

**Explainable, Security-Aware and Dependency-Aware Framework for  
Intelligent Software Refactoring**

by

**Chaima Abid**

**A dissertation submitted in partial fulfillment  
of the requirements for the degree of  
Doctor of Philosophy  
(Computer & Information Science)  
in the University of Michigan-Dearborn  
2021**

**Doctoral Committee:**

**Associate Professor Marouane Kessentini, Chair  
Professor Bruce Maxim  
Assistant Professor Alireza Mohammadi  
Assistant Professor Zheng Song**



**ISE**Lab  
Intelligent Software Engineering

© Chaima Abid 2021  
All Rights Reserved

## ACKNOWLEDGEMENTS

Through the process of researching and writing this thesis, I have received a great deal of support and assistance.

I would first like to thank my supervisor, Professor Marouane Kessentini, for the countless hours that he dedicated to this thesis. His expertise in my area of research was invaluable in formulating the research questions and methodology. Your insightful feedback pushed me to sharpen my thinking and brought my work to a higher level.

Next, I would like to thank the Department of Computer and Information Science at the University of Michigan Dearborn for providing the required support that allowed me to complete my thesis.

In addition, I would like to thank my parents and my sisters Imen, Nour, and Hajer, for their constant love, their never-ending encouragement and support all through my studies. You are always there for me.

My appreciation also goes out to my partner Kareem Khalil for his tremendous understanding and encouragement in the past few years. Your support has meant more to me than you could possibly realize.

I am also grateful to my current and former lab mates for their help and a cherished time spent together in the lab.

Finally, I could not have completed this dissertation without the support of my friends who provided happy distractions to rest my mind outside of my research.

This dissertation stands as a testament to all your love and encouragement.

## TABLE OF CONTENTS

<b>ACKNOWLEDGEMENTS</b> . . . . .	ii
<b>LIST OF FIGURES</b> . . . . .	vii
<b>LIST OF TABLES</b> . . . . .	xii
<b>LIST OF ABBREVIATIONS</b> . . . . .	xv
<b>ABSTRACT</b> . . . . .	xvii
 <b>CHAPTER</b>	
<b>I. Introduction</b> . . . . .	1
1.1 Research Context . . . . .	1
1.2 Problem Statement & Proposed Contributions . . . . .	2
1.2.1 Problem Statement . . . . .	2
1.2.2 Research Contributions . . . . .	3
1.3 Publications List . . . . .	10
1.4 Organization of the Dissertation . . . . .	12
<b>II. State of the Art</b> . . . . .	13
2.1 30 Years of Software Refactoring Research: A Systematic Literature Review . . . . .	13
2.1.1 Introduction . . . . .	13
2.1.2 Research Methodology . . . . .	16

2.1.3	Refactoring Research Platform . . . . .	26
2.1.4	Results . . . . .	27
2.1.5	Future Research Directions . . . . .	41
2.1.6	Conclusion . . . . .	48
2.2	What Refactoring Topics Do Developers Discuss? A Large Scale Em- pirical Study Using Stack Overflow . . . . .	50
2.2.1	Introduction . . . . .	50
2.2.2	Stack Overflow Data Description . . . . .	52
2.2.3	Research Method . . . . .	53
2.2.4	Results . . . . .	59
2.2.5	Implications of this Study . . . . .	69
2.2.6	Threats to Validity . . . . .	72
2.2.7	Conclusion . . . . .	72
2.3	Related Work . . . . .	74
2.3.1	Systematic Literature Reviews about Refactoring . . . . .	74
2.3.2	Mining Stack Overflow Posts . . . . .	75
2.3.3	Detecting Refactoring Opportunities . . . . .	76
2.3.4	Refactoring Recommendation . . . . .	84
2.4	Background . . . . .	92
2.4.1	Object-Oriented Static Metrics for Software Quality and Se- curity Assessment . . . . .	92
2.4.2	Metrics for Web Services . . . . .	96
2.4.3	Multi-Objective Refactoring Using NSGA-II . . . . .	99
<b>III. Improving the Process of Identifying Potential Refactoring Oppor- tunities . . . . .</b>		<b>102</b>
3.1	Understanding the Impact of Code Quality and Security Metrics of Mobile Apps on User Reviews . . . . .	103

3.1.1	Introduction . . . . .	103
3.1.2	<i>QS-URec</i> : The Proposed Approach . . . . .	107
3.1.3	Experiments and Results . . . . .	114
3.1.4	Threats to Validity . . . . .	126
3.1.5	Conclusion . . . . .	129
3.2	Early Prediction of Quality of Service Using Interface-level Metrics, Code-level Metrics, and Antipatterns . . . . .	130
3.2.1	Introduction . . . . .	130
3.2.2	Approach . . . . .	133
3.2.3	Experiment and Results . . . . .	139
3.2.4	Threats to Validity . . . . .	146
3.2.5	Conclusion and Future Work . . . . .	148
3.3	One Size Does Not Fit All: Customized Benchmark Generation for Software Quality Assessment . . . . .	149
3.3.1	Introduction . . . . .	149
3.3.2	Research Methodology . . . . .	152
3.3.3	Empirical Validation . . . . .	158
3.3.4	Threats to Validity . . . . .	170
3.3.5	Conclusion . . . . .	172
<b>IV. Improving the Refactoring Recommendation Process . . . . .</b>		<b>173</b>
4.1	How Does Refactoring Impact Security When Improving Quality? A Security-Aware Refactoring Approach . . . . .	174
4.1.1	Introduction . . . . .	174
4.1.2	Motivating Example . . . . .	176
4.1.3	Security-Aware Multi-Objective Refactoring . . . . .	178
4.1.4	Experiments and Results . . . . .	182
4.1.5	Threats to Validity . . . . .	201

4.1.6	Conclusion . . . . .	201
4.2	Prioritizing Refactorings for Security Critical Code . . . . .	203
4.2.1	Introduction . . . . .	203
4.2.2	Motivations and Challenges . . . . .	204
4.2.3	Approach . . . . .	206
4.2.4	Experiment and Results . . . . .	212
4.2.5	Threats to Validity . . . . .	222
4.2.6	Conclusion . . . . .	223
4.3	Intelligent Change Operators for Multi-Objective Refactoring . . . . .	225
4.3.1	Introduction . . . . .	225
4.3.2	Dependency-Aware Refactoring Recommendation System . . . . .	227
4.3.3	Empirical Study . . . . .	233
4.3.4	Threats to Validity . . . . .	243
4.3.5	Conclusion . . . . .	246
4.4	X-SBR: On the Use of the History of Refactorings for Explainable Search-Based Refactoring and Intelligent Change Operators . . . . .	247
4.4.1	Introduction . . . . .	247
4.4.2	X-SBR Approach . . . . .	250
4.4.3	Experiment and Results . . . . .	257
4.4.4	Threats to Validity . . . . .	270
4.4.5	Conclusion . . . . .	272
<b>V.</b>	<b>Conclusion . . . . .</b>	<b>274</b>
5.1	Future Work . . . . .	278
<b>BIBLIOGRAPHY</b>	<b>. . . . .</b>	<b>280</b>



## LIST OF FIGURES

### Figure

1.1	Overview of the contributions of this thesis. . . . .	3
2.1	SLR steps . . . . .	18
2.2	Top institutions active in the refactoring field . . . . .	28
2.3	A screenshot of the authors tab of the refactoring repository Website . . . . .	30
2.4	A screenshot of the publications tab of the refactoring repository Website . . . . .	31
2.5	Dashboard of the refactoring repository website . . . . .	32
2.6	A screenshot of the refactoring repository dashboard that shows the authors, their h-index and total number of publications and citations . . . . .	33
2.7	A screenshot of the authors network graph from the refactoring repository website . . . . .	34
2.8	Distribution of refactoring publications around the world. . . . .	35
2.9	Number of publications in the top 10 most active countries in the refactoring field . . . . .	36
2.10	Top 10 Authors with the highest number of publications and citations in the field of refactoring . . . . .	37
2.11	Evolution of the Top 10 Authors during the past 10 years . . . . .	38
2.12	Trend of publications in the field of refactoring during the last three decades. . . . .	39
2.13	Histogram illustrating the percentage of refactoring publications per refactoring life-cycle . . . . .	40

2.14	Histogram illustrating the percentage of publications dealing with manual, semi-automatic and automated refactoring . . . . .	41
2.15	Histogram illustrating the count of refactoring publications per artifact . .	42
2.16	Histogram illustrating the count of refactoring publications per paradigm .	43
2.17	Histogram illustrating the count of publications per refactoring objective .	44
2.18	Histogram illustrating the count of refactoring publications per programming language . . . . .	45
2.19	Histogram illustrating the count of refactoring publications per field . . . .	46
2.20	Pie chart illustrating the percentage of publications in which the authors used industrial and/or open source systems in the validation step . . . . .	47
2.21	Refactoring post found on Stack Overflow . . . . .	53
2.22	An overview of our Stack Overflow analysis. . . . .	55
2.23	Distribution of the number of questions per refactoring topic . . . . .	61
2.24	The distribution of the number of questions in relation to the probability of the dominant topic . . . . .	62
2.25	The four metrics used to estimate refactoring topics popularity. . . . .	64
2.26	The three used metrics to estimate the level of difficulty. . . . .	67
2.27	The evolution of the number of questions by topic overtime. . . . .	68
3.1	Overview of the QS-URec Approach . . . . .	107
3.2	Example reviews related to quality issues . . . . .	110
3.3	A screenshot of our tool that shows quality/security computation results .	111
3.4	Security keywords used in the security metrics calculations . . . . .	112
3.5	Boxplot of the evaluation metrics for <i>QS-URec</i> , <i>QS-URec</i> without weights, and the work of Palomba et al. [1] . . . . .	122
3.6	Boxplot of the <i>SecurityOverlap</i> and <i>QualityOverlap</i> of all the apps . . . .	123
3.7	Boxplot of the results of the survey conducted with our industrial partner Under Armour . . . . .	127

3.8	Approach Overview . . . . .	134
3.9	The average severity score of the different types of antipattern on the QoS attributes based on our data set of web services . . . . .	146
3.10	Quality evaluation of a project A using two different benchmarks . . . . .	150
3.11	Overview of the proposed approach. . . . .	152
3.12	QBench dashboard showing quality profile of the selected project. . . . .	157
3.13	QBench’s detailed quality report. . . . .	158
3.14	A bar-plot that shows the component importance generated by the principal component analysis . . . . .	164
3.15	A grid of boxplots that shows the distributions of repository features across the clusters generated by the K-means algorithm with K=7 . . . . .	166
3.16	A bubble chart that summarizes the Kolmogorov–Smirnov test results. . .	167
3.17	Distribution of the developers’ answers about the Sensitivity of the quality metrics. . . . .	168
4.1	An example of a security vulnerability from Django REST Registration library due to refactorings. . . . .	177
4.2	A simplified bank account system hierarchy before and after refactoring . .	177
4.3	Security-Aware Multi-Objective Refactorings . . . . .	178
4.4	Sample of outputs (refactorings) of our Web app on the Open CSV project to balance quality and security. . . . .	180
4.5	Average distribution of the refactoring types among the solutions recommended for the 30 projects that significantly improve the security objective. . . . .	191
4.6	Impact of the recommended refactorings on security metrics based on the 30 projects. . . . .	191
4.7	Distribution of refactoring solutions based on each pair of quality and security metrics for the 30 projects. . . . .	193

4.8	Average manually determined correctness of the refactorings on different open source projects generated by our tool (+Security) and an existing refactoring tool (-Security) [2].. . . . .	194
4.9	Box plots of the impact of refactoring solutions on the quality attributes based on 4 open source projects using our tool (+Security) and an existing refactoring tool (-Security) [2]. The results are statistically significant using the two-sample t-test at a 95% confidence level ( $\alpha = 5\%$ ) . . . . .	195
4.10	Distribution of the refactoring solutions using the security objective based on 4 open source projects comparing our tool (+Security) and an existing refactoring tool (-Security) [2]. . . . .	195
4.11	The important motivations for code refactoring by the participants. . . . .	196
4.12	The potential impacts of refactoring on security metrics based on the survey. . . . .	197
4.13	The potential impact of different refactoring types on security metrics based on the survey. . . . .	198
4.14	The possible positive impact of improving the security metrics on quality attributes based on the survey. . . . .	199
4.15	Box plots of the impacts of refactoring solutions on both quality and security objectives based on the 30 projects. . . . .	200
4.16	A category in the CVE security bug database [3] that includes security vulnerabilities related to poor code quality . . . . .	206
4.17	An example of a security vulnerability from NUUO CMS system due to code quality issues [4]. . . . .	207
4.18	Security-Critical Code Identification: Approach Overview . . . . .	208
4.19	List of keywords used in our approach . . . . .	209
4.20	An example of a security-critical code fragments identified by our approach . . . . .	210
4.21	An example of a Pareto front of refactoring solutions generated by our tool for OpenCSV project. . . . .	211

4.22	The manual evaluation scores (MC@k) on the seven systems with k=3, 5 and 10. . . . .	218
4.23	Average execution time, in minutes, on the seven systems. . . . .	219
4.24	The severity scores (severity@k) on the seven systems with k=3, 5 and 10. . . . .	221
4.25	Sample refactoring recommendations from JDeodorant. . . . .	226
4.26	A simplified example of refactorings that depend on each other. . . . .	228
4.27	An illustration of the intelligent crossover. . . . .	232
4.28	An illustration of the dependency-aware mutation. . . . .	233
4.29	Percentage of invalid refactorings across all solutions per generation for NSGA-II, Dep-NSGA-II, and Intel-NSGA-II. . . . .	239
4.30	Percentage of invalid refactorings in refactoring solutions using NSGA-II, Dep-NSGA-II, and Intel-NSGA-II. . . . .	240
4.31	Manual evaluation of refactoring recommendations generated by the existing multi-objective techniques [5, 6, 7, 8] and the JDeodorant Eclipse plugin [9].	244
4.32	Approach Overview . . . . .	251
4.33	Example of an association rule . . . . .	253
4.34	Improved initial population process . . . . .	254
4.35	An illustration of X-SBR crossover . . . . .	255
4.36	An illustration of X-SBR mutation . . . . .	257
4.37	Average execution time (ms) of all algorithms using the four systems . . . . .	265
4.38	Average number of invalid refactorings in the solutions of all algorithms using the four systems . . . . .	267
4.39	Automated and manual evaluation of refactoring recommendations generated by the different refactoring tools . . . . .	270
4.40	Distribution of the relevance of the explanations according to the survey results (1=not relevant-5=very relevant) . . . . .	271

## LIST OF TABLES

### Table

2.1	Final list of search strings . . . . .	19
2.2	PS quality assessment questions [10] . . . . .	20
2.3	List of countries and their replacements . . . . .	22
2.4	List of keywords used to detect the different categories . . . . .	24
2.5	Representative references for all categories . . . . .	29
2.6	Attributes describing a Stack Overflow post . . . . .	54
2.7	List of candidate tags . . . . .	57
2.8	The 6 refactoring related topics with the 10 most important words in each topic . . . . .	60
2.9	Quality attributes and their equations. . . . .	92
2.10	QMOOD metrics description. . . . .	93
2.11	Security metrics terminology. . . . .	94
2.12	Security metrics definition . . . . .	95
2.13	Web service metrics [11] . . . . .	97
2.14	Refactoring types considered in our study . . . . .	101
3.1	Summary of the mobile apps considered in our study. . . . .	116
3.2	Correlation analysis results between quality attributes and review ratings .	119
3.3	Correlation analysis results between security metrics and user review ratings	120
3.4	Percentage of files that were identified correctly . . . . .	121

3.5	Evaluation results using p-value and Vargha-Delaney A measure . . . . .	123
3.6	<i>Precision</i> and <i>recall</i> of running <i>QS-URec</i> on <i>MyFitnessPal</i> . . . . .	127
3.7	Antipattern Detection rules [12] . . . . .	138
3.8	Web services used in our dataset . . . . .	140
3.9	Support, confidence and lift of the rules that predict QoS from anti-patterns	144
3.10	Rules to predict QoS from anti-patterns . . . . .	145
3.11	Rules to predict QoS . . . . .	147
3.12	Support, confidence and lift of the Rules to predict QoS . . . . .	147
3.13	Overview of the used clustering algorithms. . . . .	156
3.14	Selected Developers and eBay projects. . . . .	162
3.15	Clustering results. . . . .	165
3.16	The sensitivity of the quality metrics . . . . .	168
3.17	Participants quality assessment vs <i>QBench</i> . . . . .	171
3.18	<i>QBench</i> correctness precision for each of the seven benchmarks. . . . .	171
4.1	Studied open source projects. . . . .	186
4.2	Correlation results between the average of security metrics and different refactoring types on the 30 projects. The results are statistically significant using the 2sample t-test with a 95% confidence level ( $\alpha = 5\%$ ) . . . . .	190
4.3	The two most common refactoring patterns with the highest impact on the improvement of the average security measure for the 30 open source projects.	190
4.4	Correlation results between the average of security metrics and quality at- tributes on the 30 projects. The results are statistically significant using the two-sample t-test at a 95% confidence level ( $\alpha = 5\%$ ) . . . . .	192
4.5	Demographics of the studied projects. . . . .	216
4.6	Selected programmers. . . . .	217
4.7	The three operation-variants of the NSGA-II algorithm. . . . .	234
4.8	Open-source projects studied. . . . .	235

4.9	Participant details. . . . .	239
4.10	Performance indicators results for NSGA-II, Dep-NSGA-II, and Intel-NSGA-II. . . . .	241
4.11	Average quality improvement of the solutions generated by NSGA-II, Dep-NSGA-II, and Intel-NSGA-II. . . . .	242
4.12	Systems considered for validation . . . . .	258
4.13	Effect Size values (Eta squared ( $\eta^2$ )) for corresponding software project and metric. . . . .	264
4.14	Participants details . . . . .	265
4.15	Evaluation metrics and statistics of the rules . . . . .	265
4.16	Results of the Hypervolume ( $I_{HV}$ ) and Generational Distance ( $I_{GD}$ ) indicators	268
4.17	Results of the Contributions ( $I_C$ ) metric . . . . .	269



## LIST OF ABBREVIATIONS

**SOA** Service-Oriented Architecture

**UML** Unified Modeling Language

**UI** User Interface

**OCL** Object Constraint Language

**IDE** Integrated Development Environment

**SLR** Systematic Literature Review

**LDA** Latent Dirichlet Allocation

**QMOOD** Quality Metrics for Object Oriented Design

**QoS** Quality of Service

**PS** Primary Studies

**RQ** Research Question

**GUI** Graphical user interface

**NSGA-II** Non-dominated Sorting Genetic Algorithm

**CI** Continuous Integration

**SQuaRE** Software product Quality Requirements and Evaluation

**SQA** software quality assurance

**CVE** Common Vulnerabilities and Exposures

## ABSTRACT

As software systems continue to grow in size and complexity, their maintenance continues to become more challenging and costly. Even for the most technologically sophisticated and competent organizations, building and maintaining high-performing software applications with high-quality-code is an extremely challenging and expensive endeavor. Software Refactoring is widely recognized as the key component for maintaining high-quality software by restructuring existing code and reducing technical debt. However, refactoring is difficult to achieve and often neglected due to several limitations in the existing refactoring techniques that reduce their effectiveness. These limitations include, but are not limited to, detecting refactoring opportunities, recommending specific refactoring activities, and explaining the recommended changes. Existing techniques are mainly focused on the use of quality metrics such as coupling, cohesion, and the Quality Metrics for Object Oriented Design (QMOOD). However, there are many other factors identified in this work to assist and facilitate different maintenance activities for developers:

1. To structure the refactoring field and existing research results, this dissertation provides the most scalable and comprehensive systematic literature review analyzing the results of 3183 research papers on refactoring covering the last three decades. Based on this survey, we created a taxonomy to classify the existing research, identified research trends and highlighted gaps in the literature for further research.
2. To draw attention to what should be the current refactoring research focus from the developers' perspective, we carried out the first large scale refactoring study on the

most popular online Q&A forum for developers, Stack Overflow. We collected and analyzed posts to identify what developers ask about refactoring, the challenges that practitioners face when refactoring software systems, and what should be the current refactoring research focus from the developers' perspective.

3. To improve the detection of refactoring opportunities in terms of quality and security in the context of mobile apps, we designed a framework that recommends the files to be refactored based on user reviews. We also considered the detection of refactoring opportunities in the context of web services. We proposed a machine learning-based approach that helps service providers and subscribers predict the quality of service with the least costs. Furthermore, to help developers make an accurate assessment of the quality of their software systems and decide if the code should be refactored, we propose a clustering-based approach to automatically identify the preferred benchmark to use for the quality assessment of a project.
4. Regarding the refactoring generation process, we proposed different techniques to enhance the change operators and seeding mechanism by using the history of applied refactorings and incorporating refactoring dependencies in order to improve the quality of the refactoring solutions. We also introduced the security aspect when generating refactoring recommendations, by investigating the possible impact of improving different quality attributes on a set of security metrics and finding the best trade-off between them. In another approach, we recommend refactorings to prioritize fixing quality issues in security-critical files, improve quality attributes and remove code smells.

All the above contributions were validated at the large scale on thousands of open source and industry projects in collaboration with industry partners and the open source community. The contributions of this dissertation are integrated in a cloud-based refactoring framework which is currently used by practitioners.

# CHAPTER I

## Introduction

### 1.1 Research Context

The growing complexity and scale of industrial software systems affects their speed, their overall performance and present difficult challenges in design, development, and asserting software quality [13, 14]. Refactoring [15, 16, 17] is a technique that improves the design structure while preserving the overall functionality and behavior. It is a key practice in agile development processes and well supported by tools integrated with major Integrated Development Environments (IDEs). Recently, software industry is becoming more and more aware of the importance of refactoring for reaching long-term goals, and they encourage their developers to continuously refactor their code to set a clean foundation for future updates. A recent study [18] shows that developers are spending considerable time struggling with existing code (e.g., understanding, restructuring, etc.) rather than creating new code, and this may have a harmful impact on developer creativity. Various tools for code refactoring have been proposed during the past two decades ranging from manual support [19, 20, 21] to fully automated techniques [22, 23, 24, 25, 26, 27, 28, 29, 30, 7]. Despite the promising results of refactoring techniques on both open-source and industry projects, developers are still reluctant to use these refactorings tools. This reluctance is due to many limitation in the existing tools that include poor consideration of the characteristics of the artifact to be refactored, ignoring the security aspect of the software as well as the dependencies among

refactorings, and history of refactorings.

## 1.2 Problem Statement & Proposed Contributions

### 1.2.1 Problem Statement

In this thesis, we first identify and synthesize all of the scholarly research on software refactoring while exploring the experiences and perspectives of both researchers and engineers. Then, we leverage different algorithms to enhance the mechanisms of identifying refactoring opportunities and generating refactoring recommendation.

- Gain an understanding of the existing research and debates relevant to software refactoring, and present that knowledge in the form of a systematic literature review and a thorough empirical study.
- Design and implement scalable approaches that combine search algorithms with machine learning for the generation of refactoring recommendation. We utilized for the first time the knowledge extracted from the history of applied refactorings and the dependencies between refactoring operations to improve the existing refactoring tools.
- Design and implement approaches to detect refactoring opportunities using static analysis and machine learning techniques. In this context, we focus on Web services, mobile apps, and GitHub repositories.

Figure 1.1 represents the different contributions of this thesis. The presented framework contains three main components. The goal of the first component is to identify, evaluate, and summarize the findings and challenges of the refactoring field from researchers and developers perspectives. For that, we submitted two research contributions. The second component consists of improving the process of identifying potential refactoring opportunities where we published one contribution (C4) and got two others accepted with major revisions (C3 and C5). The third and last component of this thesis consists of improving the refactoring

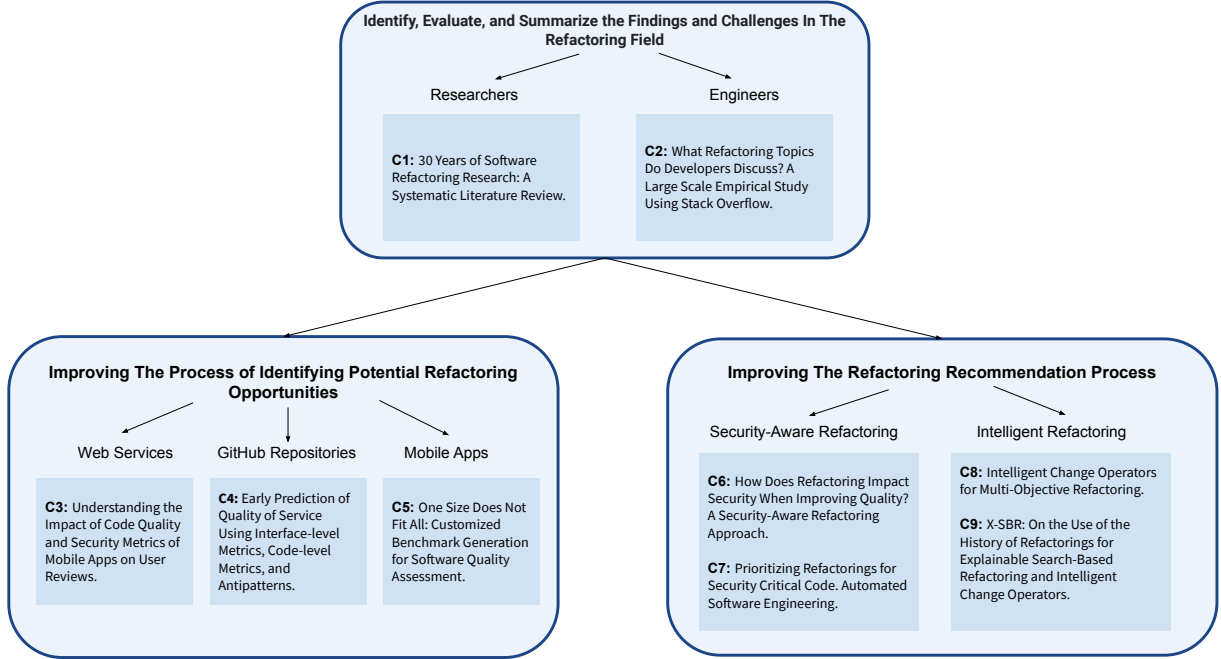


Figure 1.1: Overview of the contributions of this thesis.

recommendation process. In this context, we were able to publish four contributions all in top tier journals and conferences.

### 1.2.2 Research Contributions

In the following, we will summarize the objectives of each contribution. The refactoring research efforts are fragmented over several research communities, various domains, and objectives. To structure the field and existing refactoring research results, we have initiated our dissertation by a systematic literature review on refactoring.

#### ***Contribution 1: 30 Years of Software Refactoring Research: A Systematic Literature Review***

Refactoring studies are extensively expanded beyond code-level restructuring to be applied at different levels (architecture, model, requirements, etc.), adopted in many domains beyond the object-oriented paradigm (cloud computing, mobile, web, etc.), used in industrial settings and considered objectives beyond improving the design to include other non-functional requirements (e.g., improve performance, security, etc.).

To understand the current status of the refactoring field, structure it, and identify potential gaps, we performed a study that provides a large scale systematic literature review by analyzing the results of 3183 research papers on refactoring covering the last three decades since 1990. We also created a taxonomy to classify the existing research, identified research trends, and highlighted gaps in the literature and avenues for further research.

The increasing demand on refactoring have surpassed the academic and research community to spark the interest of software industries as well industries with software departments. For this reason, we concluded the urgent need for a study that highlights the challenges that practitioners face when refactoring software systems and what should be the current refactoring research focus from the developers' perspective.

***Contribution 2: What Refactoring Topics Do Developers Discuss? A Large Scale Empirical Study Using Stack Overflow***

Very few studies focused on the challenges that practitioners face when refactoring software systems and what should be the current refactoring research focus from the developers' perspective. Without such knowledge, tool builders invest in the wrong direction, and researchers miss many opportunities for improving the practice of refactoring. In this study, we collected data from the popular online Q&A site, Stack Overflow, and analyzed posts to identify what do developers ask about refactoring. We clustered these questions to find the different refactoring related topics using one of the most popular topic modeling algorithms, Latent Dirichlet Allocation (LDA). We found that developers are asking about design patterns, design and user interface refactoring, web services, parallel programming, and mobile apps. We also identified what popular refactoring challenges are the most difficult and the current important topics and questions related to refactoring. Moreover, we discovered gaps between existing research on refactoring and the challenges developers face.

After grasping the current stage of progress of the refactoring field in the academic



research and industry, we tried to address some of the long-standing research problems related to the identification of refactoring opportunities in mobile apps, web services, and GitHub projects through the following contributions:

**Contribution 3: *"So What?": Understanding the Impact of Code Quality and Security Metrics of Mobile Apps on User***

The timely detection of emerging quality and security code issues is critical for developers and managers to efficiently manage software maintenance activities and satisfy their customer's needs. Mobile app reviews can highlight important complaints related to quality and security. Despite the considerable work on classifying and identifying user review topics, we currently lack understanding of whether and which quality and security metrics can impact user reviews. Indeed, the current use of app reviews to identify files to fix is limited to analysis of textual similarities, not complementing them with source code metric values. In this contribution, we first studied the correlation between the evolution of user-perceived quality and security of mobile apps and source code quality and security metrics. Based on the outcomes of this study linking code quality and security metrics to user reviews, we designed a framework, *QS-URec*, to address emerging quality and security issues by analyzing both user reviews and source code metrics. *QS-URec* recommends files to be fixed and links them to user reviews. We evaluated our approach on 50 popular mobile apps from *Google Play* with 290,000 reviews, along with a large and popular mobile app provided by our industrial partner that serves millions of users and has received over 400,000 reviews. Our results show strong correlations between several code metrics and the user ratings from reviews complaining about security and quality. *QS-URec* linked security/quality issues in user reviews to the affected files with higher precision and recall than textual analysis.

**Contribution 4: *Early Prediction of Quality of Service Using Interface-level Metrics, Code-level Metrics, and Antipatterns***

**Context:** With the current high trends of deploying and using web services in practice,

effective techniques for maintaining high quality of Service are becoming critical for both service providers and subscribers/users. Service providers want to predict the quality of service during early stages of development before releasing them to customers. Service clients consider the quality of service when selecting the best one satisfying their preferences in terms of price/budget and quality between the services offering the same features. The majority of existing studies for the prediction of quality of service are based on clustering algorithms to classify a set of services based on their collected quality attributes. Then, the user can select the best service based on his expectations both in terms of quality and features. However, this assumption requires the deployment of the services before being able to make the prediction and it can be time-consuming to collect the required data of running web services during a period of time. Furthermore, the clustering is only based on well-known quality attributes related to the services performance after deployment. In this contribution, we start from the hypothesis that the quality of the source code and interface design can be used as indicators to predict the quality of service attributes without the need to deploy or run the services by the subscribers. We collected training data of 707 web services and we used machine learning to generate association rules that predict the quality of service based on the interface and code quality metrics, and antipatterns. The empirical validation of our prediction techniques shows that the generated association rules have strong support and high confidence which confirms our hypothesis that source code and interface quality metrics/antipatterns are correlated with web service quality attributes which are response time, availability, throughput, successability, reliability, compliance, best practices, latency, and documentation.

***Contribution 5: One Size Does Not Fit All: Customized Benchmark Generation for Software Quality Assessment***

It is critical that software systems meet high-quality standards to become less costly and more reliable. Though the research community has proposed various metrics and

anti-patterns to detect quality issues as well as refactoring approaches to fix them, it is still challenging to make an accurate interpretation of the quality metrics and detected anti-patterns to decide if the code should actually be refactored. It is challenging for practitioners to understand whether the values of the quality metrics are good or bad without comparison to an appropriate benchmark of other projects. To address this gap, we propose a clustering-based approach to automatically identify the preferred benchmark to use for the quality assessment of a project. We collect 29 repository features and 20 quality metrics of 54,569 open-source projects. Then, we compare seven clustering algorithms to find distinct clusters based on repository features. After identifying the best set of clusters, we investigate the sensitivity of the quality metrics with respect to the different clusters/benchmarks. Finally, we automatically identify the best benchmark for a set of industry projects and compare the quality assessment results with the manual evaluations of programmers. The results show the effectiveness of the repository features in finding clusters of projects with different characteristics and that quality metrics are sensitive to the selected cluster/benchmark.

Another area to inspect is the refactoring recommendation process. There are several gaps that are yet to be addressed in order to improve the refactoring efficiency. For this reason we propose the following four contributions:

***Contribution 6: How Does Refactoring Impact Security When Improving Quality? A Security-Aware Refactoring Approach***

While state of the art of software refactoring research uses various quality attributes to identify refactoring opportunities and evaluate refactoring recommendations, the impact of refactoring on the security of software systems when improving other quality objectives is under-explored. It is critical to understand how a system is resistant to security risks after refactoring to improve quality metrics. For instance, refactoring is widely used to improve the reusability of code, however such an improvement may increase the attack surface due to the created abstractions. Increasing the spread of

security-critical classes in the design to improve modularity may result in reducing the resilience of software systems to attacks. In this contribution, we investigated the possible impact of improving different quality attributes (e.g. reusability, extendibility, etc.), from the QMOOD model on a set of 8 security metrics defined in the literature related to the data access. We also studied the impact of different refactorings on these static security metrics. Then, we proposed a multi-objective refactoring recommendation approach to find a balance between quality attributes and security based on the correlation results to guide the search. We evaluated our tool on 30 open source projects. We also collected the practitioner perceptions on the refactorings recommended by our tool in terms of the possible impact on both security and other quality attributes. Our results confirm that developers need to make trade-offs between security and other qualities when refactoring software systems due to the negative correlations between them.

#### ***Contribution 7: Prioritizing Refactorings for Security Critical Code***

It is vitally important to fix quality issues in security-critical code as they may be sources of vulnerabilities in the future. These quality issues may increase the attack surface if they are not quickly refactored. In this contribution, we use the history of vulnerabilities and security bug reports along with a set of keywords to automatically identify a project's security-critical files based on its source code, bug reports, pull-request descriptions and commit messages. After identifying these security-related files, we estimate their risks using static analysis to check their coupling with other project components. Then, our approach recommends refactorings to prioritize fixing quality issues in these security-critical files to improve quality attributes and remove identified code smells. To find a trade-off between the quality issues and security-critical files, we adopted a multi-objective search strategy. We evaluated our approach on six open source projects and one industrial system to check the correctness and relevance of the refactorings targeting security critical code.

**Contribution 8: *Intelligent Change Operators for Multi-Objective Refactoring***

In this contribution, we propose intelligent change operators and integrate them into an evolutionary multi-objective search algorithm to recommend valid refactorings that address conflicting quality objectives such as understandability and effectiveness. The proposed intelligent crossover and mutation operators incorporate refactoring dependencies to avoid creating invalid refactorings or invalidating existing refactorings. Further, the intelligent crossover operator is augmented to create offspring that improve solution quality by exchanging blocks of valid refactorings that improve a solution's weakest objectives. We used our intelligent change operators to generate refactoring recommendations for four widely used open-source projects. The results show that our intelligent change operators improve the diversity of solutions. They also accelerate solution convergence to a feasible solution that optimizes the trade-off between the conflicting quality objectives. Finally, they reduce the number of invalid refactorings by up to 71.52% compared to existing search-based refactoring approaches, and increase the quality of the solutions. Our approach outperformed the state-of-the-art search-based refactoring approaches and an existing deterministic refactoring tool based on manual validation by developers with an average manual correctness, precision and recall of 0.89, 0.82, and 0.87.

**Contribution 9: *X-SBR: On the Use of the History of Refactorings for Explainable Search-Based Refactoring and Intelligent Change Operators***

Many of the existing refactoring tools and research are based on search-based techniques to find relevant recommendations by finding trade-offs between different quality attributes. While these techniques show promising results on open-source and industry projects, they lack explanations of the recommended changes which can impact their trustworthiness when adopted in practice by developers. Furthermore, most of the adopted search-based techniques are based on random population generation and

random change operators (e.g. crossover and mutation). However, it is critical to understand which good refactoring patterns may exist when applying change operators to either keep them or exchange with other solutions rather than destroying them with random changes. In this contribution, we propose an enhanced knowledge-informed multi-objective search algorithm, called X-SBR, to provide explanations for refactoring solutions and improve the generated recommendations. First, we generate association rules using the Apriori algorithm to find relationships between applied refactorings in previous commits, their locations, and their rationale (quality improvements). Then, we use these rules to 1) initialize the population, 2) improve the change operators and seeding mechanisms of the multi-objective search in order to preserve and exchange good patterns in the refactoring solutions, and 3) explain how a sequence of refactorings collaborate in order to improve the quality of the system (e.g. fitness functions).

### 1.3 Publications List

- Abid, C., Kessentini, M., Alizadeh, V., Dhouadi, M., & Kazman, R. (2020). How Does Refactoring Impact Security When Improving Quality? A Security-Aware Refactoring Approach. *IEEE Transactions on Software Engineering (TSE)*. DOI: 10.1109/TSE.2020.3005995. Impact Factor 6.11.
- Abid, C., Kessentini, M., & Wang, H. (2020). Early Prediction of Quality of Service Using Interface-level Metrics, Code-level Metrics, and Antipatterns. *Information and Software Technology*. DOI: 10.1016/j.infsof.2020.106313. Impact factor: 2.92.
- Abid, C., Kessentini, M., Alizadeh, V., Dhouadi, M., & Kazman, R. (2020). Prioritizing Refactorings for Security Critical Code. *Automated Software Engineering*. DOI: 10.1007/s10515-021-00281-2. Impact factor: 3.13.
- Abid, C., Rzig, D., Ferreira, T., Kessentini, M., & Tushar, S. (2021). X-SBR: On the Use of the History of Refactorings for Explainable Search-Based Refactoring and

Intelligent Change Operators. IEEE Transactions on Software Engineering (TSE). DOI: 10.1109/TSE.2021.3105037. Impact Factor 6.11.

- Abid, C., Ivers, J., Ferreira, T., Kessentini, M., Ben Kahla, F., & Ozkaya, I. Intelligent Change Operators for Multi-Objective Refactoring. (2021). The 36th IEEE/ACM International Conference on Automated Software Engineering (ASE 2021). DOI: 10.7302/3184. Acceptance rate 17%.
- Abid, C., Kessentini, M., Alizadeh, V., Dhouadi, M., & Kazman, R. (2020). How Does Refactoring Impact Security When Improving Quality? A Security-Aware Refactoring Approach. FSE 2020 : ACM SIGSOFT International Symposium on Foundations of Software Engineering. FSE2020, Journal First Track, Accepted. Acceptance rate 18.5%
- Alkhazi, B., Abid, C., Kessentini, M., & Wimmer, M. (2020). On the value of quality attributes for refactoring ATL model transformations: A multi-objective approach. Information and Software Technology, 120, 106243. DOI: 10.1016/j.infsof.2019.106243. Impact factor: 2.92.
- Alkhazi, B., Abid, C., Kessentini, M., Leroy, D., & Wimmer, M. (2020). Multi-criteria test cases selection for model transformations. Automated Software Engineering, 1-28. DOI: 10.1007/s10515-020-00271-w. Impact factor: 3.13.
- Alkhazi, B., Abid, C., Kessentini, M., Leroy, D., & Wimmer, M. (2020). Multi-criteria test cases selection for model transformations. The 35th IEEE/ACM International Conference on Automated Software Engineering (ASE2020), Journal First Track, Accepted. DOI: 10.1007/s10515-020-00271-w. Acceptance rate 17%.
- Abid, C., Rzig, D., Ferreira, T., Kessentini, M., Tushar, S., & Palomba, F. (2021). One Size Does Not Fit All: Customized Benchmark Generation for Software Quality

Assessment. **Minor Revisions** at IEEE Transactions on Software Engineering (TSE). Impact Factor 6.11.

- Abid, C., Kessentini, M., & Tushar, S. Understanding the Impact of Code Quality and Security Metrics of Mobile Apps on User Reviews. (2021). **Minor Revisions** at ACM Transactions on Software Engineering and Methodology (TOSEM). Impact Factor 6.11.
- Abid, C., Alizadeh, V., Kessentini, M., Ferreira, T. D. N., & Dig, D. (2020). 30 Years of Software Refactoring Research: A Systematic Literature Review. **Under review** at IEEE Access. Impact Factor 3.74.
- Abid, C., Alizadeh, V., Kessentini, M. (2021). What Refactoring Topics Do Developers Discuss? A Large Scale Empirical Study Using Stack Overflow. **Under review** at IEEE Access. Impact Factor: 3.74.

#### 1.4 Organization of the Dissertation

This thesis is organized as follows: Chapter II includes a systematic literature review about the refactoring field, an empirical study about the challenges that developers face when refactoring software systems, background information needed to understand the research contributions, and, finally, a summary of the work related to this thesis.

Chapter III presents three techniques to improve the process of identifying refactoring opportunities. We looked into this problem within the context of mobile apps, web services, and GitHub projects.

Chapter IV describes four approaches to enhance the process of generating refactoring recommendations by proposing intelligent change operators and introducing the security aspect in the refactoring generation.

Finally, a summary and future research directions are presented in Chapter V.



## CHAPTER II

### State of the Art

This chapter is composed of four main parts: 1) a systematic literature review that covers 30 Years of software refactoring research 2) a large scale empirical study using Stack Overflow to identify what software practitioners ask about refactoring 3) the necessary background information related to this dissertation 4) an overview of existing studies directly related to the thesis' contributions.

#### 2.1 30 Years of Software Refactoring Research: A Systematic Literature Review

##### 2.1.1 Introduction

For decades, code restructuring has been applied in informal ways before it became a research interest. The research on *Refactoring* started in the late '80s, concurrently and independently at two universities: William Griswold and David Notkin at U. of Washington were studying refactoring of functional programs in Scheme. William Opdyke and Ralph Johnson at U. of Illinois were studying the refactoring object-oriented programs, particularly in the context of reusable frameworks in C++. The first known use of the term *Refactoring* in the published literature was in an article written by Opdyke and Johnson in September 1990 [31]. Then followed William Griswold's Ph.D. dissertation [32], published in 1991. One year later, William Opdyke published his Ph.D. dissertation [33].

In 1999, Martin Fowler with co-authors from U. of Illinois refactoring group published the first book on refactoring titled *Refactoring: Improving the Design of Existing Code* [34]. This book popularized the practice of code refactoring, and provided a clear taxonomy and definitions of refactoring types. Fowler defined Refactoring in his book as a sequence of small changes - called refactoring operations - made to the internal structure of the code without altering its external behavior. The goal of these refactoring operations is to improve the code readability and reusability, as well as reduce its complexity and maintenance costs in the long run. Since then, a lot has changed in the software development world and in the academic research, but one thing has remained the same: The need for Refactoring.

The Refactoring area is growing very rapidly, and many advances, challenges, and trends have lately emerged. Recently, several researchers and practitioners have adopted the use of refactoring operations at higher degrees of abstraction than source code level (e.g., databases, Unified Modeling Language (UML) models, Object Constraint Language (OCL) rules, etc.). As a result, they often had to redefine the principles and guidelines of refactoring according to the requirements and specifications of their domains. For instance, within User Interface Refactoring developers make changes to the UI to retain its semantics and consistency for all users. These refactorings include, but not limited to, *Align entry field*, *Apply common button size*, *Apply font*, *Indicate format*, and *Increase color contrast*. In Database Refactoring, developers improve the database schema by applying changes such as *Rename column*, *Split table*, *Move method*, *Replace LOB with table*, and *Introduce column constraint*.

Although the different refactoring communities (e.g., software maintenance and evolution, model-driven engineering, formal methods, search-based software engineering, etc.) are interdependent in many ways, they remain disconnected, which may create inconsistencies. For example, when model-level Refactoring does not match the code-level practice, it can lead to incoherence and technical issues during development. The gap is visible not only between different refactoring domains but also between practitioners and researchers. The distance between them primarily originates from the lack of insights into both worlds' recent

findings and needs. For instance, developers tend to use the refactoring features provided by IDEs due to their accessibility and popularity. Most of the time, they are not informed of the benefits that can be derived from adopting state-of-the-art advances in academia. All these challenges call for a need to identify, critically appraise, and summarize the existing work published across the different domains. Existing systematic literature reviews examine findings in very specific refactoring sub-areas such as identifying the impact of refactoring on quality metrics [35] or code smells [36]. To the best of our knowledge, none of the existing surveys or systematic literature reviews collect and synthesize existing research, tools, and recent advances made in the refactoring community on a broad scale to summarize the big picture of the current state of the field.

This study includes the most comprehensive synthesis of theories and principles of refactoring intended to help researchers and practitioners make quick advances and avoid reinventing or re-implementing research infrastructure from scratch, wasting time and resources. We also build a refactoring infrastructure [37] that will connect researchers with practitioners in the industry and provide a bridge between different refactoring communities in order to advance the field of refactoring research.

This Systematic Literature Review (SLR) follows a defined protocol [38, 39, 40] to increase the study’s validity and rationality so that the output can be high in quality and evidence-based. We used various electronic databases and a large number of articles to comprise all the possible candidate studies and cover more works than existing SLRs.

This SLR contributes to the existing literature in the following ways:

- We identify a set of 3183 studies related to refactoring published until May 2020, fulfilling the quality assessment criteria. These studies can be used by the research and industry communities as a reliable basis and help them conduct further research on Refactoring.
- We present a comprehensive qualitative and quantitative synthesis reflecting the state-of-the-art in refactoring with data extracted from those 3183 high-rigor studies. Our

synthesis covers the following themes: artifacts, refactoring tools, different approaches, and performance evaluation in refactoring research.

- We provide guidelines and recommendations based on our findings to support further research in the area.
- We implement a platform [37] that includes the following components: (1) A searchable repository of refactoring publications based on our proposed taxonomy; (2) A searchable repository of authors who contributed to the refactoring community; (3) Analysis and visualization of the refactoring trends and techniques based on the collected papers. The proposed infrastructure will allow researchers and practitioners to easily report refactoring publications and upload information about active authors in the field of Refactoring. It will also bridge the different communities to advance the field of refactoring research and provide opportunities to educate the next refactoring generation.

### **2.1.2 Research Methodology**

Our literature review follows the guidelines established by Kitchenham and Charters [38], which decompose a systematic literature review in software engineering into three stages: planning, conducting, and reporting the review. We have also considered the guidelines from recent systematic literature reviews in the fields of empirical software engineering [35] and search-based software engineering [41]. All the steps of our research are documented, and all the related data are available online for further validation and exploration [37]. This section details the performed research steps and the protocol of the literature review. First, section 2.1.2.1 describes the research questions underlying our survey. Second, section 2.1.2.2 details the literature search step. Next, section 2.1.2.3 highlights the inclusion and exclusion criteria. The data pre-processing step and our proposed taxonomy are described in sections 2.1.2.4 and 2.1.2.5, respectively. The quality assessment criteria are defined in section 2.1.2.6.

Finally, Section 2.1.2.7 discusses threats to the validity of our study.

### 2.1.2.1 Research Questions

The following research questions have been derived based on the objectives described in the introduction section, which form the basis for the literature review:

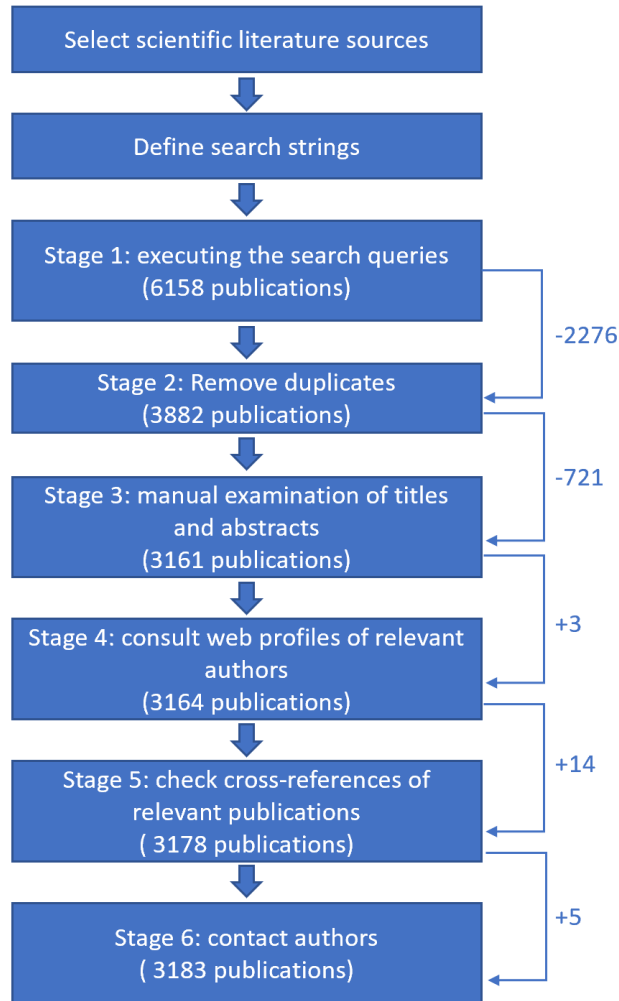
- RQ1: What are the refactoring life-cycle activities?
- RQ2: What are the types of artifacts used for refactoring?
- RQ3: Why refactoring is performed?
- RQ4: What are the different refactoring methods and targeted programming languages?
- RQ5: To what extent refactoring is validated in industry versus open source environments?

### 2.1.2.2 Literature Search Strategy

Following existing SLR guidelines, all the papers have been queried from a wide range of scientific literature sources to make our search as comprehensive as possible:

- **Digital libraries:** ACM Library, IEEE Xplore, Science- Direct, SpringerLink.
- **Citation databases:** Web of Science (formerly ISI Web of Knowledge), Scopus.
- **Citation search engines:** DBLP, Google Scholar.

We first defined a list of terms covering the variety of both application domains and refactoring techniques. Thus, we checked the title, keywords, and abstracts of the relevant papers that were already known to us. Synonyms and keywords were derived from this list. These keywords were combined using logical operators ANDs and ORs to create search terms. Before start collecting the Primary Studies (PS), we tested the search terms' effectiveness



**Figure 2.1:** SLR steps

on all the data sources. Then, we refined the queries to avoid getting irrelevant papers. The string adjustments were agreed on by all authors. The final list of search strings is shown in Table 2.1. These search strings were modified to suit the specific requirements of different electronic databases. We conducted our last search on May 31st, 2020, and identified studies published up until that date to update our database of papers collected and analyzed during a period of 3 years.

In our systematic review, we followed a multi-stage model to minimize the probability of missing relevant publications as much as possible. The different stages are shown in figure 2.1 along with the total returned publications at each stage. The first stage consists of

executing the search queries on the databases mentioned above; a total of 6158 references were found. Then, we removed the duplicates, which reduced the list of candidate papers to 3882. Then, we performed a manual examination of titles and abstracts to discard irrelevant publications based on the inclusion and exclusion criteria. We also looked at the body of the paper whenever necessary. This decreased the list of candidate papers to 3161 publications. Next, we used the resulting set as input for the snowballing process, recommended by Wohlin [39], to identify additional studies. We consulted web profiles of relevant authors and their networks. We also checked cross-references until no further papers were detected. As a result, 17 new references were added. After that, we contacted the corresponding authors of the identified publications to inquire about any missing relevant studies. This led to adding 5 studies.

**Table 2.1:** Final list of search strings

<b>search strings</b>
(software OR system OR code OR service OR diagram OR database OR architecture OR Model OR GUI OR user interface OR UI OR design OR artifact OR developer OR computer OR programming OR object-oriented OR implement OR mobile app OR cloud OR document ) AND (refactor OR refactoring)

### 2.1.2.3 Inclusion and Exclusion Criteria

To filter out the irrelevant articles among those selected in Stage 2 and determine the Primary studies, we considered the following inclusion and exclusion criteria.

**2.1.2.3.1 Inclusion Criteria** All of the following criteria must be satisfied in the selected primary studies:

1. The article must have been published in a peer reviewed journal or conference proceeding between the years 1990 and 2020. The main reason for imposing a constraint over the start year is because the first known use of the term “refactoring” in the published

**Table 2.2:** PS quality assessment questions [10]

	Question
Design	Are the applied identification techniques for refactoring opportunities clearly described?
	Are the refactoring activities considered clearly stated and defined?
	Was the sample size justified?
	Are the evaluation measures fully defined?
Conduct	Are the data collection methods adequately described?
Analysis	Are the results of applying the identification techniques evaluated?
	Are the data sets adequately described? (size, programming languages, source)
	Are the study participants or observational units adequately described?
	Are the statistical methods described?
	Are the statistical methods justified?
	Is the purpose of the analysis clear?
	Are the scoring systems (performance evaluation) described?
Conclusion	Are all study questions answered?
	Are negative findings presented?
	Are the results compared with previous reports?
	Do the results add to the literature?
	Are validity threats discussed?

literature was in a September, 1990 article by William Opdyke and Ralph Johnson [31]. We included papers up till May 31st 2020.

2. The article must be related to computer science and engineering and propose techniques, methods and tools for refactoring.
3. The paper must be written in English.
4. In case a conference paper has a journal extension, we would include both the conference and journal publications.
5. The paper must pass the quality assessment criteria that are elaborated in Section 2.1.2.6.

**2.1.2.3.2 Exclusion Criteria** Papers satisfying any of the exclusion criteria were discarded, as follows:

1. Studies that are not related to the computer science field.



2. Studies that investigated the impact of general maintenance on code quality. In this case, the maintenance tasks were potentially performed due to several reasons and not limited to refactoring, and therefore, we cannot judge whether the impact was due to refactoring or to other maintenance tasks such as corrective or adaptive maintenance.
3. Grey Literature

#### **2.1.2.4 Data Preprocessing**

A pre-processing technique was applied to improve reliability and precision, as detailed in the following sub sections.

**2.1.2.4.1 Simplifying Author's Name** In general, scientific and bibliographic databases such as *Web of Science* (WoS) and *Scopus* have the following inconsistencies in authors names:

- Most journals abbreviate the author's first name to an initial and a dot.
- Most journals use the author name's special accents.
- WoS uses a comma between the author's last name and first name initial, but Scopus does not.

These name-related inconsistencies mean that scientometrics scripts cannot find all of the similar author's names. For that reason, we applied the following steps to simplify author's name fields:

- Remove dots and coma from author's name.
- Remove special accents from author's name

**2.1.2.4.2 Fixing Inconsistent Country Names** Some authors use different naming to refer to the same country (such as USA and United States). For that reason, some country names were replaced based on Table 2.3.

**Table 2.3:** List of countries and their replacements

<b>Country</b>	<b>Replacement</b>
Republic of China	China
USA	United States
England, Scotland and Wales	England
U Arab Emirates	United Arab Emirates
Russia	Russian Federation
Viet Nam	Vietnam
Trinid & Tobago	Trinidad and Tobago

### 2.1.2.5 Study Classification

According to the research questions listed in Section 2.1.2.1, we classified the studies into five dimensions: (1) refactoring life-cycle (related to RQ1), (2) artifacts affected by refactoring (related to RQ2), (3) refactoring objectives (related to RQ3), (4) refactoring techniques (related to RQ4) and (5) refactoring evaluation (related to RQ5). The determination of the attributes of each dimension was performed incrementally. For each dimension, we started with an empty set of attributes. During a period of 3 years (2017-2020), the authors of this study screened the full texts of the articles one by one, analyzed each reported study based on the considered dimension, and determined the attributes of that dimension as considered by each primary study (PS). Table 2.4 outlines the keywords extracted for each category. It should be pointed out that, most of the time, we remove all of the affixes (i.e., suffixes, prefixes, etc.) attached to a word in order to keep its lexical base, also known as root or stem or its dictionary form or lemma. For instance, the word *document* allows us to detect the words *documentation* and *documenting*. Furthermore, we did not include bi-grams and tri-grams that can be detected using one uni-gram. For example, *Class Diagram*, *Object Diagram*, *Sequence Diagram*, and *Use Case Diagram* can all be detected using the word *Diagram* alone.

The screening of the PSs resulted in determining six stages for the refactoring life-cycle (e.g., detection, prioritization, recommendation, testing, documentation, and prediction). We also classified the papers according to the level of automation of the proposed technique

(e.g., automatic, manual, semi-automatic). The results are described in section 2.1.4.1. For the second dimension, we identified five artifacts on which the impact of refactoring is studied by at least one of the PSs. These artifacts are code, architecture, model, GUI, and database. The classification of PSs based on these artifacts is discussed in detail in Section 2.1.4.2. We subdivided the third dimension into five categories (e.g., External quality, internal quality, performance, migration, and security) to reflect the refactoring objective and six categories (e.g., Object-oriented design, Aspect-oriented design, Model-driven engineering, Documentation, Mobile development, and Cloud computing) to describe the refactoring paradigms.

The classification of PSs based on these categories is discussed in detail in Section 2.1.4.3. We divided the fourth dimension into four categories (e.g., data mining, search-based algorithms, formal methods, and fuzzy logic) to reveal the refactoring techniques adopted in the studies and into twelve categories (e.g., Java, C, C#, Python, Cobol, PHP, Smalltalk, Ruby, Javascript, MATLAB, and CSS) to show the most common programming languages used in our PSs. The details of this categorization are reported in section 2.1.4.4. Finally, for the fifth dimension, we divide the PSs into two categories: open-source and industrial. The open-source category includes studies that validate their approaches using open source systems. In contrast, the industrial category consists of the studies that validate their work on systems of their industrial collaborators. These findings are outlined in Section 2.1.4.5.

### **2.1.2.6 Study Quality Assessment**

To ensure a level of quality of papers, we only included venues that are known for publishing high-quality software engineering research in general with an h-index of at least 10, as has been done by [42]. Each of the papers that were published before 2019 has to be cited at least once. The quality of each primary study was assessed based on a quality checklist defined by Kitchenham and Charters [38]. This step aims to extract the primary studies with information suitable for analysis and answering the defined research questions. The quality

**Table 2.4:** List of keywords used to detect the different categories

Category	Keywords
Refactoring Life-cycle (RQ1)	
Detection	detect, opportunity, smell, antipattern, design defect
Prioritization	schedul, sequence, priorit
Recommendation	recommend, correction, correcting, fixing, suggest
Testing	test, regression testing, test case, unit test
Documentation	document
Prediction	predict, future release, next release, development history, refactoring history
Level of automation (RQ1)	
Manual	manual
Semi-automatic	semi-automat, semi-manual
Automatic	automat
Artifact (RQ2)	
Code	code, java, object orient, smell, antipattern, anti-pattern, object-orient
Model	design, model, UML, diagram, Unified Modeling Language
Architecture	architecture, hotspot, hierarchy
GUI	gui, user interface, UI
Database	relational, schema, database, Structured Query Language, SQL
Paradigm (RQ3)	
Object-oriented design	object orient, object-orient, oo, java, c, ++, python, C sharp, c#, css, Python, R, PHP, JavaScript, Ruby, Perl, Object Pascal, Objective-C, Dart, Swift, Kotlin, Common Lisp, MATLAB, Smalltalk
Aspect-oriented design	aspect
Model-driven engineering	model transform, uml, reverse engineering, diagram, Unified Modeling Language
Documentation	document
Mobile development	android, mobile, IOS, phone, smartphone, cellphones
Cloud computing	web service, wsdl, restful, cloud, Apache Hadoop, Docker, Middleware, Software-as-a-Service, SaaS, XaaS, Anything-as-a-Service, Platform-as-a-Service, PaaS, Infrastructure-as-a-Service, IaaS, AWS, Amazon EC2, Amazon Simple Storage Service, S3
Refactoring Objectives (RQ3)	
Internal Quality	maintainability, cyclomatic, depth of inheritance, coupling, quality, Flexibility, Portability, Re-usability, Readability, Testability, Understandability
Performance	performance, parallel, Response Time, Error Rates, Request Rate, availability
External quality	analysability, changeability, time behaviour, resource, Correctness, Usability, Efficiency, Reliability, Integrity, Adaptability, Accuracy, Robustness
Migration	migrat
Security	secure, safety, Attack surface, virus, hack, vulnerability, vulnerable, spam
Programming languages (RQ4)	
Java	java
C	c, c++
C#	c sharp, c#
Python	python
CSS	css
PHP	php
Cobol	cobol
Javascript	javascript
Ruby	ruby
Smalltalk	smalltalk
MATLAB	matlab
Adopted methods (RQ4)	
Search-based algorithms	search, search-base, sbse, genetic, fitness, simulated annealing, tabu search, search space, Hill climbing, Multi-objective evolutionary algorithms, multi objective optimization, multi-objective programming, vector optimization, multi-criteria optimization, multi-attribute optimization, Pareto optimization, Evolutionary Multi-objective Optimization, EMO, Single-Objective Optimization, Many-Objective Optimization, multi objective
Data mining	artificial intelligence, ai , machine learning, naive bayes, decision tree, SVM, support vector machine, Cluster, Classification, classify, Association, Neural networks, deep learning, random forest, regression, reinforcement learning, learning
Formal methods	model check, formal method, B-Method, RAISE, Z notation, SPARK Ada
Fuzzy logic	fuzzy
Evaluation method (RQ5)	
Open source	open source, open-source
Industrial	proprietary, industrial, industry, collaborator, collaboration

checklist, (described in Table 2.2) were defined by Galster et al. [42]. They are developed by considering bias and validity problems that can occur at different stages, including the study design, conduct, analysis, and conclusion. Each question is answered by a "Yes", "Partially", or "No", which correspond to a score of 1, 0.5, or 0, respectively. If a question does not apply to a study, we do not evaluate the study for that question. The quality assessment checklist was independently applied to all 3882 studies by two of the authors. All disagreements on the quality assessment results were discussed, and a consensus was reached eventually. Few cases where agreement could not be reached were sent to the third author for further investigation. 154 studies did not meet the quality assessment criteria.

#### **2.1.2.7 Threats to Validity**

Several limitations may affect the generalizability and the interpretations of our results. The first is the possibility of paper selection bias. To ensure that the studies were selected in an unbiased manner, we followed the well-defined research protocol and guidelines reported by Kitchenham and Charters[38] instead of proposing nonstandard quality factors. Also, the final decision on the articles with selection disagreements was performed based on consensus meetings. The Primary studies were assessed by one researcher and checked by the other, a technique applied in similar studies [41]. The second threat consists of missing a relevant study. To overcome this threat, we employed several strategies that we mentioned in Section 2.1.2.2. Few related studies were detected after performing the automatic search, which indicates that the constructed search strings and the mentioned utilized libraries were comprehensive enough to identify most of the relevant articles. Another critical issue is whether our taxonomy is complete and robust sufficient to analyze and classify the primary studies. To overcome this problem, we used an iterative content analysis method by going through the papers one by one and continuously expand the taxonomy for every new encountered concept. Furthermore, to gather sufficient keywords to detect the different categories, we followed the same iterative process, and we added synonyms based on the authors' expertise

in the field of refactoring. Another threat is related to the tagging of the papers according to our taxonomy. To mitigate this problem, we asked 27 graduate students to check the correctness of the classification results by reading the abstract, the title, and keywords. They also check the body of the paper whenever necessary.

### 2.1.3 Refactoring Research Platform

We implemented a large scale platform [43] that collects, manages, and analyzes refactoring related papers to help researchers and practitioners share, report, and discover the latest advancements in software refactoring research. The first release of the platform is based on the data collected using the methodology described in the previous section. It includes the following components:

1. **A searchable repository of refactoring publications based on our proposed taxonomy.** Figure 2.4 shows a screenshot of the publications' tab of the refactoring repository website. The papers can be searched by author, title, or year of publication. Each paper has tags that describe its content based on our taxonomy described in section 2.1.2.5. The papers can also be filtered using those tags and sorted alphabetically or chronologically according to the title and year of publication, respectively. The users can export the publications' dataset to many formats, including pdf, excel, and CSV. They can also easily report a new publication by entering its link.
2. **A searchable repository of authors who contributed to the refactoring community.** Figure 2.3 shows a screenshot of the authors' tab of the refactoring repository website. The authors can be searched and sorted alphabetically by name, affiliation, or country. They can also be sorted based on the total number of refactoring publications. The user can also check the *Google Scholar* and *Scopus* profiles of the authors if available. Finally, the user can easily report a new author by entering their information and their profile. Furthermore, we defined *the refactoring h-index*, which shows how

many papers about refactoring published by the author have been cited proportionately. A refactoring h-index of X means that the author has X papers about refactoring that have been cited at least X times. Authors can also be sorted according to the refactoring h-index and the total number of citations (see figure 2.6). Besides, we created a co-author network and corresponding visualizations (see figure 2.7) to get a snapshot view of the breadth and depth of an individual's collaborations in the field of refactoring research. Finally, we generated a histogram (see figure 2.2) that shows the number of publications issued by the top institutions active in the refactoring research by considering the authors' affiliations.

**3. Analysis and visualization of the refactoring trends and techniques based on the collected papers.** Figure 2.5 shows a screenshot of the refactoring repository dashboard. It contains histograms and pie charts that show the distribution and percentages of the categories defined in our taxonomy. It also includes maps that reflect the spread of refactoring activity across the world.

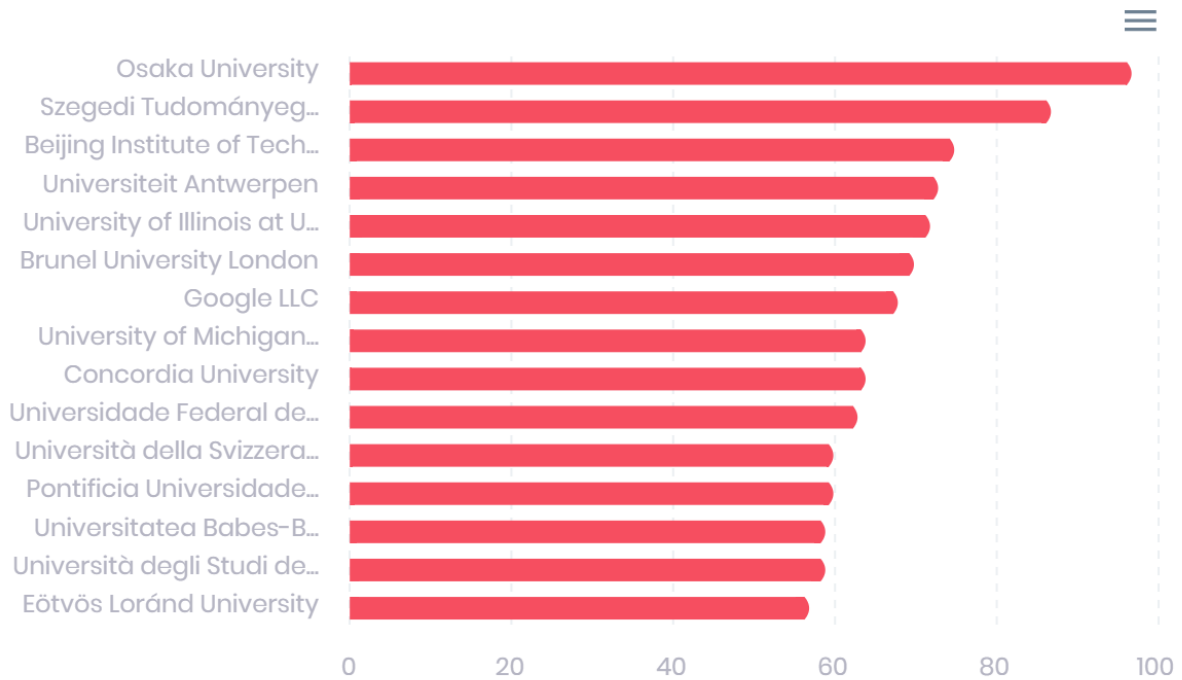
The proposed infrastructure will enable new researchers in refactoring to perform a fair comparison between the novelty of their new refactoring approach and state-of-the-art techniques; enable researchers to use refactoring data of large software systems; facilitate collaborations between researchers from currently disconnected domains/communities of refactoring (model-driven engineering, service computing, parallelism and performance optimization, software quality, testing, etc.); enable practitioners and researchers to quickly identify relevant existing research papers and tools for their problems based on the proposed taxonomy and classification; and enable effective interactions between practitioners and refactoring researchers to identify relevant problems faced by the software industry.

#### **2.1.4 Results**

In this section, we aim to answer the research questions. To provide an overview of the current state of the art in refactoring and guide the reader towards a specific set of

## Top Institutions

(Author X from Institution Y) \* #Publications



**Figure 2.2:** Top institutions active in the refactoring field

approaches, tools, and recent advances that are of interest, we classified the 3183 reviewed papers based on the taxonomy described in Section 2.1.2.5. Table 2.5 contains representative references for the categories created for each Research Question (RQ). In the table, We selected a set of 10 representative references per category as we are dealing with a total of 3183 papers. Those papers are the most cited per each category. The complete results of the classification of all the papers are provided in our repository [43]. The rest of this section summarizes the observations and insights that can be derived from the classification results.

Figure 2.8 shows the distribution of publications related to refactoring across the globe. Figure 2.9 reflects the number of publications in the top 10 most active countries in the field of Refactoring. The United States is on the top of the list of countries with a total of 1604



**Table 2.5:** Representative references for all categories

Category	Percentage	Papers
Refactoring life-cycle (RQ1)		
Detection	28.65%	[44, 45, 46, 47, 48, 49, 50, 51, 52, 53]
Prioritization	9.43%	[54, 55, 56, 57, 58, 59, 60, 61, 62, 63]
Recommendation	16.18%	[54, 64, 46, 55, 65, 66, 67, 68, 69, 70]
Testing	18.44%	[71, 72, 73, 47, 49, 50, 51, 74, 75, 56]
Documentation	5.22%	[76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86]
Prediction	4.818%	[87, 88, 89, 90, 91, 92, 93, 94, 95, 96]
Level of automation (RQ1)		
Automatic	30.95%	[97, 98, 99, 100, 101, 102, 103, 104, 105, 106]
Semi-automatic	1.95%	[107, 108, 109, 110, 111, 112, 113, 114, 115, 116, 117, 118]
Manual	8.67%	[119, 120, 121, 122, 123, 124, 125, 126, 127, 112]
Artifact (RQ2)		
Code	72.89%	[44, 54, 128, 45, 46, 129, 108, 130, 131, 132]
Model	59.25%	[71, 44, 72, 46, 133, 108, 130, 132, 134, 135]
Architecture	17.25%	[71, 136, 134, 137, 138, 139, 140, 141, 142, 143]
GUI	2.58%	[71, 133, 130, 132, 49, 51, 144, 145, 146, 147]
Database	4.12%	[108, 148, 70, 143, 149, 79, 150, 151, 152, 153]
Paradigm (RQ3)		
Object-oriented design	34.09%	[44, 128, 130, 131, 73, 154, 51, 155, 156, 144]
Aspect-oriented	10.87%	[157, 131, 158, 144, 159, 139, 160, 145, 161, 146]
Model-driven engineering	7.35%	[46, 108, 75, 162, 58, 163, 164, 165, 101, 166]
Mobile apps development	3.55%	[130, 138, 155, 167, 66, 142, 168, 169, 170, 130]
Cloud computing	4.15%	[171, 172, 173, 174, 175, 176, 177, 178, 179, 180]
Refactoring Objective (RQ3)		
Internal Quality	41.63%	[72, 64, 46, 133, 132, 55, 73, 181, 137, 182]
Performance	15.93%	[71, 129, 131, 134, 55, 135, 158, 53, 162, 139]
External quality	22.68%	[130, 134, 135, 138, 183, 184, 145, 185, 186, 187]
Migration	3.61%	[138, 156, 188, 189, 190, 191, 192, 193, 143, 194]
Security	3.11%	[156, 195, 196, 197, 198, 199, 200, 201, 202, 203]
Programming language (RQ4)		
Java	17.15%	[44, 128, 130, 131, 73, 51, 155, 156, 53, 183]
C	4.65%	[154, 139, 148, 147, 204, 189, 205, 206, 207, 102]
C#	0.66%	[104, 208, 209, 210, 211, 212, 213, 214, 215, 216]
Python	0.53%	[217, 218, 219, 220, 221, 222, 223, 224, 225, 226]
CSS	0.5%	[227, 228, 229, 230, 231, 232, 233, 234, 190, 235]
PHP	0.35%	[236, 237, 238, 212, 239, 240, 241, 242, 243, 244]
Cobol	0.31%	[245, 246, 247, 248, 249, 250, 251, 252]
MATLAB	0.28%	[253, 254, 255, 256, 257, 258, 259, 260]
Smalltalk	0.79%	[261, 262, 263, 264, 265, 266, 267, 268, 269, 270]
Ruby	0.22%	[271, 212, 224, 272, 273, 274]
Javascript	0.72%	[275, 155, 276, 277, 278, 279, 280, 281, 282, 283, 284]
Scala	4.02%	[98, 285, 286, 287, 288, 169, 289, 129, 76, 290]
Adopted Method (RQ4)		
Search-based algorithms	25.76%	[55, 291, 292, 293, 294, 295, 296, 297, 298, 299]
Data mining	15.49%	[300, 45, 228, 150, 301, 125, 302, 303, 304, 305]
Formal methods	2.92%	[306, 85, 242, 307, 308, 309, 310, 311, 312]
Fuzzy logic	0.28%	[300, 313, 314, 315, 316, 316, 317]
Evaluation method (RQ5)		
Open source	16.31%	[44, 318, 131, 55, 73, 182, 50, 155, 291, 75]
Industrial	10.4%	[163, 319, 55, 158, 52, 59, 320, 190, 30, 321]

#	NAME	AFFILIATION	COUNTRY	#REFACTORINGPUB ↓	SCOPUSPROFILE
1	Kessentini M.	University of Michigan-Dearborn	United States	43	
2	Dig D.	Oregon State University	United States	39	
3	Counsell S.	Brunel University London	United Kingdom	36	
4	Inoue K.	Osaka University	Japan	32	
5	Bavota G.	Università della Svizzera italiana	Switzerland	29	
6	Gheyi R.	Universidade Federal de Campina Grande	Brazil	27	

**Figure 2.3:** A screenshot of the authors tab of the refactoring repository Website

publications followed by Brazil and China with a total of 770 and 626 publications, respectively. During the last 4 years, the number of published refactoring studies has increased with an average of 37% in all the top 10 countries. This demonstrates a considerable increase in interest/need in Refactoring. Based on the above results, refactoring can be among the fastest-growing software engineering research areas, if not the fastest.

Over 5584 authors contributed to the field of Refactoring. We highlight the most active authors in Figure 2.10 and 2.11, based on both the number of publications and citations in the area. Many scholars started research in the refactoring field prior to 2000. Others are relatively new to the field and started their contributions after the year 2010. All top 10 authors in the field have a constantly increasing number of publications over the past 10 years. Marouane Kessentini heads the list with a total of 43 publications (51% of them were published during the past five years) followed by Danny Dig and Steve Counsell with a total of 39 and 36 publications, respectively. Figure 2.12 is a histogram showing how many publications were issued each year starting from 1990. The number of published journal articles, conference papers, and books has increased dramatically during the last decade,

Refactoring Publications  
Until May 2020

Export New Record

All Papers Target Life Cycle Languages Objectives Evaluation Fields Applied Paradigm

Search Search

#	AUTHORS	TITLE	YEAR	TAGS
1	Lyerly, R., Kim, S.H., Ravindran, B.	libMPPNode: An OpenMP Runtime For Parallel Processing Across Incoherent Domains	2019	Code, Model, Performance, Migration, SearchBased
2	Cordy J.R., Inoue K., Koschke R.	2012 6th International Workshop on Software Clones, IWSC 2012 - Proceedings: Foreword	2012	Code, Architecture, Model, InternalQuality, Industrial, SearchBased, Detect
3	Koschke R.	2013 7th International Workshop on Software Clones, IWSC 2013 - Proceedings: Foreword	2013	Code, Architecture, Model, InternalQuality
4	Namiot D.E., Romanov V.Yu.	3D visualization of architecture and metrics of the software [3D визуализация архитектуры и метрик программного обеспечения]	2018	Architecture, InternalQuality, OpenSource, SearchBased

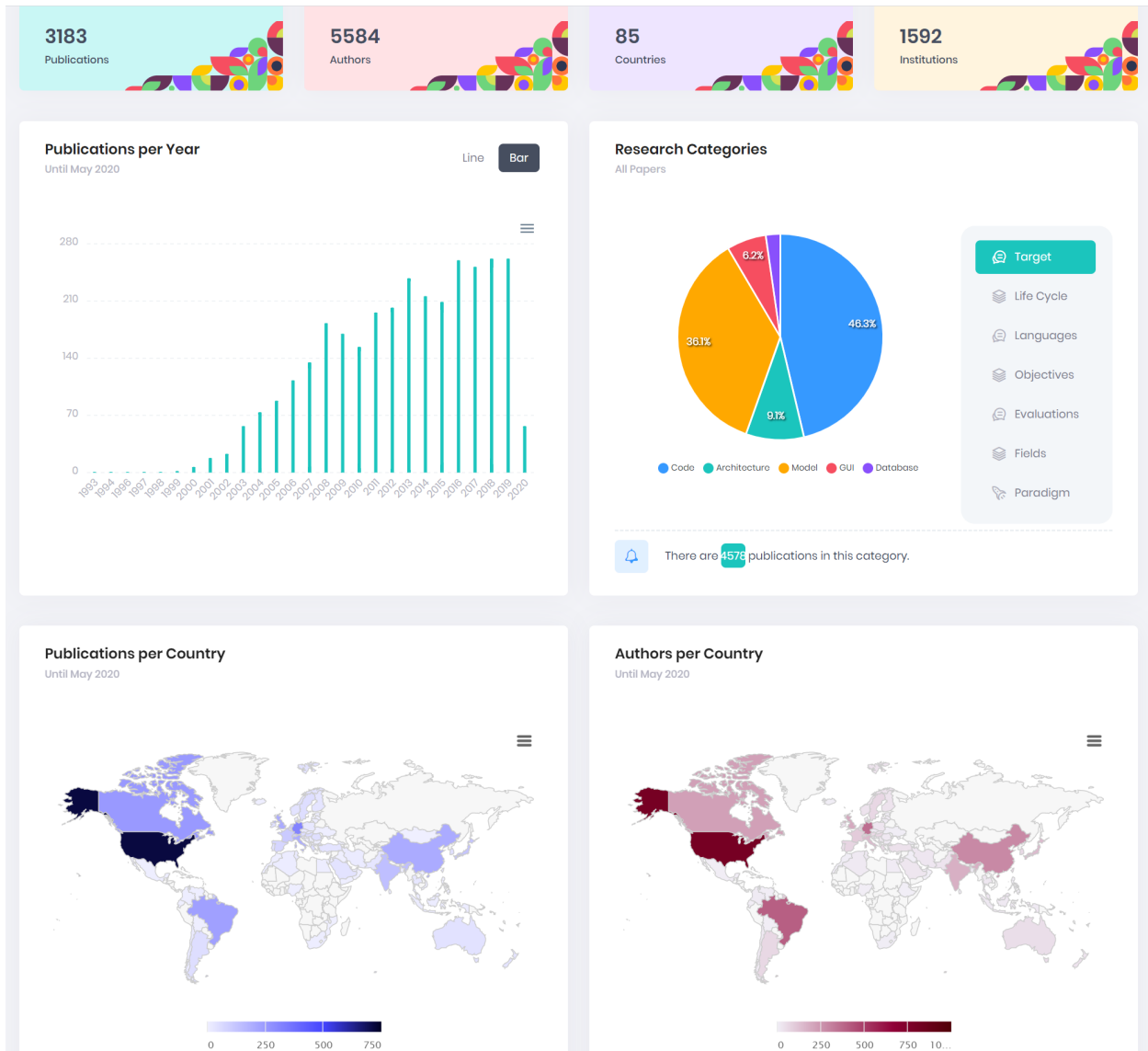
**Figure 2.4:** A screenshot of the publications tab of the refactoring repository Website

reaching a pick of 265 publications in 2016. During just the last four years (2016-2019), over 1026 papers were published in the field, with an average of 256 papers each year.

