics from QMOOD. We compared the number of graphs that improve the quality attributes and design metrics from QMOOD [45] for both trivial and non-trivial graphs. We also considered the rates of improvement, in percentage, for all graphs taking into consideration the reusability, flexibility, understandability, functionality, extendibility, and effectiveness quality attributes captured by QMOOD metrics and available in Table 4.2, as well as, basic metrics such as coupling, cohesion, etc. We also calculated a Total Quality Index (TQI), aggregating all the metrics, after normalization, with equal weights into one metric.

These evaluation metrics are useful to understand the impact of collections versus individual refactorings on improving the quality and which quality attributes are more likely to be significantly improved using non-trivial graphs or independent refactorings.

4.3.2 Experimental Settings

We considered a total of 9,595 open-source Java projects to address the above research questions. The selection process limited consideration to projects with $\geq$ 5k LOC and at least 2 collaborators. We also eliminated any duplicate (cloned) projects from consideration. We applied these criteria on the list of one million GitHub projects provided by [121]. We performed this selection process in an attempt to eliminate small projects, such as student projects and small hobby/learner programs, that were not likely to be good candidates for refactorings.

Table 4.3 shows the minimum (min), average, and maximum (max) for the num-

ber of the collaborators, code size (in LOC), # of classes, and # of recommended refactorings generated. The list of subject projects is also available in the replication package along with all results (*e.g.* refactorings, quality metrics, refactoring graphs, etc.).

Table 4.3: Statistics of the subject projects.

Metric	**Min**	**Average**	**Max**
Collaborators	2	2.6	65
Code Size	5.0k	25.9k	4,997.7k
# of Classes	1	931.3	23802
# of Refactorings	150	151.9	200

The total number of refactorings collected from recommendations for these 9,595 projects is almost 1.5 million (1,457,873 refactorings). We used the parameter settings recommended by the authors of the refactoring recommendations tool [44]: Single Point crossover with probability = 0.7, Bit Flip mutation with probability = 0.4, and stopping criterion was set to 100,000 evaluations. We also set the initial population size to 100 and utilized a tournament selection operator with n = 2. The minimum and maximum number of refactorings per solution are limited to 150 and 200, respectively.

For the manual validation of the refactoring dependencies, we recruited 27 full-time developers from our networks, each of whom was unaware of our algorithm. These participants were first asked to fill out a pre-study questionnaire containing six questions. The questionnaire helped to collect background information such as their role within their company, their programming experience, and their familiarity with software refactoring. The list of the pre- and post-study questions of all the questionnaires, the validation data, and the obtained results can be found in the online appendix. Although the vast majority of participants were already familiar with refactoring as part of their jobs and graduate studies, all the participants attended a two-hour lecture on refactoring by two the organizers of the experiments. The

details of the selected participants can be found in Table 4.4, including their years of programming experience, familiarity with refactoring, *etc.*

Table 4.4: Selected Participants.

System	# of Subjects	Prog. Exp. (Years) [Min-Avg-Max]	Avg. Ref. Exp.
Data Set #1	5	[5.5 - 6.0 - 7.5]	High
Data Set #2	5	[5.5 - 7.0 - 8.0]	High
Data Set #3	7	[6.5 - 7.5 - 10.5]	High
Data Set #4	5	[6.0 - 6.5 - 8.5]	High
Data Set #5	5	[6.5 - 7.5 - 9.0]	High

All participants had a minimum of 5.5 years experience and work as active programmers with strong backgrounds in refactoring, Java, and software quality metrics. We divided the participants into 5 data sets where each data set contains 10 samples of valid non-trivial graphs. We selected these samples based on the distribution of the refactoring types and number of refactorings in each graph as described in Figure 4.5.

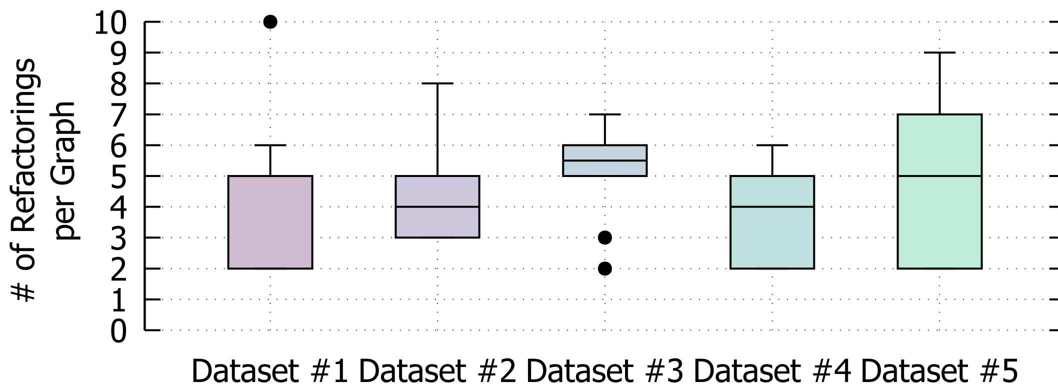

Figure 4.5: Number of refactorings per non-trivial graph in each data set.

Each participant was asked to assess the correctness of the refactoring dependencies and to identify missing dependencies between the refactorings using our DPRef tool. They were asked to execute the sequence of refactoring using our Eclipse plug-in and compile the code after applying the refactorings of each valid non-trivial graph. In addition to evaluating the refactorings, the participants were asked to configure,

run, and interact with the tool on the different systems. We assigned tasks to the participants according to the data sets and developers' experience. Each participant was given a post-study survey. This second survey was more general as it collected the practitioners' opinions and their perception of the importance and relevance of detecting refactoring dependencies along with the usability of DPRef. The samples used in the manual validation study as well as the full details of our pre-study survey results can be found in the replication package.

4.3.3 Results and Discussion

4.3.3.1 Results for RQ1

Figure 4.6 summarizes the distribution of the results of removing one tail refactoring from all non-trivial graphs with more than two refactorings. One important outcome is that all non-trivial graphs become invalid after removing a tail refactoring (RC), which confirms that the proposed algorithm accurately identifies refactoring dependencies. The RC evaluation metric has a value of 1.0 (or 100%) across all the 9,595 projects, *i.e.*, for all non-trivial graphs. The total number of non-trivial graphs is 257,725 with an average of 26.8 per project.

Another interesting result was that our algorithm detected many invalid refactorings among the solutions generated by the tool of Ouni *et al.* [44]. An average of 6.2 invalid non-trivial graphs were identified per project. The main reason is that the crossover operator used to exchange refactorings between solutions did not ensure that refactoring pre-conditions would remain satisfied after the exchange. As discussed in the future work section, the theory proposed in this paper can be integrated into existing refactoring recommendation tools to improve their correctness and contribute to the definition of intelligent change operators (including crossover) for search-based refactoring. Non-trivial graphs that were initially invalid were excluded from the calculation of RC and the removal of refactorings shown in Figure 4.6.

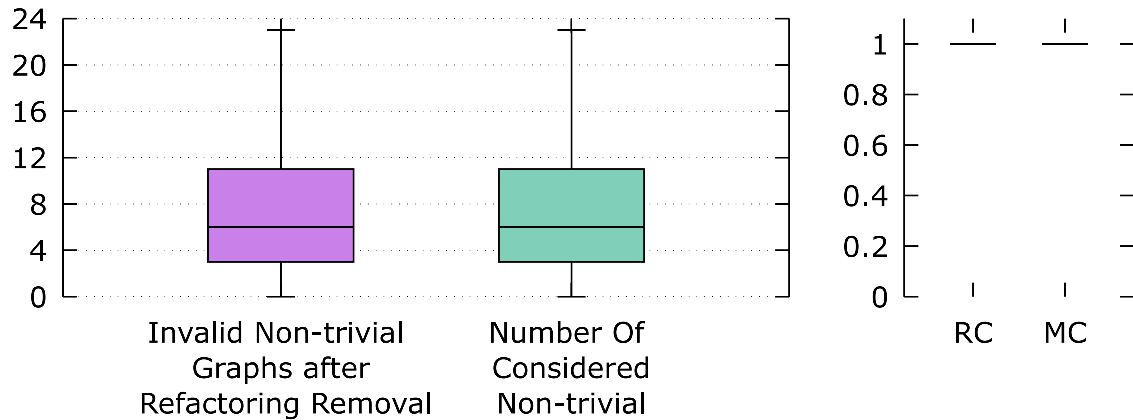

Figure 4.6: Box-plots of refactoring dependency correctness for the 9,595 projects.

For the manual evaluation, all the non-trivial graphs were correctly executed by the participants on the open source projects and they agreed that the dependencies were correctly and completely identified for each refactoring. Thus, the MCr scores were 100% on all the selected data sets as shown in Figure 4.6. Accordingly, the manual correctness results confirm the automated method.

The fact that a refactoring was recommended by Ouni *et al.* [44] and was not applied before by developers does not mean that the recommendation is not correct, but simply that most likely the developers may not have thought about that refactoring and may not have had time to implement it. The manual validation scores of [44] are more than 90% on large scale systems which means that the manual check by developers confirmed that the vast majority of the recommendations are correct and useful. We clarified these observations in the validation section when explaining our choices.

Maturation of automated tools to assist developers with complex tasks such as refactoring is an important gap to fill. The developers that we surveyed also agree with this. All the participants rated the importance of detecting refactoring dependencies as *important* or *rather important*, which confirms the need for tools to detect dependencies in a refactoring sequence. The vast majority of participants (24 out

of 27) also rated the task of manually identifying these dependencies as *difficult* or *very difficult* based on their experiences in using existing refactoring tools. These developers described that the most important motivations of automatically detecting dependencies is to understand how they can fix code smells (typically fixed using a sequence of refactorings) and also to reduce the refactoring effort by applying refactorings incrementally.

We asked also the participants to rate their agreement on a Likert scale from 1 (*complete disagreement*) to 5 (*complete agreement*) with the following question on how important it is to detect refactoring dependencies automatically.

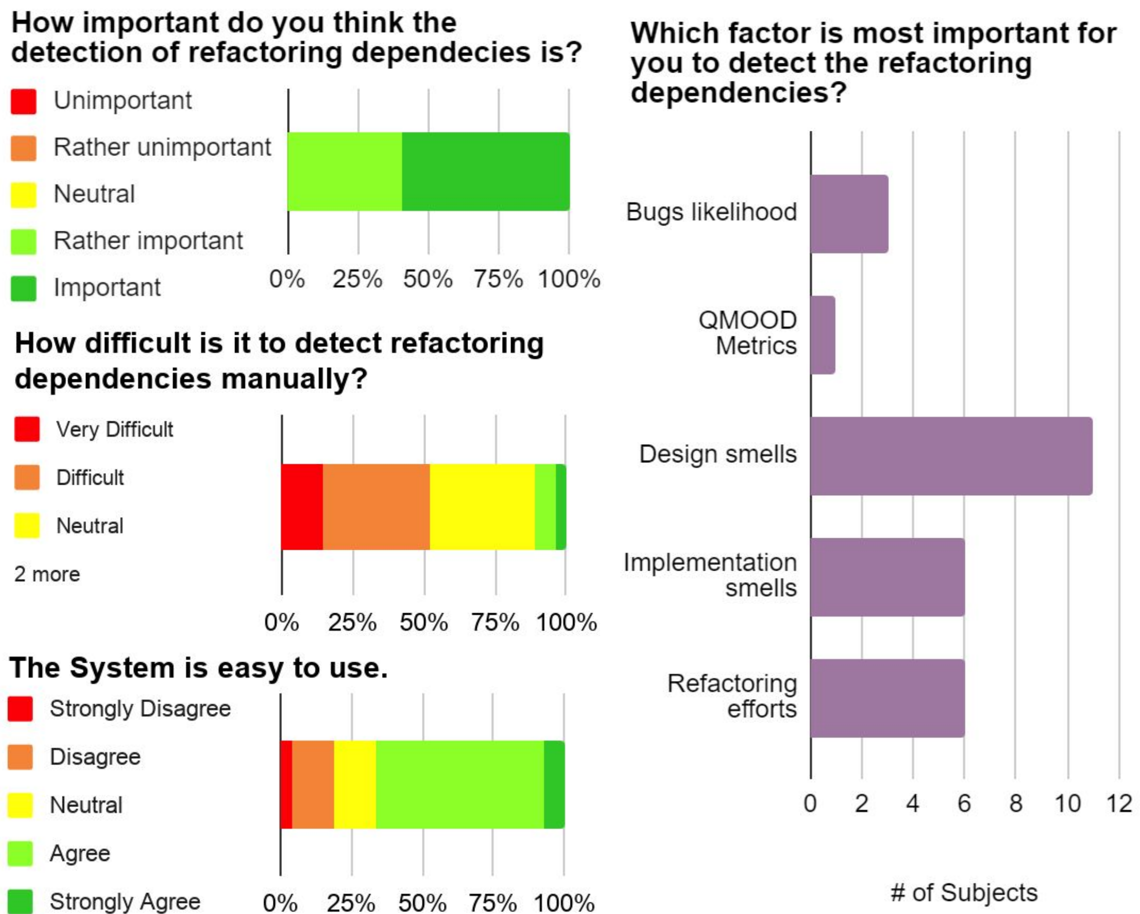

Figure 4.7: Participant Survey.

As shown in Figure 4.7, all the participants rated this feature as *important* or

rather important which confirms the need for tools to detect dependencies in a refactoring sequence. The majority of participants (14 out of 27) also rated the task of manually identifying these dependencies as *difficult/very difficult* based on their experiences in using existing refactoring tools. Figure 4.7 also describes that developers think that the most important motivations of automatically detecting dependencies is to understand how they can fix code smells (typically fixed using a sequence of refactorings) and also to reduce the refactoring effort by applying refactorings in an incremental way. The figure also shows that DPRef was easy to use by at least 18 out of the 27 participants and they were able to understand the dependencies without complications.

Key findings: To answer **RQ1**, the algorithm for detecting refactoring dependencies achieved 100% correctness on all projects.

4.3.3.2 Results for RQ2

Figure 4.8 shows that while truly independent refactorings are more common, the mean NTR shows that more than 40% of all recommended refactorings are part of non-trivial graphs and for some projects, all refactorings are part of a single non-trivial graph (NTR = 1.0). The portion of refactorings that are part of refactoring dependencies is significant.

Across all projects, non-trivial graphs have a mean number of dependencies of almost 40 (recall that this is based on solutions with a mean of roughly 150 refactorings), with nearly 100 dependencies observed for some projects. This indicates a range of connectedness in non-trivial graphs, with the more highly dependent/coupled likely being difficult for developers to understand and having a high risk of generating large numbers of invalid refactorings if they are not applied together. We noticed that the most connected refactoring graphs are more likely to be associated with small projects in which many refactorings are applied to common code locations.