#### 2.1.4.1 Refactoring Life-cycle

Based on the current studies, the refactoring life-cycle can be decomposed into six stages:

- **Refactoring opportunities detection:** Identifying refactoring opportunities can be done by manually inspecting and analyzing an artifact of a system to identify quality issues. However, this technique is time-consuming and costly. Researchers in this area typically propose fully or semi-automated techniques to identify refactoring opportunities using the concepts of code smells, quality metrics, etc.
- **Refactoring prioritization:** The number of refactoring opportunities usually exceeds the number of problems that the developer can deal with, particularly when the effort available for performing refactorings is limited. Moreover, not all refactoring opportunities are equally relevant to the goals of the developers when improving the quality.
























**Figure 2.5:** Dashboard of the refactoring repository website

In this stage, the refactorings operations are prioritized using different criteria (e.g., maximizing the refactoring of classes with a large number of anti-patterns or with the previous history of bugs, etc.) according to the needs of developers.

- **Refactoring recommendation:** Several refactoring recommendation tools have been proposed that dynamically adapt and suggest refactorings to developers. The output is sequences of refactorings that developers can apply to improve the quality of systems by fixing, for example, code smells or optimizing security metrics.

**Top Authors**  
Software Refactoring Research Report Author Information

AUTHOR	REFACTORING PUBLICATIONS			TOTAL PUBLICATIONS		PROFILES
	Publications	h-index	Citations	h-index	Citations	
 <b>Marouane Kessentini</b> University of Michigan-Dearborn United States	43	27	1582	35	2705	 
 <b>Danny Dig</b> Oregon State University United States	39	24	2329	35	4414	 
 <b>Steve J. Counsell</b> Brunel University London United Kingdom	36	12	490	34	7089	 
 <b>Katsuro Inoue</b> Osaka University Japan	32	12	634	40	8542	 
 <b>Gabriele Bavota</b> Università della Svizzera italiana Switzerland	29	19	1395	43	5604	 
 <b>Rohit Ghayi</b> Universidade Federal de Campina Grande Brazil	27	17	939	23	1922	 
 <b>Chanchal Kumar Roy</b> McGill University Canada	27	15	783	36	6113	 

**Figure 2.6:** A screenshot of the refactoring repository dashboard that shows the authors, their h-index and total number of publications and citations

- **Refactoring testing:** After choosing the refactorings to be applied, test cases need to be executed to ensure the correctness of artifacts transformations and avoid future bugs. This step includes checking the pre-and post-conditions of the refactoring operations and the preservation of the system behavior.
- **Refactoring documentation:** After applying and testing the refactorings, it is critical to document the refactorings, their locations, why they have been applied, and the quality improvements.
- **Prediction:** It is interesting for developers to know which locations are likely to demand refactoring in future releases of their software products. This step will help them focus on the relevant artifacts that will undergo changes in the future, prepare them for further improvements and extensions of functionality, and optimize the management of limited resources and time. Predicting locations of future refactoring can be, in general, done using the development history.

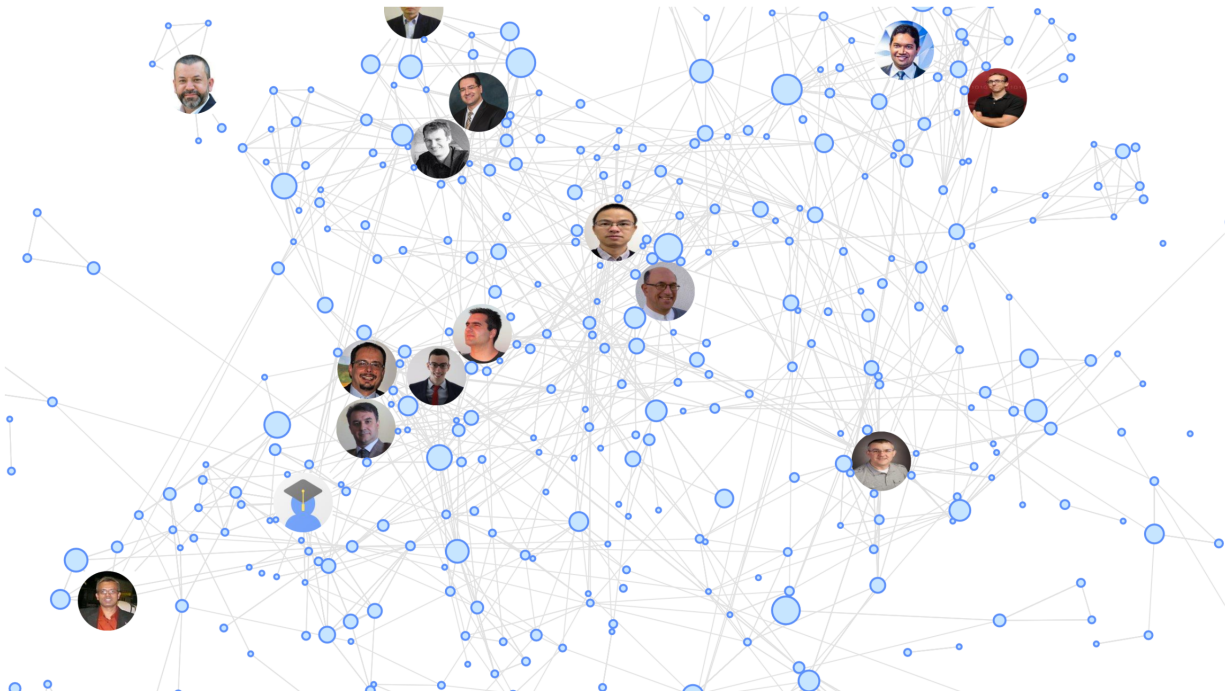
Figure 2.13 illustrates the percentage of the papers related to each stage of the refactoring life-cycle. About 33.08% of the papers deal with testing the refactorings. Researchers

### Authors Network Graph

Collaboration Network of All Refactoring Authors

Node size: # of refactoring publications - click on a node to see the relationships.

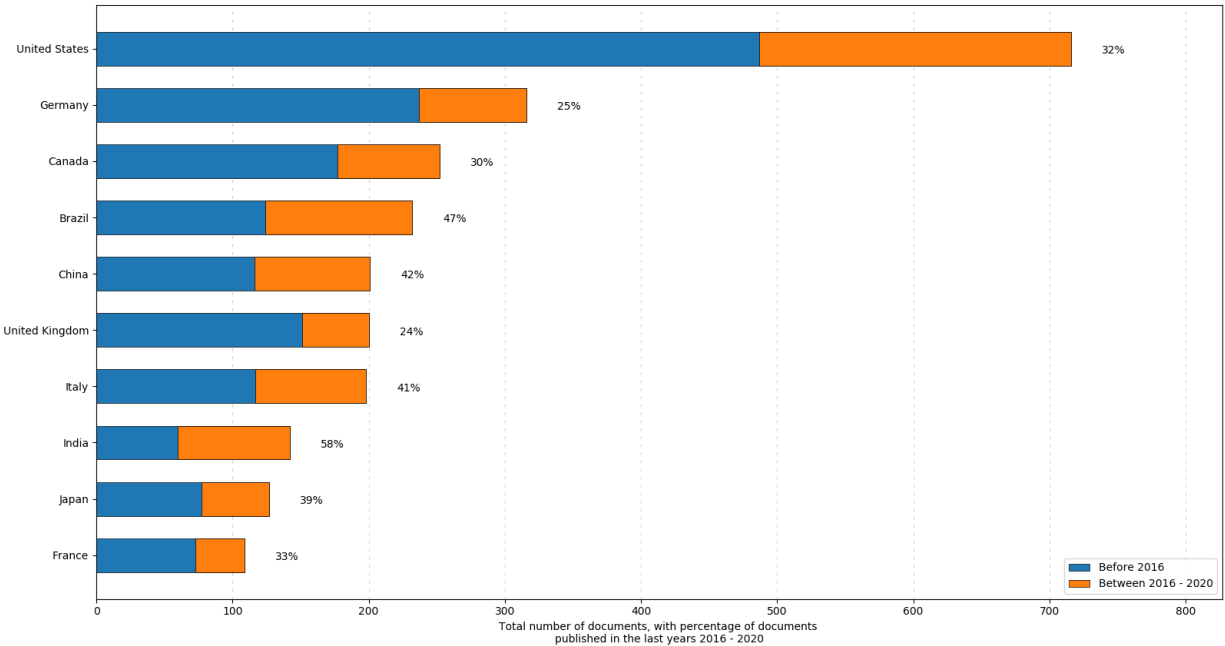
Select a node to see the link(s) to the author profile.



**Figure 2.7:** A screenshot of the authors network graph from the refactoring repository website

have invested heavily in testing to ensure the reliability of refactoring because changing the structure of code can easily introduce bugs in the program and lead to challenging debugging sessions. Plenty of effort is made towards the automation of the testing process to facilitate the adoption of refactoring [97, 98, 99]. Detecting refactoring opportunities is also a topic of interest to researchers. Several approaches have been proposed to detect refactoring opportunities including but not limited to techniques that depend on quality metrics (e.g., cohesion, coupling, lines of code, etc.), code smells (e.g., feature envy, Blob class, etc.), Clustering (similarities between one method and other methods, distances between the methods and attributes, etc.), Graphs (e.g., represent the dependencies among classes, relations between methods and attributes, etc.), and Dynamic analysis (e.g., analyzing method traces, etc.). Refactoring documentation is an under-explored area of research. Only 5.22% of the collected papers dived into refactoring documentation. Many studies examined the automation of the different refactoring stages to reduce the refactoring effort and, therefore, increase its





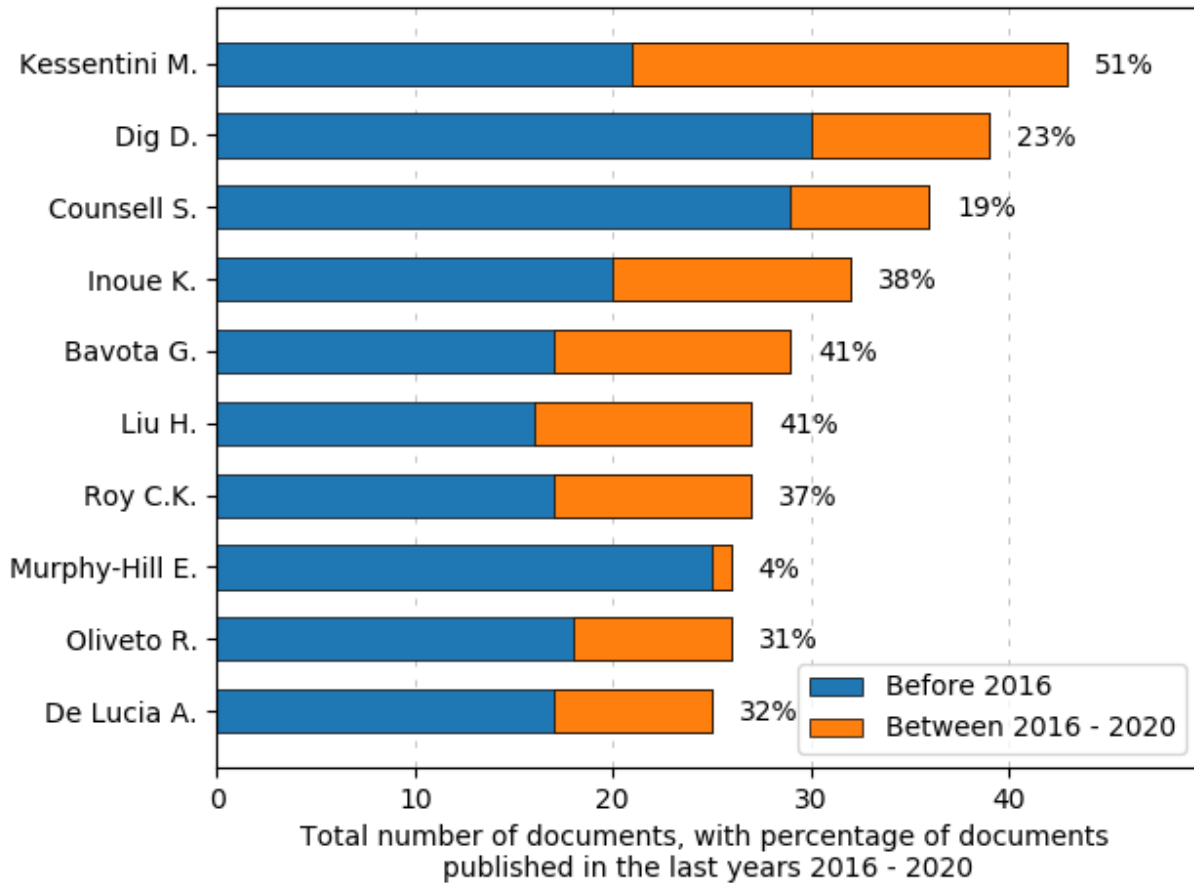
**Figure 2.9:** Number of publications in the top 10 most active countries in the refactoring field

must maintain informational semantics. Also, GUI refactoring is very demanding, requiring the adoption of user interface architectural patterns from the early software design stages. Future research should explore database and user interface refactoring further as they are an indispensable part of today’s software.

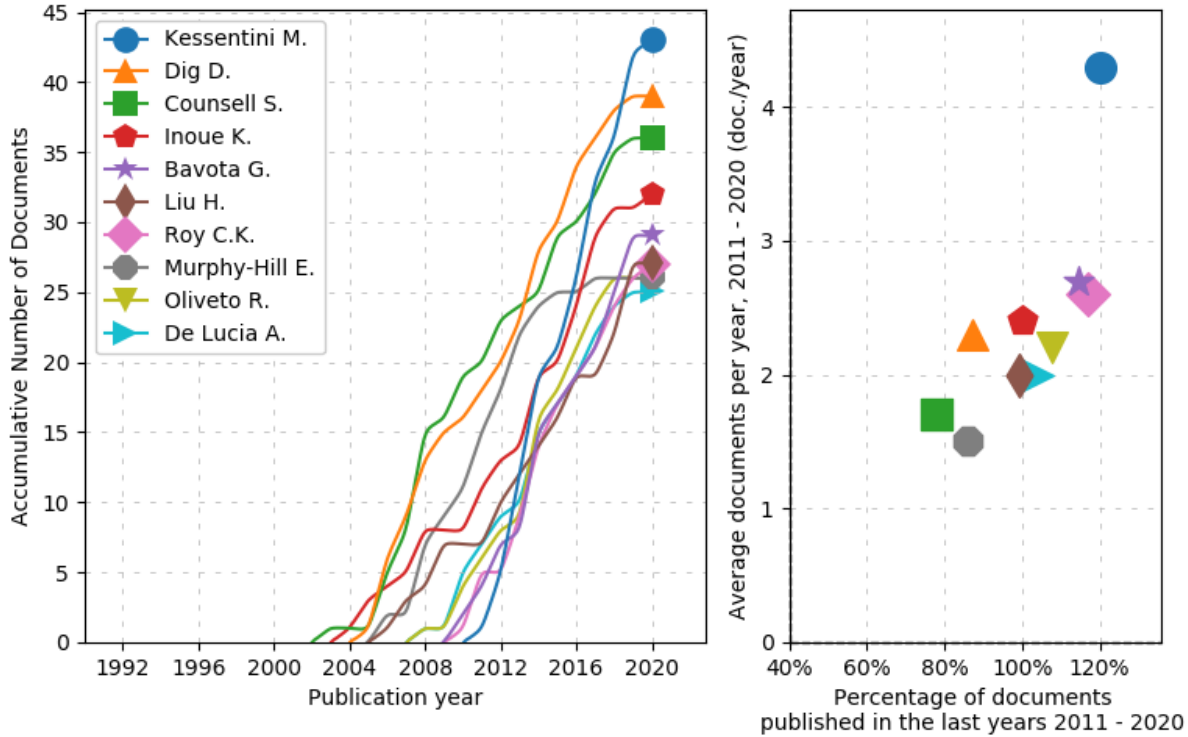
### 2.1.4.3 Refactoring Objectives

Five paradigms have been identified from analyzing the primary studies: object-oriented designs, cloud computing, mobile apps, model-driven, and aspect-oriented. Object-oriented programming has gained popularity because it matches the way people actually think in the real world, structuring their code into meaningful objects with relationships that are obvious and intuitive. The increased popularity of the object-oriented paradigm has also increased the interest in object-oriented refactoring. This can be observed in figure 2.16 where more than 34% of the studies related to refactoring focus on object-oriented designs. Less than 5% of the papers investigated refactoring for cloud computing and mobile app development. For the refactoring objectives classification of the taxonomy, five subcategories are considered:





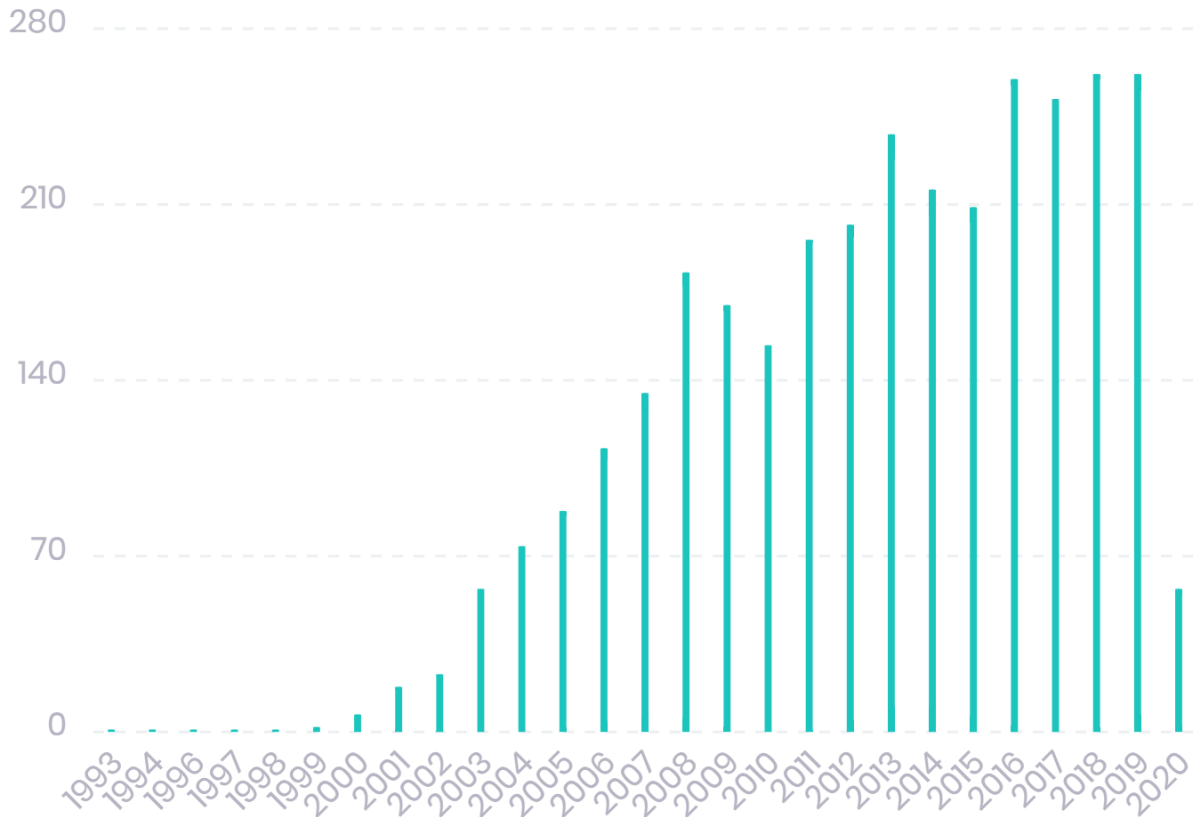
**Figure 2.10:** Top 10 Authors with the highest number of publications and citations in the field of refactoring external quality (e.g. correctness, usability, efficiency, reliability, etc.) , internal quality (e.g. maintainability, flexibility, portability, re-usability, readability etc.) , performance (e.g. response time, error rate, request rate, memory use, etc.), migration (e.g. Dispersion in the Class Hierarchy, number of referenced variables, number of assigned variables etc. ), security (e.g. time needed to resolve vulnerabilities, Number of viruses and spams blocked, Number of port probes, number of patches applied, Cost per defect, Attack surface etc.). Figure 2.17 is illustrating the reasons why people refactor their systems. Improving the internal quality takes up the largest portion (41.63%) followed by refactoring to improve the external quality (22.68%). Although security is a major concern for almost all systems, only 3.11% of the papers investigated refactorings for security reasons.



**Figure 2.11:** Evolution of the Top 10 Authors during the past 10 years

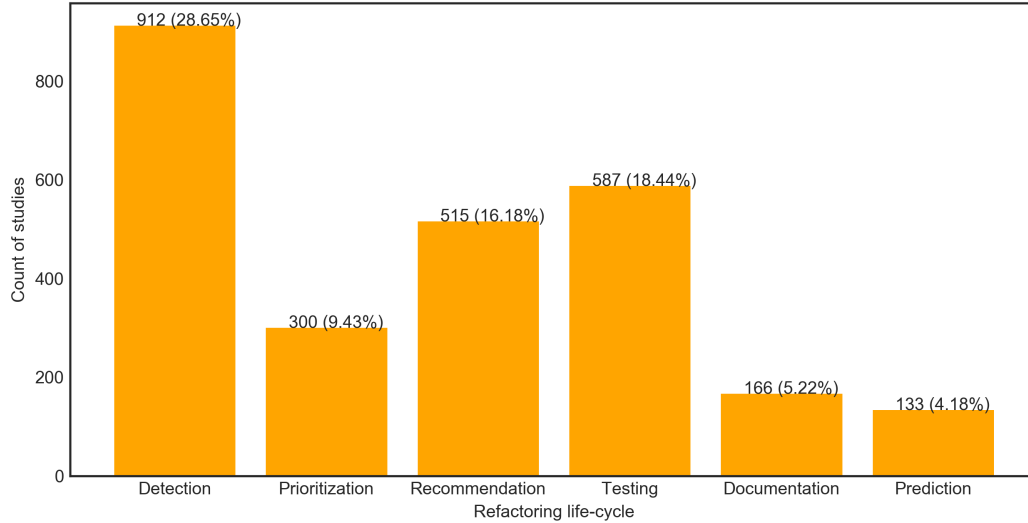
#### 2.1.4.4 Refactoring Techniques

Object-oriented programming languages have common traits/properties that facilitate the development of widely automated source code analysis and transformation tools. Many studies [261] have given sufficient proof that a refactoring tool can be built for almost any object-oriented language (Python, PHP, Java, and C++). Support for multiple languages in a refactoring tool is mentioned by [322]. Java is probably the most commercially important recent object-oriented language with an infrastructure that is designed to support analysis. It has generic parsing, tree building, pretty printing, tree manipulation, source-to-source rewriting, attribute grammar evaluations, control, and data flow analysis. This explains the fact that 17.15% of refactoring studies (see figure 2.18) provided refactoring techniques and tools that support Java. At the same time, most of the other programming languages have a fraction of less than 1%. We classified the refactoring techniques into four main categories: data mining (e.g., Clustering, Classification, Decision trees, Association, Neural



**Figure 2.12:** Trend of publications in the field of refactoring during the last three decades.

networks, etc.), search-based methods (e.g., Genetic algorithms, Hill climbing, Simulated annealing, Multi-objective evolutionary algorithms, etc.), formal methods (B-Method, the specification languages used in automated theorem proving, RAISE, the Z notation, SPARK Ada, etc.), and fuzzy logic. More than 25% of the papers use Search-based techniques to address refactoring problems (see figure 2.19). This can be explained by the fact that search-based approaches have been proven to be efficient at finding solutions for complex and labor-intensive tasks. With the growing complexity of software systems, there's an infinite amount of improvement/changes you can make to any piece of artifact. Exact algorithms are hard to use to solve the refactoring problem within an instance-dependent, finite run-time. That's why finding optimal refactoring solutions are sacrificed for the sake of getting perfect solutions in polynomial time using heuristic methods like search-based algorithms. Data mining techniques have also received significant attention (17.59%) as they are known to be

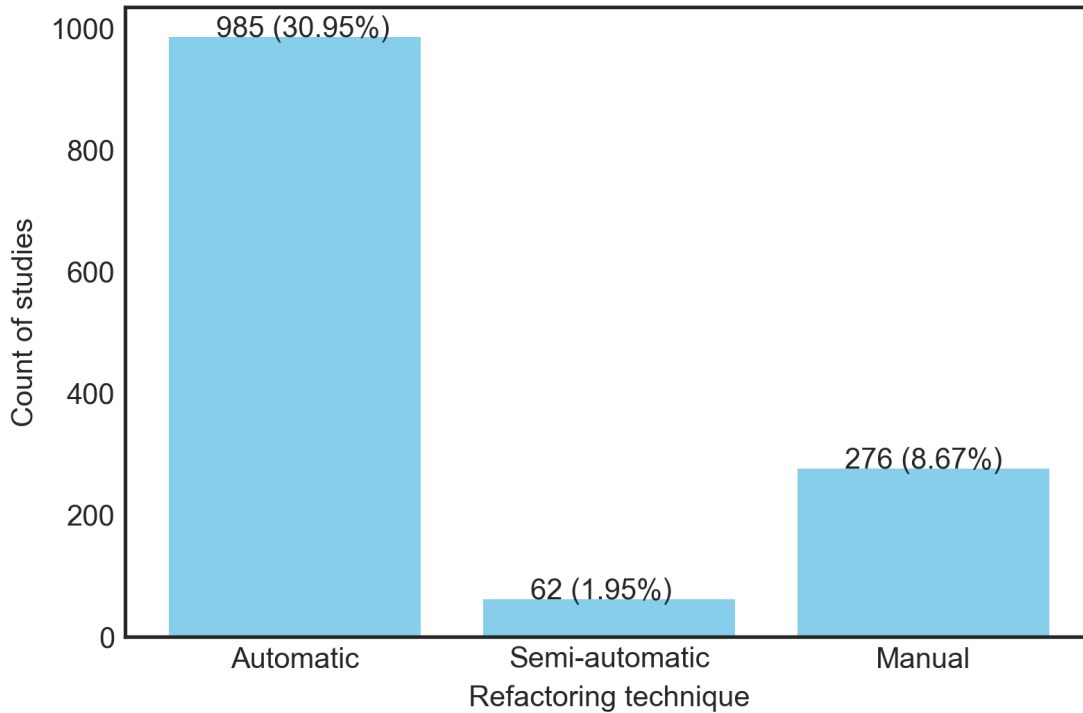


**Figure 2.13:** Histogram illustrating the percentage of refactoring publications per refactoring life-cycle

efficient at discovering new information, such as unknown patterns or hidden relationships, from huge databases like, in our case, large code repositories.

#### 2.1.4.5 Refactoring Evaluation

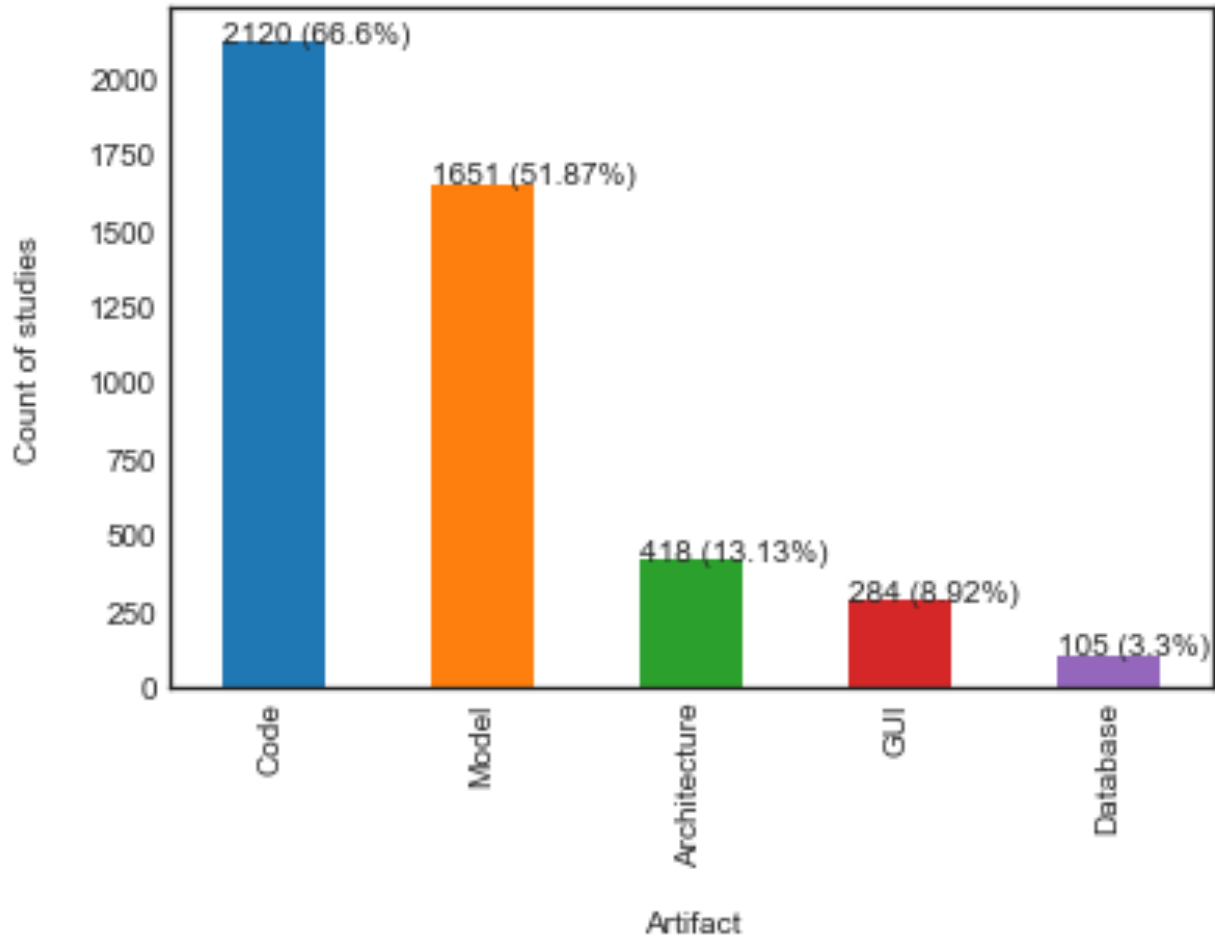
Open-source software systems are becoming increasingly important these days. 61.1% of the studies (see figure 2.20) used open-source systems to validate their work compared to 38.9% of studies that validated their work on industrial projects. This result is expected because of the availability and accessibility of open source systems. However, open-source software is often developed with a different management style than the industrial ones. Thus, refactoring techniques and tools must be validated and checked for quality and reliability using industrial systems. More industrial collaborations are needed to bridge the gap between academic research and the industry’s research needs, and therefore, produce groundbreaking research and innovation that solves complex real-world problems.



**Figure 2.14:** Histogram illustrating the percentage of publications dealing with manual, semi-automatic and automated refactoring

### 2.1.5 Future Research Directions

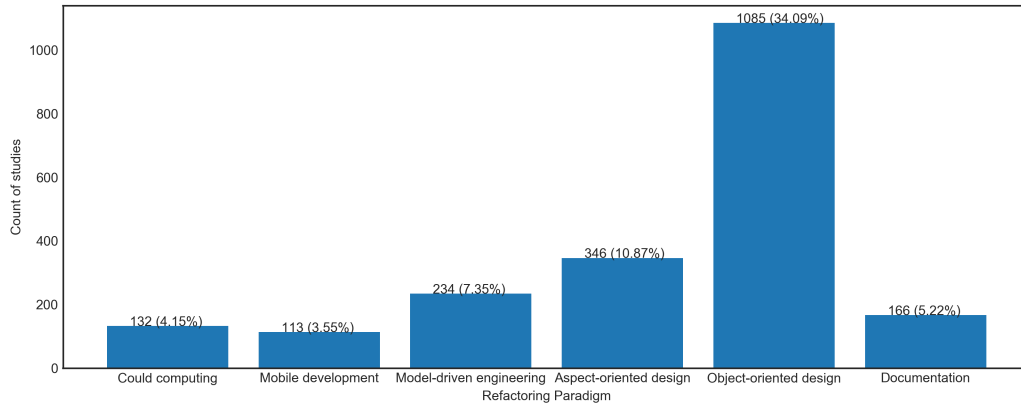
In this section, we identified new opportunities for future research directions related to refactoring based on the outcomes of the systematic literature review. One important observation from the obtained results of 30 years of refactoring research is that the core definition of refactoring dramatically changed over time. The recent and future research directions in the field require relaxing the behavior preservation constraints and going beyond simply changing the code structure. Thus, we proposed the following new definition of refactoring out of this systematic literature review that can be aligned as well with the current and future research directions: Refactoring can be defined as the automation, insight, testing, and prioritization of changes to the artifacts of software to improve non-functional requirements which may change part of its intended behavior.



**Figure 2.15:** Histogram illustrating the count of refactoring publications per artifact

### 2.1.5.1 Refactoring Bots

Many software organizations have moved toward adopting Continuous Integration (CI) processes, allowing teams to deliver features and detect problems rapidly. These development processes, such as DevOps, are based on frequent small releases, which change how systems are built as compared with prior discrete integration processes. While testing in CI has received much attention, the detection and correction of quality issues in CI lifecycles is not well-explored especially for embedded software as shown in this SLR study. In particular, researchers and practitioners lack a clear understanding of how refactoring tools should be adopted for CI. Existing refactoring tools are challenging to configure and integrate into development pipelines because they are adopted for discrete integration lifecycles and tend



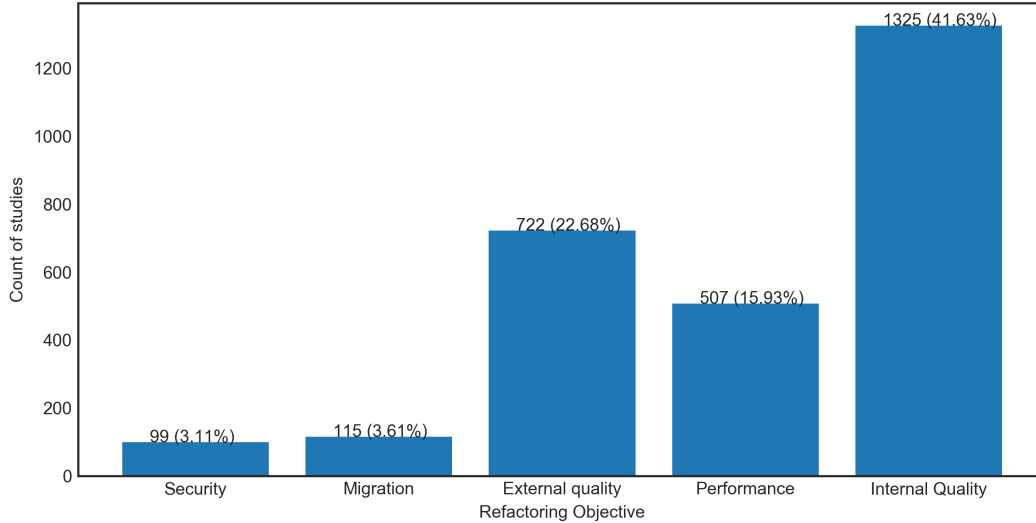
**Figure 2.16:** Histogram illustrating the count of refactoring publications per paradigm

to disrupt development. Existing refactoring recommendation tools interrupt developers, who need to review recommended changes frequently, and these changes may be unrelated to their current focus and interests.

A research agenda in this direction can be (1) to design, implement, and validate usable artificial assistants for refactoring code and to fix quality issues that interact with developers in proactive ways, becoming a “real” member of the development team; (2) to make an artificial assistant for refactoring in CI intelligent by considering the profile and context of the developer (e.g., knowledge, past experiences, current tasks) when making recommendations via mining a large history of refactorings data; (3) to evaluate and refine an intelligent refactoring bot using a large number of open-source and industrial projects and conducting user studies and controlled experiments with professional developers. This intelligent refactoring bot can be built as an extension of existing refactoring tools currently limited to discrete integration.

### 2.1.5.2 Interactive Refactoring

In manual refactoring, the developer refactors with no tool support at all, identifying the parts of the program that require attention and performing all aspects of the code transformation by hand. Manual refactoring is very limited; several studies have shown that

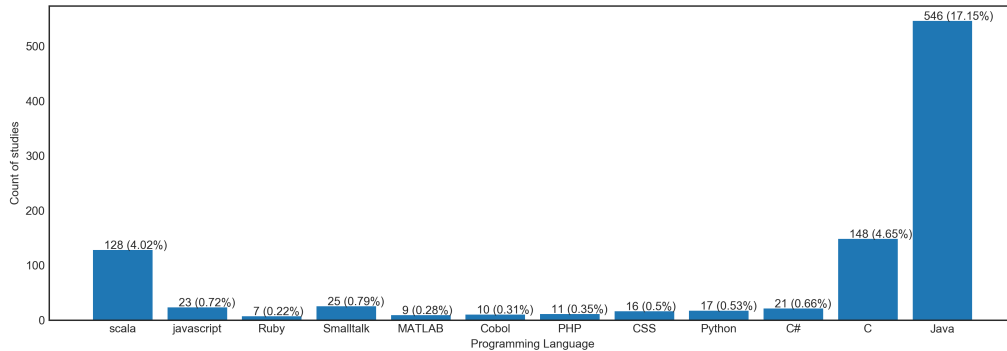


**Figure 2.17:** Histogram illustrating the count of publications per refactoring objective

manual refactoring is error-prone, time-consuming, not scalable, and not useful for radical refactoring that requires an extensive application of refactorings to correct unhealthy code.

In fully-automated refactoring, a search-based process is employed to find an entire refactoring sequence that improves the program in accordance with the employed fitness function (involving e.g., code smells, software quality metrics, etc.). This approach is appealing in that it is a complete solution and requires little developer effort, but it suffers from several serious drawbacks as well. Firstly, the recommended refactoring sequence may change the program design radically and this is likely to cause the developer to struggle to understand the refactored program. Secondly, it lacks flexibility since the developer has to either accept or reject the entire refactoring solution. Thirdly, it fails to consider the developer’s perspective, as the developer has no opportunity to provide feedback on the refactoring solution as it is being created. Furthermore, as development must halt while the refactoring process executes, fully-automated refactoring methods are not useful for floss refactoring where the goal is to maintain good design quality while modifying existing functionality. The developers have to accept the entire refactoring solution even though they prefer, in general, step-wise approaches where the process is interactive and they have control of the refactorings being





**Figure 2.18:** Histogram illustrating the count of refactoring publications per programming language

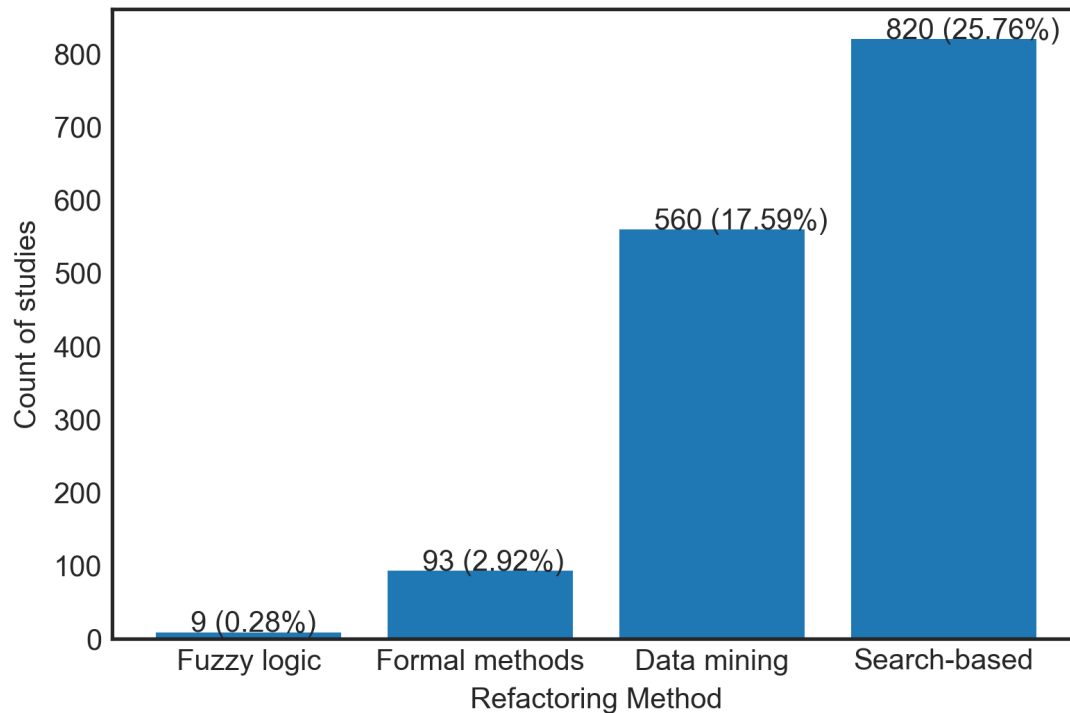
applied.

In light of the discussion above and the current limited work on interactive refactoring, the next generation of refactoring tools should (1) provide human-centric interaction for refactoring, (2) enable refactoring and development to proceed in parallel, and (3) collect information in a non-intrusive manner that can be used to inform dynamically the refactoring process. We postulate that enabling the developer to interact with the refactoring solution is essential both to creating a better refactoring solution and creating a solution that the developer understands and can work with.

Refactoring and development must be allowed to proceed in parallel, as this is part of test-driven development and the Agile approach to software development in general. Thus, the developer can continue to extend the program with new functionality or bug fixes while the refactoring recommendation process executes. Any development carried out can be used where possible to improve the refactoring recommendations, e.g., the developer is more likely to value refactorings that affect recently updated code.

### 2.1.5.3 Refactoring for Software Security

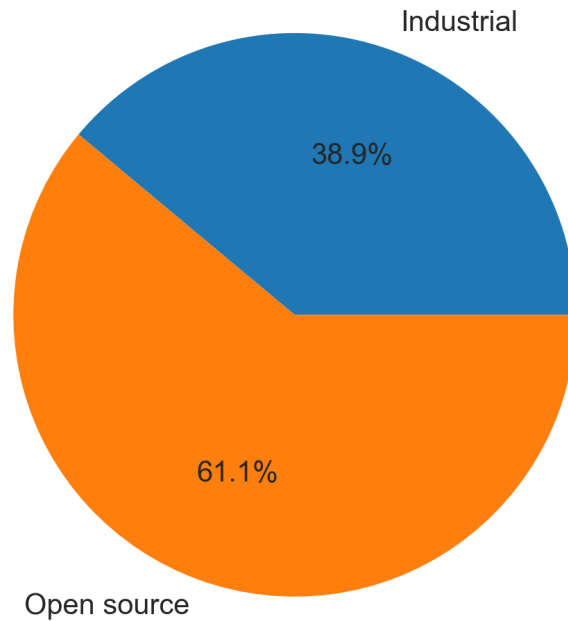
The ISO/IEC-25000 Software product Quality Requirements and Evaluation (SQuaRE) classifies software quality in a structured set of eight characteristics and sub-characteristics. In this classification, security is a new characteristic that was created to measure how much



**Figure 2.19:** Histogram illustrating the count of refactoring publications per field

a software is resistant to attacks and risks. Therefore, it is crucial to take this characteristic into account when improving the quality of the software via refactoring.

Several researchers and practitioners have assumed that improving a quality metric of software, such as modularity, will have a positive impact on security, making the design more robust and resilient to attacks. However, this assumption is poorly supported by empirical validations. Architects and developers may not pay much attention to design fragments containing data and logic pertinent to security properties, which makes them overexposed while still improving some quality aspects of their architecture. For instance, a developer may create a hierarchy in a set of classes to improve the reusability of the code. However, these actions may expand the attack surface if the superclass contains critical attributes and methods. Another example that we observed in practice is that improving modularity may result in spreading dependencies on security-critical files into many other components.



**Figure 2.20:** Pie chart illustrating the percentage of publications in which the authors used industrial and/or open source systems in the validation step

A security-critical file contains data (e.g., attributes) and logic (e.g., methods) that can potentially be misused to violate fundamental security properties such as confidentiality, integrity, or availability of a system.

As shown in this study, most existing refactoring research focuses on handling conflicting quality attributes. However, the impact of refactoring on security is poorly understood and under-studied. Recent studies estimate the impact of a few refactoring operations on some security metrics based on their definitions, but without empirically validating these assumptions on real software projects. Thus, it is important that the next generation of refactoring tools should consider enhancing the resilience of software applications while improving traditional software quality aspects.

#### 2.1.5.4 Refactoring Documentation

The effective understanding and documentation of refactorings can play a critical role in reducing and monitoring the *technical debt* by different stakeholders including executives, managers, and developers. In particular, refactoring documentation can help developers, managers, and executives keep track of applied refactorings, their rationale, and their impact on the system.

The commit messages and the pull-requests descriptions are becoming the most common ways to document code changes, including refactoring. Modern collaborative coding platforms (e.g. GitHub), have advocated for the development of automated recommendation systems to generate commit and pull-request messages. Thus, several automated techniques for the generation and recommendation of documentation of diffs and atomic changes have been recently proposed. However, most of the current development workflows/pipelines in the industry are lacking tools/steps to document refactorings and quality changes/technical debt. To the best of our knowledge, there are no standards to document refactorings or any prior empirical studies about understanding refactoring documentation. The current set of commonly used tools offer to see and document diff-changes but not dependent on atomic changes/diffs.

We advocate that a critical and fundamental step in providing efficient support for developers in documenting refactoring is to discover the specific pieces of information, called components, that are necessary to include in commit messages to describe introduced refactorings.

#### 2.1.6 Conclusion

In this contribution, we have conducted a systematic literature review on refactoring accompanied by meta-analysis to answer the defined research questions. After a comprehensive search that follows a systematic series of steps and assessing the quality of the studies, 3183 publications were identified. Based on these selected papers, we derived a taxonomy focused

on five key aspects of Refactoring: refactoring life-cycle, artifacts affected by refactoring, refactoring objectives, refactoring techniques, and refactoring evaluation. Using this classification scheme, we analyzed the primary studies and presented the results in a way that enables researchers to relate their work to the current body of knowledge and identify future research directions. We also implemented a repository that helps researchers/practitioners collect and report papers about Refactoring. It also provides visualization charts and graphs that highlight the analysis results of our selected studies. This infrastructure will bridge the gap among the different refactoring communities and allow for more effortless knowledge transfer.

The results of our systematic review will help both researchers and practitioners to understand the current status of the field, structuring it, and identify potential gaps. Since we expect this research area to continue to grow in the future, the proposed repository and taxonomy will continue to be updated by the organizers of this study and the community to include new approaches, tools and researchers.

## 2.2 What Refactoring Topics Do Developers Discuss? A Large Scale Empirical Study Using Stack Overflow

### 2.2.1 Introduction

Given the current growth of refactoring research with more than 3000 peer-reviewed papers published in the last decade, the gap is growing larger between research and practice. Is the research community paying attention to the needs of developers? What informs the design of new refactoring technology? We believe it is crucially important to understand the current trends in the field, the challenges that developers face when refactoring in the wild, and the most discussed refactoring topics on developer forums. Without such understanding, tool builders invest in the wrong direction, and researchers miss many opportunities for improving the practice of refactoring. We need to understand the new drivers for refactoring innovation from the practitioners' vantage point.

In this dissertation, we performed the first large scale refactoring study on the most popular online Q&A forum for developers, Stack Overflow. Developers use the forum to seek help and advice from their peers about the technical challenges they face in different development topics. Stack Overflow moderates millions of posts from developers, with different backgrounds, asking questions about a wide range of topics including refactoring. The analysis of the discussed topics in this repository could provide various key insights about the topics of interest to the developers related to refactoring such as the most addressed quality issues, the domains where refactoring is extensively discussed, the preferred level abstractions, the widely addressed anti-patterns, and patterns. Recent studies analyzed Stack Overflow posts in several areas including software security [323], mobile apps [324, 325, 326], and more general programming topics [327, 328] and came up with useful recommendations. We believe applying a similar approach for studying refactoring needs could be equally useful.

The analysis of Stack Overflow refactoring posts is beneficial to developers, researchers, and educators in different ways. Developers can educate themselves about the common issues

that others have faced so they can learn about the peer-best-practices. Researchers can use this analysis to understand the real problems faced by programmers in refactoring. Finally, educators may use the result of these analyses to update their courses and focus on the main weaknesses in the background of programmers that may need to be addressed. The mapping between Stack Overflow discussions and existing research topics helps us and others identify the gap between the practitioners and research communities.

Stack Overflow contains more than 42 million posts and associated attributes such as questions, answers, tags that are most representative of the post etc [329, 330]. We first selected the posts related to refactoring by choosing a list of tags such as "refactoring", "anti-patterns", etc. Then, we used an advanced topic model based on, LDA, to identify the topics. Using this data, we answer the following five research questions:

- **RQ1. What questions and issues related to refactoring are developers discussing?** We found that developers are interested in six main topics related to refactoring which are Creational pattern, Parallel programming, Models Refactor, mobile/UI, Service-Oriented Architecture (SOA), and Design pattern (Section 2.2.4.1).
- **RQ2. What are the most popular topics among the questions related to refactoring?** Our results show that Creational Pattern topic has the largest popularity while parallel programming, and mobile/user-interface topics have the lowest (Section 2.2.4.2).
- **RQ3. Which refactoring-related topics are the most difficult to answer?** Design patterns topic has the lowest rate of questions with unsatisfactory answers. It also has the lowest average number of views without a relevant answer. The model refactoring is the topic that was the least answered by developers(Section 2.2.4.3)
- **RQ4. How do the interests of developers on refactoring topics change over time?** SOA and Design patterns are the refactoring topics that have the highest evolution in the number of questions throughout the years (Section 2.2.4.4)

- **RQ5. What are the implications of our empirical study on practitioners, educators, and researchers?** Our study helps researchers focus on practical refactoring problems, practitioners know more about current challenges and build better refactoring tools, and educators revise curriculum to target current needs on refactoring (Section 2.2.5).

### 2.2.2 Stack Overflow Data Description

Stack Overflow is a question and answer website used by beginners as well as professionals belonging to stack-exchange network. It has the largest community compared to the other Q&A websites in the Network. Stack Overflow was launched on September 15, 2008 and it kept growing in popularity. Nowadays more than 17 million questions were posted in Stack Overflow, and an average of 5956 questions was asked per day in the last four years.

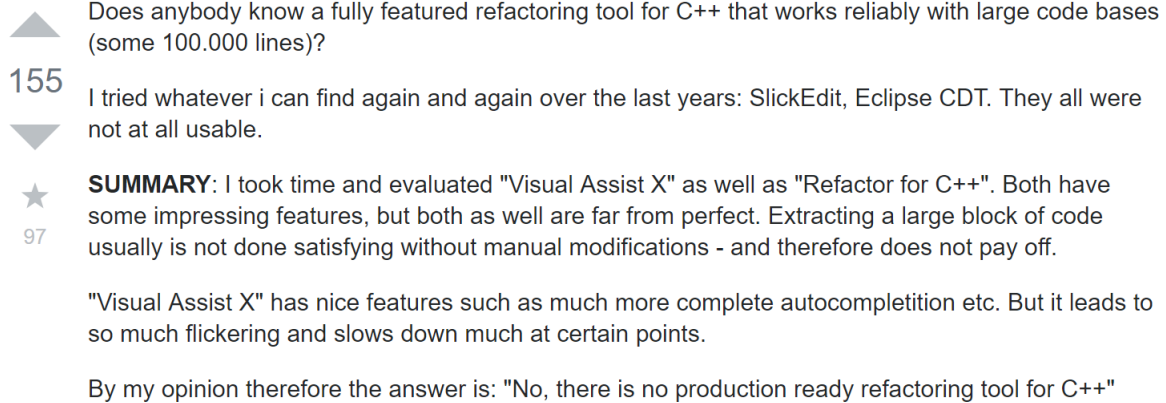
There are currently 54637 tags in Stack Overflow. Among them, the tag "Javascript" holds the biggest number of related questions which exceed 1700000 whereas the "refactoring" tag helps to identify 6445 questions. Figure 2.21 shows one example of a refactoring question on Stack Overflow with the title "Is there a working C++ refactoring tool?". Two tags are used for this post: "refactoring" and "C++". Furthermore, several metadata are related to a post such as the edit date, the number of views, etc.

To easily access Stack Overflow data, one of the best ways is based on Stack-exchange data-dumps. They represent collections of data archived by the Stack-exchange community. The data is collected yearly and uploaded on the archive.org website. The data-set is divided into several XML files which are Posts.xml, Users.xml, Votes.xml, Comments.xml, PostHistory.xml, and PostLinks.xml. In this study, we have used Posts.xml which contains around 42 million posts. There are five types of posts: Question, Answer, Orphaned Tag Wiki, Tag Wiki Excerpt, and Tag Wiki.

Each type of post can be filtered using the PostTypeID attribute. Besides the PostTypeID, each post has 21 defined attributes which could have value or not depending on



## Is there a working C++ refactoring tool?



Does anybody know a fully featured refactoring tool for C++ that works reliably with large code bases (some 100.000 lines)?

155  
I tried whatever i can find again and again over the last years: SlickEdit, Eclipse CDT. They all were not at all usable.

★  
97  
**SUMMARY:** I took time and evaluated "Visual Assist X" as well as "Refactor for C++". Both have some impressive features, but both as well are far from perfect. Extracting a large block of code usually is not done satisfying without manual modifications - and therefore does not pay off.

"Visual Assist X" has nice features such as much more complete autocompletion etc. But it leads to so much flickering and slows down much at certain points.

By my opinion therefore the answer is: "No, there is no production ready refactoring tool for C++"

**Figure 2.21:** Refactoring post found on Stack Overflow

the type of the post. Table 2.6 shows the different attributes defined in the posts.xml file. Of course, one of the main attributes is the tag used to classify the question. We will give details in the next section how we identified the tags related to refactorings to filter the Stack Overflow posts.

In the future, we are planning to expand our data-set to include all the Stack Overflow data available in the "archive.org" website. We will also work on a survey with practitioners from multiple programming domains to qualitatively evaluate the outcomes of the Stack Overflow analysis performed in this contribution.

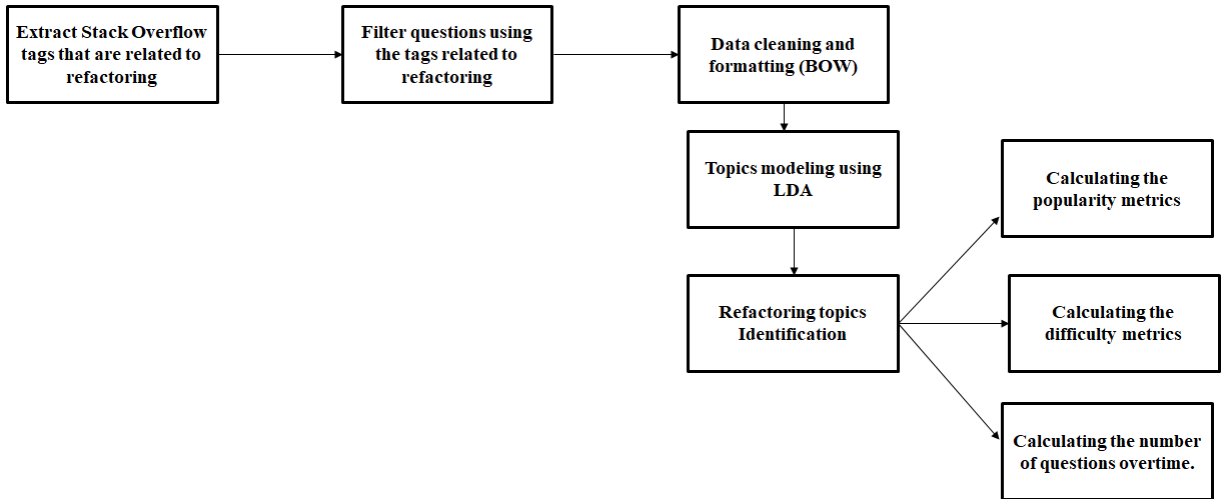
### 2.2.3 Research Method

The main goal of this study is to identify the main refactoring-related topics discussed by developers, highlighting the most popular ones and the most difficult ones and understanding developers' interest trends. We will describe, in this section, the details of the steps adopted to achieve the main goals of this study. In the remainder of the contribution, we will use "document" to refer to a question and "corpus" to refer to the set of questions.

Figure 2.22 summarizes the different main steps of our Stack Overflow analysis. The first step consists of identifying the list of tags related to refactoring. The second step filters the

**Table 2.6:** Attributes describing a Stack Overflow post

<b>Attribute</b>	<b>Description</b>
Id	represent a unique id for posts
PostTypeId	Digit that define the type of the post
AcceptedAnswerId	If the post is a question and there is accepted Answer to this question this field will contain the accepted Answer id
ParentId	If the post is an answer this field contains the question id of that answer
CreationDate	This field contains the date time creation of the post eg.:"2008-09-06T08:07:10.730"
DeletionDate	it is defined if the post was deleted and it has the same form as the creation date
Score	this is an integer that represents the upvotes giving to the post. It represents how helpful was the post
ViewCount	This is defined for questions, and it represents how many people viewed the post
Body	Text of the posts
OwnerId	The id of the post owner
OwnerDisplayName	the name of the post owner
LastEditorUserId	Id of the last user that modified the post
LastEditorDisplayName	the name of the user that modified the post
LastEditDate	The date of the last post update
LastActivityDate	The date of last Activity related to this post
Title	The title of the post is not an answer
Tags	comma separated strings that list all the tags for the post (defined only for a question)
AnswerCount	defined for a question represent the number of answers related to this question
CommentCount	represent the number of comments for the post it is defined for the question and answer
FavoriteCount	defined only for question post, and it represents the number of users that liked the post
ClosedDate	Defined if moderators of the website closed the post
CommunityOwnedDate	the date when the post was converted to community wiki



**Figure 2.22:** An overview of our Stack Overflow analysis.

list of questions using the selected tags. The third main step runs LDA to identify the list of topics related to refactoring by mining the selected questions and answers. Finally, we answered several questions about these topics including trends, difficulty, and evolution over time.

To select the refactoring related documents, we extracted refactoring related tags and filtered the documents dataset using these tags. Then, we pre-processed each document for the LDA topic modeling approach by cleaning it and translating it into a vector of features using the Bag of Words (BOW) representation [331]. We used LDA topic modeling approach because it was widely used in similar problems [323, 327, 324] and it was proven to be able to generate topics that are highly interpretable and provide deep insights to the data.

To filter the questions, we used multiple steps. First, we manually defined an initial list of tags including 10 words: refactoring, design patterns, architecture, anti-patterns, code-cleanup, software-design, software-quality, code-metrics, automated-refactoring. The manual definition of tags is limited and may not cover all the relevant refactoring questions. For instance, some posts are related to refactoring but are not tagged with the refactoring tag in several cases. In order to extract more tags using the initial set of tags, we extracted all the tags defined in Stack Overflow, and for each extracted tag, we assessed to what extent it is relevant and related to the initial tag list. Therefore, we used two heuristics taking

inspiration from a similar study [323]. These heuristics are based on  $a(t)$  : the number of questions that contain both tag  $t$  and a refactoring related tag (one of the above 10 words),  $b(t)$ : number of questions that contains the tag  $t$ , and  $c(t)$ : the number of questions that contain a refactoring related tag (one of the above 10 words).

- The first heuristic H1 is defined by the ratio of the number of questions that contain both the tag and a refactoring related tag to the number of questions that contain the tag  $t$ .  $H1(t) = a(t)/b(t)$
- The second heuristic H2 is the ratio of the number of questions that contain both the tag and a refactoring related tag to the questions identified by the initial set of 10 tags.  $H2(t) = a(t)/c(t)$

We defined thresholds empirically for both heuristics to select relevant tags by trial and error: 0.08 for the first heuristic and 0.0004 for the second heuristic. We thus extracted 94 tags shown in the Table 2.7

After inspecting these 94 extracted tags, we removed 5 irrelevant tags. For instance, OOP(object-oriented programming) was one of the tags that we removed as it had the largest number of questions which is 46618, most of them not being related to refactoring. We finally considered 89 tags which we used to extract 105,463 questions for this empirical study. We checked the relevance of these tags being chosen via validating random samples from the included documents to make sure they are all relevant.

In order to identify discussed refactoring related topics, we have used a topic modeling technique: Latent Dirichlet Allocation (LDA) [332]. Topic modeling is an approach aiming at finding patterns of words in document collections using hierarchical probabilistic models. Topic modeling may be used to classify the documents of the corpus by discovered latent topics. It specifies a procedure by which documents can be generated by choosing a distribution over topics. Each topic is a distribution that defines how likely each word may appear in a given topic. For more details about LDA, the reader can refer to [332].

**Table 2.7:** List of candidate tags

oop	design-patterns	design	architecture	singleton	refactoring
domain-driven-design	microservices	decorator	repository-pattern	factory	scalability
dao	soa	mvp	dry	observer-pattern	builder
cqrs	composition	data-access-layer	dto	software-design	factory-pattern
unit-of-work	class-design	abstraction	composite	solid-principles	idioms
modularity	n-tier	business-logic	strategy-pattern	code-duplication	separation-of-concerns
object-oriented-analysis	n-tier-architecture	code-cleanup	legacy-code	methodology	visitor
anti-patterns	ddd-repositories	service-layer	service-locator	3-tier	srp
bridge	domain-model	facade	decoupling	conceptual	automated-refactoring
command-pattern	mediator	maintainability	code-readability	design-principles	visitor-pattern
code-metrics	project-planning	oad	code-smell	cyclomatic-complexity	module-pattern
application-design	onion-architecture	abstract-factory	business-logic-layer	loose-coupling	lsp
factory-method	cocoa-design-patterns	coupling	software-quality	builder-pattern	revealing-module-pattern
clean-architecture	system-design	code-design	law-of-demeter	data-transfer-objects	open-closed-principle
chain-of-responsibility	template-method-pattern	multi-tier	proxy-pattern	architectural-patterns	n-layer
flyweight-pattern	memento	prototype-pattern	gang-of-four		

As the LDA model is expecting a frequency-weighted document-term matrix, we performed the following steps:

- First, we aggregated the values from the title, the body and the tag attributes and we removed all the useless meta-data.
- Second, we removed the code snippets and all the HTML tags.
- Third, we tokenized and removed any useless special characters like punctuation and characters that do not belong to English alphabet except for '\_' and '-' which are used to join two relevant words together.
- Fourth, we removed stop-words: very common words used in the English language which are not relevant for the clustering of the documents; for example ('do','like','what', 'I', 'they',...). Thus, we used the stop-word list provided by NLTK [333] and we added other stop-words that are not relevant for the clustering of refactoring related questions. We also removed words containing less than two characters.
- The fifth step was mainly for normalization based on lemmatization of words which reduces the noise in the data by removing inflectional endings and to return the base or dictionary form of a word, which is known as the lemma [334].
- Finally, we used an automated approach to determine the vocabulary words that we will use as features for BOW (Bag-of-words) representation. The technique consists of calculating the portion of documents that contain a specific word. Then, based on two thresholds we eliminated the very rare keyword that appears in less than 1% of the documents and the very frequent ones that appear in more than 80% of the documents as used in another similar study [323]. We translated the corpus into a TF-IDF matrix. The dimension of the matrix is  $M*N$  where  $M$  is the number of documents (105463),

and  $N$  is the number of words in the vocabulary (4872). The values in the matrix are calculated as  $TF * IDF$ :

$$Matrix(d, w) = \frac{frequency(w, d)}{number\_of\_words(d)} * IDF(w) \quad (2.1)$$

where  $w$  is the corresponding word,  $d$  is the corresponding document,  $frequency(w, d)$  is frequency of  $w$  in  $d$ , and  $number\_of\_words(d)$  is number of words in  $d$ .

$$IDF(w) = \log\left(\frac{\#\_of\_documents}{1 + \#\_of\_documents\_that\_contains\_w}\right) \quad (2.2)$$

This TF-IDF matrix was used in the LDA model to cluster the questions into topics.

## 2.2.4 Results

In this section, we summarize the results of the five research questions.

### 2.2.4.1 RQ1. What Questions and Issues Related to Refactoring Are Discussed by Developers ?

The LDA model identified six main topics discussed by developers. A set of keywords identified each of these topics. To better characterize each topic, we labeled it to match the set of keywords identified by LDA. Table 2.8 shows the six topics and keywords associated with them sorted by the relevance score of the LDA model.

Most of the words identified by LDA for the first topic are related to object creation: "singleton", "factory", "instance", "constructor" and "create". For the second topic, most of the words refer to parallel programming. This topic includes words like message, request, server, thread and observer. The third topic was related to model refactoring with many keywords about UML diagrams, requirements, and design issues. The fourth topic includes android and other words related to user interface. In fact, it is normal that most of the questions around Android apps are around refactoring the User Interface (UI) since it is

the most crucial part in mobile applications. The fifth topic, service-oriented architecture, includes words like service, user layer, database, and architecture. The high reusability of services in SOA architecture makes refactoring very important to simplify the code and makes it easy to understand. It also helps to achieve modularization at the application level. Finally, the design pattern topic was the last topic with mainly common words like design, pattern, code, singleton, etc. The questions that fall in this topic deal with code standard refactoring, specifically the application of general design pattern to achieve high code quality.

**Table 2.8:** The 6 refactoring related topics with the 10 most important words in each topic

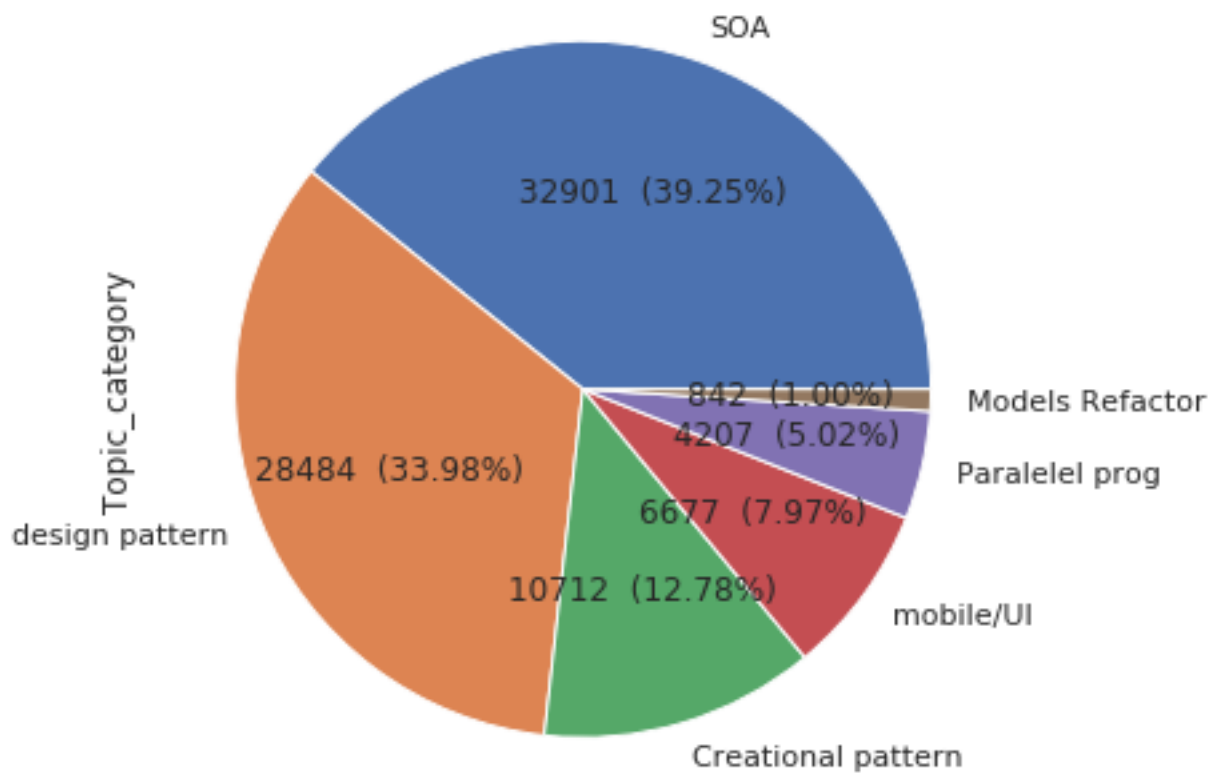
Topic	Words
Creational pattern	singleton, instance, method, factory, java, constructor, create, pattern, call, code
Parallel programming	message, server, observer, microservice, request, time, thread, java, client, connection, performance
Models Refactor	decorator, visitor, decorate, factory, co-evolution, re-design, objects, extract, UML
mobile/UI	Android, view, button, image, presenter, design, page, HTML, text, color
SOA	service, availability, model, coupling, micro-service, layer, repository, interface, database, architecture
Design pattern	design, pattern, principles, hierarchy, reusability, extend

We may highlight that the creational pattern is a specific type of design pattern similar to well-known design patterns like observer and decorator patterns. These patterns were extensively discussed in the refactoring posts on Stack Overflow.

Figure 2.23 shows the number of questions per dominating topic: it includes questions with a higher probability than 0.5 to belong to a topic based on the LDA output. We notice that the largest number of questions about refactoring is dedicated to SOA architecture. The creational patterns also have a high number of questions even if it is only a sub-type of the design patterns. Although the number of questions about parallel programming was initially small, there is a massive growth in the number of questions asked during the last few years about refactoring for parallel programming.

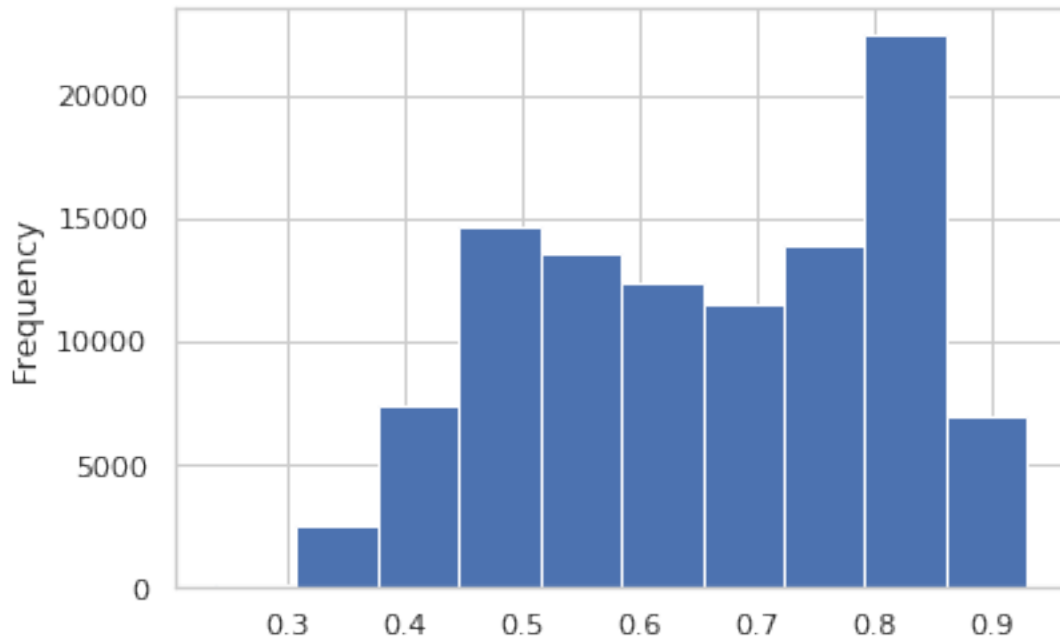
Figure 2.24 shows the distribution of the number of questions per dominant topic.





**Figure 2.23:** Distribution of the number of questions per refactoring topic

According to this figure, 79% of the questions have a dominant topic, and more than 25% of them have 0.8 probability of belonging to their dominating topic. When we have a dominant topic for a question, it does not mean that the question cannot belong to another topic with small probability. Some questions without dominating topic are more likely to belong to more than one topic.



**Figure 2.24:** The distribution of the number of questions in relation to the probability of the dominant topic

#### 2.2.4.2 RQ2. What are the Most Popular Topics Among the Questions Related to Refactoring?

In order to assess popularity, we used four metrics. After collecting all the questions related to that topic, we computed:

- The average number of views by exploring the "ViewCount" attribute.
- The average number of comments using the CommentCount attribute.
- The average number of favorites using the FavoriteCount attribute.

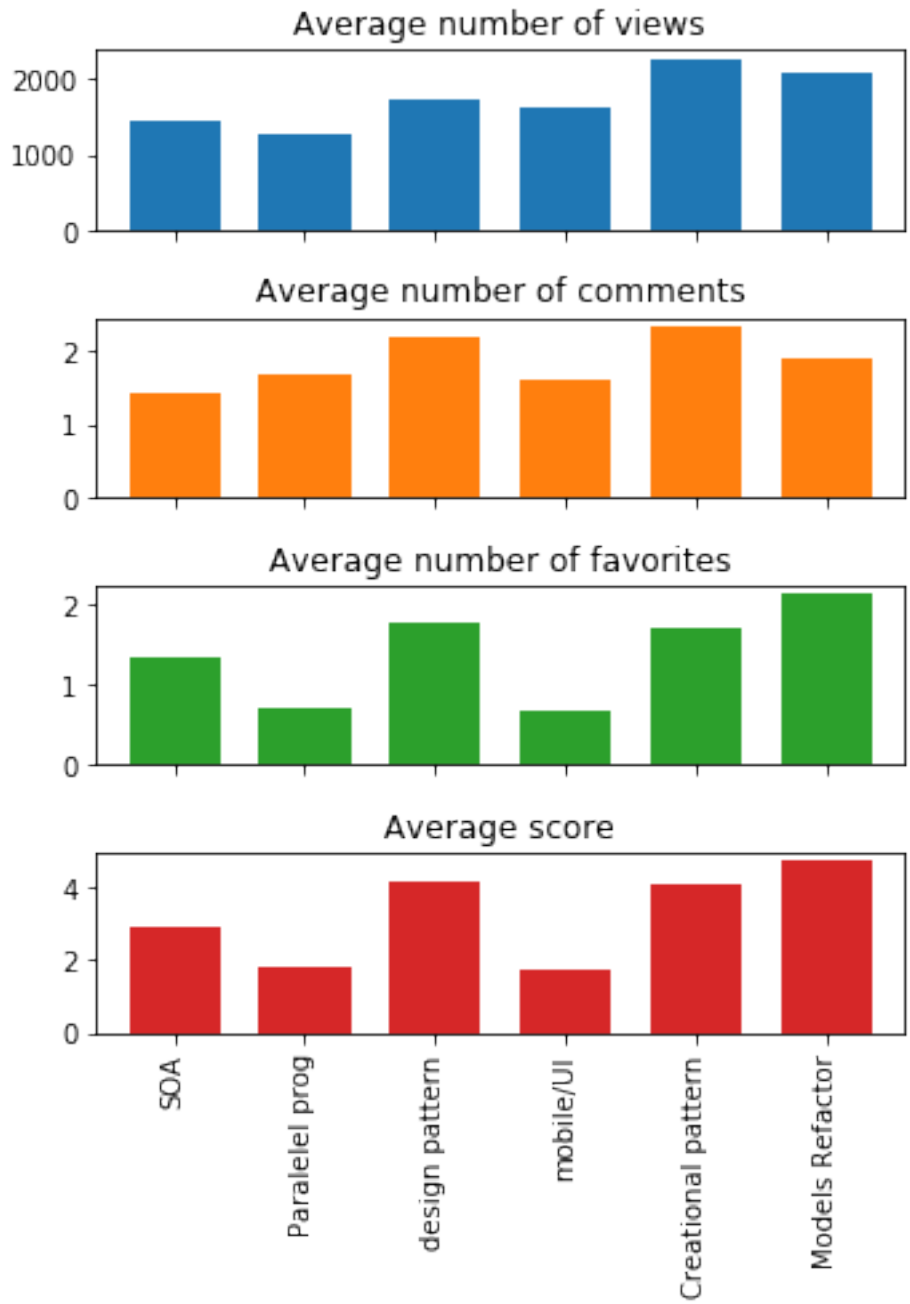
- The "Score" attribute which reflects the relevance of a question to Stack Overflow users, to compute the average score of this set of questions.

It is clear from Figure 2.25 that the creational pattern topic has the largest average number of views which exceed SOA refactoring despite a large number of questions around refactoring web services. This observation may lead to the conclusion that several of the SOA refactoring related questions did not have a considerable number of views meaning not all questions were relevant or important for refactoring of SOA architecture. However, we still observed more than 1500 views for several of these questions. We have also observed in Figure 2.25 that most the topics received the same average of number comments and favorites which confirms that all of them are important from practitioners' perspective.

Although the number of questions related to models refactoring is not high, but we clearly see that these few questions are very relevant to practitioners. For instance, the average number of views of a question related to models refactoring exceeds 2000 views which is high compared to the total number of views on the large number of SOA refactoring questions. However, this observation can be balanced based on the number of questions asked per topic since the average number of views may decrease when the number of questions per topic are high (e.g. higher probability for redundancy). Besides, it is clear that refactoring related to parallel programming, and mobile/user-interface topics have the lower popularity since they have the smallest average number of views and the smallest average score compared to the others topics.

#### **2.2.4.3 RQ3. Which Refactoring-related Topics are the Most Difficult to Answer?**

We included the answers that are related to the selected refactoring questions. These answers can be tagged as an accepted answer or not. Stack Overflow gives the user who asked a question the ability to accept only one of the answers. To estimate the difficulty, we counted the number of users that found an answer useful (based on the score attribute of



**Figure 2.25:** The four metrics used to estimate refactoring topics popularity.

the answer) similar to other studies on mining Stack Overflow [323, 335, 326].

We have defined three metrics to estimate difficulty. The first metric is the rate of questions that do not have a relevant answer. For the second metric, we computed the average number of views for unanswered questions in the topic. For the third metric, we calculated the average number of days that are needed to get a relevant answer.

All the results are presented in Figure 2.26. The number of unanswered question is highly correlated with the number of questions. Thus we presented the ratio of unanswered questions by the total number of questions to ensure a fair comparison between the different topics. First, we can see that most of the questions in a topic have a good percentage of relevant answers. The largest percentage of questions that do not have many relevant answers belong to the refactoring of parallel programming. It may be explained by the challenges associated with making programs running on multiple processors, which is not an easy task. This percentage does not exceed 31% of the questions. We can check from the results that design patterns have the smallest ratio of questions without a relevant answer with a ratio of around 18%. The integration of design patterns into existing architectures using refactorings is not an easy task and requires significant design changes. Thus, it could be challenging and time-consuming for practitioners to understand and answer these questions.

The same figure shows the average number of views for questions that did not get a relevant answer. This metrics can give us an insight about the difficulty as well. The questions that have no relevant answer got on average more than 200 views for all topics. Thus, it may mean that they are important questions. The ones related to mobile/UI have on average more than 300 views, but no user was able to give a relevant answer which others find it useful. We can conclude that even when a large number of developers accessed to these questions, they find it challenging to answer refactoring questions related to mobile and user interface or it may be an indication that most of the developers viewing these questions are not expert and community of Stack Overflow needs to pay more attention to this kind of topics.

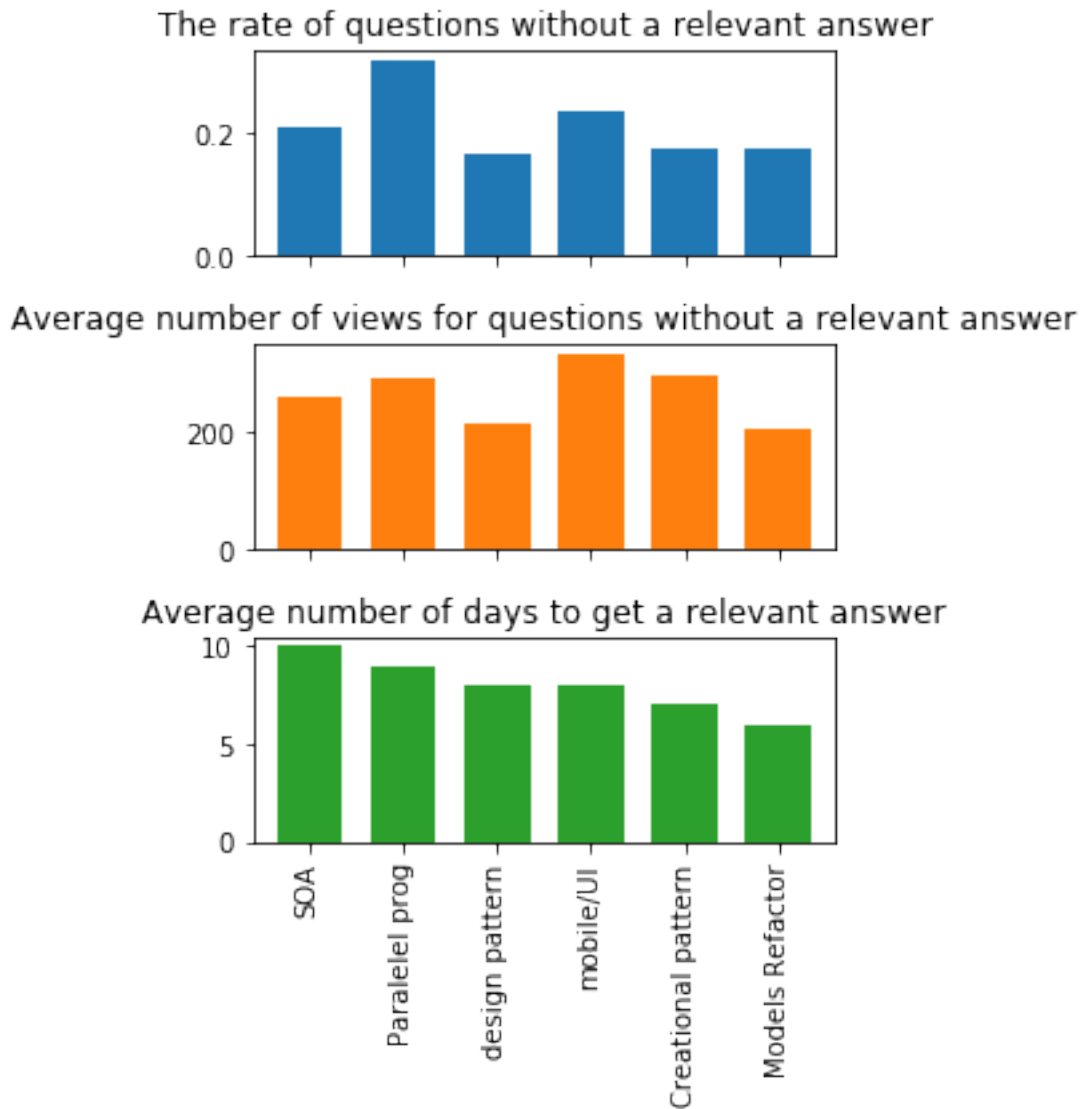
Another important aspect is the average number of days to get a relevant answer to a question. We found that models refactoring have the smallest duration compared to the other topics with only 6 days based on Figure 2.26. The other values are very similar as we can see that the topic that take the longer time to answer is refactoring of SOA with an average of 10 days to get a relevant answer to the question. In addition, it took between 6 to 10 days to get a relevant answer for the other topics. This means that in average developer does not need to wait very long before getting an answer to their questions. However, 10 days could be a long duration to get an accepted answer for refactoring related questions. Thus, many developers could have moved on and found another solution or abandon the refactoring step because of this long time to get a relevant answer.

Finally, we presented the ratio of the average number of answers to the average number of views. This metrics represents how many answers did the question get compared to the number of views. When this metric is high, it means that many developers can provide an answer to that specific question. It is clear that more than 10% of the people viewing design pattern topic answer that topic. The same observation is valid for SOA architecture. However, developers seem not very interested in answering questions around the model refactoring topic.

#### **2.2.4.4 RQ4. How the Interests of Developers on Refactoring Topics Change Over Time?**

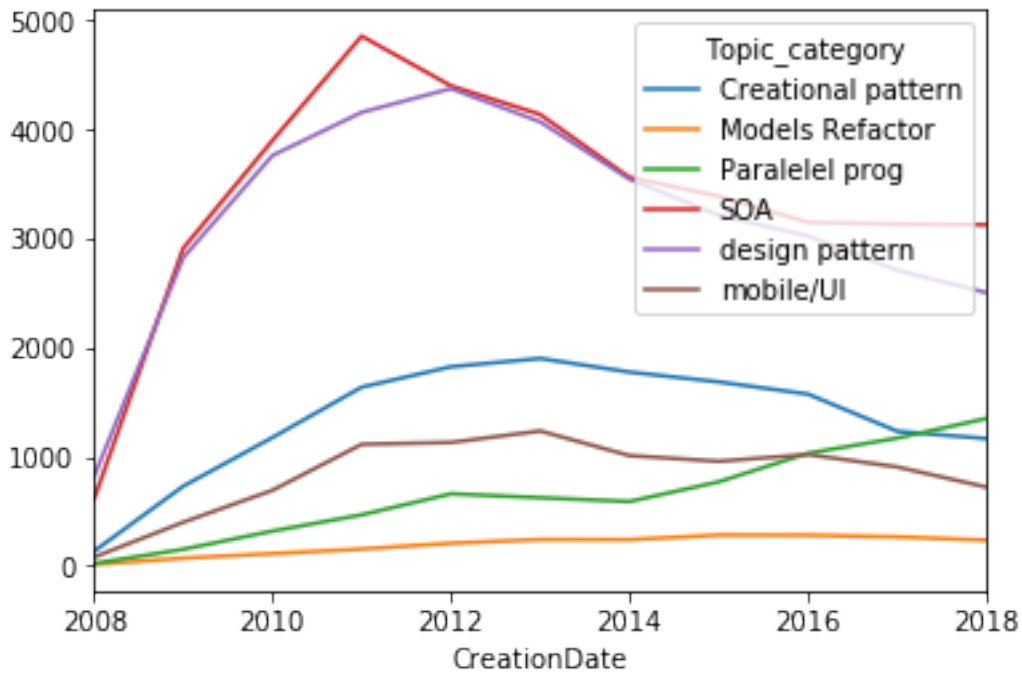
For this research question, we investigated the evolution of the number of asked questions throughout the years. We calculated the number of questions of each topic yearly and the evolution is presented in Figure 2.27. This evolution is related to different refactoring topics. It does not reflect the popularity as a question can be viewed much more times in the future compared to the year where it was asked.

It is clear that throughout the years from 2008 to 2011, refactoring of SOA have seen a significant evolution in the number of questions throughout the years but then there is a



**Figure 2.26:** The three used metrics to estimate the level of difficulty.

very important decrease in the number of asked questions. The same observation goes for the design patterns. One of the reasons that could lead to this evolution is probably because developers no longer need to ask questions since they are already found their questions answered on Stack Overflow. One important observation from this figure is the evolution of refactoring for parallel programming, and the number of questions is still increasing. Before 2016, the number of yearly asked question for the refactoring of parallel programming was less than both Mobile/UI and creational pattern. However, we can observe that in 2016 the number of questions asked about refactoring for parallel programming exceeded the refactoring of mobile app and user interface. In 2017, it exceeded the number of questions that are asked about creational design patterns. Thus, developers are showing high interest recently to refactoring for parallel programming.



**Figure 2.27:** The evolution of the number of questions by topic overtime.



## 2.2.5 Implications of this Study

We summarize, in this section, the main implications out of our study for researchers, educators and practitioners.

### 2.2.5.1 Implications for Researchers

Refactoring now expands beyond code-restructuring and targets different artefacts (architecture, model, requirements, etc.) [336, 337, 338, 339, 340, 341, 342, 343, 28, 344, 345, 346, 347], is pervasive in many domains beyond the object-oriented paradigm (cloud computing, mobile, web, etc.) [348, 293, 180, 11, 12, 349, 350, 351, 352, 353], is widely adopted in industrial settings [354, 355], and the objectives expand beyond improving design into other non-functional requirements (e.g., improve performance, security, etc) [356, 357, 358, 359, 360, 28, 25].

It is clear that the focus of the refactoring research community nowadays goes beyond code transformation to include, but not limited to, scheduling the opportune time to carry refactoring [19, 361, 362, 363], recommending specific refactoring activities [22, 24, 27, 28, 29, 30, 19, 364, 356, 365], inferring refactorings from the code [340, 181, 366], and testing the correctness of applied refactorings [367, 368, 369]. Therefore, the refactoring research efforts are fragmented over several research communities, various domains, and different objectives.

It is clear that there are many intersections between the researchers and practitioner's topics especially in emerging fields such as SOA, Mobile apps, model-driven engineering, and parallel programming. The main surprising outcome is that there are few discussions on Stack Overflow around refactoring for security purposes while it is a growing research topic in academia. Another interesting outcome is related to design patterns. While the academic community is mainly interested in using refactoring to fix anti-patterns, it is clear that practitioners are interested in integrating patterns using refactoring. The object-oriented paradigm seems to be still a dominant area for refactoring from both researchers and practitioners perspective especially with the increasing interests for model/design refactorings.

This study shows that practitioners are not mainly focusing on JAVA when asking questions on refactoring. However, the current research trends on refactoring are focusing mainly on JAVA. The practitioners are asking more questions on Python while there is a little of tools support and research to refactor Python code. Furthermore, most of existing research studies on refactoring are focusing on the automation of this process while the majority of questions on Stack Overflow are not about automated tools for refactoring but around bugs observed after manually applying refactorings. Thus, the research community may focus more on providing automated regression testing approaches to increase developers trust on applied refactorings.

#### **2.2.5.2 Implications for Educators**

Based on the large number of questions asked by practitioners on Stack overflow about refactoring, it is clear that educators need to increase students' awareness and expertise in the evolution of software systems. When students graduate and join the software industry they rarely build software systems from scratch but often spend more time studying and modifying existing systems. Traditionally, students have had a preconceived notion that evolution is a secondary concern. In order to better prepare them for the challenges they will face, we must invest in these curriculum innovations now.

We believe that this study will help educators shift through what's out there and determine the current issues in software quality. They will be able to understand the importance of refactoring and integrating it into education. This makes the students' education in software quality assurance (SQA) in general and in refactoring in particular more efficient and up to date.

While most of SQA courses focus mainly applying refactorings, this study show that practitioners are facing challenges beyond just the execution of refactorings to manage, detect, prioritize and test the refactorings. Thus, educator may think about training the current and next generation of practitioners on the whole refactoring life cycle. Another interesting

observation from our study is the large amount of questions about refactoring of Service Oriented Architectures. However, most of existing curricula focuses on JAVA refactoring and Object Oriented design restructuring in general. Thus, educators may consider introducing more background and material related to micro-services migration via refactoring.

### **2.2.5.3 Implications for Practitioners**

Due to the growing complexity of software systems, the last ten years have seen a dramatic increase and industry demand for tools and techniques on software refactoring which is confirmed in our study by the large number of refactoring questions asked by practitioners on Stack overflow.

Our study may help developers be more aware of the importance of writing clean code that follow well defined design patterns. This way, they will be able to prevent the issues that other developers are facing. In addition, they'll be able to know the hot topics in refactoring and therefore what to focus on their self-training efforts. We observed in our study that practitioners are mainly performing refactorings manually. Thus, it is important for them to try some recent semi-automated refactoring tools or prototypes offered for several programming languages rather than spending a lot of energy on the time-consuming and risky manual refactoring.

Another observation is the important focus of developers on introducing design patterns which is an area widely explored in refactoring research. Thus, practitioners may identify some interesting research prototypes that can automated the integration of design patterns. The lack of a refactoring community infrastructure prevents practitioners from using the state-of-the-art advances. They are only aware of refactoring tools that are standard in widely-used IDEs. There is a clear need for an effective communication platform between practitioners and refactoring researchers to identify relevant problems faced by the industry. Practitioners can upload a description of refactoring challenges and provide feedback on existing refactoring tools proposed by researchers.

### 2.2.6 Threats to Validity

Several threats can affect the validity of our results. The first threat is related to the selection of the tags related to refactoring. In fact, we may miss some important tags, but we believe that using the current list of tags we were able to generate an extensive list of questions from Stack Overflow.

The second threat is that not all questions have the appropriate tags since some people could have easily identified a wrong tag to a specific question. Thus, it is possible that we collected some irrelevant questions in our study.

In addition, it is possible that our results may not be generalizable. In this study, we focused on Stack Overflow, which is one of many Q&A websites, therefore, our results may not generalize to other Q&A websites. In our future work, we're planning to explore other development communities like GitHub.

In the experiments, we tried many configurations for the LDA model by tuning the probability state of the model and the number of topics. However, these parameters may impact the quality of our results. As for the data cleaning, we can probably introduce more stop\_words to reduce the noise in the vocabulary when identifying the refactoring topics.

We believe that the study of the popularity and difficulties of topics is very subjective giving that there is no way to get this measurement directly from the meta-data of the questions. Therefore, we tried to use a combination of metrics to answer these questions, and these metrics could be open to several possible interpretations.

### 2.2.7 Conclusion

We performed, in this contribution, the first large scale refactoring study on the most popular online Q&A forums for developers, Stack Overflow. We used 89 tags to extract 105463 questions about refactoring. We used the Latent Dirichlet Allocation (LDA) technique to generate the discussed topics in this repository. We found 6 main topics which are "Creational pattern", "Parallel programming", "Models refactor", "Mobile/UI", "SOA",

and "Design pattern". The analysis of these topics provided various key insights about the interests of developers related to refactoring such as the most addressed quality issues, the domains where refactoring is extensively discussed, the widely addressed anti-patterns, and patterns. We have also investigated how the interests of developers on refactoring topics change over the years.

## 2.3 Related Work

### 2.3.1 Systematic Literature Reviews about Refactoring

Mens et al. [247] provided an overview of existing research in the field of software refactoring. They compared and discussed different approaches based on different criteria such as refactoring activities, techniques and formalisms, types of software artifacts that are being refactored, and the effect of refactoring on the software process. Elish et al. [370] proposed a classification of refactoring methods based on their measurable effect on software quality attributes. The investigated software quality attributes are adaptability, completeness, maintainability, understandability, reusability, and testability. Du Bois et al. [371] provided an overview of the field of software restructuring and Refactoring. They summarized Refactoring's current applications and tool support and discussed the techniques used to implement refactorings, refactoring scalability, dependencies between refactorings, and application of refactorings at higher levels of abstraction. Mens et al. [372] identified emerging trends in refactoring research (e.g., refactoring activities, techniques, tools, processes, etc.), and enumerates a list of open questions, from a practical and theoretical point of views. Misbhauddin et al. [373] provide a systematic overview of existing research in the field of model Refactoring. Al Dallal et al. [10] presented a systematic literature review of existing studies, published through the end of 2013, identifying opportunities for code refactoring activities. In another of their work [35], they presented a systematic literature review that summarizes the impact of refactoring on several internal and external quality attributes. Singh et al. [36] published a systematic literature review of refactoring concerning code smells. However, the review of Refactoring is done in a general manner, and the identification of code smells and anti-patterns is performed in-depth. Abebe et al. [374] conducted a study to reveal the trends, opportunities, and challenges of software refactor researches using a systematic literature review. Baqais et al. [375] performed a systematic literature review of papers that suggest, propose, or implement an automated refactoring process.

The different studies mentioned above are mainly about identifying the studies related to very specific or specialized topics and sub-areas of refactoring. In this dissertation, we propose a large-scale refactoring systematic literature review by collecting, categorizing, and summarizing all the papers related to refactoring in general during the last 30 years.

### 2.3.2 Mining Stack Overflow Posts

Stack Overflow was created to help developers with computer programming, but it is becoming a useful knowledge repository for researchers. Therefore, several studies have used Stack Overflow to get an insight into the different questions discussed in practice [376]. Recent studies have focused on mining issues addressed by developers and clustering the related questions [323, 327, 324]. They all used the LDA topic modeling techniques. Yang et al. [323] clustered security related questions using a combination of LDA and genetic algorithms. They highlighted the most difficult and most popular security-related questions. Hassan et al. [327] adopted LDA to analyze the topics that developers talked about in software engineering, in general, and highlighted the main popular trends in the field such as mobile computing. Rosen et al. [324] addressed mobile specific questions and they also used LDA to understand the main challenges faced by mobile developers. Pinto et al. [377] performed an empirical investigation of the top-250 most popular questions about concurrent programming on Stack Overflow. They analyzed the text of both questions and answers to extract the dominant topics of discussion using a qualitative methodology. They observed that even though some questions are related to practical problems like fixing bugs etc., most of them are related to understanding basic concepts. Jin et al. [378] presented a study of how gamification affects online community members tendencies in terms of response time. They analyzed the distribution of gamification-influenced tendencies on Stack Overflow. They defined metrics related to response time to a question post. Results indicate that most members do not undertake in such rapid response activities.

In the software design and refactoring domain, Tian et al. [379] conducted a study on

developers' conception of Architecture Smells by collecting and analyzing related posts from Stack Overflow. They used 14 terms to extract 207 relevant posts. They used Grounded Theory method to analyze the extracted posts and find out developers' description of Architecture Smells and their causes. They also collected the approaches and tools for detecting and refactoring the different types of Architecture Smells, quality attributes affected by them, and difficulties in detecting and refactoring Architecture Smells. In another preliminary study, Choi et al. [335] used Stack Overflow to investigate practitioner's needs for clone detection and analysis and find out whether code clone techniques and tools have met the requirements of programmers. Tahir et al. [380] investigated how developers discuss code smells and anti-patterns in Stack Overflow in order to understand their perceptions of these design problems. They applied quantitative and qualitative techniques to analyse posts containing terms related to code smells and anti-patterns. They found out that developers use Stack Overflow to ask for general assessments of code smells or anti-patterns, rather than asking for refactoring solutions. They also noticed that developers usually ask people to check whether their code contain code smells/anti-patterns or not, and therefore, Stack Overflow is often used as crowd-based code smell/anti-pattern detector. Finally, Pinto et al. [381] conducted a qualitative and quantitative study to categorize questions from Stack Overflow about refactoring tools. They presented flaws and desirable features in refactoring tools. Even though all the studies mentioned above tried to mine posts from Stack Overflow to address different problems faced by developers, none of them has looked at the big picture of refactoring to identify the challenges related to refactoring in general faced by practitioners and what could be the current refactoring trends from the developers' perspective.

### **2.3.3 Detecting Refactoring Opportunities**

#### **2.3.3.1 Detecting Refactoring Opportunities in Mobile Apps**

In this category, we summarize the main related research in mining user reviews of mobile apps and linking them to the source code. A comprehensive literature review concerning



these topics can be found in the surveys carried out by Martin et al. [382] and Mao et al. [383].

**2.3.3.1.1 Mining User Reviews** A large number of studies analyzed the topics and content of app store reviews [384, 385, 386] and the possible correlation between price, reviews and ratings [386]. Mcilroy et al. [387] proposed an approach that can automatically assign multiple labels to user reviews based on different multi-labelling approaches such as Binary Relevance (BR), Classifier Chains (CC), and Pruned Sets with threshold extension (PSt).

Panichella et al. [388] manually analyzed users' review to determine a taxonomy of reviews categories (*i.e.*, bug fixing, feature adding, etc.). Then, they extracted a set of features from user reviews data using natural language processing, text analysis, and sentiment analysis. Finally, the app reviews are classified according to the taxonomy deduced in the first step using the standard probabilistic naive Bayes classifier, logistic regression, support vector machines, J48, and the alternating decision tree (ADTree).

Chen et al. [384] designed a framework for app review mining called *AR-Miner*. It can extract the most informative reviews and suggest weights on negative sentiment reviews. *AR-Miner* used topic modeling to group the informative reviews automatically based on their semantics similarity. Gao et al. [389] proposed *AR-Tracker*, a similar tool to *AR-Miner* [384], to automatically collect user reviews of apps and rank them in terms of frequency and importance. Another similar work, based on topic modeling, was proposed by Guzman et al. [385] to automatically identify application features mentioned in user reviews, as well as the sentiments and opinions associated to these features.

**2.3.3.1.2 Linking User Reviews to Source Code** Several techniques have been proposed for tracing documentation such as feature descriptions, emails and forums onto source code [390, 391, 392, 393]. The majority of these studies are based on lightweight textual analysis and information retrieval techniques. We will focus in the following mainly on linking

mobile user reviews to source code.

Palomba et al. [1] filtered and classified user feedback into the following categories: *information giving*, *information seeking*, *feature request*, and *problem discovery*. Then, they linked the user feedback clusters to source code classes by measuring the asymmetric Dice similarity coefficient. Ciurumelea et al. [394] extended this work by defining mobile specific categories (e.g. performance, resources, battery, memory, etc.). A tool, called the User Request Referencer (URR), is proposed to automatically classify reviews and recommend the source code files that should be modified for a particular review. The Vector Space Model (VSM) and information retrieval techniques are used to compute the textual similarity between user reviews and the source code. Another work of Palomba et al. [395] proposed the CRISTAL approach that helps developers in keeping track of the informative reviews while working on a new app release. CRISTAL links user reviews to the corresponding code changes (*i.e.*, code commits and bug reports) using text similarity.

Grano et al. [396] built a dataset of Android applications to provide an overview of the types of feedback that users may report. The extracted reviews are labeled based on topics-related keywords and n-grams used in the SURF summarizer tool [397]. In another related work, Noei et al. [398] studied the relationships between device attributes, such as the CPU and the display size, and the user perceived quality using linear mixed effect models. However, the authors did not consider understanding the impact of code quality and security metrics on user reviews or linking the discussed topics to source code.

### 2.3.3.2 Detecting Refactoring Opportunities in Web Services

**2.3.3.2.1 Quality of Service (QoS) Prediction for QoS-driven Web Services Recommendation** For this category of Web services recommendation, the goal is to predict the unknown QoS values between different service users and different web services, with partially available information, as the result, the optimal web service with the best QoS value can be recommended to the service user for composition [399, 400, 401, 402, 403]. The

common approach to recommend web service using QoS prediction is collaborative filtering which includes two main sets of algorithms: Model-based approaches and memory-based approaches [404, 297, 405, 295, 294, 406, 293, 292, 407].

In collaborative filtering, the goal is to calculate the similarities between service users to make prediction for the missing QoS data. Model-based approaches utilize machine learning, pattern recognition and data mining algorithms in order to predict the unknown QoS values. In memory-based collaborative filtering the similarity between users or services is calculated using a user-item rating matrix and then making prediction using a certain algorithm [408].

Shao et al. use [409] collaborative filtering to find the users similarity and predict the unknown QoS of the web services using the available invocation history for similar users. Zhang et al. [410] present another collaborative filtering method to rank the web services using QoS query information. The authors in [411] take an extra step and combine the user-based approach and item-based approach to propose a hybrid collaborative filtering, called WSRec, to predict the QoS in order to recommend a web service based on a computed rank for the QoS values. WSrec has been shown to achieve a good overall prediction accuracy, however it depends on historical QoS data and can suffer from the sparsity problem.

Some other studies [412, 413, 414, 415] predict the quality of web services to recommend web services with acceptable throughput or response time. Zhang et al. [416] propose a fuzzy clustering approach to predict the QoS of a web service in order to make web service recommendation satisfying the user requirements without sacrificing the quality. The work in [417] presents an example of model-based QoS prediction that uses a pattern recognition method. There are also studies focusing on the use of QoS for composing multiple services [418, 419, 420].

Zhu et al. [399] proposed an approach that takes a set of fixed landmarks as references. These references monitor QoS values of all the available web services. The approach clusters all the available services around the references. To predict the QoS value of the users in one cluster, the algorithm uses the QoS information of the similar landmarks in that cluster.

The main shortcoming of the collaborative filtering methods is that they heavily depend on the historical web service invocation information. Although, in practice, each user only invokes one or several web services. Therefore, the user-service invocation information is sparse when the number of services is large.

Most of the existing work focus on predicting web service performance based on other consumers' experiences to target the problem of web service recommendation. They use clustering-based approach to predict the quality of service. Their approach is based on the assumption that the consumers, who have similar historical experiences on some services, would have similar experiences on other services which is not always true. They ignore the large heterogeneity among users' views on the QoS. Furthermore, the clustering method presented in their work applies the hard technique that includes the use of a number of computers, known as landmarks, to perform the gathering of the real time QoS data, which is different from our mining technique presented in this proposal.

**2.3.3.2.2 Prediction of Web Services Evolution** Another category of related work in the area of web services prediction is to predict the evolution of web services. WSDLDiff [421] is a tool that uses structural and textual similarity metrics to detect the changes between different versions of a web services interface. VTracker, a tracking tool suggested in [422], detects changes in WSDL documents using XML differencing techniques. However, these tools are capable of detecting changes between Web Service releases, they do not provide any future changes prediction or recommendation on quality of service interface to the users. In order to address this challenge, [11] proposes a machine learning approach using an Artificial Neural Network to predict the evolution of web services interface from the history of previous release's metric. They utilized these predicted interface metrics to predict and estimate the risk and the quality of the studied web services.

In the area of code quality, there are some studies focusing on antipattern detection in Service-Oriented architecture (SOA) and web services. Rotem-Gal-Oz described the symp-

toms of a range of SOA antipatterns [423]. Kral et al. [424] listed seven “popular” SOA antipatterns that violate accepted SOA principles. A number of research works have addressed the detection of such antipatterns. Moha et al. [425] have proposed a rule-based approach called SODA for SCA systems (Service Component Architecture). Later, Palma et al. [426] extended this work for Web service antipatterns in SODA-W using declarative rule specification based a domain-specific language (DSL) to specify/identify the key symptoms that characterize an antipattern using a set of WSDL metrics. Rodriguez et al. [427] and Mateos et al. [428] provided a set of guidelines for service providers to avoid bad practices while writing WSDLs based on eight bad practices in the writing of WSDL for web services. Recently, Ouni et al. [12] proposed a search-based approach based on standard GP to find regularities, from examples of web service antipatterns, to be translated into detection rules.

Mateos et al. [429] as an attempt to provide the developers with some metrics as early indications of services interfaces with low quality, low maintainability or high complexity at development time, they have investigated the statistical correlation between complexity/quality and maintainability related WSDL-level service metrics and traditional code-level Object Oriented (OO) quality metrics and they confirmed a significant correlation. In their analysis, for the OO quality metrics they have included Modularity, Adaptability, Reusability, Testability, Portability, and Conformity attributes.

To automate the process of predicting the performance of the web services, Li et al. [430] proposed WebProphet. They extract the dependencies, compute the metrics and then predict the performance. They infer dependencies between web objects by perturbing the download times of individual objects. The shortcoming of this techniques is that, it is time consuming and imprecise.

Tariq et.al. in [431] introduce a tool called What-If Scenario Evaluator (WISE) to predict the response time based on packet traces from web transactions. However the downside of their proposed tool is that they’re not taking into account some of the client-side factors affecting the response time experienced by users.

In another study, in order to predict the response time, Chen et al.[432] introduced a new metric, called Link-Gradients to measure the affect of logical link latency on end-to-end response time for distributed applications. However to compute this metric, they assume all the individual changes are independent from each other in the system which can be a correct assumption in smaller application, but not necessarily applicable to more complex web services. They use this metric to predict the response time for untested configuration as well.

To summarize, none of the above studies analyzed the relationships between code/interface metrics/antipatterns and the QoS attributes which is one of the main contributions of this dissertation.

### **2.3.3.3 Software Quality Benchmarking**

Munaiah et al. [433] described the characteristics of software engineering projects using eight quality indicators. They proposed a framework, referred to as Reaper, that enables researchers to select GitHub repositories that fit these characteristics using supervised learning and a manually labeled dataset. Thakur et al. [434] implemented a platform, referred to as QScored, that hosts detailed code quality analysis information for a large number of repositories. QScored computes quality scores and assigns relative ranking of the repositories based on their architecture, design, and code smells. The platform also allows comparison between the quality of a user’s project and thousands of open-source projects. Pickerill et al. [435] developed a method, PHANTOM, to filter a large database of software projects in a resource-efficient way. This method extracts five measures from Git logs. Each measure is transformed into a time-series, which is represented as a feature vector for clustering using the K-means algorithm.

Lochmann et al. [436] proposed a benchmarking-inspired approach to determine threshold values for metrics. They also investigated the influence of the employed benchmarking base on the result of the software quality assessment. For that, they conducted a quality

assessment of a series of test systems for different benchmarking bases and compared the generated results. They found that 1) the bigger the benchmarking base, the less divergent are the rankings, and the less is the variance of the results and 2) the size of the systems contained within a benchmarking base does not influence the results. Chatzigeorgiou et al. [437] suggested a technique for benchmarking object-oriented designs by transferring a tool for performance measurement, called Data Envelopment Analysis, that is employed in economics. They investigated whether libraries exhibit a superior design quality compared to applications. They computed relative efficiency scores for several open-source libraries and applications. Benchmarking is performed by comparing each software design to its best-performing peers rather than to a theoretical baseline. They estimated the efficiency by considering design principles and the metrics that reflect conformance to these principles enabling the comparison of projects with diverse size characteristics. They found that libraries excel, at least within the context of the study, since their average efficiency score is higher than that of applications.

Correia et al. [438] proposed a technique for the systematic comparison of the technical quality of software products. They defined a model composed of three levels: source code metrics, system properties, and quality sub-characteristics. They collected measurement data from a wide range of systems into a benchmark repository. They suggested dividing the systems into groups based on their characteristics. They also suggested several types of comparisons such as comparison of individual systems to a group average and comparison of individual systems within a group.

Kalibera et al. [439] suggested a tool, referred to as BEEN, to automate the detection of performance changes during software evolution in a distributed heterogeneous environment. This tool gives developers timely feedback on their work. It involves compilation of software to be benchmarked, compilation of benchmarks, deployment, running the benchmarks and collecting, evaluating and visualizing the results. The authors base the evaluation of their still developing tool on its handling of a comparison analysis with the Xampler benchmark

from the CORBA benchmarking project.

Moses et al. [440] proposed a simple benchmarking procedure for companies wishing to develop measures for software quality attributes of software artefacts. They asked experts to rate the quality metrics of modules. Each proposed measure is expressed as a set of error rates for measurement on an ordinal scale and these error rates enable simple benchmarking statistics to be derived. These statistics can be used to benchmark subjective direct measurement of a quality attribute by a company's software developers.

Gruber et al. [441] presented a methodology to estimate the quality of source code without involving a quality expert. They first build the benchmark database, Then they calculate all the metrics of the benchmark suite. After that, the measured values of the assessed project are compared with the benchmark values. Finally, they aggregate the results to a quality score. They validated their work on both Java and C# projects. They found that Java projects provided promising result to use the benchmarking-oriented assessment more intensively. However, the experiment with C# showed that the results of the automatic benchmark assessment cannot be trusted blindly.

None of the attempts mentioned above has proposed a methodology to find the right benchmark that reflects individual characteristics of software systems. The majority of existing studies propose either standard general purpose benchmark suites or suggest procedures to compare a release with another.

## **2.3.4 Refactoring Recommendation**

### **2.3.4.1 Search-Based Refactoring**

Many studies have used search-based techniques to automate software refactoring by optimizing different sets of quality metrics [5, 442, 443, 6, 7, 8]. One interesting observation is that evolutionary algorithms are the dominant ones in search-based refactoring (e.g. NSGA-II, NSGA-III, etc.). Thus, we refer to evolutionary techniques when using the term search-based in this section. The reader can refer to the systematic literature review on search-based



refactoring [444].

O’Keeffe and Cinnéide [445] presented the idea of formulating the refactoring task as a search problem in the space of alternative designs, generated by applying a set of refactoring operations. The search is guided by a quality evaluation function based on eleven object-oriented design metrics that reflect refactoring goals.

Ouni et al.[446] presented a multi-objective refactoring formulation that generates solutions that maximize the number of detected defects after applying the proposed refactoring sequence and minimize the semantics similarity of the elements to be changed by the refactoring. They also tried to find recommendations that tend to maximize the use of refactoring rules applied in the past to similar contexts from one side, and to minimize semantic errors and the number of defects from another [447]. In another work [448], they focused on refactoring solutions minimizing the number of bad-smells while maximizing the use of development history and semantic coherence.

Alizadeh et al.[2] generated refactoring solutions that optimize the QMOOD metrics while minimizing the deviation from the initial design. In another work [321], they considered the QMOOD metrics as objectives for their optimization problem. Then, they used an unsupervised learning algorithm to cluster the different trade-off solutions in order to reduce the developers’ interaction effort when refactoring systems.

Harman and Tratt [5] were the first to introduce the concept of Pareto optimality to search-based refactoring. They used it to combine two metrics, namely CBO (Coupling Between Objects) and SDMPC (Standard Deviation of Methods Per Class), into a fitness function and showed its superior performance as compared to a mono-objective technique [5]. Ó Cinnéide et al. [443] proposed as well multi-objective search-based refactoring to conduct an empirical investigation to explore relationships between several structural metrics. They used different search techniques such as Pareto-optimal search and semi-random search guided by a set of cohesion metrics.

None of the work mentioned has directly addressed the problem of finding refactoring

solutions while considering the code quality and security as conflicting objectives. Also, all the above studies used the traditional random change operators (e.g. 1-point crossover, random mutation, etc.). These change operators can destroy relevant patterns inside good refactoring solutions when applied randomly on discrete problems. Furthermore, the existing search-based refactoring studies are generating the initial population randomly, which may have a negative impact on the execution time and the quality of the final solutions. With the large amount of data on GitHub projects about refactorings applied by developers and their impact, it may be possible to inject good patterns extracted from the history of refactorings when generating the initial population or designing knowledge-based change operators. This hypothesis is investigated and tested in this dissertation.

#### **2.3.4.2 Refactoring Dependencies**

Chavez et al. [449] investigated how refactoring types affect five quality attributes based on the version history of 23 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. [443] studied the impact of individual refactorings on quality attributes, such as using Move Method to reduce the coupling of a class. None of these studies considered the impact of a sequence of refactorings on quality attributes.

Bibiano et al. [450] 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 refactorings. They highlight 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. [451] analyzed dependencies at the model-level working with UML. They did not investigate dependencies at the code-level working directly with transformations on the

code rather than on UML models where the type of refactorings are different and simplified when compared to the code-level refactorings. Overall, existing studies mainly define what might be better considered similarity relations, such as a collection of refactorings that have similar effects (fixing a code smell) or similar context (applied by the same developer or to the same code location) [452, 453]. None of the existing studies rigorously define refactoring dependencies to integrate them into recommendation tools, including search-based refactoring.

### 2.3.4.3 Seeding and Genetic Operators in Search-Based Software Engineering

Search-based software engineering studies proposed few studies on improving the seeding mechanism and the change operators in order to optimize the performance and convergence of search algorithms as well as the quality of generated solutions.

Oliveira et al. [454] propose a reformulation of program repair operators such that they explicitly traverse three subspaces that underlie the search problem (i.e. Operator, Fault Space, and Fix Space). They implemented new crossover operators that respect the subspace division.

Zhu et al. [455] propose two mechanisms to avoid premature convergence of genetic algorithms: i) dynamic application of crossover and mutation operators; and ii) population partial re-initialization. They implemented two crossover and two mutation operators and, dynamically choose one crossover and one mutation operators to apply in each generation, based on a selection probability that is dependent on average progress. Abido et al. [456] propose improved crossover and mutation algorithms to directly devise feasible offspring chromosomes.

Fraser et al. [457] evaluated different strategies to seed the initial population in search-based techniques as well as techniques to seed values introduced during the search when generating tests for object-oriented code. They focused on three contexts: the first one is seeding of constants extracted from source code or bytecode throughout the search (e.g.,

initial population, mutation operators). The second one is related to strategies intended to improve the diversity of the initial population and its suitability for the optimization target. The last context targets the reuse of previously generated or hand-crafted solutions to seed the initial population of the search.

However, none of the studies mentioned above addressed the refactoring problem or designed new change operators and seeding mechanism to deal with the issues of solution correctness or the impact of randomness on solution quality.

#### **2.3.4.4 Security-Aware Refactoring**

**2.3.4.4.1 Code Fragments Accessibility** Grothoff et al. [458] present a tool called JAMIT to restrict access modifiers from security perspective. Specifically, the authors analyzed whether a class is confined to the package to which it is declared so the goal is to guarantee that a reference to a class cannot be obtained outside its package. The validation focused on reporting the percentage of classes that could be confinable.

Bouillon et al. [459] present a tool that checks for over-exposed methods in Java applications. Their tool determines the best access modifier by analyzing the references to each method. Muller [460] uses bytecode analysis to detect those access modifiers of methods and fields that should be more restrictive.

Steimann and Thies [461] highlight the difficulties of carrying out refactoring in the presence of non-public classes and methods. The authors formalize accessibility constraints in order to check the preconditions of a refactoring (e.g., moving a class to another package requires checking whether the accessibility of the class allows its users to still reference it). In particular, the authors analyze the cases in which a class or a method is moved between packages or classes with the goal of adapting their access modifiers to preserve the original behavior.

Zoller and Schmolitzky [462] present a tool called AccessAnalysis to detect over-exposed methods and classes by analyzing the references to code elements. Kobori et al. [463] investi-

gated the evolution of over-exposed methods and fields for a set of open-source applications. They reported that the change of access modifiers of methods is not frequent. They also found that the number of over-exposed methods and fields tends to increase in time.

Vidal et.al., presented two empirical studies on over-exposed methods [464] with the goal of analyzing their impact on information hiding and the interfaces of classes, and over exposed classes [465]. In both studies, they analyzed the history of the systems with the goal of understanding the variations in the over-exposed methods. They expanded and improved their work on method accessibility to class accessibility in [465] and presented an Eclipse plugin to make component public interfaces match with the developer's intent.

To summarize, the goal of this category of work is mainly to use static analysis to identify over-exposed code fragments whether related to security or not but without recommending refactorings.

**2.3.4.4.2 Software Security Metrics** In this category of studies, the main focus is to measure the security of software components [466, 467, 468, 469, 470, 471, 472, 473].

Chowdhury et. al.,[473] proposed an approach to measure the security of the code using a set of quality metrics. They proposed metrics that aim to assess how securely a system's source code is structured. The metrics are stall ratio, coupling corruption propagation, and critical element ratio. One shortcoming related to this work is that some of the metric values are decided based on intuition. For example, finding out the critical elements in a class depends on the intuition of the data collectors and it should be manually tagged.

Alshammari et.al, presented a set of metrics to measure the security of each class in an object oriented design projects [468]. To measure the security of each class, they utilized two properties of object oriented design: the accessibility of, and interactions within, classes. To measure the security of object oriented design, they defined the metrics based on quality metric, including composition, coupling, extensibility, inheritance, and design size. In order to identify if an attribute is critical (i.e. carrying critical information), they assumed that de-

velopers/designers have annotated class diagrams such as UMLsec and SPARK's annotations with a secrecy tag for each critical attribute in the design.

Agraval and Khan presented in [467] an investigation of how coupling induces vulnerability propagation in an object oriented design. They introduce a metric to measure Coupling Induced Vulnerability Propagation Factor (CIVPF) for an object oriented design. Their main idea behind this research is that Coupling is one of the means responsible for the vulnerability propagation. In order to compute CIVPF, they introduce some characteristics for an attribute to be vulnerable. Then, they defined the vulnerable method and class based on their access to vulnerable attributes.

The same authors later studied the role of cohesion for object oriented design security and proposed security metrics measuring the impact of cohesion on security vulnerability [466]. Highly cohesive classes are more understandable, modifiable and maintainable[466]. Their work is based on the assumption that in object oriented design, when a vulnerable attribute is spreading from one class to another it may compromise the whole system. They have proposed three metrics to measure the vulnerable association of a method in a vulnerable class, vulnerable association within a class and vulnerable association of an object oriented design. However they claim that computing these three metrics does not require any type of documents including Collaboration Diagrams, Sequence Diagram, State Diagram and Class Hierarchy, but it does require that an attribute should be labeled as vulnerable manually.

**2.3.4.4.3 Refactoring for Security** Maruyama et al. [474] presented a tool named Jsart (Java security-aware refactoring tool) that supports two types of refactorings related to software security, which is built as an Eclipse plug-in. It helps programmers to estimate the impact of the application of refactorings on security characteristics of the changed files by detecting the downgrading of the access level of a field variable within the modified code.

Alshammari et al.[475] studied the impact of refactoring rules on the security of an object-oriented design using the security design metrics [476, 477, 468, 466]. They also introduced

new security refactoring rules per analogy to existing ones and distinguished their effects on classified and non-classified features. They proposed one case study to illustrate how applying the refactoring rules improves the security of the design. Therefore, their findings are not general.

Ghaith and Cinnéide [478] presented an approach to automated improvement of software security based on search-based refactoring using the Code-Imp platform. When this platform is used to improve software design, the fitness function is a combination of quality metrics. In their work, they redefined this fitness function based uniquely on security metrics. Therefore, they neither studied the relationship between security and quality, nor the impact of the security-aware refactorings on the quality of the system. They also looked at the impact of certain refactorings on the security metrics, but since they considered just one study case, their results cannot be generalized.

Ghaith et al. [479] present a search-based approach to automate the refactoring process while improving software security. They used the search-based refactoring platform, Code-Imp, to refactor the code. The fitness function used to guide the search is based on a set of software security metrics they collected from existing work. However, the main objective of the refactoring process is to improve the security of the system and they did not focus on the quality of the code and design.

To the best of our knowledge, there is no previous research on the correlations between security metrics and quality attributes, or that provided a tool to recommend refactorings based on the preferences of developers from both quality and security perspectives, and the possible conflicts between them.

**Table 2.9:** Quality attributes and their equations.

Design Metric	Design Property	Description
Design Size in Classes ( <i>DSC</i> )	Design Size	Total number of classes in the design.
Number Of Hierarchies ( <i>NOH</i> )	Hierarchies	Total number of "root" classes in the design ( $count(MaxInheritanceTree(class)=0)$ )
Average Number of Ancestors ( <i>ANA</i> )	Abstraction	Average number of classes in the inheritance tree for each class.
Direct Access Metric ( <i>DAM</i> )	Encapsulation	Ratio of the number of private and protected attributes to the total number of attributes in a class.
Direct Class Coupling ( <i>DCC</i> )	Coupling	Number of other classes a class relates to, either through a shared attribute or a parameter in a method.
Cohesion Among Methods of class ( <i>CAMC</i> )	Cohesion	Measure of how related methods are in a class in terms of used parameters. It can also be computed by: $1 - LackOfCohesionOfMethods()$
Measure Of Aggregation ( <i>MOA</i> )	Composition	Count of number of attributes whose type is user defined class(es).
Measure of Functional Abstraction ( <i>MFA</i> )	Inheritance	Ratio of the number of inherited methods per the total number of methods within a class.
Number of Polymorphic Methods ( <i>NOP</i> )	Polymorphism	Any method that can be used by a class and its descendants. Counts of the number of methods in a class excluding private, static and final ones.
Class Interface Size ( <i>CIS</i> )	Messaging	Number of public methods in class.
Number of Methods ( <i>NOM</i> )	Complexity	Number of methods declared in a class.

## 2.4 Background

### 2.4.1 Object-Oriented Static Metrics for Software Quality and Security Assessment

#### 2.4.1.1 Software Quality Attributes

QMOOD is a widely used quality model, based on the ISO 9126 product quality model [480]. QMOOD defines six high-level design quality attributes (reusability, flexibility, understandability, functionality, extendibility, and effectiveness) (Table 2.9) that can be calculated using 11 lower-level design metrics defined in Table 2.10. We selected this model because it is a widely accepted quality model in industry and it has been validated based on hundreds of industrial projects[480, 481, 482, 28, 55].



Table 2.10: QMOOD metrics description.

Quality attributes	Definition
	Computation
Reusability	A design with low coupling and high cohesion is easily reused by other designs.
	$0.25 * Coupling + 0.25 * Cohesion + 0.5 * Messaging + 0.5 * DesignSize$
Flexibility	The degree of allowance of changes in the design.
	$0.25 * Encapsulation - 0.25 * Coupling + 0.5 * Composition + 0.5 * Polymorphism$
Understandability	The degree of understanding and the easiness of learning the design implementation details.
	$0.33 * Abstraction + 0.33 * Encapsulation - 0.33 * Coupling + 0.33 * Cohesion - 0.33 * Polymorphism - 0.33 * Complexity - 0.33 * DesignSize$
Functionality	Classes with given functions that are publicly stated in interfaces to be used by others.
	$0.12 * Cohesion + 0.22 * Polymorphism + 0.22 * Messaging + 0.22 * DesignSize + 0.22 * Hierarchies$
Extendibility	Measurement of a design's ability to incorporate new functional requirements.
	$0.5 * Abstraction - 0.5 * Coupling + 0.5 * Inheritance + 0.5 * Polymorphism$
Effectiveness	Design efficiency in fulfilling the required functionality.
	$0.2 * Abstraction + 0.2 * Encapsulation + 0.2 * Composition + 0.2 * Inheritance + 0.2 * Polymorphism$

### 2.4.1.2 Software Security Metrics

Code elements containing confidential or sensitive information such as userIDs, transactions, credit card, authentication, security constraints, may be security-critical. These code elements may be attributes, methods, classes, or packages. If these code fragments are over-exposed, this may result in vulnerabilities that can be exploited. Thus, developers should ensure that these code fragments are not over-exposed. Several software security metrics have been defined in the research literature at different levels of abstraction [293]. We focus in this study on those that are related to the code level and can be measured by static analyses.

For the selected security metrics, we have adopted the terminology and definitions proposed in existing studies [468, 466]. We consider that classified, confidential, and vulnerable attributes all refer to attributes that need to be secured. Tables 2.11 and 2.12 summarizes the definition of these 8 security metrics: Classified Instance Data Accessibility (CIDA), Classified Class Data Accessibility (CCDA), Classified Operation Accessibility (COA), Classified Mutator Attribute Interactions (CMAI), Classified Accessor Attribute Interactions (CAAI), Classified Attributes Interaction Weight (CAIW), Classified Methods Weight (CMW) and Vulnerable Association within a class (VAClass).

**Table 2.11:** Security metrics terminology.

<b>Term</b>	<b>Definition</b>
Classified Attribute	An attribute which is defined in UMLsec [483] as secrecy.
Instance Attribute	An attribute which value is stored by each instance of a class.
Class Attribute	An attribute which value is shared by all instances of that class.
Classified Methods	A method which interacts with at least one classified attribute.
Mutator	A method that sets the value of an attribute.
Accessor	A method that returns the value of an attribute.

**Table 2.12:** Security metrics definition

<b>Metric</b>	<b>Definition</b>
Classified Instance Data Accessibility (CIDA)	consider CA as a set of classified attributes in a class C, $CA = ca_i, i \in \{1, 2, \dots, n\}$ , and CIPA its classified public attributes as $CIPA = cipa_i, i \in \{1, 2, \dots, n\}$ $CIDA(C) =  CIPA / CA $
Classified Class Data Accessibility (CCDA)	consider CA as a set of classified attributes in a class C, $CA = ca_i, i \in \{1, 2, \dots, n\}$ , and CCPA its classified class public attributes as $CCPA = ccpa_i, i \in \{1, 2, \dots, n\}$ $CCDA(C) =  CCPA / CA $
Classified Operation Accessibility (COA)	consider CM as a set of classified methods in a class C, $CM = cm_i, i \in \{1, 2, \dots, n\}$ , and CPM classified public methods as $CPM = cpm_i, i \in \{1, 2, \dots, n\}$ $COA(C) =  CPM / CM $
Classified Mutator Attribute Interactions (CMAI)	consider a set of mutator methods in a class C as $MM = mm_i, i \in \{1, 2, \dots, mm\}$ , and CA the classified attributes $CA = ca_j, j \in \{1, 2, \dots, ca\}$ . Let $(CA_j)$ be the number of mutator methods which may access classified attribute $(CA_j)$ . Then, CMAI can be expressed as: $CMAI(C) = \sum_{j=1}^{ca} (CA_j) / ( MM  *  CA )$
Classified Accessor Attribute Interactions (CAAI)	consider a set of accessor methods in a class C as $AM = am_i, i \in \{1, 2, \dots, am\}$ , and CA the classified attributes $CA = ca_j, j \in \{1, 2, \dots, ca\}$ . Let $(CA_j)$ be the number of accessor methods which may access classified attribute $(CA_j)$ . Then, CAAI can be expressed as: $CAAI(C) = \sum_{j=1}^{ca} (CA_j) / ( AM  *  CA )$
Classified Attributes Interaction Weight (CAIW)	consider a set of classified attributes CA in a class C as $CA = ca_i, i \in \{1, 2, \dots, ca\}$ , and A the set of attributes $A = a_j, j \in \{1, 2, \dots, a\}$ . Let $(CA_j)$ be the number of methods which may access classified attribute $(CA_j)$ , and $(A_i)$ be the number of methods which may access the attribute $(A_i)$ , Then, CAIW can be expressed as: $CAIW(C) = \sum_{j=1}^{ca} (CA_j) / \sum_{i=1}^a (A_i)$
Classified Methods Weight (CMW)	consider CM as a set of classified methods in a class C, $CM = cm_i, i \in \{1, 2, \dots, m\}$ , and M the set of all methods as $M = m_j, j \in \{1, 2, \dots, n\}$ $COA(C) =  CM / M $
Vulnerable Association with in a class (VAClass)	consider CA as a set of classified attributes in a class C, $CA = ca_i, i \in \{1, 2, \dots, m\}$ , and M the set of all methods as $M = m_j, j \in \{1, 2, \dots, n\}$ , and $(M_j)$ the number of classified attributes associated with the method $m_j$ . Then VAClass is: $VAClass(C) = \sum_{j=1}^n (m_j) / ( CA  *  M )$

### 2.4.1.3 Code Smells

Code smells violate fundamental design principles and indicate software quality deterioration that makes software hard to maintain [484, 485]. Smells are indicators of deeper design issues in the software that negatively impact software design quality [485]. We employed DesigniteJava [486] to detect smells on the three granularities listed below. The tool has been validated [487] and used in empirical studies [488, 487].

**Architecture smells:** Cyclic Dependency, Unstable Dependency, Ambiguous Interface, God Component, Feature Concentration, Scattered Functionality, Dense Structure.

**Design smells:** Abstraction Design Smells (Duplicate Abstraction, Imperative Abstraction, Feature Envy, Multifaceted Abstraction, Unnecessary Abstraction, Unutilized Abstraction), Encapsulation Design Smells (Deficient Encapsulation, Unexploited Encapsulation), Modularization Design Smells (Broken Modularization, Cyclically-dependent Modularization, Hub-like Modularization, Insufficient Modularization), Hierarchy Design Smell (Broken Hierarchy, Cyclic Hierarchy, Deep Hierarchy, Missing Hierarchy, Multipath Hierarchy, Rebellious Hierarchy, Unfactored Hierarchy, Wide Hierarchy).

**Implementation smells:** Long Method, Complex Method, Long Parameter List, Long Identifier, Long Statement, Complex Conditional, Virtual Method Call from Constructor, Empty Catch Block, Magic Number, Duplicate Code, Missing Default.

## 2.4.2 Metrics for Web Services

### 2.4.2.1 Interface, Code and Service Metrics

We identified a set of metrics for Web Services that can be divided into three categories: interface, code and quality of service attributes. Interface level metrics are used to measure the complexity and the usage of service interfaces (e.g. WSDL files) such as the number of operations. Code level metrics are more related to measure the quality of the source code

of the services using mainly static analysis. It is possible for any web service to extract the pseudo code of the implementation of the operations in the interface which is enough to get code level static metrics such as coupling and cohesion.

As Web service technology suggests that the Web service is accessible only through its WSDL, we use the Java<sup>TM</sup> API for XML Web Services (JAX-WS)<sup>1</sup> to generate the Java artifacts of the Web service including: Depth of Inheritance Tree (DIT), Weighted Methods per Class (WMC), and Coupling Between Objects (CBO). Our approach is based on the *ckjm* tool (Chidamber & Kemerer Java Metrics)<sup>2</sup>. Note that for all code-level metrics were extracted using our parser implemented in our previous work [348].

Table 2.13 summarizes all the used metrics at different levels.

**Table 2.13:** Web service metrics [11]

Metric Name	Definition	Metric Level
NPT	Number of port types	Interface
NOPT	Average number of operations in port types	Interface
NBS	Number of services	Interface
NIPT	Number of identical port types	Interface
NIOP	Number of identical operations	Interface
ALPS	Average length of port-types signature	Interface
AMTO	Average meaningful terms in operation names	Interface
AMTM	Average meaningful terms in message names	Interface
AMTMP	Average meaningful terms in message parts	Interface
AMTP	Average meaningful terms in port-type names	Interface
NOD	Number of operations declared	Code
NAOD	Number of accessor operations declared	Code
ANIPO	Average number of input parameters in operations	Code
ANOPO	Average number of output parameters in operations	Code
NOM	Number of messages	Code
NBE	Number of elements of the schemas	Code
NCT	Number of complex types	Code
NST	Number of primitive types	Code
NBB	Number of bindings	Code
NPM	Number of parts per message	Code
COH	Cohesion: The degree of the functional relatedness of the operations of the service	Code
COU	Coupling: A measure of the extent to which inter-dependencies exist between the service modules	Code
ALOS	Average length of operations signature	Code
ALMS	Average length of message signature	Code
Response Time	Time taken to send a request and receive a response	QoS
Availability	How often is the service available for consumption	QoS
Throughput	Total Number of invocations for a given period of time	QoS
Successability	Number of response / number of request messages	QoS
Reliability	Ratio of the number of error messages to total messages	QoS
Compliance	The extent to which a WSDL document follows WSDL specification	QoS
Best Practices	The extent to which a web service follows WS-I Basic Profile	QoS
Latency	Time taken for the server to process a given request	QoS
Documentation	Measure of documentation (i.e. description tags) in WSDL	QoS

<sup>1</sup><http://docs.oracle.com/javase/6/docs/technotes/tools/share/wsimport.html>

<sup>2</sup>[http://gromit.iar.pwr.wroc.pl/p\\_inf/ckjm/](http://gromit.iar.pwr.wroc.pl/p_inf/ckjm/)

### 2.4.2.2 Service Antipatterns

Service antipatterns are examples of recurrent bad design solutions that designers and developers use when implementing a service [489]. They initially appear to be appropriate and effective solutions to a problem, but they end up having bad consequences that outweigh any benefits. Software engineers often introduce antipatterns unintentionally during the initial design or during software development due to bad design decisions, ignorance or time pressure [12]. Antipatterns make the maintenance and the evolution of services hard and time-consuming. Most of these antipatterns can be detected using the interface and code quality metrics that were defined in the previous sub-section [12]. We selected the following types of antipatterns extracted from previous work [12]:

- **Multi Service:** Also called God object web service, represents a service implementing a multitude of methods related to different business and technical abstractions. This service aggregates too many methods into a single service, and it is not easily reusable because of the low cohesion of its methods and is often unavailable to end-users because it is overloaded [490]
- **Nano Service:** Is a too fine-grained service whose overhead (communications, maintenance, and so on) outweighs its utility. This antipattern refers to a small web service with few operations implementing only a part of an abstraction. It often requires several coupled web services to complete an abstraction, resulting in higher development complexity, reduced usability [490]
- **Chatty Service:** Represents an antipattern where a high number of operations, typically attribute-level setters or getters, are required to complete one abstraction. This antipattern may have many fine-grained operations, which degrades the overall performance with higher response time [491, 492]
- **Data Service:** An antipattern that contains typically accessor operations, i.e., getters and setters. In a distributed environment, some web services may only perform some

simple information retrieval or data access operations. A Data web service usually deals with very small messages of primitive types and may have high data cohesion [426]

- **Ambiguous Service:** Is an antipattern where developers use ambiguous or meaningless names for denoting the main elements of interface elements (e.g., port-types, operations, and messages). Ambiguous names are not semantically and syntactically sound and affect the discoverability and the reusability of a web service [493]

These five antipatterns are the most frequently occurring ones in service based systems based on recent studies [426, 424, 494].

### 2.4.3 Multi-Objective Refactoring Using NSGA-II

#### 2.4.3.1 Algorithm Overview

Multi-objective optimization has been widely applied to refactoring problems to find trade-offs when searching for solutions. Non-Dominated Sorting Genetic Algorithm II (NSGA-II) [495] (1) is the dominant multi-objective optimization algorithm that has been used in search-based software engineering, including search-based refactoring [5, 6, 496, 321, 55, 497]. NSGA-II is designed to find a set of non-dominated solutions (a Pareto-front) in which each solution is a sequence of refactorings that provides a compromise among conflicting objectives (e.g., quality metrics).

---

**Algorithm 1** NSGA-II algorithm.

---

**Input:** System to evaluate and list of refactoring types

**Output:** Non-dominated refactoring solutions

Generate a random population  $P$  and evaluate the objectives **while** *the stopping condition is not reached*

**do**

    Select individuals  $M$  from  $P$  using Binary Tournament Selection Apply **crossover** operation on  $M$  to generate the offspring population  $O$  Apply **mutation** operation on  $O$  Update  $P$  by combining the parent and offspring populations

**return**  $P$

---

Initially, a starting population  $P$  is created using a random procedure. These solutions then undergo crossover and mutation, producing offspring  $O$ , and the process is repeated

until the stopping condition is reached (in our case, a maximum number of generations). The objective values of the solutions are computed and change operators are applied to create the next generation. In most of existing adaptations the algorithm finds non-dominated solutions balancing several conflicting objectives (i.e. software metrics). The different objectives can be normalized if they have different scales. Each objective can be written as follow:

$$Objective_i = \frac{M_i^{after} - M_i^{before}}{M_i^{before}} \quad (2.3)$$

where  $M_i^{before}$  and  $M_i^{after}$  are the values of the  $Metrics_i$  before and after applying a solution (or sequence of refactorings), respectively.

The search space explored by NSGA-II consists of different refactoring operations applied to different code locations where each operation is represented by a refactoring type (e.g., Move Method) and its parameters (e.g., source class, target class, attributes). In this dissertation, we selected 18 refactoring types discussed in the next section 2.14. A vector in which each element represents a refactoring operation is used to represent a solution. Each refactoring operation must satisfy a set of pre- and post-conditions defined by Opdyke [33] to maintain the behavior of the system.

### 2.4.3.2 Refactoring Operations

The refactoring operations considered in the approaches proposed in this thesis cover 18 operations selected from different categories: "Moving features", "Data organizers", "Method calls simplifiers", and "Generalization modifiers". These refactorings are listed in Table 2.14. We selected these refactoring operations because they have the most impact on code quality attributes. Recent empirical studies on refactoring show that these refactorings are widely used in open-source projects[498, 475, 73].



**Table 2.14:** Refactoring types considered in our study

<b>Refactoring Types</b>	<b>Definition</b>
Encapsulate Field	Changes the access modifier of public fields to private and generates its getter and setter.
Increase Field Security	Changes the access modifier of protected fields to private, and of public fields to protected.
Decrease Field Security	Changes the access modifier of protected fields to public, and of private fields to protected.
Pull Up Field	If two subclasses have the same field then this rule moves this field to their superclass.
Push Down Field	If a field is used by only some subclasses then this rule moves this field to those subclasses.
Move Field	Moves a field to another class.
Increase Method Security	Changes the access modifier of protected methods to private, and of public methods to protected.
Decrease Method Security	Changes the access modifier of protected methods to public, and of private methods to protected.
Pull Up Method	If two subclasses have the same method then this rule moves the method to their superclass.
Push Down Method	If a method is used by only some subclasses then this rule moves the method to those subclasses.
Move Method	Moves a method to another class.
Extract Class/Method	Creates a new class/method from an existing one.
Extract Superclass	If two subclasses have similar features, this rule creates a superclass and moves these features into it.
Extract Subclass	If two superclasses have similar features, this rule creates a subclass and moves these features into it.
Rename Method/Class/Field	Changes the name of a code element.

## CHAPTER III

### Improving the Process of Identifying Potential Refactoring Opportunities

Identifying refactoring opportunities in object-oriented code is an important stage that precedes the actual refactoring process. Manually inspecting and analyzing the source code of a system to identify refactoring opportunities is a time-consuming and costly process [499, 500, 501, 298, 502, 297, 296, 503]. Researchers in this area typically propose fully or semi-automated techniques to identify refactoring opportunities [22, 23, 24, 27, 28, 30, 7]. However, these techniques are usually common for object-oriented programs and revolve around the use of quality metrics such as coupling, cohesion, and the QMOOD quality attributes. Existing work fails to consider the context when finding refactoring opportunities which can make the identification process less efficient. For example, Web services have their unique design and components that are different from mobile apps and vice versa. Exploring the unique characteristics of each artifact is important when performing refactoring because systems are different from one another, so they should each be refactored differently, too. With the increasing use of Web services, mobile apps, and GitHub repositories, there is a need to come up with a process to identify the refactoring opportunities tailored to each project type.

To address this gap, we propose the following contributions:

## 3.1 Understanding the Impact of Code Quality and Security Metrics of Mobile Apps on User Reviews

### 3.1.1 Introduction

Current software development practices, such as DevOps [504], are moving away from traditional processes of releasing versions based on a fixed timeline and towards shorter development cycles. These cycles feature frequent releases to deliver features, fixes, and updates in close alignment with business objectives to better address customer needs [505]. In particular, continuous development and frequent release practices are widely used for mobile apps. Due to the large number of users and the competitive market, developers and managers strive to respond quickly to user needs, preferences, and complaints.

The timely detection of emerging quality and security issues, especially for mobile apps, is critical for software development teams to efficiently prioritize software maintenance activities and to satisfy their customers [388, 506, 1]. Most marketplaces such as *Google Play Store*, *Apple Store*, and *Windows Phone App Store* allow end-users to review apps based on scores of one to five stars, as well as free text that may highlight bugs, feature requests, security issues and quality challenges such as stability, energy usage, response time, etc. These reviews can reveal important concerns about quality and security [388, 507]. This feedback is relevant for developers to understand the impact of their changes and to schedule and prioritize their maintenance efforts.

Due to the large amount of review data, most of the existing studies have focused on the analysis and mining of user reviews to automatically classify them into topics (e.g. security, bugs, features, etc.) using keywords and topic modeling [384, 389, 385, 387, 388]. Recent related work used textual similarities, e.g., cosine similarity, between the reviews and the files of the system to identify candidate files to be inspected [1, 394], similar to existing bug localization techniques [508, 509, 510]. However, textual similarity has several limitations due to the amount of noise (e.g., misspelled words, non-English words, spam, etc.) in user

reviews and the difference between technical vocabulary used by programmers in the source code and the terms employed in end-user feedback.

Our industrial partner, Under Armour, developed a large mobile app, known as *MyFitnessPal*,<sup>1</sup> that tracks diet and exercise to determine optimal caloric intake and nutrients according to a user’s objectives. This app received 10,347 very low ratings in January and February 2019 alone which are 73% of all the reviews received by the app in the previous year (*i.e.*, 2018). Many users removed the app from their phones during that period. The major complaints were about severe performance issues: the app became very slow after adding a new feature to provide personalized recommendations and to support many new languages. This issue was fixed in June 2019. But this was not an isolated event; a similar scenario happened to the company in March 2018. A new release affected 150 million accounts due to security problems in *MyFitnessPal*.<sup>2</sup> These critical situations might have been mitigated more quickly if the issues raised in the reviews were identified earlier. However, the programmers had a hard time understanding the impact of code security metric changes as they created new versions, failing to determine if and when they should have fixed their app. For this reason, we claim that user reviews *complement* quality and security assessments based on source code metrics to identify critical quality issues efficiently and in a timely way.

While the detection of code quality and security issues is widely studied in the literature [55, 321, 511, 512], the majority of existing works identify issues by analyzing the source code using a set of static analysis metrics. One common response of developers when existing tools report quality problems (*i.e.*, the values of quality metrics violate the recommended threshold) is **”So what?”**. Clearly, developers need to see a connection between quality metrics and end-user perceived quality to motivate urgent action. In fact, there is a lack of understanding of the impact of code quality and security metric changes on customer satisfaction, which makes risk management, and the management of technical debt, challenging.

In this contribution, we first studied the correlation between user-perceived quality and

<sup>1</sup><https://www.myfitnesspal.com/>

<sup>2</sup><https://money.cnn.com/2018/03/29/technology/business/myfitnesspal-data-breach-stolen/index.html>

security of mobile apps based on reviews and code metrics over multiple releases to identify the relevant metrics that can be used for linking user reviews to source code. To filter quality and security topics, we reproduced a recent study [513] based on Adaptive Online Latent Dirichlet Allocation (AOLDA) [514], as this is a common method to cluster and track the variations of the topics of text streams. Furthermore, AOLDA showed high accuracy based on an existing large dataset of manually inspected reviews [513]. We analyzed the source code of each release to extract the QMOOD quality attributes [515], and a set of code security metrics [472, 466] based on static analysis. We selected these metrics since they can be calculated by static analysis, and they were widely used and validated in the research literature (including industrial projects) [55, 321, 516].

Hence, our first contribution is an empirical inquiry into the relationship between mobile app code quality and security metrics and user reviews. This results in our first research question (RQ1):

**RQ1:** *Is there a strong correlation between the user-perceived quality and security of apps and source code quality and security metrics?*

Based on the outcomes of this empirical study linking code quality and security metrics to user reviews, we designed and tuned a framework—*QS-URec*. This framework is used to address emerging quality and security issues by analyzing both user reviews and source code metrics. *QS-URec* recommends the files to be inspected and links them to specific user reviews. We identified the files responsible for such significant changes of code quality and security metrics, validated in RQ1, that may correspond to the highlighted issues in user reviews.

The second contribution is an approach to automatically link quality attribute change requests from user reviews to files. The empirical assessment of *QS-URec* leads to the formulation of the next two research questions.

**RQ2:** *How effective is our approach, QS-URec, in linking files to security and quality issues identified in user reviews?*

**RQ3:** *How does our approach, QS-URec, perform compared to state-of-the-art techniques in linking files to user reviews?*

By identifying files that are likely to be problematic, we can help developers prioritize their efforts, allowing them to find quality and security issues more efficiently. In addition, alerting developers about these potential problems might help them determine files that are good candidates for re-architecting. We also conducted an industrial validation of *QS-URec* to benefit from the evaluation and insights of the original developers of a popular mobile app.

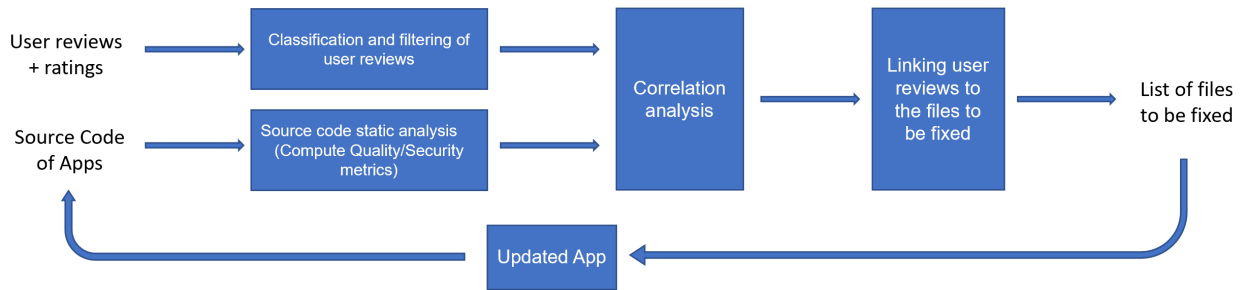
To address these research questions, we adapted an existing dataset of 50 popular mobile apps from *Google Play*<sup>3</sup> with 290,000 reviews [517, 1] to focus only on quality and security topics. In addition, we analyzed a popular mobile app, *MyFitnessPal*, provided by our industrial partner, Under Armour.<sup>4</sup> The app serves millions of users and has received over 400,000 reviews. Experimental results show strong correlations between several security and quality metrics and user ratings and reviews. *QS-URec* identified emerging quality and security app issues and prioritized the files to be inspected and fixed with higher precision and recall. Our framework outperforms a state-of-the-art approach relying on textual similarities [1]. A manual analysis of the results with the original app developers emphasizes the efficiency of *QS-URec* and the importance of considering user reviews to detect and fix security and quality issues.

To sum up, the primary contributions of this project are as follows:

1. An empirical study to understand the impact of source code quality and security metrics on user feedback and vice-versa;
2. A novel framework, coined *QS-URec*, that automatically links identified security and quality related user reviews to the files responsible for these issues;

<sup>3</sup><https://play.google.com/store>

<sup>4</sup><https://www.underarmour.com/en-us/>



**Figure 3.1:** Overview of the QS-URec Approach

3. An empirical validation of this framework on both open-source Android apps and a large real-world app as well as a comparison with an existing baseline.

### 3.1.2 *QS-URec*: The Proposed Approach

The goal of our approach, *QS-URec*, is to validate possible correlations between user-perceived quality and security of mobile apps, and the evolution of the code quality and security of these apps. We aim to identify relevant code quality and security aspects (e.g. quality/security metrics [480, 468, 466]) that can match the issues identified in the user reviews. Based on the outcomes of this empirical validation, *QS-URec* is able to link the quality and security issues of user reviews to the files to be fixed in the source code. To this end, *QS-URec* applies the following steps:

1. Classification and filtering of user reviews (related to security and quality topics) and static analysis of source code to extract quality and security metrics;
2. Correlation analysis to identify relevant code metrics that reflect user perceived quality and security issues;
3. Linking user reviews to the files to be fixed.

Figure 3.1 summarizes our approach. It takes a large set of user reviews of mobile apps as input, along with multiple releases of the corresponding code bases. There is a large body of work to classify user reviews [518, 388, 519]. We leveraged an existing study [513] that

classifies user reviews with high accuracy. We classified user reviews and kept only the ones related to quality and security issues. We mapped user reviews to the releases of the apps automatically based on the time dimension (e.g. releases and review dates). For each release, we analyzed the source code automatically to extract significant security or quality deviation using QMOOD quality attributes (refer to Table 2.10), and a set of code security metrics (refer to Table 2.12). Based on the collected data of user reviews and code metrics, we identified relevant metrics that can reflect the user-perceived quality and security issues. To do that, we performed a correlation analysis to prove that there is indeed a relationship between quality and security metrics and user rating of reviews related to security and quality. We used the correlation results to identify the files responsible for significant changes in code quality and security metrics that may correspond to the highlighted issues in the user reviews. Then, the developer can decide which files to inspect and fix to address the problems identified in the reviews and therefore improve the app rating for the next releases.

### **3.1.2.1 Preprocessing**

In this step, we aim to prepare the data required for the correlation analysis between user perceived quality and security, and issues identified in the source code. This phase is executed whenever new data on mobile apps are collected.

**3.1.2.1.1 Online User Reviews.** Typically, users provide app reviews via mobile phones that have small keyboards. Therefore, reviews may contain noisy data including misspelled and repetitive words, as well as uninformative details such as feelings of the users. We replicated the steps of a recent study [513] to identify topics from mobile user reviews. We filtered the identified topics and discarded reviews that were not related to quality and security.

**Extraction and formatting of User Reviews.** Users express their overall opinions about apps through the star-rating mechanism on a scale of one to five. A five-star rating means



that the user is very satisfied with the app and a one-star review represents dissatisfaction. It is common for mobile apps to receive thousands of reviews per week [394].

We first converted all the words in the reviews into lowercase and applied the preprocessing method described by Man et al. [520] for lemmatization. We adopted a well-known rule-based technique [521, 520] to fix repetitive, misspelled, and non-English words. Then, we applied filtering to reduce irrelevant words related to emotions and abbreviations using a list of 78 pre-defined keywords together with the stop words provided by NLTK [522] as detailed by Gao et al. [513].

**Topic Identification and User Review Mapping.** We used the Adaptive Online Latent Dirichlet Allocation (AOLDA) [514] method to cluster and track the variations of topics in text streams following the work of Gao et al. [513]; AOLDA identified the following topics: “Complexity”, “Design UX”, “Use cases”, “Bugs”, “Feature Requests”, “Frequency”, “Update”, “Camera & Photos”, “Video”, “Performance”, “Security Accounts”, “Streaming”, “Devices”, “Privacy”, “Connectivity”, “Notifications & Alerts”, “Audio”, “Gaming”, “Customer Support”, “Location Services”, “Sign Up & Login”, “Advertising”, “Payment”, “Pricing”, “Social & Collaboration”, “Battery”, “Internationalization”, “Operating System”, and “Import Export”. In this study, we focus on reviews related to quality and security issues because they are major challenges in mobile app development and can be quantified, in part, using static analysis of the source code. By software quality, we mean structural quality which refers to how the code meets non-functional requirements that support the delivery of the functional requirements. We selected the following tags for security: “Security & Accounts”, “Privacy”, “Sign Up & Login” and “Payment”, and the following tags for quality: “Complexity” and “Performance”. We chose to use the unsupervised technique AOLDA because of the large number of unlabeled reviews that we are including in our study. It is also easy to use and various options and parameters can be selected. Then, we manually checked a large sample of the reviews and kept only the ones related to security and quality topics as detailed in the validation section. Figure 3.2 shows examples of reviews related to

- This used to be a fantastic app. However, it has recently become **so slow**, it is virtually unusable. The app takes a **long time to load**, screen **transition time lags** horribly, puzzle completion is often not registered, and it is not possible to quit the app without force closing it.
- After several years of great service, the app is now **so laggy** as to be unusable (a **full minute** to open a puzzle or acknowledge completion). I see it hasn't been updated in over a year. Uninstalled it.
- Used to download 7 crosswords daily. now only downloads 4. They **don't respond to queries**. Other than loss of puzzles, the program is easy to use.

**Figure 3.2:** Example reviews related to quality issues

quality issues that were identified by our approach. The slowness and unreliability of the app were linked to the effectiveness and functionality quality metrics.

We do not dwell on the topic identification for three reasons: (1) clustering of topics is not a contribution of this project—it is an input for *QS-URec*; (2) we have manually checked the reviews related to security and quality for accuracy; and (3) we used publicly available datasets on clustered user reviews [1] in our validation, and AOLDA was primarily used to extend the available data with more projects including the industrial system.

We automatically mapped user reviews to the releases of the apps. If the app had only one release  $v$ , then all the reviews collected for that app were assigned to  $v$ . If the app had two or more releases, then for any two consecutive releases  $v_i$  and  $v_{i+1}$  that were between the dates  $d_i$  and  $d_{i+1}$ , we assign all the reviews that were submitted between  $d_i$  and  $d_{i+1}$  to  $v_i$ . We are aware that this automated mechanism might suffer from imprecision in cases where a user who has the version  $v_i$  installed on their phone comments the app only after the release of  $v_{i+1}$ , i.e., in these cases, the user review is wrongly associated to the version  $v_{i+1}$ . Nevertheless, previous work has shown that this happens rarely and, in any case, it is not possible to control for this type of imprecision [523, 395].

**3.1.2.1.2 Mobile Apps Source Code Analysis.** After collecting the releases of multiple mobile apps, they were parsed to extract quality and security metrics using static analysis.

**Quality metrics** We selected the Quality Model for Object Oriented Design (QMOOD) to evaluate code quality (see Table 2.10). The design metrics can be easily computed using

#	Package	Class Name	effectiveness	extendibility	flexibility	CAAI ↓	CAIW	CMW	VA
33	 opencsv au.com.bytecode	CSVParserTest	0.100	0.000	0.125	0.500	0.500	0.250	0.125
31	 opencsv au.com.bytecode	CSVWriterTest	0.000	0.000	0.000	0.200	0.200	0.059	0.012
38	 opencsv au.com.bytecode	CSVReaderTest	0.200	0.500	0.500	0.000	0.000	0.000	0.000
37	 opencsv au.com.bytecode	ResultSetHelp...	0.200	0.000	0.250	0.000	0.000	0.000	0.000

**Figure 3.3:** A screenshot of our tool that shows quality/security computation results

static analysis of the code. We implemented the QMOOD model, based on the Soot library [524], to extract metrics at the file level. Figure 3.3 shows a table from the report generated by our tool for a mobile application. It contains the QMOOD metrics calculations for each class of the system as well as the security metrics that we are going to see in the next subsection.

**Security Metric Extraction.** To quantify code security, we focused on the security metrics described in Table 2.12. To compute these security metrics, we needed to identify the security-sensitive elements. To do that, we took inspiration from existing studies [525, 468, 526, 527] that made use of text mining. We first gathered a set of keywords related to security [525, 526, 527] and indicators of confidential information from different sources such as code, release notes, security bugs, vulnerability reports, commit messages, and security questions/tags on Stack Overflow—Figure 3.4 reports these security keywords. Next, we computed a textual criticality score, based on cosine similarity, for each file to estimate the extent to which the file was related to security concerns. The higher the score was the more likely the file needed to be protected. For that, we preprocessed the source code using tokenization, lemmatization, stop word filtering, and punctuation removal [526, 527]. Then, we computed the cosine similarity between each file and the set of keywords. Finally, we manually validated the top 10 critical files and use their critical attributes (fields that have names that match one of the keywords from the list we gathered at the beginning) to identify the critical attributes in all the other files that will be used to compute the security metrics.