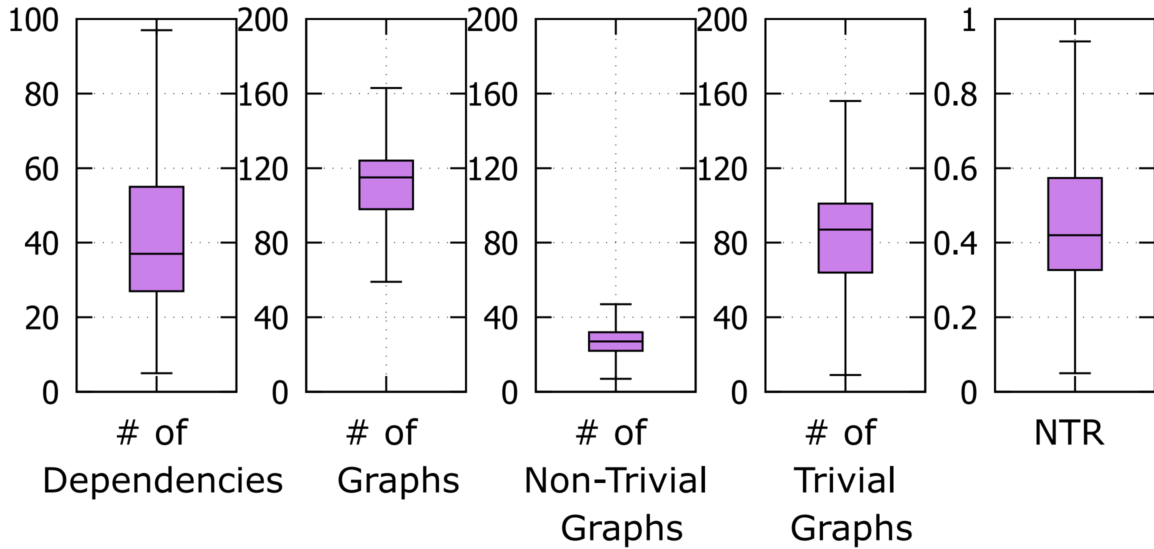

Figure 4.8: Distribution of refactorings in trivial versus non-trivial graphs based on the 9,595 projects.

Comparing the range and distribution of the number of graphs, most projects have a mean of more than 100 graphs per solution in which the number of trivial graphs is greater than non-trivial. However, as shown by NTR, the number of refactorings in non-trivial graphs is very similar or exceeds the number of refactorings in trivial graphs.

Figure 4.9 shows the distribution of the size of the non-trivial graphs; notice that there is a small number of non-trivial graphs including 31+ refactorings. However, the vast majority of non-trivial graphs include 2-5 refactorings. This may confirm that collections of dependent refactorings tend to be small, which can offer flexibility to developers if they want to modify these collections. Projects vary, however, with several including non-trivial graphs with 10-50 refactorings and with one including a graph of 138 refactorings.

Regarding refactoring types, Figure 4.10 shows that the most common refactoring types found in non-trivial graphs are *Extract Class*, *Extract Super Class*, and *Extract Sub Class*. This finding confirms that modifying the hierarchy of code requires a combination of several refactorings and cannot be done with one isolated refactoring.

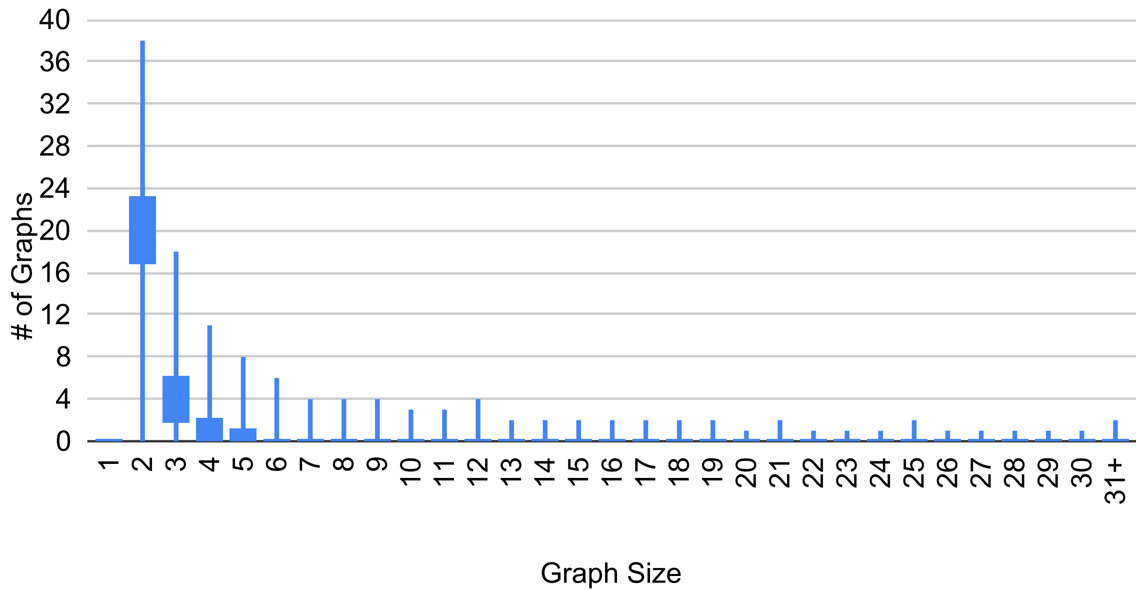

Figure 4.9: Size of non-trivial refactoring graphs in the 9,595 projects.

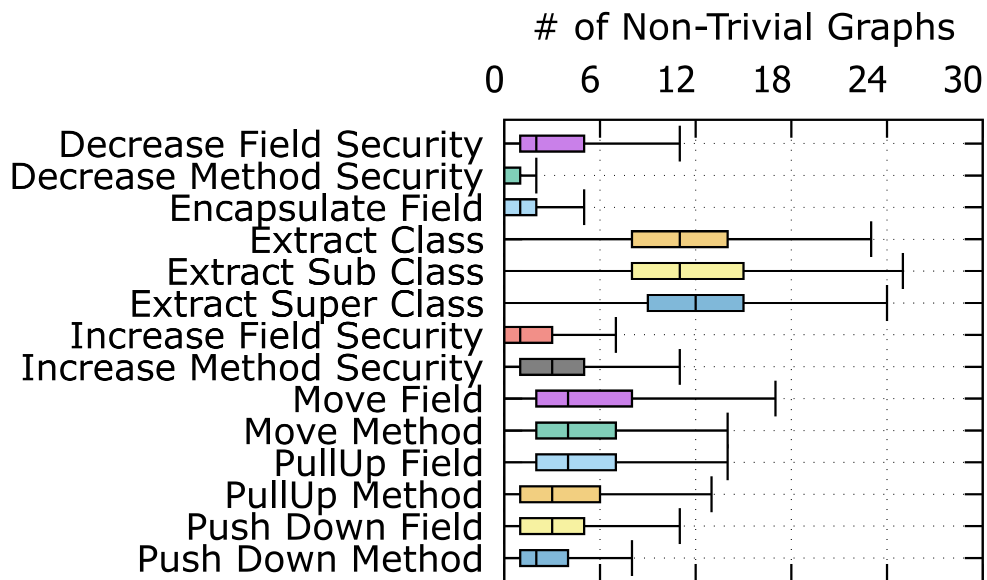

Figure 4.10: Distribution of the refactoring types among non-trivial graphs for the 9,595 projects.

Furthermore, the figure also shows that *Decrease Method Security*, *Encapsulate Field*, and *Increase Field Security* are the least common refactoring types in non-trivial graphs. It most likely indicates that these refactorings can be applied independently without requiring major restructuring effort.

🔍 **Key findings:** To answer **RQ2**, while more refactorings appear in trivial graphs than non-trivial graphs, the difference is not large. The mean value of NTR is 43%, indicating nearly half of all refactorings participate in refactoring dependencies and as such cannot be applied without consideration of other refactorings. Some refactoring types are more likely to be applied with other dependent refactorings than others.

4.3.3.3 Results for RQ3

Figure 4.11 shows how trivial and non-trivial refactoring graphs improve quality attribute and design metrics. The number of trivial graphs that improves each metric is greater than the number of non-trivial graphs that do so. However, the impact of each kind of graph on the improvement of each quality metric can vary considerably. We also analysed the improvements caused by the two kinds of refactoring graphs for a subset of the quality attribute metrics.

Figure 4.12 shows the rate of improvement in % for Effectiveness, Extendibility, Flexibility, Functionality, Reusability, and Understandability. This data shows that, for most projects, the improvement caused by the non-trivial graphs is greater than for trivial graphs. The implication is that non-trivial graphs may be more useful in practice for developers than marginal improvements obtained by individual refactorings. Some metrics, like Extendibility (note the different scale) and Reusability, are more significantly improved using dependent refactorings. Even small changes are significant when considered as aggregate measures across an entire code base. This result is consistent with the fact that Extract [Super/Sub] Class refactorings are more likely to occur within a non-trivial graph and that these refactorings are natural choices for improving the Extendibility and Reusability of software. However, quality metrics may be conflicting, hence the execution of a refactoring could increase some metrics and deteriorate others. The cumulative impact will depend on the types of

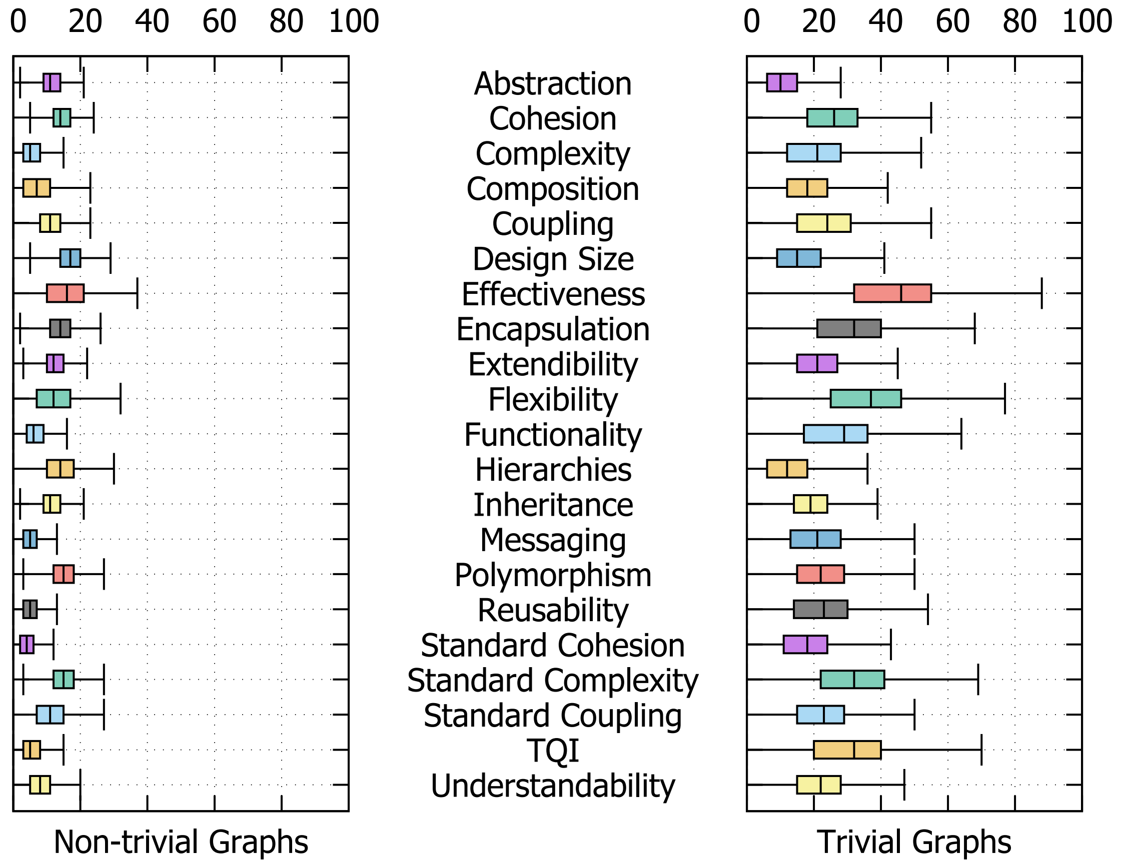

Figure 4.11: The number of graphs that improved the quality metrics.

the refactorings in the non-trivial graph.

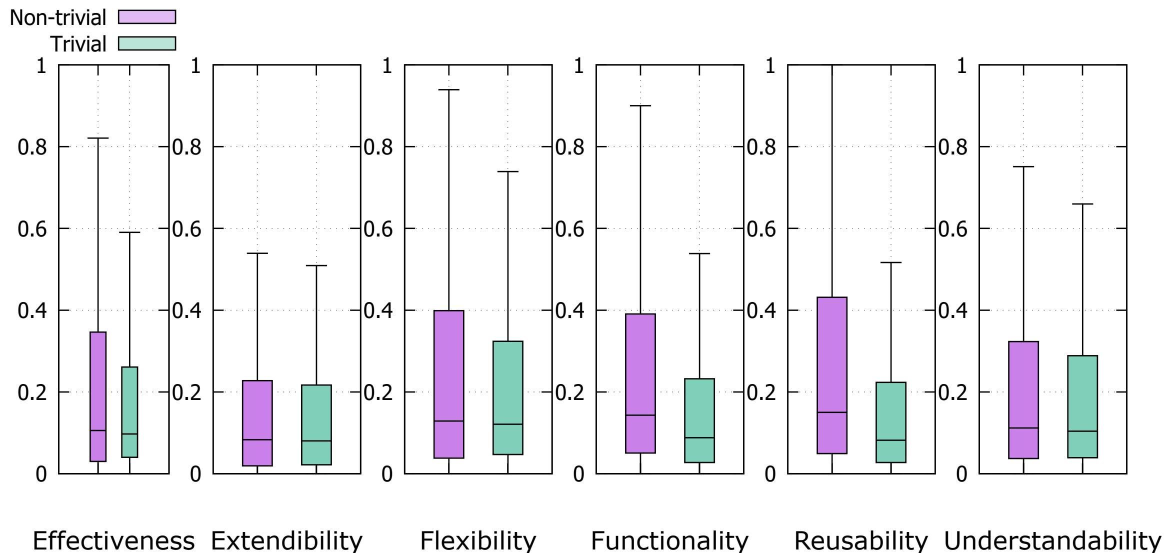

Figure 4.12: Rate of quality improvement (%) for the refactoring graphs per metric.

Key findings: To answer **RQ3**, non-trivial refactoring graphs improve all six quality attribute metrics in our experiments better than independent refactorings. In particular, the improvement from the application of non-trivial graphs over trivial graphs is particularly significant for Functionality and Reusability.

4.4 Threats to Validity

Conclusion validity. We used Design of Experiments (DoE) [122] to mitigate the internal threat related to parameter tuning used in our experiments. DoE is a methodology for systematically applying statistics to experimentation and is one of the most efficient techniques for parameter settings of evolutionary algorithms for the used refactoring recommendation tool. Each parameter has been uniformly discretized in some intervals. Values from each interval have been tested for our application and we chose the best values.

Internal validity. We collected data from a very large number of repositories resulting in about 1.5 million refactorings for 9,595 projects. A possible internal threat is the diversity of these projects in terms of domains, size, *etc.* To mitigate this threat, we used several criteria (*e.g.* more than one contributor per project, over 5K lines of code, *etc.*) to select the projects and eliminate redundant ones or those with small size to avoid considering student projects on GitHub and so on. Also, different tools, with different recommendation strategies might be more or less dominated by non-trivial graphs. Historical refactorings from commit histories could likewise have a different composition. When we conclude that dependencies are common, the source of the source data matters. We make a replication package available including all the collected data that can be used and improved by the community.

Another possible internal threat is the technique used for the manual validation. We did not inform the participants that the non-trivial graphs are all valid based on our tool and we wanted them to confirm that by validating the code after refactoring

and executing it. The invalid graphs can be easily validated as the code will not even compile after refactoring. As described in the pre-study questionnaire, we asked the participants about their experience in interacting with the tool and measured the time that they spent as well. Thus, our questions were not just to ask them about the importance of detecting dependencies, but more on how they could be useful for them in understanding the refactoring recommendations and so on.

Construct validity. The refactorings used in our experiments are generated using an existing refactoring tool [44]. Thus, it is possible that some of them are not relevant (*e.g.* small impact on quality). However, our goal is to evaluate the dependencies among the refactorings independently from their relevance. In addition, our approach can take as input any sequence of refactorings.

External validity. The types of refactorings considered in our experiments may threaten the generalizability of our results. Additionally, our study was limited to the use of specific quality attributes to measure the impact of the application of refactoring graphs. Future replications of this study are necessary to further confirm our findings. Also, the number of participants can be extended in our future work to validate more refactoring dependencies. Moreover, there is no consensus in refactoring studies about the most representative types. Several existing works [123, 124, 125, 126] show that the used refactoring types in this study are the most frequently used by developers.

4.5 Implications and Future Work

Our proposed theory of organizing collections of refactoring instances as a set of refactoring graphs offers several advantages that address the challenges confirmed by developers:

- *Explainability*: each refactoring graph is smaller and more coherent than a long sequence of refactorings. As each can be explained independently, the cognitive

burden on a developer is much lower, *i.e.*, contrast with determining which refactorings scattered across a sequence of dozens or hundreds of refactorings are related.

- *Comparability*: search-based refactoring recommendation tools typically generate multiple recommendations on a Pareto front, leaving developers to choose one. Identifying common elements of different recommendations is simplified by comparing sets of graphs that do not contain the spurious orderings found in sequence representations.

- *Search Efficiency*: search-based refactoring recommendation tools that use genetic algorithms gain new options. Specifically, crossover operations can be more reliable (reducing failures) when using dependency analysis; graphs may also be better genomes for crossover than individual refactorings.

Consequently, there are several directions for future work:

4.5.1 Refactoring Pattern Extraction

One important implication of the proposed refactoring dependency theory is the ability to extract common refactoring patterns by mining software repositories using tools such as RefMiner [16]. These patterns are the common non-trivial graphs that can be extracted on different commits/pull-requests of the same project or multiple projects. Such patterns of non-trivial graphs can be linked to refactoring opportunities such as resolving different types of code smells repeatably. In the future, we plan to use the refactoring dependencies to understand the common refactoring patterns from the history of commits and pull requests of software repositories using existing refactoring detection tools such as RefMiner.

4.5.2 Refactoring Collaborations Between Developers

Studying the collaborations among multiple developers when refactoring code is a promising next step. Refactoring graphs extracted from commit histories can be linked to the authors of those commits. Then, a graph of collaborations among developers can be generated based on the dependencies among the applied refactorings. This can lead to new insights into why and when developers collaborate for refactoring.

4.5.3 Change Operator in Search-based Refactoring

Random selection and application of crossover and mutation when evolving a population of solutions is a challenge in search-based refactoring. Refactoring dependency analysis can be used to avoid destroying good patterns in refactoring solutions and make change operators more intelligent, which can lead to better solutions and faster convergence.

4.5.4 Interactive Refactoring Tool Support

Developers can more easily understand the implications of selecting which refactorings from a recommendation to apply, improving the interactive process and increasing their confidence in the recommendation tool. The only restriction in applying non-trivial refactoring graphs is that a refactoring can only be applied if every other refactoring that it depends on (transitively) is also applied. Thus, invalid refactorings can be detected and highlighted on the fly.

4.6 Conclusion

Although manually applying a collection of refactorings is common practice, existing empirical studies and refactoring recommendation/detection tools treat refac-

torings in isolation. In this contribution, we created a definition for ordering dependencies among refactorings and an algorithm for detecting these dependencies. We also defined refactoring recommendations as sets of refactoring graphs rather than as refactoring sequences, and illustrated these concepts with a web-tool for visualizing refactoring dependencies and sets of refactoring graphs. We also elaborated our research agenda for future work in Section 4.5.

We validated the proposed approach on 1,457,873 refactorings recommended for 9,595 projects. Our results show that the proposed approach achieved 100% in correctly detecting all dependencies among refactorings. Furthermore, we found that 43% of the 1,457,873 recommended refactorings are part of dependent refactoring graphs, which confirms that refactorings are commonly involved in dependent relations and cannot be applied truly independently. These concepts advance a theory for reasoning about refactorings collectively, rather than individually, and offer clear benefits both to developers applying refactoring recommendations (*e.g.* explainability and comparability) and to authors of tools for recommending refactorings (*e.g.* search efficiency and improving the correctness of recommendations).

CHAPTER V

Investigating the Relationships between Architecture and Code Anti-patterns Using Random Forest and Grid Search

In this chapter we present contribution 3: investigating the relationships between architecture and code anti-patterns using random forest and grid search. In this contribution, we begin with Section 5.1 that demonstrates a real instance of the complementary nature of anti-patterns and code anomalies. In Section 5.2, we discuss our machine learning approach. In Section 5.3, we present our experimental design and research questions. We highlight the results of our study in Section 5.4. Then, we describe the threats to validity in Section 5.5. Finally, we conclude in Section 5.6 with a summary of our work and a brief discussion of future work.

5.1 A Motivating Example

Without knowing the commonalities of real-world code anti-patterns and their architecture-level impacts, developers lack knowledge about where and how to apply architecture-level software refactoring. It is easy to see how developers can quickly get lost in hundreds of code anti-patterns to fix without any indication of their impact on important architecture quality issues such as Cyclic Dependencies, SAP Breaker, *etc.*

We provide a motivating example from the BioInfo Project[1] to illustrate the problem of linking architecture anti-patterns and code anomalies. Figure 5.1 shows the anti-patterns and code anomalies detected for the BioInfo project. In Figure 5.1, the yellow dots are packages identified as God Package [127] and the pink dots are packages with other architecture anti-patterns. Following the lines going from the packages, we can observe large numbers of red dots indicating the existence of code anomalies at the same time. The pink lines reveal Cyclic Dependencies among packages. This figure makes it clear that in this system the God Package anti-pattern is highly correlated with many code anomalies. The data shows that 4 of 35 packages are God Packages and 84% of 579 code anomalies are located in these packages.

Table 5.1 lists a partial sample of anti-patterns detected from Apache Cassandra[2]. These six instances of anti-patterns are ranked by the number of files involved and belong to four types of architecture-level anti-patterns: God Package, Scattered Functionality, Cyclic Dependency, and Feature Concentration. Table 5.1 also reveals associated code-level and file-level anomalies. The largest architectural problem involving 73 files also contains the most instances of code anomalies (54) which include: Blob Class, Spaghetti Code, Functional Decomposition, Feature Envy, and Shotgun Surgery.

The two above examples show the highly complementary nature of anti-patterns and code anomalies. These code anomalies can be used as indicators of severe architecture-level quality issues. Thus, they can be grouped together and ranked based on their impact upon the architecture of the system.

Based on the large amount of anti-pattern data that we collected in this work, we will explore a set of research questions including: *What properties and types of code anomalies can serve as indicators of architectural problems? Are there any common patterns? Which types of high-level anti-patterns are* ***not*** *correlated with low-level*

[1]https://github.com/ohnosequences/bioinfo-util
[2]https://cassandra.apache.org/

Table 5.1: Anti-patterns and code anomalies detected from Apache Cassandra.

Architecture-level Anti-pattern	**File-level Anomaly**	# **Files**	# **LOC**	# **Code-level Anomalies**
God Package	1: UI-Schema.java	73	63738	54
God Package	2: UI-ColumnDefinition.java	62	69058	37
Cyclic Dependency	3: UI-SSTableReader.java	53	11813	29
Scattered Functionality	4: UI-Keyspace.java	47	11753	34
Cyclic Dependency	5: UI-Token.java	32	8341	14
Cyclic Dependency	6: UI-DecoratedKey.java	23	5318	12

Files: the number of files influenced by the given architecture-level anti-pattern

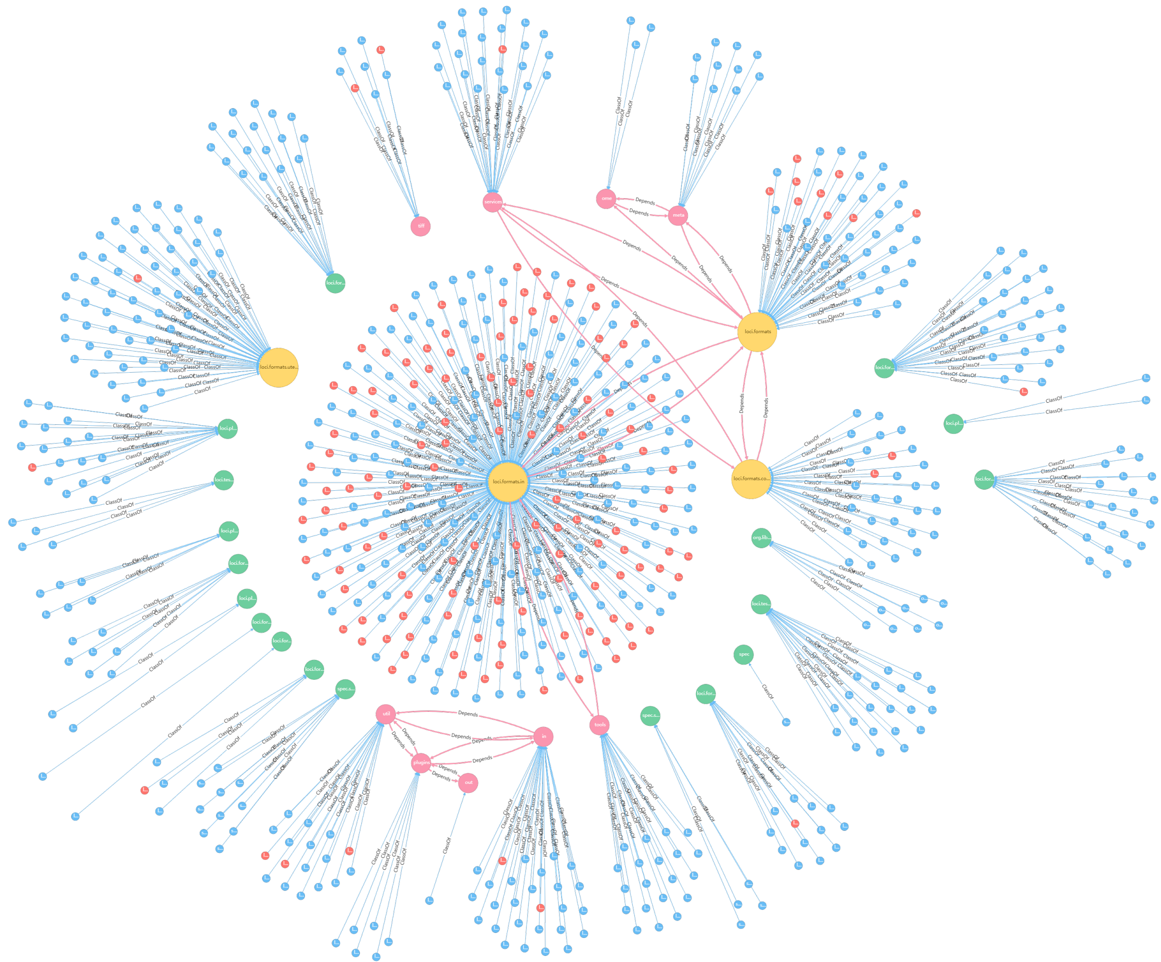

Figure 5.1: Visualization of anti-patterns and code anomalies for the BioInfo Project.

anomalies, and under what circumstances? The answers to these questions will help developers understand where abstraction is needed, where refactoring is needed, and to what extent refactoring can be automated.

5.2 Approach Overview

Our approach includes three main components as illustrated in Figure 5.2. The first component consists of extracting the code smells using a detection tool based on genetic programming [80] and the architecture anti-patterns using another tool named Understand[3]. Once the data is extracted, the second component is executed

[3]https://scitools.com/features/

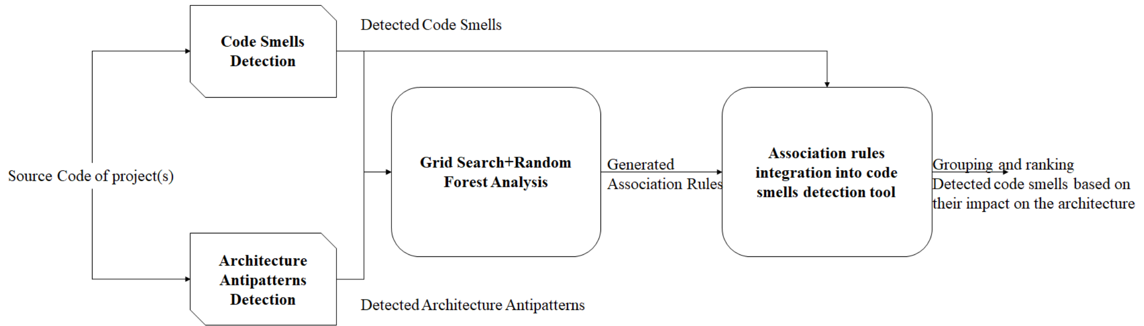

Figure 5.2: An overview of our approach.

to identify the association rules and correlations between the code anomalies and architecture anti-patterns. This part is the most important component of our approach and is described in more detail in the following paragraph. The generated association rules are then integrated into our web-application to visualize and rank the code anomalies based on their impact on design/architecture quality by creating anti-patterns as detailed in Figure 5.3 which was run on the Opencsv[4] project. Then, the developer/architect can decide which code anomalies to fix now and the ones that may be postponed for later since they have not impacted the architecture's quality yet.