Keywords				
id	private	lock	code	protect
userid	privacy	algorithm	permission	securitymanagement
uuid	secure	salt	access	security constraint
password	credential	nonce	token	auth constraint
pwd	undercover	host	certificate	
username	crypted	port	cover	
account	hashed	backdoor	job	
creditcard	top secret	digital certificate	payment	
phonenumber	restricted	biometrics	transaction	
socialsecuritynumber	hidden	safe	ip-address	
dateofbirth	encrypt	confidentiality	transcoded	
secret	personal	sensitive	restricted access	
confidential	address	admin	sensitive information	
classified	cached	access	sensitive data	
login	security	administrator	card	
identifier	encoded	auth	credit	
unique	connectionString	authenticate	email	
name	path	credentials	content secure	
critical	signature	credit card number	user details	
vulnerable	role	encrypted	private field	
authenticator	hostname	hash	private member	
key	covered	undercovered	secret key	
			client id	
			hidden field	

**Figure 3.4:** Security keywords used in the security metrics calculations

### 3.1.2.2 Correlation Analysis

To identify the files that needed to be modified based on the user reviews, we first needed to confirm the correlation between the quality and security metrics, and the user ratings of reviews related to quality and security. After collecting data, filtering the reviews, and mapping them to their releases, we computed the average of quality and security metrics for each release as well as the average rating of reviews related to security and quality assigned to that release. Then, we computed the correlation between metrics and user ratings using the Spearman correlation coefficient ( $\rho$ ) [528]. We chose this coefficient because the data is not normally distributed. The Spearman coefficient may take values between +1 to -1. A value of +1 means that there is a perfect association of ranks, a value of zero means that there is no association between ranks and a value of -1 means that there is a perfect negative association of ranks [529].

### 3.1.2.3 Linking User Reviews to Source Code Quality and Security Issues

After confirming that there is a correlation between source code quality and security metrics and the increase/decrease of ratings, we are going to find the files that are the root causes of complaints reported in user feedback. The idea can be summarized as follows: if we notice that there is a considerable number of feedback with low rating that co-occurred or followed a drop in the security or quality metrics of some of the files, it is safe to conclude that the complaints in the user reviews are the results of the degradation of the security/quality metrics in those files. Therefore, the developers need to focus their attention on those files and try to refactor their code to improve their quality and security. To identify the files responsible for the highlighted quality and security issues in the user reviews at each release, we did not consider individual user reviews when linking to files. Instead, we identified if there is a large enough number of complaints to create a trend of issues in quality or security between each pair of releases. Then, we computed the change in quality/security metrics between releases. Next, we multiplied each metric by its corresponding correlation value to assign it a lighter or heavier importance in reflecting quality/security issues. After that, we summed up the new values of the security and quality metrics separately per file as defined in the following two formulas:

$$RQ = \sum_{n=1}^6 c_i Q_i \quad (3.1)$$

$$RS = \sum_{n=1}^8 c_i S_i \quad (3.2)$$

Where  $Q_i$  is a QMOOD metric defined in Table 2.10,  $S_i$  is a security metric defined in Table 2.12 and  $c_i$  is their corresponding correlation coefficient. We ranked the files by assuming that the ones with the largest decrease in security/quality measures are the ones responsible for the security/quality deterioration. Finally, we asked developers to manually check their relevance and correctness.

### 3.1.3 Experiments and Results

This section presents the methodology adopted to address our research questions as well as the results achieved on the considered set of mobile applications.

#### 3.1.3.1 Study Design

To address each of the three research questions described in the introduction section, we defined the following metrics and applied them on a dataset, described in the next section, containing a total of 50 mobile apps.

**3.1.3.1.1 Evaluation Metrics** For RQ1, we studied the correlation between the evolution of the ratings of the reviews related to quality and security for each release and the evolution of the code quality and security metrics for the same release. In this context, evolution refers to either an increase or decrease of the ratings and code metrics over time. The goal of this research question is to check if some code metrics we considered are correlated with the user-perceived quality and security. Then, we used the results of RQ1 to tune the weights of our approach, *QS-URec*, to establish links between files and security/quality issues identified in the trend of user reviews for each release.

For RQ2, we evaluated the performance of our approach, *QS-URec*, in linking files to security and quality issues identified in user reviews. Thus, we defined three measures: *precision*, *recall*, and *overlap*. *Precision* and *recall* for both security and quality recommended files are computed as:

$$Precision = \frac{|F_{QS-URec} \cap F_{QS}|}{|F_{QS-URec}|} \quad (3.3)$$

$$Recall = \frac{|F_{QS-URec} \cap F_{QS}|}{|F_{QS}|} \quad (3.4)$$

Where  $F_{QS-URec}$  is the set of ranked files related to security or quality issues recommended by our approach, and  $F_{QS}$  is the set of expected files that are responsible for the security

or quality issues as identified in an existing dataset [1]—reported in the next section. We have calculated the *precision* and *recall* measures separately for security and quality issues for each release when the average ratings of identified security/quality reviews decrease by at least 1. We ranked the files using our approach based on the formulas defined in Section 3.1.2.3 and calculated the *precision* and *recall* for the top 10 files. We selected 10 files since the dataset [1] we used identified a maximum of 10 issues/files per cluster. The *Overlap* measure computes a ratio between files correctly recommended by our approach based on user reviews and the total number of files recommended by our approach. To calculate this measure, we downloaded the repositories containing all the commits for all the apps used in our validation and determined the files that were actually changed from one release to the next one. We checked these files to understand if the changes performed by the developers were actually addressing quality or security issues. The *Overlap* measure is defined as follows:

$$\text{Overlap} = \frac{|F_{QS-URec} \cap F|}{|F_{QS-URec}|} \quad (3.5)$$

Where  $F_{QS-URec}$  is the set of files related to security or quality issues recommended by  $QS-URec$ , and  $F$  is the set of files modified by the developers as extracted from the commits. We have also answered RQ2 using an app provided by our industrial partner, Under Armour. We decided not to combine the results of the open-source apps with those of the industrial one since the original developers of the Under Armour app were involved in the validation, unlike the open-source apps.

To answer RQ3, we calculated the quality/security *precision* and *recall* of  $QS-URec$  as well as  $QS-URec$  without considering the correlation results (e.g., equal weights for all metrics). We compared the above *precision* and *recall* results with the work of Palomba et al. [1] that is based on textual analysis to link reviews clusters to source code, as discussed in the related work. We selected these two approaches as our baseline for the following reasons. The comparison with the equal-weights  $QS-URec$  approach can be used to estimate the benefits of our correlation analysis in selecting the relevant code quality and security metrics.

**Table 3.1:** Summary of the mobile apps considered in our study.

Category	Apps	Avg releases	Avg reviews
Books & Reference	3	9.33	8,199.33
Personalization	4	6	6,889
Tools	16	9.4	3,713
Music & Audio	3	24.66	2524
Photography	3	2	12,962.33
Maps & Navigation	2	3.5	1,269
Lifestyle	1	1	308
Education	1	23	15,693
Productivity	4	8.5	1,366
Video Players & Editors	2	9.5	14,603.5
Board	1	1	302
Communication	2	23	12,233.5
News & Magazines	1	16	701
Puzzle	2	1	3253.5
Travel & Local	2	2.5	10,070.5
Role Playing	1	1	22,337
Social	1	14	4,134
Arcade	1	1	511
<b>Total</b>	<b>50</b>	<b>453</b>	<b>290,330</b>

Furthermore, the comparison with the work of Palomba et al. [1] is useful to validate whether the use of code quality and security metrics can be more accurate than textual analysis in linking review issues to source code.

Since we are comparing multiple techniques to our approach on multiple releases of projects, we used a one-way ANOVA statistical test with a 95% confidence level ( $\alpha = 5\%$ ) to find out whether the sample results of different approaches are significantly different when compared to our approach based on each of the evaluation metrics (*precision*, *recall*, and *overlap*). Since the one-way ANOVA does not report the size of the difference, we used the Vargha-Delaney A measure [530], which is a non-parametric effect size measure. Given the different evaluation metrics, the A measure provides the probability that running our approach yields better performance than running the two other techniques. If the two algorithms are equivalent, then the measure provides  $A = 0.5$ .



**3.1.3.1.2 Dataset** We used and extended an existing dataset [1] with a total of 290,330 reviews and their ratings belonging to 50 android apps of different sizes and categories. Table 3.1 summarizes category-wise information about these apps that we used in our study. We added new apps to the dataset and classified them. All the reviews, their ratings, and the detailed list of the apps can be found in our online appendix [517]. All the apps that we use are written in Java and their source code is available on Github. We used the Adaptive Online Latent Dirichlet Allocation (AOLDA) [514] method to cluster the reviews and find the discussed topics. After that, we manually checked them, filtered the reviews and kept only the ones related to quality and security. We followed the procedure explained in the study by Palomba et al. [1] to extend the current dataset. Two of the authors, as inspectors, analyzed the change requests included in the clusters of reviews and detected the classes that needed to be fixed. Each inspector performed the task independently. Once the task was completed, the two different sets of links were compared and the inspectors discussed the differences they found in order to resolve the disagreement and reach a common decision. This procedure was performed before running any of the techniques used in these experiments. To avoid possible bias, our final dataset included mainly apps that are not inspected by the authors, along with few apps that we added. As an additional evaluation, we also assessed our approach in collaboration with the original developers of a widely used industrial mobile app without the intervention of the authors (as detailed later).

Last but not least, we also downloaded the source code of 453 releases of these 50 apps. We computed the quality and security metrics and we extracted the files that were changed in the commits of the considered releases. Then, we manually inspected the files related to the security and quality issues to double-check if the introduced code changes were actually fixing them. The security/quality issues were actually identified/located within a problematic file based on the tags and opened issues within the projects.

**Replication Package.** For the sake of verifiability and reproducibility, all the data, results, and tools used in these experiments are publicly available in the online appendix

[517].

### 3.1.3.2 Analysis of the Results

We report the results of the study by discussing each research question independently.

**3.1.3.2.1 Results for RQ1** Table 3.2 and Table 3.3 summarize the results of correlation analysis between the evolution of quality/security metrics and the ratings of user reviews related to quality and security issues. The symbol “++” in the table implies a strong correlation where the Spearman correlation coefficient has a value greater than 0.7. Similarly, the symbol “+” refers to a moderate strength of the correlation ( $0.5 \leq \rho < 0.7$ ). The symbol “\*” reflects that the correlation coefficient is less than 0.5 and thus a poor correlation. We combined all the data belonging to different apps when checking the correlation to ensure that we have enough data to validate our hypotheses.

The results, as shown in Table 3.2, revealed a strong positive correlation between user ratings and the metrics *Extendibility*, *Reusability*, and *Functionality*. The other metrics of *Effectiveness*, *Flexibility*, and *Understandability* did not show a strong correlation with the user ratings. The strong correlation with the *Functionality* metric implies that changes in the functionality of the app are consistent with user review ratings. The strongest positive correlation was observed for the *Extendibility* metric, with a correlation coefficient of 0.8. It is clear that several performance issues, such as response time, can be related to the large number of code clones due to poor modularization of the code. In fact, code clones are mainly observed within projects that have low abstraction/extendibility and the consequence is a high number of calls which can impact the performance/response time. Regarding the absence of correlation with some quality metrics, such as *Understandability*, this can be explained by the fact that these metrics are more related to the quality of the code from the perspective of developers rather than users, who may not notice the short-term impact of deterioration in these metrics when using the app.

**Table 3.2:** Correlation analysis results between quality attributes and review ratings

QMOOD metrics	Spearman correlation coefficient
Effectiveness	*(0.037)
Extendibility	++(0.862)
Flexibility	*(0.061)
Functionality	+(0.574)
Reusability	++(0.768)
Understandability	*(0.241)

**Q Key findings:** Our analysis reported that all the QMOOD quality attributes are positively correlated with the user-perceived quality. *Extendibility*, *Re-usability*, and *Functionality* are strongly correlated with user ratings, while there is no correlation between *Effectiveness*, *Flexibility*, and *Understandability* and the user ratings.

Table 3.3 shows strong positive correlations between the evolution of the ratings and the metrics *CCDA*, *COA*, *VAclass*, *CIDA*, and *CAIW*. The other metrics of *CAAI*, *CMAI*, and *CMW* show low or no correlation with the ratings. The highest positive correlations were observed for the *CCDA* and *COA* metrics (0.8 and 0.7, respectively). Both of these metrics are related to the level of access to classes in the code. When the level of access is high, an attack or code injection can be carried out relatively easily. The lowest correlation is observed on the *CAAI* metric ( $\rho = 0.1$ ) since this metric is mainly related to the access for internal attributes within a class. Thus, the impact on the app in terms of security is limited.

**Q Key findings:** All the security metrics used in our study are positively correlated with the user-perceived security. *CCDA*, *COA*, and *VAclass* metrics are strongly correlated with the user ratings. *CIDA* and *CAIW* are moderately correlated with the user rating. There is no correlation between *CAAI*, *CMAI*, and *CMW* and the user ratings.

**3.1.3.2.2 Results for RQ2** We evaluated the performance of *QS-URec* in linking files to security and quality issues identified in user reviews. *QS-URec* was able to achieve an average of 0.835, 0.860, 0.867, 0.864 in  $Precision_Q$ ,  $Recall_Q$ ,  $Precision_S$ , and  $Recall_S$ , respectively. Figure 3.5 shows a boxplot for each metric calculated over all 50 apps. We decided to

**Table 3.3:** Correlation analysis results between security metrics and user review ratings

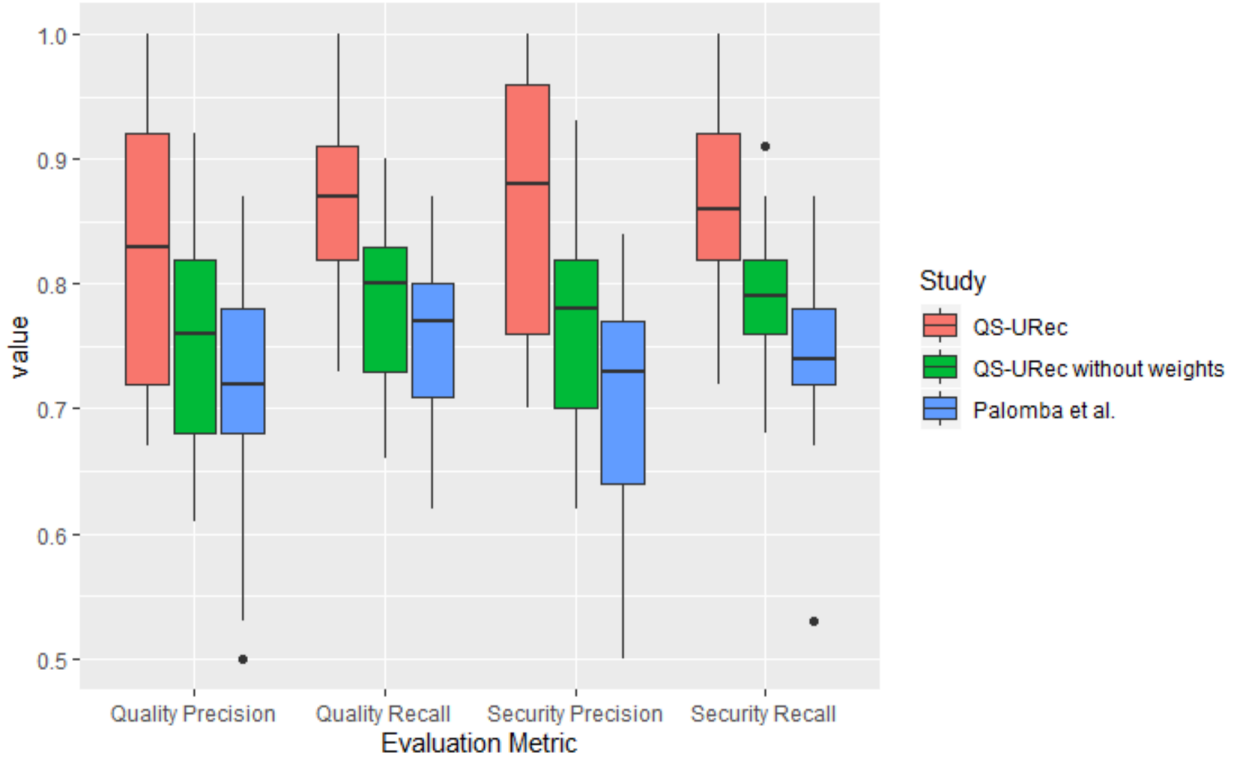
Security metrics	Spearman correlation coefficient
CCDA	++(0.826)
CIDA	+(0.593)
COA	++(0.782)
CAAI	*(0.136)
CMAI	*(0.223)
CAIW	+(0.617)
CMW	*(0.024)
VAclass	++(0.738)

calculate the *precision* and *recall* scores separately for the clusters of user reviews related to security and quality. For the cluster of reviews related to quality, the lowest *precision* was 0.67 for the *Recurrence* app, and the lowest *recall* was 0.73 for the *Materialistic* app. For the cluster of reviews related to security, the lowest *precision* was 0.7 for the *Recurrence* app and the lowest *recall* was 0.72 for *Easy-Token*. Thus, the *Recurrence* app had the lowest *precision* and *recall* among all 50 apps. This is likely due to the low number of reviews related to security and quality for this app; this app has only 292 reviews related to quality and security, while other apps, such as *Pixel Dungeon*, have more than 22,000. Furthermore, the *Recurrence* app has among the lowest number of releases among the 50 apps. Our approach was able to reach a perfect 1 in at least one of the evaluation metrics for 11 apps. However, we did not find that our approach achieved full correctness (*precision*) and coverage (*recall*) in one app. Surprisingly, we found that our approach achieved either full correctness or coverage for some apps with large numbers of security and quality reviews. Thus, clusters with a large number of reviews can help identify important issues so that they can be efficiently linked to the source code.

We carried out a manual analysis to investigate whether the developers actually fixed these classes. This investigation evaluated the benefits of our tool in terms of identifying relevant classes for developers to fix and to better manage their release plans. Figure 3.6 shows the distribution of the *SecurityOverlap* and *QualityOverlap* of all the apps. Table 3.4 shows the values of those metrics for each app. For the apps *Avare*, *Open soduku*, and *Shortyz*

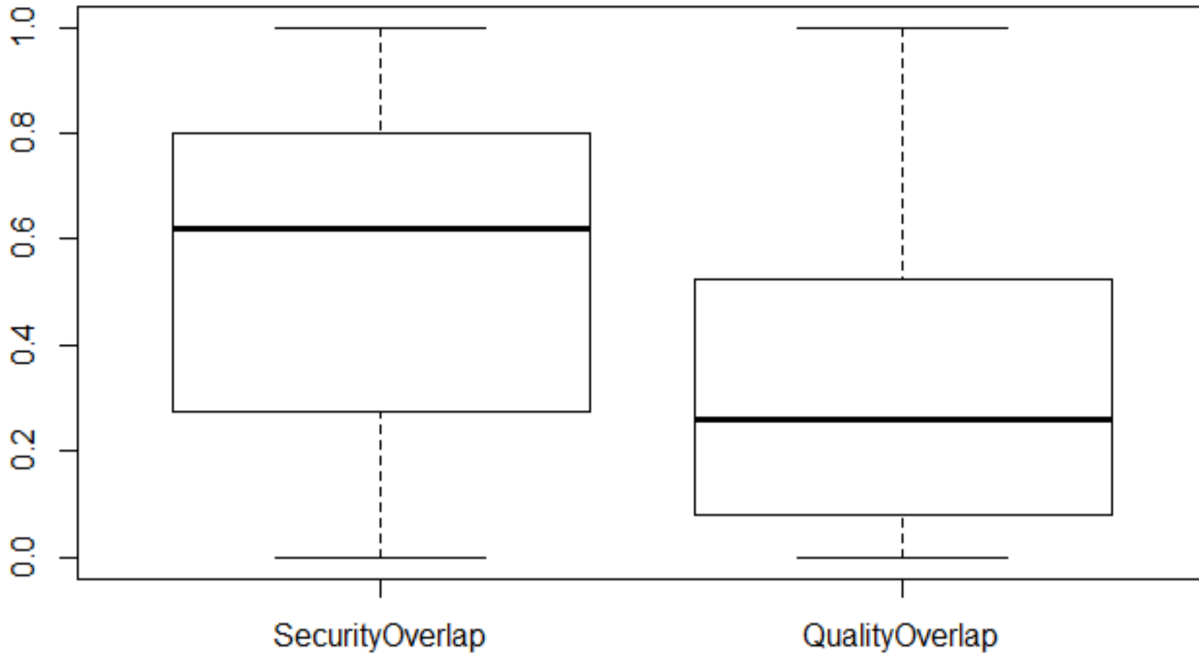
**Table 3.4:** Percentage of files that were identified correctly

<b>App Name</b>	<b>SecurityOverlap</b>	<b>QualityOverlap</b>
Aard dictionary	0.61	0.55
Abstract art	0.3	0.11
AcDisplay	0.87	0.2
AFwallFirewall	0.7	0.26
Amaze File Manager	0.8	0.12
android squeezer	0.84	0.18
AsciiCam	0.7	0.5
avare	1	1
Bart Runner	0.5	0.14
Battery Circle	0	0
Battery Idicator Pro	0.41	0
Bodhi Timer	0.4	0.1
Camera MX	0.21	0
CamTimer	0.5	0.3
Catlog	0.75	0.64
clip stack	0	0
Coin Flip	0	0
color namer	0	0
EasyToken	0.2	0.2
edx	0.85	0.27
FastHub	0.96	0
FB reader	0.2	0.06
FrostWire	0.96	0.41
Hex	0.7	0.14
k-9 mail	0.62	0.18
Materialistic	0.89	0.33
MicDroid	0.2	0
Micopi	0.5	0.5
missed notification reminder	0.18	0.05
MobileOrg	0.98	0.64
MTG familiar	0.77	0.25
Multipicture live wallpaper	0.25	0.07
osmAnd	0.2	0.2
open sudoku	1	0.8
path finder open reference	0.74	0.57
Persian Calendar	0.87	0.62
Pixel Dungeon	0.1	0.3
recurrence	0.44	0.44
share via http	0.67	0.67
shortyz crosswords	1	0.6
sls	0.71	0.39
sms backup plus	0.84	0.59
TaskBar	0.45	0.65
Terminal Emulator for android	0.68	0.32
Tinfoil for facebook	0.6	0.6
Trum Hunter	0.75	0.07
Vanilla Music	0.65	0.36
Vector Pinball	0.8	0.67
VX ConnectBot	0	0
wifi fixer	0.41	0.09
<b>Average</b>	<b>0.5552</b>	<b>0.3028</b>



**Figure 3.5:** Boxplot of the evaluation metrics for *QS-URec*, *QS-URec* without weights, and the work of Palomba et al. [1]

*crosswords*, all the files recommended by *QS-URec* to solve security issues were actually modified by the developers in subsequent releases. The apps that have a *QualityOverlap* and *SecurityOverlap* equal to zero are the apps that have only one release, which means that the repositories were inactive after their first release. There are a few apps, like *AsciiCam* and *Pixel Dungeon*, that also have only one release; their developers submitted many commits after the first release but they did not release a new version of the app. Figure 3.6 shows that the average for the *SecurityOverlap* is larger than *QualityOverlap* with 0.55 and 0.3 as values, respectively. Developers are more likely to modify the files related to security issues recommended by our approach compared to the ones related to quality. This might be explained by the fact that security problems are easier to detect from reviews than quality problems, which supports the usefulness and the need for our tool *QS-URec*. For instance, *Camera Mx* made frequent releases, but its developers fixed only 21% on average of the files related to security and none of the files that have quality issues.



**Figure 3.6:** Boxplot of the *SecurityOverlap* and *QualityOverlap* of all the apps

**Table 3.5:** Evaluation results using p-value and Vargha-Delaney A measure

Comparison	Security Precision		Security Recall		Quality Precision		Quality Recall	
	p-value	A measure	p-value	A measure	p-value	A measure	p-value	A measure
<i>QS-URec</i> vs <i>QS-URec</i> without weights	6.4E-19	0.78	7.9E-17	0.83	8.40637E-18	0.71	1.87E-19	0.77
<i>QS-URec</i> vs Palomba et al. [1]	1.39E-22	0.84	8.92E-19	0.92	1.91037E-15	0.79	3.99E-19	0.84

**Key findings:** *QS-URec* demonstrates high *precision* and *recall* in detecting the files responsible for quality and security problems discussed in user feedback. We have also found that developers missed files that are connected to user reviews related to quality or security issues.

**3.1.3.2.3 Results for RQ3** In this section, we compare the *precision* and *recall* values exhibited by *QS-URec*, *QS-URec* without considering the correlation results (equal weights), and the technique of Palomba et al. [1]. Figure 3.5 represents the boxplot of the distribution of those four metrics for the three approaches. By looking at the figure it is clear that *QS-URec* outperforms the other techniques in detecting the files that are responsible for quality

and security issues in the user reviews. *QS-URec* achieved an average of 0.83, 0.86, 0.86, and 0.86 in *Precision<sub>Q</sub>*, *Recall<sub>Q</sub>*, *Precision<sub>S</sub>*, and *Recall<sub>S</sub>* respectively. Without considering the weights while ranking the files, *QS-URec* reached an average of 0.74, 0.78, 0.75, and 0.78 for the same metrics. This difference in performance confirms the importance of our correlation analysis to tune our approach and eliminate irrelevant metrics. Furthermore, the textual similarity technique of Palomba et al. [1] was the lowest with an average of 0.71, 0.75, 0.7, and 0.74, respectively, in *Precision<sub>Q</sub>*, *Recall<sub>Q</sub>*, *Precision<sub>S</sub>*, and *Recall<sub>S</sub>*. The results confirm that the use of static analysis outperforms the textual analysis when linking user reviews to source code. In fact, the vocabulary used in the source code is significantly different from the natural language vocabulary used in user reviews. For instance, the *Easy Token* app did not have enough documentation (e.g. comments, release notes, etc.), with a low numbers of reviews (limited to 35 reviews); thus, the evaluation metrics of *precision* and *recall* were less than 0.7, with 0.5 for the security *precision*.

Table 3.5 summarizes the p-value and A measure results of comparing *QS-URec* with *QS-URec* without considering the correlation results and the technique of Palomba et al. [1]. We chose a threshold probability value of  $p \leq 0.05$  to indicate statistical significance. The results show that all the p-values computed when comparing *QS-URec* with the two other techniques are less than 0.05, and therefore we can conclude that there is a statistically significant difference between our tool and the other approaches. Similarly for the Vargha-Delaney A measure, we obtained values larger than 0.5 on all 50 apps when comparing *QS-URec* with the other two approaches, which means that our tool outperforms the baseline with a large effect value.



**Q** **Key findings:** According to our manual and statistical analysis, *QS-URec* outperformed the baseline techniques in identifying quality and security issues from user reviews. This confirms that static analysis can provide better results than textual analysis when linking user reviews of quality and security issues to the source code. We found that the performed empirical study in RQ1 helped to find the right weights for the code quality and security metrics.

### 3.1.3.3 Industrial Validation: *MyFitnessPal*

To better investigate the performance of our approach (RQ2), we conducted an industrial validation with Under Armour in collaboration with six original developers of the *MyFitnessPal* app that includes 1283 classes implemented over 10 years. These developers were selected using the following criteria: they (1) have significantly contributed to all of the last 15 releases of the app, (2) have over 10 years of experience in software development, and (3) are the most knowledgeable developers working on the app (as indicated by the app's product manager).

These developers executed our tool on the 6 latest releases of the *MyFitnessPal* app based on a total of 6,473 reviews and their ratings related to quality and security issues identified in these releases between July and December 2019. Then they analyzed the recommended files to fix the security and quality issues identified from the reviews for each release to calculate the *precision*. The developers did not agree to calculate a *recall*, since it is almost impossible to explore over 1,200 classes to look for quality and security issues at each release, especially with at least 128 classes changed per release. However, we checked the overlap between the classes correctly linked by our tool based on their feedback and the actual changes introduced to these classes. Furthermore, we conducted a short survey with these 6 developers of the app to check the relevance of the results. We asked them the following two questions:

- Q1 How useful did you find linking the user reviews of quality and security to the classes that need to be modified? Please rate your opinion from 1 (not useful at all) to 5 (very

useful).

Q2 Do you agree that you must fix the set of classes that need to be changed in order to satisfy the user requests in security and quality? Please rate your opinion from 1 (strongly disagree) to 5 (strongly agree).

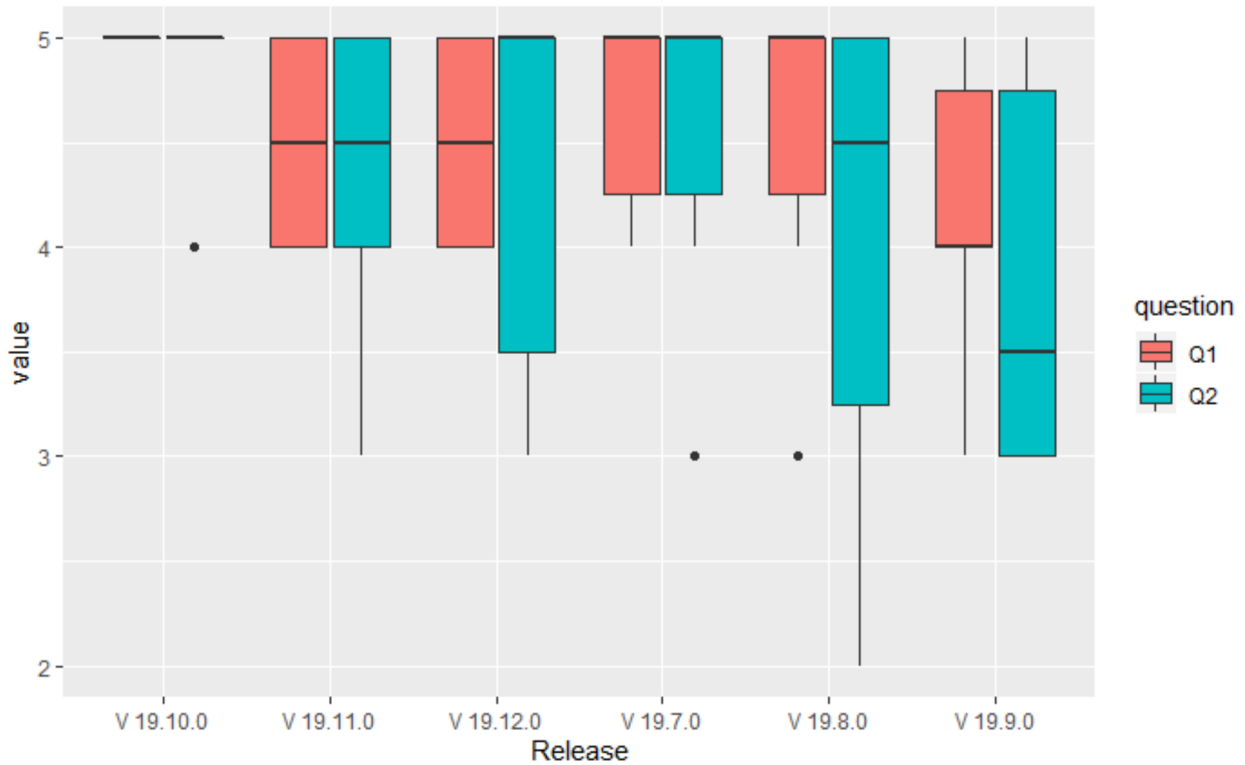
Table 3.6 summarizes the results of the evaluation metrics. The *precision* for both quality and security reached up to 1, which means that all the files recommended by our approach were correct for many releases of *MyFitnessPal*. The minimum achieved *precision* was 0.82, which is also considered to be high. For the overlap, *QS-URec* reached up to 0.62, which confirms that many classes still need to be fixed. Thus, the results confirm the need for our tool in practice, since it can be used to help developers better prioritize and manage their technical debt.

Figure 3.7 is a boxplot of the developers' answers for the two previous questions. The averages of their responses are 4.5 and 4.3 across all releases for the first and second questions, respectively. They found the files recommended by our tool relevant, and related to the problems discussed in the reviews. They also said that *QS-URec* will help them save a lot of time and improve the quality and security and therefore the ratings of their app. This confirms the usefulness of *QS-URec*. However, the developers suggested that we consider more features, such as fixing bugs, functional requirements, and user interface issues, rather than just focusing on quality and security issues. We plan to accommodate this request and extend our approach as part of our future research agenda.

### 3.1.4 Threats to Validity

A number of threats might have biased our results. This section discusses them as well as the mitigation strategies we put in place.

**Construct validity.** Regarding construct validity (the relationship between theory and observation), one threat can be related to the calculated precision and recall, since it can



**Figure 3.7:** Boxplot of the results of the survey conducted with our industrial partner Under Armour

**Table 3.6:** Precision and recall of running *QS-URec* on *MyFitnessPal*

Release version	Date	$Overlap_Q$	$Precision_Q$	$Overlap_S$	$Precision_S$
V 19.12.0	December 2019	0	0.92	0.24	0.86
V 19.11.0	November 2019	0.33	0.83	0	0.9
V 19.10.0	October 2019	0.23	1	0.44	0.82
V 19.9.0	September 2019	0.58	0.87	0.32	1
V 19.8.0	August 2019	0.29	1	0	0.94
V 19.7.0	July 2019	0.62	1	0.28	0.86

sometimes be subjective to decide if a file/class is linked to a set of user reviews. To mitigate this threat, we combined the use of (1) a publicly available dataset, (2) the original developers of a widely used mobile app, and (3) manual analysis of a few new apps by the authors of this

contribution. We found that the results were consistent in the data inspected from all three of these categories. For the manual inspection performed by the authors, we limited bias by preparing the oracle before running our tool on the apps. Another threat is related to the level of subjectivity when linking the identified issues in the user reviews to the recommended files. To counter this issue, we asked more than one evaluator to inspect the results. For instance, 6 developers were asked to evaluate the results for the Under Armour mobile app. They discussed their results whenever they had divergent opinions until they reached a final decision, but we rarely observed these situations in our experiments (e.g., Figure 3.5).

**Internal validity.** Threats to internal validity can be related to the relationship between the coverage of quality and security reviews and the decrease of the ratings. In fact, it is possible that several reviews can include a combination of functional and quality issues. For example, an important feature added to a new release may increase the ratings even when quality and security issues are observed. However, the aim of our first question is to provide a quantitative correlation analysis rather than showing a cause-effect relationship. Furthermore, in the industry validation we focused mainly on the reviews that are clearly related to security and quality issues. Moreover, mapping reviews to releases may pose another threat to the internal validity of our approach. Unfortunately, there is no way we can determine in an indisputable way the actual versions of the apps that each user had reviewed. In fact, a user may not keep his app up to date and submit a review based on an old version independently of the current version in Google Play.

**External validity.** External threats concern the generalization of our findings. We validated our approach on a dataset of reviews from 50 open-source applications and an industrial mobile app. It is uncertain whether our approach can have similar good results when applied to other kinds of *Android* apps (e.g., apps in the *Amazon App store*) and apps on other platforms (e.g., iOS). To improve the generalizability of our approach, we selected apps of different sizes and categories. In addition, we focused on issues relevant to mobile applications that are not specific to just one platform. Nevertheless, our dataset is relatively

small compared to the total number of apps available on *Google Play* and, therefore, further replications of our work would be desirable.

### 3.1.5 Conclusion

In this project, we proposed a novel framework, *QS-URec*, to detect files responsible for quality and security issues based on user reviews and source code metrics. We evaluated our approach on 50 popular mobile apps from *Google Play* with 290,000 reviews along with a large and popular mobile app provided by our industrial partner. Our results demonstrate strong correlations between several security and quality metrics and user ratings. *QS-URec* outperforms an existing textual analysis technique in terms of precision and recall when linking emerging quality and security app issues to relevant files to be fixed or refactored. We conducted experiments and a brief survey with the original developers of *MyFitnessPal* that supported the effectiveness of *QS-URec* and the importance of considering user reviews to prioritize and fix security and quality issues.

As part of our future work, we plan to extend our study to consider other types of quality and security metrics. Furthermore, we are planning to validate our work using a larger number of apps from different platforms. We will also include paid apps and compare the results with free apps. Last but not least, we plan to extend our approach to consider additional features like the treatment of functional requirements, user interface issues, and more, as suggested by the developers involved in our industrial assessment of *QS-URec*.

## 3.2 Early Prediction of Quality of Service Using Interface-level Metrics, Code-level Metrics, and Antipatterns

### 3.2.1 Introduction

Web services are nowadays increasingly used in most of industrial software systems [531, 532, 533]. Thus, it is critical to maintain high quality standards in terms of reliability, reusability, extendability etc. when designing and evolving services such as Google, Amazon, eBay, PayPal, FedEx, etc. The quality of service, related to the code and interface, is important for both the providers and subscribers/users. The providers may want to ensure a high quality of service before releasing them to the users. The users/subscribers prefer to use the service with the best quality of service and reasonable price among those offering the same features. Large-scale web services run on complex systems, spanning multiple data centers and distributed networks, with quality of service depending on diverse factors related to systems, networks, and servers [534]. This dynamic, distributed, and unpredictable nature of the web services infrastructure makes estimating and predicting the quality of service (QoS) metrics challenging, time-consuming, and expensive task.

Several studies have been conducted in the literature to predict the quality of web services based on a set of quality attributes (response time, availability, throughput, successability, reliability, compliance, best practices, latency, and documentation) [535, 536]. The majority of existing work help users selecting the best services based on their preferences and expectations [399, 418, 419, 420]. Clustering algorithms were adapted to classify existing services into multiple preferences then the user can select the cluster of services to investigate based on his preferred quality attributes. Thus, these studies are not actually dedicated to make prediction of services before deployment to potential users so they are not useful for services providers but mainly beneficial for subscribers. Some other studies are related to the prediction of the evolution of web services interface from the history of previous releases' metrics [11]. In another category of work, several approaches have been proposed to detect

quality issues such as antipatterns for web services [426, 537, 12]. Antipatterns are defined as commonly occurring design solutions to problems that lead to negative consequences [489]. Ouni et al. [12] defined a set of rules manually based on a combination of quality metrics to identify antipatterns. However, to the best of our knowledge, the problem of predicting the quality of service based on the interface and code quality attributes was not addressed before this contribution, which represents the main gap of existing literature.

In this project, we start from the hypothesis that source code and interface metrics and antipatterns are early indicators for the quality of service (QoS). We focused on the following types of antipatterns: Multi Service, Nano Service, Chatty Service, Data Service and Ambiguous Service. The source code/interface metrics and antipatterns can be used as an early detector of potential QoS issues before the service gets deployed on the cloud. For example, low cohesion of a web service may induce a high response time and a low availability due to the large number of calls between operations at multiple web services that will be generated whenever a request/query is submitted. Another motivation to validate our hypothesis is that service interface attributes, such as the number of port types or messages, can be measured relatively easily compared with measuring the QoS attributes that requires the deployment of the service.

Based on the above hypothesis, we empirically validated that source code and interface level metrics can be used to predict the quality of service (QoS) attributes. Thus, we proposed a novel approach for predicting quality of service by mining interface and code level metrics and antipatterns of 707 services extracted from an existing QoS benchmark [538].

In our approach, we adapted an Apriori clustering algorithm [539] to generate association rules that link source code and interface level metrics with the quality of service. We considered a two-step approach. The first step consists of extracting association rules between interface/code metrics and quality of service attributes. Then, the second step extracts rules that link antipatterns with interface/code/quality metrics. We divided our approach into two steps since the types of antipatterns are limited, not often easy to detect due to their

subjectivity, and could vary from one service to the other. We made the dataset that we created to validate all these new hypotheses available in the following link <sup>5</sup> so it can be used by other researchers and practitioners to answer the following research questions :

- **RQ1:** To what extent code/interface quality metrics can predict the QoS attributes?
- **RQ2:** To what extent code/interface can predict the QoS attributes of services with antipatterns?
- **RQ3:** To what extent the severity of different types of antipattern can be estimated based on their impact on the QoS?

Our contributions are not limited to only a prediction technique but also to validate a new scientific knowledge to the community about the connections between the code/interface/antipatterns and execution of services. The main contributions of this study can be summarized as follows:

1. We propose an approach to predict the quality of service based on antipatterns and code/interface level quality metrics. our approach is based on understanding the relationships between code/interface metrics and quality of services unlike most of the existing work for QoS prediction which are more based on the clustering of services based on the quality attributes.
2. Our results confirm that several of the antipatterns and code/interface quality metrics are correlated with quality of service attributes based on an extensive empirical validation over 707 web services.
3. We have also identified in our empirical validation the antipatterns that negatively affects QoS attributes the most.

<sup>5</sup><http://kessentini.net/tscdataset.zip>



### 3.2.2 Approach

In this section, we present an overview of our approach and then we provide details about the algorithm used and how we adapted it for the prediction of the quality of web services.

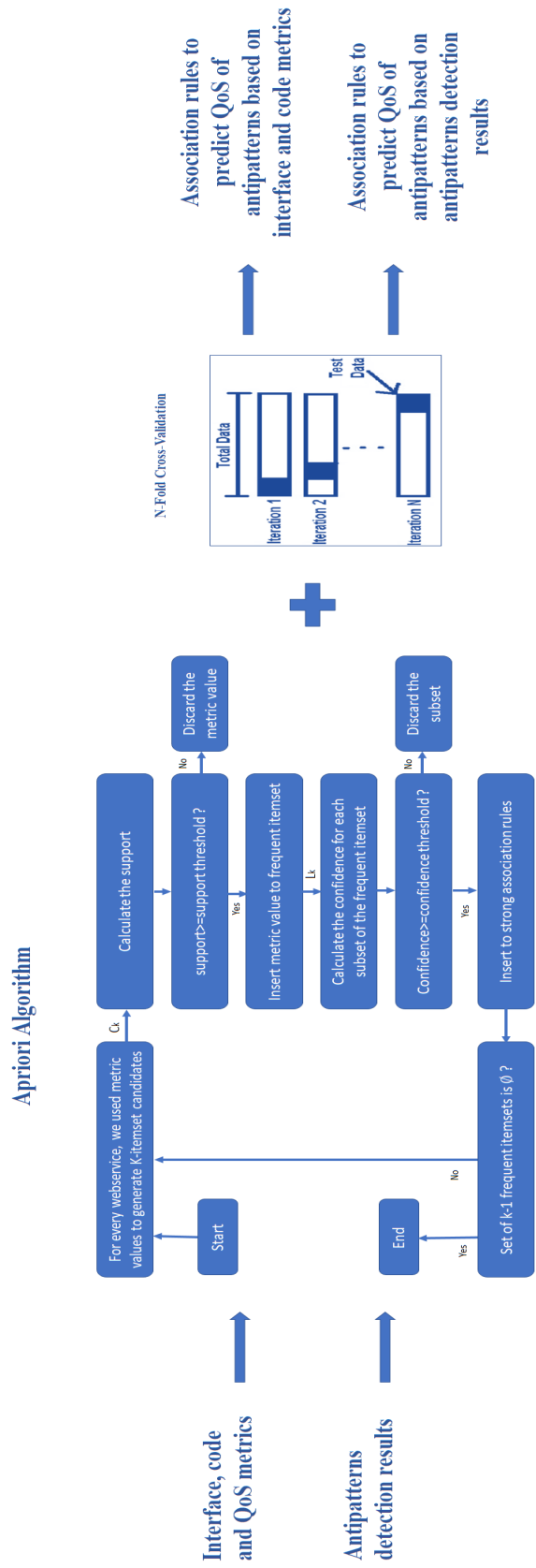
#### 3.2.2.1 Overview

As described in Figure 3.8, our approach has two main outcomes: 1) the association rules between the code/interface quality metrics and QoS attributes, and 2) the association rules between the Service antipatterns, and code/interface/QoS attributes. Thus, we generate two different predictive models. The outcome of the second predictive model is also important to understand the severity of antipatterns since no prior work investigated it.

To generate these outputs, our approach takes as inputs the set of code/interface/QoS metrics calculated on a large data-set of web services along with a list of antipatterns detected on the same data-set using our existing tool [12]. The detection rules used in that tool are described in Table 3.7. Then, the best association rules are found based on mining the inputs.

Association rules mining is one of the widely studied techniques of data mining [540, 541, 542, 543]. The generated rules represent possible correlations, causality, and redundant patterns between the different dimensions of the analyzed data (e.g. web services quality metrics and antipatterns). In our study, the generated rules take the following template  $M \Rightarrow Q$ , where  $M$  represents either a set of the interface quality metrics or antipatterns,  $Q$  is a set of the performance quality attributes (QoS). Therefore, we have  $M \cap Q = \emptyset$ .

To generate the association rules, the algorithm needs to find, first, the most common itemsets then these patterns will be formalized as a set of rules. The itemset represents the set of our input metrics related to the interface and source code. The frequent itemsets have a high support value which is the percentage of data points in the training data of web services that contain both  $M$  and  $Q$ . This support value of frequent itemsets should be above the threshold defined as minimum support. Once these frequent itemsets are identified, we



**Figure 3.8:** Approach Overview

adopted the k-fold cross validation method for the association rules generation.

We selected an algorithm called Apriori [539] based on the size and type of data manipulated in our study and the successful application of Apriori to address similar problems [544, 545, 546]. In the next subsections, we present an overview of this algorithm and describe its adaptation to our QoS prediction problem.

### 3.2.2.2 The Apriori Algorithm

The Apriori algorithm was proposed by Agrawal and Srikant in 1994 [539] and has been widely used for frequent itemset mining and association rules learning in databases. The name of the algorithm is Apriori, because it uses the prior knowledge of the frequent itemset properties. It computes the frequent itemsets in the training set through several iterations. Apriori uses the monotonicity property of the support measure to reduce the search space especially with the large data-set of quality metrics and services used in this project. This rule indicates that all subsets of a frequent itemset must be frequent. Consequently, if an itemset is infrequent then all of its supersets will be infrequent as well. This way, it can eliminate many of the itemsets that are not able to participate in a frequent itemset; and therefore, reduces considerably the running time of the algorithm. There are many versions of Apriori algorithm that improves the performance of association rule mining [547, 541].

The pseudo code for the algorithm is given below for the training set  $D$  that consists of the list of metrics, and a support threshold of  $\varepsilon$ . The Apriori algorithm iteratively find frequent item sets with cardinality from 1 to  $k$  ( $k$ -itemset). In each iteration,  $k$ -frequent item sets are used to find  $k+1$  item sets. For example, we first find the set of frequent 1-itemsets by scanning the dataset, accumulating the count for each item and keeping only those that satisfy minimum support. The results is denoted  $L_1$ . Next,  $L_1$  is used to find  $L_2$  the set of frequent 2-itemsets, which is used to find  $L_3$ , and so on, until no more frequent. Then, it uses the frequent item sets to generate association rules.  $C_k$  is the candidate set for level  $k$ .

We describe, in the next sub-section, our adaptation of the Apriori algorithm to our

---

**Algorithm 2** Pseudo code of the Apriori Algorithm

---

```
1: Input: A transaction database  $D$ , and a support threshold of  $\epsilon$  .
2: Output: Association rules of support  $\geq \epsilon$  .
3:  $L_1 = \{ \text{large1 - itemsets} \}$ 
4:  $k = 2$ 
5: while ( $L_{k-1} \neq \emptyset$ ) do
6:    $C_k = \{a \cup \{b\} | a \in L_{k-1} \wedge b \notin a\} - \{c | \{s | s \subseteq c \wedge |s| = k - 1\} \not\subseteq L_{k-1}\}$ 
7:   for transaction  $d \in D$  do
8:      $D_t = \{c | c \in C_k \wedge c \subseteq t\}$ 
9:     for candidates  $c \in D_t$  do
10:       $count[c] = count[c] + 1$ 
11:    end for  $L_k = \{c | c \in C_k \wedge count[c] \geq \epsilon\}$   $k = k + 1$ 
12:  end for
13: end while
14: return  $\bigcup_{i=1}^{\infty} L_i$ 
```

---

problem.

### 3.2.2.3 Adaptation of the Apriori Algorithm

Figure 3.8 presents an overview of our adaptation of the Apriori Algorithm. The algorithm is executed twice: a first execution to extract the rules between the code/interface metrics and QoS attributes and a second execution to generate the rules between the antipatterns and QoS attributes. We used Apriori because it is the first proposed algorithm to mine frequent patterns and has been widely used, studied, and is easily accepted. The rules generated by this algorithm are easy to understand and apply. We did not need to use an optimized version of the Apriori because our dataset is relatively small and does not require special computational power or memory. the goal of this contribution is to validate the correlations between interface/code metrics and QoS attributes then our plan later is compare which prediction algorithm could be better.

The first execution takes as input an exhaustive list of QoS attributes, presented in Table 2.13, of a large set of web service releases provided by eBay, Amazon, Yahoo!, etc. and their code/interface quality metrics as detailed later in the experiments section. The output of

this step is association rules that predict the performance of web services (QoS).

The second execution of the Apriori learning algorithm takes as input the same data of the first execution along with a list of antipatterns detected on a data-set of web services. The antipatterns are detected using our previous work [12] based on a set of rules presented in Table 3.7. We selected this detection tool because of the high accuracy and the low false positive as reported in [12]. The output of this step is a set of association rules that can predict the QoS attributes based on the detected antipatterns. This output can be used to understand the severity of different antipattern types on QoS attributes. The two steps of our approach are independent. The goal of the first step is to extract association rules between interface/code metrics and quality of service. The goal of the second step is to generate association rules between antipatterns and quality of service.

Both executions follow almost the same pattern. We took inspiration from an existing study [548]. In their work, the authors partition the database into two subsets. As a first step, they choose one of the subsets for training, and leave the other for testing. After that, they mine frequent itemsets from the training subset and use testing subset to compute itemsets' support in whole database. Then, They switch the subsets, so that the previous training set becomes the test set and vice versa. Again, they mine frequent itemsets from training subset and use the testing set to compute supports in whole database. We extended the theorem described in [548] to a more general case by using 5-fold cross-validation based on the number of dimensions (metrics and antipatterns) in the data considered in this project. Since we have a relatively small dataset size, we used cross-validation as it was proven to be a powerful preventative technique against overfitting [549, 550]. Our approach also guarantees the elimination of the itemsets that are not  $\mu$ -frequent relative to the whole data set. The training set  $D$  is randomly divided into 5 mutually exclusive subsets (the folds)  $D_1, D_2, \dots, D_5$  of approximately equal size where  $D_1 \cup D_2 \cup D_3 \cup D_4 \cup D_5 = D$ . Partitioning the original data in several different ways helps us avoid the possible bias introduced by relying on any one particular partition into test and train components.

**Table 3.7:** Antipattern Detection rules [12]

Antipattern	Detection rule
Multiservice(s)	$(NOD(s) \geq 17 \ \& \ COH(s) \leq 0.43 \ \& \ NOPT(s) \geq 7.8) \ OR \ (NOD(s) \geq 24 \ \& \ COH(s) \leq 0.39 \ \& \ NPT(s) \geq 2 \ \& \ NST(s) \geq 41 \ OR \ NCT(s) \geq 32)$
NanoService(s)	$(NCT(s) \leq 5 \ OR \ NST(s) \leq 8 \ \& \ NPT(s) \leq 2 \ \& \ NOD(s) \leq 5 \ \& \ COH(s) \geq 0.42) \ OR \ (NOPT(s) \leq 4.2 \ \& \ COUP(s) \geq 0.36 \ \& \ COH(s) \geq 0.39 \ \& \ NOD(s) \leq 6 \ OR \ NPT(s) \leq 2)$
DataService(s)	$((ANIPO(s) \geq 4 \ OR \ ANOPO(s) \geq 4) \ \& \ (NCT(s) \geq 31 \ OR \ NOM(s) \geq 79) \ \& \ COH(s) \geq 0.31 \ \& \ NAOD(s) \geq 13)$
ChattyService(s)	$(NPT(s) \leq 3 \ \& \ NOD(s) \geq 10 \ \& \ RAOD(s) \geq 0.38 \ \& \ (NCT(s) \geq 15 \ OR \ ANOPO(s) \geq 8.1) \ \& \ (NOM(s) \geq 38 \ OR \ NPM(s) \geq 2.2) \ \& \ COH(s) \leq 0.42)$
AmbiguousService(s)	$(ALOS(s) \leq 1.6 \ OR \ ALOS(s) \geq 4.9 \ \& \ AMTO(s) \leq 0.6 \ \& \ NIOP(s) \geq 4 \ OR \ AMTM(s) \leq 0.52)$

After the partitioning step, Apriori algorithm is used to find the set  $F_{D/D_1}^\mu$ ,  $F_{D/D_2}^\mu$ ,  $F_{D/D_3}^\mu$ ,  $F_{D/D_4}^\mu$  and  $F_{D/D_5}^\mu$ . It contains all  $\mu$ -frequent itemsets relative respectively to  $D/D_1, D/D_2, D/D_3, D/D_4$  and  $D/D_5$ . It is possible that some of them are not  $\mu$ -frequent relative to the whole transaction data set D. Itemsets that are  $\mu$ -frequent in a subset of the partitions, but not  $\mu$ -frequent in T are eliminated in the next step.

For every  $i \in \{1, 2, \dots, 5\}$ , We calculate the support of each itemset from  $F_{D/D_i}^\mu$  relative to D. Those itemsets that have  $suppcount_D \geq \mu$  are  $\mu$ -frequent relative to D. They are stored in  $F_{D/D_i, D_i}^\mu$ . The set  $F_{D/D_i, D_i}^\mu$ , contains all  $\mu$ -frequent itemsets relative to D that appear and are also  $\mu$ -frequent in  $D / D_i$ . We end up with  $F_{D/D_1, D_1}^\mu$ ,  $F_{D/D_2, D_2}^\mu$ ,  $F_{D/D_3, D_3}^\mu$ ,  $F_{D/D_4, D_4}^\mu$  and  $F_{D/D_5, D_5}^\mu$  that contain itemsets respectively from  $F_{D/D_1}^\mu$ ,  $F_{D/D_2}^\mu$ ,  $F_{D/D_3}^\mu$ ,  $F_{D/D_4}^\mu$  and  $F_{D/D_5}^\mu$  that are also  $\mu$ -frequent relative to D. Finally, we obtain the set  $F_D^\mu = F_{D/D_1, D_1}^\mu \cup F_{D/D_2, D_2}^\mu \cup F_{D/D_3, D_3}^\mu \cup F_{D/D_4, D_4}^\mu \cup F_{D/D_5, D_5}^\mu$  with  $\mu$ -frequent itemsets in D. Generally, sets  $F_{D/D_1, D_1}^\mu$ ,  $F_{D/D_2, D_2}^\mu$ ,  $F_{D/D_3, D_3}^\mu$ ,  $F_{D/D_4, D_4}^\mu$  and  $F_{D/D_5, D_5}^\mu$  are not disjoint.

At the end of our process, we obtain our association rules that predict the QoS from the metrics and the detected quality issues of the web services. The outcome of this research will help both service clients and providers know more about the quality of their web services with the least cost based only on interface and code metrics. To the best of our knowledge, this is the first study aiming to empirically validating the relationships between code/interface quality metrics (or antipatterns) and QoS attributes.

### 3.2.3 Experiment and Results

In this section, we first define our research questions. Next, we cover the data collection and experimental settings. Then, we summarize and discuss the obtained results.

#### 3.2.3.1 Research Questions

- **RQ1:** To what extent code/interface quality metrics can predict the QoS attributes?
- **RQ2:** To what extent code/interface can predict the QoS attributes of services with antipatterns?
- **RQ3:** To what extent the severity of different types of antipattern can be estimated based on their impact on the QoS?

#### 3.2.3.2 Data Collection and Evaluation Measures

To answer the different research questions, we built our prediction model for QoS using a large data-set of 707 releases of web services provided by eBay, Amazon, Yahoo!, etc. Besides code and interface metrics, the raw data contains antipatterns detection results of each web service extracted using our previous work as described in the previous section. An important step in generating the association rules is the pre-processing phase for the Apriori algorithm. We did the discretization of the data using a combination of strategies: equal interval width, equal frequency, k-means clustering and categories specifies interval boundaries. We also removed the outliers whenever necessary. To remove the outliers, we performed data visualization (box plot, scatter plot, etc.) and we removed points that are very separate/different from the crowd. Table 3.8 gives a summary of the considered training set in our experiments that includes a total of 707 services. We selected these 707 active services by contacting each of the web services from that existing benchmark [538] and we found that several of them are not active anymore. Since the existing benchmark is limited to QoS attributes [538], we extended it by calculating the interface/code metrics using a parser

that was implemented as part of our previous work [54]. [348]. The new dataset is available at the link <sup>6</sup>. The first file, Dataset1.csv, contains the dataset used to generate association rules linking the code/interface metrics and different quality of service (QoS) attributes. The second file, Dataset2.csv, contains the dataset used to generate association rules linking the code/interface metrics and different quality of service (QoS) attributes for each type of antipattern. It contains code/interface metrics, quality of service (QoS) attributes and the antipatterns detection results. we used the “apriori” function from the “arules” library of R for the apriori algorithm and the “discretize” function from the same library for the discretization. Thus, the experiments are conducted mainly using the R language.

**Table 3.8:** Web services used in our dataset

Category	#services	# antipatterns
Financial	107	126
Science	74	104
Search	63	98
Shipping	73	131
Travel	103	154
Weather	53	109
Media	106	214
Education	49	97
Messaging	38	54
Location	41	93
Total	707	1180

To answer **RQ1** and **RQ2**, we validate, first, the proposed approach using a 5-fold cross validation [548], to check if there is significant correlations between the metrics/antipatterns and QoS thus the ability to generate association rules. A small K value for the cross validation means less variance (more bias) while a large K value means more variance (lower bias). We tried different values of k in the cross-validation and we found that k=5 gives the best results in terms of number, meaning and consistency of the rules. The dataset was randomly partitioned into 5 equal size subsamples, again, to avoid any bias. We did take into account

<sup>6</sup><http://kessentini.net/tscdataset.zip>



the metrics values in the pre-processing phase for the Apriori algorithm by performing the discretization of data. To this end, we used the following evaluation metrics:

**Support:** Support is the statistical significance of an association rule interpreting as the ratio (in percentage) of the web services that contain  $M1 \cup M2$  (metrics/antipattern types with their thresholds) to the total number of web services in the data-set. Therefore, if that the support of a rule is 5% then it means that 5% of the total web services contain  $M1 \cup M2$ . In other words,

$$support(M1 \Rightarrow M2) = P(M1 \cup M2) \quad (3.6)$$

, where  $P(M1)$  is the probability of cases containing  $M1$ .

**Confidence:** For a specific number of web services in the data-set, confidence is defined as the ratio of the number of web services that contain  $M1 \cup M2$  to the number of web services that contain  $M1$ . Thus, if we say that a rule has a confidence of 85%, it means that 85% of the covered web services containing  $M1$  also contain  $M2$ . In other words,

$$\begin{aligned} confidence(M1 \Rightarrow M2) &= P(M2|M1) \\ &= \frac{P(M1 \cup M2)}{P(M1)} \end{aligned} \quad (3.7)$$

, where  $P(M1)$  is the probability of cases containing  $M1$ . The confidence of a rule indicates the degree of correlation in the dataset between the different types of metrics/antipatterns and QoS attributes. The Confidence level is considered as a measure to evaluate the strength of the rule. A high confidence is required for the selected association rules.

**Lift:** Another important measure to evaluate the generated association rules is the lift defined as the confidence of the rule divided by the expected level of confidence. In other words,

$$\begin{aligned}
lift(M1 \Rightarrow M2) &= \frac{confidence(M1 \Rightarrow M2)}{P(M2)} \\
&= \frac{P(M1 \cup M2)}{P(M1) * P(M2)}
\end{aligned}
\tag{3.8}$$

, where  $P(M1)$  is the probability of cases containing M1.

In general, we consider a lift value that is higher than 1 as an indication that the occurrence of M1 has a positive effect on the occurrence of M2 or it confirms that positive correlation between M1 and M2.

If the lift score is smaller than 1, it is considered as an indication that M1 and M2 are not appearing frequently thus the occurrence of M1 has a negative effect on the occurrence of M2 and M1 is negatively correlated with M2. A lift value almost equal to 1 indicates that we cannot conclude about the correlation of M1 and M2.

After validating the correlations to be able to generate statistically significant association rules, we qualitatively validated the rules by identifying the most important interface and code metrics for each of the quality of service QoS attributes. Since this is the first study to generate these association rules, we were not able to compare with any existing studies.

To answer **RQ3**, we evaluated the impact of the 5 different types of antipattern on the QoS attributes by checking the average severity score on each of the QoS attributes. The severity score is defined as the average value of the quality attribute in web services of our data set that did not contain a specific antipattern type divided by the average value of the quality attribute on the web services of our data set containing that antipattern type. We normalized all the quality attributes between 0 and 1 using the min-max normalization (to be minimized). Thus, the highest value is the most severe indication of the impact of an antipattern on each of the quality of service.

### 3.2.3.3 Results

**Results for RQ1.** Table 3.11 summarizes the list of the best three association rules linking the code/interface metrics and three different quality of service (QoS) attributes related to response time, reliability and compliance. These rules are obtained by using 5 fold cross validation as described in [548]. We also tried, by trial and error, other values of k for the cross validation but 5 folds gave us the best results. Our model was able to find positive correlations mainly with three out of the eight well-know QoS attributes: response time, availability, throughput, successability, reliability, compliance, latency and documentation. It is expected that not all these quality attributes can be predicted using code and interface level metrics. In fact, some quality attributes such as availability may depend more on hardware requirements but not the quality of the code/interface implementation. Thus, we believe that the results are consistent.

Table 3.12 contains the average, max and min support, confidence and lift of each rule in table 3.11. When generating the rules, we used the value 0.6 as a threshold for the support and confidence.

It is clear that the three rules are confirming the strong correlation between response time, compliance and reliability; and many of the quality metrics. For instance, a high response time is correlated with low coupling, an acceptable number of operations per interface (around 12), and high cohesion. Typically, the estimation of these quality of service requires to deploy and run the service then the values will be calculated during a period of time. However, the outcomes of RQ1 confirms that it is possible to predict three of the QoS attributes from the quality of the implementation.

The outcome of the first research question is important for service providers so they can estimate the impact of the quality of their code/interface on the QoS attributes before approving new releases for the users. The generated association rules can be used for reviewing any new pull requests by linking the quality of the code on the QoS attributes.

To summarize, there are strong correlations between three QoS attributes and the quality

**Table 3.9:** Support, confidence and lift of the rules that predict QoS from anti-patterns

Rule ID	average support	max support	min support	average confidence	max confidence	min confidence	average lift	max lift	min lift
4	0.66	0.715	0.6125	0.674493	0.731304	0.626087	1	1.021739	0.98913
5	0.783992	0.922045	0.66232	0.789475	0.922045	0.66232	0.999714	1	0.995068
6	0.726667	0.8	0.633333	0.726667	0.8	0.633333	1	1	1
7	0.811288	0.899159	0.718348	0.893765	0.91536	0.872267	0.998636	1.003929	0.992246
8	0.770913	0.947742	0.613238	0.786026	0.966349	0.625359	1.000092	1.00559	0.997162
9	0.810105	0.831005	0.789204	0.941341	0.96561	0.917072	0.999718	1.000758	0.998679
10	0.815322	0.939007	0.618464	0.840257	0.967647	0.637442	1.001737	1.005185	0.999242

of the code/interface of services.

**Results for RQ2.** Table 3.10 summarizes our findings. All the different five types of antipatterns are strongly correlated with different types of QoS attributes. These rules are obtained using the same fold cross validation to answer RQ1. The table shows the activate rule for each antipattern and the associated QoS attributes. Ambiguous Service antipatterns are experiencing, in general, a high response time which is understandable due to the low reusability and the high coupling in these services. Chatty services and Multi services have the highest negative impact on quality attributes: response time, latency, availability and successability. In fact, these two antipatterns are related to large services including high number of operations and low cohesion which increase the probability of decreasing the quality. Nano service is also correlated based on three different association rules with low best practices and latency due to the small size of these services including few operations.

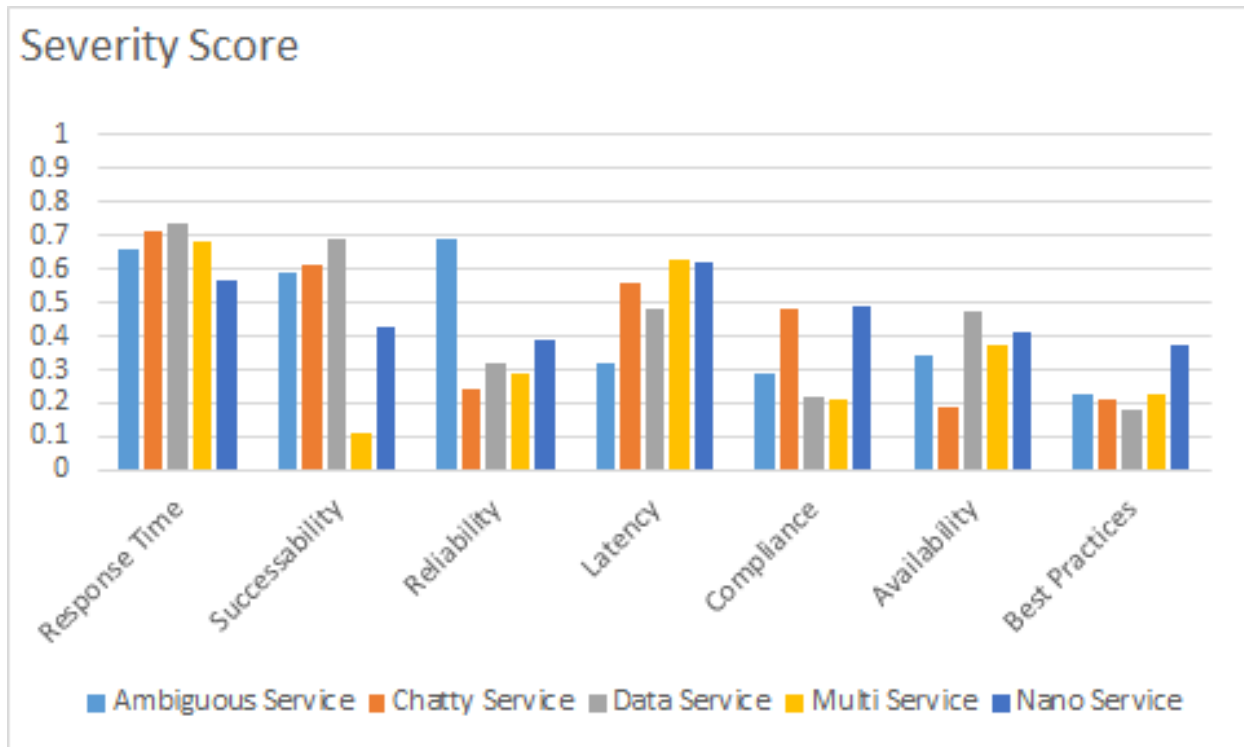
Table 3.9 contains the average, max and min support, confidence and lift of each rule in Table 3.10. We also used 0.6 as a threshold for the support and confidence when generating the rules. The way we read the rules in Table 3.10 is the following: when we know we have one of the antipatterns and one of the left hand sides of the corresponding rules is true, then the right hand side of that rule is also true. For example, when we know we have the antipattern Ambiguous Service and we have AMTO in the range of  $[0,0.6]$  then we can conclude that Response Time is in the range of  $[80.4,368]$ , Successability is within  $[86.8,100]$  and Reliability is in the range of  $[63.8,76.6]$ .

To conclude, we found that the five types of antipatterns have negative impacts on the performance of services and can be used to predict the QoS.

**Table 3.10:** Rules to predict QoS from anti-patterns

Rule ID	Anti-Pattern	QoS Prediction Rules
4	Ambiguous Service	$AMTO = [0, 0.6) \Rightarrow ResponseTime = [80.4, 368) \ \& \ Successability = [86.8, 100] \ \& \ Reliability = [63.8, 76.6)$
5	Chatty Service	$(COH = [0.21, 0.42]) \ OR \ (NOM = [47, 85]) \ OR \ (NCT = [51, 69]) \ OR \ (RAOD = [0.55, 0.74]) \ OR \ (NOD = [23, 42]) \ OR \ (NPT = [1, 3]) \Rightarrow ResponseTime = [55.5, 401) \ \& \ Latency = [0.33, 58.9) \ \& \ Compliance = [86.9, 100] \ \& \ Successability = [87.7, 100] \ \& \ Reliability = [63, 75.9)$
6	Data Service	$ANOPO = [5.32, 28.5] \ OR \ NOM = [84, 462] \ OR \ COH = [0.36, 0.98] \ OR \ NAOD = [18, 141] \Rightarrow Latency = [1.23, 58.7) \ \& \ ResponseTime = [227, 1290) \ \& \ Successability = [91.9, 100] \ \& \ Documentation = [4, 28.3) \ \& \ Availability = [90.7, 100]$
7	Multi Service	$NOPT = [7.8, 78] \ OR \ NCT = [32, 287] \ OR \ COH = [0.01, 0.43] \ OR \ NOD = [17, 231] \Rightarrow ResponseTime = [55.5, 574) \ \& \ Latency = [0.33, 89.7)$
8	Nano Service	$NST = [0, 8) \Rightarrow Successability = [89.6, 100] \ \& \ BestPractices = [79.6, 93] \ \& \ Latency = [0.33, 227) \ \& \ ResponseTime = [46, 635)$
9		$COUP = [0.36, 0.99] \Rightarrow ResponseTime = [46, 635) \ \& \ Latency = [0.33, 227)$
10		$NPT = [0, 2) \Rightarrow ResponseTime = [46, 635) \ \& \ Latency = [0.33, 227) \ \& \ BestPractices = [79.6, 93]$

**Results for RQ3:** Table 3.10 shows that the most severe antipattern in terms of response time is the Data Service. Among Chatty Service, Data Service, Nano Service and Multi Service, the most severe antipattern in terms of latency is Nano Service. To better investigate the severity of the antipatterns, we compare between the average values of the quality attributes in web services containing a specific type of antipattern comparing to the quality attributes average for the ones without antipatterns. Figure 3.9 summarizes the severity of antipatterns results. It is clear that the response time quality is the main attribute negatively impacted by most of the antipattern types. Ambiguous services have



**Figure 3.9:** The average severity score of the different types of antipattern on the QoS attributes based on our data set of web services

a high severity on the reliability comparing to the remaining types of antipattern. Chatty services decreased all the quality of service attribute based on the obtained results. Response time, successability and latency are heavily decreased comparing to the remaining quality of service attributes. The results of RQ3 can be used by the service providers to identify the types of antipattern to be fixed based on which quality attribute they want to improve.

In summary, data service, chatty service and multi service are among the severest antipattern types on the quality of service attributes.

### 3.2.4 Threats to Validity

In our experiments, construct validity threats are related to the absence of similar work based on machine learning to predict the QoS. Thus, we were not able to compare our results with any of existing studies. A construct threat can also be related to the fact that we had to manually choose the best discretization method for every metric and train our model based

**Table 3.11:** Rules to predict QoS

Rule ID	Right hand side of the rule: Performance Metric	Left hand side of the rule: Interface Metrics
1	<i>ResponseTime</i> = [46, 617)	<i>ALMS</i> = [1, 4.24) <i>OR</i> <i>ALOS</i> = [1, 2.77) <i>OR</i> <i>NBB</i> = 1 <i>OR</i> <i>COH</i> = [1.00e - 02, 7.46e - 01) <i>OR</i> <i>NPT</i> = 1 <i>OR</i> <i>NPM</i> = [0.5, 1.58) <i>OR</i> <i>NBE</i> = [0, 16.1) <i>OR</i> <i>NIOP</i> = [0, 4.62) <i>OR</i> <i>NAOD</i> = [0, 16.2) <i>OR</i> <i>NOPT</i> = [0.33, 12.59) <i>OR</i> <i>ANIPO</i> = [0, 4.75) <i>OR</i> <i>NOM</i> = [2, 29.5)
2	<i>Compliance</i> = [86.7, 100]	<i>NBB</i> = 1 <i>OR</i> <i>ALMS</i> = [1, 4.24) <i>OR</i> <i>NPT</i> = 1 <i>OR</i> <i>ALOS</i> = [1, 2.77) <i>OR</i> <i>COH</i> = [1.00e - 02, 7.46e - 01) <i>OR</i> <i>NIOP</i> = [0, 4.62) <i>OR</i> <i>NPM</i> = [0.5, 1.58) <i>OR</i> <i>NAOD</i> = [0, 16.2)
3	<i>Reliability</i> = [66.1, 89]	<i>NBE</i> = [0, 16.1) <i>OR</i> <i>NOPT</i> = [0.33, 12.59) <i>OR</i> <i>NOM</i> = [2, 29.5) <i>OR</i> <i>NAOD</i> = [0, 16.2) <i>OR</i> <i>NIOP</i> = [0, 4.62) <i>OR</i> <i>NPM</i> = [0.5, 1.58)

**Table 3.12:** Support, confidence and lift of the Rules to predict QoS

Rule ID	Average Support	Max Support	Min Support	Average Confidence	Max Confidence	Min Confidence	Average Lift	Max Lift	Min Lift
1	0.73063	0.872308	0.647052	0.924195	0.949536	0.901303	1.00808	1.03572	0.98310
2	0.668805	0.713046	0.615475	0.802937	0.875489	0.726074	1.086702	1.18489	0.982685
3	0.694400	0.754661	0.619794	0.861134	0.934675	0.784948	1.13252	1.22924	1.032310

on that.

Internal threats to validity are related to the fact that, in our approach, the prediction is made for each QoS property separately. This isolated prediction is reasonable when the QoS properties are independent, but many QoS properties are correlated, such as response time and latency. The same observation is also valid for the possible combination of multiple antipattern types to predict some quality of service attributes. For instance, multiple instances of both Multi-Service and Chatty-Service can be grouped to predict some quality attributes. We are planning to extend our work to consider such dependencies.

External validity refers to the generalization of our findings. In this study, we performed our experiments on more than 700 web services. A larger dataset is needed to give more reliable results. Since existing studies have confirmed that the programming language affects

the quality of the software [551, 552], the impact of antipatterns on the quality of the code might vary from one programming language to another. This can affect the generalization of our results since all Web services considered in our study are written in Java. In our future work, we are planning to include Web services written in other programming languages.

### 3.2.5 Conclusion and Future Work

We propose, in this contribution, a novel approach to predict QoS with the least cost using code/interface quality metrics and antipatterns. The output of our approach consists of 10 association rules that predict the performance of web services. We used 5 fold cross validation to evaluate the rules. The obtained results based on 707 web services confirm the correlation between both code/interface metrics/antipatterns and the QoS attributes. This important outcome can be used to understand the severity of antipatterns and predict the quality of the services based on the current quality of the implementation.

Our results show that data service, chatty service and multi service are the most severe antipatterns types on the quality of service attributes among the studied antipatterns. All the QMOOD metrics are affected by antipatterns at different levels. Best practices, availability and compliance are the quality metrics deteriorated the most by antipatterns.

As part of our future work, we plan to extend our work to consider other types of antipatterns (such as Redundant PortTypes (RPT), CRUDy Interface (CI) and Maybe It is Not RPC (MNR) [348]) and metrics (such as Performance, Integrity and Usability [536]). Furthermore, we are planning to try other machine learning algorithms such as decision trees for generating association rules and compare their outputs with this work. Finally, we will extend our work to consider the correlation between metrics when predicting the QoS using dimensionality reduction techniques.

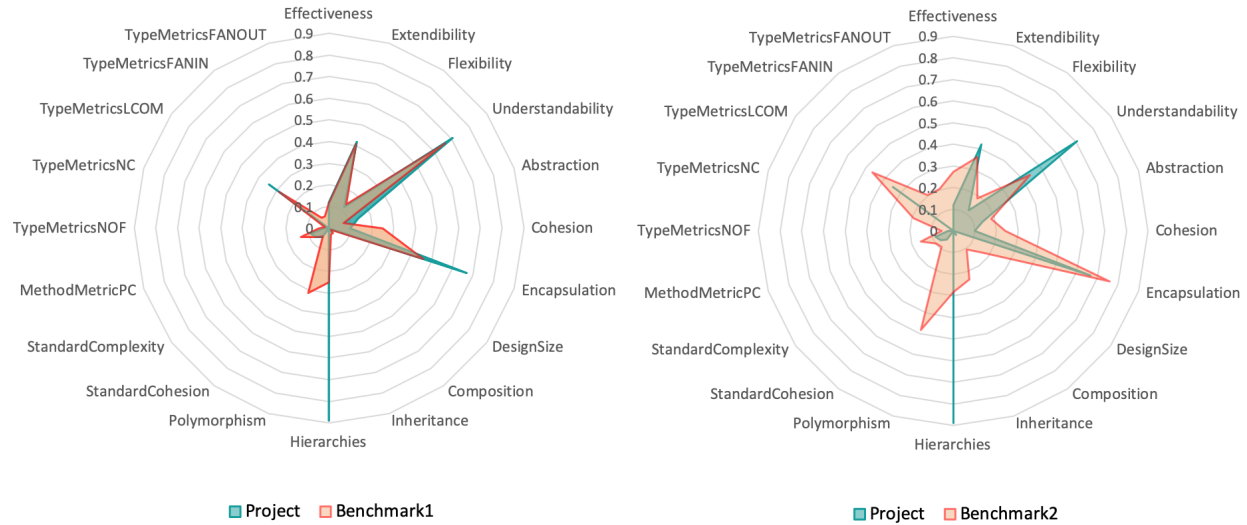


### 3.3 One Size Does Not Fit All: Customized Benchmark Generation for Software Quality Assessment

#### 3.3.1 Introduction

Detection of quality issues has received much attention from the research community [553]. The majority of these studies are based on quality metrics to detect issues and anti-patterns using a variety of techniques such as rules [554, 555, 556], search-based methods [299, 406, 28], machine learning [557, 558, 559], and formal methods [560, 561]. Once the quality issues are identified, the next step is to prioritize them and refactor them [562, 563, 110]. Extensive refactoring approaches are proposed to improve quality metrics and to fix anti-patterns [10, 564, 565, 444]. One of the main challenges when adopting quality metrics is the selection of the thresholds for determining the need for refactoring. These thresholds are impossible to generalize as they are very dependent on the context. Though some studies based on machine learning [566, 567] and genetic programming [176] are proposed to define the thresholds for quality metrics, they are all dependent on the employed dataset (i.e., benchmark). Currently, the sensitivity of quality metrics *w.r.t.* the selected benchmarks is not known and therefore identifying the appropriate benchmark to determine quality issues has not been explored.

Benchmarking is a common practice to establish baselines, to define best practices, to set performance expectations, and to identify improvement opportunities. In software development, benchmarking allows companies to gain an independent perspective on how well their software performs from the chosen quality perspective compared to other products. Such comparisons create a healthy competitive environment within the organization and enable a culture of continuous improvement. When a wrong benchmark is selected, the corresponding quality model produces inaccurate results and therefore inaccurate understanding of the relative quality performance. This may lead the organization to pursue the wrong goals, move the organization in the wrong direction, or at minimum waste valuable resources.



**Figure 3.10:** Quality evaluation of a project A using two different benchmarks

Figure 3.10 shows the quality evaluation of an open-source project, Opencsv<sup>7</sup>, (shown in blue) against two different benchmarks created from randomly selected projects that we collected from Github (shown in orange). The figure reveals that the two different benchmarks give different results and therefore lead to different interpretations and plan of actions. The presented benchmarks have different quality shapes on the radar charts and thus represent different quality profiles. For example, benchmark 2 suggests that Opencsv has good quality values (better than the average values of the benchmark) in all metrics except *Extensibility*, *Understandability* and *Hierarchies* [515]. Thus, developers may decide to refactor their code to improve these three metrics. However, benchmark 1 would suggest otherwise. All three of these metrics have values above those of the benchmark, and the interpretation suggests that developers need to focus on improving *Cohesion*, *MethodmetricPC*, *Polymorphism*, *TypeMetricsFanin*, and *TypeMetricsFanOut*. From this discussion, it is evident that the identification of the right benchmark is important for an accurate quality assessment. Although existing work has not directly addressed automated customized benchmarking for software quality, a few studies provided mechanisms to differentiate small, inactive, or low-quality repositories from high-quality active ones [433, 435, 434]. Furthermore, some other attempts evaluated

<sup>7</sup><http://opencsv.sourceforge.net/>

the quality of different releases of the same software [440, 439].