In order to explore the potential relationships between code anomalies and architecture anti-patterns, we adapted an ensemble learning algorithm called the Random Forest Algorithm [128]. In this technique, Breiman describes that the algorithm combines predictions from multiple models which in this case is accomplished through a process known as bootstrap aggregation, or more commonly as "bagging", where the algorithm generates a large number of trees independent of each other in parallel using a bootstrap sample of the data [128]. Ultimately, a simple majority vote is taken to generate the prediction model, but Random Forest adds an additional twist to improve the construction of each tree where each node uses the best from a subset of randomly chosen predictors at the current node [128]. Breiman claims

[4]https://opencsv.sourceforge.net/

#		Package	ClassName	DesignSmells	ImplementationSmells ↓
24	A	opencsv au.com.bytecode	CSVWriter	deficient encapsulation	complex conditional complex method empty catch clause long identifier long parameter list long statement
22	A	opencsv au.com.bytecode	CSVParser	deficient encapsulation	complex conditional complex method long identifier long parameter list long statement magic number
20	A	issue3402853 integrationTest	MockUserBean	deficient encapsulation	complex conditional
26	A	opencsv au.com.bytecode	ResultSetHelp...	deficient encapsulation	complex method long parameter list missing default
32	A	opencsv au.com.bytecode	MockResultSet...		complex method long parameter list
17	A	issue3189428 integrationTest	CsvSample		complex method long statement magic number
31	A	opencsv au.com.bytecode	CSVWriterTest	unutilized abstraction insufficient modularization	empty catch clause long statement magic number
21	A	opencsv au.com.bytecode	CSVReader	deficient encapsulation	long parameter list
33	A	opencsv au.com.bytecode	CSVParserTest	unutilized abstraction insufficient modularization	long statement magic number
38	A	opencsv au.com.bytecode	CSVReaderTest	unutilized abstraction insufficient modularization	long statement magic number

Figure 5.3: Our web-app showing the relationships between code anomalies and architecture anti-patterns for the Opencsv project.

this strategy helps Random Forest outperform other classifiers such as discriminant analysis, support vector machines, and neural networks [128]. Further, it prevents over-fitting which is a significant concern with machine learning techniques [128]. It also reduces the likelihood of using imbalanced training data [129] which is a very common problem in real applications.

We formulated the problem as a three-class, classification problem where each of the architecture anti-patterns is a class to be predicted and each of the code anomalies

is a feature for the classification model. We generate the prediction rules for each class from the best tree in the forest. The types of anti-patterns and code anomalies considered in this contribution were discussed in Subsection 2.6.1.

5.3 Experimental Design

In this section, we describe the research questions and our experimental technique.

5.3.1 Research Questions

In this study, we assessed the performance of our approach by finding whether it could predict the existence of architecture anti-patterns from code anomalies. This contribution aims at addressing the following research questions:

RQ1 : What types of code anomalies can serve as indicators of architectural problems?

RQ2 : What are the most severe types of code anomalies causing architecture anti-patterns?

RQ3 : Did practitioners agree with the tool identified relationships between code anomalies and architecture anti-patterns?

5.3.2 Experimental Setup and Formulae

5.3.2.1 Metrics for RQ1

We used three metrics to evaluate the random forest's performance on our data set: Precision, Recall, and F_1-score. For Class A in a 3-class classification, we compute the following metrics:

- *T(AA)* : Truly predicted class A as class A
- *F(AB)* : Falsely predicted class A as class B
- *F(AC)* : Falsely predicted class A as class C

The same definition applies to *F(BA)*, *F(CA)*, *etc.*

$$Precision = \frac{T(AA)}{T(AA) + F(AB) + F(AC)} \tag{5.1}$$

$$Recall = \frac{T(AA)}{T(AA) + F(BA) + F(CA)} \tag{5.2}$$

$$F_1 - score = \frac{2T(AA)}{X} \tag{5.3}$$

$$X = 2T(AA) + F(AB) + F(BA) + F(AC) + F(CA) \tag{5.4}$$

We compute precision and recall since accuracy alone can be quite misleading especially in cases where the distribution of the training data is uneven over all classes. Given all the predicted labels for a given class C, **Precision** computes how many instances were correctly predicted. **Recall** is only focused on True Positives. For all instances that should have a label C (truly labeled as C), **Recall** calculates how many of these were correctly predicted as class C. F_1**-score** is the weighted average of **Precision** and **Recall**.

5.3.2.2 Metrics for RQ2

We defined the metric *Contribution* to measure the importance of each class level code anomaly to predict each architecture-level anti-pattern. The *Number of Times Code Anomaly Appeared (NTCL)* is defined as the number of times code anomaly $i, i \in$

$\{0, 1, 2, \ldots, 15\}$ appeared in the decision path for predicting architecture-level anti-pattern $j, j \in \{0, 1, 2\}$ and the *Total Number of Test Samples (TNTS)* is defined as the total number of test samples predicted as class j, and the contribution or importance of class level code anomaly (i) for predicting architecture-level anti-pattern (j) is then as follows:

$$Contribution(i, j) = \frac{NTCL(i)}{TNTS(j)} \tag{5.5}$$

5.3.2.3 Metrics for RQ3

We asked 30 software practitioners who are experienced in software quality to investigate the results of our classifier. These developers were instructed for two hours on anti-patterns, and given a thorough explanation of the classifier and sample of its results. The developers were then given a set of 22 classifications from the classifier on three different, open-source, Java software projects: Glotaran [130], WordPress [131], and the Meta Protean Analyzer (MPA) [132]. These practitioners were given the complete project source code in addition to the packages and classes of the code directly effected by the discovered classifications. In addition, they were supplied with the definitions of the architecture anti-patterns and class code anomalies for reference and review which can be found summarized in Table 2.1. We equally distributed the developers on the three systems based on the number of anti-pattern instances to inspect. The participants have a good level of experience as software engineers and they are part-time graduate students in a master in software engineering program. They were recruited from a graduate Software Quality Assurance course after attending extensive lectures on refactoring and anti-patterns.

We provided to the developers a list of 5 architecture anti-patterns that were linked to a group of code anomalies as identified by our tool and listed in Table 5.8 to investigate and analyze the results. Then, we calculated the following three metrics:

Architecture Correctness measures the correctness of the predicted architecture anti-pattern for the test samples inspected by the interviewees. The developers measured the architecture correctness on a scale from 1 to 5 where 1 = *strongly reject* as not an architecture anti-pattern and 5 = *strongly accept* as an architecture anti-pattern. This measure will confirm if our approach actually predicts architecture anti-patterns from code anomalies.

Relational Correctness measures how related the code anomalies are to the predicted architecture anti-pattern. This was measured through a binary choice (*Yes* = related or *No* = unrelated) in addition to a short written explanation as to why the developer chose the option. This measure will enable the individual assessment of the impact of one type of code anomaly on the architecture anti-patterns.

Severity evaluates how harmful the predicted architecture anti-pattern is on the quality of the system based on the type and also the generated code anomalies. The participants were asked to measure the severity of the architecture anti-patterns on a scale from 1 to 5 where 1 = *minor severity* and 5 = *very high severity* on the system.

5.3.2.4 Data Collection

In order to build a model capable of predicting the existence of architecture anti-patterns for a variety of projects, we have analyzed 113 open-source Java projects and have detected the instances of 16 different code anomalies and 3 architecture

anti-patterns. The raw data contains information on the project ID and packages belonging to each project in addition to the classes belonging to each package. We used this information to reformat the data into a training set for a multi-label classification problem where there are multiple labels and each can be present or absent in the predicted set of labels. Then, we transformed the problem into a three-class classification problem. After the pre-processing, there are 305 samples in the data set. Each sample corresponds to one package and contains 16 input values as the number of instances of code anomalies in the package and 3 label values as the instances of architecture anti-patterns.

5.3.2.5 Parameter Tuning

Based on the size of our training set, we tuned only the parameters that control the performance in terms of the prediction. The three main parameters that control the prediction power in random forest are: (1) the *Maximum Number of Features (MNF)* Random Forest is allowed to try in an individual tree, (2) the *Number of Trees in the Forest (NT)*, and (3) the *Minimum Sample Leaf Size (MSL)*.

In our study, we selected the parameters resulting in the best performance for our training data. We compared the best parameter selection for each parameter tuning method and the performance of the algorithm using each recommended parameter selection as reported in Table 5.2. The results of the parameter tuning are presented in Table 5.3. Table 5.2 shows that although Random Search is significantly faster, the parameters discovered by Grid Search lead us to more distinguishable trees to predict the existence of the architecture anti-patterns.

Table 5.2: Parameter Tuning Results

Search Method	**Time (s)**	**MVS**	**MNF**	**NT**	**MSL**
Random Search	69.18	0.643	9	600	4
Grid Search	1107.56	0.946	6	900	0.1

Table 5.3: Evaluation Results

Class	**Precision**	**Recall**	F_1**-score**
Cyclic Dependencies	0.88	0.60	0.71
SAP Breaker	0.54	0.90	0.68
God Package	0.91	0.63	0.75
Weighted Average	0.78	0.71	0.71

5.4 Experimental Results

5.4.1 Results for RQ1

We trained a decision tree to formulate the correlation between code anomalies and architecture anti-patterns. Figure 5.4 illustrates part of the tree presenting the correlation between code anomalies and the Cyclic Dependency anti-pattern at the architecture-level. Visiting the nodes starting from the root to the leaves represents the correlation between the code anomalies presented in the path and the architecture anti-pattern in the leaf as a rule to predict the presence of that specific architecture anti-pattern.

Figure 5.5 and Figure 5.6 illustrate part of the tree presenting the correlation between code anomalies and both the God Package and SAP Breaker anti-patterns respectively. The equivalent rules to predict each architecture-level anti-pattern are presented in Table 5.4. The higher the feature in the tree the more important the value of that feature is to predict a test data point as the leaf (class). One of our observations from all these trees was that they have overlapping features, especially at the top of the tree. However, having a closer look at Figure 5.7, this is not unexpected. As illustrated in Figure 5.7, the architectural anti-patterns happen together. For example, the second bar means about 500 times in the entire data set two architecture-level anti-patterns are happening together for a given set of code anomalies. Considering

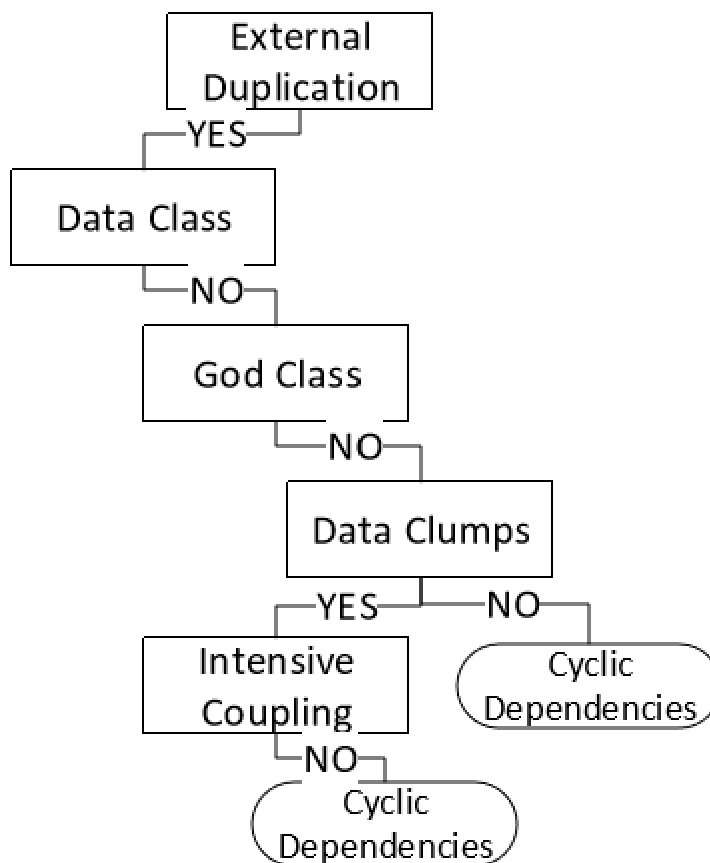

Figure 5.4: Predicting **Cyclic Dependencies** architecture-level anti-patterns from class-level code anomalies.

the total size of the data set we worked with (1,058 data points), this case is roughly happening in 48% of the data which influences the extracted rule for each individual architecture-level anti-pattern even when we transform this problem into a multi-class classification.

In Table 5.3, we have reported the Precision, Recall, and F_1-score for each class along with the weighted average. Using the best parameters recommended by Grid Search (Table 5.2), the accuracy of the best tree is 87%. Precision is a measure of how accurate a specific architecture-level anti-pattern has been predicted. God Package architecture anti-pattern has the highest precision of 91%, while SAP Breaker

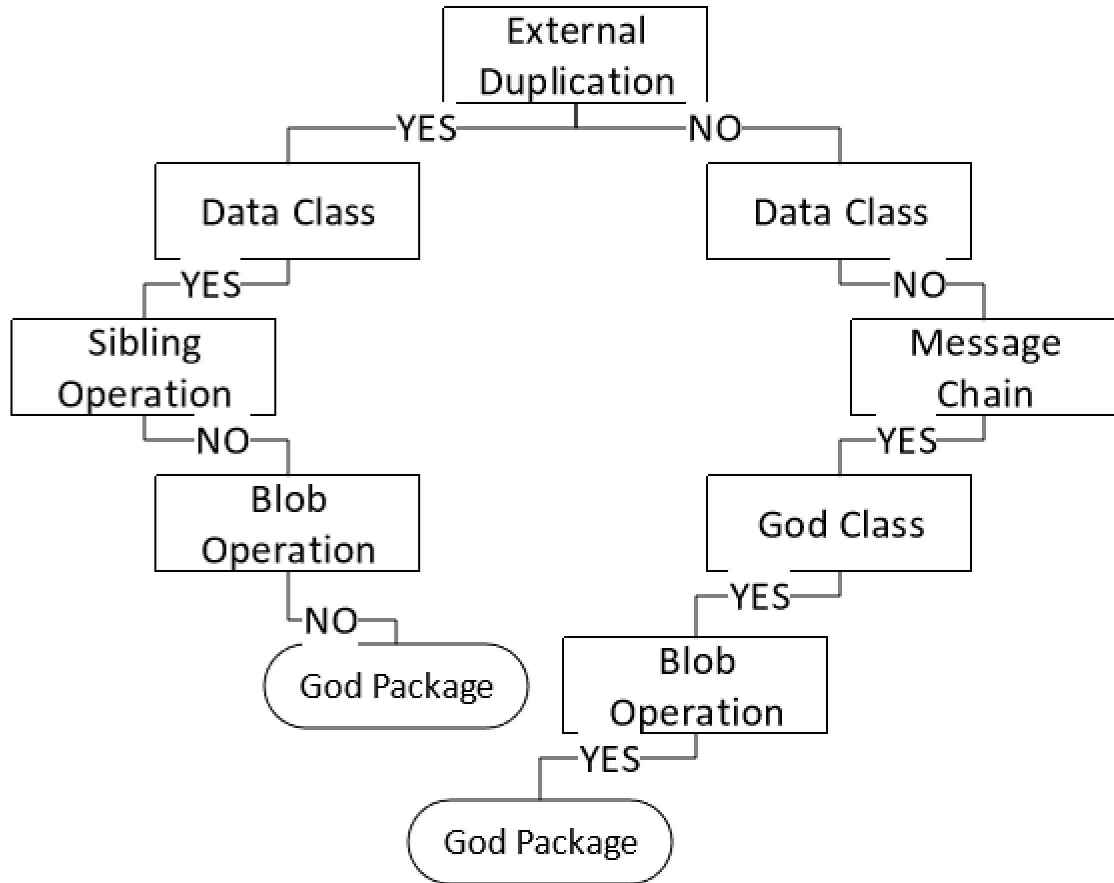

Figure 5.5: Predicting **God Package** architecture-level anti-patterns from class-level code anomalies.

architecture anti-pattern has the lowest precision of 54%. Looking at Figure 5.8 explains the reason for low precision for the SAP Breaker architecture anti-pattern. There are less than 240 data points (less than 22% of the data) with a SAP Breaker anti-pattern occurrence which is significantly lower than the other two architecture-level anti-patterns we trained the model on.

Recall, for each architecture anti-pattern, is the fraction of data points in the test set where we correctly predicted the data point as that specific architecture anti-pattern, out of all of the cases where the true label of the data point is that specific architecture anti-pattern. SAP Breaker has the highest recall of 90%, while the Cyclic Dependency anti-pattern has the lowest recall of 60%. While only less than 22% of

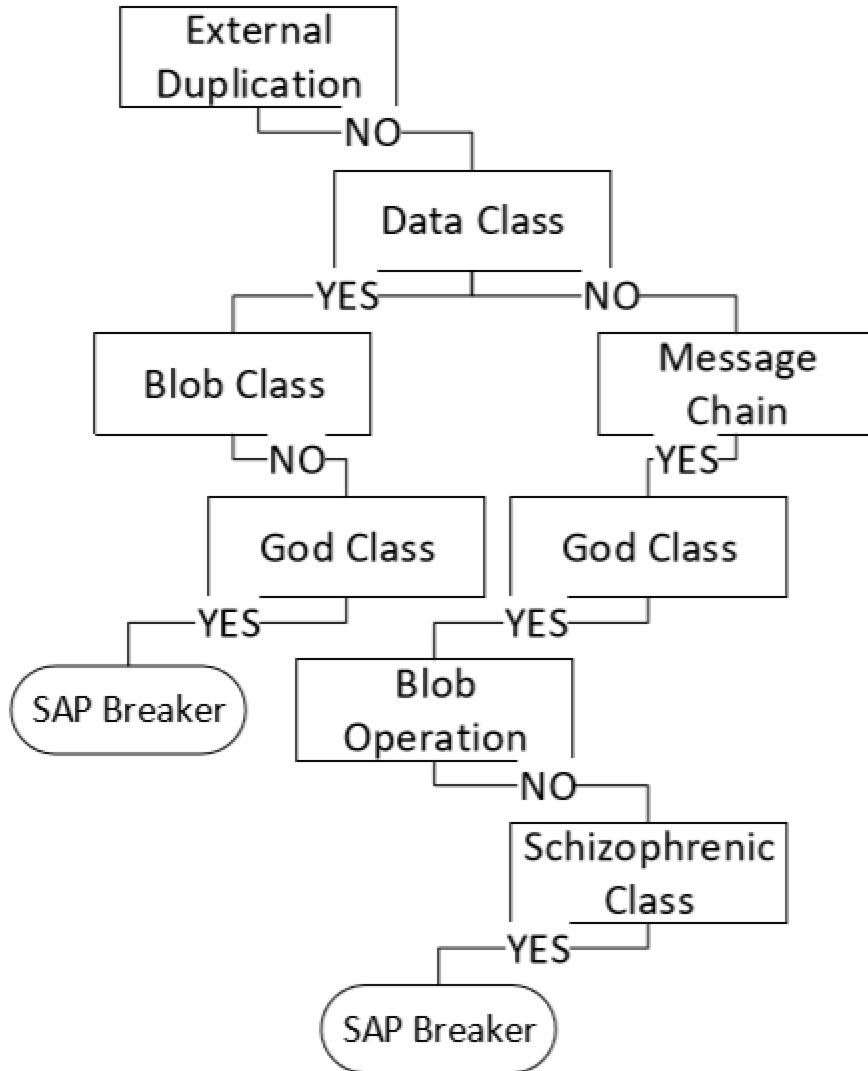

Figure 5.6: Predicting **SAP Breaker** architecture-level anti-patterns from class-level code anomalies.

the data is labelled as SAP Breaker, the algorithm can successfully recognize the data points that truly are labelled as SAP Breaker and predict the correct class for them. However, we have more data in each of the Dense Structure and Cyclic Dependency anti-patterns, and the reason for lower recall for these two classes is that they share the same set of features (code anomalies) and it makes it more challenging for the model to distinguish unseen data accurately between these two classes.

To conclude, the outcomes of RQ1 confirm the strong correlation between the 3

Table 5.4: Detection rules for three architecture-level anti-patterns.

Architecture-level Anti-pattern	**Prediction rule**
Cyclic Dependencies	[(ExternalDuplication) & (DataClass) & (not(GodClass)) & (not(DataClumps))] OR [(ExternalDuplication) & (DataClass) & (not(GodClass)) & (DataClumps) & (not(IntensiveCoupling))]
SAP Breaker	[(not(ExternalDuplication)) & (DataClass) & (not(BlobClass)) & (GodClass)] OR [(not(ExternalDuplication) & (not(DataClass)) & (messageChain) & (GodClass) & (not(BlobOperation)) & (SchizophrenicClass)]
God Package	[(ExternalDuplication) & (DataClass) & (not(SiblingOperation)) & (not(BlobOperation))] OR [(not(ExternalDuplication) & (not(DataClass)) & (messageChain) & (GodClass) & (BlobOperation))]

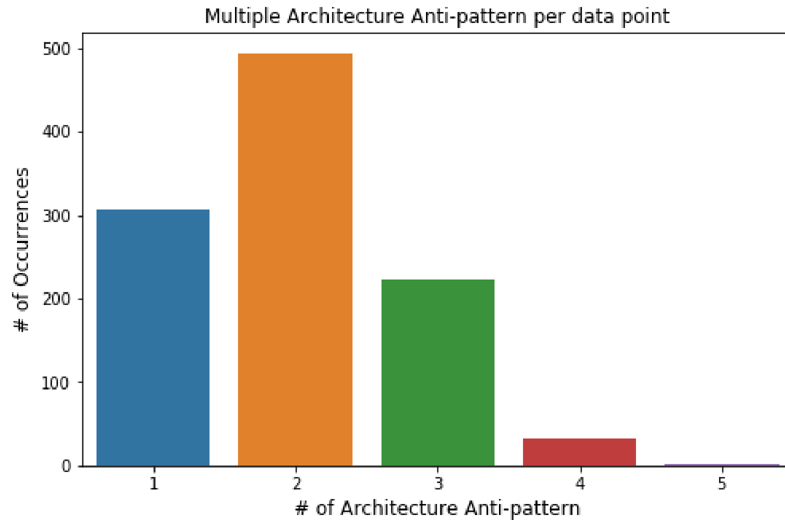

Figure 5.7: Each bar represents the number of occurrences of architecture-level anti-patterns that exist for each set of code anomalies. For example, the second bar indicates for almost 500 of the input data (a set of 16 code anomalies) two architecture-level anti-patterns exist at the same time.

types of architecture anti-patterns and code anomalies where all instances of architecture anti-patterns are associated with different types of code anomalies.

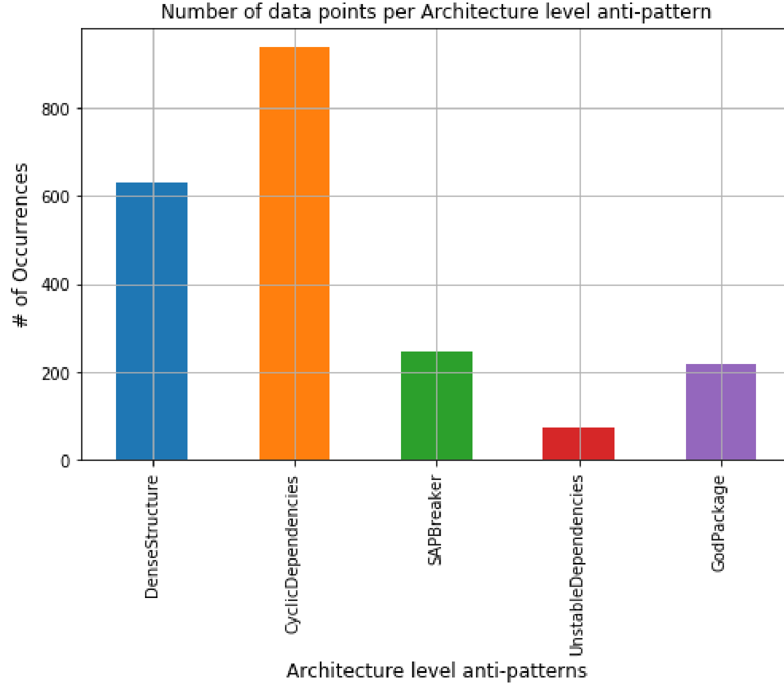

Figure 5.8: Each bar represents how many of the data points have that specific architecture anti-pattern independent from other architecture anti-patterns that may exist in those data points.

5.4.2 Results for RQ2

We investigated the contribution of the code anomalies in each class of architecture anti-patterns to conclude what are the most severe code anomalies causing each type of architecture anti-pattern. We analyzed the decision path for each sample in the test-set to find out what are the most visited code anomalies when predicting architecture anti-patterns. We computed the contribution of each type of code anomalies to predict each type of architecture anti-pattern using the *Contribution* metric introduced in Subsection 5.3.2.

Table 5.5: Most important features for the classification of the **SAP Breaker** Anti-pattern

Feature	**ExtDup**	**DataClass**	**GodC**	**BlobOp**	**SchizC**
Contribution	1.0	1.0	0.92	0.88	0.80

Table 5.6: Most important features for the classification of the **Cyclic Dependencies** Anti-pattern

Feature	**ExtDup**	**DataClass**	**GodC**	**BlobOp**	**SchizC**
Contribution	1.0	1.0	0.90	0.72	0.77

Table 5.7: Most important features for the classification of the **God Package** Anti-pattern

Feature	**ExtDup**	**DataClass**	**GodC**	**BlobOp**	**SchizC**
Contribution	1.0	1.0	0.95	0.95	0.90

The most important code anomalies for the classification of the SAP Breaker anti-pattern, as seen in Table 5.5, are External Duplication (100%) and Data Class (100%) code anomalies with God Class (92%), Blob Operation (88%), and Schizophrenic Class (80%) also playing a significant part of the classification. The rules generated for the classification as seen in Table 5.4 and Figure 5.6 show the reason for the importance of these code anomalies for SAP Breaker. External Duplication and Data Class are the the first checked code anomalies. This is because you would not expect an External Duplication code anomaly as this is caused by duplication between unrelated modules of the system whereas with SAP Breaker we are discussing packages that are inherently related. Data class is also one of the highest rated code anomalies for classification as it is the fork in the decision tree. When a Data Class code anomaly is involved we do not see involvement of a Blob Class, but we do find God class involvement (always as seen in Figure 5.6). This is likely because God Classes are those the concentrate functionality from several unrelated modules which is what is inherently occurring when stability is low since the shared subsystem is not abstracted appropriately. Blob Classes are not involved despite being the result of strongly coupled classes because it implies a significant large size and complexity not generally seen when one discusses classes that are involved in abstraction inheritance relationships. Schizophrenic Classes make sense as being key for classification since

they result from capturing and concentrating multiple abstractions together.

Meanwhile, the most important code anomalies for the classification of Cyclic Dependencies, as seen in Table 5.6, are External Duplication (100%) and Data Class (100%) code anomalies with God Class (90%), Schizophrenic Class (77%), and Blob Operation (72%) also playing a significant part. Here, External Duplication is always present in a Cyclic Dependency anti-pattern as seen in the classification prediction rules in Table 5.4 and Figure 5.4. External Duplication is always involved since in a Cyclic Dependency you have multiple packages involved in a relationship which implies a large degree of duplication between unrelated components. Data Class anomalies are important for classification as they are always not involved given that they are modules without complex functionality and lack encapsulation which is the opposite of the situation with the packages in the Cyclic Dependency anti-pattern. God Class code anomalies while an important step of the classifier are not involved since they imply a concentration of functionality whereas a Cyclic Dependency is a cyclic relationship amongst packages.

Lastly, the most important code anomalies for the classification of God Package anti-patterns, as seen in Table 5.7, are also External Duplication (100%) and Data Class (100%) code anomalies with Blob Operation (95%), God Class (95%), and Schizophrenic Class (90%) also playing a significant part. Again External Duplication and Data Class play a significant role in the classification as seen in Table 5.4 and Figure 5.5. This is likely since a God Package anti-pattern is a pattern that knows too much and tries to perform too much functionality. This can result from External Duplication between unrelated modules in the package and/or large amounts of data holders without the complex functionality. However, God Class are nearly always present as it logically makes sense that a module that hordes data and functionality would contribute to a package that does that same.