To address this gap, we start from the observation that an appropriate benchmark for a project should be based mainly on characteristics such as size, number of contributors, number of releases, and number of commits along with other repository features to describe the context. We collect a set of quality and repository metrics of open-source projects of different sizes and categories. We filter these projects and apply different clustering algorithms to find clusters with distinct characteristics based on the repository features. We picked the best set of clusters based on well-defined criteria that will be discussed later in this contribution. Each cluster is considered as a benchmark for projects with similar repository features. After that, we investigate the sensitivity of quality metrics with different clusters/benchmarks. Finally, we validate our approach by applying it on several projects from eBay and compare the obtained quality assessment results with the manual evaluations of programmers. The results show the effectiveness of repository features in finding clusters of projects with different characteristics and also show that quality metrics are sensitive to the selected cluster/benchmark. The validation of our approach with our industrial partner, eBay, confirms the effectiveness of our approach in finding a suitable benchmark and evaluating software quality.

Finally, our study makes the following contributions to the field:

- A method to create benchmarks for software quality evaluations based on repository features.
- A demonstration of the usage of clustering algorithms and comparison of their performance to find clusters of projects that can be used as benchmarks for evaluating the quality of software projects.
- An empirical study to understand the impact of the benchmark on the quality assessment results and the sensitivity of quality metrics.

**Replication Package.** All material and data used in our study are available in our

replication package [568].

### 3.3.2 Research Methodology

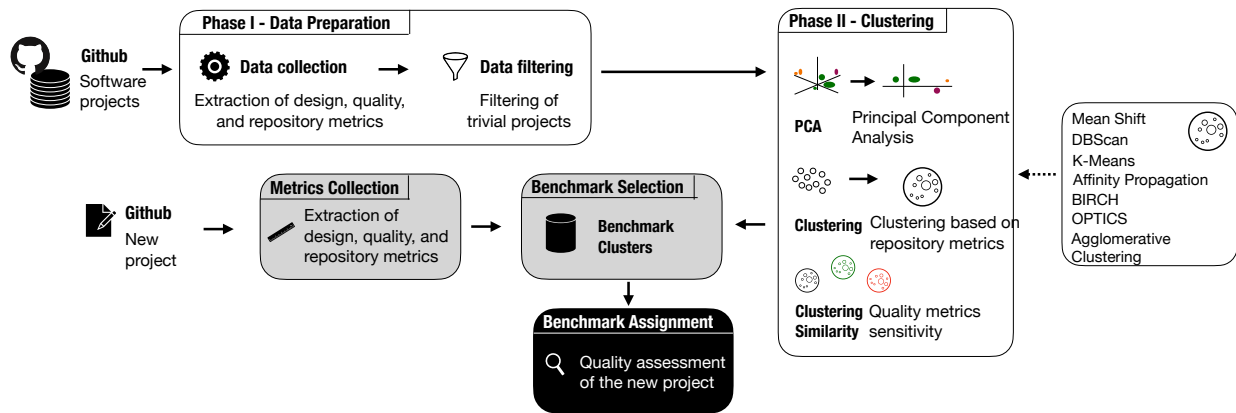


Figure 3.11: Overview of the proposed approach.

Figure 3.11 summarizes our approach. The main goal of the proposed approach is to automatically identify the most suitable benchmark for a given project to enable an appropriate and fair assessment of its quality. Intuitively, each benchmark should include projects with similar characteristics such as the number of commits, number of contributors, size, age, domain, and language. We analyze a set of filtered projects and extract these repository metrics using Git API. We created a set of filtered projects to avoid analyzing repositories that are too small or with one contributor. We included GitHub projects having at least 5K LOC and at least two contributors.

We cluster the projects based on the repository features to find clusters with distinct characteristics. We use different clustering algorithms (K-means, Affinity propagation, BIRCH, Agglomerative clustering, OPTICS, DBSCAN, and Mean shift) and compared their performance. The goal of this step was to validate the hypothesis that software projects can be grouped together into multiple clusters based on their characteristics to form benchmarks. We identify the best clustering algorithm balancing the following criteria: number of projects per cluster, high evaluation metrics, and distinct characteristics per cluster.

After the validation of the first hypothesis, we parse source code of 54,569 GitHub

projects to extract quality attributes, code smells, and repository features. We filtered these projects based on several criteria such as the number of contributors and the size so we finally considered 5,079 projects. We perform principal component analysis [569] to reduce the dimensions of the feature space. Then, we study the sensitivity of the quality metrics to the different benchmarks using statistical tests to validate the hypothesis of whether the quality of projects between clusters are significantly different and to identify which metrics are more sensitive to the benchmarks. Finally, we validate the usefulness and efficiency of our approach on industrial projects. The next sections explain each of the above components.

### 3.3.2.1 Data Collection Phase

Munaiah et al. [433] implemented a framework called *Reaper* to enable researchers select GitHub repositories that contain evidence of an engineered software project. They provided a publicly-accessible dataset composed of 1,857,423 GitHub repositories<sup>8</sup>. We cloned 54,569 projects from that dataset and extracted design, quality, and repository features as described in the following subsections.

**3.3.2.1.1 Repository Metrics** A repository is a basic unit in GitHub that contains the source code and resource files of a software project. It also stores information related to the project’s evolution history and high-level features as well as to the persons who create, contribute, fork, start, and watch it. After collecting the list of repositories, we used a wrapper<sup>9</sup> for the GitHub API to extract 29 repository metrics described in the online appendix [568] such as the number of commits, age, number of contributors, etc. The repository features and their detailed definitions are available in the appendix.

**3.3.2.1.2 Object-oriented Design Metrics** Object-oriented design metrics represent the structural health of a software system. We used DesigniteJava [486], a software design

<sup>8</sup><https://reporeapers.github.io/results/1.html>

<sup>9</sup><https://github3py.readthedocs.io/en/master/>.

quality assessment tool to detect a comprehensive set of metrics. The tool classifies these metrics into three categories.

**Class-level metrics:** Number of Fields (NOF), Number of Methods (NOM), Number of Public Fields (NOPF), Number of Public Methods (NOPM), Lines of Code (C.LOC), Weighted Methods per Class (WMC), Number of Children (NC), Depth of Inheritance Tree (DIT), Lack of Cohesion of Methods (LCOM), Fan-in (FANIN), Fan-out (FANOUT).

**Method-level metrics:** Lines of Code (M.LOC), Cyclomatic Complexity (CC), Parameter Count (PC).

**Component-level metrics:** Lines of Code (C.LOC).

**3.3.2.1.3 Code Smells** In this project, we employed DesigniteJava [486] to detect architecture, design, and implementation smells as described in section 2.4.1.3. We aggregated the code smells detection results by creating four project-level metrics (i.e., Smell Density (P\_SMD), Architecture Smell Count (ASC), Design Smell Count (DSC), Implementation Smell Count (ISC)) and one component-level metric (i.e., Smell Density (C\_SMD)).

### 3.3.2.2 Clustering Phase

The goal of the clustering phase is to find categories of projects that can be used as benchmarks. The idea here is to partition the projects into groups based on the similarity in their repository attributes. The intuition behind this is that repositories that have similar characteristics, such as number of contributors, number of lines of code, and number of classes are likely to represent similar context with one another. Due to the large number of data points, high dimensional input spaces, and variable noise, we first perform principal component analysis (PCA) as described in Section 3.3.2.2.1. Then, we use multiple clustering algorithms to classify the projects and compare their performance.

**3.3.2.2.1 Principal Component Analysis** Principal component analysis (PCA) is a technique commonly used to reduce the dimensionality of large feature set. The technique finds a subset of variables while retaining most of the information from the original set. This method increases interpretability while minimizing information loss. The method takes a collection of data points in two-, three-, or higher-dimensional space and draws a “best fitting” line that minimizes the average squared perpendicular distance from a point to the line. Similarly, the next best-fitting line can be chosen from directions perpendicular to the first line. Repeating this process leads to an orthogonal basis in which different individual dimensions of the data are uncorrelated. These basis vectors are referred to as principal components.

**3.3.2.2.2 Clustering Algorithms** We compare seven widely used unsupervised learning algorithms described in Table 3.13 to choose a clustering algorithm best suited for the dataset at hand. We chose these algorithms because they belong to three different clustering groups namely, hierarchical clustering algorithms, density-based clustering algorithms, and Partitioning Clustering algorithms. These are the most popular clustering algorithm categories and were used in similar studies [321, 291]. For each algorithm, we try various combinations of hyper-parameters that we describe in Section 3.3.3.2.

### 3.3.2.3 Quality Metrics Sensitivity

The goal of this phase is to investigate the difference in the quality assessment obtained using different benchmarks. In other words, we study the sensitivity of the quality metrics to the different benchmarks using statistical tests to validate the hypothesis whether the quality of projects between clusters are significantly different and identify the metrics that are more sensitive to benchmarks. For each quality metric  $Q_i$  and each pair of clusters  $C_x$  and  $C_y$ , we perform statistical tests to see whether the distribution of the values of  $Q_i$  is different within

<sup>9</sup><https://scikit-learn.org/stable/modules/clustering.html>

**Table 3.13:** Overview of the used clustering algorithms.

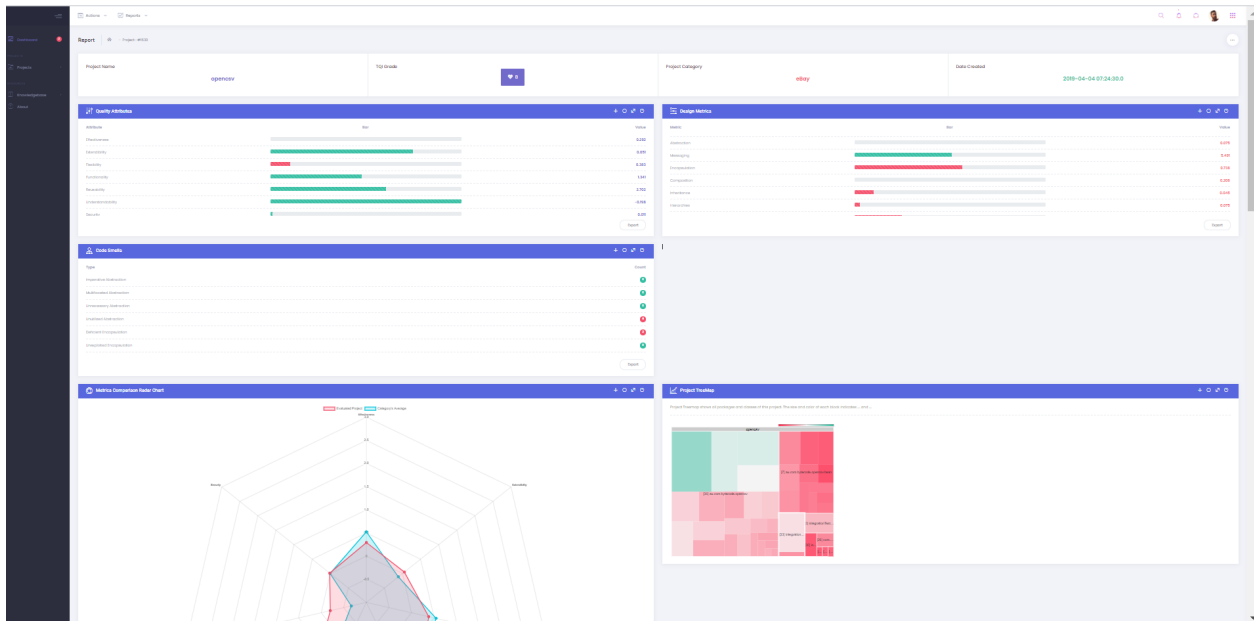
Algorithm	Definition	Parameters	Use Case	Geometry (metric used)
K-Means	It partitions $n$ observations into $k$ clusters in which each observation belongs to the cluster with the nearest mean (i.e., centroid)	number of clusters	General-purpose, even cluster size, flat geometry, not too many clusters	Distances between points
Affinity Propagation	It creates clusters by sending messages between pairs of samples until convergence.	damping, sample preference	Many clusters, uneven cluster size, non-flat geometry	Graph distance (e.g. nearest-neighbor graph)
Mean-shift	It aims to discover blobs in a smooth density of samples.	bandwidth	Many clusters, uneven cluster size, non-flat geometry	Distances between points
Agglomerative Clustering	It uses a bottom-up approach to build nested clusters by merging or splitting them successively based on their similarity.	number of clusters or distance threshold, linkage type, distance	Many clusters, possibly connectivity constraints, non-Euclidean distances	Any pairwise distance
DBSCAN	It groups together points that are close to each other based on a distance measurement and a minimum number of points.	neighborhood size	Non-flat geometry, uneven cluster sizes	Distances between nearest points
OPTICS	It can be considered a generalization of DBSCAN that relaxes the distance requirement from a single value to a value range.	minimum cluster membership	Non-flat geometry, uneven cluster sizes, variable cluster density	Distances between points
Birch	It uses hierarchical methods to cluster and reduce data. It constructs a tree data structure with the cluster centroids being read off the leaf.	branching factor, threshold, optional global clusterer.	Large dataset, outlier removal, data reduction.	Euclidean distance between points

each cluster. If the distribution is different, it means that  $Q_i$  is sensitive to the benchmark. Since the data is not normally distributed and the clusters have different sizes, we used a non-parametric test—the Kolmogorov–Smirnov test [570]. Developers need to carefully consider the selected benchmark, especially when evaluating sensitive quality metrics. They need to target quality metric values better or equal to the selected benchmark values when generating refactoring recommendations. If the metric is not sensitive to the benchmark, developers may avoid wasting their time on the benchmarking process as all benchmarks would reflect a similar ranking of  $Q_i$ . The details of the statistical tests will be discussed later in this contribution.



### 3.3.2.4 QBench: A Software Quality Benchmarking Platform

To enable the use of our study for software quality assessment, we implemented a cloud platform, *QBench*, that helps developers to select the right benchmark for their projects and get an evaluation report about the quality issues. Figure 3.12 shows a screenshot of QBench’s dashboard which provides an overview of the current quality status of the different projects in the database. The tool also generates a detailed report for each project that provides easy to understand insights about its quality. The developer starts with providing the link to the GitHub repository of the project to evaluate. The tool clones the repository and collects all the required repository features, calculates quality metrics, and detects code smells. Next, it identifies the most suitable benchmark to assess the quality of that project using our approach as detailed in the experiments section. The user can then view the detailed report page for that project as shown in Figure 3.13. The red/green bars indicate by how much quality metrics are lesser/greater than the average of those metrics in the benchmark. The user can also change the benchmark manually and can also select the quality metrics on the radar chart according to his/her individual or organizational needs and preferences.



**Figure 3.12:** QBench dashboard showing quality profile of the selected project.

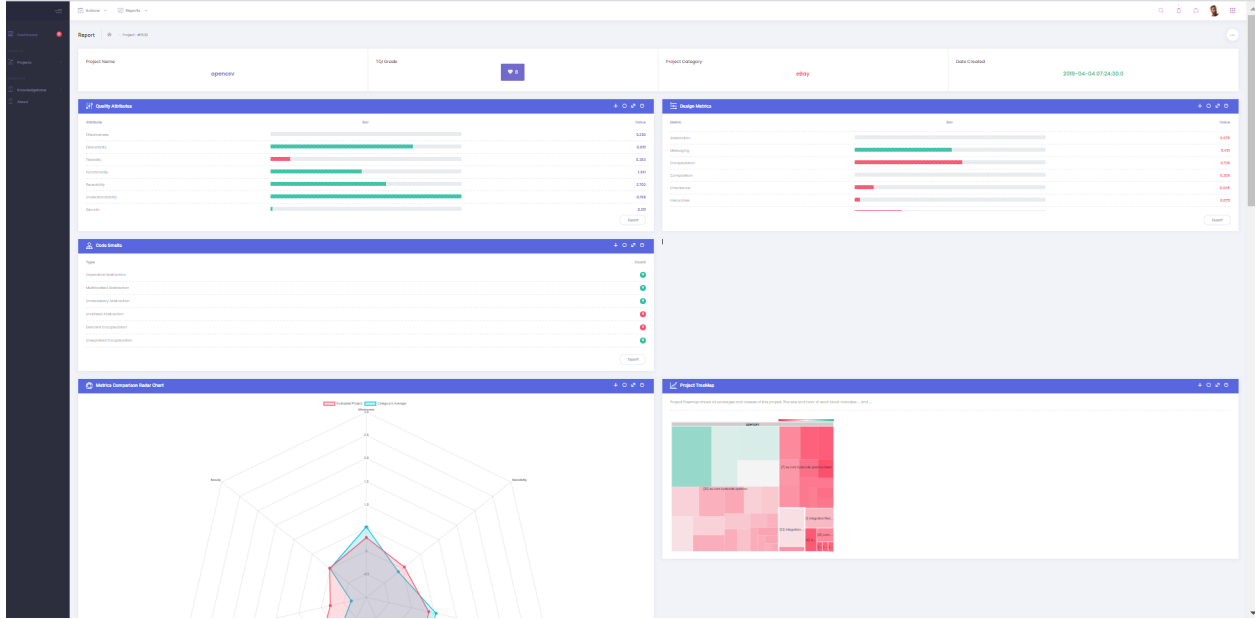


Figure 3.13: QBench’s detailed quality report.

### 3.3.3 Empirical Validation

In this section, we first present our research questions and validation methodology. Then, we discuss the obtained results.

#### 3.3.3.1 Research Questions

In this study, we addressed three main research questions.

**RQ1. Benchmarks generation.** Do clustering algorithms distinguish between categories of software projects based on repository features?

**RQ2. Quality metrics sensitivity.** How sensitive are the quality metrics to the generated clusters?

**RQ3. Industry validation.** To what extent can the proposed approach identify the right benchmark and quality profiles in practice?

Intuitively, developers think about those repository features (i.e., number of contributors, number of commits, size, etc.) when they decide to compare their projects to others. Thus,

RQ1 aims to validate whether projects can be grouped based on similar repository features behavior. We started by collecting a set of 54,569 open-source projects (over 1 billion lines of code) and extracting 29 repository metrics. We then filtered these projects and extracted a list of non-trivial repositories that satisfy the following criteria to eliminate toy projects: the repository must be written in Java and must have at least two contributors and 5,000 lines of codes. We finally considered 5,079 after the filtering step. We performed the Principal Component Analysis (PCA) and used the minimum number of components that capture at least 95% of the total variance in the dataset to eliminate redundant/unnecessary repository features. After the data cleaning, we applied the following widely used clustering algorithms: K-means, affinity propagation, BIRCH, Agglomerative clustering, OPTICS, DBSCAN, and Mean shift. We used different combinations of features and hyper-parameters to compare their performances and determine the best clustering algorithm for our problem. Since we do not know the ground truth class assignments, we used the *Silhouette coefficient* to evaluate the different clustering results. The *Silhouette Coefficient* [571] is a measure that determines how similar an object is to its own cluster (cohesion) compared with other clusters (separation). It is calculated as follows:

$$\frac{(b - a)}{\max(a, b)} \quad (3.9)$$

Where **a** is the mean intra-cluster distance and **b** is the mean nearest-cluster distance for each sample. We chose this metric because it provides a sound mechanism to evaluate the clustering results and has been extensively used in research studies [572].

After the validation of the first hypothesis, the goal for RQ2 is to validate the second hypothesis that investigates whether and to what degree quality metrics are sensitive to the benchmarks. We performed the *Kolmogorov–Smirnov test* (also referred to as K–S test or KS test) which is a popular non-parametric test and distribution-free test (i.e., it makes no assumption about the distribution of data). The KS test can be used to compare a sample

with a reference probability distribution, or to compare two samples. We wanted to validate that the clusters of projects generated by our approach have different quality distributions and therefore provide a different quality assessment. We chose this test because it can be applied to unequal sample sizes unlike other statistical tests (e.g. the Silhouette score, the Student's t-test, Analysis of Variance Test (ANOVA), Analysis of cluster variability (ANOCVA) etc.). This condition is necessary because the clusters of projects we obtained are unequal in size. We set our null hypothesis to be the following.

$H_0$ : The distribution of a given quality metric  $Q_i$  is equal between the two benchmarks  $B_x$  and  $B_y$ .

This means that our alternative hypothesis is as follows.

$H_A$ : The distribution of a given quality metric  $Q_i$  is not equal between the two benchmarks  $B_x$  and  $B_y$ .

We determined the level of significance to be 0.05. A  $p$ -value  $\leq 0.05$  is statistically significant. It indicates strong evidence against the null hypothesis. In that case, we reject the null hypothesis and accept the alternative hypothesis. A  $p$ -value  $> 0.05$  is not statistically significant and indicates strong evidence for the null hypothesis. Therefore, we retain the null hypothesis and reject the alternative hypothesis in that case. After computing the  $p$ -value for each pair of benchmarks, we computed a sensitivity metric  $S_{Q_i}$  for each quality metric  $Q_i$  that we define as follows.

$$S_{Q_i} = \frac{\sum_{x,y \in \text{Benchmarks}} \delta(x,y)}{n(n-1)/2},$$

$$\delta(x,y) = \begin{cases} 1, & \text{if } P\text{-val}_{B_x, B_y} \geq 0.05 \\ 0, & \text{if } P\text{-val}_{B_x, B_y} < 0.05 \end{cases} \quad (3.10)$$

where  $n$  is the number of benchmarks,  $n(n-1)/2$  is the total number of unique pairs of benchmarks, and  $P\text{-val}_{B_x, B_y}$  is the  $P$ -value of the *Kolmogorov-Smirnov test* applied on the

benchmarks  $B_x$  and  $B_y$ .  $S_{Q_i}$  takes values between 0 and 1; 1 means that the  $Q_i$  has a different distribution in all pairs of benchmarks. In other words,  $Q_i$  is very sensitive to the benchmarks.  $S_{Q_i} = 0$  means that we cannot conclude anything about the distribution of a given quality metric  $Q_i$  between any pairs of benchmarks. We also assigned labels (i.e., high, medium, and low) to each quality metric. *High* means that  $S_{Q_i}$  is between 1 and 0.66. *Medium* means that  $S_{Q_i}$  is between 0.66 and 0.33. *Low* means that  $S_{Q_i}$  is between 0.33 and 0. This sensitivity measure is useful to understand how severe is the sensitivity of each quality metric to the benchmarks which can impact the decision of developers to fix the code or not.

For RQ3, the goal is to evaluate the relevance of this study in practice for the assessment of the quality of software projects. Thus, we developed a platform, QBench, integrating the proposed benchmarking approach as detailed in section 3.3.2.4. The best way to validate the relevance of our study in practice is to perform a manual validation with the **active and original developers** of large-scale industrial projects who are truly knowledgeable about the projects and can make an accurate evaluation of the quality based on their extensive experience. Thus, we opted to perform a manual evaluation with four developers who are the main contributors and original developers of the four most critical back-end software projects of eBay. These projects are very critical from a quality perspective for eBay as poor quality of code will impact, for example, the response time of their platform. *Xoneor* is the main project managing online payment, *Relational-data-service* manages calls between databases, *Xodbutil* enables the data integration from multiple databases, and *Picasso* analyzes the profile of users and their history. Since the selection of the right participants (i.e., original and knowledgeable senior developers) is the key rather than the number of participants, the four participants were selected using the following criteria as detailed in Table 3.14: (1) number of commits; (2) number of years working on the project(s); (3) number of modified files; and (4) their experience in detecting and fixing quality issues. Each participant evaluated one repository and they were not aware of the goals of our study to avoid any potential

bias. We extracted the quality and repository features and predicted the most suitable benchmark for each industry project using our predictive model selected in the first research question. Then, for each project, we asked its contributor to rate every quality metric based on their experience without revealing our benchmarking results. They assigned one of the following labels: *very low*, *low*, *medium*, *high*, and *very high* to each of the quality metrics for the evaluated project. Moreover, participants were allowed to leave an optional comment justifying their assessment. After that, we compared the obtained quality assessment results with the manual evaluation of developers. We finally asked them to give their opinion on a scale from 1 to 5 about the sensitivity of each quality metric.

**Table 3.14:** Selected Developers and eBay projects.

Project	KLOC	Participants		
		Exp (Years)	Commits	Modified files
<i>Xoneor</i>	456	8	327	913
Relational-data-service	703	12.5	281	603
<i>Xodbutil</i>	681	11	331	498
<i>Picasso</i>	642	14	369	703

### 3.3.3.2 Hyper-parameter Settings

Some of the clustering algorithms (e.g. K-means, Birch, and Agglomerative Clustering) require specifying the number of clusters up-front; therefore, we tried all cluster numbers between 2 and 10 following trail-and-error. For the Agglomerative Clustering algorithm, we tried the different types of distance—*Euclidean distance*, *Manhattan distance*, and *cosine similarity* as the affinity parameters. For the Affinity propagation algorithm, the Damping factor should be a value between 0.5 and 1, exclusive; we tried the following damping factors: 0.5, 0.6, 0.7, 0.8, and 0.9. We set the preferences to the median of the input similarities. For the Mean-shift algorithm, we used the Python *sklearn* module that offers a function to estimate the bandwidth based on a nearest-neighbor analysis. The DBSCAN algorithm requires two parameters: Minimum samples (“MinPts”) which is the minimum number of points required to form a cluster and  $\epsilon$  (epsilon or “eps”) which is the maximum distance two

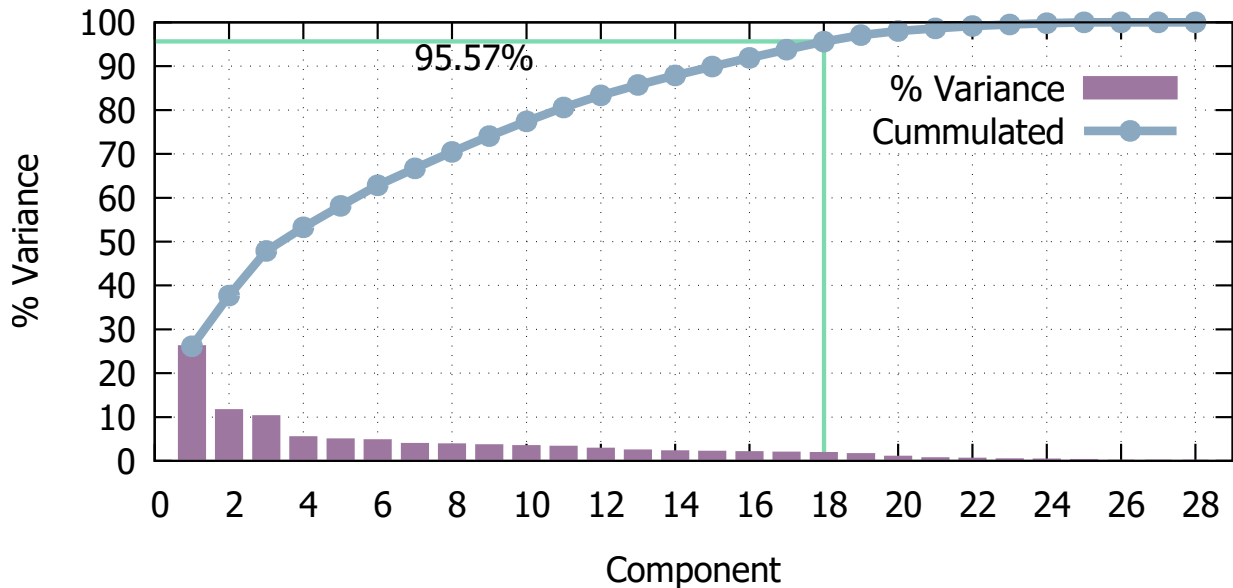
points can be from each other while still belonging to the same cluster. There is no automatic way to determine the *MinPts* value for DBSCAN. For this reason, we tried all the following values: 20, 50, 100, 200, 300, 500, and 700. One technique to automatically determine their optimal  $\epsilon$  value is described by Rahmah et al. [573]. This technique calculates the average distance between each point and its  $k$  nearest neighbors, where  $k$  = the selected *MinPts* value. The average  $k$ -distances are then plotted in ascending order on a  $k$ -distance graph. The optimal value for  $\epsilon$  resides at the point of maximum curvature (i.e., where the graph has the greatest slope). We found that the optimal *eps* parameters for 20, 50, 100, 200, 300, 500, and 700 are equal to 0.4, 0.5, 0.5, 0.6, 0.7, 0.7 and 0.8, respectively. Finally, for OPTICS, we used the default value of *Max\_eps* which is infinity. This will identify clusters across all scales. For the Min samples, we used the values 20, 50, 100, 200, 300, 500, and 700. The online appendix [568] includes the extensive results using all the above configurations discussed in this section.

### 3.3.3.3 Results

**3.3.3.3.1 Results for RQ1** Figure 3.14 shows the variance percentage of each component created by the principal component analysis. To capture at least 95% of the data variance, we selected the first 18 components that have a cumulative variance of 95.18%. One advantage of PCA is its ability to deal with highly correlated variables, if any. If a set of repository variables are highly correlated then they will all show on the same principal component. The weights of each repository feature in every component can be found in our online appendix [568].

RQ 3.15 shows the clustering results as well as the Silhouette Coefficients. For each algorithm, we picked the hyper-parameters that balance the following criteria: number of projects per cluster, high evaluation metrics, number of unclassified projects, and distinct characteristics per cluster. The detailed results with the different hyper-parameters can be found in our online appendix [568]. The algorithm that has the best Silhouette score is

K-means with K=7. K-means also has relatively balanced clusters with a minimum size of 279 projects and a maximum size of 1356 projects. DBSCAN and OPTICS have high numbers of unclassified projects. Affinity propagation generated 104 clusters. Most clusters contain a very small number of projects. Thus, we can conclude that K-means provides the best clustering results among all 7 algorithms based on the different clustering evaluation metrics.



**Figure 3.14:** A bar-plot that shows the component importance generated by the principal component analysis

To better understand the main differences between the generated clusters of the best algorithm (K-means), Figure 3.15 summarizes grid of boxplots that shows the distributions of repository features across the clusters generated by the K-means algorithm with K=7. Clusters 0, 2 and 6 have the lowest max, min, and median in network count, number of comments, subscribers count, number of releases, number of contributors, open issues, stargazers count, and total number of commits. According to the metrics Project component count, project LOC, project method count, and project type count, clusters 0, 2, and 6 contain small projects compared to the other clusters. Clusters 4 and 5 have the maximum max, min, and median in network count, number of comments, subscribers count, stargazers count, number



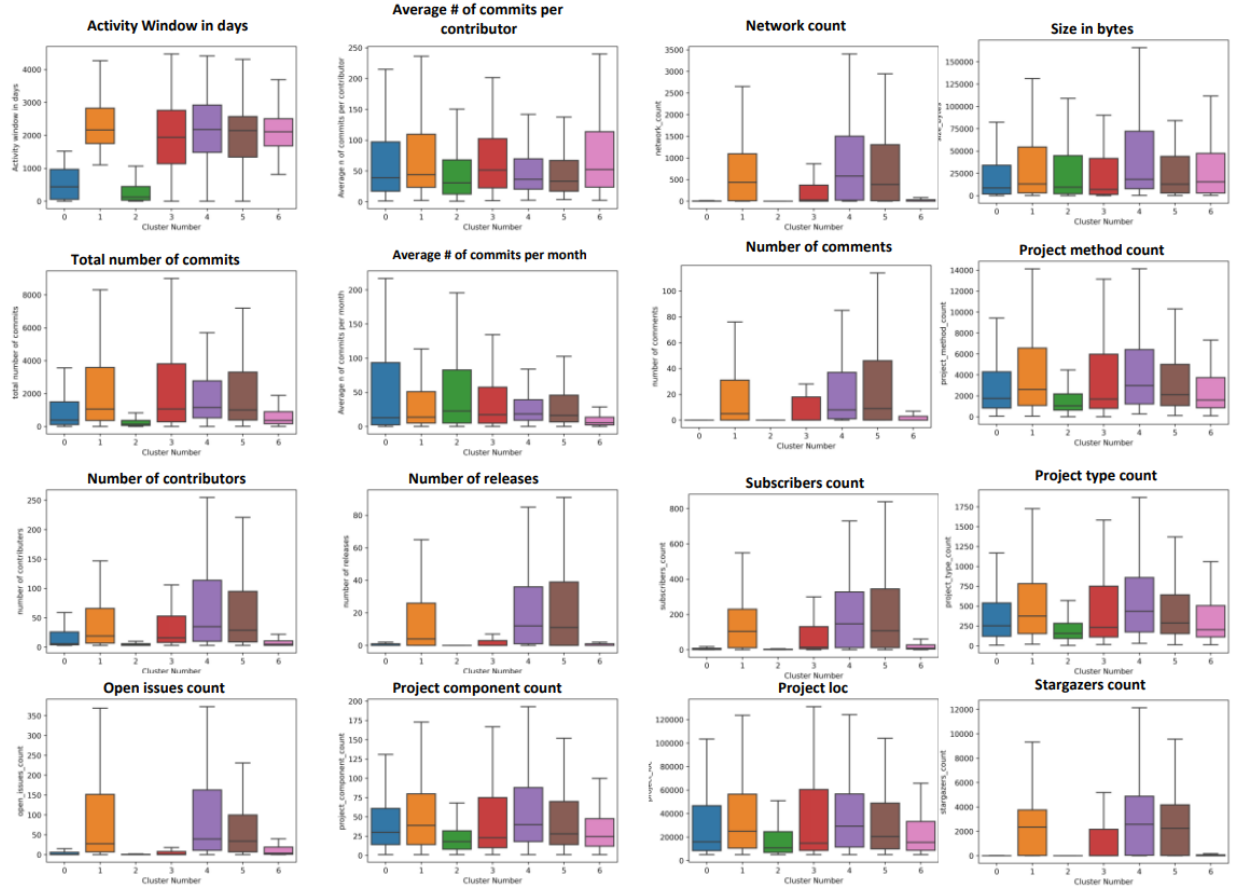
**Table 3.15:** Clustering results.

Algorithm	# of Clusters	Clusters Size	Unclassified Elements	Silhouette Score
K-means	7	1356, 998, 989, 590, 497, 370, 279	0	0.35
DBSCAN	4	1849, 1512, 512, 318	888	0.35
Agglomerative Clustering	7	1309, 1243, 653, 569, 535, 444, 326	0	0.34
OPTICS	6	1866, 1512, 535, 324, 155, 106	581	0.32
Birch	7	3622, 555, 441, 168, 147, 77, 69	0	0.18
Mean Shift	6	4246, 321, 255, 164, 67, 26	0	0.12
Affinity Propagation	104	Max: 638, Min: 2	0	0.00

of contributors, and number of releases. Based on the obtained results, we can confirm that the repository features can be used to automatically generate clusters of projects sharing similar characteristics. These clusters are evaluated in the next RQ to see if they could be used as benchmarks for software quality to simulate the way how developers intuitively select the best benchmark for quality assessment.

**🔍 Key findings:** Software projects can be grouped together into multiple clusters based on their repository features.

**3.3.3.3.2 Results for RQ2** Figure 3.16 summarizes the p-value results of the *Kolmogorov-Smirnov test* for all pairs of benchmarks. The x-axis contains all possible pairs of benchmarks. The y-axis contains the quality metrics. The bubbles in the intersection between a quality metric  $Q_i$  and a pair of benchmarks  $(B_x, B_y)$  means that  $B_x$  and  $B_y$  are significantly different in  $Q_i$  (i.e., p-value < 0.05). We notice that *ASC* and *DSC* have the highest number of bubbles which means that they have significant differences across almost all pairs of benchmarks which can lead to completely different quality assessment based on the selected benchmark. In other words, if developers change the benchmark when evaluating *ASC* or *DSC*, they will most likely get a completely different interpretation of the quality metrics value. On the other hand, *PC*, *FANIN* and, *FANOUT* have the least number of



**Figure 3.15:** A grid of boxplots that shows the distributions of repository features across the clusters generated by the K-means algorithm with  $K=7$

bubbles. However, that number is still considerable.

To have a clearer idea about the sensitivity of the quality metrics, we present the values of  $S_{Q_i}$  for all quality metrics as well as their corresponding labels in RQ **RQ2**. We notice that all metrics have either high or medium sensitivity to the benchmarks with the predominant label being high. As we mentioned before,  $ASC$  and  $DSC$  have the highest sensitivity with a value of 0.95. This observation can be explained by the fact that these anti-patterns are dependent on the history and structure of the projects. Meanwhile, this is an important outcome for the software maintenance and evolution community to carefully consider the benchmark use when deciding about anti-patterns to fix. The class metrics  $FANIN$  and  $FANOUT$  and  $PC$  have the lowest sensitivity with a value of 0.52. However, it is still be considered at least as a medium sensitivity level to the benchmark. We believe that the main outcomes of

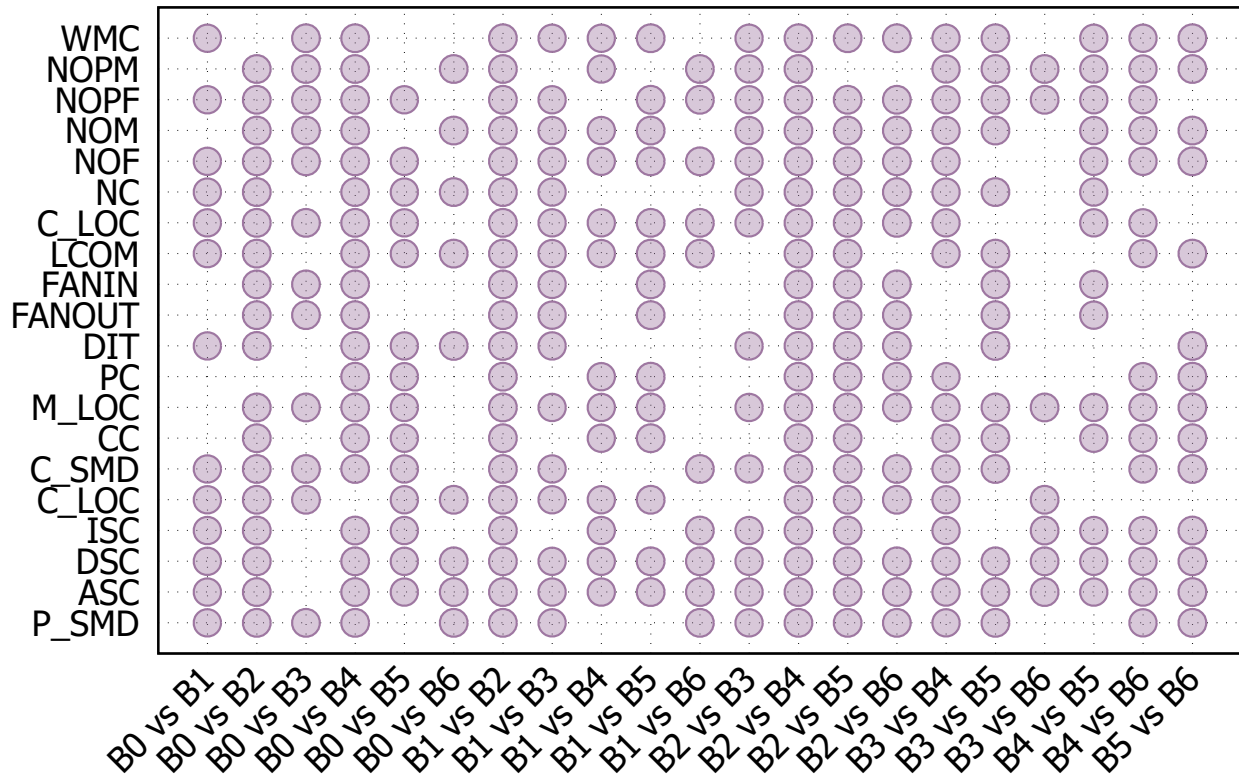


Figure 3.16: A bubble chart that summarizes the Kolmogorov–Smirnov test results.

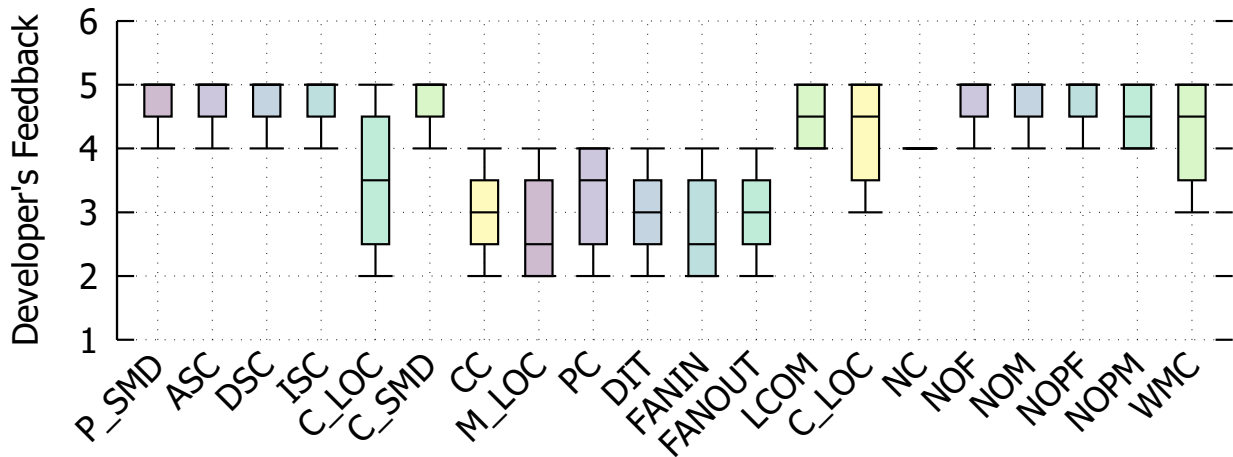
this research question are very important to the communities of practitioners and software maintenance: *Quality metrics are sensitive to the selected benchmark thus developers and researchers should carefully select the right benchmark to get the right interpretation of the quality assessments.*

**Q Key findings:** All the quality metrics used in our study are sensitive to the benchmarks. This outcome suggests that the appropriate selection of the benchmark is critical for an accurate assessment of the quality of software systems.

**3.3.3.3.3 Results for RQ3** Table 3.17 summarizes the results of the comparison between the quality evaluation of *QBench* and the ones conducted manually by the senior developers from eBay. For each quality metric  $Q_i$ , the eBay participants labeled each of the quality metric values as *very low*, *low*, *medium*, *high*, and *very high* based on their experience and knowledge of the projects. To reduce the subjectivity of this process, we asked the participants to evaluate all the four projects quality metrics since they are knowledgeable about

**Table 3.16:** The sensitivity of the quality metrics

Quality Metrics	<i>QBench</i> $S_{Q_i}$	Label	Survey $S_{Q_i}$
P_SMD	16/21=0.76	high	0.95
ASC	20/21=0.95	high	0.95
DSC	20/21=0.95	high	0.95
ISC	15/21=0.71	high	0.95
C_LOC	14/21=0.67	high	0.7
C_SMD	16/21=0.76	high	0.95
CC	13/21=0.61	medium	0.6
M_LOC	18/21=0.86	high	0.55
PC	11/21=0.52	medium	0.65
DIT	12/21=0.62	medium	0.6
FANIN	11/21=0.52	medium	0.55
FANOUT	11/21=0.52	medium	0.6
LCOM	16/21=0.76	high	0.9
C_LOC	17/21=0.81	high	0.85
NC	14/21=0.67	high	0.8
NOF	18/21=0.86	high	0.95
NOM	17/21=0.81	high	0.95
NOPF	18/21=0.86	high	0.95
NOPM	15/21=0.71	high	0.9
WMC	16/21=0.76	high	0.85



**Figure 3.17:** Distribution of the developers' answers about the Sensitivity of the quality metrics.

all of them and the results were the same for all projects and metrics (Cohen's Kappa score of 1.0). We extracted the necessary repository features for each eBay project. The evaluation of *QBench* is based on the preferred cluster for each project from the 7 clusters identified in the

previous research question using K-means. Then, *QBench* labeled automatically the quality metrics based on the ranking of their values in the preferred cluster: 0-20% (very low), 20%-40% (low), 40%-60% (medium), 60%-80% (high), and 80%-100% (very high). Then, the developer can make the interpretation of the assessment based on the type of the metric (to maximize or to minimize). We notice that our approach provided a correct assessment for all quality metrics except for four. *QBench* was also able to provide an accurate quality assessment for the projects *Relational data service* and *Picasso*. *Xodbutil* has a divergence in the quality evaluation mainly in the metrics related to code smells. However, all the differences are slight deviations of the assessment such as high vs. very high which may not impact significantly the overall assessment by the developers.

Table 3.18 shows the precision of the quality assessment of each project using all seven benchmarks. We used the assessment provided by the participants as our ground-truth (expected evaluation). Then, we computed the precision of *QBench* assessment for each benchmark/cluster by calculating the size of the intersection between the expected assessments of developers and those of *QBench* divided by the total number of quality metrics for each project. The preferred benchmarks predicted by our approach for *Xoneor*, *Relational Data Service*, *Xodbutil*, and *Picasso* are benchmarks 2, 4, 1, and 2, respectively. For the projects *Xoneor* and *Picasso*, the preferred cluster detected by *QBench* was able to provide the best quality assessment (i.e., The highest precision compared to the ground truth) which confirms the similarity between the selected cluster/benchmark of *QBench* and the opinions of developers.

Finally, Table 3.16 shows the average sensitivity judged by all the participants for each quality metric which is consistent with our approach. Figure 3.17 also shows the distribution of the developers' opinion. According to the developers, all metrics are sensitive to the benchmark. The lower sensitivity is indicated in the metrics *FANIN*, *FANOUT*, *DIT PC*, *CC* and *M\_LOC*. This outcome confirms our results where our approach demonstrates medium sensitivity in five of those six metrics.

The current version of QBench is integrated within eBay’s development pipeline, as part of the quality gates, for specific critical projects/services running in the backend of ebay.com. When the developers are pushing pull-requests for merging, QBench estimates its impact on the project metrics based on the most appropriate benchmark and highlights the quality metrics that may need to be improved before merging the pull-request. QBench is also currently used by managers and executives to estimate the amount of accumulated technical debt compared to the identified benchmark before shipping a new release. Basically, QBench is used in two scenarios by developers. First, within the continuous integration process where QBench provides an impact summary of the code changes in the pull-request from the quality perspective compared to an automatically identified benchmark. Depending on the differences with the benchmark, QBench (quality gate) may generate a warning that the pull-request cannot be merged until some issues get refactored or another code reviewer/manager approves bypassing the quality gate of QBench. The second scenario is during major refactoring phases where the amount of accumulated issues (technical debt) becomes high compared to other benchmarks. However, we have seen more activities and interests for the first scenario during this pilot.

<p><b>Q Key findings:</b> <i>QBench</i> demonstrates high effectiveness in automatically finding a suitable benchmark when evaluating the quality of software projects in practice.</p>
---

### 3.3.4 Threats to Validity

**Internal validity.** The first threat to our approach is the selection of an appropriate clustering algorithm and its corresponding hyper-parameters. Clustering algorithms have many parameters, and finding the best combination of parameters is not easy. To mitigate this problem, we tried many combinations of parameters from different ranges. We also used heuristics such as the elbow and k nearest neighbors methods as well as the scikit-learn function for mean-shift bandwidth selection.

**Construct validity.** In our experiments, these threats are related to the absence of similar works that automatically identifies customized benchmarks to evaluate the quality of

**Table 3.17:** Participants quality assessment vs *QBench*.

<i>QBench</i> / Develop. Opinion	Xoneor	Relational Data Service	Xodbutil	Picasso
P.SMD	Medium/Medium	Low/Low	Medium/High	High/High
ASC	Medium/Medium	Low/Low	Medium/Medium	Very High/Very High
DSC	High/High	Low/Low	Low/Medium	High/High
ISC	Medium/Medium	Low/Low	Medium/High	High/High
C.LOC	Medium/Medium	Medium/Medium	High/High	High/High
C.SMD	High/Medium	Low/Low	Medium/Medium	High/High
CC	Medium/Medium	Medium/Medium	High/Very High	High/High
M.LOC	Medium/Medium	Low/Low	High/High	Medium/Medium
PC	Medium/Medium	Medium/Medium	Medium/Medium	Medium/Medium
DIT	Low/Low	High/High	Low/Low	Low/Low
FANIN	High/High	Low/Low	Medium/Medium	High/High
FANOUT	Medium/Medium	Low/Low	Medium/Medium	High/High
LCOM	Medium/Medium	Low/Low	High/High	Very High/Very High
C.LOC	Medium/Medium	Medium/Medium	Medium/Medium	High/High
NC	Low/Low	High/High	Medium/Medium	Low/Low
NOF	Medium/Medium	Medium/Medium	Low/Low	Medium/Medium
NOM	Medium/Medium	Low/Low	High/High	Very High/Very High
NOPF	Medium/Medium	Medium/Medium	Medium/Medium	High/High
NOPM	Medium/Medium	Low/Low	Medium/Medium	High/High
WMC	High/High	Low/Low	High/High	High/High

**Table 3.18:** *QBench* correctness precision for each of the seven benchmarks.

Projects/ Pref Bench.	Bench						
	1	2	3	4	5	6	7
<i>Xoneor</i> (Benchmark 2)	64%	95%	51%	63%	53%	55%	39%
<i>Relational data service</i> (Benchmark 4)	48%	68%	42%	44%	100%	63%	23%
<i>Xodbutil</i> (Benchmark 1)	80%	56%	71%	52%	66%	84%	47%
<i>Picasso</i> (Benchmark 2)	52%	100%	39%	74%	38%	48%	61%

software systems. Thus, we were not able to compare our results with any existing studies. A construct threat can also be related to the fact that we had to manually choose the best clustering results to be considered as our benchmarks. Further work needs to investigate how the results would change in cases in which we choose other clustering results.

**External threats.** External threats concern the generalization of our findings. We included only projects written in Java to create the benchmarks. To mitigate this threat, we made sure that we used projects of different sizes and domains. We filtered trivial projects and kept only the ones that had more than two contributors and more than 5000 lines of code. Moreover, we included only 29 repository metrics and 20 quality measures. We are

planning to include more quality and repository features and consider other programming languages to extend our empirical validation. Another threat is related to the validation of our approach. We validated our approach on only 4 industry projects with 4 senior developers. The reason is that we wanted to have relevant feedback from mainly the original developers of the projects.

### **3.3.5 Conclusion**

In this study, we propose a novel approach that aims at finding the most suitable benchmark to evaluate the quality of a software project in a fair and unbiased way. We first showed that clustering algorithms are efficient in finding clusters of projects with distinct characteristics based on repository features. We then performed statistical analysis to compare the different clusters and to check the sensitivity of each quality metric. We finally validated our approach with developers from eBay. The results provide strong evidence that our approach helps developers automate and effectively manage the benchmarking process for software quality assessment.



## CHAPTER IV

### Improving the Refactoring Recommendation Process

Refactorings constitute an effective means to improve quality and maintainability of evolving object-oriented programs. Search-based techniques have shown promising results in finding sequences of behavior-preserving program transformations that maximize code quality metrics, minimize the number of code smells and minimize the number of changes. However, we identified two major research gaps that received little or no attention so far in the refactoring generation process. The first gap consists of the impact of refactorings on extra-functional properties like security. Security is an important software quality aspect that reflects the ability of a system to prevent data exposure and loss of information. A basic aim of secure software is to prevent unauthorized access and modification of information. The fulfillment of security requirements at the design and implementation level is imperative to minimize the cost of addressing this issue at later stages of development and maintenance. Similar to other quality attributes, it is important to perform refactoring operations to improve the security of software. The second gap consists of the random application of change operators and seeding mechanism in search-based refactoring techniques. Failing to understand the good/bad patterns in a refactoring sequence or the dependency between the refactoring operations can simply destroy them, deteriorate the quality, and delay the convergence towards good solutions.

To address these two gaps, we propose the following contributions:

## 4.1 How Does Refactoring Impact Security When Improving Quality? A Security-Aware Refactoring Approach

### 4.1.1 Introduction

The National Institute of Standards and Technology (NIST) estimates that the US economy loses an average of \$60 billion per year by either implementing patches to fix security vulnerabilities or the impact of these security issues [574, 575]. These vulnerabilities depend on how a system is designed and implemented. At the same time, code quality is also critical: it impacts programmer productivity and may cause project failure as maintenance consumes over 70% of the lifetime budget of a typical software project.

The ISO/IEC-25000 *SQuaRE* (Software product Quality Requirements and Evaluation) [576] classifies software quality in a structured set of eight characteristics and sub-characteristics. In this classification, security is a new characteristic that was created to measure how much a software is resistant to attacks and risks. Therefore, it is crucial to take this characteristic into account when improving the quality of the software.

Several researchers and practitioners have assumed that improving a quality metric of software, such as modularity, will have a positive impact on security, making the design more robust and resilient to attacks [577, 578, 292]. However, this assumption is poorly supported by empirical validations. Architects and developers may not pay much attention to design fragments containing data and logic pertinent to security properties, which makes them overexposed while still improving some quality aspects of their architecture. For instance, a developer may create a hierarchy in a set of classes to improve the reusability of the code. However, these actions may expand the attack surface if the superclass contains critical attributes and methods. Another example that we observed in practice is that improving modularity may result in spreading dependencies on security-critical files into many other components. A security-critical file contains data (e.g., attributes) and logic (e.g., methods) that can potentially be misused to violate fundamental security properties

such as confidentiality, integrity, or availability of a system.

Refactoring to improve the design structure while preserving behavior is widely used to enhance the quality of software systems [579]. Most existing refactoring research focuses on handling conflicting quality attributes [446, 2, 448, 321, 447]. However, the impact of refactoring on security is poorly understood and under-studied. Recent studies estimate the impact of a few refactoring operations on some security metrics based on their definitions, but without empirically validating these assumptions on real software projects [580, 581, 475, 478]. To the best of our knowledge, there is no previous research on the correlations between security metrics and quality attributes, or that provided a tool to recommend refactorings based on the preferences of developers from both quality and security perspectives, and the possible conflicts between them.

In this contribution, we investigate the possible correlations between the Quality Model for Object-Oriented Design (QMOOD) quality attributes[515] and a set of security metrics extracted from source code widely used in the current literature and practice [466, 468]. We also empirically validated the impact of different refactoring types on 8 code security metrics that are primarily related to data access.

We analyzed a total of 30 open-source projects and, based on the outcomes of these analyses showing the conflicting nature of the studies security and quality metrics, we propose a security-aware multi-objective refactoring approach to find a balance between code qualities and security metrics. We formulated the different quality and security objectives as fitness functions to guide the search for relevant refactorings and find trade-offs between them using Non-dominated Sorting Genetic Algorithm (NSGA-II) [495].

We evaluated our tool on this set of 30 projects. Furthermore, we compared our results with an existing multi-objective refactoring tool [2] that only considers code quality, to understand the sacrifice in security measures when improving code quality and vice-versa. The comparison shows that our security-aware approach performed better than the existing approach when it comes to improving the security of systems, and with low cost in terms of

sacrificing code quality. Our survey of 15 practitioners confirmed the efficiency of our tool and the importance of considering security while improving other qualities. More details about the surveys, experiments and tool can be found in the online appendix [3].

The primary contributions are as follows:

1. The study introduces one of the first empirical studies to understand the impact of source code refactoring on both quality and security metrics and the correlations between them.
2. The creation of a framework to recommend refactorings to find trade-offs between quality and security objectives considering the correlation results between them.
3. A validation of this framework on open source systems. The survey with practitioners shows the potential of our work in improving refactoring recommendations by taking into account both security and quality.

#### 4.1.2 Motivating Example

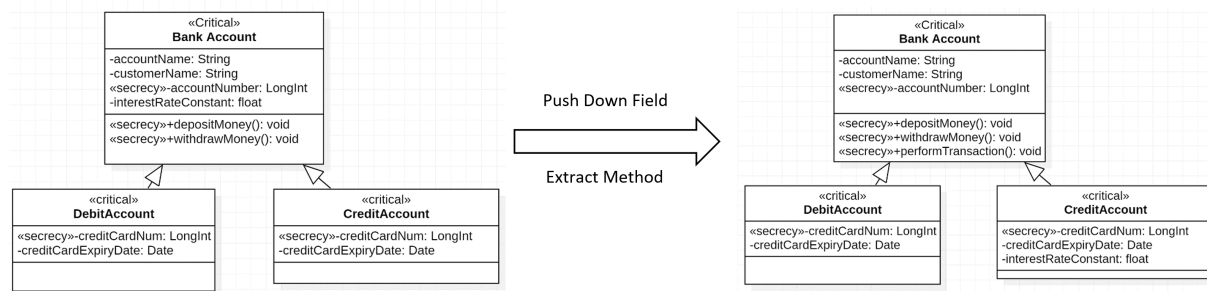
In this section, we describe a motivating example related to the possible negative impact of refactoring on security.

By mining the well-known Common Vulnerabilities and Exposures (CVE) security bug database, we found a total of 269 security vulnerabilities that were introduced by code refactorings. These 269 vulnerabilities were manually identified by the authors out of 681 reports containing the keyword "refactor". Figure 4.1 shows an example of a vulnerability, **CVE-2019-13177**<sup>1</sup>, from Django REST Registration library due to refactorings resulted in allowing remote attackers to trick the verification process. This security bug impacted the confidentiality of Django REST Registration library in several releases before 0.5.0. Thus, it is essential to evaluate the impact of the recommended refactorings on the security of the application.

<sup>1</sup><https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-13177>

CVE-ID	
<b>CVE-2019-13177</b>	<a href="#">Learn more at National Vulnerability Database (NVD)</a> <ul style="list-style-type: none"> <li>CVSS Severity Rating</li> <li>Fix Information</li> <li>Vulnerable Software Versions</li> <li>SCAP Mappings</li> <li>CPE Information</li> </ul>
Description	
verification.py in django-rest-registration (aka Django REST Registration library) before 0.5.0 relies on a static string for signatures (i.e., the Django Signing API is misused), which allows remote attackers to spoof the verification process. This occurs because incorrect code refactoring led to calling a security-critical function with an incorrect argument.	

**Figure 4.1:** An example of a security vulnerability from Django REST Registration library due to refactorings.



**Figure 4.2:** A simplified bank account system hierarchy before and after refactoring

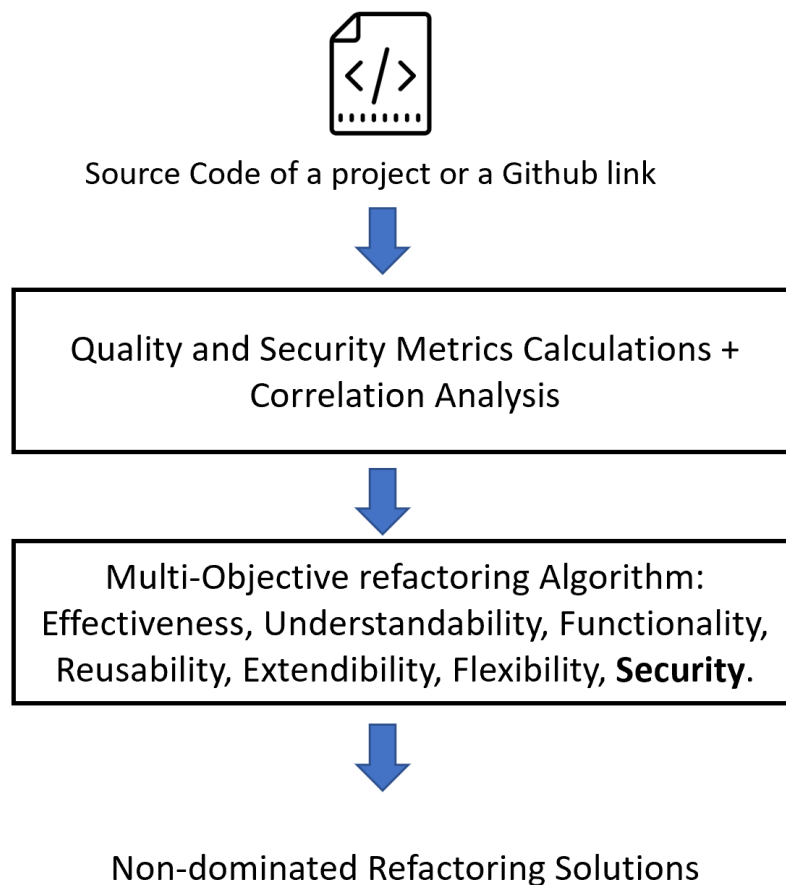
We introduce, in the following, another motivating example to show how refactoring may improve code quality while making the design weaker from a security perspective. The design fragment in Figure 4.2 is responsible for storing information about customer accounts, which, by definition, requires careful attention in terms of security to access those classes. A bank account can be either a debit or credit account. The *interestRateConstant* is an attribute that stores the value of the interest rate of the credit account. Thus, it is only used by the *creditAccount* class. The deposit and withdraw operations have duplicated code that performs the transactions. This code can be extracted to a new separate method that can be used by both operations. Both *accountNumber* and *creditCardNum* are sensitive and are meant to be kept confidential.

The developer applied the refactoring “push down field” by moving the *interestRateConstant* from the *BankAccount* class to its subclass *CreditAccount* as well as the refactoring “extract method” by moving the duplicated code to a separate new method called *performTransaction* and replacing the old code with a call to this new method. These refactorings improved cohesion and messaging [498], which results in increasing the following quality attributes: Understandability, Functionality, and Reusability [582]. However, these trans-

formations might increase the security metrics CMAI, CAAI, CMW, and CAIW [475] which will reduce the security of the design due to the fact that the classes are becoming more exposed and easier to access than before. This example motivates our research to investigate further the impact of refactorings on security when improving code quality.

### 4.1.3 Security-Aware Multi-Objective Refactoring

#### 4.1.3.1 Overview



**Figure 4.3:** Security-Aware Multi-Objective Refactorings

Our approach, as sketched in Figure 4.3, takes as input the source code (or GitHub link) of a project to be analyzed and generates a list of refactoring recommendations that balance code quality and security based on developer preferences.

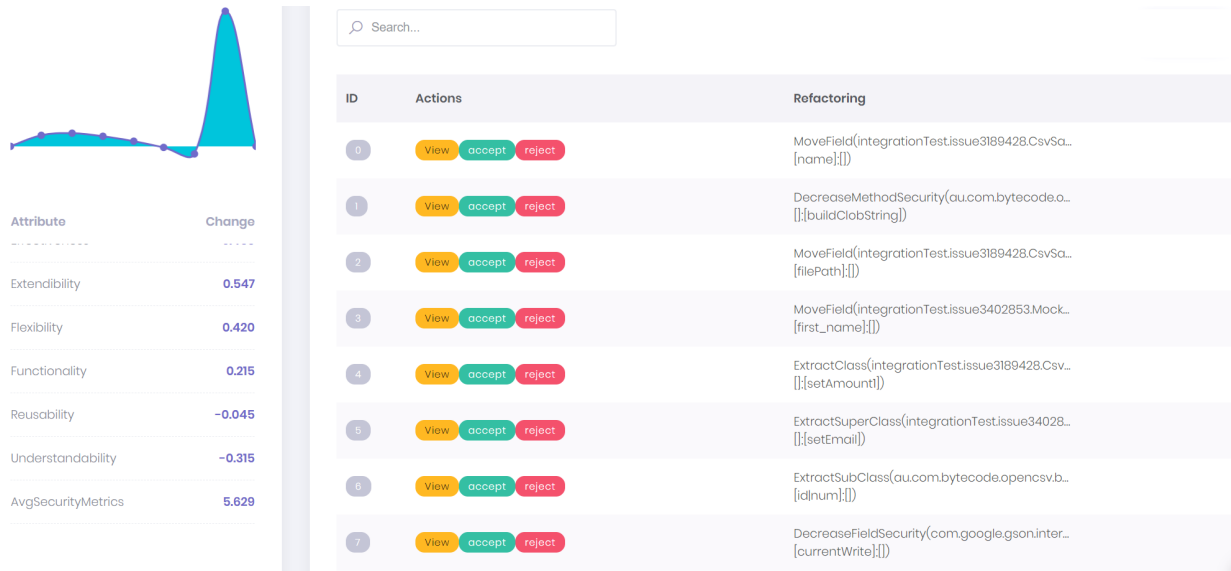
The first component parses the code to calculate the security metrics and quality attributes defined in the background section of this dissertation (Tables 2.10 and 2.12). Then, the collected data is used to analyze the correlation between the different quality and security metrics (without the need for refactoring at this point).

For the second component, we adapted a multi-objective search algorithm, based on NSGA-II [495], used in our previous work [2] to integrate the security and quality objectives. We selected this algorithm due to its ability to find trade-offs between independent or conflicting objectives, and it has previously been applied for various software engineering problems [446, 2, 448, 321, 447]. Our goal is to find a set of non-dominated refactoring solutions capable of improving both the quality and security of the project taken as input. A code refactoring activity may be focused on quality improvements, and the developers care less about security (e.g., the component is used internally and never exposed to attacks). In this case, users of the tool may want to assign higher weights to quality metrics. In another scenario, it could be the opposite, especially for critical code fragments. In our multi-objective formulation, the developer is not required to enter any weights to the objectives since the output of the algorithm is a Pareto-front of a diverse set of solutions that the user can select one of them based on their preferences. Finally, a user can interact with our tool to accept or reject the refactoring recommendations. A detailed demo can be found in [3]. In the remainder of this section, we will explain the steps of the approach.

#### **4.1.3.2 Computing the Security Metrics**

We adopted the Soot parser [524], based on static analysis, to calculate these metrics including the automated identification of classified versus non-classified code elements, as shown in the video of our tool [3].

We chose the 8 security metrics described in Table 2.12 among the ones available in the literature specifically because their definition is clear and relatively easy to implement. We also wanted to highlight that our parser is based on Soot [524] for static analysis—we did not



**Figure 4.4:** Sample of outputs (refactorings) of our Web app on the Open CSV project to balance quality and security.

create a custom parser from scratch. The source code is analyzed to extract the relevant code elements such as classes, methods, attributes, etc. and the relationships between them. Each code element has several attributes that describe its level of access/visibility, whether it is considered to be classified or not.

We describe in the following the different steps to identify the security sensitive attributes by taking inspiration from existing studies [527, 472, 526, 525] based on text similarities/mining. We first use a set of keywords [527, 526, 525] related to security and indicators of sensitive information extracted from multiple sources such as source codes, comments, security bugs, vulnerability reports, commit messages, and security questions/tags on Stack Overflow. We included these keywords in the online appendix associated with this submission. Second, we calculated a textual criticality score, based on cosine similarity, for each file to estimate the extent to which the file is related to security concerns. The higher the score is the more likely the file needs to be protected. We pre-processed the source code using tokenization, lemmatization, stop words filtering and punctuation removal [527, 526]. Then, we computed the cosine similarity between each file and the set of keywords. Finally, we manually validated the top 10 critical files and use their critical attributes (fields that have names that match



one of the keywords from the list we gathered at the beginning) to identify the critical attributes in all the other files that will be used to compute the security metrics. This process, including security metrics calculation, is not time-consuming since it takes a few seconds to minutes to extract the security-critical attributes and compute the metrics, depending on the size of the project to be analyzed. We note that the identification of code elements as security sensitive is not a core contribution of this dissertation since we leveraged the use of existing work for this step.

### 4.1.3.3 Algorithm Adaptation

The search space is composed of the different refactoring operations as well as an exhaustive combination of code locations, attributes, and methods. The algorithm is executed for some iterations to find non-dominated solutions balancing the 7 objectives of improving the 6 QMOOD quality metrics, and the last objective of minimizing the security objective (aggregating the 8 security metrics) in the proposed solutions. The output of this step is a set of Pareto-equivalent refactoring solutions that optimize the above objectives. These solutions are not dominated with respect to each other. A refactoring solution is represented by an ordered vector of refactoring operations as shown in Figure 4.4 and described in the background section 2.4.3.2 of this dissertation .

Our approach takes into consideration seven objectives: the first six are the relative changes of the 6 QMOOD attributes [515] after applying a refactoring solution. Each objective can be written as follow:

$$QualityObjective_i = \frac{Q_i^{after} - Q_i^{before}}{Q_i^{before}} \quad (4.1)$$

where  $Q_i^{before}$  and  $Q_i^{after}$  are the values of the *qualityAttribute<sub>i</sub>* before and after applying a refactoring solution, respectively.

Since all metrics in the table 2.12 are at the class level, we consider the corresponding

system-level metrics as the average of all class level metrics. For instance, *AvgCCDA* is defined as the ratio of the sum of the *CCDA* values of all classes of the system to the number of classes in that system. In a similar manner, we define the other system-level security metrics *AvgCIDA*, *AvgCOA*, *AvgCAAI*, *AvgCMAI*, *AvgCMW*, *AvgCAIW* and *AvgVA*. Therefore, the seventh objective, which is the security objective, corresponds to the relative change in the average of the average of all eight security metrics in the table 2.12 after applying a refactoring solution. We can represent the fitness function of the security objective as follows:

$$SecurityObjective = \frac{S^{after} - S^{before}}{S^{before}} \quad (4.2)$$

where  $S = ( AvgCCDA + AvgCIDA + AvgCOA + AvgCAAI + AvgCMAI + AvgCMW + AvgCAIW + AvgVA ) / 8$

Unlike the quality objectives, we decided to aggregate the security metrics into one objective since they are not conflicting to each other based on our analysis of the data on the open-source systems detailed later in our experiments. Furthermore, the performance of the multi-objective algorithm will decrease when the number of objectives becomes large.

#### 4.1.4 Experiments and Results

We used a set of 30 open source projects to study the possible correlations between 1) the quality and security metrics and 2) refactoring types and security metrics. To evaluate the ability of our security-aware multi-objective refactoring tool to generate good refactoring recommendations that balance both quality and security, we conducted a set of experiments based on 4 out of the 30 open source systems. The obtained results are subsequently statistically analyzed with the aim of comparing our proposal with a variety of existing approaches.

The relevant data related to our experiments and a demo about the main features of the tool can be found in [3]. We have also conducted a survey with practitioners to manually evaluate the refactoring recommendations and the obtained correlations between quality, refactoring types and security.

In this section, we first present our research questions and validation methodology followed by the experimental setup. Then we describe and discuss the obtained results.

#### 4.1.4.1 Research Questions

In this study, we defined four main research questions:

- **RQ1: Impact of refactoring on code security.** Can automated refactoring have a significant impact on security metrics?
- **RQ2: Impact of improving quality on security and vice-versa.** Are there strong correlations between code quality attributes, as measured by the QMOOD metrics, and code security metrics?
- **RQ3: Comparison with an existing work for refactoring recommendation**  
How does our security-aware refactoring tool perform compared to refactoring approaches that only focus on improving quality (and not security)?
- **RQ4: Insights.** Do professional programmers highly value considering security while improving quality?

To answer RQ1, we collected a dataset of refactorings applied on 30 medium to large-size open-source systems, listed in Table 4.1, to understand the impact of the refactoring types on 8 different security metrics. We selected these systems based on their domains, size and large history of evolution (e.g. commits). We did not extract refactorings from previous commits due to the challenges related to differentiating between functional and non-functional changes and the limited number of refactorings that developers apply manually.

Instead, we obtained the data by running the refactoring recommendation tool of Alizadeh et al. [2] on these projects, selecting the obtained refactoring solution and recording its impact on the security metrics. We selected the tool based on its high accuracy in recommending relevant refactorings that significantly improve the quality. Then, we statistically analyzed the impact of these refactoring types on code security metrics for the 30 projects.

To answer RQ2, we used a procedure similar to the one used for RQ1: we collected data from the execution of our tool on the 30 projects by recording the impact of the refactorings on both the QMOOD quality attributes and the 8 code security metrics. Unlike the impact of QMOOD on quality, we note that the security level increases when the security metrics decrease. Finally, we ran statistical tests to understand the correlations between the different metrics using the Pearson correlation coefficient [583] (chosen due to the normal distribution of the data).

To answer RQ3, we compared our approach with an existing technique that considers only the QMOOD attributes [2] as objectives using 4 projects, as described later. Since meta-heuristic algorithms are stochastic optimizers, they can provide different results for the same problem instance from one run to another. For this reason, our study is based on 30 independent simulation runs for each problem instance to make sure that the results were statistically significant. The goal of this research question is to understand the cost of improving code quality on security and vice-versa. We selected the work of Alizadeh et al. [2] since it is the closest to our proposed approach and outperformed most of the existing refactoring tools based on the same systems used in this evaluation. We only considered five systems in this comparison due to the very time-consuming task to run the different heuristic algorithms 30 times to check if the results are statistically significant. Furthermore, it is difficult to find knowledgeable participants who can manually evaluate the results on all 30 open source projects. Thus this part of our evaluation poses a threat to validity.

To answer RQ4, we used a post-study questionnaire to collect the opinions of developers regarding our tool and the relevance of considering security when refactoring. Furthermore,

the participants manually evaluated the refactoring recommendations of our approach. We asked the developers about their opinions on the possible correlations between 1) quality and security metrics; and 2) refactoring types and security. The survey allowed us to compare the quantitative results obtained in our experiments with developer opinions. The full details of our extensive validation, including a demo of our tool and the survey details, can be found at [3].

#### **4.1.4.2 Software Projects and Experimental Setting**

**4.1.4.2.1 Studied Projects** We used a set of 30 well-known open-source Java projects as detailed in Table 4.1. We selected these systems for our validation because they range from medium to large-sized and have been actively developed in recent years. Table 4.1 also provides some descriptive statistics about these programs.

**4.1.4.2.2 Subjects** Our qualitative study involved 15 software developers. All participants were volunteers who were knowledgeable in software security, Java, refactoring, and quality assurance. They were all hired from our former and current industry partners of refactoring projects.

Participants were first asked to fill out a pre-study questionnaire. The questionnaire helped to collect background information such as their role within the company, their programming experience, and their familiarity with software security, quality, and refactoring. They all had a minimum of 2 years of experience as programmers and 5 out of 15 have over 5 years of experience. 12 participants were working on software assurance tasks as part of their regular duties, which was one of the main criteria used to solicit their participation, based on our previous collaborations and contacts. The other criteria were related to their level of expertise in refactoring and security, and also their familiarity with the five selected open source systems. The pre-study survey shows that the majority of the developers (11 out of 15) have high experience and are knowledgeable about software refactoring and security. A

**Table 4.1:** Studied open source projects.

<b>System</b>	<b>Release</b>	<b>#lasses</b>	<b>KLOC</b>	<b>GitHub Link</b>
jFreeChart	v1.0.9	521	170	jfree/jfreechart.git
ArgoUML	v0.3	1358	114	marcusvnac/argouml-spl.git
atomix	v3.0.11	2719	188	atomix/atomix.git
JHotDraw	v7.5.1	585	25	wumpz/jhotdraw.git
GanttProject	v1.10.2	241	48	bardsoftware/ganttproject.git
Apache Ant	v1.8.2	1191	112	apache/ant.git
moshi	v1.8.0	289	27	square/moshi.git
opencsv	v1.7	50	7	jlawrie/opencsv.git
zerocell	v0.3.2	39	3	creditdatamw/zerocell.git
gson	v2.8.5	691	69	google/gson.git
jolt	v0.1.1	370	31	bazaarvoice/jolt.git
Hystrix	v1.5.18	1117	85	Netflix/Hystrix.git
btm	v2.1.3	375	40	bitronix/btm.git
packr	v1.2	8	3	libgdx/packr.git
tracer	v2.0.0	33	3	zalando/tracer.git
JSAT	v0.0.9	1171	185	EdwardRaff/JSAT.git
smile	v1.5.2	1206	8316	haifengl/smile.git
dkpro-core	v1.10.0	1269	1323	dkpro/dkpro-core.git
Erdos	v1.0	128	7	Erdos-Graph-Framework/Erdos.git
jgrapht	v1.3.0	1257	171	jgrapht/jgrapht.git
mockito	v2.27.3	1880	94	mockito/mockito.git
tablesaw	v0.32.7	583	714	lwhite1/tablesaw.git
bazel	v0.25.0	11267	2753	bazelbuild/bazel.git
spotbugs	v4.0.0	5207	389	spotbugs/spotbugs.git
FreeBuilder	v2.3.0	1636	58	google/FreeBuilder.git
async-http	v2.8.1	602	52	AsyncHttpClient/async-http-client.git
javaparser	v3.13.10	1414	251	javaparser/javaparser.git
vavr	v0.10.0	838	135	vavr-io/vavr.git
javamelody	v1.77.0	662	109	javamelody/javamelody.git
commons-cli	v1.4	63	10	apache/commons-cli.git

minimum of 12 of 15 participants per system have medium or above expertise regarding the evaluated open source systems. The full details of our pre-study survey results can be found at [3].

Each participant was asked then to complete an evaluation form to evaluate 5 refactoring solutions that had different impacts on quality and security on 4 different systems: JHotDraw, Gantt, Apache Ant and JFreeChart. The participants were asked to evaluate the refactorings on all the systems; we did not divide them into groups. After that, each participant was given a post-study survey. This second survey was more general as it collected the practitioners' opinions on the relevance of the outcomes and their perception of the importance of considering security when refactoring their code.

**4.1.4.2.3 Parameter Tuning and Statistical Tests** Parameter setting significantly influences the performance of a search algorithm on a problem. For this reason, for each algorithm and for each system, we performed a set of experiments using several population sizes: 50, 100, 200, 300 and 500. The stopping criterion was set to 10,000 evaluations for all algorithms to ensure fairness of comparison. The other parameter values were fixed by trial and error and are as follows: crossover probability = 0.8 and mutation probability = 0.5 where the probability of gene modification is 0.3. We also limited the size of the refactoring solutions to no more than 30 operations.

To have significant results, for each pair (algorithm, system), we used one of the most efficient and popular approaches for parameter setting of evolutionary algorithms which is Design of Experiments (DoE) [584]. Each parameter was 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 were applied.

The following statistical tests show that all the comparisons performed between our approach and existing ones are statistically significant based on all the metrics and the systems considered in our experiments. We used a 2-sample t-test with a 95% confidence

level ( $\alpha = 5\%$ ) to find out whether our sample results of different approaches are significantly different. We also calculated the Pearson coefficient to study the various correlations.

#### 4.1.4.3 Results

**Results for RQ1.** Table 4.2 summarizes the correlations between the different types of refactorings and averaged security metrics considered in our experiments by analyzing the refactoring recommendations generated by our tool on the 30 projects. The results show either a positive or negative correlation based on the Pearson Correlation Coefficient except for the Move Field refactoring. The symbol "++" (strong positive correlation) means that the Pearson correlation coefficient has a value higher than 0.5 while "--" (Strong negative correlation) means the opposite (lower than -0.5). The symbol "+" means that the Pearson correlation coefficient is between 0.1 and 0.5 and "-" means the opposite (between -0.1 and -0.5). The symbol "\*" reflects that the correlation coefficient is around 0 (between -0.1 and 0.1) and there is no statistically significant correlation. For each refactoring type, we filtered the solutions to keep only the ones containing that type and counted its occurrence within the solution. Then, we checked the correlation between the appearance of the refactoring type and its impact on the security metric.