Figure 5.9 shows the distribution of the code anomalies per architecture anti-

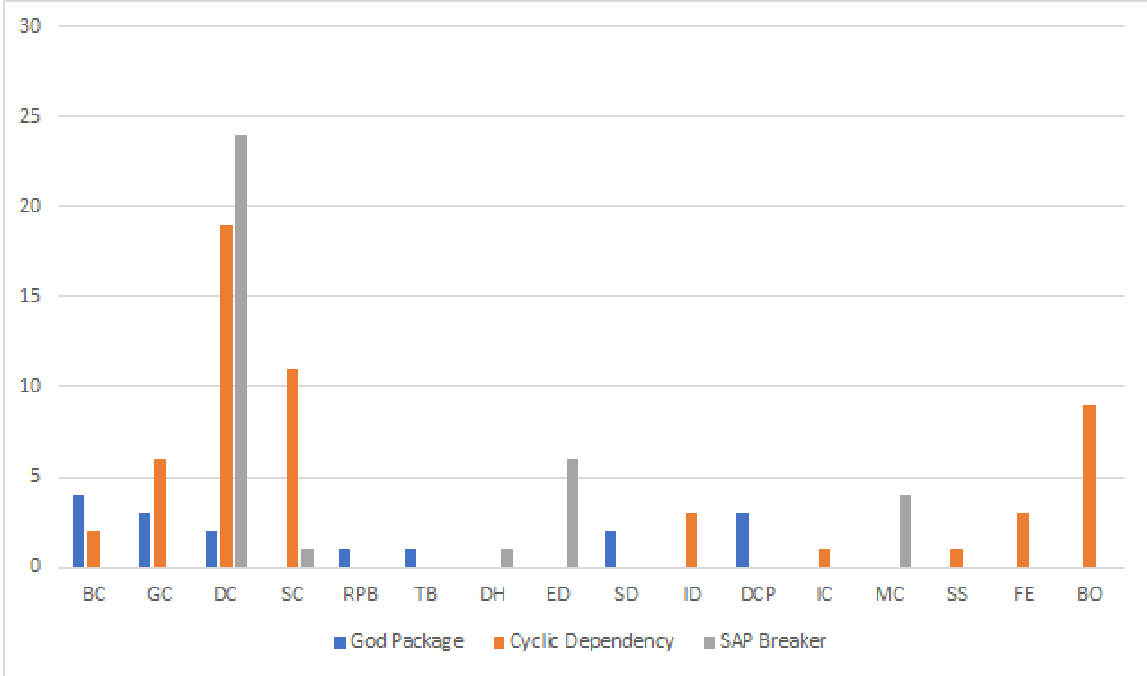

Figure 5.9: Distribution of code anomalies per architecture anti-pattern type.

pattern on all the systems of our experiments. It is clear that the God Package generated various types of code anomalies, nearly all of which involve the duplication of data or functionality such as Blob Class, God Class, Data Class, Sibling Duplication, and Data Clumps; while the SAP breaker created the highest number of code anomalies. Thus, we may consider that these two types of architecture anti-patterns are the most severe ones in terms of generating code anomalies. For SAP Breaker, we see Data Class, External Duplication, and Message Chain as the most frequent anomalies which follows from an anti-pattern that breaks abstraction stability through too tightly coupled modules. Lastly, the Cyclic Dependency anti-pattern has high frequencies of Data Class, Blob Operation, and God Class code anomalies which follows from an anti-pattern that results from cyclic dependencies amongst packages thereby having duplicated and extremely centralized functionality in modules.

5.4.3 Results for RQ3

Table 5.8 summarizes our results. The results are organized by software project and architecture anti-patterns as detected by the results of our classifier. 29 out of the 30 participants returned results. From those 29 practitioners, we had a total of 174 evaluations of the classified architecture anti-patterns. We recorded the time spent by the participants for each evaluation and removed responses where participants spent only 5 minutes or less in order to remove poor evaluations from the response pool. Only 7 responses were removed leaving 167 useful evaluations. The individual evaluations were summarized into Table 5.8. The detailed evaluations per each of the 22 classifications data is included as an appendix in our replication package[5] for download.

The architecture anti-pattern correctness results as shown in Table 5.8, are displayed with the mean, median, and mode in order to describe the true central tendency of the metric. The developers rated all architecture anti-patterns in the 3 (*neutral*) to 4 (*weakly accept*) range with median values all at 4. This indicates that the our survey participants agreed that there was likely an architecture anti-pattern present and indicates that these architecture anti-patterns exist and are detectable in real project code.

The relational correctness as shown in Table 5.8 shows similar weak indications. However, it seems that the majority of participants thought there was compelling evidence that the Glotaran software project had a SAP Breaker architecture anti-pattern which was related to the Data Class (75%) code anomaly. This is opposed to the SAP Breaker anti-pattern in the MPA software project which had fewer Data Class anomalies and a Schizophrenic Class code anomaly where the participants evaluated the relationship at 40% correctness. Further, the Glotaran Cyclic Dependency (61%) anti-pattern involved God Class, Data Class, Schizophrenic Class, External Du-

[5]https://sites.google.com/umich.edu/ieee2022-anti-patterns/home

Table 5.8: Results from the manual validation of the classifier.

Software	Architecture Anti-Pattern	Severity Rating *1 = minor, 5 = very high*			N	Class Code Anomaly																Relational Corr.	Architecture Correctness *1 = strongly reject, 5 = strongly accept*		
		Mean	Median	Mode		BC	GC	DC	SC	RPB	TB	DH	ED	SD	ID	DCP	IC	MC	SS	FE	BO	Yes / Total	Mean	Median	Mode
Glotaran	Cyclic Dependency	3.21	3	3	38	0	3	1	2	0	0	0	1	0	0	0	0	1	0	5	2	23/37 (62%)	3.37	4	5
Glotaran	SAP Breaker	3.42	3	3	24	0	0	20	0	0	0	0	0	0	0	0	0	0	0	0	0	18/24 (75%)	3.96	4	5
WordPress	God Package	3.61	4	4	49	0	0	2	0	0	0	0	0	0	0	0	0	0	0	0	0	26/49 (53%)	4	4	4
MPA	Cyclic Dependency	3.30	3	4	46	2	6	19	11	0	0	0	0	0	3	0	1	0	0	3	9	27/46 (59%)	3.5	4	5
MPA	SAP Breaker	2.9	3	2	45	0	0	4	1	0	0	0	0	0	0	0	0	0	0	0	0	4/10 (40%)	3.7	4	4

plication, Message Chain, Feature Envy, and Blob Operation code anomalies whereas the MPA Cyclic Dependency (59%) anti-pattern involved a significantly larger number of Data Class, Schizophrenic Class, and Blob Operation code anomalies. This may indicate that the anti-patterns may relate more to the severity of underlying code anomalies rather than simply some combination of specific code anomalies given that the number of code anomalies and the types of code anomalies were different in each case. The WordPress God Package (53%) anti-pattern resulted only from two Data Class code anomalies which gives furtherance to the suggestion that the severity caused by the code anomalies, two in this case, is significant enough to result in the symptoms of an architecture anti-pattern. Some participants' comments from their evaluation of the relational correctness evaluation are included below:

- For the Cyclic Dependency anti-pattern when looking at the Glotaran project one participant wrote: *"Yes I believe that the following code anomalies (God Class, Message Chains, Feature Envy and Blob Operation) are connected to the suggested Cyclic Dependency anti-pattern. All of these code anomalies suggest high coupling and high dependency of classes upon other classes that can be inside or outside the package which increases the probability of getting a Cyclic Dependency anti-pattern."*

- For the Cyclic Dependency anti-pattern when looking at the MAP project: *"Yes, the code consists of a few methods like stopTransaction and getGraphDatabaseService which are dummy data holders and depends on other methods. addTaxonomy, addPeptides, addProtein are large and complex leading to Blob Operations. The code also contains Intensive Coupling, as God Class and Blob Class leads to Cyclic Dependencies in the code."*

The developers also evaluated the severity of the detected architecture anti-patterns. It is interesting that the WordPress God Package anti-pattern was rated with an aver-

age score of 3.61 mean and 4 median (*moderate severity*) and had the highest average architecture correctness score. One possibility is that the easier the architecture anti-pattern is to detect the more severe it likely will be rated. Alternatively, the severity of the architecture anti-pattern is tied to the severity of the underlying code anomalies and is a property derived from them rather than being inherent to the architecture anti-pattern. This would explain the differences in severity between the Glotaran SAP Breaker and MPA SAP Breaker.

To conclude, the manual evaluation by the developers of the classification of our architecture anti-patterns based on code anomalies confirm the correctness of our approach and its outcomes. Although further research needs to be performed to determine if these results will hold for other projects, anti-patterns, and code anomalies.

5.5 Threats to Validity

In our experiments, construct validity threats are related to the absence of similar work that uses machine learning to predict architecture anti-patterns based on code anomalies. We did not compare the outcomes of our study to existing work based on the same types of quality issues. Furthermore, parameter tuning is another challenge of our approach. We used one of the most efficient and popular approaches for the parameter setting of the learning algorithms which is Design of Experiments (DOE) [122, 133]. Each parameter has been uniformly discretized in some intervals. Values from each interval have been tested for our application. Finally, we picked the best values for all parameters. Hence, a reasonable set of parameter values have been experimented.

Internal threats to validity are related to the fact that, in our approach, the prediction is made based on considering the data for each class level separately. The more realistic model to formulate this problem is a multi-label classification problem

where all the classes can be present in each decision path. Thus, having a set of code anomalies can result in all three architecture-level anti-patterns at the same time. However, since our training set is not large enough to build a strong, multi-label predictor, we reduced the complexity of the problem to predict each architecture-level anti-pattern separately. In future work, we plan to collect more data to investigate this correlation with multi-label classification. Additionally, our manual validation could have produced skewed results due to the inexperience of the graduate students working with anti-patterns and unfamiliar code.

External validity refers to the generalization of our findings. In this study, we performed our experiments on 113 open-source projects. A larger data set is needed to result in more reliable results. We will also extend the number of participants who manually evaluated the outcomes of our tool combining both code anomalies and architecture anti-patterns.

5.6 Conclusion

We have analyzed 113 open-source Java projects and detected the instances of 16 different code-level anomalies and 3 architecture-level anti-patterns in this work. We formulated the problem as a multi-class classification task and applied Random Forest along with 5-fold cross validation Grid Search to extract the best rules representing the correlation between two levels of anti-patterns. The manual validation by developers confirmed the benefits of ranking the code anomalies based on their impact on the architecture.

Future work involves validating our technique with additional code anomalies and architecture anti-patterns detected on a variety of projects of different sizes. We will also examine if the relationships hold among other programming languages. In another future research direction, we will integrate these findings into refactoring strategies helping developers to manage and fix the most severe anti-patterns.

CHAPTER VI

Conclusion

In this chapter, we conclude with a summary of the contributions of this thesis in Section 6.1 and discuss potential directions of future work in Section 6.2.

6.1 Summary

In Chapter I we presented the context of the work in this thesis placing it in the domain of software maintenance and evolution, in particular work on software refactoring and anti-patterns. The importance of software maintenance and maintaining high quality was discussed as a motivation for the importance of the work. We also identified the general gaps in the current body of knowledge and introduced the three contributions of this thesis to fill this gap.

Next, Chapter II reviewed the relevant background and related works on software refactoring, automated software refactoring recommendation tools, software refactoring dependencies, regression testing, and software anti-patterns.

Chapter III presents a first attempt to unify two different software engineering problems using the tasks of refactoring and regression test case selection as a case study. In Section 3.2 we explain our simultasking algorithm and the key unified solution representation scheme that serves as a common platform for knowledge transfer between the tasks. We evaluated the effectiveness of our technique against two current state-of-the-art approaches for software refactoring and two for regression testing

in Section 3.3. To perform the evaluation we executed our approach and the competing state-of-the-art approaches on six-open sourced systems and one industrial project. Our findings indicated that our approach performs better than the competing state-of-the-art approaches and indicated the strong potential for a simultasking approach.

In Chapter IV we examined a new theory on software refactoring correctness dependencies in Section 4.2 and developed an algorithm and web-tool to detect these dependencies in input refactoring recommendation lists. We then performed an extensive empirical study in Section 4.3 to evaluate the approach on 1,457,873 refactorings recommended for 9,595 Java open-source projects from GitHub. Our study confirms that refactorings are commonly involved in dependent relations and cannot be applied truly independently. We discuss the implications from the study in greater depth in Section 4.5.

We investigated the relationship between architecture anti-patterns and code-level anomalies in Chapter V. Section 5.2 describes our machine learning approach to building a random-forest classifier with 5-fold cross validation Grid Search for architecture anti-patterns based on the code anomalies present in source code. We explain our experimental approach in Section 5.3 where we analyzed 113 open-source Java projects from GitHub and detected the instances of 16 different code-level anomalies and 3 architecture-level anti-patterns. We then validated our work through manual examination of the results of our classifier using developers who confirmed the benefits of ranking the code anomalies based on their impact on the architecture in Section 5.4.

6.2 Future Work

For future work, we envision multiple projects branching off of each of the contributions from this thesis. Our wider aim is to use the knowledge of the techniques, algorithms, and theories present in this thesis in order to improve automated software

refactoring and developers' ability to improve the quality of their code.

In contribution 1 we designed a simultaksing algorithm to unite software refactoring and regression testing. Future work will initially involve validating our technique further with additional refactoring types, test cases, programming languages, and code smells. Further, we wish to investigate adding in additional source of knowledge by adding additional tasks or looking at uniting different software maintenance tasks. One example would be looking at the next release problem where developers need to find what feature, big fixes, or other quality improvements should be included in the next release of the software.

For contribution 2, we plan to investigate several different directions:

1. Pattern Extraction: One important implication of the proposed refactoring dependency theory is the ability to extract common refactoring patterns by mining software repositories using tools such as RefMiner [16]. These patterns are the common non-trivial graphs that can be extracted on different commits/pull-requests of the same project or multiple projects. Such patterns of non-trivial graphs can be linked to refactoring opportunities such as resolving different types of code smells repeatably. In the future, we plan to use the refactoring dependencies to understand the common refactoring patterns from the history of commits and pull requests of software repositories using existing refactoring detection tools such as RefMiner.

2. Refactoring Collaborations Between Developers: Studying the collaborations among multiple developers when refactoring code is a promising next step. Refactoring graphs extracted from commit histories can be linked to the authors of those commits. Then, a graph of collaborations among developers can be generated based on the dependencies among the applied refactorings. This can lead to new insights into why and when developers collaborate for refactoring.

3. Change Operator in Search-based Refactoring: Random selection and application of crossover and mutation when evolving a population of solutions is a challenge in search-based refactoring. Refactoring dependency analysis can be used to avoid destroying good patterns in refactoring solutions and make change operators more intelligent, which can lead to better solutions and faster convergence.

4. Interactive Refactoring Tool Support: Developers can more easily understand the implications of selecting which refactorings from a recommendation to apply, improving the interactive process and increasing their confidence in the recommendation tool. The only restriction in applying non-trivial refactoring graphs is that a refactoring can only be applied if every other refactoring that it depends on (transitively) is also applied. Thus, invalid refactorings can be detected and highlighted on the fly.

Contribution 3's future works involves validating our technique with additional code anomalies and architecture anti-patterns detected on a variety of projects of different sizes. We will also examine if the relationships hold among other programming languages. In another future research direction, we will integrate these findings into refactoring strategies helping developers to manage and fix the most severe anti-patterns in their projects.

BIBLIOGRAPHY

[1] S. R. Schach, *Software Engineering 4th ed.* Boston, MA, USA: McGraw-Hill, 1999.

[2] Galorath, "Software maintenance cost." https://galorath.com/software-maintenance-costs/, 2021. [Online; accessed 15 March 2022].

[3] R. S. Pressman and B. R. Maxim, *Software Engineering A Practitioner's Approach 9th ed.* New York, NY, USA: McGraw-Hill, 2020.

[4] P. Koopman, "Lecture notes: A case study of toyota unintended acceleration and software safety." https://users.ece.cmu.edu/~koopman/pubs/koopman14_toyota_ua_slides.pdf, 18 September 2014. [Online; accessed 17 March 2022].

[5] A. Ouni, M. Kessentini, H. Sahraoui, K. Inoue, and M. S. Hamdi, "Improving multi-objective code-smells correction using development history," *Journal of Systems and Software*, vol. 105, pp. 18–39, 2015.

[6] R. Fenton, "5 staggering medical device recall statistics that should concern everyone." https://www.qualio.com/blog/medical-device-recall-statistics, 10 December 2019. [Online; accessed 1 April 2022].

[7] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley, 1999.

[8] W. G. Griswold and W. F. Opdyke, "The birth of refactoring: A retrospective on the nature of high-impact software engineering research," *IEEE Software*, vol. 32, no. 6, pp. 30–38, 2015.

[9] R. Marinescu, "Detection strategies: Metrics-based rules for detecting design flaws," ICSM '04, (USA), p. 350–359, IEEE Computer Society, 2004.

[10] P. L. Roden, S. Virani, L. H. Etzkorn, and S. Messimer, "An empirical study of the relationship of stability metrics and the qmood quality models over software developed using highly iterative or agile software processes," in *Seventh IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2007)*, pp. 171–179, IEEE, 2007.

[11] A. Ouni, M. Kessentini, H. Sahraoui, K. Inoue, and K. Deb, "Multi-criteria code refactoring using search-based software engineering: An industrial case study," *ACM Trans. Softw. Eng. Methodol.*, vol. 25, pp. 23:1–23:53, 2016.

[12] R. Guru, "Refactoring techniques." https://refactoring.guru/refactoring/techniques, 2014. [Online; accessed 1 April 2022].

[13] E. Murphy-Hill, C. Parnin, and A. P. Black, "How we refactor, and how we know it," *IEEE Transactions on Software Engineering (TSE)*, vol. 38, no. 1, pp. 5–18, 2012.

[14] V. Alizadeh, M. Kessentini, W. Mkaouer, M. Ocinneide, A. Ouni, and Y. Cai, "An interactive and dynamic search-based approach to software refactoring recommendations," *IEEE Transactions on Software Engineering*, 2018.

[15] M. W. Mkaouer, M. Kessentini, S. Bechikh, K. Deb, and M. O. Cinnéide, "Recommendation system for software refactoring using innovization and interactive dynamic optimization," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE '14)*, (Vasteras, Sweden), pp. 331–336, ACM, 2014.

[16] N. Tsantalis, M. Mansouri, L. M. Eshkevari, D. Mazinanian, and D. Dig, "Accurate and efficient refactoring detection in commit history," in *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*, (Gothenburg, Sweden), pp. 483–494, ACM, 2018.

[17] M. Harman and L. Tratt, "Pareto optimal search based refactoring at the design level," in *Proceedings of the 9th annual conference on Genetic and evolutionary computation (GECCO '07)*, (London, England), pp. 1106–1113, ACM, 2007.

[18] Y. Lin, X. Peng, Y. Cai, D. Dig, D. Zheng, and W. Zhao, "Interactive and guided architectural refactoring with search-based recommendation," in *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '16)*, (Seattle, US), pp. 535–546, ACM, 2016.

[19] T. Sharma and D. Spinellis, "A survey on software smells," *Journal of Systems and Software*, vol. 138, pp. 158 – 173, 2018.

[20] M. Mäntylä, J. Vanhanen, and C. Lassenius, "A taxonomy and an initial empirical study of bad smells in code," *International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings.*, pp. 381–384, 2003.

[21] B. Vasilescu, K. Blincoe, Q. Xuan, C. Casalnuovo, D. Damian, P. T. Devanbu, and V. Filkov, "The sky is not the limit: Multitasking across github projects," *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pp. 994–1005, 2016.

[22] N. Moha, Y.-G. Gueheneuc, A.-F. Duchien, *et al.*, "Decor: A method for the specification and detection of code and design smells," *IEEE Transactions on Software Engineering (TSE)*, vol. 36, no. 1, pp. 20–36, 2010.

[23] F. Palomba, G. Bavota, M. D. Penta, R. Oliveto, D. Poshyvanyk, and A. De Lucia, "Mining version histories for detecting code smells," *IEEE Transactions on Software Engineering*, vol. 41, no. 5, pp. 462–489, 2015.

[24] M. Harman and L. Tratt, "Pareto optimal search based refactoring at the design level," in *In GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pp. 1106–1113, ACM, 2007.

[25] N. Tsantalis and A. Chatzigeorgiou, "Identification of move method refactoring opportunities," *IEEE Transactions on Software Engineering*, vol. 35, no. 3, pp. 347–367, 2009.

[26] S. Yoo, M. Harman, and S. Ur, "Highly scalable multi objective test suite minimisation using graphics cards," in *SSBSE '11: Proceedings of the 2011 International Symposium on Search Based Software Engineering*, pp. 219–236, Springer, 2011.

[27] A. Panichella, R. Oliveto, M. D. Penta, and A. Lucia, "Improving multi-objective test case selection by injecting diversity in genetic algorithms," *IEEE Transactions on Software Engineering*, vol. 41, pp. 358–383, 2015.

[28] K. Wang, C. Zhu, A. Celik, J. Kim, D. Batory, and M. Gligoric, "Towards refactoring-aware regression test selection," in *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, (New York, NY, USA), p. 233–244, Association for Computing Machinery, 2018.

[29] E. L. G. Alves, P. D. L. Machado, T. Massoni, and M. Kim, "Prioritizing test cases for early detection of refactoring faults," *Software Testing, Verification and Reliability*, vol. 26, no. 5, pp. 402–426, 2016.

[30] M. Harman, "Refactoring as testability transformation," in *Proceedings of the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, ICSTW '11, (USA), p. 414–421, IEEE Computer Society, 2011.

[31] M. Fokaefs, N. Tsantalis, E. Stroulia, and A. Chatzigeorgiou, "Jdeodorant: Identification and application of extract class refactorings," ICSE '11, (New York, NY, USA), p. 1037–1039, Association for Computing Machinery, 2011.

[32] S. Yoo and M. Harman, "Pareto efficient multi-objective test case selection," in *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, ISSTA '07, (New York, NY, USA), p. 140–150, Association for Computing Machinery, 2007.

[33] J. J. Yackley, M. Kessentini, G. Bavota, V. Alizadeh, and B. R. Maxim, "Simultaneous refactoring and regression testing," in *2019 19th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pp. 216–227, IEEE, 2019.

[34] E. Tom, A. Aurum, and R. Vidgen, "An exploration of technical debt," *Journal of Systems and Software*, vol. 86, no. 6, pp. 1498–1516, 2013.

[35] M. Kuutila, M. Mäntylä, U. Farooq, and M. Claes, "Time pressure in software engineering: A systematic review," *Information and Software Technology*, vol. 121, p. 106257, 2020.

[36] S. A. Slaughter, D. E. Harter, and M. S. Krishnan, "Evaluating the cost of software quality," *Communications of the ACM*, vol. 41, no. 8, pp. 67–73, 1998.

[37] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

[38] G. Bavota, A. De Lucia, M. Di Penta, R. Oliveto, and F. Palomba, "An experimental investigation on the innate relationship between quality and refactoring," *Journal of Systems and Software*, vol. 107, pp. 1–14, 2015.

[39] M. Ó Cinnéide, L. Tratt, M. Harman, S. Counsell, and I. Hemati Moghadam, "Experimental assessment of software metrics using automated refactoring," in *Proceedings of the ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '12, pp. 49–58, ACM, 2012.

[40] A. C. Bibiano, E. Fernandes, D. Oliveira, A. Garcia, M. Kalinowski, B. Fonseca, R. Oliveira, A. Oliveira, and D. Cedrim, "A quantitative study on characteristics and effect of batch refactoring on code smells," in *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pp. 1–11, 2019.

[41] E. Murphy-Hill, C. Parnin, and A. P. Black, "How we refactor, and how we know it," *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 5–18, 2012.

[42] N. Tsantalis and A. Chatzigeorgiou, "Identification of move method refactoring opportunities," *IEEE Transactions on Software Engineering*, vol. 35, no. 3, pp. 347–367, 2009.

[43] M. Kim, T. Zimmermann, and N. Nagappan, "An empirical study of refactoring challenges and benefits at microsoft," *Software Engineering, IEEE Transactions on*, vol. 40, pp. 633–649, July 2014.

[44] A. Ouni, M. Kessentini, H. Sahraoui, K. Inoue, and K. Deb, "Multi-criteria code refactoring using search-based software engineering: an industrial case study," *ACM Transactions on Software Engineering and Methodology*, vol. 25, no. 3, p. 23, 2016.

[45] J. Bansiya and C. G. Davis, "A hierarchical model for object-oriented design quality assessment," *IEEE Transactions on Software Engineering*, vol. 28, no. 1, pp. 4–17, 2002.

[46] T. Ferreira, J. Ivers, J. J. Yackley, M. Kessentini, I. Ozkaya, and K. Gaaloul, "Dependent or not: Detecting and understanding collections of refactorings," *IEEE Transactions on Software Engineering*, Submitted and Under Review 2022.

[47] R. Mo, J. Garcia, Y. Cai, and N. Medvidovic, "Mapping architectural decay instances to dependency models," in *Proceedings of the 2013 4th International Workshop on Managing Technical Debt*, MTD 2013, pp. 39–46, IEEE, 2013.

[48] W. Oizumi, A. Garcia, L. da Silva Sousa, B. Cafeo, and Y. Zhao, "Code anomalies flock together: Exploring code anomaly agglomerations for locating design problems," in *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*, pp. 440–451, IEEE, 2016.

[49] M. Feathers, *Working Effectively with Legacy Code.* Prentice Hall PTR, 2004.

[50] J. Kerievsky, *Refactoring to Patterns.* Pearson Higher Education, 2004.

[51] R. Kazman, Y. Cai, R. Mo, Q. Feng, L. Xiao, S. Haziyev, V. Fedak, and A. Shapochka, "A case study in locating the architectural roots of technical debt," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 2, pp. 179–188, 2015.

[52] J. Carriere, R. Kazman, and I. Ozkaya, "A cost-benefit framework for making architectural decisions in a business context," in *2010 IEEE 32nd International Conference on Software Engineering (ICSE)*, vol. 2, pp. 149–157, IEEE, May 2010.

[53] A. Yamashita and L. Moonen, "Exploring the impact of inter-smell relations on software maintainability: An empirical study," in *Proceedings of the 2013 International Conference on Software Engineering*, pp. 682–691, IEEE Press, 2013.

[54] M. Abbes, F. Khomh, Y.-G. Gueheneuc, and G. Antoniol, "An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension," in *Software maintenance and reengineering (CSMR), 2011 15th European conference on*, pp. 181–190, IEEE, 2011.

[55] D. I. Sjoberg, A. Yamashita, B. C. Anda, A. Mockus, and T. Dyba, "Quantifying the effect of code smells on maintenance effort," *IEEE Transactions on Software Engineering*, no. 8, pp. 1144–1156, 2013.

[56] V. Alizadeh and M. Kessentini, "Reducing interactive refactoring effort via clustering-based multi-objective search," in *Proceedings of the 33rd ACM/IEEE*

International Conference on Automated Software Engineering, ASE 2018, pp. 464–474, ACM, 2018.

[57] M. Kim, M. Gee, A. Loh, and N. Rachatasumrit, "Ref-finder: a refactoring reconstruction tool based on logic query templates," in *Proceedings of the International Symposium on Foundations of Software Engineering*, FSE, pp. 371–372, 2009.

[58] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson, "Automated detection of refactorings in evolving components," in *ECOOP*, vol. 4067, pp. 404–428, 2006.

[59] D. Batory, J. N. Sarvela, and A. Rauschmayer, "Scaling step-wise refinement," in *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, (USA), p. 187–197, IEEE, 2003.

[60] J. Kim, D. Batory, D. Dig, and M. Azanza, "Improving refactoring speed by 10x," in *Proceedings of the 38th International Conference on Software Engineering*, pp. 1145–1156, ACM, 2016.

[61] W. Kessentini, M. Kessentini, H. Sahraoui, S. Bechikh, and A. Ouni, "A cooperative parallel search-based software engineering approach for code-smells detection," *IEEE Transactions on Software Engineering*, vol. 40, no. 9, pp. 841–861, 2014.

[62] A. Ouni, M. Kessentini, H. Sahraoui, and M. Boukadoum, "Maintainability defects detection and correction: A multi-objective approach," *Automated Software Engg.*, vol. 20, p. 47–79, Mar. 2013.

[63] M. W. Mkaouer, M. Kessentini, S. Bechikh, K. Deb, and M. Ó Cinnéide, "Recommendation system for software refactoring using innovization and interactive dynamic optimization," in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pp. 331–336, ACM, 2014.

[64] B. Du Bois, S. Demeyer, and J. Verelst, "Refactoring-improving coupling and cohesion of existing code," in *11th Working Conference on Reverse Engineering (WCRE)*, pp. 144–151, 2004.

[65] I. M. Bertrán, *On the detection of architecturally-relevant code anomalies in software systems.* PhD thesis, PhD thesis, Pontifical Catholic University of Rio de Janeiro (PUC-Rio), Rio de Janeiro, Brazil, 2013.