Increase Field Security refactoring has the strongest positive correlation with the average of the 8 security metrics. It is expected that the frequent use of this refactoring type will reduce access to the attributes which may reduce their visibility and reduce the attack surface when a set of classes are exposed to malicious code. The same observation is also valid for Increase Method Security which also has a positive correlation with security improvements. Encapsulate Field, Push Down Field, and Push Down Method refactorings have also positive correlations with the security average measure. It is clear that all these refactoring types reduce the level of abstraction of classes which may increase the protection of the fields and methods.

Decrease Field Security, Decrease Method Security, and Extract Superclass have a strong



negative correlation with the security measure since the Pearson Correlation Coefficient is lower than -0.6. All these refactorings can either make the fields and methods overexposed or increase the abstraction of the code which may have a negative impact on security. The Encapsulate Field refactoring increases the ability to conceal object data. Otherwise, all objects would be public and other objects could get and modify the object's data without any constraints. Furthermore, the encapsulate field refactoring can help in bringing data and behaviors closer together which will reduce unnecessary access and public visibility of attributes. Thus, the security metrics should be improved after application of Encapsulate Field refactorings.

Table 4.2 also shows that Extract Superclass is negatively correlated with the security metrics. One of the main explanations of this outcome is the fact that creating superclasses may expose all the child classes under the created superclass. Thus, the attack surface could be rapidly expanded when this refactoring type is extensively used. In fact, someone who has access to a superclass can affect its subclasses' behavior by modifying the implementation of an inherited method that is not overridden. If a subclass overrides all inherited methods, a superclass can still affect subclass behavior by introducing new methods.

Figure 4.5 and Table 4.3 show the most frequent refactorings and patterns in the solutions that significantly increased the security measure. In this study, a refactoring pattern is an ordered sequence of refactoring operations. We found that the most frequent refactoring types are the ones making the methods and fields less exposed and accessed, which confirms the correlation results. Figure 4.6 describes the impact of refactorings generated by our tool on the 8 security metrics aggregated into one objective. The results show that none of the metrics are conflicting with other security metrics since they were all minimized using the refactoring solutions. This observation confirms our choice to aggregate them rather than considering them as separate objectives. All the security metrics are normalized in the range of [0,1].

To summarize, refactoring can impact code security metrics both positively and nega-

**Table 4.2:** Correlation results between the average of security metrics and different refactoring types on the 30 projects. The results are statistically significant using the 2sample t-test with a 95% confidence level ( $\alpha = 5\%$ )

	<b>Pearson Correlation Coefficient</b>
Encapsulate Field	+ (0.237)
Increase Field Security	++ (0.728)
Decrease Field Security	-- (-0.624)
Pull Up Field	- (0.361)
Push Down Field	+ (0.471)
Move Field	* (0.026)
Increase Method Security	+ (0.358)
Decrease Method Security	-- (-0.681)
Pull Up Method	- (-0.316)
Push Down Method	+ (0.247)
Move Method	- (-0.235)
Extract Class	- (-0.437)
Extract Superclass	-- (-0.694)
Extract Subclass	- (-0.424)
Extract method	- (- 0.472)

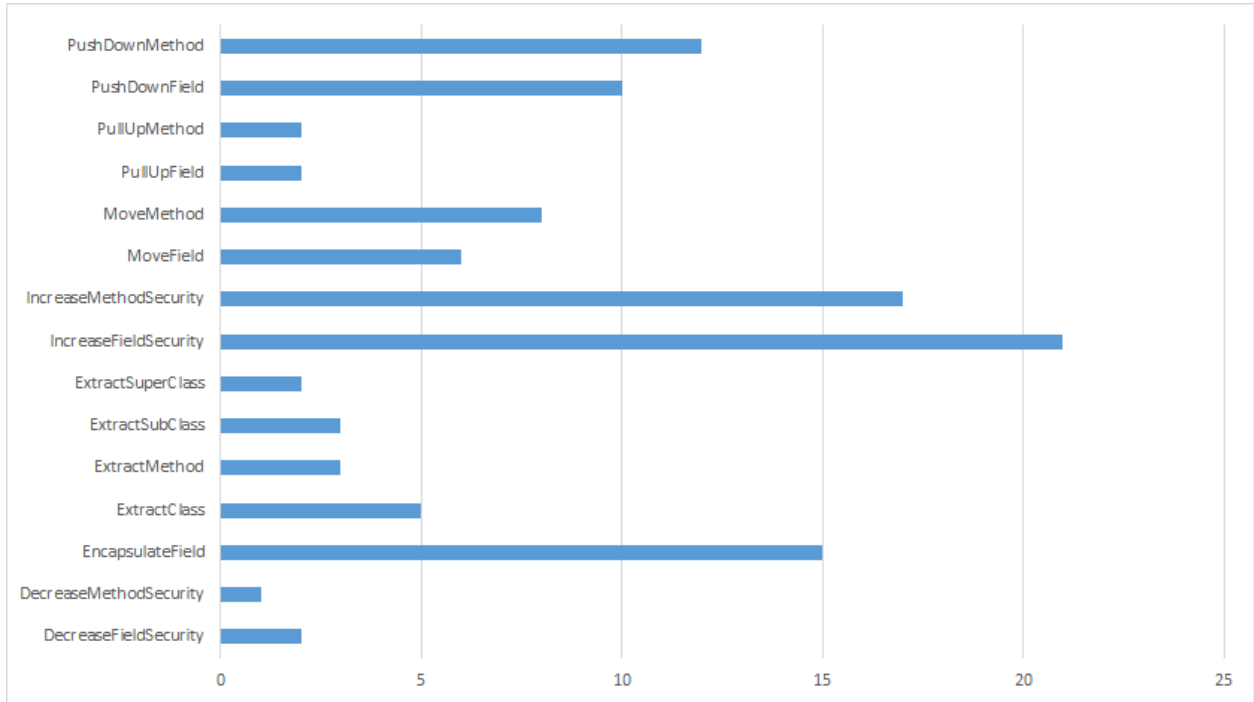
**Table 4.3:** The two most common refactoring patterns with the highest impact on the improvement of the average security measure for the 30 open source projects.

<b>Refactoring patterns</b>	<b>Average Security Improvement</b>
Encapsulate Field, Increase Field Security, Increase Method Security, Push Down Field, Move Method	0.42
Increase Field Security, Increase Method Security, Move Field, Push Down Method	0.34

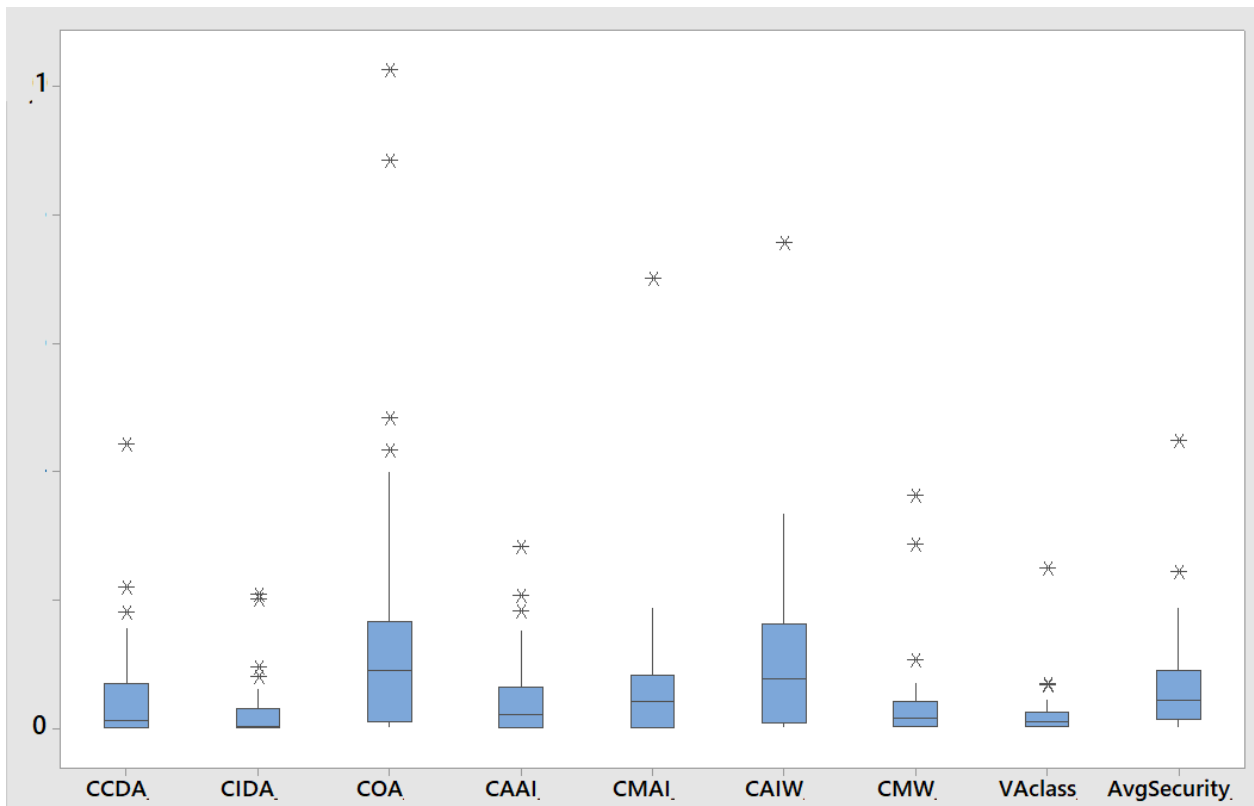
tively based on our analysis of the refactoring solutions proposed for 30 open source projects.

**Q Key findings:** Encapsulate Field, Increase Field Security, Push Down Field, Increase Method Security, Push Down Method are all positively correlated with the avg security metrics. Decrease Field Security, Pull Up Field, Decrease Method Security, Pull Up Method, Move Method, Extract Class, Extract Superclass, Extract method and Extract Subclass are all negatively correlated with the avg security metrics. There is no statistically significant correlation between the Move Field refactoring and the avg security metrics.

**Results for RQ2.** Table 4.4 confirms the conflicting nature between several of the quality attributes and most of the security metrics by analyzing the impact of the refactoring



**Figure 4.5:** Average distribution of the refactoring types among the solutions recommended for the 30 projects that significantly improve the security objective.



**Figure 4.6:** Impact of the recommended refactorings on security metrics based on the 30 projects.

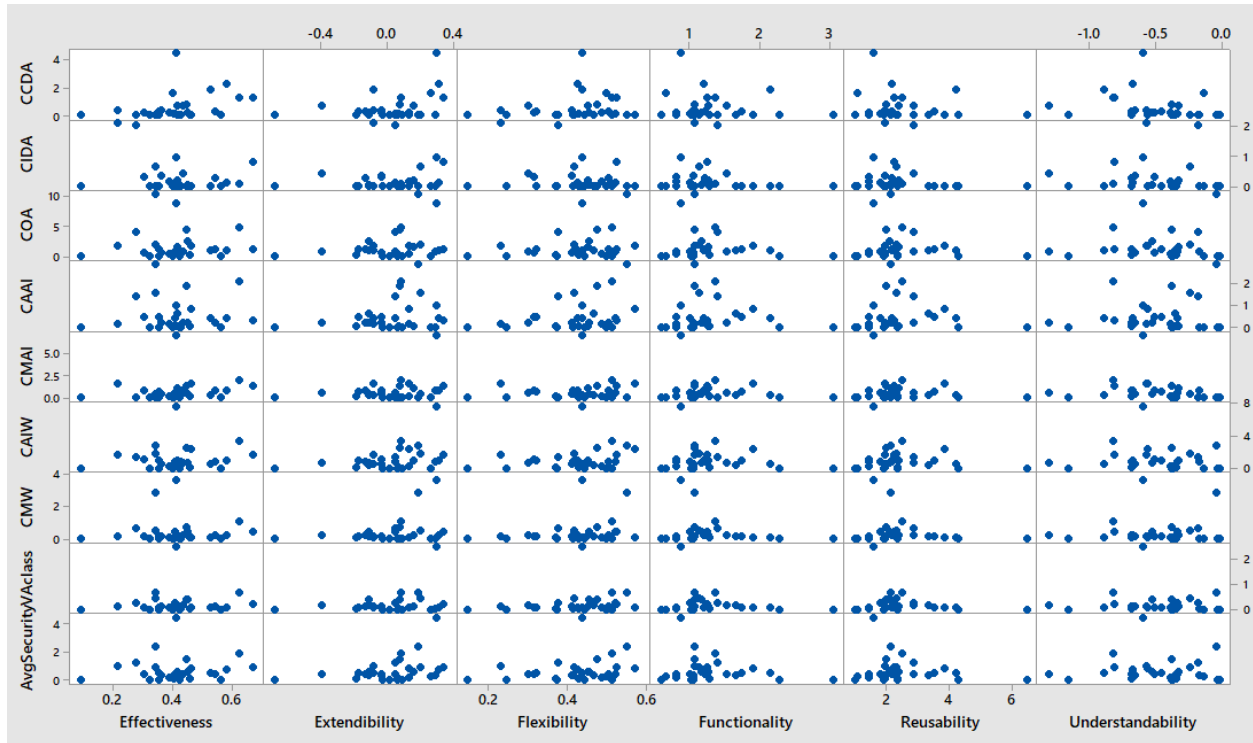
**Table 4.4:** Correlation results between the average of security metrics and quality attributes on the 30 projects. The results are statistically significant using the two-sample t-test at a 95% confidence level ( $\alpha = 5\%$ )

	<b>Understandability</b>	<b>Reusability</b>	<b>Functionality</b>	<b>Flexibility</b>	<b>Extendibility</b>	<b>Effectiveness</b>
CIDA	- (-0.237)	-- (-0.617)	- (-0.184)	- (-0.318)	- (-0.391)	+ (0.116)
CCDA	- (-0.224)	- (-0.382)	- (-0.137)	+ (0.281)	- (-0.232)	++ (0.589)
COA	+ (0.192)	- (-0.373)	- (-0.183)	+ (0.219)	-- (-0.619)	+ (0.314)
CMAI	- (-0.217)	- (-0.387)	- (-0.120)	+ (0.113)	- (-0.382)	+ (0.221)
CAAI	+ (0.114)	- (-0.234)	+ (0.131)	++ (0.612)	- (-0.224)	- (-0.122)
CAIW	- (-0.213)	- (-0.346)	- (-0.114)	+ (0.116)	-- (0.563)	+ (0.138)
CMW	+ (0.194)	- (-0.213)	- (-0.233)	+ (0.221)	- (0.241)	+ (0.187)
VA	- (-0.226)	- (-0.362)	- (-0.341)	+ (0.412)	- (-0.268)	+ (0.224)
AvgSecurity	- (-0.382)	-- (-0.731)	- (-0.114)	+ (0.183)	-- (-0.618)	+ (0.213)

solutions generated by our tool on the 30 open source projects. Four of the quality attributes were negatively correlated with the security metrics except Flexibility and Effectiveness. Reusability and Extendibility are negatively correlated with most of the security metrics which confirms the results of RQ1. In fact, these quality attributes can be improved using the extract super/sub class and pull-up method/field refactoring types that were already negatively correlated with security metrics.

Figure 4.7 presents more details related to the distribution of the refactoring solutions on the 30 open source projects based on each pair of quality and security metrics (all the metrics are to minimize based on our formulation). The distribution of the solutions is consistent with the correlation results reported in Table 4.4. For instance, the refactoring solutions with good reusability (low values) have the worst security impacts (high values) on the open source projects.

Since it is not enough to check the ability of our refactoring solutions to improve the quality and security objectives, we asked the 15 selected participants to evaluate the generated refactorings for 5 of the 30 open source projects using our tool (+Security) and an existing refactoring tool (-Security) [2]. The average manual correctness on the five systems is 86% for our approach compared to 73% for [2] (without the consideration of security objective) as described in Figure 4.8. Thus, it is clear that refactoring solutions addressing both quality and security issues were preferred compared to only improving the quality metrics. We presented the refactorings in a random way (not on the same code locations) to the participants and

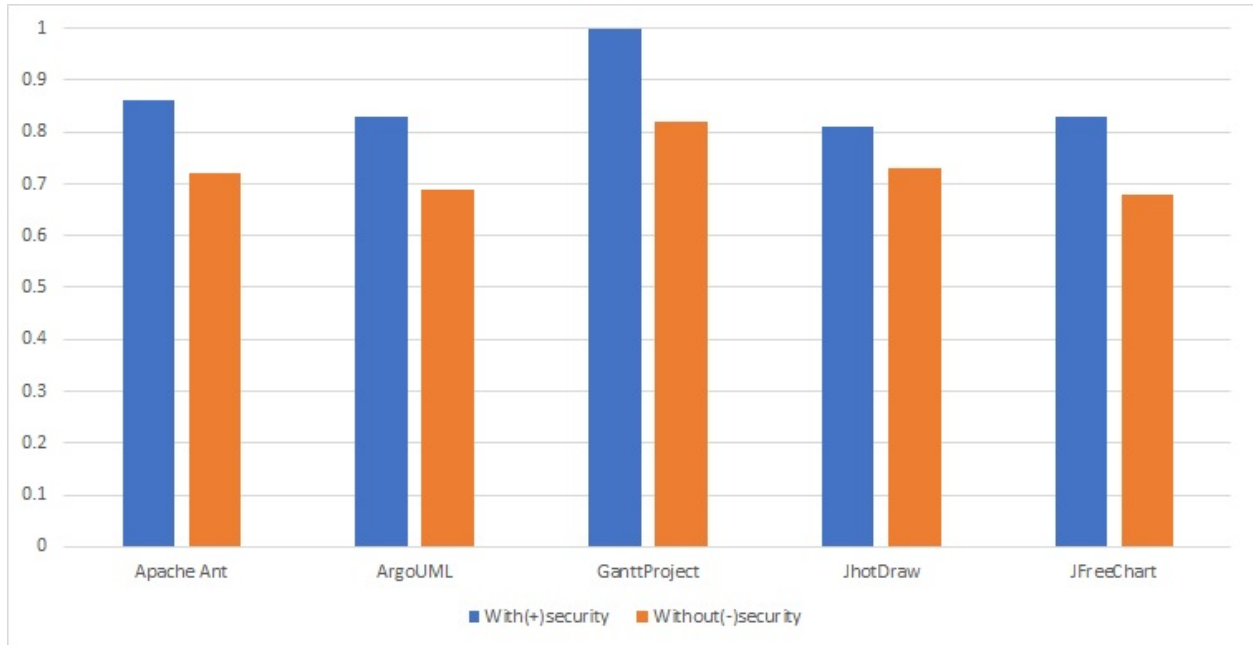


**Figure 4.7:** Distribution of refactoring solutions based on each pair of quality and security metrics for the 30 projects.

they were not aware of which tool is used to generate them. The refactorings recommended for the Gantt project were all considered relevant by the participants. The obtained results confirm that the combination of both quality and security objectives reasonably match the preferences of the participants.

**Key findings:** Understandability, Reusability, Functionality and Extendibility are all negatively correlated with the avg security metric. Flexibility and Effectiveness are positively correlated with the avg security metric. Reusability and Extendibility are negatively correlated with all of the eight security metrics.

**Results for RQ3.** Figure 4.9 summarizes the comparison of our tool with the work of Alizadeh et al. [2], not considering the security objective. The goal is to understand the sacrifice in quality when improving the security objective using the generated refactoring solutions. While Alizadeh et al.'s tool [2] improved the quality attributes more than our tool, the improvements are very similar to our security-aware approach for almost all the quality metrics. The major difference is for the extendibility measure, which is understandable based



**Figure 4.8:** Average manually determined correctness of the refactorings on different open source projects generated by our tool (+Security) and an existing refactoring tool (-Security) [2].

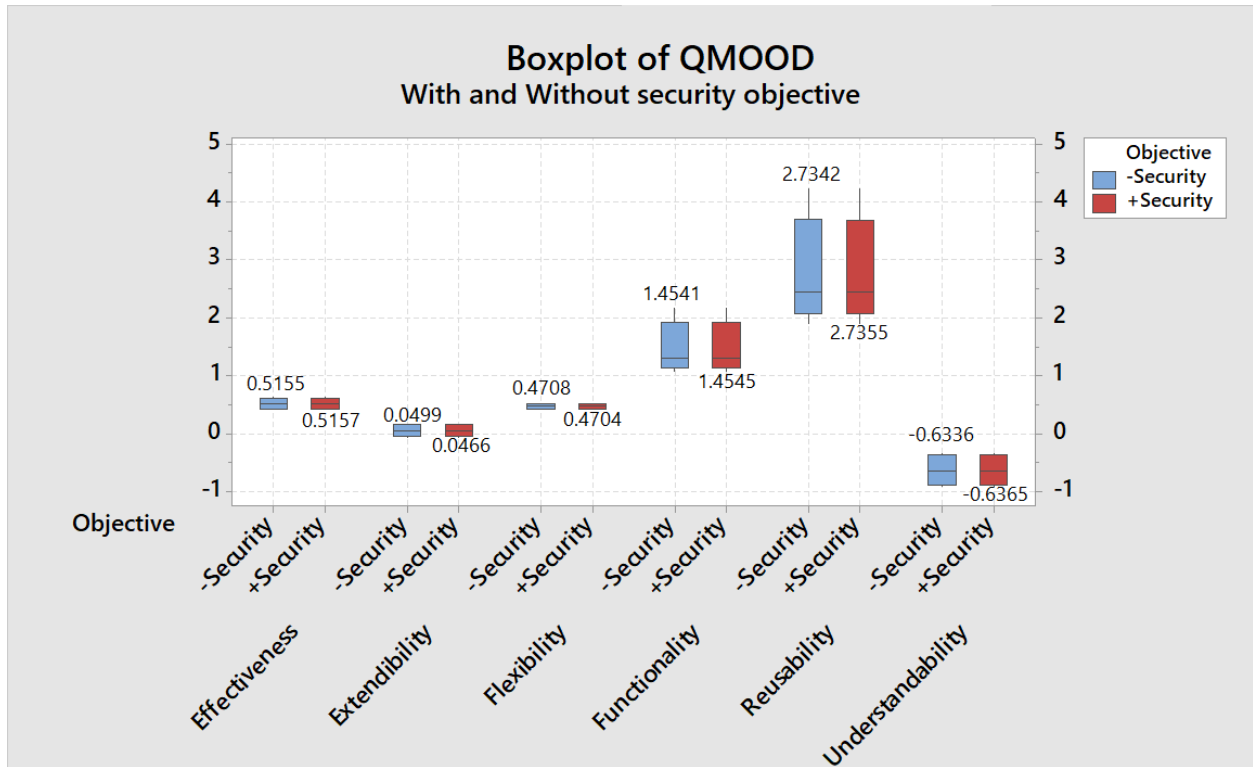
on the results of RQ1 and RQ2, and the difference is rather small.

Figure 4.10 shows that the multi-objective security-aware approach was able to generate a diverse set of refactoring solutions in terms of security improvements. The tool of Alizadeh et al. [2] was not able to generate any refactoring solution that can have a security objective value lower than 0.183 while our approach was able to improve better the security metric to reach lower than 0.175. While the deviation in terms of value may look small, the formulation of the security objective actually requires significant code changes to slightly improve security values.

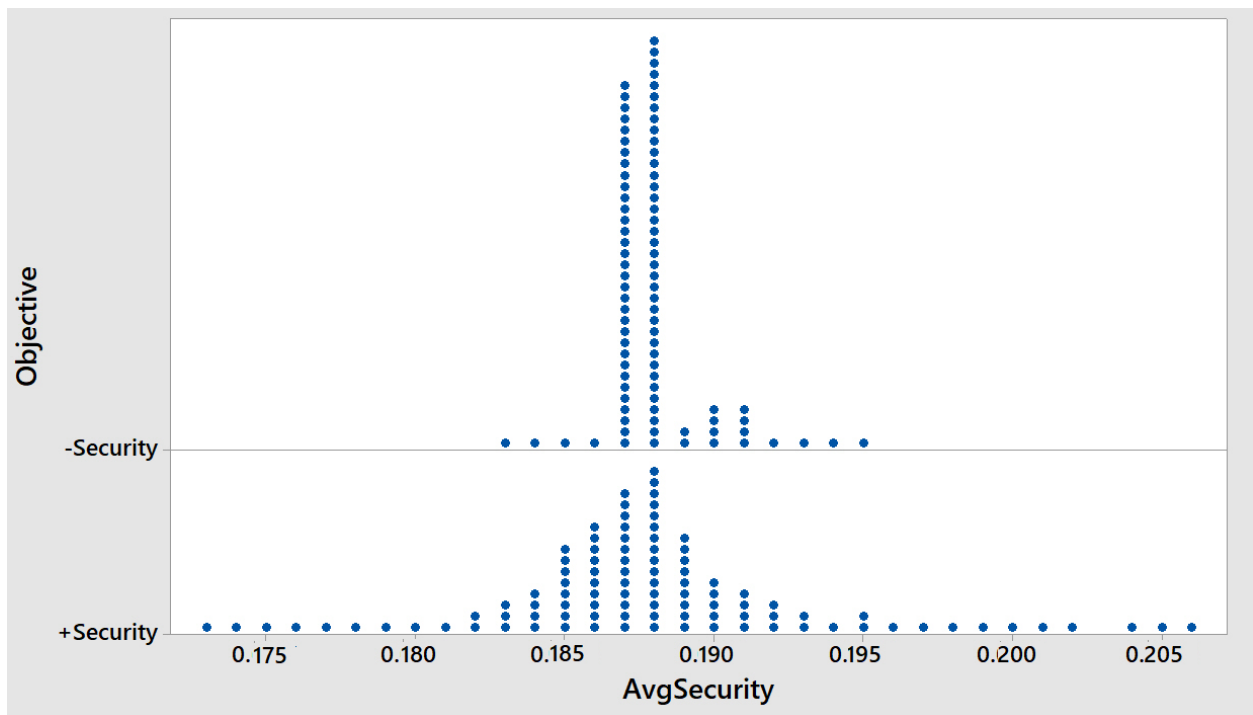
**Q Key findings:** The sacrifice, by our approach, in terms of quality improvements is very limited when enhancing code security comparing to an existing work only based on quality [2].

**Results for RQ4.** We asked participants to rate their agreement on a Likert scale from 1 (complete disagreement) to 5 (complete agreement) with the following questions:

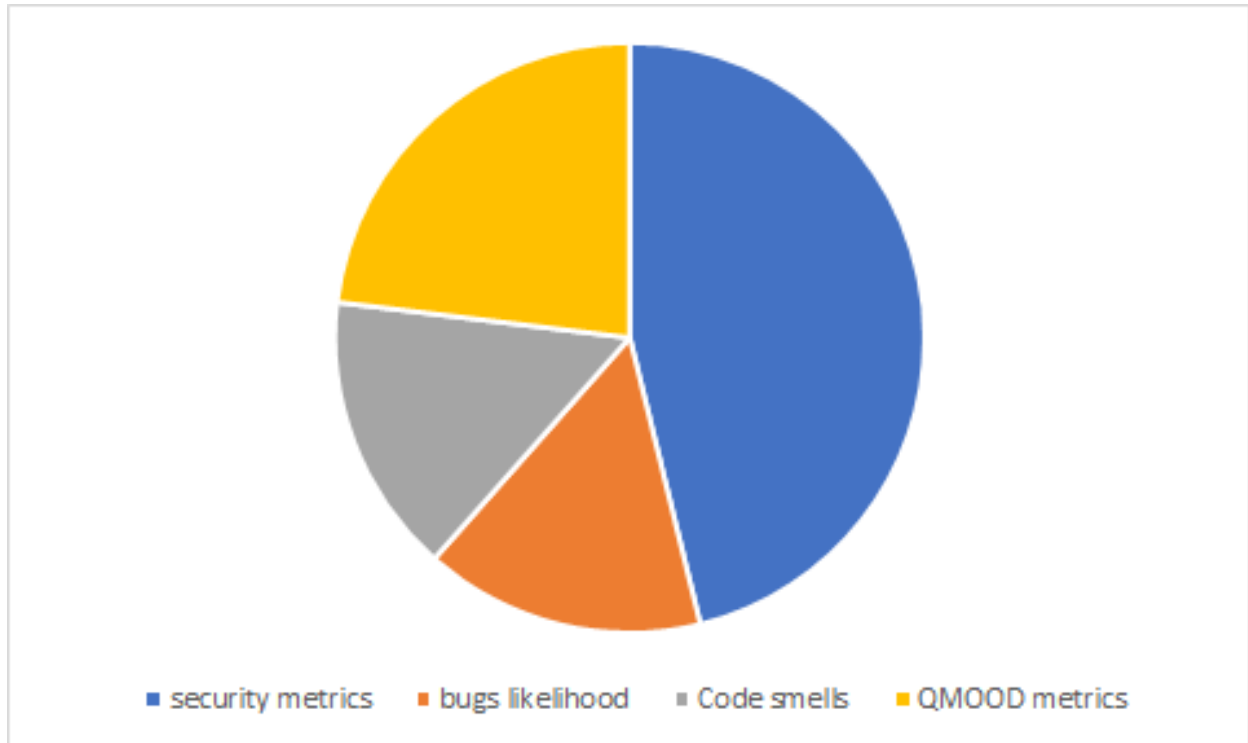
- The security-aware refactoring recommendations are a desirable feature in integrated development environments to improve code security while enhancing quality.



**Figure 4.9:** Box plots of the impact of refactoring solutions on the quality attributes based on 4 open source projects using our tool (+Security) and an existing refactoring tool (-Security) [2]. The results are statistically significant using the two-sample t-test at a 95% confidence level ( $\alpha = 5\%$ )



**Figure 4.10:** Distribution of the refactoring solutions using the security objective based on 4 open source projects comparing our tool (+Security) and an existing refactoring tool (-Security) [2].



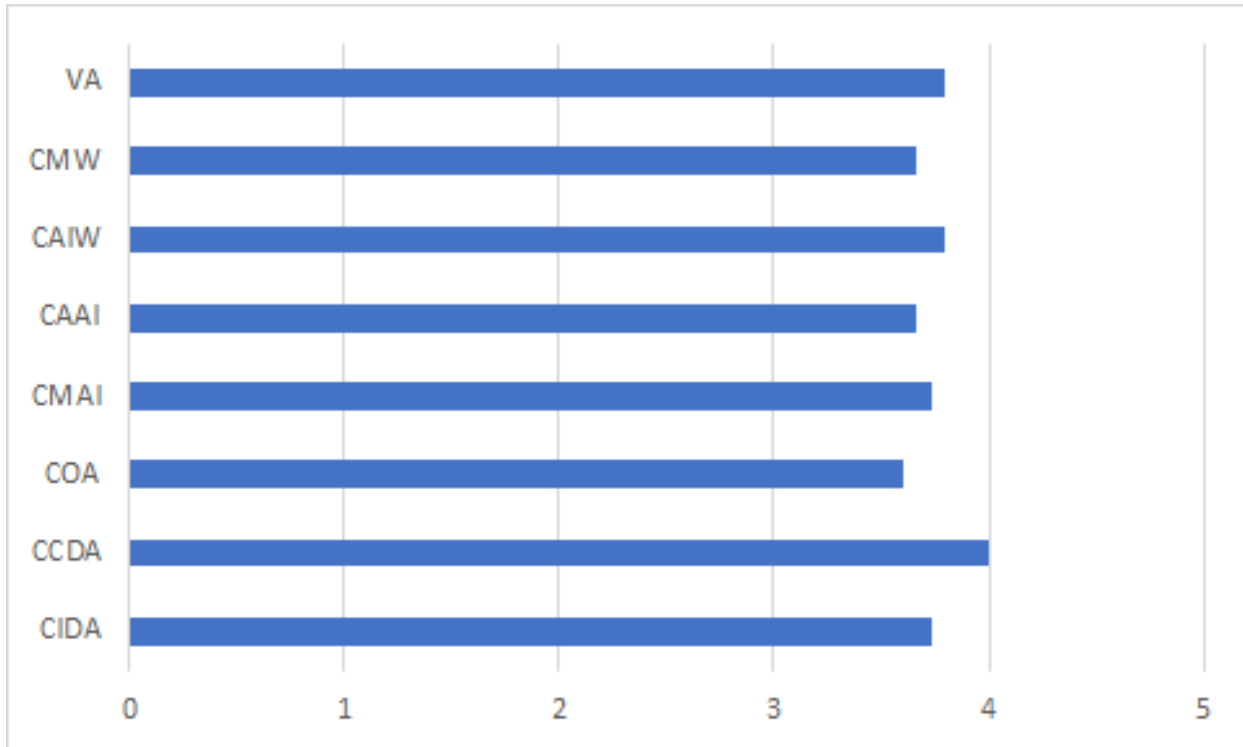
**Figure 4.11:** The important motivations for code refactoring by the participants.

- The security-aware refactoring web app is easy to use compared to fully-automated or manual refactoring tools that you used in the past.

The post-study questionnaire results show the average agreement of the participants was 3.96 and 4.12 based on a Likert scale for the first and second statements, respectively. This confirms both the relevance and usability of our security-aware tool to find a trade-off between code security and quality metrics. More details can be found in our appendix [3] showing the simple steps developers can follow to evaluate and fix both the quality and security issues of their projects.

We also asked the participants about the most important reasons to refactor their code. Figure 4.11 shows, surprisingly, that most of the participants considered security as *the most critical reason* for refactoring, even compared to improving quality metrics which is the second most important motivation for refactoring. Bug likelihood and code smells were also considered important by some participants. The outcomes of this question on why

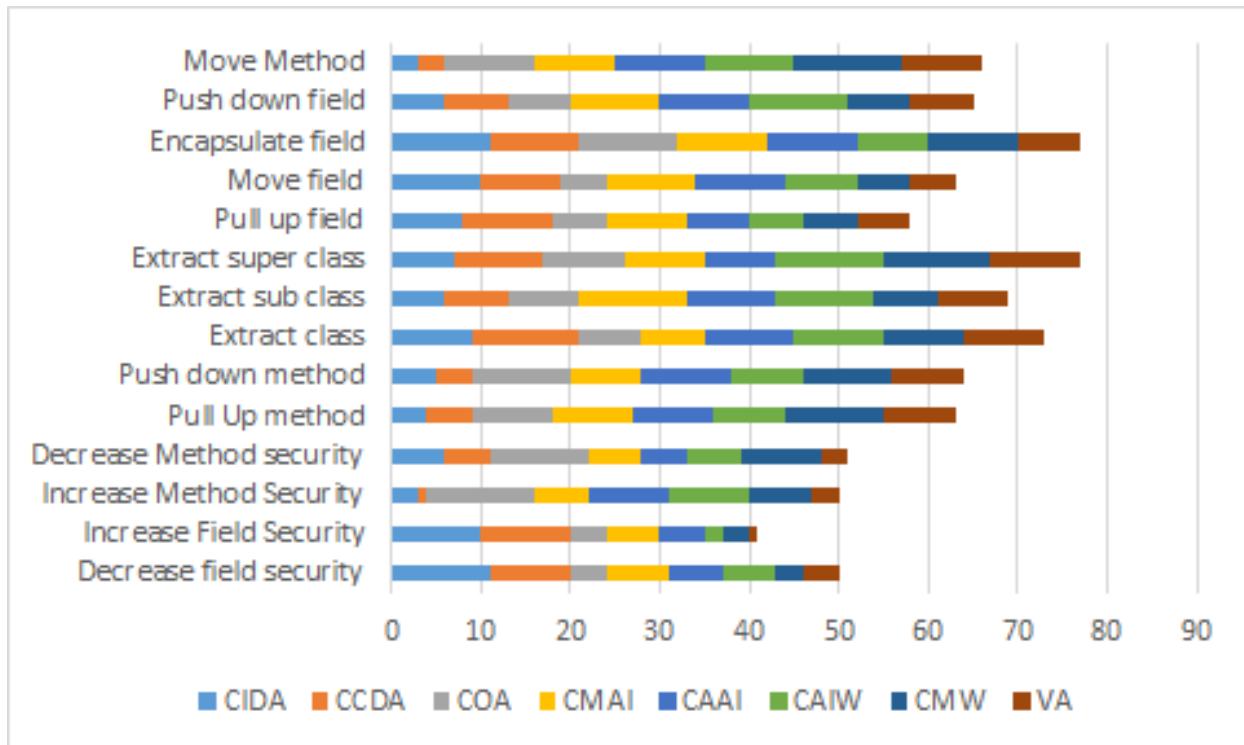




**Figure 4.12:** The potential impacts of refactoring on security metrics based on the survey.

to refactor the code are aligned with the motivations of this contribution advocating for considering both security and quality metrics when recommending refactorings.

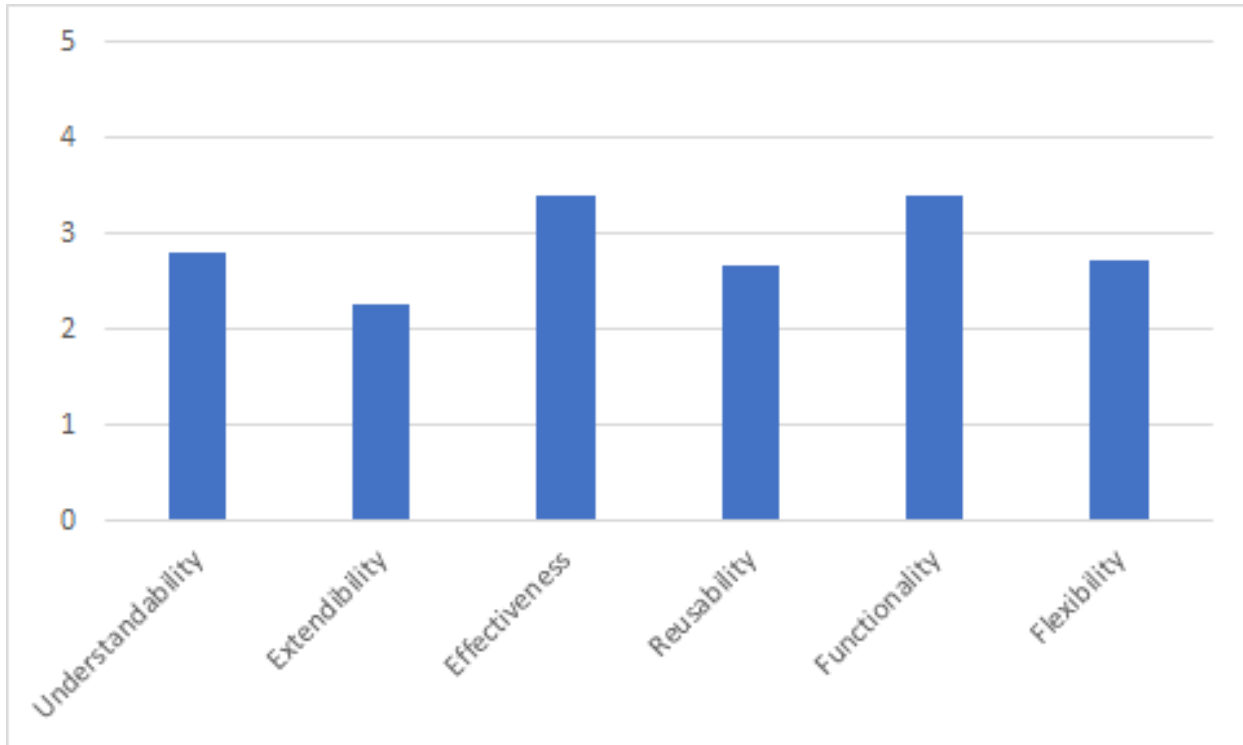
The next questions asked the respondents about the impact of refactoring on the code security metrics. Figure 4.12 shows that the developers think that refactoring can improve and positively impact most of the security metrics considered in our experiments. This confirms our selection of the security metrics and the outcomes of RQ1 obtained by analyzing the code. The developers think that the CCDA metric is the one that can be most improved by refactoring. The CCDA metric measures the direct access of classified class attributes of a class. It aims to protect the internal representations of a class, i.e. class attributes, from direct access. In fact, the accessibility of class attributes is one of the most critical entry points for security attacks to the architecture, and so the use of refactorings such as Increase Field Security can improve this metric. Figure 4.13 describes more detailed results on the possible impact of each refactoring type on the various static code security metrics. It is clear that Encapsulate field can have the most positive impact on several security metrics



**Figure 4.13:** The potential impact of different refactoring types on security metrics based on the survey.

based on the developers’ feedback. They also suggested that Extract Superclass will impact the security metrics, but in a negative way. The participants found as well that Push down method refactoring can improve several security metrics since it will reduce accessibility to the methods after refactoring. The results of these questions also confirm the results obtained in RQ1 about the impact of different refactoring types on security when we analyzed the code before and after refactoring.

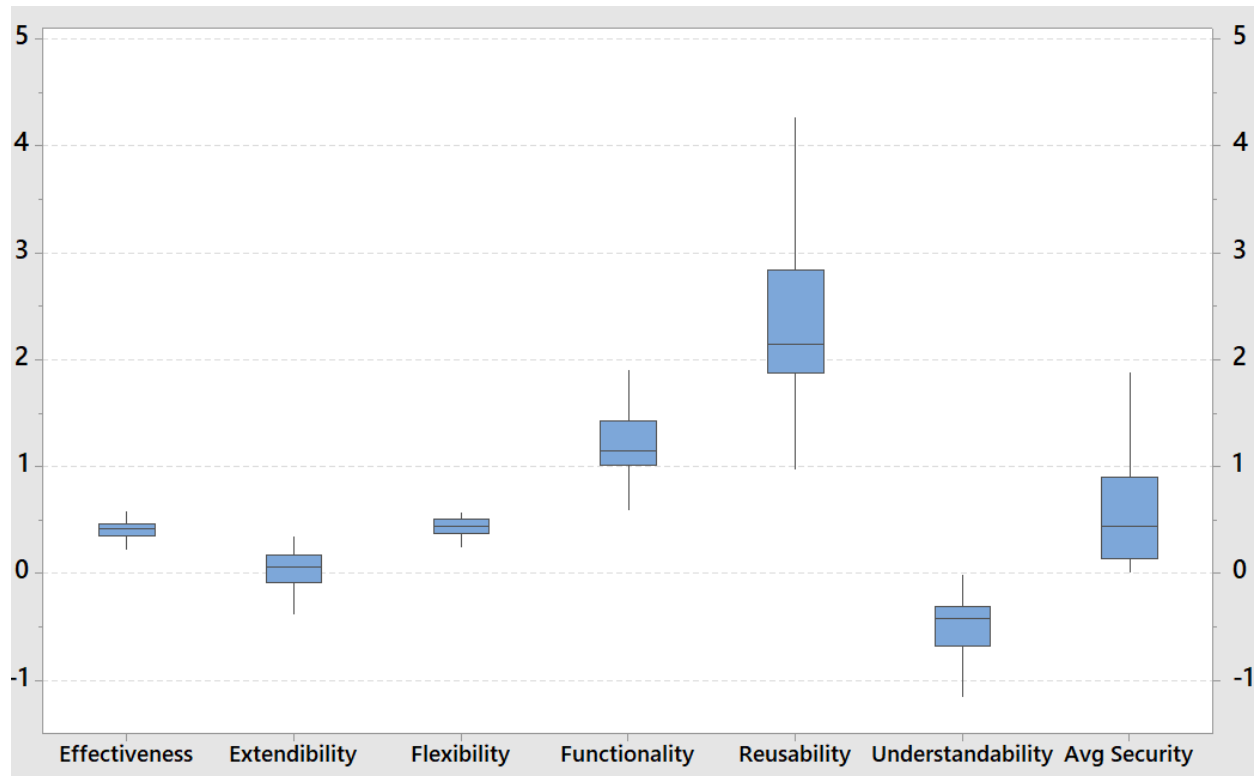
Figure 4.14 shows the opinion of developers on whether improving the security metrics will positively impact some quality attributes. The results show that effectiveness and functionally quality attributes can be improved if the refactorings improved security. The developers also suggested that improving security will have a negative impact on both understandability and extendibility since they have the least support from developers (around 2.7 out of 5). These outcomes are also partially consistent with the results found in RQ2 when we analyzed the correlation between the security metrics and quality attributes based on the code level information before and after refactoring.



**Figure 4.14:** The possible positive impact of improving the security metrics on quality attributes based on the survey.

Figure 4.15 shows that our approach based on multi-objective search can find good trade-offs between the various quality and security objectives. The box plots describe the diversity of the refactoring solutions generated by our multi-objective approach where the developer can find solutions that impact both quality and security at different levels. This aspect is important since a developer can select solutions that impact their specific quality or security objectives based on their preferences.

The impact of the refactorings on the different quality and security metrics is calculated based on the differences between their values before and applying the refactorings. Thus, we just measured the difference of these metric values before and after applying the refactorings to estimate the improvements. The box plots show that the generated refactorings can improve the majority of the quality and security objectives with varying levels of improvement, but sometimes it is possible to deteriorate (or sacrifice) some of the metric values/improvements due to their conflicting nature. However, Figure 4.15 shows that the multi-objective



**Figure 4.15:** Box plots of the impacts of refactoring solutions on both quality and security objectives based on the 30 projects.

algorithm was able to generate solutions improving the objectives at different levels with little deterioration. Thus, we conclude that the tool was successful in finding trade-offs rather than merely improving one or two specific objectives.

To summarize, the participants found the tool unique in terms of enabling them to understand the impact of refactoring on both security and quality. They highlighted that it is one of the first tools in their opinion that enables the identification of refactoring solutions to offer trade-offs between quality and security. The developers found the tool flexible as it provides multiple options to select a solution based on their preferences. A suggested improvement is to use visualization techniques to evaluate the impact of applying a refactoring sequence on the different security and quality metrics.

**🔍 Key findings:** The evaluation of our tool by 15 developers confirmed its efficiency in helping to understand the impact of refactoring on both security and quality and generating refactoring solutions that find a trade-offs between quality and security.

#### 4.1.5 Threats to Validity

The parameter tuning of the NSGA-II optimization algorithm used in our experiments is the first internal threat since these values were found by trial-and-error[585]. Since we used a limited number of evaluated systems and participants, the generalizability of our results is threatened. Besides, we only considered 14 refactoring types in our study. Furthermore, for the manual validation and comparison with an existing refactoring study, we used a selected subset of projects rather than the full 30 systems. Therefore, we estimate that a potential replication of our work is necessary to validate our results completely. We are also planning to consider more security and quality metrics to extend our empirical validation. The opinions of the practitioners involved in our study may be divergent when it comes to the recommended refactorings, and they might have different priorities for the security of the system which could have an impact on our results. Furthermore, our security metrics are limited to 8 measures thus we may need to include further metrics in our future studies and not only the easiest ones to implement.

Another potential threat is related to the identification of security sensitive attributes which can impact the calculation of the security metrics. To mitigate this threat, we manually validated the top 10 critical files and use their critical attributes (fields that have names that match one of the keywords from the list we gathered at the beginning) to identify the critical attributes in all the other files that will be used to compute the security metrics.

Finally, there is a possible threat due to experimenter bias in the surveys as the subjects had some prior contact with the researchers.

#### 4.1.6 Conclusion

We have presented an empirical study to validate the correlations between the QMOOD quality attributes [515] and a set of security metrics [466, 468] and to understand the correlations between refactoring types and security metrics. Based on the outcomes of these studies, we proposed a security-aware multi-objective refactoring approach to find a balance

between quality and security goals. We evaluated our tool on the same projects used for the empirical validations. Furthermore, we compared our results to an existing refactoring work not considering security to understand the sacrifice in security measures when improving the quality. The comparison shows that our security-aware approach performed significantly better than the existing approach when it comes to preserving and improving the security of the system but with low cost in terms of sacrificing quality. The survey with the 15 practitioners confirmed the efficiency of our tool and the importance of considering security while improving several quality attributes.

We are planning as part of our future work to expand the set of supported security metrics to include design-level metrics [476, 477, 586] as well, in a similar study. We are also planning to study the correlation between security metrics and the impact of improving one on the other. We are planning to expand our set of refactorings by those that can change the relationship between classes, such as Replace Inheritance with Delegation. It is an accepted principle in industry that a delegation relationship should be preferred to inheritance, particularly in the context of inversion of control containers such as Spring. Thus, we are planning to study the impact of these new types of refactoring on security and quality then check their acceptability by developers. Another research direction would be to generate refactoring recommendations that include third-party libraries [587, 588] in order to understand their impact on the security of JAVA apps. Finally, we are planning to perform a survey with developers to investigate the importance of considering security as a goal/motivation for refactoring.

## 4.2 Prioritizing Refactorings for Security Critical Code

### 4.2.1 Introduction

The National Institute of Standards and Technology (NIST) estimates that the US economy loses an average of \$60 billion per year as a cost of either implementing patches to fix security bugs and vulnerabilities or the actual impact of these security issues [589]. Vulnerability is defined as a property of system security requirements, design, implementation, or operation that could be accidentally or intentionally exploited to create a security failure [590]. These vulnerabilities heavily depend on the way how the system is designed and implemented. For instance, many software companies use third-party code and libraries [591] and many vulnerabilities are introduced through these external components [592]. Thus, it is critical to identify the security-critical code fragments when integrating new modules or to locate them in internally developed code to protect the system against possible attacks. Security-critical code refers to code fragments that contain data (e.g., attributes) and logic (e.g., methods) that can potentially be misused to violate security properties such as confidentiality, integrity, or availability of a system in production.

Several studies on the detection and fixing of vulnerabilities and security bugs [593, 594] show that poor quality indicators are one of the main sources of vulnerabilities, as also emphasized by CWE (CWE-398) [595]. However, existing refactoring research is mainly focused on improving quality attributes and fixing code smells [447, 496, 292, 596, 497]. For instance, a developer may create a hierarchy in a set of classes to improve the reusability quality attribute. However, these actions may expand the attack surface if the super class contains security-critical attributes and methods. Furthermore, the few existing studies on the prioritization of refactorings mainly focus on the identified quality issues but without considering security as one of the criteria, despite its importance and relevance in practice [597, 598, 110].

In this project, we used the history of vulnerabilities and security bug reports along with

a set keywords (defined in the literature [468, 472]) to automatically identify security-critical files in a project based on source code, bug reports, pull-request descriptions and commit messages. After identifying these security-related files, we estimated their risk based on static analysis to check their coupling with other components of the project. For instance, a highly coupled class which contains security-critical code fragments may contribute to compromising the whole system if an attacker takes advantage of the code to inject malicious payloads. Then, our approach recommends refactorings to prioritize fixing quality issues in these security-critical files to improve quality attributes and remove identified code smells. To find a trade-off between the quality issues and security-critical files, we adopted a multi-objective search [495] approach.

We evaluated our approach on six open source projects and one industrial system to check the relevance of our refactoring recommendations. The results confirm the effectiveness of our approach comparing to existing refactoring studies based on quality attributes and ranking the recommendations only based on their code smells and quality severity [7, 599]. Our survey with practitioners who used our tool supports our hypothesis that quality and security need to be considered together to provide relevant refactoring recommendations and to rank them.

#### **4.2.2 Motivations and Challenges**

Security-critical code fragments in a software project can represent code elements (e.g. classes, methods, files, etc.) containing confidential or sensitive information such as IDs, transactions, credit card data, authentication information, security constraints, etc. If these code fragments are over-exposed then they may result in vulnerabilities that may be exploited in violating security properties. Code fragments are frequently cited in security bug reports, vulnerability reports, or Stack Overflow posts which suggests that they are at the heart of many security problems. We will show, in our experiments, that we identified some heavily discussed libraries as origins of several vulnerabilities, and our approach proposed refactoring



these libraries. And often vulnerable code fragments, identified during code reviews, are analyzed to ensure that they are carefully designed so as to reduce the attack surface in case of potential attacks in the future.

The identification of security-relevant code in a software project is critical (1) for designers to be careful when they are designing or maintaining a system. For instance, they have to make sure that coupling is low in these security-related fragments to reduce the attack surface; (2) for developers to ensure that these code fragments are not over-exposed; (3) for reviewers to pay a lot of attention when reviewing these files; and (4) for the organization to evaluate the use of third-party code from a security perspective before any adoption or integration work. However, most existing research tools for refactoring recommendation and prioritization [447, 496, 292, 596, 497] do not consider the security aspect but focus more on general quality improvements and removal of code smells when ranking and recommending refactorings.

Nowadays, maintaining both quality and security of software systems is not optional. Many contemporary applications are cloud-based and therefore potentially exposed to malicious attacks. Developers are under increasing pressure to deliver clean and reliable software systems that generate the intended outputs while making sure that sensitive customers data is secure.

One of the main challenges when integrating both code quality and security concerns into a single refactoring tool is that they may be conflicting. For example, improving the reusability of the code may increase the attack surface due to newly created abstractions. Also, increasing the spread of classes that contain sensitive information in the design to improve modularity may reduce the resilience of the system to attacks.

The Common Vulnerabilities and Exposures (CVE) database is a large, publicly available source of vulnerability reports [600]. It aims to provide common names for publicly known problems. As described in Figure 4.16, one of the main CVE categories is "Indicator of Poor Code Quality" (CWE-398) providing additional evidence that code quality issues are

CWE - 398 : Indicator of Poor Code Quality	
CWE Definition	<a href="http://cwe.mitre.org/data/definitions/398.html">http://cwe.mitre.org/data/definitions/398.html</a>
Number of vulnerabilities:	1
Description	The code has features that do not directly introduce a weakness or vulnerability, but indicate that the product has not been carefully developed or maintained. Programs are more likely to be secure when good development practices are followed. If a program is complex, difficult to maintain, not portable, or shows evidence of neglect, then there is a higher likelihood that weaknesses are buried in the code.

**Figure 4.16:** A category in the CVE security bug database [3] that includes security vulnerabilities related to poor code quality

frequently responsible for security issues. The description of this category highlights that when the code is complex and not well-maintained it is more likely to cause security problems and weaknesses.

Figure 4.17 shows an example of one detected vulnerability in the CWE-398 category on the NUUO Intelligent Surveillance software system [4] due to the use of multiple outdated software components that needed to be refactored. Whenever developers introduced changes to the system, they faced challenges to make those changes consistently across classes; thus introducing refactoring to fix this quality issue was critical. This vulnerability had a score of 7.5 which is considered to be high and urgent to fix.

To overcome such challenges, in the next section we propose an approach to prioritize and recommend refactorings to target the classes that have both quality and security issues. If successful, this will enable developers to spend less effort on refactoring non-critical issues and make systems more secure while maintaining high code quality.

### 4.2.3 Approach

The structure of our approach is sketched in Figure 4.18. The first component consists of identifying a project's security-critical files, to evaluate and refactor, based on a list of keywords, along with the history of security bugs and vulnerabilities detected in the analyzed system. The list of keywords are the most common security-related words that developers may use in naming code elements, writing comments, security bugs and vulnerabilities reports, commits messages, and security questions/tags on Stack Overflow <sup>2</sup>. The full list of

<sup>2</sup><https://stackoverflow.com/>

**Vulnerability Details : [CVE-2018-17890](#)**

NUUO CMS all versions 3.1 and prior, The application uses insecure and outdated software components for functionality, which could allow arbitrary code execution.  
 Publish Date : 2018-10-12 Last Update Date : 2019-10-09

[Collapse All](#) [Expand All](#) [Select](#) [Select&Copy](#) [▼ Scroll To](#) [▼ Comments](#) [▼ External Links](#)  
[Search Twitter](#) [Search YouTube](#) [Search Google](#)

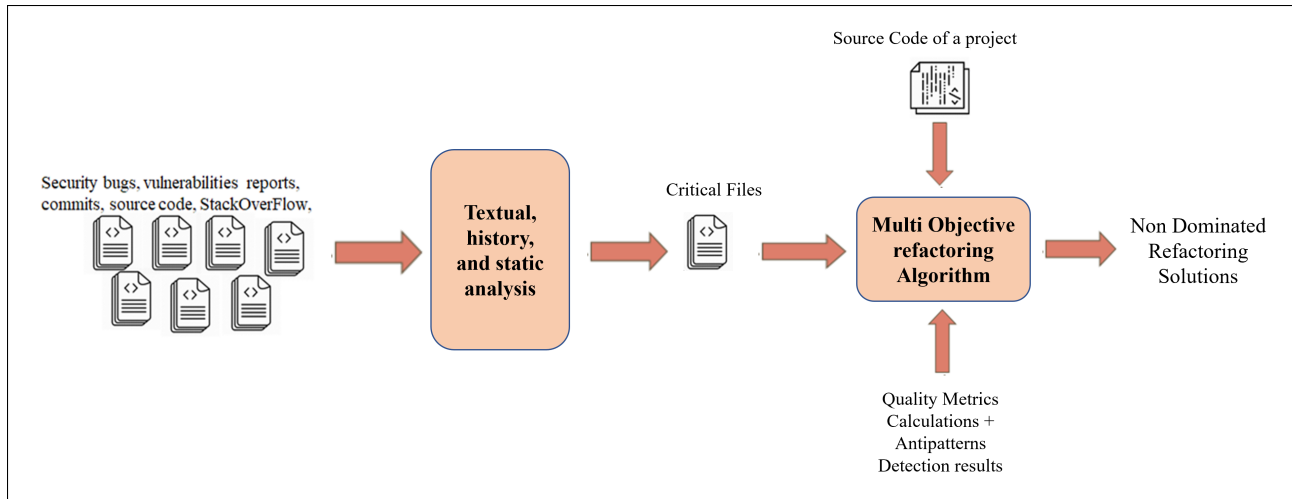
**- CVSS Scores & Vulnerability Types**

CVSS Score	<b>7.5</b>
Confidentiality Impact	Partial (There is considerable informational disclosure.)
Integrity Impact	Partial (Modification of some system files or information is possible, but the attacker does not have control over what can be modified, or the scope of what the attacker can affect is limited.)
Availability Impact	Partial (There is reduced performance or interruptions in resource availability.)
Access Complexity	Low (Specialized access conditions or extenuating circumstances do not exist. Very little knowledge or skill is required to exploit. )
Authentication	Not required (Authentication is not required to exploit the vulnerability.)
Gained Access	None
Vulnerability Type(s)	Execute Code
CWE ID	<a href="#">398</a>

**Figure 4.17:** An example of a security vulnerability from NUUO CMS system due to code quality issues [4].

keywords used in our approach can be found in Figure 4.19. Figure 4.20 shows an example of security-critical code in Apache Tomcat identified automatically by our approach. We have also implemented a parser that can find all the files involved in previous security bug, vulnerability reports and pull-requests with security tags.

After identifying the list of security-critical files, we used a multi-objective genetic algorithm, based on NSGA-II [495], to generate refactoring solutions that prioritize and fix the files associated with quality and security issues. The quality objectives are based on code smells detected using a set of rules [601] and the potential improvements in QMOOD quality measures [480] defined in table 2.10. We considered code smells and QMOOD as separate objectives since developers may want to understand the impact of fixing the code smells on the quality attributes to reason about the relevance of recommended refactorings. The second objective estimates the importance of the refactored security-critical files based on a combination of textual, history and static analysis measures. The textual analysis is based on matching scores between the keywords and the source code files (e.g. names of code elements, comments, etc.). The static analysis calculates a class’s coupling score with other classes in the project: the most severe security-critical code fragments are the ones that are highly coupled. The third measure counts the number of occurrences of the security-critical



**Figure 4.18:** Security-Critical Code Identification: Approach Overview

files in previous security bugs, vulnerability reports, and pull-requests with security tags to evaluate and refactor. Thus, the second objective will favor refactoring solutions targeting important security-critical files.

In the next sub-sections, we give details about each of these two major components.

#### 4.2.3.1 Security-critical File Detection

To detect security-critical files, we combined three different measures of textual, static and history analyses. All three measures are normalized to values between 0 and 1. We then calculated their average score to rank the security-critical files.

With the chosen set of keywords, we calculated a textual security-criticality-score for each file to estimate the extent to which the file is related to security concerns and hence needs to be protected. The higher the score is the more likely the file is security-critical. We compute the textual security-criticality-score based on cosine similarity between each file and the set of keywords. Let  $n$  be the number of files in the source code and  $W$  an array containing the set of keywords. After pre-processing the source code including tokenization, lemmatization, stop words filtering and punctuation removal, we calculate the tf-idf score considering the file  $f_i, i \in \{1, 2, \dots, n, \}$  and  $W$  as corpus. Cosine similarity is then calculated as follows:

Keywords				
id	private	lock	code	protect
userid	privacy	algorithm	permission	securitymanagement
uuid	secure	salt	access	security constraint
password	credential	nonce	token	auth constraint
pwd	undercover	host	certificate	
username	crypted	port	cover	
account	hashed	backdoor	job	
creditcard	top secret	digital certificate	payment	
phonenumber	restricted	biometrics	transaction	
socialsecuritynumber	hidden	safe	ip-address	
dateofbirth	encrypt	confidentiality	transcoded	
secret	personal	sensitive	restricted access	
confidential	address	admin	sensitive information	
classified	cached	access	sensitive data	
login	security	administrator	card	
identifier	encoded	auth	credit	
unique	connectionString	authenticate	email	
name	path	credentials	content secure	
critical	signature	credit card number	user details	
vulnerable	role	encrypted	private field	
authenticator	hostname	hash	private member	
key	covered	undercovered	secret key	
			client id	
			hidden field	

**Figure 4.19:** List of keywords used in our approach

$$sim(f_i, W) = tf - idf(f_i, W) * transpose(tf - idf(f_i, W)) \quad (4.3)$$

A file with security-critical code may spread its vulnerability to its connected files in the system. Therefore, we parse the source code and compute a coupling metric as a second measure for all classes in each file. The coupling metric for each file is then equal to the average of the coupling metrics of all of its classes.

We define coupling of a class as the number of Call-Ins Call-Outs from that class [8]. Let  $m$  be the number of classes in file  $f_i$ ,  $C = \{c_1, c_2, \dots, c_m\}$  the set of classes in  $f_i$  and  $cp_j$  the coupling metric for class  $c_j$ , the coupling metric for file  $f_i$  is :

$$cp_i = \frac{\sum_{j=1}^m cp_j}{m} \quad (4.4)$$

which is the average of the coupling of the classes contained in file  $f_i$ .

```

package org.asynchttpclient.oauth;

import org.asynchttpclient.util.Utf8UrlEncoder;

/**
 * Value class used for OAuth tokens (request secret access secret);
 * simple container with two parts, public id part ("key") and
 * confidential ("secret") part.
 */
public class RequestToken {
    private final String key;
    private final String secret;
    private final String percentEncodedKey;

    public RequestToken(String key, String token) {
        this.key = key;
        this.secret = token;
        this.percentEncodedKey = Utf8UrlEncoder.percentEncodeQueryElement(key);
    }

    public String getKey() {
        return key;
    }

    public String getSecret() {
        return secret;
    }

    public String getPercentEncodedKey() {
        return percentEncodedKey;
    }
}

/**
 * An interceptor that adds the request header needed to use HTTP basic authentication.
 */
public class BasicAuthRequestInterceptor implements RequestInterceptor {
    private final String headerValue;

    /**
     * Creates an interceptor that authenticates all requests with the specified username and password
     * encoded using ISO-8859-1.
     * @param username the username to use for authentication
     * @param password the password to use for authentication
     */
    public BasicAuthRequestInterceptor(String username, String password) {
        this(username, password, ISO_8859_1);
    }

    /**
     * Creates an interceptor that authenticates all requests with the specified username and password
     * encoded using the specified charset.
     * @param username the username to use for authentication
     * @param password the password to use for authentication
     * @param charset the charset to use when encoding the credentials
     */
    public BasicAuthRequestInterceptor(String username, String password, Charset charset) {
        checkNotNull(username, "username");
        checkNotNull(password, "password");
        this.headerValue = "Basic " + base64Encode(username + ":" + password.getBytes(charset));
    }
}

```

Figure 4.20: An example of a security-critical code fragments identified by our approach

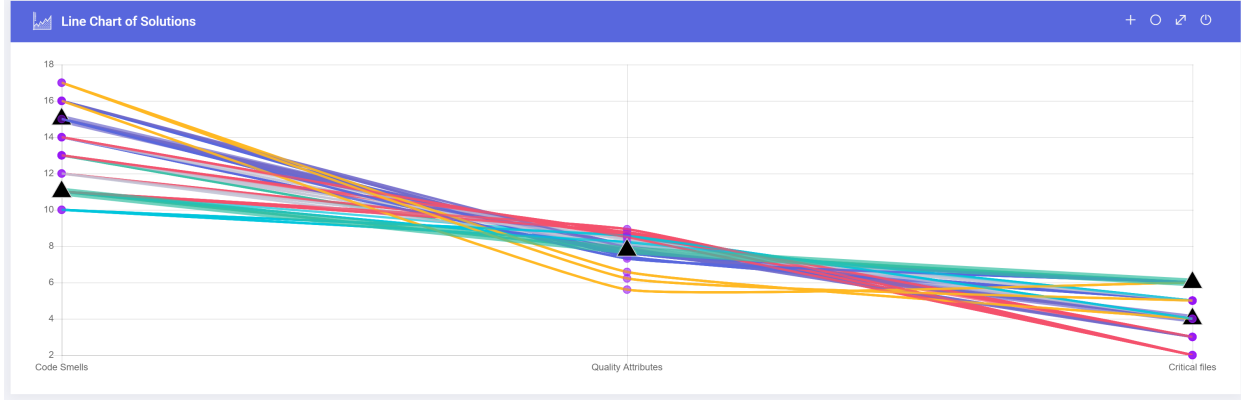
The third history-based measure simply counts the number of occurrences of the source code files in previous security bugs, vulnerability reports and pull-requests of code reviews with security tags.

Thus, if a file with high cosine similarity score is highly coupled and appeared in previous security bugs or vulnerability reports, then it is considered as *critical* and should be refactored if the confidential data it contains is accessible or could be compromised. If a file with high cosine similarity score is not highly coupled and was not vulnerable before, then it is not urgent to be refactored. The average score of all the three measures reflect these intuitions.

#### 4.2.3.2 Refactoring Prioritization for Identified Security-critical Code

We adapted a multi-objective search algorithm, based on NSGA-II [495], to optimize three objectives that take into account both the security and the quality of the software system. We chose this algorithm because it was used before in similar software engineering problems [30, 447, 448] and was proven to be able to balance independent or even conflicting objectives.

The algorithm is executed to find a set of non-dominated solutions balancing the objectives of security, code smells and quality attributes. With our multi-objective tool, the developer does not have to assign weights to the objectives. The user can select a solution



**Figure 4.21:** An example of a Pareto front of refactoring solutions generated by our tool for OpenCSV project.

from the Pareto-front of non-dominated solutions based on his needs and priorities as shown in Figure 4.21. The developer can also interact with our tool and give feedback by accepting or rejecting the refactoring recommendations.

Our approach takes into consideration 3 objectives: the first one is the sum of the relative changes of the 6 QMOOD attributes after applying a refactoring solution. This objective can be written as follow:

$$\sum_{n=1}^6 \frac{Q_i^{after} - Q_i^{before}}{Q_i^{before}} \quad (4.5)$$

where  $Q_i^{before}$  and  $Q_i^{after}$  are the values of the  $QualityAttribute_i$  before and after applying a refactoring solution, respectively.

Most code anti-patterns can be detected using interface and code quality metrics. In our study, we used an existing antipattern detection tool based on rules [601] that can detect 11 types of antipatterns defined in Section 2.4.2.2. We have chosen this tool because of its high accuracy. Using this measure we have defined the second fitness function as the value of the

anti-patterns "fixed" by the refactoring solution. This objective can be written as follow:

$$\sum_{i=1}^{FS} antipatterns_i \quad (4.6)$$

Where FS is the total number of files in the system and  $antipatterns_i$  is the number of fixed antipatterns in the file i by the refactorings solution.

In the third fitness function, we maximize the number of critical files to refactor. This objective can be written as follows:

$$\sum_{i=1}^F Severity_i \quad (4.7)$$

Where F is the total number of selected critical files and  $severity_i$  is the severity score of file i selected for refactoring. This severity score is the average of the three textual, history and static measures described previously.

#### 4.2.4 Experiment and Results

In this section, we first present our research questions and validation methodology followed by our experimental setup and our results.

##### 4.2.4.1 Research Questions

We defined three main research questions to measure the relevance and benefits of our approach comparing to the state of the art [7, 599] based on several practical scenarios. It is important to evaluate, first, the manual correctness of the recommended refactorings. Since it is not sufficient to make correct recommendations, we evaluated the ranking of the of these refactorings in terms of importance to developers. In practice, they are not interested to check and apply *all* the correct refactorings due to limited resources but they focus on the most important ones before the release deadline. We have also used post-study questionnaires to evaluate the benefits of our approach and the relevance of our results.



The three research questions are as follows:

- **RQ1: relevance and comparison to existing refactoring techniques.** To what extent are the refactorings recommended by our approach relevant, compared to existing refactoring techniques based on improving quality measures [7, 599]?
- **RQ2: Ranking evaluation.** To what extent can our approach **efficiently rank** recommended refactorings compared to existing techniques [7, 599] ?
- **RQ3: Insights.** How do programmers evaluate the **usefulness** of our approach?

To answer RQ1, we validated our approach on six medium to large-size open-source systems and one industrial project to manually evaluate the relevance of the recommended refactorings based on both quality and security. To this end, we used the **Manual Correctness (MC@k)** precision metric. MC@k denotes the number of correct refactorings in the top k recommended refactorings by the solution divided by k. It is unrealistic to calculate the recall since it requires the inspection of the entire system. We further address RQ1 by interviewing the participants who analyzed the output of our approach on the industrial project, who are among the original developers of that system (as detailed in the next section).

We asked a group of 32 participants to manually evaluate the relevance of the top k refactorings that they selected using the different tools. We compared our approach to two fully-automated refactoring tools: Ouni et al. [7] and JDeodorant [599]. Ouni et al. [7] proposed a multi-objective refactoring formulation based on NSGA-II that generates a solution to maximize treatment of several quality attributes and antipatterns. JDeodorant [599] is an Eclipse plugin to detect antipatterns and recommended refactorings based on a set of templates. As JDeodorant supports a lower number of refactoring types with respect to the ones considered by our tool, we restrict our comparison with it to just these refactorings. Furthermore, we implemented a sanity check approach where we used our multi-objective algorithm with only the quality and antipatterns objectives and then ranked the recommended refactorings based on the security severity measure. Thus, we can evaluate the

benefits of considering maximizing the refactoring of security-critical files with quality issues as a separate objective rather than using that function to rank the recommended refactorings. Finally, we compared our work with a mono-objective genetic algorithm combining all the three objectives into one function with equal weights so we can evaluate whether the various objectives are conflicting.

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

We preferred not to use the antipatterns and internal quality indicators as proxies for estimating the refactoring relevance since the developers' manual evaluation already includes a review of the impact of suggested changes on quality. We also wanted to avoid any bias in our experiments since antipatterns and quality attributes are considered in the fitness functions of our approach. Furthermore, not all the refactorings that improve a quality attribute are relevant to the developers. The only fair way to evaluate the relevance of our tool is thus manual evaluation of the results by active developers.

To answer RQ2, we evaluated the ranking of the refactorings by asking the participants to manually rate their importance: high, medium, or low. Then, the evaluation metric **importance@k** calculates the number of refactorings rated “high” in the top k, divided by k. Of course, this measure is applied in the order of refactorings generated by the various approaches.

To answer RQ3, we used a post-study questionnaire that collected the opinions of developers on our tool and the relevance of refactoring security-critical files on software projects.

#### 4.2.4.2 Software Projects and Experimental Setting

**4.2.4.2.1 Studied Projects** We used a set of well-known open-source and one system from our industrial partner, a software company with a focus on e-commerce and web development. We applied our approach to six open-source Java projects: tink, pac4j, atomix, securitybuilder, rest.li and firefly. Tink provides a simple and misuse-proof API for common cryptographic tasks. Pac4j is a security engine system. Atomix is an event-driven framework for coordinating fault-tolerant distributed systems. Securitybuilder is a fluent builder API for JCA and JSSE classes and especially X.509 certificates. Rest.li is a framework for building scalable RESTful architectures. Firefly is an asynchronous framework for rapid development of high-performance web application. Among the 6 systems, there are only 2 security projects that we selected intentionally to check if our approach can propose similar results to non-security projects.

To get feedback from the original developers of a system, we ran our experiment on a large industrial project, called DAS, provided by our industrial partner. The analyzed project can collect, analyze and synthesize a variety of data and sources related to online customers such as their shopping behavior. It was implemented over a period of 9 years, frequently changed over time, and had experienced several vulnerabilities.

We selected these systems for our validation because they range from medium to large-sized and have been actively developed over several years, they are widely used by companies as third party code and several previous vulnerabilities were detected on them. The data collected on these systems included the history of bug reports, vulnerability reports and pull-requests to identify ones with security tags. Table 4.5 provides some demographic data on these systems.

**4.2.4.2.2 Subjects** Our study involved 30 graduate students and 2 software developers from the industrial partner. Participants included 24 Master’s students in Software Engineering, 6 Ph.D. students in Software Engineering and 2 software developers. All participants

**Table 4.5:** Demographics of the studied projects.

System	Release	#Classes	KLOC	GitHub Link
tink	v1.2.2	590	185	google/tink.git
pac4j	v3.6.1	975	67	pac4j/pac4j.git
atomix	v3.0.11	2719	188	atomix/atomix.git
securitybuilder	v1.0.0	313	81	tersesystems/securitybuilder.git
rest.li	v15.0.3	4185	478	linkedin/rest.li.git
firefly	v4.9.5	2188	154	hypercube1024/firefly.git
DAS	v7.6.1	973	326	N.A.

were volunteers who were familiar with software security, refactoring, Java and quality assurance. All the Master’s students were working full-time in industry as developers, managers, or architects. They average 6 years of experience in industry and 16 out of the 24 have worked on either fixing security bugs or patching vulnerabilities.

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 the company, their programming experience, and their familiarity with software refactoring and security. Although the vast majority of participants were already familiar with refactoring, all the participants attended one lecture of two hours on software refactoring by the organizers of the experiments. The details of the selected participants can be found in Table 4.6, including their programming experience (years) and level of familiarity with refactoring. Each participant was asked to assess the meaningfulness of the refactorings recommended after using one of the five tools on one system to avoid a training threat. The participants did not only evaluate the suggested refactorings but were asked to configure, run and interact with the tools on the different systems. The only exceptions were related to the two participants from the industrial partner where they agreed to evaluate only their industrial software. We assigned tasks to participants according to the studied systems, the techniques to be tested and developers’ experience.

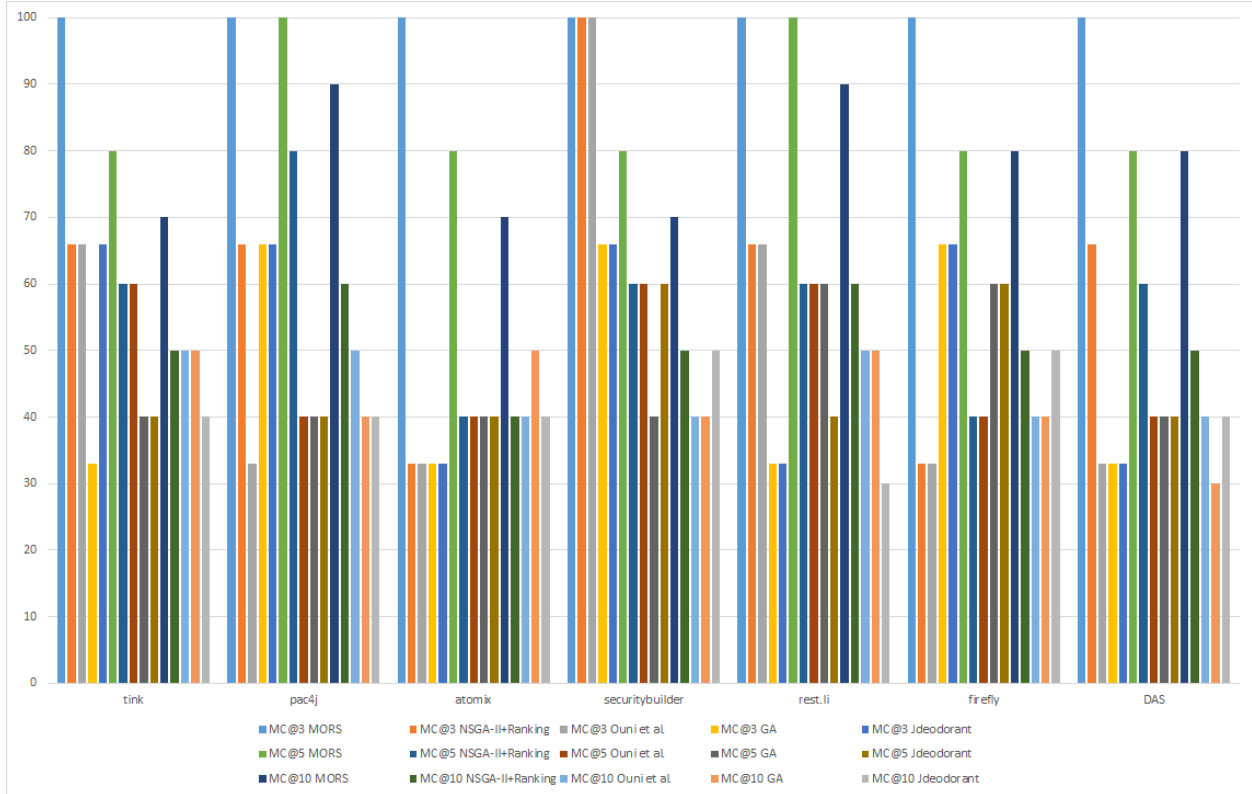
**Table 4.6:** Selected programmers.

System	#Subjects	Avg. Prog. Exp.	Avg. Refactoring Exp.
tink	5	6.5	Very High
pac4j	5	7.5	High
atomix	5	9	High
securitybuilder	5	8	Very High
rest.li	5	8	Very High
firefly	5	9	High
DAS	2	12.5	Very High

**4.2.4.2.3 Experimental Setting** For each algorithm and for each system, we performed a set of experiments using several population sizes: 50, 100, 150 and 200. Then, we specified the maximum chromosome length (maximum number of refactorings). The resulting vector length is proportional to the size of the program to refactor. Thus, the upper and lower bounds on the chromosome length were set to 10 and 100, respectively. The stopping criterion was set to 10,000 fitness evaluations for all algorithms to ensure fairness. To have significant results, for each pair (algorithm, system), we used a trial and error method [602] for parameter configuration. Trial and error is a fundamental method of problem solving. It is characterized by repeated and varied attempts of algorithm configurations.