[66] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, and D. Poshyvanyk, "When and why your code starts to smell bad," in *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pp. 403–414, IEEE Press, 2015.

[67] M. D'Ambros, A. Bacchelli, and M. Lanza, "On the impact of design flaws on software defects," in *Quality Software (QSIC), 2010 10th International Conference on*, pp. 23–31, IEEE, 2010.

[68] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner, "Do code clones matter?," in *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, (USA), p. 485–495, IEEE, 2009.

[69] F. Khomh, M. D. Penta, Y. G. Guéhéneuc, and G. Antoniol, "An exploratory study of the impact of antipatterns on class change- and fault-proneness," *Empirical Software Engineering*, vol. 17, no. 3, pp. 243–275, 2012.

[70] W. H. Brown, R. C. Malveau, and T. J. Mowbray, *AntiPatterns: refactoring software, architectures, and projects in crisis*. John Wiley and Sons, 1998.

[71] J. Garcia, D. Popescu, G. Edwards, and N. Medvidovic, "Identifying architectural bad smells," in *Proceedings of the 2009 European Conference on Software Maintenance and Reengineering*, CSMR '09, (USA), p. 255–258, IEEE, 2009.

[72] R. Mo, Y. Cai, R. Kazman, and L. Xiao, "Hotspot patterns: The formal definition and automatic detection of architecture smells," in *2015 12th Working IEEE/IFIP Conference on Software Architecture*, pp. 51–60, 2015.

[73] J. Letouzey and M. Ilkiewicz, "Managing technical debt with the sqale method," *IEEE Software*, vol. 29, no. 6, pp. 44–51, 2012.

[74] L. Xiao, Y. Cai, R. Kazman, R. Mo, and Q. Feng, "Identifying and quantifying architectural debt," in *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, (New York, NY, USA), p. 488–498, Association for Computing Machinery, 2016.

[75] J. J. Yackley, S. Molaei, M. Kessentini, and B. R. Maxim, "Investigating the relationships between architecture and code anti-patterns using random forest and grid search," *IEEE Access*, Submitted and Under Review 2022.

[76] C. Abid, V. Alizadeh, M. Kessentini, T. d. N. F. Ferreira, and D. Dig, "30 years of software refactoring research: A systematic literature review," *arXiv preprint arXiv:2007.02194*, vol. NA, 2020.

[77] G. Bavota, A. De Lucia, A. Marcus, and R. Oliveto, "Recommending refactoring operations in large software systems," in *Recommendation Systems in Software Engineering* (M. P. Robillard, W. Maalej, R. J. Walker, and T. Zimmermann, eds.), pp. 387–419, Springer Berlin Heidelberg, 2014.

[78] M. O'Keeffe and M. Ó Cinnéide, "A stochastic approach to automated design improvement," in *International Conference on Principles and practice of programming in Java*, pp. 59–62, Computer Science Press, Inc., 2003.

[79] O. Seng, J. Stammel, and D. Burkhart, "Search-based determination of refactorings for improving the class structure of object-oriented systems," in *International conference on Genetic and evolutionary computation*, pp. 1909–1916, ACM, 2006.

[80] M. Kessentini, W. Kessentini, H. Sahraoui, M. Boukadoum, and A. Ouni, "Design defects detection and correction by example," in *2011 IEEE 19th International Conference on Program Comprehension*, pp. 81–90, IEEE, 2011.

[81] A. Ouni, M. Kessentini, and H. Sahraoui, "Search-based refactoring using recorded code changes," in *Proceedings of the 17th European Conference on Software Maintenance and Reengineering (CSMR 2013)*, pp. 221–230.

[82] W. Mkaouer, M. Kessentini, A. Shaout, P. Koligheu, S. Bechikh, K. Deb, and A. Ouni, "Many-objective software remodularization using nsga-iii," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 24, no. 3, pp. 17:1–17:45, 2015.

[83] A. Chávez, I. Ferreira, E. Fernandes, D. Cedrim, and A. Garcia, "How does refactoring affect internal quality attributes? a multi-project study," in *Proceedings of the 31st Brazilian Symposium on Software Engineering*, SBES'17, p. 74–83, Association for Computing Machinery, 2017.

[84] T. Mens, G. Taentzer, and O. Runge, "Analysing refactoring dependencies using graph transformation," *Software & Systems Modeling*, vol. 6, no. 3, pp. 269–285, 2007.

[85] T. Mens, G. Taentzer, and O. Runge, "Detecting structural refactoring conflicts using critical pair analysis," *Electronic Notes in Theoretical Computer Science*, vol. 127, no. 3, pp. 113–128, 2005.

[86] H. Liu, G. Li, Z. Ma, and W. Shao, "Conflict-aware schedule of software refactorings," *IET software*, vol. 2, no. 5, pp. 446–460, 2008.

[87] H. Liu, Z. Ma, W. Shao, and Z. Niu, "Schedule of bad smell detection and resolution: A new way to save effort," *IEEE transactions on Software Engineering*, vol. 38, no. 1, pp. 220–235, 2011.

[88] L. Sousa, D. Cedrim, A. Garcia, W. Oizumi, A. C. Bibiano, D. Oliveira, M. Kim, and A. Oliveira, "Characterizing and identifying composite refactorings: Concepts, heuristics and patterns," in *Proceedings of the 17th International Conference on Mining Software Repositories*, pp. 186–197, 2020.

[89] H. Melton and E. Tempero, "Identifying refactoring opportunities by identifying dependency cycles," in *Proceedings of the 29th Australasian Computer Science Conference-Volume 48*, pp. 35–41, 2006.

[90] N. Yoshida, Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue, "On refactoring support based on code clone dependency relation," in *11th IEEE International Software Metrics Symposium (METRICS'05)*, pp. 10–pp, IEEE, 2005.

[91] G. Rothermel and M. J. Harrold, "A safe, efficient algorithm for regression test selection," in *Proceedings of the Conference on Software Maintenance*, ICSM '93, (USA), p. 358–367, IEEE Computer Society, 1993.

[92] M. J. Harrold, R. Gupta, and M. L. Soffa, "A methodology for controlling the size of a test suite," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 2, no. 3, pp. 270–285, 1993.

[93] S. McMaster and A. Memon, "Call-stack coverage for gui test suite reduction," *IEEE Transactions on Software Engineering*, vol. 34, no. 1, pp. 99–115, 2008.

[94] H.-Y. Hsu and A. Orso, "Mints: A general framework and tool for supporting test-suite minimization," in *2009 IEEE 31st International Conference on Software Engineering*, pp. 419–429, IEEE, 2009.

[95] S. Yoo and M. Harman, "Using hybrid algorithm for pareto efficient multi-objective test suite minimisation," *Journal of Systems and Software*, vol. 83, no. 4, pp. 689 – 701, 2010.

[96] H. Hemmati, L. Briand, A. Arcuri, and S. Ali, "An enhanced test case selection approach for model-based testing: An industrial case study," in *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, (New York, NY, USA), p. 267–276, Association for Computing Machinery, 2010.

[97] A. De Lucia, M. Di Penta, R. Oliveto, and A. Panichella, "On the role of diversity measures for multi-objective test case selection," AST '12, p. 145–151, IEEE Press, 2012.

[98] E. Rogstad, L. Briand, and R. Torkar, "Test case selection for black-box regression testing of database applications," *Information and Software Technology*, vol. 55, no. 10, pp. 1781 – 1795, 2013.

[99] S. Elbaum, A. Malishevsky, and G. Rothermel, "Incorporating varying test costs and fault severities into test case prioritization," in *In Proceedings of the 23rd International Conference on Software Engineering*, pp. 329–338, 2001.

[100] A. G. Malishevsky, J. R. Ruthruff, G. Rothermel, and S. Elbaum, "Cost-cognizant test case prioritization," tech. rep.

[101] K. R. Walcott, M. L. Soffa, G. M. Kapfhammer, and R. S. Roos, "Timeaware test suite prioritization," in *Proceedings of the 2006 International Symposium on Software Testing and Analysis*, ISSTA '06, (New York, NY, USA), p. 1–12, Association for Computing Machinery, 2006.

[102] Z. Li, M. Harman, and R. M. Hierons, "Search algorithms for regression test case prioritization," *IEEE Transactions on Software Engineering*, vol. 33, no. 4, pp. 225–237, 2007.

[103] R. C. Bryce, S. Sampath, and A. Memon, "Developing a single model and test prioritization strategies for event-driven software," *IEEE Transactions on Software Engineering*, vol. 37, pp. 48–64, 2011.

[104] M. Kessentini, M. Wimmer, H. Sahraoui, and M. Boukadoum, "Generating transformation rules from examples for behavioral models," BM-FA '10, (New York, NY, USA), Association for Computing Machinery, 2010.

[105] U. Mansoor, M. Kessentini, M. Wimmer, and K. Deb, "Multi-view refactoring of class and activity diagrams using a multi-objective evolutionary algorithm," *Software Quality Journal*, vol. 25, pp. 473–501, 2015.

[106] U. Mansoor, M. Kessentini, B. R. Maxim, and K. Deb, "Multi-objective code-smells detection using good and bad design examples," *Software Quality Journal*, vol. 25, pp. 529–552, 2016.

[107] F. Palomba, A. De Lucia, G. Bavota, and R. Oliveto, "Anti-pattern detection: Methods, challenges, and open issues," in *Advances in Computers*, vol. 95, pp. 201–238, Elsevier, 2014.

[108] Y. Lin, X. Peng, Y. Cai, D. Dig, D. Zheng, and W. Zhao, "Interactive and guided architectural refactoring with search-based recommendation," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, (New York, NY, USA), pp. 535–546, ACM, 2016.

[109] U. A. Mannan, I. Ahmed, R. A. M. Almurshed, D. Dig, and C. Jensen, "Understanding code smells in android applications," in *Proceedings of the International Conference on Mobile Software Engineering and Systems*, MOBILESoft '16, (New York, NY, USA), p. 225–234, Association for Computing Machinery, 2016.

[110] D. Di Nucci, F. Palomba, D. A. Tamburri, A. Serebrenik, and A. De Lucia, "Detecting code smells using machine learning techniques: are we there yet?," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 612–621, IEEE, 2018.

[111] V. Garousi and B. Küçük, "Smells in software test code: A survey of knowledge in industry and academia," *Journal of Systems and Software*, vol. 138, pp. 52–81, 2018.

[112] M. S. Haque, J. Carver, and T. Atkison, "Causes, impacts, and detection approaches of code smell: a survey," in *Proceedings of the ACMSE 2018 Conference*, p. 25, ACM, 2018.

[113] G. Bavota, B. De Carluccio, A. De Lucia, M. Di Penta, R. Oliveto, and O. Strollo, "When does a refactoring induce bugs? an empirical study," SCAM '12, (USA), p. 104–113, IEEE Computer Society, 2012.

[114] F. Palomba, G. Bavota, M. D. Penta, F. Fasano, R. Oliveto, and A. D. Lucia, "On the diffuseness and the impact on maintainability of code smells: A large scale empirical investigation," *Empirical Softw. Engg.*, vol. 23, p. 1188–1221, June 2018.

[115] K. Deb, A. Member, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: Nsga-ii," *IEEE Transactions on Evolutionary Computation*, pp. 182–197, 2002.

[116] J. F. Gonçalves and M. G. Resende, "A parallel multi-population biased random-key genetic algorithm for a container loading problem," vol. 39, p. 179–190, Feb. 2012.

[117] Y. Kataoka, M. D. Ernst, W. Griswold, and D. Notkin, "Automated support for program refactoring using invariants," *Proceedings IEEE International Conference on Software Maintenance. ICSM 2001*, pp. 736–743, 2001.

[118] J. Bansiya and C. G. Davis, "A hierarchical model for object-oriented design quality assessment," *IEEE Transactions on Software Engineering*, vol. 28, no. 1, pp. 4–17, 2002.

[119] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. Lucia, and D. Poshyvanyk, "When and why your code starts to smell bad (and whether the smells go away)," *IEEE Transactions on Software Engineering*, vol. PP, pp. 1–1, 01 2017.

[120] A. Arcuri and L. Briand, "A practical guide for using statistical tests to assess randomized algorithms in software engineering," in *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, (New York, NY, USA), p. 1–10, Association for Computing Machinery, 2011.

[121] N. Munaiah, S. Kroh, C. Cabrey, and M. Nagappan, "Curating github for engineered software projects," *Empirical Software Engineering*, vol. 22, pp. 3219–3253, 2017.

[122] J. R. Koehler and A. B. Owen, "Computer experiments. handbook of statistics," *Elsevier Science*, pp. 261–308, 1996.

[123] M. Alshayeb and Mohammad, "Empirical investigation of refactoring effect on software quality," *Information and Software Technology*, vol. 51, pp. 1319–, 09 2009.

[124] F. Palomba, A. Zaidman, R. Oliveto, and A. De Lucia, "An exploratory study on the relationship between changes and refactoring," in *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, pp. 176–185, 2017.

[125] C. Vassallo, G. Grano, F. Palomba, H. C. Gall, and A. Bacchelli, "A large-scale empirical exploration on refactoring activities in open source software projects," *Science of Computer Programming*, vol. 180, pp. 1–15, 2019.

[126] G. Szke, G. Antal, C. Nagy, R. Ferenc, and T. Gyimthy, "Empirical study on refactoring large-scale industrial systems and its effects on maintainability," *J. Syst. Softw.*, vol. 129, p. 107–126, jul 2017.

[127] T. Sharma and D. Spinellis, "A survey on software smells," *Journal of Systems and Software*, vol. 138, pp. 158–173, 2018.

[128] L. Breiman, "Random forests," *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.

[129] D. J. Dittman, T. M. Khoshgoftaar, and A. Napolitano, "The effect of data sampling when using random forest on imbalanced bioinformatics data," in *Information Reuse and Integration (IRI), 2015 IEEE International Conference on*, pp. 457–463, IEEE, 2015.

[130] J. J. Snellenburg, S. P. Laptenok, R. Seger, K. M. Mullen, and I. H. M. van Stokkum, "Glotaran: a Java-based Graphical User Interface for the R-package TIMP," *Journal of Statistical Software*, vol. 49, no. 3, pp. 1–22, 2012.

[131] M. Mullenweg and M. Little, "WordPress Github Repository," 2003.

[132] T. Muth, F. Kohrs, R. Heyer, D. Benndorf, E. Rapp, U. Reichl, L. Martens, and B. Y. Renard, "Mpa portable: A stand-alone software package for analyzing metaproteome samples on the go," *Analytical Chemistry*, vol. 90, no. 1, pp. 685–689, 2018.

[133] NIST, "Nist/sematech e-handbook of statistical methods," OCT 2013.