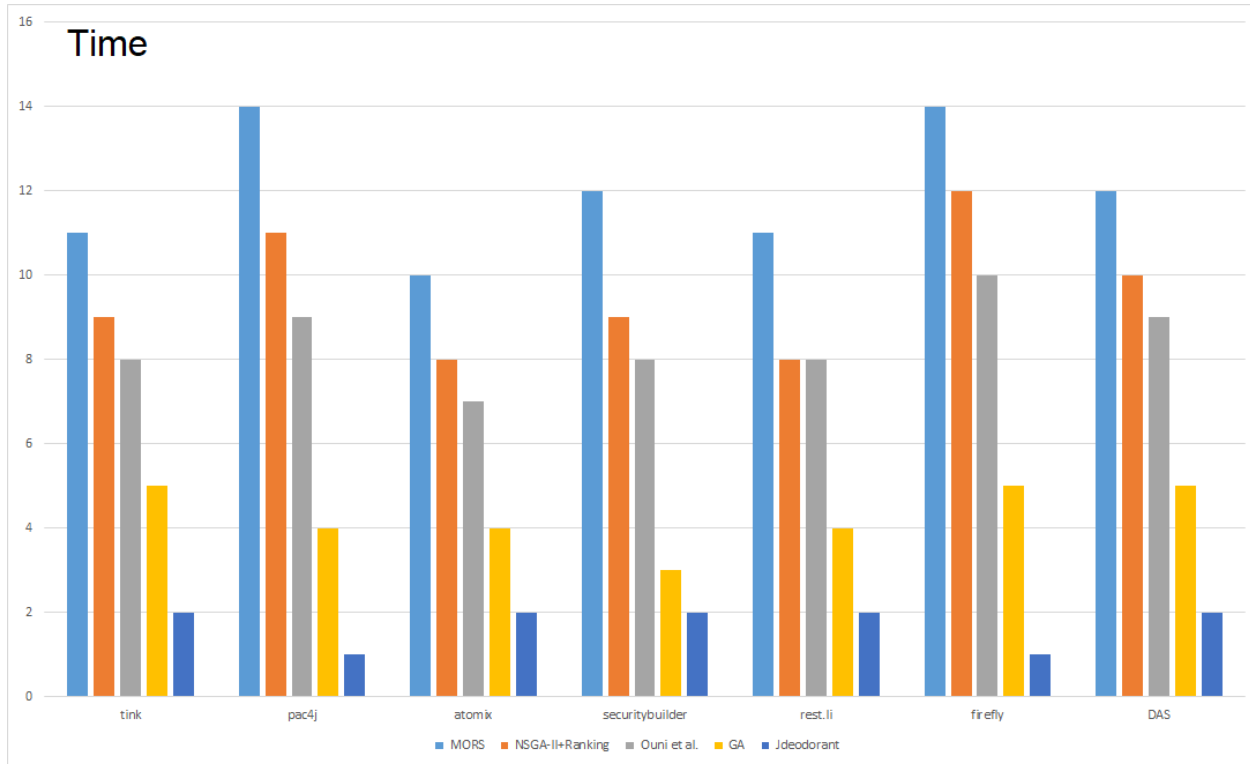
### 4.2.4.3 Results

**Results for RQ1.** The results of Figures 4.22-4.23 confirm the efficiency of our approach to identify relevant refactorings among the top recommendations on the six open source systems and the industrial project. Figure 4.22 shows the average manual correctness (MC@k) results of our technique on the different seven systems, with k ranging from 3 to 10. For example, all the recommended refactorings in the top 3 are considered relevant by the participants. Most of the refactorings recommended by our Multi-Objective refactoring search (MORS) approach in the top 5 (k=5) are relevant with an average MC@5 of 92%. The lowest average of manual correctness is 70% for k=10 which is still acceptable since it means that just 3 recommendations out of the 10 are not relevant (even if they are correct). For instance, the refactoring recommendations on the DAS (industrial) system, evaluated by



**Figure 4.22:** The manual evaluation scores (MC@k) on the seven systems with k=3, 5 and 10.

the original developers, are considered all correct for k=3 and k=5 and only two refactorings were not relevant for k=10. It is normal that some refactorings related to security-critical files may not be considered relevant for several reasons such as the risk to break the code versus their benefits. However, our results suggest that developers are interested to refactor security-critical files, particularly considering that the participants were not aware of the goals of the study (e.g., security). Figure 4.23 summarizes the execution times of our approach on the different systems. The average execution time is 11 minutes. The highest execution time was observed on the industrial system (14 minutes). It is normal that execution time is correlated with the size of the analyzed systems since the tool has to parse the files to identify the most security-critical ones, and then run NSGA-II with the different three objectives. We consider the execution times reasonable since we are not addressing a real-time problem. Furthermore, execution times can be reduced further during subsequent executions of the tool since we may only focus on recently modified, instead of



**Figure 4.23:** Average execution time, in minutes, on the seven systems.

running the algorithms on the entire system.

In terms of comparison with existing refactoring techniques, it is clear from the results that our Multi-Objective refactoring search (MORS) approach generated more relevant refactorings as compared to the tools of Ouni et al., JDeodorant, the mono-objective search, and a multi-objective search based on two quality objectives combined with a ranking of refactorings based on the security measure (NSGA-II+Ranking). When manually comparing the results of the different tools, we found that the remaining automated refactorings generated a lot of refactorings comparing to our approach. In fact, the participants were not interested to blindly change anything in the code just to improve quality attribute measures. The mono-objective search performed worse than the different multi-objective approaches on all the systems which confirms the conflicting nature of the different objectives.

In terms of execution time, we found that our performance was worse than existing techniques, due to the higher number of objectives, parsing the files to identify the critical

ones, etc. JDeodorant has the best execution time of approximately 2 minutes per project. To conclude, the execution time is still reasonable for all the studied approaches, especially considering that there are no hard time constraints when performing refactoring.

To answer RQ1, our results for the six open source systems and the industrial system using the different evaluation metrics of relevance and execution time clearly validate the hypothesis that our approach can efficiently refactor security-critical files.

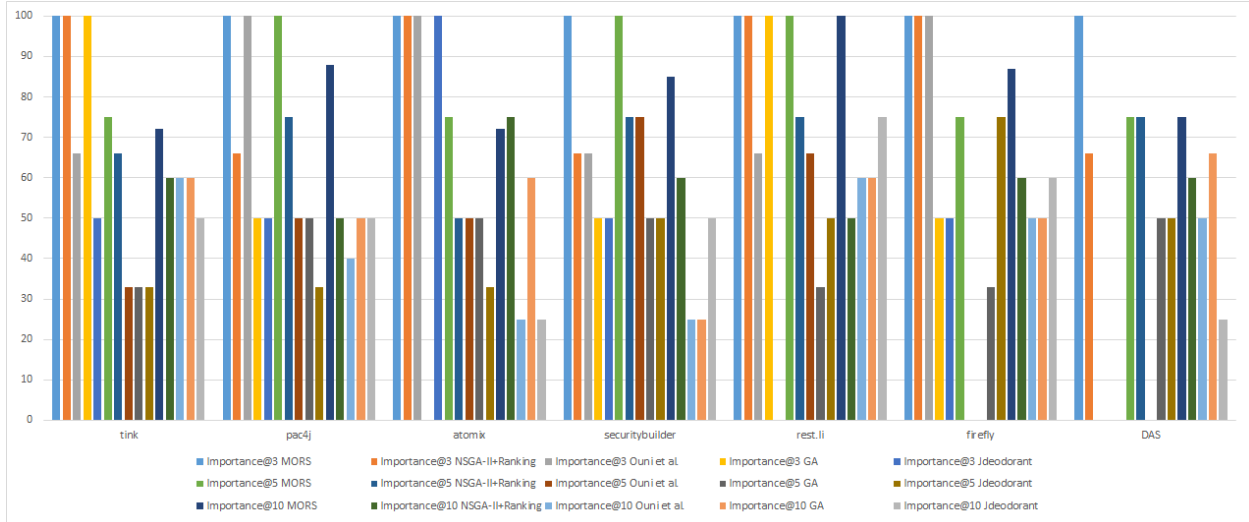
**Results for RQ2.** To evaluate the efficiency of our approach in ranking the refactoring operations based on the combination of quality and security objectives, we used the importance@k measure to evaluate the importance of recommended refactorings. The participants only considered refactorings that were evaluated as relevant in RQ1. Thus, the goal is to validate the hypothesis that the refactorings related to security-critical files are the most important ones, from the developers' perspectives, by comparing the proposed ranking to existing approaches that ranking primarily on code quality measures.

Figure 4.24 confirms that the use of the three objectives of code quality and security to find and rank the refactorings based on NSGA-II was efficient. The majority of the identified refactorings located in the top 3, 5 and 10 were rated high in terms of importance by the participants. An average of 100%, 91%, and 83% of importance@k scores are achieved for k = 3, 5, and 10 respectively on all the systems. It is clear from the figure as well that our MORS approach outperforms all the other techniques including the ranking of the refactorings based on the security measure after running NSGA-II on only the two quality objectives (NSGA-II+Ranking). This confirms the relevance of our choice to integrate security as a separate objective to help the algorithm converge on refactorings targeting security-critical files. However, the NSGA-II+Ranking approach outperformed all the existing refactoring approaches based only on quality measures to rank the recommended refactorings.

**Results for RQ3.** We summarize in the following the feedback of the developers based on the post-study questionnaire.

All participants agreed on the benefits of refactoring security-critical code. They men-





**Figure 4.24:** The severity scores (severity@k) on the seven systems with k=3, 5 and 10.

tioned a number of advantages such as the early identification and prevention of vulnerabilities, the reduction of security breaches as well as the maintenance effort and prioritizing of files that should be carefully reviewed and refactored before approving new commits or releases. Some participants highlighted the benefit of catching test credentials that developers forgot to remove via some refactorings, which is a common mistake among developers. The misuse of these hard-coded credentials is actually an example of the Broken Authentication vulnerability, which is ranked second among the OWASP-TOP 10 Vulnerabilities in web applications in 2018.

The participants emphasized the relevance of refactoring security-critical code fragments for the potential increase in code quality and avoiding confidential data exposure which may result in less cost and better reputation for the organization. Two developers also mentioned the benefit of predicting and avoiding security issues when integrating third party code. For instance several companies are concerned about vulnerabilities when integrating open source projects. These libraries can easily be more loosely coupled with the other parts of the code via refactorings. Several comments mentioned the potential use of our tool at different stages of the software life cycle, whether during the development stage where developers are alarmed that they are dealing with critical-code, or during code reviews before releases when

reviewers focus on the changes made on that portion of code, or even during documentation where all these alerts are recorded for developers in the future. The participants see our tool as relevant for existing continuous integration (CI) and continuous delivery (CD) tools.

Finally, all developers confirmed that they have never used or heard of a tool that leverages this technique for automated refactoring of security-critical code. The practitioners from the industrial partner confirmed that they are not aware of a similar tool. Currently security-critical code fragments are manually identified and refactored during code reviews and a lot of them are missed during that process. And the proposed tool has subsequently been licensed to this partner (in collaboration with the university technology transfer office of the authors).

#### **4.2.5 Threats to Validity**

In our experiments, construct validity threats are related to the absence of similar work that prioritize refactoring for both security and quality purposes. For that reason, we compared our proposal mainly with existing studies that focus on improving quality via refactoring. A construct threat can also be related to the corpus of data used in our experiments since it may introduce some noise to the quality of our results especially with the subjective nature of refactoring. Since we used a variety of computational search and machine learning algorithms, the parameter tuning used in our experiments creates an internal threat that we need to evaluate in our future work. The parameters' values used in our experiments are found by trial-and-error. However, it would be an interesting perspective to design an adaptive parameter tuning strategy for our approach so that parameters are updated during the execution in order to provide the best possible performance.

The variation of correctness and speed between the different participants when using our approach and other tools can be one internal threat. Our approach may not be the only reason for the superior performance because the participants have different programming skills and familiarity with refactoring tools. To counteract this, we assigned the developers

to different systems according to their programming experience so as to reduce the gap between the different groups, and we also adopted a counter-balanced design. Regarding the selected participants, we have taken precautions to ensure that our participants represent a diverse set of software developers with experience in refactoring, and also that the groups formed had, in some sense, a similar average skill set in the refactoring area.

Also, the fact that we did not ask all the participants to evaluate all the systems using all the tools can be considered another threat to the validity of our work. The reason is that it is not reasonable to ask programmers to evaluate more than 30 executions per algorithm to perform the statistical tests. We sacrificed a bit the rigorous of the analysis to have our research validated by practitioners and get meaningful results.

External validity refers to the generalization of our findings. In this study, we performed our experiments on 6 different widely-used open-source systems belonging to the different domains and with different sizes and one industrial project. We considered a mix of security and non-security projects to evaluate the performance of our approach. However, we cannot assert that our results can be generalized to other applications, to programming languages other than JAVA, and to other developers than the 32 participants of our experiments.

#### **4.2.6 Conclusion**

We have presented an approach to recommend refactorings for security critical files while concurrently improving the code quality of a software project. We used the history of vulnerabilities and security bug reports along with a selected set of keywords [468, 472] to automatically identify security-critical files in a project based on source code, bug reports, pull-request descriptions and commit messages. After identifying these security-related files we estimated their risk based on static analysis to check their coupling with other components of the project. Then, our approach recommended refactorings to prioritize fixing quality issues in these security-critical files to improve code quality measures and remove code smells using multi-objective search. We evaluated our approach on six open source projects and one

industrial system to check the relevance of our refactoring recommendations. Our results confirm the effectiveness of our approach as compared to existing refactoring approaches. We are planning, as part of our future work, to extend our validation with a larger set of systems and data sets and to study the potential correlations between security and code quality metrics during the refactoring process.

## 4.3 Intelligent Change Operators for Multi-Objective Refactoring

### 4.3.1 Introduction

Existing refactoring recommendation tools, including those that use non-search-based approaches, routinely generate solutions that include invalid refactorings because they do not account for dependencies among refactorings. Manually applying a sequence of refactorings is common practice in existing tools [450, 603, 604], however these tools treat each refactoring in the sequence in isolation. For instance, Cinnéide et al. [443] investigated the impact only of individual refactorings on quality attribute metrics, such as using Move Method to reduce the coupling of a class, without studying the impact of a sequence of refactorings. Figure 4.25 shows an example of the refactoring recommendations generated by JDeodorant [605] where, similar to other refactoring recommendation tools, the dependencies between the refactorings are not apparent, thus leaving the challenging task of dealing with invalid refactorings to developers. Consequently, developers often prefer manually applying refactorings to using such tools. A key contributor to this problem is that search-based refactoring approaches employ random change operators (e.g., crossover and mutation) to evolve solutions without considering the dependencies among refactorings. Without detecting which refactoring dependencies exist, the change operators used by algorithms routinely invalidate solutions by breaking refactoring dependencies or introducing refactorings whose dependencies are not satisfied. Furthermore, refactoring dependencies provide clues that could be exploited in more intelligent crossover operations to improve decisions on which part(s) of solutions to exchange to produce higher quality offspring.

In this study, we propose intelligent change operators and integrate them into a multi-objective search algorithm, based on NSGA-II [495], to recommend valid refactorings that address conflicting quality objectives such as Reusability, understandability, and effectiveness. The proposed intelligent crossover and mutation operators use: i) the dependencies detected among refactorings to decompose a solution into blocks of refactorings; and ii)

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 4.25:** Sample refactoring recommendations from JDeodorant.

the effects of these blocks on objectives to identify good genes from parents to generate high-quality offspring. A refactoring dependency exists when one refactoring cannot be successfully applied without first applying another. Partitioning refactorings into blocks such that no dependencies span blocks allows change operators to use blocks as the unit of change to avoid invalidating refactorings. Our tool calculates the effect of each block within a solution on the objectives and uses this data to select which blocks to exchange between solutions to improve the first solution’s weaknesses (e.g. the objective with smallest values).

We applied our intelligent change operators to generate refactoring recommendations for four widely used open-source projects and compared this approach to five existing refactoring techniques in terms of the diversity of the solutions, number of invalid refactorings, and the quality of generated solutions. We also conducted a survey with 14 developers to evaluate the correctness and relevance of the refactorings generated by the different algorithms for these projects.

The results show that our technique performed significantly better than the four existing search-based refactoring approaches [8, 5, 6, 7] and an existing refactoring tool not based on heuristic search, JDeodorant [9], with an average manual correctness, precision and recall of 0.89, 0.82, and 0.87, respectively. We used these five refactoring tools and open source projects because: i) they are representative of automated multi-objective search-

based refactoring recommendation techniques; ii) they are publicly available (including the non search-based tool); and iii) the familiarity of the participants with these open source systems.

**Replication Package.** All material and data used in our study are available in our replication package [606].

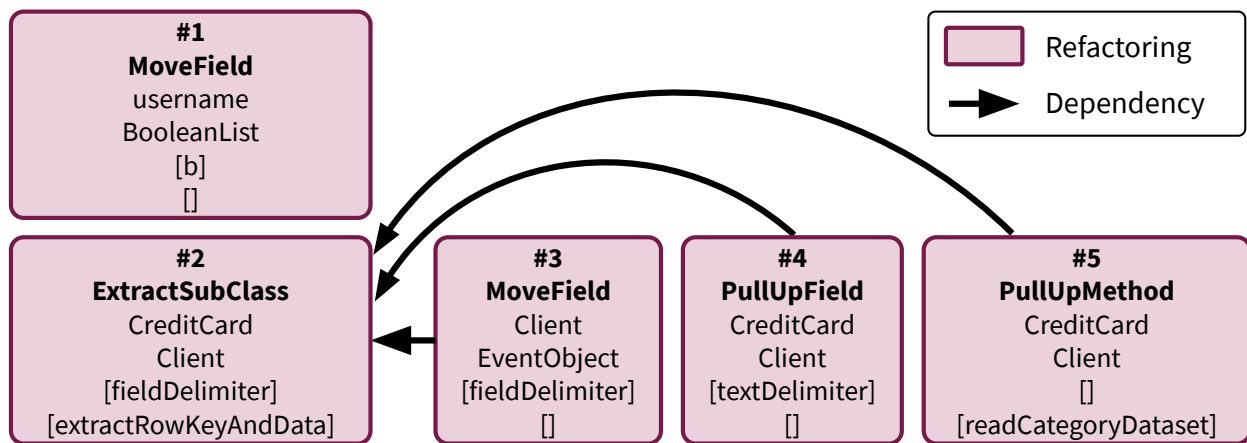
### 4.3.2 Dependency-Aware Refactoring Recommendation System

The most common change operators used in search-based refactoring approaches are the random crossover and mutation operators. In these operators, refactorings are selected randomly from solutions for exchange or replacement with others, which can generate invalid refactorings or invalidate other refactorings (e.g., by removing a refactoring another one depends on). We developed three components to improve the change operators used in the NSGA-II algorithm: i) a refactoring dependency detection algorithm; ii) an intelligent crossover that factors in dependency correctness and the implications of collections of refactorings on fitness functions; and iii) a dependency-aware mutation. Finally, we note that the proposed approach, as described later, can be integrated for both NSGA-II and NSGA-III as they are using the same change operators. The difference between them is that NSGA-III uses a set of reference directions (identified via a niching function), while NSGA-II uses a more adaptive scheme through its crowding distance operator for the same purpose. This difference does not affect our goal of comparing the impact of our intelligent change operators on the final Pareto-front.

#### 4.3.2.1 Refactoring Dependency Theory

Our dependency-aware refactoring recommendation technique relies on an ordering dependency between pairs of refactorings. Specifically, an *ordering dependency* ( $rf_2 \mapsto rf_1$ ) between two refactorings ( $rf_1$  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. 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), 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 are described in our online appendix [606] and were extensively validated for correctness and completeness in current literature [607, 608, 609, 452].



**Figure 4.26:** A simplified example of refactorings that depend on each other.

Figure 4.26 shows a simplified example of a refactoring solution that is composed of refactoring operations that depend on each other. Three of the refactorings (#3, #4, #5) depend on another refactoring (#2) because the Extract Super Class refactoring (#2) creates a new class (Client), 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, #5 to #2.

Refactoring solutions have traditionally been represented as a sequence, likely originating with the common vector representation used in many genetic algorithms. In some cases, a solution could be appropriately represented as a set of sequences, but only if the *refactoring graphs* are simplistic enough. A *refactoring graph* is a weakly connected directed acyclic graph composed of refactoring vertices and ordering dependency edges. In practice, there



are many examples where a sequence vs. graph representation is misleading. For example, if two refactorings (rf2 and rf3) both depend on a common refactoring (rf1), we have a graph for which a sequence representation would be misleading. rf1 must precede rf2 and rf3, but there is no dependency between rf2 and rf3.  $\langle \text{rf1}, \text{rf2}, \text{rf3} \rangle$  would be as acceptable as  $\langle \text{rf1}, \text{rf3}, \text{rf2} \rangle$ . A sequence representation indicates an ordering, and the choice of a graph over a sequence allows us to unambiguously indicate only “real” dependencies. As for the initial refactoring sequence, it is true that the order in that sequence does shape the original graphs. However, the initial sequence is generated randomly for each solution in the population, much as if random graphs were generated.

Using the ordering dependencies as the basis for forming refactoring graphs, algorithm 3 results in a set of graphs with the following traits:

- Each refactoring in a solution is an element of exactly one refactoring graph.
- Some graphs contain a single refactoring because that refactoring is independent of all others. We call these *trivial graphs*.
- The remaining graphs contain multiple refactorings, 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.

The dependencies, as described in the algorithm 3, are detected based on comparisons between pre- and post-conditions of refactorings. The algorithm takes a list of refactorings as input and generates a set of refactoring graphs as output.

Line 1 initializes the lists of refactorings (nodes,  $V$ ) and refactoring dependencies (edges,  $E$ ). Then, the post-conditions of each refactoring of the solution  $C$  (collection of refactorings) are evaluated for matching with the remaining refactorings in  $C$  (Lines 2–12). Specifically, the algorithm looks for any match between predicates of pre- and post-conditions. That is, if any predicate of the post-condition of one refactoring (any element of  $P$ ) matches any

---

**Algorithm 3** Dependency Detection Algorithm.

---

**Input:** Refactoring solution  $C = \{r_1, r_2, r_3, \dots, r_n\}$ **Output:** Set of refactoring graphs  $F = \{f_1, f_2, f_3, \dots, f_m\}$  $V \leftarrow \emptyset, E \leftarrow \emptyset$  **foreach**  $r_i \in C$  **do** $\left[ \begin{array}{l} V \leftarrow V \cup r_i \quad P \leftarrow \text{post\_conditions}(r_i) \quad \text{foreach } r_j \in C - j > i \text{ do} \\ \left[ \begin{array}{l} Q \leftarrow \text{pre\_conditions}(r_j) \quad M \leftarrow P \cap Q \quad \text{if } -M- \neq 0 \text{ then} \\ \left[ \begin{array}{l} E \leftarrow E \cup \{r_j, r_i\} \end{array} \right. \end{array} \right. \end{array} \right.$  $G \leftarrow (V, E) \quad F \leftarrow \text{partition}(G)$  **return**  $F$ 

---

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 4–11).

We repeat this process until all the refactorings have been visited.

### 4.3.2.2 Proposed Intelligent Change Operators

**4.3.2.2.1 Dependency-aware Crossover** We developed a baseline dependency-aware crossover that only preserves the dependencies among refactorings (e.g., without fixing the weaknesses of refactoring solutions). This version, as shown in the algorithm 4, reduces the occurrence of invalid refactorings in solutions because it preserves refactoring dependencies.

---

**Algorithm 4** Dependency-Aware Crossover Algorithm.

---

**Input:** population  $S = \{s_1, s_2, s_3, \dots, s_n\}$  and a probability  $P$ **Output:** offspring population  $S' = \{s'_1, s'_2, s'_3, \dots, s'_n\}$  $S' \leftarrow \emptyset$  **for**  $i \leftarrow 1$  **to**  $|S|/2$  **do** $\left[ \begin{array}{l} \{s_a, s_b\} \leftarrow \text{select random solutions from } S \quad \text{if } \text{random\_number} \leq P \text{ then} \\ \left[ \begin{array}{l} B_a \leftarrow \text{group refactorings of } s_a \text{ into blocks} \quad B_b \leftarrow \text{group refactorings of } s_b \text{ into blocks} \quad \{s'_a, s'_b\} \leftarrow \\ \text{apply single point crossover on } \{B_a, B_b\} \quad S' \leftarrow S' \cup \{s'_a, s'_b\} \end{array} \right. \\ \text{else} \\ \left[ \begin{array}{l} S' \leftarrow S' \cup \{s_a, s_b\} \end{array} \right. \end{array} \right.$ **return**  $S'$ 

---

We start by randomly selecting two solutions,  $s_a$  and  $s_b$ , as parents for new offspring (Line 3). Then, we group the refactorings of  $s_a$  and  $s_b$  into blocks (Lines 5–6) based on the dependencies detected by the algorithm 3. Each block contains a single trivial or non-trivial graph. We then perform a single-point crossover (Line 7) that exchanges blocks of refactorings rather than individual refactorings, which avoids invalidating refactorings because all dependencies are isolated within blocks. This results in two offspring, each with

genetic information from both parents.

**4.3.2.2.2 Intelligent Crossover** Our intelligent crossover operator is an improvement over random crossover in two ways: it uses refactoring dependencies to reduce the occurrence of invalid refactorings and it chooses blocks of refactorings for exchange that will improve a solution’s weaknesses, producing higher quality offspring. The pseudo-code of our proposed intelligent crossover operator is presented in the algorithm 5.

---

**Algorithm 5** Intelligent Crossover Algorithm.

---

**Input:** Population  $S = \{s_1, s_2, s_3, \dots, s_n\}$  and a probability  $P$

**Output:** Offspring population  $S' = \{s'_1, s'_2, s'_3, \dots, s'_n\}$

$S' \leftarrow \emptyset$  **for**  $i \leftarrow 1$  **to**  $|S|/2$  **do**

$\{s_a, s_b\} \leftarrow$  select random solutions from  $S$  **if**  $random\_number \leq P$  **then**

$s_{best} \leftarrow$  higher quality solution of  $s_a$  and  $s_b$   $s_{worst} \leftarrow$  lower quality solution of  $s_a$  and  $s_b$   $B_{best} \leftarrow$

        group refactorings of  $s_{best}$  into blocks  $B_{worst} \leftarrow$  group refactorings of  $s_{worst}$  into blocks  $W_{best} \leftarrow$

        get all weaknesses of  $s_{best}$  **if**  $W_{best} = \emptyset$  **then**

$W_{best} \leftarrow$  get the objective that improves the least with  $s_{best}$

$I \leftarrow$  sort the blocks of  $B_{worst}$  based on potential improvement to  $S_{best}$   $I' \leftarrow$  select the blocks from

$I$  that improve  $S_{best}$   $n \leftarrow$  select random number between 0 and  $|I'|$   $\{s'_{best}, s'_{worst}\} \leftarrow$  apply single

        point crossover, exchanging  $n$  blocks between  $B_{best}$  and  $I$   $S' \leftarrow S' \cup \{s'_{best}, s'_{worst}\}$

**else**

$S' \leftarrow S' \cup \{s_a, s_b\}$

**return**  $S'$

---

In essence, the intelligent crossover operator mixes the best genes of the weaker solution with random genes of the better solution. First, we randomly select two solutions,  $s_a$  and  $s_b$  (Line 3). We then determine the better solution by computing how much each solution improves the objectives (using a weighted sum) of the project to be refactored (Lines 5–6). As before, we group refactorings of both solutions into blocks (Lines 7–8) to preserve refactoring dependencies during crossover. We then determine which objectives are considered the weaknesses of the better solution  $S_{best}$  (Lines 9–12) (part a in Figure 4.27). Any objectives that are worse after applying the better solution are considered weaknesses (e.g. objective 2 in Figure 4.27 part a). If no objectives are worse after applying the better solution, we select the objective that improves the least after applying the solution as the sole weakness. Then, we sort the blocks of the weaker solution  $B_{worst}$  based on how each would impact the objectives (using a weighted sum) of the better solution  $S_{best}$  (Line 13) (part b in Figure 4.27). In

part c of algorithm 5, We pick a random number between 1 and the number of blocks in the weaker solution that would improve the better solution (Line 15) to determine the number of blocks for crossover. Finally, we create two offspring using single point crossover (Line 16) that moves the  $n$  blocks from the weaker solution with the best impact on the stronger solution's objectives to the stronger solution and  $n$  random blocks from the stronger solution to the weaker solution.

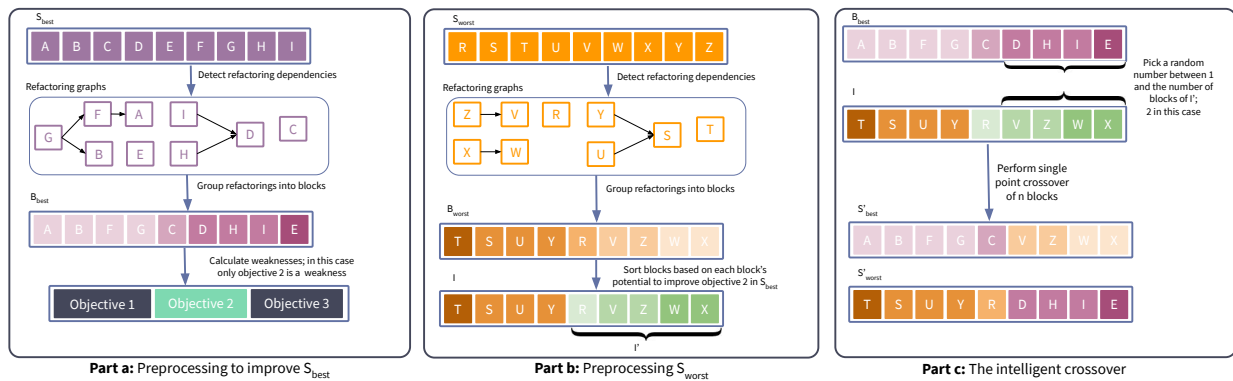


Figure 4.27: An illustration of the intelligent crossover.

**4.3.2.2.3 Dependency-aware Mutation** Our proposed dependency-aware mutation operator is defined in algorithm 6 and illustrated in Figure 4.28. We modified the random mutation operator to preserve refactoring dependencies. For each solution  $S$ , we randomly select a floating-point value. If this value is less than the mutation probability (Line 1), we detect refactoring dependencies (Part a in Figure 4.28) and identify mutable refactorings (Line 2) (Part b in Figure 4.28). A mutable refactoring must satisfy at least one of the following:

- It does not participate in any dependencies (e.g.,  $E$  and  $C$  in Figure 4.28).
- It is part of a non-trivial graph, but no other refactorings depend on it (e.g.,  $G$ ,  $I$  and  $H$  in Figure 4.28).
- It is part of a non-trivial graph, but it has an unsatisfied pre-condition and is already invalid (e.g.,  $A$  in Figure 4.28).

Then, we chose a random number between 1 and the number of mutable refactorings (Line 3). This number represents the number of refactorings that we will mutate in refactoring solution  $S$ . Finally, we replace  $N$  refactorings in  $S$  with random refactoring operations and parameters (Line 4–7) (Part c in Figure 4.28).

---

**Algorithm 6** Dependency-aware Mutation Algorithm.

---

**Input:** Solution  $S = \{r_1, r_2, r_3, \dots, r_n\}$  and a probability  $P$

**Output:** Mutated solution  $S$

**if**  $random\_number \leq P$  **then**

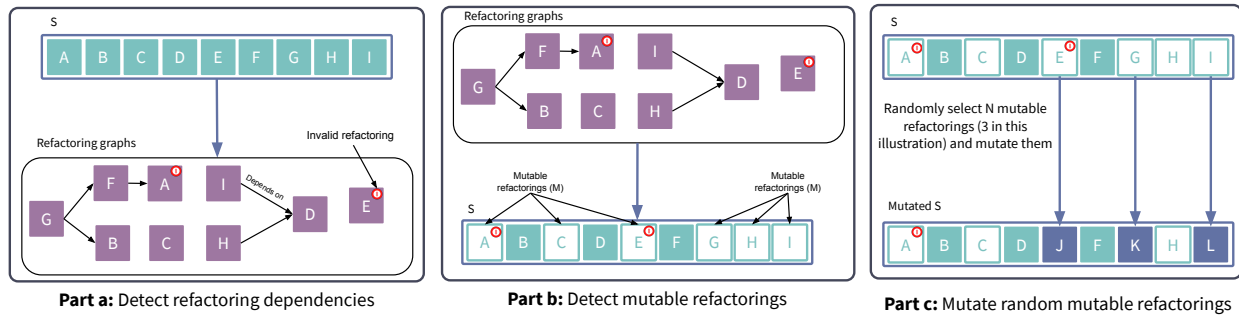
$M \leftarrow$  detect mutable refactorings from  $S$   $N \leftarrow$  random number between 1 and  $|M|$  **for**  $i \leftarrow 0$  **to**  $N - 1$

**do**

$r_j \leftarrow$  random refactoring from  $M$  replace  $r_j$  in  $S$  with a random refactoring

**return**  $S$

---



**Figure 4.28:** An illustration of the dependency-aware mutation.

### 4.3.3 Empirical Study

#### 4.3.3.1 Research Questions

The following research questions guide the evaluation of our proposed approach:

**RQ1. Correctness.** To what extent can our approach reduce the number of invalid refactorings compared to other multi-objective refactoring recommendation techniques?

**RQ2. Quality.** To what extent can our approach generate refactoring solutions with better diversity, convergence, and quality improvement compared to other multi-objective refactoring techniques?

**RQ3. Relevance.** How do developers evaluate the impact of our approach in practice?

To answer RQ 1, we chose the algorithm proposed by Mkaouer et al. [8] based on NSGA-III, because it outperforms the existing multi-objective techniques [5, 6, 7] that use random change operators. Please note that NSGA-II and NSGA-III are using the same change operators as explained in the previous section. We also considered two operation-variants of NSGA-II that optimize the same quality objectives as summarized in Table 4.7.

**Table 4.7:** The three operation-variants of the NSGA-II algorithm.

Algorithm	Definition
NSGA-II	NSGA-II with random Single Point crossover and Bit Flip mutation (Mkaouer et al. [8])
Dep-NSGA-II	NSGA-II with dependency-aware change operators (Sections 4.3.2.2.1 and 4.3.2.2.3)
Intel-NSGA-II	NSGA-II with intelligent crossover and dependency-aware mutation (Sections 4.3.2.2.2 and 4.3.2.2.3)

We selected four open-source Java projects (show in Table 4.8) that were used in the work of Mkaouer et al. [8]. These projects are from different domains and have different sizes along with a significant number of contributors over more than 10 years. Furthermore, the selected projects are widely used and extensively involved over time which may justify the need for refactoring.

Also, we checked the validity of pre- and post-conditions of all refactorings in all solutions in each generation for all three algorithms on the four projects. We measured the total number of conflicts for each generation as the percentage of invalid refactorings among all refactorings in all solutions in that generation. We also measured the percentage of invalid refactorings per solution in each generation to see the distribution of invalid refactorings across solutions.

To answer RQ 2, we compared the three algorithms in terms of execution time, performance indicators, and improvement in quality metrics of the Pareto-front solutions. Due to

**Table 4.8:** Open-source projects studied.

System	Release	# of Classes	KLOC
ArgoUML	v0.3	1358	114
JHotDraw	v7.5.1	585	25
GanttProject	v1.11.1	245	49
Apache Ant	v1.8.2	1191	112

the stochastic and non-deterministic nature of meta-heuristic algorithms, different runs of the same algorithm solving the same problem typically give different outcomes. For this reason, we performed 30 runs for each algorithm on each project to make sure that the results are statistically significant.

Finally, to answer RQ 3, we conducted a survey with a group of 14 active developers to identify and manually evaluate the relevance of the refactorings generated by our approach. At the top of the criteria mentioned above, the projects used for answering RQ 1 were selected since the participants are familiar with them so they can provide relevant feedback given their knowledge.

#### 4.3.3.2 Evaluation Metrics

We validate our results using the following metrics.

For RQ 1, we want to estimate the correctness of the solutions generated by the three algorithms. For that, we compute the percentage of invalid refactorings in each generation by inspecting the validity of pre- and post-conditions of each refactoring operation. These conditions are discussed by Opdyke et al. [33]. The exhaustive list can be found in the online appendix [606]. We also computed the percentage of invalid refactorings per refactoring solution generated by the three algorithms at each generation.

For RQ 2, we use the following three metrics as performance indicators to evaluate the quality of solutions generated by the three algorithms:

- *Contributions* ( $I_C$ ) [610] measures the proportion of solutions that lie on the reference front (RS) [611]. The higher this proportion, the better the quality of solutions.

- *Inverted Generational Distance* ( $I_{GD}$ ) [612] is a convergence measure that corresponds to the average Euclidean distance between the approximate Pareto-front provided by an algorithm and the reference Pareto-front. Small values are desirable.
- *Hypervolume* ( $I_{HV}$ ) [613] measures the volume covered by members of a Pareto-front in objective space delimited by a reference point. An important feature of this metric is its ability to capture diversity and convergence of solutions. A higher hypervolume value is desirable.

We also calculated another metric based on QMOOD that estimates the quality improvement for the project by comparing the quality before and after refactorings generated by the three algorithms. For each refactoring solution  $S$ , the quality improvement after applying  $S$  is estimated as:

$$Q_S = \sum_{i=1}^6 Q_{q_i} \text{ where } Q_{q_i} = q'_i - q_i \quad (4.8)$$

where  $q_i$  and  $q'_i$  represent the value of QMOOD quality attribute  $i$  before and after applying  $S$ , respectively. For each algorithm, we average the normalized quality improvements across solutions in the Pareto-front generated by each algorithm and we compare them. In addition, we compute the execution time of each generation using the three algorithms.

Finally, for RQ 3, we validated the generated refactoring solutions quantitatively and qualitatively. For qualitative assessment, we compared our solutions to a baseline of solutions generated by other multi-objective techniques [5, 6, 7, 8] and by JDeodorant [9], a tool not based on heuristic search. All the search-based refactoring techniques are based on multi-objective search, but each uses different objectives and solution representations. All use the same random change operators, which helps to confirm whether good recommendations result from using our intelligent change operators. The current Eclipse plug-in version of JDeodorant identifies some types of design defects using quality metrics and proposes a list of refactorings to fix them. For the comparison with JDeodorant, we limited the



comparison to the same refactoring types supported by both our approach and JDeodorant. For the quantitative assessment, we calculated precision and recall scores by comparing the refactorings recommended by each of the multi-objective algorithms and JDeodorant with those refactoring manually suggested by the participants (the expected refactorings).

$$Precision = \frac{\text{Recommended Refactorings} \cap \text{Expected Refactorings}}{\text{Recommended Refactorings}} \quad (4.9)$$

$$Recall = \frac{\text{Recommended Refactorings} \cap \text{Expected Refactorings}}{\text{Expected Refactorings}} \quad (4.10)$$

After the developers manually suggested refactorings for the projects, we asked them to evaluate the tools' recommendations since their suggestions may not be the only reasonable solution. We asked the participants to assign 0 or 1 to every refactoring solutions generated by the multi-objective algorithms and JDeodorant. A 0 means that the refactoring is not relevant or invalid, and 1 means that the refactoring is meaningful and relevant.

We computed manual correctness as the number of meaningful refactorings divided by the total number of recommended refactorings. Meaningful refactorings were identified by considering the majority opinion across participants for each refactoring.

$$\text{Manual Correctness} = \frac{|\text{Meaningful Refactorings}|}{|\text{Recommended Refactorings}|} \quad (4.11)$$

#### 4.3.3.3 Parameters Tuning

In order to fairly compare the results among the three algorithms in Table 4.7 and the multi-objective algorithms used in our survey [5, 6, 7, 8], we performed the same number of evaluations per run (3k) and used the same initial population size (100). We used the maximum number of evaluations as our stopping criterion. The crossover and mutation probabilities are set to 0.95 and 0.02 respectively. The minimum and maximum number of refactorings per solutions are set to 100 and 200, respectively.

#### 4.3.3.4 Subjects

We evaluated our approach with 14 active industry developers who volunteered to participate in our survey as part of an industry-sponsored research collaboration. We selected individuals with extensive experience applying refactorings in industry and using the selected open source projects in their work. Each filled out a pre-study survey that collects background information, such as their programming experience and their role within their companies.

We divided the participants into four groups balancing skill level and familiarity with the open source projects. The details of the participants and the projects they evaluated are found in Table 4.9. We gave participants a two-hour lecture about software quality assessment and refactoring. During the two-hour lecture, we did not reveal to the participants which refactorings were generated by which app to avoid any possible bias. We provided general knowledge regarding refactoring and showed them how to read and interpret the refactoring solutions and focused on explaining the required steps to complete the survey.

We assessed their knowledge on the open source projects and their performance in evaluating and suggesting refactoring solutions. The participants were asked to assess the correctness and relevance of the refactorings recommended by the multi-objective algorithms [5, 6, 7, 8] and JDeodorant [9] on all four projects. They were shown refactoring recommendations per project without knowing where the recommendations came from.

Since the multi-objective algorithms generate many refactoring solutions in the Pareto front, it was not feasible to ask the participants to evaluate all the solutions. Therefore, to perform meaningful and fair comparisons for each project and algorithm, we selected the solution using a knee-point strategy [614]. The knee point corresponds to the solution with the maximal trade-off among the objectives, which could be seen as the mono-objective solution with equally weighted objectives if the objectives do not conflict. Thus, we selected the solution with the median hypervolume IHV value. The average number of refactorings evaluated by each participant is 58. We ensured that each refactoring was evaluated by

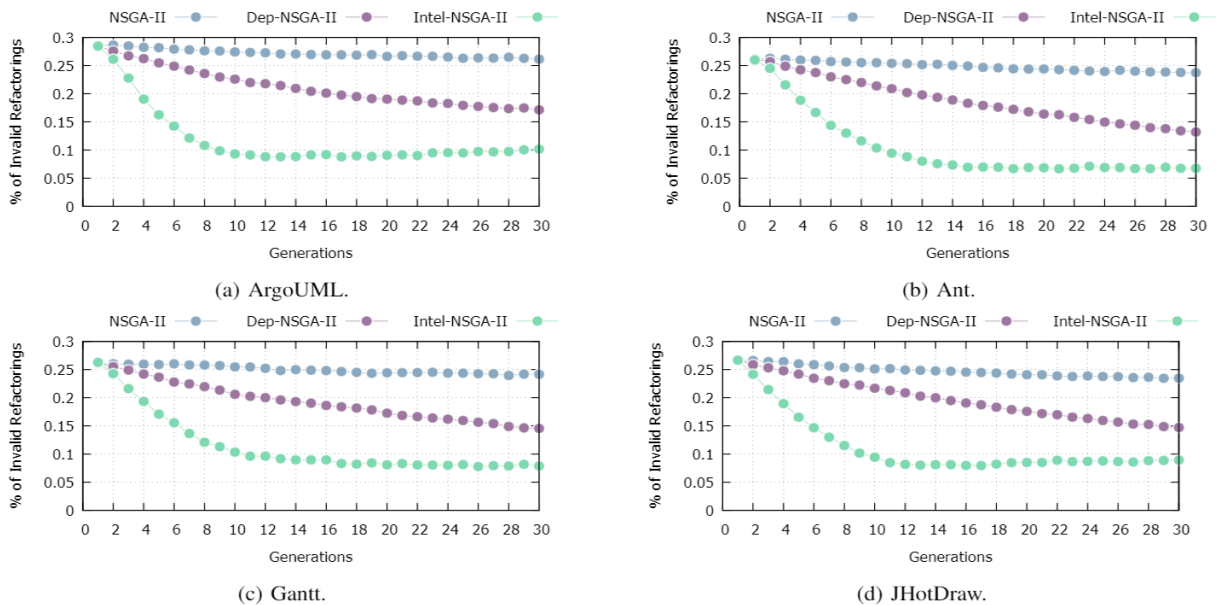
two developers, and we considered it relevant if both agreed (the overall Cohen’s kappa was 0.91).

**Table 4.9:** Participant details.

System	# of Subjects	Avg. Prog. Experience (Years)	Refactoring Experience
ArgoUML	4	10	High
JHotDraw	3	11.5	Very High
GanttProject	3	10.5	High
Apache Ant	4	12	Very High

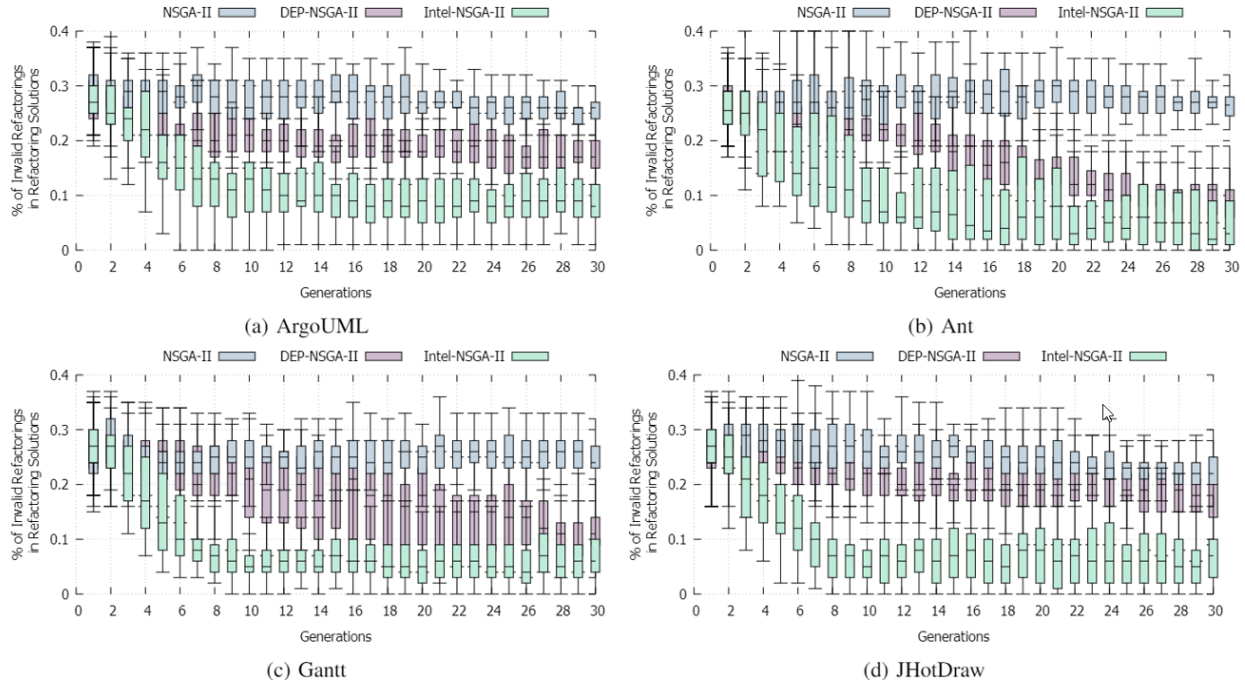
### 4.3.3.5 Results

**4.3.3.5.1 RQ1: Correctness** Figure 4.29 shows the percentage of invalid refactorings across all solutions in each generation for each algorithm for each open source project. All algorithms have 100 non-dominated solutions in the final Pareto-front.



**Figure 4.29:** Percentage of invalid refactorings across all solutions per generation for NSGA-II, Dep-NSGA-II, and Intel-NSGA-II.

The highest percentages of invalid refactorings for all projects was produced by NSGA-II, though it does reduce the percentage of invalid refactorings by a negligible amount as generations progress. Dep-NSGA-II reduces the percentage of invalid refactorings compared



**Figure 4.30:** Percentage of invalid refactorings in refactoring solutions using NSGA-II, Dep-NSGA-II, and Intel-NSGA-II.

to regular NSGA-II [8] by 44.34%, 34.42%, 39.77%, and 37.29% for Ant, ArgoUML, Gantt, and JHotDraw, respectively. Intel-NSGA-II, however, outperformed the other algorithms and reduces the percentage of invalid refactorings compared to NSGA-II [8] by 71.52%, 61.15%, 67.43%, and 61.95% for Ant, ArgoUML, Gantt, and JHotDraw, respectively. Intel-NSGA-II also reduces the percentage of invalid refactorings more quickly than the other algorithms at the population level.

Also, Figure 4.29 reveals that NSGA-II generates a roughly constant percentage of invalid refactorings equal to or greater than 25%. By introducing the dependency-aware change operators, Dep-NSGA-II reduced the number of invalid refactorings to roughly 15% in the 30th generation. Figure 4.29 also reveals a major decrease in the number of invalid refactorings caused by Intel-NSGA-II in the first 12 generations; then it becomes roughly constant and equal to less than 10%. Thus, the number of generations to reach a stable fraction of invalid refactorings is almost the same per algorithm independently from the evaluated project.

Finally, we examined the impact of our proposed change operators at the solution level.

Figure 4.30 shows the distribution of the percentage of invalid refactorings within solutions. Intel-NSGA-II achieves the lowest percentage of invalid refactorings in solutions across all generations for all projects followed by Dep-NSGA-II and NSGA-II, respectively.

**Q Key findings:** *Intel-NSGA-II* reduces the percentage of invalid refactorings in the population and refactoring solutions by an average of 65.51% and 43.71% compared to NSGA-II [8] and Dep-NSGA-II, respectively.

**Table 4.10:** Performance indicators results for NSGA-II, Dep-NSGA-II, and Intel-NSGA-II.

System	Algorithm	$I_C$	$I_{GD}$	$I_{HV}$
ArgoUML	NSGA-II	0.0172	$0.0343 \pm 0.0342$	$0.0222 \pm 0.0205$
	Dep-NSGA-II	0.3172	$0.0303 \pm 0.0081$	$0.0349 \pm 0.0186$
	Intel-NSGA-II	<b>0.6655</b>	<b><math>0.0262 \pm 0.0078</math></b>	<b><math>0.0801 \pm 0.0855</math></b>
Ant	NSGA-II	0.0041	$0.0242 \pm 0.0049$	$0.0176 \pm 0.0205$
	Dep-NSGA-II	0.1632	$0.0205 \pm 0.0047$	$0.0329 \pm 0.0119$
	Intel-NSGA-II	<b>0.8326</b>	<b><math>0.0122 \pm 0.0035</math></b>	<b><math>0.1080 \pm 0.0555</math></b>
GanttProject	NSGA-II	0.0036	$0.0205 \pm 0.0027$	$0.0218 \pm 0.0111$
	Dep-NSGA-II	0.1749	$0.0193 \pm 0.0037$	$0.0302 \pm 0.0209$
	Intel-NSGA-II	<b>0.8215</b>	<b><math>0.0103 \pm 0.0024</math></b>	<b><math>0.1191 \pm 0.0536</math></b>
JHotDraw	NSGA-II	0.1044	$0.0253 \pm 0.0050$	$0.0266 \pm 0.0214$
	Dep-NSGA-II	0.0413	$0.0225 \pm 0.0040$	$0.0349 \pm 0.0175$
	Intel-NSGA-II	<b>0.8544</b>	<b><math>0.0136 \pm 0.0036</math></b>	<b><math>0.1341 \pm 0.0635</math></b>

**4.3.3.5.2 RQ2: Quality** Table 4.10 shows the average  $I_C$ ,  $I_{GD}$ , and  $I_{HV}$  of the 30 runs of the three algorithms. The values in bold are the best values achieved for each performance indicator per project. *Intel-NSGA-II* achieved the highest  $I_{HV}$  and  $I_C$  and the lowest  $I_{GD}$  for all projects. Dep-NSGA-II was able to improve the  $I_{HV}$ ,  $I_C$ ,  $I_{GD}$  compared to NSGA-II by up to 86.93%, 4758.33%, and 15.28%, respectively. Intel-NSGA-II was able to improve the  $I_{HV}$ ,  $I_C$ ,  $I_{GD}$  compared to NSGA-II by up to 513.63%, 22719.44%, and 49.75%, respectively. This shows that *Intel-NSGA-II* produces better convergence and diversity than the other algorithms.

Table 4.11 shows the average quality improvement of solutions, as well as their standard deviations. The bold values are the best values obtained for each metric for each project. *Intel-NSGA-II* produced the best quality improvement in almost all cases. *NSGA-II* pro-

**Table 4.11:** Average quality improvement of the solutions generated by NSGA-II, Dep-NSGA-II, and Intel-NSGA-II.

System	Algorithm	Effectiveness	Extendibility	Flexibility	Functionality	Reusability	Understandability
ArgoUML	NSGA-II	0.0557 ± 0.0147	0.1484 ± 0.0335	0.0077 ± 0.0077	0.0077 ± 0.0042	0.0130 ± 0.0051	0.0260 ± 0.0099
	Dep-NSGA-II	0.0615 ± 0.0112	0.1639 ± 0.0297	<b>0.0109 ± 0.0084</b>	0.0082 ± 0.0045	0.0129 ± 0.0054	0.0255 ± 0.0098
	Intel-NSGA-II	<b>0.0646 ± 0.0174</b>	<b>0.1798 ± 0.0332</b>	0.0094 ± 0.0104	<b>0.0115 ± 0.0045</b>	<b>0.0206 ± 0.0035</b>	<b>0.0302 ± 0.0095</b>
Apache Ant	NSGA-II	0.0177 ± 0.0049	0.0296 ± 0.0112	0.0073 ± 0.0088	0.0070 ± 0.0046	0.0086 ± 0.0021	0.0125 ± 0.0074
	Dep-NSGA-II	0.0214 ± 0.0050	<b>0.0362 ± 0.0098</b>	0.0086 ± 0.0085	0.0083 ± 0.0046	0.0099 ± 0.0020	<b>0.0136 ± 0.0072</b>
	Intel-NSGA-II	<b>0.0230 ± 0.0054</b>	0.0338 ± 0.0098	<b>0.0164 ± 0.0111</b>	<b>0.0123 ± 0.0055</b>	<b>0.0139 ± 0.0022</b>	0.0119 ± 0.0083
GanttProject	NSGA-II	0.0285 ± 0.0077	0.0677 ± 0.0168	0.0045 ± 0.0093	0.0059 ± 0.0048	0.0080 ± 0.0029	0.0098 ± 0.0080
	Dep-NSGA-II	<b>0.0340 ± 0.0077</b>	<b>0.0775 ± 0.0166</b>	0.0046 ± 0.0107	0.0067 ± 0.0052	0.0094 ± 0.0026	<b>0.0124 ± 0.0095</b>
	Intel-NSGA-II	0.0335 ± 0.0097	0.0710 ± 0.0195	<b>0.0120 ± 0.0153</b>	<b>0.0123 ± 0.0073</b>	<b>0.0147 ± 0.0033</b>	0.0093 ± 0.0124
JHotDraw	NSGA-II	0.0451 ± 0.0074	0.1028 ± 0.0175	0.0138 ± 0.0109	0.0122 ± 0.0047	0.0141 ± 0.0028	0.0126 ± 0.0117
	Dep-NSGA-II	0.0463 ± 0.0079	0.1058 ± 0.0191	0.0084 ± 0.0102	0.0085 ± 0.0054	0.0109 ± 0.0040	0.0132 ± 0.0084
	Intel-NSGA-II	<b>0.0487 ± 0.0111</b>	<b>0.1062 ± 0.0193</b>	<b>0.0169 ± 0.0176</b>	<b>0.0154 ± 0.0084</b>	<b>0.0180 ± 0.0041</b>	<b>0.0136 ± 0.0137</b>

duced the lowest quality improvement in 18 out of 24 cases. *Dep-NSGA-II* was able to improve the Effectiveness, Extendibility, Flexibility, Functionality, Reusability, and Understandability compared to *NSGA-II* by an average of 13.31%, 51.89%, 5.61%, 2.07%, 2.28%, 9.54%, respectively. *Intel-NSGA-II* was able to improve the Effectiveness, Extendibility, Flexibility, Functionality, Reusability, and Understandability compared to *NSGA-II* by an average of 17.86%, 46.94%, 83.96%, 64.94%, 57.87%, and 3.54%, respectively. This demonstrates that our intelligent crossover strategy that targets fixing a solution’s weaknesses leads to higher quality solutions in the final Pareto-front. There is, however, a performance penalty for the extra work performed by intelligent change operators; on average, execution time doubled. In most cases, this is a more than acceptable trade-off for higher quality refactoring recommendations.

We noticed that NSGA-II never produced the best quality improvement in any cases, which means that the dependency-aware change operators play a significant role in improving the quality of the Pareto-front. In addition, whenever Intel-NSGA-II does not produce “the best quality improvement”, the difference between the quality values of Intel-NSGA-II and Dep-NSGA-II is very small. Indeed, the quality improvements rate depends on the number of code smells, size and evolution of the analyzed projects. In our future work, we are planning to validate our approach using more projects to have a clearer understanding of when and why Intel-NSGA-II does not produce “the best quality improvement”.

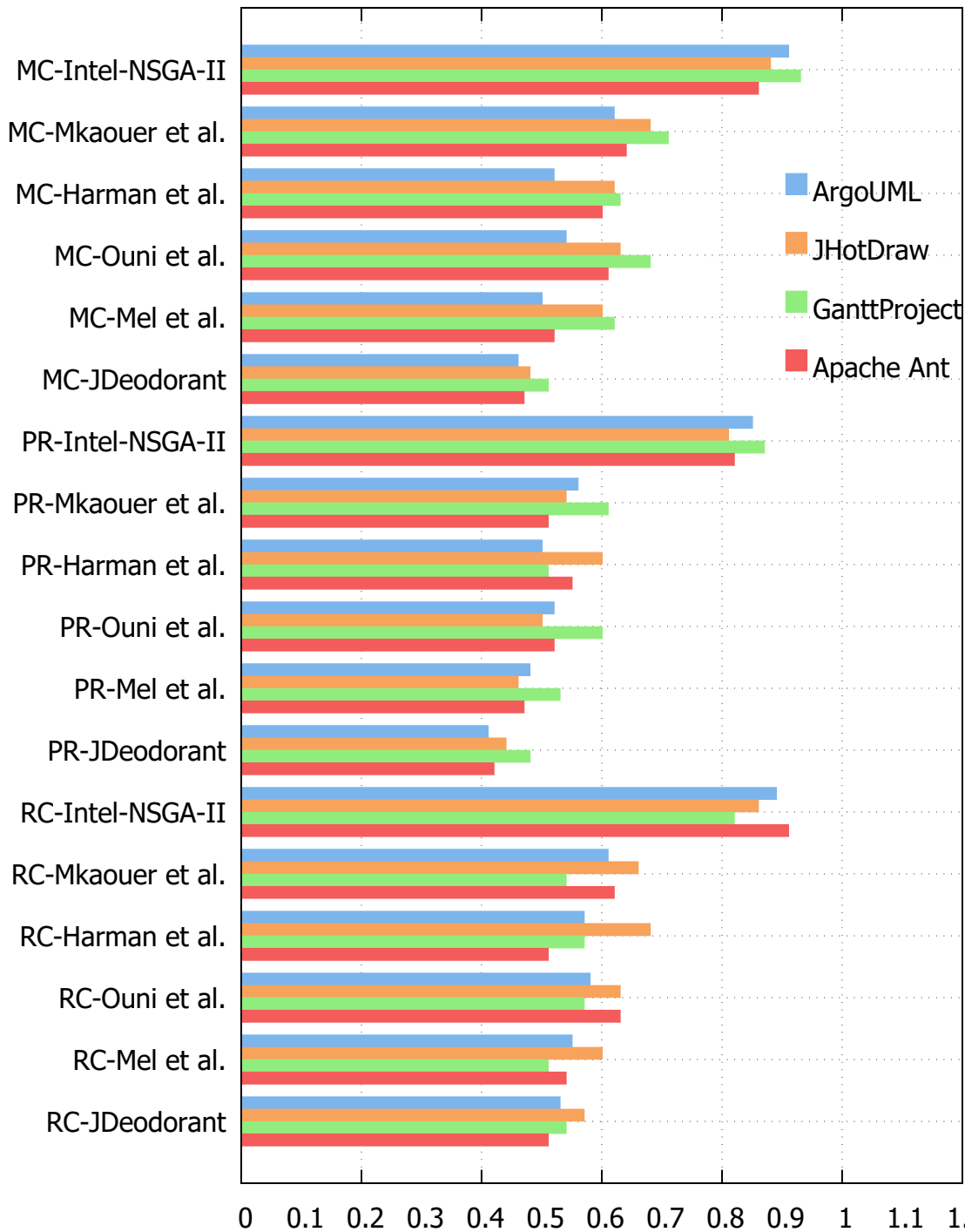
**Q Key findings:** *Intel-NSGA-II* outperforms the other algorithms in terms of diversity, convergence, and quality improvement of the Pareto-front using the different evaluation metrics  $I_C$ ,  $I_{GD}$ , and  $I_{HV}$  by at least 50% with a modest sacrifice in execution time.

**4.3.3.5.3 RQ3: Relevance** Figure 4.31 presents the results of manual correctness, precision, and recall for our Intel-NSGA-II algorithm and state of the art refactoring techniques. The detailed responses of the 14 participants can be found in our appendix [606]. Intel-NSGA-II achieved better manual evaluation scores than [8] and existing approaches in all the metrics for all projects. Indeed, the average manual correctness, precision and recall of our algorithm compared to that of Mkaouer et al. [8] are 0.89, 0.82, and 0.87 to 0.67, 0.56, and 0.67 respectively and much better than the remaining tools. Thus, the participants found our refactoring recommendations applicable and consistent with the source code and their design issues. All participants agreed on the benefits of considering dependencies among refactorings when generating refactoring solutions. They mentioned that Intel-NSGA-II increases their trust in refactoring tools and would save them time and effort on filtering out invalid refactorings.

**Q Key findings:** *Intel-NSGA-II* provided more relevant and meaningful refactorings than state of the art refactoring recommendation techniques based on manual evaluation of recommended refactorings.

#### 4.3.4 Threats to Validity

**Conclusion validity.** We used Design of Experiments (DoE) [615] to mitigate the threat related to parameter tuning. DoE is a methodology for systematically applying statistics to experimentation and is one of the most efficient techniques for tuning parameter settings of evolutionary algorithms. Each parameter has been uniformly discretized in intervals. To mitigate the stochastic nature of the search algorithms, we performed 30 runs per project and algorithm and analyzed the mean results along with the appropriate statistical tests using the Wilcoxon test with a 95% confidence level ( $\alpha < 5\%$ ).



**Figure 4.31:** Manual evaluation of refactoring recommendations generated by the existing multi-objective techniques [5, 6, 7, 8] and the JDeodorant Eclipse plugin [9]).

**Internal validity.** Validation exercise participants had different programming skills and familiarity with refactoring tools. To counter this, we assigned developers to groups according to their experience to reduce the gap between the groups and we adopted a counter-balanced



design. Asking the participants to evaluate the refactoring recommendations for all projects would be too much work for them and would reduce the quality of the survey responses. For this reason, we divided the participants into four groups balancing skill level and familiarity with the open-source projects and we asked each one of them to evaluate a single project. We grouped the participants based on their familiarity with the projects to be evaluated. Indeed, it is critical that the participants are knowledgeable about the code of the evaluated projects so they can make accurate judgment about the recommended refactorings. Also, the relatively small number of participants could also be considered a threat to validity. We selected 14 developers to participate in our validation, targeting developers with knowledge of the studied projects. In-depth interviews with a relatively small number of developers familiar with the studied projects yields deep, quality insights that are more useful than those extracted using an online survey with random participants who are not familiar with the studied projects.

**Construct validity.** Developers might have different opinions about the relevance of recommended refactorings, which may impact our results. Some might think that it is important to refactor, while others might think otherwise. To mitigate this threat, we ensured that each refactoring was evaluated by two developers, and we considered it relevant if both agreed. The overall Cohen's kappa was 0.91 which confirms that there is a significant consensus among developers.

**External threats.** External threats concern the generalization of our findings. Our validation includes only four projects. One reason for this is to attract more quality responses from survey participants. The more tedious the task that participants must complete, the lower the quality of their responses. The second reason is that running all of the algorithms on all of the projects 30 times takes considerable time.

### 4.3.5 Conclusion

To improve the correctness and quality of refactoring recommendations and increase developer trust in search-based refactoring recommendation tools, we proposed a dependency-aware multi-objective refactoring approach with intelligent change operators that find a balance among quality objectives while reducing the number of invalid refactorings. We evaluated this approach on four open-source projects. We compared our results to existing refactoring techniques that use random change operators, as well as to a dependency-aware technique, to understand the impact of considering refactoring dependencies and fixing quality weaknesses in refactoring solutions. The comparisons show that our proposed approach performs significantly better than the baselines in terms of convergence, diversity, and correctness with a reasonable cost in terms of increased execution time. The survey with 14 practitioners confirmed the relevance of our approach.

## 4.4 X-SBR: On the Use of the History of Refactorings for Explainable Search-Based Refactoring and Intelligent Change Operators

### 4.4.1 Introduction

A wide range of work has been done on finding refactoring recommendations using a variety of techniques including template/rule-based tools [616, 617], static and lexical analysis, and search-based software engineering [444]. Recent surveys show that search-based software engineering is widely adopted to find refactoring recommendations [618, 444] due to the conflicting nature of many quality metrics and the large search space of potential refactoring strategies that can be useful depending on the context. For instance, O’Keeffe et al. [6] compared the ability of different local search-based algorithms such as hill climbing and simulated annealing to generate refactoring recommendations that improve the QMOOD [480]. Harman et al. proposed to use multi-objective search for refactoring to improve coupling and reduce cohesion [5]. Ouni et al. [496] and Mkaouer et al. [497] proposed multi-objective and many-objective techniques to balance different conflicting quality metrics when finding refactoring recommendations. Hall et al. [619] and Alizadeh et al. [55] improved the state-of-the-art of search-based refactoring by enabling interaction with the developers and learning their preferences. More detailed descriptions of existing search-based refactoring studies can be found in the following surveys [444, 618].

Despite the promising results of search-based refactoring on both open-source and industry projects, several limitations can still be addressed in order to improve their efficiency. These limitations can apply, in general, to most of the existing search-based software engineering studies [620, 442, 537] but we focus only on search-based refactoring in this contribution. First, the random generation of the initial population can have a significant impact on the execution time and the quality of final solutions [621, 622]. Despite the large amount of data of the history of commits about applied refactorings, existing search-based refactoring studies are still generating the initial population of solutions randomly without exploiting

the prior knowledge of what could construct a good refactoring solution. Second, most of software engineering problems, including refactoring, are discrete. However, the majority of existing studies are using regular change operators such as the random one-point crossover that is more adequate for continuous problems [623]. In fact, a random application of change operators without understanding the good/bad patterns in a refactoring sequence of the solution can simply destroy them, deteriorate the quality, and delay the convergence towards good solutions. Third, current search-based refactoring techniques generate a large sequence of refactorings as one solution without explaining to developers how the different operations in the solution are depending to each other in terms of fixing specific quality issues or improving the fitness functions which can impact their trustworthiness by developers in practice. Finally, the recommendation of refactorings is highly dependent to the developers interest and preferences such as files owned or targeted quality goals. Thus, refactoring recommendations should be customized to the needs of the developers after understanding and learning their behavior and preferences.

In this project, we propose an approach for refactoring recommendations based on a novel knowledge-informed multi-objective optimization algorithm to guide the generation of the initial population, define intelligent genetic operators and explain the generated refactoring solutions (also called the Pareto front). The proposed approach is a combination of an Apriori algorithm and multi-objective search. The first component of our approach is based on an Apriori algorithm [539] to generate association rules using the refactoring history and quality analysis of 18 projects of different sizes and categories. We used RMiner [624] to detect the refactoring operations performed between the commits. These association rules represent patterns linking a combination of refactoring types with their location, characterized using structural metrics, to their impact on improving the quality attributes/fitness functions (e.g. extendibility, functionality, flexibility etc.). Thus, these patterns were used to 1) initialize the first population of solutions, 2) select which refactorings of a solution to replace during crossover and mutation in order to avoid destroying good patterns and 3) explain the obtained

refactoring sequence per solution to the developers by decomposing it to sub-sequences with their potential impact on quality improvements.

We evaluated the execution time, quality of refactoring recommendations and identified refactoring patterns using different evaluation metrics. Statistical analysis of our experiments based on 4 open source systems showed that our proposal performed significantly better than four existing search-based refactoring approaches [8, 5, 6, 7] and an existing refactoring tool not based on heuristic search, JDeodorant [9], in terms of improving the quality and enhance the trustworthiness to apply the recommended refactorings. We used these 5 refactoring tools and the 4 open source projects because 1) they are representative of existing automated multi-objective search-based refactoring techniques, 2) they are publicly available including the non search-based tool and 3) the familiarity of the participants with the open source systems that already part of an existing benchmark not constructed by the authors of this project to avoid any potential bias [55]. We did not compare with manual and interactive refactoring techniques to ensure a fair comparison and focus on the scope of the contributions of this project.

**Replication Package.** All material and data used in our study are available in our replication package [625].

## 4.4.2 X-SBR Approach

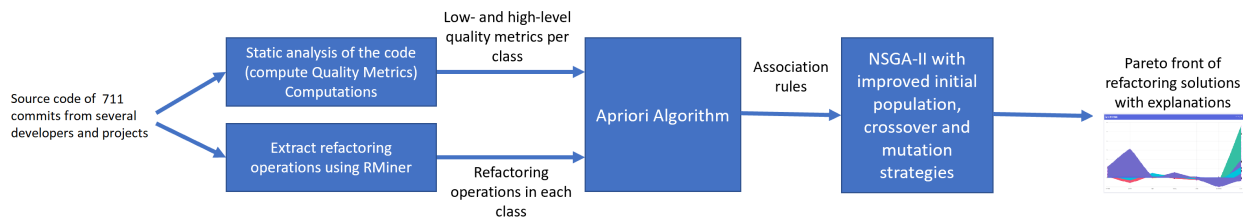
### 4.4.2.1 Overview

The goals of this study are to 1) develop a knowledge-informed NSGA-II [495] by designing operators that prevent the destruction of good patterns in a solution 2) explain the decision made by the algorithm and give justifications to the users about why a refactoring solution can improve specific quality objectives by extracting the relevant patterns and 3) improve the population initialization by using the knowledge from the history of refactorings to create the individuals of the first generation rather than randomly generating them. To reach the stated goals, our approach takes as input the source code of several commits from different developers and projects and generates as output a Pareto front of refactoring solutions along with their explanations presented in a user friendly graphical interface. A refactoring solution is an ordered sequence of refactoring operations. The steps of our approach are as follows:

- **Step 1:** Detect the refactoring history using Rminer [624] and compute quality metrics (described in Tables 2.10 and 2.9).
- **Step 2:** Generation of association rules to link the quality metrics with refactoring operations collected in Step 1.
- **Step 3:** Design of a knowledge-informed NSGA-II including the population generation and change operators based on the rules extracted in Step 2.

We note that only Step 3 needs to be executed on a new system to generate refactoring recommendations. Figure 4.32 summarizes our approach. It takes multiple commits of different systems that the developer worked on as input. For each commit, we analyze the source code automatically to extract low- and high-level quality metrics (refer to Tables 2.10 and 2.9) and we extracted the refactoring using RMiner [624]. Based on the collected data, we applied the Apriori algorithm to find association rules to link low-level quality metrics and

refactoring operations with high-level quality metrics. The rules are composed by two sides. The left-hand side includes only item-sets with elements belonging to the design properties (structure of the code) AND applied refactoring operations. The right-hand side needs to include only item-sets with elements belonging to the QMOOD metrics. The association rules were used to 1) initialize the first population of solutions, 2) select which refactorings of a solution to replace during crossover and mutation in order to avoid destroying good patterns and 3) explain the obtained refactoring sequence per solution to the developers by decomposing it to sub-sequences with their potential impact on quality improvements. Then, we designed and implemented a knowledge-informed NSGA-II to efficiently generate the initial population and perform change operators as detailed later. Finally, our approach can identify the specific refactoring patterns in each solution responsible for the fitness values of each solution improvement or deterioration in the Pareto front.



**Figure 4.32:** Approach Overview

## 4.4.2.2 Training Data

### 4.4.2.2.1 Extracting History of Refactorings

In this study, we used RMiner, a tool proposed by Tsantalis et al. [624], to extract the refactoring operations performed between Git commits. RMiner detects a total of 28 refactoring types at multiple granularity levels—Package, Type, Method, and Field. These types are the following: change package, extract and move method, extract class, extract interface, extract method, extract subclass, extract superclass, extract variable, inline method, inline variable, move and rename attribute, move and rename class, move attribute, move class, move method, move source folder, parameterize variable, pull up attribute, pull up method, push down attribute, push down method,

rename attribute, rename class, rename method, rename parameter, rename variable, replace attribute, and replace variable with attribute.

We selected RMiner since it achieved accurate results in detecting refactorings compared to the state-of-the-art tools, with a precision of 98% and recall of 87%. We provide in the validation section the details of the collected data related to refactorings and quality metrics on open source projects.

#### 4.4.2.3 Association Rule Mining

Apriori is an algorithm for frequent item-set mining and association rule learning that was first defined by Agrawal et al. [539]. A frequent item-set is a set of items appearing together in a database meeting a user-specified threshold. The algorithm starts by finding the frequent individual items in a database and expand them to larger and larger item-sets as long as the appearance of those item-sets is larger than the threshold set by the user. The frequent item-sets found by Apriori can be used to generate association rules which highlight general trends in the database. The pseudo code of the Apriori algorithm can be found in the online appendix [625]. In our study, the transaction database  $D$  consists of the list of classes of all commits, their QMOOD/design metrics after discretization, and applied refactoring operations. The support threshold we considered was equal to 0.936. We defined three types of constraints on the generation of the rules:

- The left-hand side needs to include only item-sets with elements belonging to the design properties AND applied refactoring operations.
- The right-hand side needs to include only item-sets with elements belonging to the QMOOD metrics.
- The left-hand side needs to have at least 4 elements from the design properties item-set.

We included both the design metrics and the refactoring operations in the left-hand side of the rules to have a more relevant association of the refactoring operations with



<b>Extract And Move Method</b> , <b>Inline Variable</b> , <b>CIS</b> :(-2.484, 496.8], <b>MOA</b> :(-0.042, 8.4], <b>NOH</b> :(-0.0002, 0.0002], <b>NOM</b> :(-2.485, 497.0] → <b>Extendibility</b> :(-0.2, 0.5], <b>Flexibility</b> :(-0.2, 0.5]
--

**Figure 4.33:** Example of an association rule

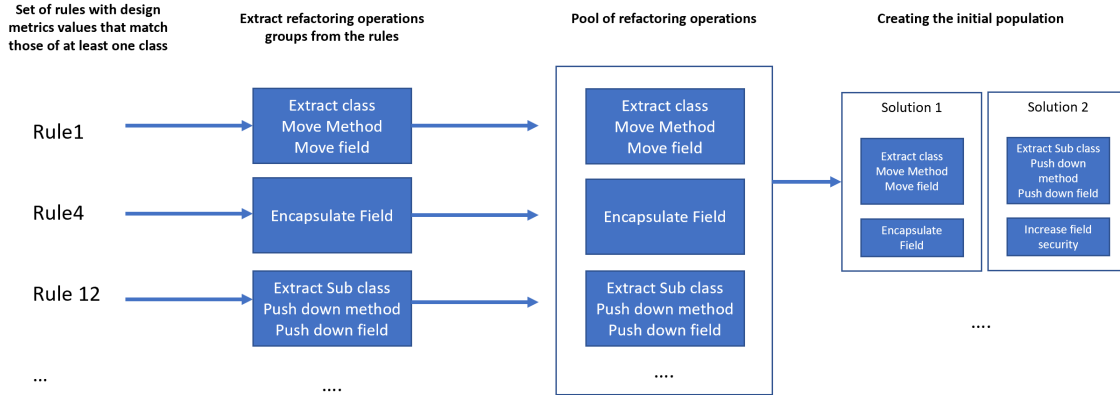
the high-level metrics. For example, we tend to apply the refactoring operator *Increase field Security* when the *Direct Access Metric*—ratio of the number of private and protected attributes to the total number of attributes in a class—is low. Figure 4.33 represents an example of one of the rules generated by the Apriori algorithm. The items in blue, red, and green are respectively the refactoring operations, design metrics, and QMOOD metrics, respectively. The rule can be interpreted as follows: when developers have applied the refactoring types *Extract and move method* and *Inline Variable* in a class that has the design metric CIS, MOA, NOH and NOM within the intervals of  $(-2.484, 496.8]$ ,  $(-0.042, 8.4]$ ,  $(-0.0002, 0.0002]$ , and  $(-2.485, 497.0]$  respectively, then the change (as the difference between before and after refactoring) in extendibility and flexibility will be in the range of  $(-0.2, 0.5]$ ,  $(-0.2, 0.5]$  respectively. We designed a user-friendly interface in our web-app supporting the implementation of the approach proposed in this project so the users can easily understand the explanations rather than reading mined association rules. For example, the UI highlighted the metrics contributing to the recommendation of the refactoring and so on.

#### 4.4.2.4 Knowledge-Informed and Explainable NSGA-II for Search-Based Refactoring

We detail in the following three main components that we design to improve the regular NSGA-II algorithm: 1) the population generation; 2) change operators and 3) the explanations for the selected solution from the Pareto front.

##### 4.4.2.4.1 Initial Population

The initial population strategy is one of the important factors that affect the performance of search algorithms. The initial population has a key



**Figure 4.34:** Improved initial population process

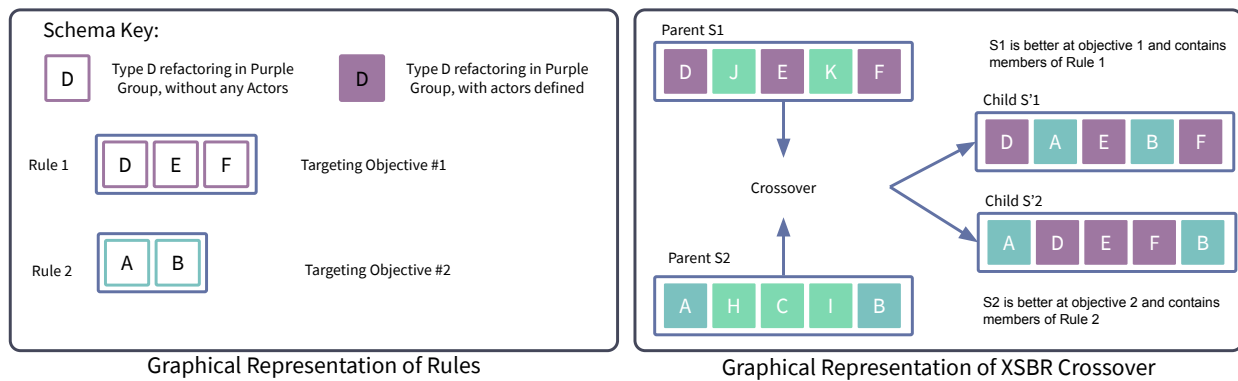
impact on the execution time and the quality of the generated Pareto front. Figure 4.34 summarizes the steps of the improved seeding mechanism. We first start by looking for all the rules, generated by the Apriori algorithm from the refactoring history, that can be applied to the classes of the system to be refactored. In other words, we look for the rules where there exist at least one class, from the system we are trying to refactor, with design metric values that satisfy/match the left-hand side of the rules. Then, we add all the refactoring operations of those rules in one unified pool. We note that we keep the refactorings of each rule as a group—also referred to as pattern—in a way that they are used together as a subsequence in the refactoring solution vector. The reason behind this grouping is that each group of refactorings tend to occur together according to the frequent item-set principle and the refactoring history of developers. Therefore, suggesting them together in a refactoring solution provides more personalized and practical recommendations. To create an initial population of size  $N$ , we randomly choose groups of refactorings from the pool we formed until we fill  $N$  ordered vectors.

**4.4.2.4.2 Crossover** Figure 4.35 is a simplified illustration of how our improved crossover works. We first start by randomly picking two parents,  $S_1$  and  $S_2$ , from the current population.  $S_1$  and  $S_2$  are vectors where each dimension represents a refactoring operation to apply. Then, we create cloning copies of the parents for the new pair of offspring  $S'_1$  and

$S'_2$ . Next, we extract the Apriori rules that satisfy the following two conditions:

- The refactoring pattern in the left-hand side of the rule exists in  $S_1$
- The design metric intervals in the left-hand side of the rule contain the values of the source class design metrics in the refactoring operations of  $S_1$ .

We do the same for the second parent  $S_2$ . We end up having two rule sets  $R_1$  and  $R_2$  related to  $S_1$  and  $S_2$  respectively. Let  $O_1$  and  $O_2$  be the objectives (e.g. the QMOOD metrics) in the right-hand side of the rules in  $R_1$  and  $R_2$  respectively. Now, we compute the fitness function of  $S_1$  and  $S_2$  for all the objectives in  $O_1 \cap O_2$  and we compare them. Let us consider that  $S_1$  has a higher reusability than  $S_2$ . Thus, the algorithm will look for the rule R in  $R_1$  that contains reusability in its right-hand side. We extract the refactoring operations from  $S_1$  that match the refactoring pattern contained in R and transfer it to  $S'_2$ . We replace the genes of  $S_2$  in  $S'_2$  that are not used by any patterns contained in  $S'_2$  for other objectives for which  $S_2$  has a higher value in comparison to  $S_1$ . We do the same for all the objectives in  $O_1 \cap O_2$ . This crossover strategy allows us to keep the strengths and fix the weaknesses of the parents in the next generation while conserving the personalization aspect and practical abilities of the solutions.



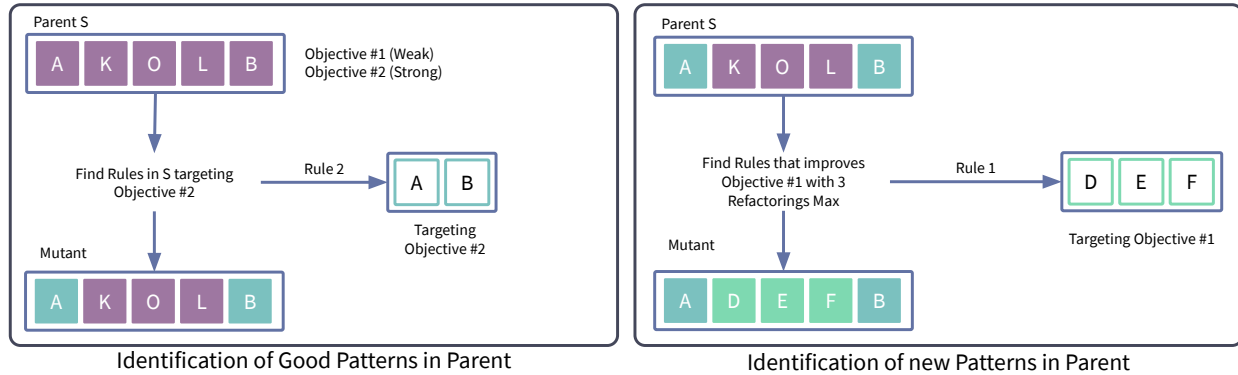
**Figure 4.35:** An illustration of X-SBR crossover

**4.4.2.4.3 Mutation** Mutation is a genetic operator used to preserve genetic diversity from one generation to the next in a genetic algorithm. Mutation involves a change in

chromosome structure by altering one or more genes in a chromosome. It occurs according to a user-definable mutation probability. In our study, we set this probability to 0.1 . Figure 4.36 is a simplified illustration of how our improved mutation works. For each solution S, we randomly select a floating-point value. If this value is less than the mutation probability, we follow the steps below:

- We use the Apriori rules to find the refactoring patterns in S that improve one or more objectives. For example, in Figure 4.36, Rule 2 improves Objective 2.
- We deduce the refactorings that are not associated with any pattern. In figure 4.36, the refactorings that are not associated with any objective are K, O, and L.
- We look for the rules that improve the weakest objective of S (i.e., the objective with the worst value). In figure 4.36, the weakest objective is Objective 1 which can be improved using Rule 1.
- We choose the refactoring pattern that modifies the maximum number of refactorings that are not associated with any objective and we add it to S. In figure 4.36, Rule 1 is composed of three refactorings that can replace the three refactorings that are not associated with any pattern (e.g. K, O, and L)
- If no rules are found, we choose a random number N between 1 and half the size of S and we randomly modify N refactorings in S from the possible refactoring operations that the tool supports.

**4.4.2.4.4 Explanations Generation** The implemented tool includes several features to understand the explanations. First, the impact of the refactoring on the quality can be visualized to the developers via bar charts by showing the delta between before and after refactorings. Second, the extracted refactoring patterns are represented as dependencies tree to the developer and s/he can visualize the impact of each of the refactorings in the



**Figure 4.36:** An illustration of X-SBR mutation

tree on the quality improvements or deteriorations. Third, the user can select any of the refactorings in the sequence and can get the pattern (other dependent refactorings with a significant impact on some of the quality metrics) associated with it to explain the relevance of that refactoring.

### 4.4.3 Experiment and Results

#### 4.4.3.1 Research Questions

In this study, we defined three main research questions.

**RQ1:** *To what extent can X-SBR generate good refactoring solutions compared to multi-objective refactoring techniques?*

**RQ2:** *To what extent can X-SBR reduce the number of invalid refactorings compared to multi-objective refactoring techniques?*

**RQ3:** *To what extent can X-SBR provide relevant solutions and explanations compared to the state of the art refactoring techniques?*

To answer RQ1, we collected the source code of 711 commits from 18 open-source systems. We performed static analysis on the code to compute low- and high-level code quality metrics. Then, we used RMiner [624] to detect the refactoring operations performed between the commits. Our dataset can be found in the appendix website [625]. After that,

we used the Apriori algorithm [539] to generate association rules that link design metrics and refactoring operations with the QMOOD quality metrics. Then, we used these rules to choose strategically the initial population and improve the change operators of the traditional NSGA-II [8]. The rules are used to favor good patterns of the solutions and penalize bad ones.

To evaluate the efficiency of our algorithm, we selected four systems described in Table 4.12 since they are used in existing refactoring benchmark [8] and the participants of our study are familiar with them (RQ3). We compared four NSGA-II variations that optimize the same quality objectives: (1) traditional NSGA-II (Mkaouer et al.[8]), (2) NSGA-II with an improved initial population strategy, (3) NSGA-II with improved change operators, and (4) NSGA-II with improved change operators and initial population strategy (*X-SBR*). To ensure a fair comparison, we only limited the baseline to these four techniques since our proposal is a variation of the work of Mkaouer et al.[8]. However, we extended our baseline in RQ3 when evaluating the relevance of the refactoring recommendations.

**Table 4.12:** Systems considered for validation

<b>System</b>	<b>Release</b>	<b># of Classes</b>	<b>KLOC</b>
ArgoUML	v0.3	1358	114
JHotDraw	v7.5.1	585	25
GanttProject	v1.11.1	245	49
Apache Ant	v1.8.2	1191	112

To answer RQ2, we computed the number of conflicts in the solutions generated by the four algorithms mentioned above (RQ1) on the four systems listed in Table 4.12. For that, we calculated the number of invalid refactorings in each solution of the Pareto fronts by checking the validity of pre-and post-conditions of each refactoring operation.

To answer RQ3, we present to developers those association rules that lead to the generation of each refactoring solution in the Pareto front and their frequencies. Since the association rules are hard to understand if they are presented as the explanation for the recommended refactorings, we implemented a user-friendly interface in our refactoring webapp that can

highlight the code locations and metrics associated with the recommended refactorings. To validate the usefulness of our explanations, we conducted a survey with a group of 14 active programmers to identify and manually evaluate the relevance of the refactorings that they found using *X-SBR*.

Since the manual validation is limited to 14 participants, we considered another evaluation which is based on the percentage of fixed code smells ( $NF$ ) by the refactoring solution. The detection of code smells after applying a refactoring solution is performed using the detection rules of [601]. The detection of code smells is subjective and some developers prefer not to fix some smells because the code is stable or some of them are not important to fix. To this end, we considered another metric based on QMOOD that estimates the quality improvement of the system by comparing the quality before and after refactoring independently from the number of fixed design defects. Based on the two above metrics, we can evaluate the different approaches without the need of developers evaluation. The baseline to answer RQ3 includes the different existing multi-objective techniques [5, 6, 7, 8] and also a tool, called JDeodorant [9], not based on heuristic search. All the selected search-based refactoring techniques for the baseline of RQ2 are based on multi-objective search but using different fitness functions and solution representation which may confirm if good refactoring recommendations are actually due our knowledge-based component and not the design of the algorithm. The current version of JDeodorant is implemented as an Eclipse plug-in that identifies some types of design defects using quality metrics and then proposes a list of refactoring strategies to fix them. For the comparison with JDeodorant, we limited the comparison to the same refactoring types supported by both X-SBR and JDeodorant.

#### 4.4.3.2 Evaluation Metrics

To address the three research questions described in the introduction section, we defined the following metrics and applied them on a data set, described in the next subsection. For RQ1, we generated association rules that link design metrics and refactoring operations with

QMOOD metrics. To evaluate these rules, we computed support, confidence, and lift [626].

**Support:** Support reflects how frequently the item set appears in the dataset. In our problem, it is defined as the ratio of the classes that contain  $D \cup R \cup Q$  to the total number of classes in the dataset where D is a set of design metrics intervals, R is a set of refactoring operations and Q is a set of QMOOD intervals.

$$support(D, R \Rightarrow Q) = P(D \cup R \cup Q) \quad (4.12)$$

where  $P(D \cup R \cup Q)$  is the probability of cases containing D, R and Q all in the same transaction.

**Confidence:** Confidence reveals how often the rule has been considered to be correct. In our approach, confidence is defined as the ratio of the number of classes that contain  $D \cup R \cup Q$  to the number of classes that contain  $D \cup R$ . It evaluates the strength of a rule. The higher the confidence the more likely it is for Q to be present in transactions that contain  $D \cup R$ .

$$\begin{aligned} confidence(D, R \Rightarrow Q) &= P(Q|D \cup R) \\ &= P(D \cup R \cup Q)P(D \cup R) \end{aligned} \quad (4.13)$$

**Lift:** Lift is defined as the confidence of the rule divided by the expected level of confidence. A lift value higher than 1 means that there is a positive correlation between  $D \cup R$  and Q. If the lift is smaller than 1, it means that  $D \cup R$  is negatively correlated with Q. A lift value almost equal to 1 means that we cannot say anything about the correlation of  $D \cup R$  and Q.

$$\begin{aligned} lift(D, R \Rightarrow Q) &= confidence(D, R \Rightarrow Q)P(Q) \\ &= P(D \cup R \cup Q)P(D \cup R) * P(Q) \end{aligned} \quad (4.14)$$



To evaluate the quality of solution sets obtained by all four algorithms mentioned above, we used the following three metrics as performance indicators:

- *Contributions* ( $I_C$ ) [610]: It measures the proportion of solutions that lie on the reference front RS (i.e., best know approximation set, computed as the non-dominated elements of all known solutions) [611]. The higher this proportion the better is the quality of the solutions.
- *Hypervolume* ( $I_{HV}$ ) [613]: It computes the volume covered by members of a non-dominated set of solutions in the objective space. A higher value of hypervolume is desirable, as it demonstrates better spread and convergence of solutions.
- *Inverted Generational Distance* ( $I_{GD}$ ) [612]: It computes the average Euclidean distance in the objective space between each solution in the Pareto front and its closest point in the reference front RS. Small values are desirable.

For RQ2, we want to estimate the feasibility of the solutions generated by the four algorithms. For that, we compute the number of invalid refactorings in each solution of the Pareto fronts by checking the validity of pre-and post-conditions of each refactoring operation. These conditions are discussed by Opdyke et al. [33].

For RQ3, the goal is to validate the refactoring solutions generated by *X-SBR* from both quantitative and qualitative perspectives and compare them with those generated with baseline. For the quantitative validation, we calculated precision and recall scores to compare between refactorings suggested by *X-SBR* and those expected based on the participants assessment. We also did the same using the tools of the baseline.

$$Precision = \frac{\text{X-SBR solutions} \cap \text{Expected Refactorings}}{\text{X-SBR solutions}} \quad (4.15)$$

$$Recall = \frac{\text{X-SBR solutions} \cap \text{Expected Refactorings}}{\text{Expected Refactorings}} \quad (4.16)$$

For the qualitative validation, we asked the participants to assign 0 or 1 to every refactoring of the solutions generated by both tools. A 0 means that the refactoring is not applicable and inconsistent with the source code; 1 means that the refactoring is meaningful and relevant. We computed manual correctness which is defined as the number of meaningful refactorings divided by the total number of recommended refactorings.

$$\text{Manual Correctness} = \frac{|\text{Meaningful Refactorings}|}{|\text{Recommended Refactorings}|} \quad (4.17)$$

We have also calculate the number of code smells fixed by the recommended refactorings. Formally,  $NF$  is defined as:

$$NF = \frac{\#fixed\ code\ smells}{\#code\ smells} \in [0, 1] \quad (4.18)$$

The gain for each of the considered QMOOD quality attributes and the average total gain in quality after refactoring can be easily estimated as:

$$G = \frac{\sum_{i=1}^6 G_{q_i}}{6} \text{ and } G_{q_i} = q'_i - q_i \quad (4.19)$$

where  $q'_i$  and  $q_i$  represents the value of the QMOOD quality attribute  $i$  after and before refactoring, respectively.

We finally asked the participants to evaluate the rules that are intended to explain the creation of the Pareto front solutions. For that, we randomly picked between 2 and 5 refactoring solutions per system and their explanations. Then, we asked them to assign a grade on a Likert scale of 1-5, 1 being the lowest (not relevant), 5 being the highest (very relevant) to every rule to indicate how helpful it is in explaining the creation and relevance of the refactoring solution.

#### 4.4.3.3 Parameters Tuning

Parameters setting plays an important role in the performance of a search-based algorithm. We have used one of the most efficient and popular approach for parameter setting of evolutionary algorithms which is Design of Experiments (DoE) [627]. Each parameter has been uniformly discretized in some intervals. Values from each interval have been tested for our application. Finally we pick the best values for all parameters. Hence, a reasonable set of parameter values have been experimented. We picked the combination based on the number of evaluations without improvement and convergence of the population. We tried to find a balance between wide exploration and deep exploitation during the evolutionary process. In order to ensure a fair comparison of the results of the four algorithms, we performed the same number of evaluations per run and used the same sizes for the initial population. We ended up by choosing 100 for the initial population and 10 000 for the maximum number of evaluations (the stopping criterion). We did not chose the execution time as a stopping criterion because it is known in the computational intelligence field that execution time is not suitable to ensure a fair comparison as it is very sensitive to the used hardware resources. The crossover and mutation probabilities are set to 0.8 and 0.1 respectively.

Because of the stochastic nature of the used meta-heuristic algorithms, different runs of the same algorithm solving the same problem typically lead to different results. For this reason, we performed 30 runs for each algorithm and each project to make sure that the results are statistically significant. For each evaluation metric, we used the Wilcoxon rank sum test [628] in a pairwise fashion in order to detect significant performance differences between the algorithms (X-SBR vs each of the competitors) under comparison based on 30 independent runs as recommended by existing guidelines [602].

We found that all the results based on the different measures were statistically significant on 30 independent runs using the Wilcoxon test with a 95% confidence level ( $\alpha < 5\%$ ). The p-values of the pairwise analysis were lower than 0.01 in all cases. We have also calculated Eta squared ( $\eta^2$ ) [629] which is a measure of the effect size (strength of association) and it

estimates the degree of association between the independent factor and dependent variable for the sample. Eta squared is the proportion of the total variance that is attributed to a factor (the “refactoring methods” in this study). Table 4.13 reports Eta squared values for each pair of software projects and metrics. These values shows to what extent different algorithms are the cause of variability of the metrics.

**Table 4.13:** Effect Size values (Eta squared ( $\eta^2$ )) for corresponding software project and metric.

System	G	NF	MC	PR	RC
ApacheAnt	0.898	0.919	0.924	0.936	0.924
GanttProject	0.873	0.902	0.946	0.931	0.962
JHotDraw	0.826	0.903	0.918	0.836	0.962
ArgoUML	0.813	0.842	0.931	0.901	0.951

#### 4.4.3.4 Subjects

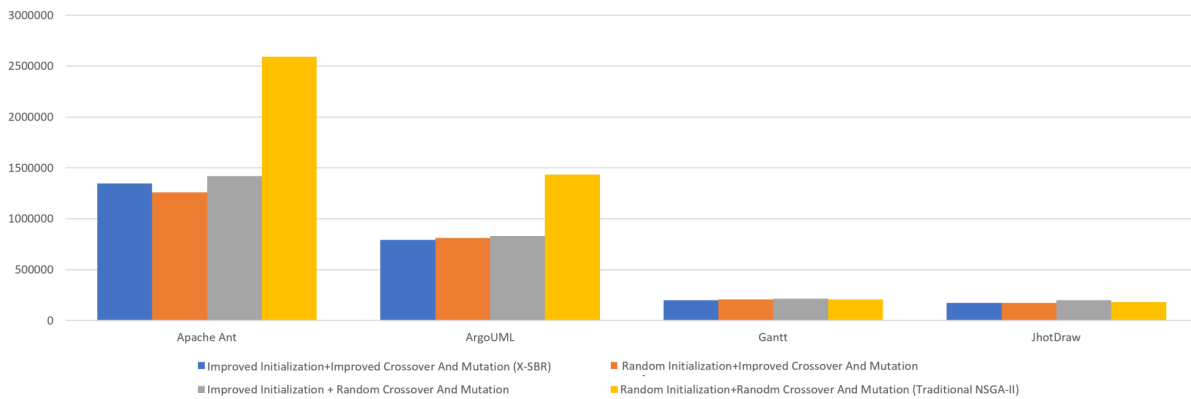
We selected 14 participants to evaluate *X-SBR* on the 4 systems described in Table 4.12. We carefully selected them to make sure that they extensively applied refactorings during their previous experiences in development and also used the open source systems extensively in their previous and current projects in industry. They had to fill a pre-study survey that collects background information on them such as their programming experience, their role within their companies etc. The details of the selected participants and the projects they evaluated can be found in Table 4.14 (the depicted values averages across the four participants in each row). To improve the survey outcome, we organized a two-hour lecture about software quality assessment in general and refactoring in particular. We also presented a demo for *X-SBR* and gave them enough time to explore and test the tool themselves. We tested the trustfulness of participants and their knowledge on both the open source systems and refactoring beforehand by asking them to pass ten tests to evaluate their performance in evaluating and suggesting refactoring solutions. Each participant was asked to assess the meaningfulness and relevance of the refactorings recommended using our tool and all the four systems. The participants were shown recommendations created by the authors’ approach as well as by the baseline, but without knowing which recommendations came from which

approach.

**Table 4.14:** Participants details

System	#Subjects	Avg. prog. experience (years)	Refactoring experience
ArgoUML	4	10	High
JHotDraw	4	11.5	Very High
GanttProject	4	10.5	High
Apache Ant	4	12	Very High

#### 4.4.3.5 Results



**Figure 4.37:** Average execution time (ms) of all algorithms using the four systems

**Table 4.15:** Evaluation metrics and statistics of the rules

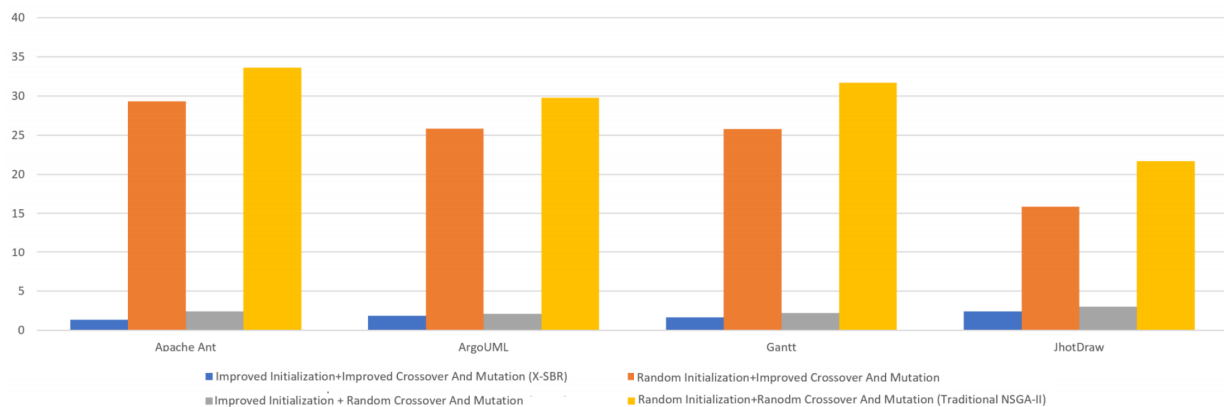
Evaluation Metric	Mean	Max	Min
Support	0.945	0.986	0.935
Confidence	0.986	0.992	0.959
Lift	1.000	1.002	0.999

**4.4.3.5.1 Results for RQ1** We generated a total of 3097 association rules that link the design metrics and refactoring operations with the QMOOD quality metrics. Figure 4.33 shows an example of a rule created by the Apriori algorithm. The complete list can be found in our online appendix [606]. Table 4.15 contains the average, max and min support, confidence and lift of all the rules. The minimum support, confidence and lift are 0.935, 0.959 and 0.999, respectively. This confirms the strong correlation between design metrics,

refactoring operations and the QMOOD metrics. After that, we compared the execution time of the four algorithms: (1) traditional NSGA-II (Mkaouer et al.[8]), (2) NSGA-II with an improved initial population strategy, (3) NSGA-II with improved change operators, and (4) NSGA-II with improved change operators and initial population strategy (*X-SBR*). Figure 4.37 shows the average time spent to run the four systems 30 times. In small systems (e.g. Gantt and JhotDraw), the four algorithms have almost the same execution time. X-SBR outperforms the other variations with a slight difference. However, when dealing with large systems (e.g. Apache Ant and ArgoUML), the traditional NSGA-II [8] has the highest execution time which is expected since both, the initialization and change operators, are done randomly without any guidance. This confirms the usefulness of our strategy of guiding the creation of solutions towards the construction of good refactoring patterns. The other three variations performed clearly better than the the traditional NSGA-II [8]. The difference in performance is more noticeable in large systems than in small systems because the execution time of the improved algorithms include the running time of the Apriori algorithm. Thus, the running time of the Apriori algorithm is compensated when we are dealing with a large number of classes by removing excessive diversity from the search space. Table 4.16 shows the mean, min and max of the Hypervolume ( $I_{HV}$ ) and Generational Distance ( $I_{GD}$ ) Indicators of all algorithms using the four systems. Table 4.17 contains the results of the Contribution ( $I_C$ ) metric of the three modified algorithms compared to the traditional NSGA-II [8]. For each performance indicator, we highlighted in bold the best min/max/average values. Please note that the Contributions ( $I_C$ ) and the Hypervolume ( $I_{HV}$ ) are to be maximized and the Generational Distance ( $I_{GD}$ ) is to be minimized. All these indicators show that the traditional NSGA-II exhibits more diversity in the solutions than other algorithms. This observation is expected as the traditional NSGA-II relies on randomness when generating the solutions, unlike the modified versions where the creation of solutions is guided towards the construction of good refactoring patterns based on the Apriori rules. It is important to note that excessive diversity can diverge the algorithm

from generating good quality solutions due to the large search space and infinite number of possible combinations. In other words, we can end up having a diverse Pareto front but with many infeasible refactoring solutions. Therefore, it is necessary to have a strategy to push the algorithm towards creating correct solutions. However, guiding the algorithm too much might also hurt the exploration. Maintaining diversity is one important aim of multi-objective optimization. When clear user preferences are not available, it is highly desirable that a large number of solutions can be obtained that uniformly spread over the whole Pareto front and are as diverse as possible. However, we want to stay away from excessive diversity that leads the algorithm to diverge from generating good quality solutions due to the large search space and infinite number of possible combinations. On the other hand, selection pressure pushes the algorithm to focus more and more on the already discovered better performing regions in the search space and as a result population diversity declines, gradually reaching a homogeneous state. Through our approach, we are trying to maintain an optimal level of diversity in the population to ensure that progress of the search algorithm is unhindered by premature convergence to suboptimal solutions.

**Key findings:** The variants of NSGA-II with random initialization and/or genetic operators demonstrate higher diversity than *X-SBR* but the difference is small. *X-SBR* outperforms the other variations in terms of execution time, especially with large systems.



**Figure 4.38:** Average number of invalid refactorings in the solutions of all algorithms using the four systems

**4.4.3.5.2 Results for RQ2** Figure 4.38 shows the average number of invalid refactorings in the solutions of the Pareto front in all four systems using the different algorithms. We would like to point out that all algorithms have the same number of non-dominated solutions in the final Pareto front which is equal to 100. The traditional NSGA-II and NSGA-II with random initialization and improved change operators had the largest number of invalid refactorings in their Pareto front with values exceeding 15 invalid refactorings. The lowest number of invalid refactorings was achieved by X-SBR. The latter algorithms had less than four invalid refactorings in their Pareto fronts. The reason why the combination of the random initialization and the random or improved crossover produce a significant number of invalid refactorings is that the new crossover and mutation operators care more about improving the QMOOD quality metrics rather than checking the correctness of refactorings. However, this problem is mitigated by initializing the gene pool with valid chromosomes based on mining the refactoring history of several projects. This can be observed by the reduced number of infeasible refactorings in the solutions generated by the improved initialization method when combined with either the random or improved change operators.

**🔍 Key findings:** Based on the results of RQ1 and RQ2, *X-SBR* was able to achieve a better quality of solutions in comparison to the traditional NSGA-II with small sacrifices in terms of diversity and execution time.

**Table 4.16:** Results of the Hypervolume ( $I_{HV}$ ) and Generational Distance ( $I_{GD}$ ) indicators

System	Algorithm	Hypervolume ( $I_{HV}$ )			Generational Distance ( $I_{GD}$ )		
		Average	Min	Max	Average	Min	Max
Apache Ant	Improved Initialization + Random Crossover And Mutation	0.680742	0.432318	0.935184	0.015524	0.010209	0.020846
Apache Ant	Random Initialization+Random Crossover And Mutation (Traditional NSGA-II)	0.693186	0.398396	1.117279	0.031465	0.00819	0.051153
Apache Ant	Improved Initialization+Improved Crossover And Mutation ( <i>X-SBR</i> )	0.499433	0.293363	1.124596	0.064079	0.002978	0.093633
Apache Ant	Random Initialization+Improved Crossover And Mutation	0.809312	0.485615	1.085356	0.019873	0.008611	0.037001
ArgoUML	Improved Initialization + Random Crossover And Mutation	0.642199	0.404763	0.857439	0.024575	0.00818	0.034475
ArgoUML	Random Initialization+Random Crossover And Mutation (Traditional NSGA-II)	0.777583	0.52648	1.112845	0.03008	0.002679	0.047454
ArgoUML	Improved Initialization+Improved Crossover And Mutation ( <i>X-SBR</i> )	0.641947	0.444483	1.136336	0.044481	0.002322	0.057297
ArgoUML	Random Initialization+Improved Crossover And Mutation	0.690078	0.444543	1.141642	0.041118	0.005496	0.055032
GanttProject	Improved Initialization + Random Crossover And Mutation	0.68693	0.566786	0.907115	0.021973	0.012951	0.029777
GanttProject	Random Initialization+Random Crossover And Mutation (Traditional NSGA-II)	0.861142	0.666087	1.133707	0.022668	0.002095	0.032585
GanttProject	Improved Initialization+Improved Crossover And Mutation ( <i>X-SBR</i> )	0.782098	0.626555	0.978024	0.022406	0.011344	0.02651
GanttProject	Random Initialization+Improved Crossover And Mutation	0.776723	0.655532	1.242082	0.022349	0.006756	0.029976
JhotDraw	Improved Initialization + Random Crossover And Mutation	0.771315	0.588192	1.33879	0.04945	0.000903	0.071506
JhotDraw	Random Initialization+Random Crossover And Mutation (Traditional NSGA-II)	0.933738	0.555179	1.281501	0.026886	0.007105	0.056431
JhotDraw	Improved Initialization+Improved Crossover And Mutation ( <i>X-SBR</i> )	0.564916	0.393056	1.083154	0.08246	0.024441	0.20624
JhotDraw	Random Initialization+Improved Crossover And Mutation	0.756657	0.592614	1.217705	0.052932	0.006605	0.072263



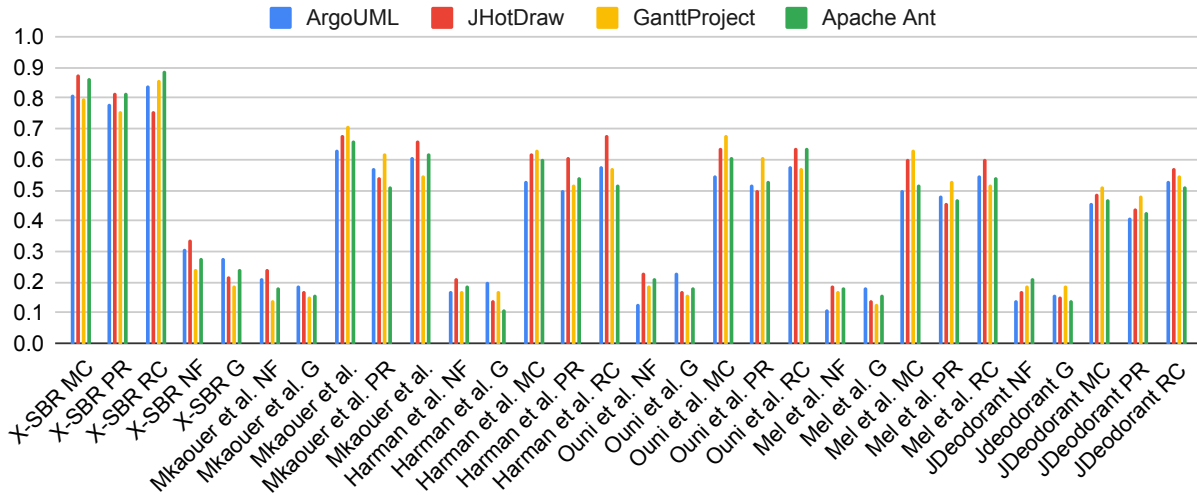
**Table 4.17:** Results of the Contributions ( $I_C$ ) metric

Algorithms	Contribution value
Contribution of NSGA-II with random initialization + improved change operators to traditional NSGA-II	0.34030526
Contribution of NSGA-II with improved initialization + random change operators to traditional NSGA-II	0.247601151
Contribution of NSGA-II with improved initialization + improved change operators to traditional NSGA-II	0.241613462

**4.4.3.5.3 Results for RQ3** We summarize in the following the feedback of the developers based on the survey. Figure 4.39 contains the results of the manual correctness, precision and recall of both our tool (*X-SBR*) and the state of the refactoring techniques. *X-SBR* was able to achieve better scores than [8] and existing approaches in all the previous metrics for all systems. The average manual correctness, precision and recall of our tool compared to that of Mkaouer et al. [8] are 0.839, 0.795, and 0.83 to 0.67, 0.56, and 0.67 respectively and much better than the remaining tools. The participants also found our refactoring recommendations applicable and consistent with the source codes and their design issues.

Figure 4.40 summarizes what the participants think about the explanations provided by *X-SBR*. For all the four systems, more than 85% of the rules are judged relevant (score 4) and very relevant (score 5). Only less than 3% of the rules were judged not relevant (score 1). They mentioned that *X-SBR* provided trust, clarity and understanding compared to existing refactoring tools. They highlighted that the black-box nature of existing refactoring tools, giving results without a reason, is hindering them from adopting their refactoring recommendations. According to them, this obstacle is alleviated by our proposed approach.

**🔍 Key findings:** *X-SBR* provided more relevant and meaningful refactorings than the state of the art refactoring techniques and helped the participants understand why and how the solutions are generated which boosted their trust in the refactoring tool.



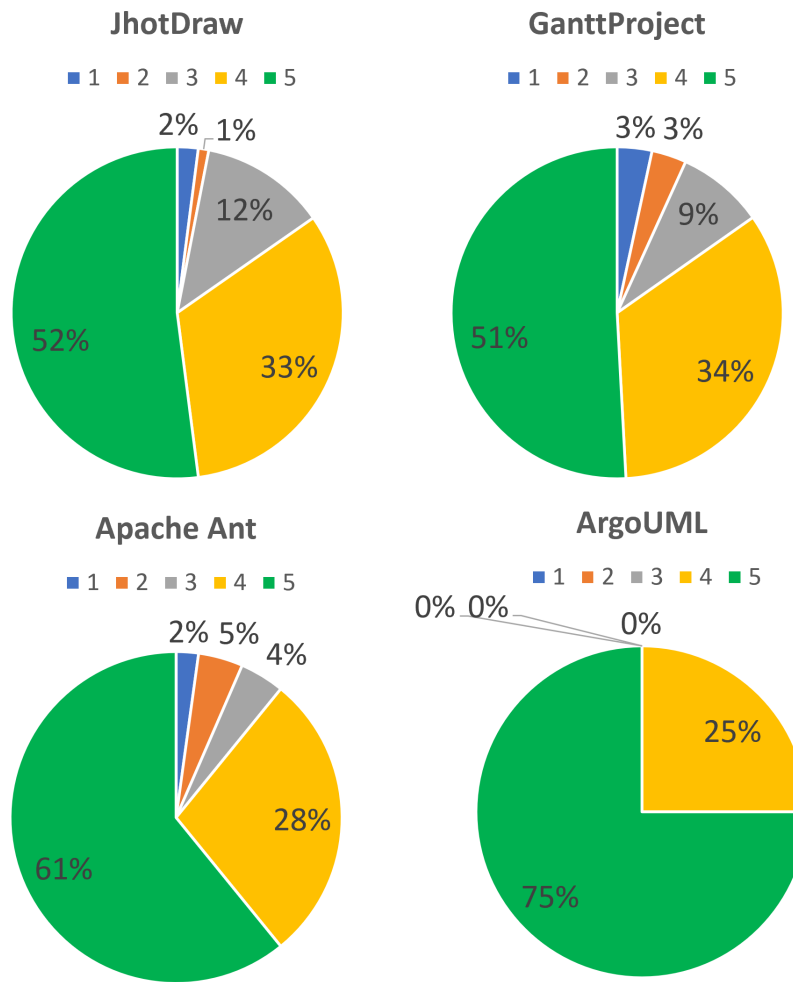
**Figure 4.39:** Automated and manual evaluation of refactoring recommendations generated by the different refactoring tools

#### 4.4.4 Threats to Validity

**Conclusion validity.** The parameter tuning of the different search-based algorithms used in our experiments creates an internal threat that needs to be evaluated in our future work. The parameters' values used in our experiments were found by trial-and-error [630].

**Internal validity.** The variation of correctness and speed between the different groups when using our approach and other tools is one potential internal threat. In fact, our approach may not be the only reason for the superior performance because the participants have different programming skills and familiarity with refactoring tools. To counteract this, we assigned the developers to different groups according to their programming experience so as to reduce the gap between the different groups and we also adapted a counter-balanced design.

**Construct validity.** The different developers involved in our experiments may have divergent opinions about the recommended refactorings in terms of relevance which may impact our results. Almost all of our industrial collaborators in the refactoring area are selecting major refactoring strategies based on discussions between the architects to adopt the best alternative. For the selection threat, the participant diversity in terms of experience could affect the results of our study. We addressed the selection threat by giving a lecture and



**Figure 4.40:** Distribution of the relevance of the explanations according to the survey results (1=not relevant-5=very relevant)

tests.

**External threats.** We used 18 projects to generate the association rules. To mitigate these threats, we used projects of different sizes and domains. Moreover, we only included four projects in our validation. The reason behind that is, first, to attract the most amount of responses with good quality from participants in our survey. The more tedious the task that the participant must complete the less the quality of their input is. The second reason is the long execution time due to running all of the four algorithms on all of the systems 30 times.

#### 4.4.5 Conclusion

Existing refactoring tools lack adaptability and explainability towards the developers. As a result, developers seem to be more inclined to abandon them and make changes by hand. We propose in this study, *X-SBR*, an enhanced knowledge-informed multi-objective search algorithm to provide personalized and relevant refactoring recommendations. *X-SBR* implements new initial population and change operators methods using the refactoring and quality history of 18 projects and provides explanations regarding why and how the solutions are formed and impacted the fitness functions. Based on our quantitative and qualitative validation using four open-source systems, our tool was able to achieve more relevant refactoring solutions than existing refactoring techniques with a small sacrifice in terms of diversity and execution time. The results of the survey conducted with 14 software developers provide strong evidence that our tool improves the quality of refactoring solutions and helps developers understand, appropriately trust, and effectively manage the refactoring process.

There are multiple ways within which this work can be expanded upon. First, we believe it's a natural step to validate our work with additional programming languages, developers, projects, and quality metrics in order to draw conclusions about the general applicability of our methodology. Second, we intend to try out other algorithms for frequent item-set mining, beyond the Apriori Algorithm, to extend our empirical validation. Third, we think that

adding support for more quality metrics and other fine-grained refactoring operations, such as Decompose Conditional, Replace Conditional with Polymorphism, and Replace Type Code with State/Strategy can prove an interesting addition and extension of our work. Fourth, we are planning to validate the change operators with other evolutionary algorithms such as a many-objective variant of MOEA/D, Global WASF-GA, and/or RVEA. We clarified this in the conclusion section. Last but not least, using code smell history and bug reports in addition to or in place of Low Level metrics when generating association rules can be an interesting future research direction

## CHAPTER V

### Conclusion

The features and improvements that were delivered in this dissertation and the results that were achieved are summarized in this chapter. In addition, the suggested possible improvements to the proposed contributions are discussed.

#### Summary

In **Chapter I** and **Chapter II**, we defined the research context and the challenges, the contributions of this thesis, required background, and state-of-the-art and related works to our approaches.

In **Chapter II Section 2.1**, we have conducted a systematic literature review on refactoring accompanied by meta-analysis to answer the defined research questions. After a comprehensive search that follows a systematic series of steps and assessing the quality of the studies, 3183 publications were identified. Based on these selected papers, we derived a taxonomy focused on five key aspects of Refactoring: refactoring life-cycle, artifacts affected by refactoring, refactoring objectives, refactoring techniques, and refactoring evaluation. Using this classification scheme, we analyzed the primary studies and presented the results in a way that enables researchers to relate their work to the current body of knowledge and identify future research directions. We also implemented a repository that helps researchers/practitioners collect and report papers about Refactoring. It also provides visualization charts and graphs that highlight the analysis results of our selected studies. This infrastructure will bridge the gap among the different refactoring communities and allow for more

effortless knowledge transfer. The results of our systematic review will help both researchers and practitioners to understand the current status of the field, structuring it, and identify potential gaps. Since we expect this research area to continue to grow in the future, the proposed repository and taxonomy will continue to be updated by the organizers of this study and the community to include new approaches, tools and researchers.

In **Chapter II Section 2.2** , we performed the first large scale refactoring study on the most popular online Q&A forums for developers, Stack Overflow. We used 89 tags to extract 105463 questions about refactoring. We used the Latent Dirichlet Allocation (LDA) technique to generate the discussed topics in this repository. We found 6 main topics which are "Creational pattern", "Parallel programming", "Models refactor", "Mobile/UI", "SOA", and "Design pattern". The analysis of these topics provided various key insights about the interests of developers related to refactoring such as the most addressed quality issues, the domains where refactoring is extensively discussed, the widely addressed anti-patterns, and patterns. We have also investigated how the interests of developers on refactoring topics change over the years.

In the context of improving the identification of potential refactoring opportunities, we proposed, in **Chapter III Section 3.1**, a novel framework, *QS-URec*, to detect files responsible for quality and security issues based on user reviews and source code metrics. We evaluated our approach on 50 popular mobile apps from *Google Play* with 290,000 reviews along with a large and popular mobile app provided by our industrial partner. Our results demonstrate strong correlations between several security and quality metrics and user ratings. *QS-URec* outperforms an existing textual analysis technique in terms of precision and recall when linking emerging quality and security app issues to relevant files to be fixed or refactored. We conducted experiments and a brief survey with the original developers of *My-FitnessPal* that supported the effectiveness of *QS-URec* and the importance of considering user reviews to prioritize and fix security and quality issues.

In **Chapter III Section 3.2**, we propose, in this paper, a novel approach to predict QoS

with the least cost using code/interface quality metrics and antipatterns. The output of our approach consists of 10 association rules that predict the performance of web services. We used 5 fold cross validation to evaluate the rules. The obtained results based on 707 web services confirm the correlation between both code/interface metrics/antipatterns and the QoS attributes. This important outcome can be used to understand the severity of antipatterns and predict the quality of the services based on the current quality of the implementation. Our results show that data service, chatty service and multi service are the most severe antipatterns types on the quality of service attributes among the studied antipatterns. All the QMOOD metrics are affected by antipatterns at different levels. Best practices, availability and compliance are the quality metrics deteriorated the most by antipatterns.

In **Section 3.3** of the same Chapter, we propose a novel approach that aims at finding the most suitable benchmark to evaluate the quality of a software project in a fair and unbiased way. We first showed that clustering algorithms are efficient in finding clusters of projects with distinct characteristics based on repository features. We then performed statistical analysis to compare the different clusters and to check the sensitivity of each quality metric. We finally validated our approach with developers from eBay. The results provide strong evidence that our approach helps developers automate and effectively manage the benchmarking process for software quality assessment.

Regarding improving the generation of refactoring recommendations, we propose, in **Chapter IV Section 4.1**, an empirical study to validate the correlations between the QMOOD quality attributes [515] and a set of security metrics [466, 468] and to understand the correlations between refactoring types and security metrics. Based on the outcomes of these studies, we proposed a security-aware multi-objective refactoring approach to find a balance between quality and security goals. We evaluated our tool on the same projects used for the empirical validations. Furthermore, we compared our results to an existing refactoring work not considering security to understand the sacrifice in security measures when improving the quality. The comparison shows that our security-aware approach performed



significantly better than the existing approach when it comes to preserving and improving the security of the system but with low cost in terms of sacrificing quality. The survey with the 15 practitioners confirmed the efficiency of our tool and the importance of considering security while improving several quality attributes.

In **Chapter IV Section 4.2**, we presented an approach to recommend refactorings for security critical files while concurrently improving the code quality of a software project. We used the history of vulnerabilities and security bug reports along with a selected set of keywords [468, 472] to automatically identify security-critical files in a project based on source code, bug reports, pull-request descriptions and commit messages. After identifying these security-related files we estimated their risk based on static analysis to check their coupling with other components of the project. Then, our approach recommended refactorings to prioritize fixing quality issues in these security-critical files to improve code quality measures and remove code smells using multi-objective search. We evaluated our approach on six open source projects and one industrial system to check the relevance of our refactoring recommendations. Our results confirm the effectiveness of our approach as compared to existing refactoring approaches.

To improve the correctness and quality of refactoring recommendations and increase developer trust in search-based refactoring recommendation tools, we proposed, in **Chapter IV Section 4.3**, a dependency-aware multi-objective refactoring approach with intelligent change operators that find a balance among quality objectives while reducing the number of invalid refactorings. We evaluated this approach on four open-source projects. We compared our results to existing refactoring techniques that use random change operators, as well as to a dependency-aware technique, to understand the impact of considering refactoring dependencies and fixing quality weaknesses in refactoring solutions. The comparisons show that our proposed approach performs significantly better than the baselines in terms of convergence, diversity, and correctness with a reasonable cost in terms of increased execution time. The survey with 14 practitioners confirmed the relevance of our approach.

Finally, in **Chapter IV Section 4.4**, we propose, *X-SBR*, an enhanced knowledge-informed multi-objective search algorithm to provide personalized and relevant refactoring recommendations. *X-SBR* implements new initial population and change operators methods using the refactoring and quality history of 18 projects and provides explanations regarding why and how the solutions are formed and impacted the fitness functions. Based on our quantitative and qualitative validation using four open-source systems, our tool was able to achieve more relevant refactoring solutions than existing refactoring techniques with a small sacrifice in terms of diversity and execution time. The results of the survey conducted with 14 software developers provide strong evidence that our tool improves the quality of refactoring solutions and helps developers understand, appropriately trust, and effectively manage the refactoring process.

## 5.1 Future Work

In the context of refactoring recommendation, we plan to investigate how many refactoring operations actually depend on each other and how many operations can be executed independently. We are also planning to explore further techniques to implement the change operators and compare them with each other. In fact, there are multiple ways of how we choose the refactorings that participate in the mutation and the crossover processes as well as how we perform the change operators. There is a practical balance to study between smarter mutations (more expensive, but more reliable) vs. simpler, more error prone mutations (faster, but not guaranteed). This operation shows benefits from introducing modest constraints in the selection of mutable refactorings. There are certainly other variants that explore different trade-offs between speed and error reduction and that is just for mutating a single refactoring at a time.

In the context of benchmarking GitHub repositories, even when considering systems with similar repository metrics (number of contributors, number of classes, number of commits, etc.), they may still have very different quality profiles. In other words, developers may have

different possible quality targets based on the used benchmark of similar projects. These targets might be clearly associated with different refactoring costs as well to reach them. Therefore, it could be interesting to investigate the quality profiles (e.g. possible common quality patterns) included in each of the repository clusters by performing another clustering inside each of the repository clusters but this time based on quality metrics. Furthermore, we can implement a search-based refactoring technique that estimates the refactoring cost needed to move from one quality profile to another by finding the refactorings sequence.

## BIBLIOGRAPHY

- [1] F. Palomba, P. Salza, A. Ciurumelea, S. Panichella, H. Gall, F. Ferrucci, and A. De Lucia, “Recommending and localizing change requests for mobile apps based on user reviews,” in *Proceedings of the 39th international conference on software engineering*, pp. 106–117, IEEE Press, 2017.
- [2] 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.
- [3] TSE, “Online appendix for this publication,” 2020. <https://doi.org/10.7302/0bgnvt27>.
- [4] “Nuovo cms.” <https://www.cvedetails.com/cve/CVE-2018-17890/>.
- [5] 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*, pp. 1106–1113, 2007.
- [6] M. O’Keeffe and M. O. Cinnéide, “Search-based refactoring for software maintenance,” *Journal of Systems and Software*, vol. 81, no. 4, pp. 502–516, 2008.
- [7] 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 (TOSEM)*, vol. 25, no. 3, pp. 1–53, 2016.
- [8] M. W. Mkaouer, M. Kessentini, S. Bechikh, M. Ó. Cinnéide, and K. Deb, “On the use of many quality attributes for software refactoring: a many-objective search-based software engineering approach,” *Empirical Software Engineering*, vol. 21, no. 6, pp. 2503–2545, 2016.
- [9] N. Tsantalis, T. Chaikalis, and A. Chatzigeorgiou, “Jdeodorant: Identification and removal of type-checking bad smells,” in *2008 12th European Conference on Software Maintenance and Reengineering*, pp. 329–331, IEEE, 2008.
- [10] J. Al Dallal, “Identifying refactoring opportunities in object-oriented code: A systematic literature review,” *Information and software Technology*, vol. 58, pp. 231–249, 2015.

- [11] H. Wang, M. Kessentini, and A. Ouni, “Prediction of web services evolution,” in *International Conference on Service-Oriented Computing*, pp. 282–297, Springer, 2016.
- [12] A. Ouni, R. Gaikovina Kula, M. Kessentini, and K. Inoue, “Web service antipatterns detection using genetic programming,” in *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, pp. 1351–1358, ACM, 2015.
- [13] G. Huang, H. Mei, and Q.-x. Wang, “Towards software architecture at runtime,” *ACM SIGSOFT Software Engineering Notes*, vol. 28, no. 2, p. 8, 2003.
- [14] S. Das, W. G. Lutters, and C. B. Seaman, “Understanding documentation value in software maintenance,” in *Proceedings of the 2007 Symposium on Computer human interaction for the management of information technology*, pp. 2–es, 2007.
- [15] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., 1999.
- [16] W. F. Opdyke, *Refactoring Object-oriented Frameworks*. PhD thesis, Champaign, IL, USA, 1992.
- [17] W. G. Griswold, *Program Restructuring As an Aid to Software Maintenance*. PhD thesis, Seattle, WA, USA, 1992.
- [18] “The developer coefficient.” URL: <https://stripe.com/reports/developer-coefficient-2018>.
- [19] B. Du Bois, S. Demeyer, and J. Verelst, “Refactoring-improving coupling and cohesion of existing code,” in *11th working conference on reverse engineering*, pp. 144–151, IEEE, 2004.
- [20] S. R. Foster, W. G. Griswold, and S. Lerner, “Witchdoctor: Ide support for real-time auto-completion of refactorings,” in *2012 34th International Conference on Software Engineering (ICSE)*, pp. 222–232, IEEE, 2012.
- [21] X. Ge and E. Murphy-Hill, “Benefactor: a flexible refactoring tool for eclipse,” in *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, pp. 19–20, 2011.
- [22] M. Kim, M. Gee, A. Loh, and N. Rachatasumrit, “Ref-finder: a refactoring reconstruction tool based on logic query templates,” in *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pp. 371–372, ACM, 2010.
- [23] R. Marinescu, “Detection strategies: Metrics-based rules for detecting design flaws,” in *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.*, pp. 350–359, IEEE, 2004.
- [24] 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, 2011.

- [25] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson, “Automated detection of refactorings in evolving components,” in *European Conference on Object-Oriented Programming*, pp. 404–428, Springer, 2006.
- [26] D. Batory, J. N. Sarvela, and A. Rauschmayer, “Scaling step-wise refinement,” *IEEE Transactions on Software Engineering*, vol. 30, no. 6, pp. 355–371, 2004.
- [27] J. Kim, D. Batory, D. Dig, and M. Azanza, “Improving refactoring speed by 10x,” in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pp. 1145–1156, IEEE, 2016.
- [28] 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.
- [29] A. Ouni, M. Kessentini, H. Sahraoui, and M. Boukadoum, “Maintainability defects detection and correction: a multi-objective approach,” *Automated Software Engineering*, vol. 20, no. 1, pp. 47–79, 2013.
- [30] 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, 2014.
- [31] W. F. Opdyke, “Refactoring: An aid in designing application frameworks and evolving object-oriented systems,” in *Proc. SOOPPA’90: Symposium on Object-Oriented Programming Emphasizing Practical Applications*, 1990.
- [32] W. G. Griswold, “Program restructuring as an aid to software maintenance.,” *PhD thesis, University of Washington, Seattle, WA, USA*, 1992.
- [33] W. F. Opdyke, “Refactoring object-oriented frameworks,” *PhD thesis, University of Illinois at Urbana-Champaign Champaign, IL, USA*, 1992.
- [34] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, “Refactoring: Improving the Design of Existing Code,” *Xtemp01*, pp. 1–337, 1999.
- [35] J. Al Dallal and A. Abdin, “Empirical evaluation of the impact of object-oriented code refactoring on quality attributes: A systematic literature review,” *IEEE Transactions on Software Engineering*, vol. 44, no. 1, pp. 44–69, 2017.
- [36] S. Singh and S. Kaur, “A systematic literature review: Refactoring for disclosing code smells in object oriented software,” *Ain Shams Engineering Journal*, vol. 9, no. 4, pp. 2129–2151, 2018.
- [37] “Slr website,” 2020. URL: <https://slr.iselab.us/>.
- [38] B. Kitchenham and S. Charters, “Guidelines for performing systematic literature reviews in software engineering,” *Vol. 5. Technical report, Ver. 2.3 EBSE Technical Report. EBSE*, 2007.

- [39] C. Wohlin, “Guidelines for snowballing in systematic literature studies and a replication in software engineering,” in *Proceedings of the 18th international conference on evaluation and assessment in software engineering*, pp. 1–10, 2014.
- [40] B. Kitchenham, O. P. Brereton, D. Budgen, M. Turner, J. Bailey, and S. Linkman, “Systematic literature reviews in software engineering—a systematic literature review,” *Information and software technology*, vol. 51, no. 1, pp. 7–15, 2009.
- [41] A. Ramirez, J. R. Romero, and C. L. Simons, “A systematic review of interaction in search-based software engineering,” *IEEE Transactions on Software Engineering*, vol. 45, no. 8, pp. 760–781, 2018.
- [42] M. Galster, D. Weyns, D. Tofan, B. Michalik, and P. Avgeriou, “Variability in software systems—a systematic literature review,” *IEEE Transactions on Software Engineering*, vol. 40, no. 3, pp. 282–306, 2013.
- [43] “Slr website,” 2020. URL: <https://slr.iselab.us/>.
- [44] H. Sajnani, V. Saini, and C. V. Lopes, “A comparative study of bug patterns in java cloned and non-cloned code,” in *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*, pp. 21–30, IEEE, 2014.
- [45] J. Ghofrani, M. Mohseni, and A. Bozorgmehr, “A conceptual framework for clone detection using machine learning,” in *2017 IEEE 4th International Conference on Knowledge-Based Engineering and Innovation (KBEI)*, pp. 0810–0817, IEEE, 2017.
- [46] I. Verebi, “A model-based approach to software refactoring,” in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 606–609, IEEE, 2015.
- [47] M. Tufano, F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk, “An empirical investigation into the nature of test smells,” in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pp. 4–15, 2016.
- [48] B. Zhang, G. Huang, Z. Zheng, J. Ren, and C. Hu, “Approach to mine the modularity of software network based on the most vital nodes,” *IEEE Access*, vol. 6, pp. 32543–32553, 2018.
- [49] G. Balogh, T. Gergely, Á. Beszédes, and T. Gyimóthy, “Are my unit tests in the right package?,” in *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pp. 137–146, IEEE, 2016.
- [50] N. Tsantalis, D. Mazinianian, and G. P. Krishnan, “Assessing the refactorability of software clones,” *IEEE Transactions on Software Engineering*, vol. 41, no. 11, pp. 1055–1090, 2015.

- [51] G. Soares, R. Gheyi, and T. Massoni, “Automated behavioral testing of refactoring engines,” *IEEE Transactions on Software Engineering*, vol. 39, no. 2, pp. 147–162, 2012.
- [52] J. Zhang, S. Han, D. Hao, L. Zhang, and D. Zhang, “Automated refactoring of nested-if formulae in spreadsheets,” in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 833–838, 2018.
- [53] Y. Kataoka, M. D. Ernst, W. G. Griswold, and D. Notkin, “Automated support for program refactoring using invariants,” in *Proceedings IEEE International Conference on Software Maintenance. ICSM 2001*, pp. 736–743, IEEE, 2001.
- [54] M. Mondal, C. K. Roy, and K. A. Schneider, “A comparative study on the bug-proneness of different types of code clones,” in *2015 IEEE International conference on software maintenance and evolution (ICSME)*, pp. 91–100, IEEE, 2015.
- [55] 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.
- [56] W. Snipes, B. Robinson, and E. Murphy-Hill, “Code hot spot: A tool for extraction and analysis of code change history,” in *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, pp. 392–401, IEEE, 2011.
- [57] H. Liu, Q. Liu, Z. Niu, and Y. Liu, “Dynamic and automatic feedback-based threshold adaptation for code smell detection,” *IEEE Transactions on Software Engineering*, vol. 42, no. 6, pp. 544–558, 2015.
- [58] V. Cosentino, S. Duenas, A. Zerouali, G. Robles, and J. M. González-Barahona, “Graal: The quest for source code knowledge,” In *2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, pp. 123–128, 2018.
- [59] S. Charalampidou, A. Ampatzoglou, A. Chatzigeorgiou, A. Gkortzis, and P. Avgeriou, “Identifying extract method refactoring opportunities based on functional relevance,” *IEEE Transactions on Software Engineering*, vol. 43, no. 10, pp. 954–974, 2016.
- [60] A. Rani and J. K. Chhabra, “Prioritization of smelly classes: A two phase approach (reducing refactoring efforts),” in *2017 3rd International Conference on Computational Intelligence & Communication Technology (CICT)*, pp. 1–6, IEEE, 2017.
- [61] P. Rachow, “Refactoring decision support for developers and architects based on architectural impact,” in *2019 IEEE International Conference on Software Architecture Companion (ICSA-C)*, pp. 262–266, IEEE, 2019.
- [62] 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.



- [63] J. Kim, D. Batory, and D. Dig, “Scripting parametric refactorings in java to retrofit design patterns,” in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 211–220, IEEE, 2015.
- [64] M. A. Parande and G. Koru, “A longitudinal analysis of the dependency concentration in smaller modules for open-source software products,” in *2010 IEEE International Conference on Software Maintenance*, pp. 1–5, IEEE, 2010.
- [65] H. Liu, Z. Xu, and Y. Zou, “Deep learning based feature envy detection,” in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pp. 385–396, 2018.
- [66] R. Morales, R. Saborido, F. Khomh, F. Chicano, and G. Antoniol, “Earmo: An energy-aware refactoring approach for mobile apps,” *IEEE Transactions on Software Engineering*, vol. 44, no. 12, pp. 1176–1206, 2017.
- [67] H. Liu, L. Yang, Z. Niu, Z. Ma, and W. Shao, “Facilitating software refactoring with appropriate resolution order of bad smells,” in *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pp. 265–268, 2009.
- [68] H. Liu, Q. Liu, Y. Liu, and Z. Wang, “Identifying renaming opportunities by expanding conducted rename refactorings,” *IEEE Transactions on Software Engineering*, vol. 41, no. 9, pp. 887–900, 2015.
- [69] B. Lin, S. Scalabrino, A. Mocci, R. Oliveto, G. Bavota, and M. Lanza, “Investigating the use of code analysis and nlp to promote a consistent usage of identifiers,” in *2017 IEEE 17th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pp. 81–90, IEEE, 2017.
- [70] G. Bavota, R. Oliveto, M. Gethers, D. Poshyvanyk, and A. De Lucia, “Methodbook: Recommending move method refactorings via relational topic models,” *IEEE Transactions on Software Engineering*, vol. 40, no. 7, pp. 671–694, 2013.
- [71] C. Hinds-Charles, J. Adames, Y. Yang, Y. Shen, and Y. Wang, “A longitude analysis on bitcoin issue repository,” in *2018 1st IEEE International Conference on Hot Information-Centric Networking (HotICN)*, pp. 212–217, IEEE, 2018.
- [72] T. D. Oyetoyan, D. S. Cruzes, and C. Thurmman-Nielsen, “A decision support system to refactor class cycles,” in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 231–240, IEEE, 2015.
- [73] N. Rachatasumrit and M. Kim, “An empirical investigation into the impact of refactoring on regression testing,” in *2012 28th Ieee International Conference on Software Maintenance (Icsm)*, pp. 357–366, IEEE, 2012.
- [74] M. Mirzaaghaei, F. Pastore, and M. Pezze, “Automatically repairing test cases for evolving method declarations,” in *2010 IEEE International Conference on Software Maintenance*, pp. 1–5, IEEE, 2010.

- [75] B. Van Rompaey, B. Du Bois, and S. Demeyer, “Characterizing the relative significance of a test smell,” in *2006 22nd IEEE International Conference on Software Maintenance*, pp. 391–400, IEEE, 2006.
- [76] A. Sherwany, N. Zaza, and N. Nystrom, “A refactoring library for scala compiler extensions,” in *International Conference on Compiler Construction*, pp. 31–48, Springer, 2015.
- [77] S. Paydar and M. Kahani, “A semantic web based approach for design pattern detection from source code,” in *2012 2nd International eConference on Computer and Knowledge Engineering (ICCKE)*, pp. 289–294, IEEE, 2012.
- [78] T. Haendler, “A card game for learning software-refactoring principles,” *Proceedings of the 3rd International Symposium of Gamification and Games for Learning (GamiLearn '19)*, 2019.
- [79] C. Kastner, S. Apel, and D. Batory, “A case study implementing features using aspectj,” in *11th International Software Product Line Conference (SPLC 2007)*, pp. 223–232, IEEE, 2007.
- [80] T. Viana, “A catalog of bad smells in design-by-contract methodologies with java modeling language,” *Journal of Computing Science and Engineering*, vol. 7, no. 4, pp. 251–262, 2013.
- [81] D. Foetsch and E. Pulvermueller, “A concept and implementation of higher-level xml transformation languages,” *Knowledge-Based Systems*, vol. 22, no. 3, pp. 186–194, 2009.
- [82] J. Reutelshoefer, J. Baumeister, and F. Puppe, “A data structure for the refactoring of multimodal knowledge,” in *Proceedings of the 5th Workshop on Knowledge Engineering and Software Engineering*, pp. 33–45, 2009.
- [83] S. Mouchawrab, L. C. Briand, and Y. Labiche, “A measurement framework for object-oriented software testability,” *Information and software technology*, vol. 47, no. 15, pp. 979–997, 2005.
- [84] I. Cassol and G. Arévalo, “A methodology to infer and refactor an object-oriented model from c applications,” *Software: Practice and Experience*, vol. 48, no. 3, pp. 550–577, 2018.
- [85] G. De Ruvo and A. Santone, “A novel methodology based on formal methods for analysis and verification of wikis,” in *2014 IEEE 23rd International WETICE Conference*, pp. 411–416, IEEE, 2014.
- [86] S. Rebai, O. B. Sghaier, V. Alizadeh, M. Kessentini, and M. Chater, “Interactive refactoring documentation bot,” in *2019 19th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pp. 152–162, IEEE, 2019.

- [87] J. Krinke, “Mining execution relations for crosscutting concerns,” *IET software*, vol. 2, no. 2, pp. 65–78, 2008.
- [88] D. Bowes, D. Randall, and T. Hall, “The inconsistent measurement of message chains,” in *2013 4th International Workshop on Emerging Trends in Software Metrics (WET-SoM)*, pp. 62–68, IEEE, 2013.
- [89] J. Liu, “Feature interactions and software derivatives,” *Journal of Object Technology*, vol. 4, no. 3, pp. 13–19, 2004.
- [90] A. Swidan, F. Hermans, and R. Koesoemowidjojo, “Improving the performance of a large scale spreadsheet: a case study,” in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1, pp. 673–677, IEEE, 2016.
- [91] H. Li, S. Thompson, and T. Arts, “Extracting properties from test cases by refactoring,” in *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, pp. 472–473, IEEE, 2011.
- [92] S. Ducasse, O. Nierstrasz, N. Schärli, R. Wuyts, and A. P. Black, “Traits: A mechanism for fine-grained reuse,” *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 28, no. 2, pp. 331–388, 2006.
- [93] R. Ramos, J. Castro, J. Araújo, F. Alencar, and R. Penteado, “Divide and conquer refactoring: dealing with the large, scattering or tangling use case model,” in *Proceedings of the 8th Latin American Conference on Pattern Languages of Programs*, pp. 1–11, 2010.
- [94] E. Murphy-Hill, A. P. Black, D. Dig, and C. Parnin, “Gathering refactoring data: a comparison of four methods,” in *Proceedings of the 2nd Workshop on Refactoring Tools*, pp. 1–5, 2008.
- [95] A. Derezińska, “A structure-driven process of automated refactoring to design patterns,” in *International Conference on Information Systems Architecture and Technology*, pp. 39–48, Springer, 2017.
- [96] E. Selim, Y. Ghanam, C. Burns, T. Seyed, and F. Maurer, “A test-driven approach for extracting libraries of reusable components from existing applications,” in *International Conference on Agile Software Development*, pp. 238–252, Springer, 2011.
- [97] Y. Zhang, S. Dong, X. Zhang, H. Liu, and D. Zhang, “Automated refactoring for stampedlock,” *IEEE Access*, vol. 7, pp. 104900–104911, 2019.
- [98] H. Xue, S. Sun, G. Venkataramani, and T. Lan, “Machine learning-based analysis of program binaries: A comprehensive study,” *IEEE Access*, vol. 7, pp. 65889–65912, 2019.

- [99] Y. Zhang, S. Shao, H. Liu, J. Qiu, D. Zhang, and G. Zhang, “Refactoring java programs for customizable locks based on bytecode transformation,” *IEEE Access*, vol. 7, pp. 66292–66303, 2019.
- [100] M. F. Dolz, D. D. R. Astorga, J. Fernández, J. D. García, and J. Carretero, “Towards automatic parallelization of stream processing applications,” *IEEE Access*, vol. 6, pp. 39944–39961, 2018.
- [101] B. K. Sidhu, K. Singh, and N. Sharma, “A catalogue of model smells and refactoring operations for object-oriented software,” in *2018 Second International Conference on Inventive Communication and Computational Technologies (ICICCT)*, pp. 313–319, IEEE, 2018.
- [102] F. Medeiros, M. Ribeiro, R. Gheyi, and B. F. dos Santos Neto, “A catalogue of refactorings to remove incomplete annotations.,” *J. UCS*, vol. 20, no. 5, pp. 746–771, 2014.
- [103] P. Ma, Y. Bian, and X. Su, “A clustering method for pruning false positive of clone code detection,” in *Proceedings 2013 International Conference on Mechatronic Sciences, Electric Engineering and Computer (MEC)*, pp. 1917–1920, IEEE, 2013.
- [104] G.-S. Cojocar and A.-M. Guran, “A comparative analysis of monitoring concerns implementation in object oriented systems,” in *2018 IEEE 12th International Symposium on Applied Computational Intelligence and Informatics (SACI)*, pp. 000355–000360, IEEE, 2018.
- [105] S. Negara, N. Chen, M. Vakilian, R. E. Johnson, and D. Dig, “A comparative study of manual and automated refactorings,” in *European Conference on Object-Oriented Programming*, pp. 552–576, Springer, 2013.
- [106] T. Chen and C. He, “A comparison of approaches to legacy system crosscutting concerns mining,” in *2013 International Conference on Computer Sciences and Applications*, pp. 813–816, IEEE, 2013.
- [107] A. Martini, E. Sikander, and N. Madlani, “A semi-automated framework for the identification and estimation of architectural technical debt: A comparative case-study on the modularization of a software component,” *Information and Software Technology*, vol. 93, pp. 264–279, 2018.
- [108] M. T. Valente, V. Borges, and L. Passos, “A semi-automatic approach for extracting software product lines,” *IEEE Transactions on Software Engineering*, vol. 38, no. 4, pp. 737–754, 2011.
- [109] K. Garcés, J. M. Vara, F. Jouault, and E. Marcos, “Adapting transformations to metamodel changes via external transformation composition,” *Software & Systems Modeling*, vol. 13, no. 2, pp. 789–806, 2014.
- [110] S. A. Vidal, C. Marcos, and J. A. Díaz-Pace, “An approach to prioritize code smells for refactoring,” *Automated Software Engineering*, vol. 23, no. 3, pp. 501–532, 2016.

- [111] C. Brown, H. Li, and S. Thompson, “An expression processor: a case study in refactoring haskell programs,” in *International Symposium on Trends in Functional Programming*, pp. 31–49, Springer, 2010.
- [112] M. Marin, A. van Deursen, L. Moonen, and R. van der Rijst, “An integrated cross-cutting concern migration strategy and its semi-automated application to jhotdraw,” *Automated Software Engineering*, vol. 16, no. 2, pp. 323–356, 2009.
- [113] A. O’Riordan, “Aspect-oriented reengineering of an object-oriented library in a short iteration agile process,” *Informatica*, vol. 35, no. 4, 2011.
- [114] K. Fujiwara, K. Fushida, N. Yoshida, and H. Iida, “Assessing refactoring instances and the maintainability benefits of them from version archives,” in *International Conference on Product Focused Software Process Improvement*, pp. 313–323, Springer, 2013.
- [115] B. Alkhazi, T. Ruas, M. Kessentini, M. Wimmer, and W. I. Grosky, “Automated refactoring of atl model transformations: a search-based approach,” in *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*, pp. 295–304, 2016.
- [116] M. Tanhaei, J. Habibi, and S.-H. Mirian-Hosseiniabadi, “Automating feature model refactoring: A model transformation approach,” *Information and Software Technology*, vol. 80, pp. 138–157, 2016.
- [117] V. Alizadeh, H. Fehri, and M. Kessentini, “Less is more: From multi-objective to mono-objective refactoring via developer’s knowledge extraction,” in *2019 19th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pp. 181–192, IEEE, 2019.
- [118] V. Alizadeh, M. A. Ouali, M. Kessentini, and M. Chater, “Refbot: intelligent software refactoring bot,” in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 823–834, IEEE, 2019.
- [119] Z. Mushtaq, G. Rasool, and B. Shehzad, “Multilingual source code analysis: A systematic literature review,” *IEEE Access*, vol. 5, pp. 11307–11336, 2017.
- [120] F. Schmidt, S. G. MacDonell, and A. M. Connor, “An automatic architecture reconstruction and refactoring framework,” in *Software Engineering Research, Management and Applications 2011*, pp. 95–111, Springer, 2012.
- [121] G. Cong, H. Wen, I.-h. Chung, D. Klepacki, H. Murata, and Y. Negishi, “An efficient framework for multi-dimensional tuning of high performance computing applications,” in *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, pp. 1376–1387, IEEE, 2012.
- [122] J. Park, M. Kim, and D.-H. Bae, “An empirical study of supplementary patches in open source projects,” *Empirical Software Engineering*, vol. 22, no. 1, pp. 436–473, 2017.

- [123] T. L. Nguyen, A. Fish, and M. Song, “An empirical study on similar changes in evolving software,” in *2018 IEEE International Conference on Electro/Information Technology (EIT)*, pp. 0560–0563, IEEE, 2018.
- [124] M. Bruntink, A. Van Deursen, T. Tourwe, and R. van Engelen, “An evaluation of clone detection techniques for crosscutting concerns,” in *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.*, pp. 200–209, IEEE, 2004.
- [125] Y. Kosker, B. Turhan, and A. Bener, “An expert system for determining candidate software classes for refactoring,” *Expert Systems with Applications*, vol. 36, no. 6, pp. 10000–10003, 2009.
- [126] B. L. Sousa, M. A. Bigonha, and K. A. Ferreira, “An exploratory study on cooccurrence of design patterns and bad smells using software metrics,” *Software: Practice and Experience*, vol. 49, no. 7, pp. 1079–1113, 2019.
- [127] O. Mehani, G. Jourjon, T. Rakotoarivelo, and M. Ott, “An instrumentation framework for the critical task of measurement collection in the future internet,” *Computer Networks*, vol. 63, pp. 68–83, 2014.
- [128] M. Schäfer, A. Thies, F. Steimann, and F. Tip, “A comprehensive approach to naming and accessibility in refactoring java programs,” *IEEE Transactions on Software Engineering*, vol. 38, no. 6, pp. 1233–1257, 2012.
- [129] D. Dig, “A practical tutorial on refactoring for parallelism,” in *2010 IEEE International Conference on Software Maintenance*, pp. 1–2, IEEE, 2010.
- [130] X. Li and J. P. Gallagher, “A source-level energy optimization framework for mobile applications,” in *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pp. 31–40, IEEE, 2016.
- [131] R. Khatchadourian, Y. Tang, M. Bagherzadeh, and S. Ahmed, “[engineering paper] a tool for optimizing java 8 stream software via automated refactoring,” in *2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pp. 34–39, IEEE, 2018.
- [132] Z. Xing and E. Stroulia, “Api-evolution support with diff-catchup,” *IEEE Transactions on Software Engineering*, vol. 33, no. 12, pp. 818–836, 2007.
- [133] R. Gheyi, T. Massoni, and P. Borba, “A rigorous approach for proving model refactorings,” in *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pp. 372–375, 2005.
- [134] B. Cyganek, “Adding parallelism to the hybrid image processing library in multi-threading and multi-core systems,” in *2011 IEEE 2nd International Conference on Networked Embedded Systems for Enterprise Applications*, pp. 1–8, IEEE, 2011.

- [135] R. Hardt and E. V. Munson, “An empirical evaluation of ant build maintenance using formiga,” in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 201–210, IEEE, 2015.
- [136] R. Kolb, D. Muthig, T. Patzke, and K. Yamauchi, “A case study in refactoring a legacy component for reuse in a product line,” in *21st IEEE International Conference on Software Maintenance (ICSM’05)*, pp. 369–378, IEEE, 2005.
- [137] D. Strein, R. Lincke, J. Lundberg, and W. Löwe, “An extensible meta-model for program analysis,” *IEEE Transactions on Software Engineering*, vol. 33, no. 9, pp. 592–607, 2007.
- [138] Y.-W. Kwon, “Automated s/w reengineering for fault-tolerant and energy-efficient distributed execution,” in *2013 IEEE International Conference on Software Maintenance*, pp. 582–585, IEEE, 2013.
- [139] B. Adams, H. Tromp, K. De Schutter, and W. De Meuter, “Design recovery and maintenance of build systems,” in *2007 IEEE International Conference on Software Maintenance*, pp. 114–123, IEEE, 2007.
- [140] R. Bahsoon and W. Emmerich, “Evaluating architectural stability with real options theory,” in *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.*, pp. 443–447, IEEE, 2004.
- [141] J. O’neal, K. Weide, and A. Dubey, “Experience report: refactoring the mesh interface in flash, a multiphysics software,” in *2018 IEEE 14th International Conference on e-Science (e-Science)*, pp. 1–6, IEEE, 2018.
- [142] M. A. Khan and H. Tembine, “Meta-learning for realizing self-x management of future networks,” *IEEE Access*, vol. 5, pp. 19072–19083, 2017.
- [143] A. Cleve, “Program analysis and transformation for data-intensive system evolution,” in *2010 IEEE International Conference on Software Maintenance*, pp. 1–6, IEEE, 2010.
- [144] D. Binkley, M. Ceccato, M. Harman, F. Ricca, and P. Tonella, “Automated refactoring of object oriented code into aspects,” in *21st IEEE International Conference on Software Maintenance (ICSM’05)*, pp. 27–36, IEEE, 2005.
- [145] F. Castor Filho, A. Garcia, and C. M. F. Rubira, “Extracting error handling to aspects: A cookbook,” in *2007 IEEE International Conference on Software Maintenance*, pp. 134–143, IEEE, 2007.
- [146] M. Bajammal, D. Mazinianian, and A. Mesbah, “Generating reusable web components from mockups,” in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pp. 601–611, 2018.
- [147] N. A. Kraft, E. B. Duffy, and B. A. Malloy, “Grammar recovery from parse trees and metrics-guided grammar refactoring,” *IEEE Transactions on Software Engineering*, vol. 35, no. 6, pp. 780–794, 2009.

- [148] D. Spinellis, “Global analysis and transformations in preprocessed languages,” *IEEE Transactions on Software Engineering*, vol. 29, no. 11, pp. 1019–1030, 2003.
- [149] C. Noguera, A. Kellens, C. De Roover, and V. Jonckers, “Refactoring in the presence of annotations,” in *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pp. 337–346, IEEE, 2012.
- [150] S. Rongrong, Z. Liping, and Z. Fengrong, “A method for identifying and recommending reconstructed clones,” in *Proceedings of the 2019 3rd International Conference on Management Engineering, Software Engineering and Service Sciences*, pp. 39–44, 2019.
- [151] Y. Khan and M. El-Attar, “A model transformation approach towards refactoring use case models based on antipatterns,” in *21st International Conference on Software Engineering and Data Engineering (SEDE’12), Los Angeles, California, USA*, pp. 49–54, 2012.
- [152] K. Grolinger and M. A. Capretz, “A unit test approach for database schema evolution,” *Information and Software Technology*, vol. 53, no. 2, pp. 159–170, 2011.
- [153] O. Febbraro, K. Reale, and F. Ricca, “Aspide: Integrated development environment for answer set programming,” in *International Conference on Logic Programming and Nonmonotonic Reasoning*, pp. 317–330, Springer, 2011.
- [154] A. Garrido and R. Johnson, “Analyzing multiple configurations of a c program,” in *21st IEEE International Conference on Software Maintenance (ICSM’05)*, pp. 379–388, IEEE, 2005.
- [155] A. Paltoglou, V. E. Zafeiris, E. A. Giakoumakis, and N. Diamantidis, “Automated refactoring of client-side javascript code to es6 modules,” in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 402–412, IEEE, 2018.
- [156] R. Khatchadourian, J. Sawin, and A. Rountev, “Automated refactoring of legacy java software to enumerated types,” in *2007 IEEE International Conference on Software Maintenance*, pp. 224–233, IEEE, 2007.
- [157] M. Marin, L. Moonen, and A. van Deursen, “A classification of crosscutting concerns,” in *21st IEEE International Conference on Software Maintenance (ICSM’05)*, pp. 673–676, IEEE, 2005.
- [158] M. Mortensen, S. Ghosh, and J. Bieman, “Aspect-oriented refactoring of legacy applications: An evaluation,” *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 118–140, 2010.
- [159] M. Mondal, C. K. Roy, and K. A. Schneider, “Automatic identification of important clones for refactoring and tracking,” in *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*, pp. 11–20, IEEE, 2014.



- [160] A. Kellens, K. De Schutter, T. D’Hondt, V. Jonckers, and H. Doggen, “Experiences in modularizing business rules into aspects,” in *2008 IEEE International Conference on Software Maintenance*, pp. 448–451, IEEE, 2008.
- [161] R. Stoiber, S. Fricker, M. Jehle, and M. Glinz, “Feature unweaving: Refactoring software requirements specifications into software product lines,” in *2010 18th IEEE International Requirements Engineering Conference*, pp. 403–404, IEEE, 2010.
- [162] G. Zhao and J. Huang, “Deepsim: deep learning code functional similarity,” in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 141–151, 2018.
- [163] S. Demeyer, S. Ducasse, and O. Nierstrasz, “Object-oriented reengineering: patterns and techniques,” in *21st IEEE International Conference on Software Maintenance (ICSM’05)*, pp. 723–724, IEEE, 2005.
- [164] P. Hegedus, “Revealing the effect of coding practices on software maintainability,” in *2013 IEEE International Conference on Software Maintenance*, pp. 578–581, IEEE, 2013.
- [165] T. Feng, J. Zhang, H. Wang, and X. Wang, “Software design improvement through anti-patterns identification,” in *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.*, p. 524, IEEE, 2004.
- [166] S. Meng and L. S. Barbosa, “A coalgebraic semantic framework for reasoning about uml sequence diagrams,” in *2008 The Eighth International Conference on Quality Software*, pp. 17–26, IEEE, 2008.
- [167] P. Mayer and A. Schroeder, “Cross-language code analysis and refactoring,” in *2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation*, pp. 94–103, IEEE, 2012.
- [168] R. Moser, P. Abrahamsson, W. Pedrycz, A. Sillitti, and G. Succi, “A case study on the impact of refactoring on quality and productivity in an agile team,” in *IFIP Central and East European Conference on Software Engineering Techniques*, pp. 252–266, Springer, 2007.
- [169] A. L. Cândido, F. A. Trinta, L. S. Rocha, P. A. Rego, N. C. Mendonça, and V. C. Garcia, “A microservice based architecture to support offloading in mobile cloud computing,” in *Proceedings of the XIII Brazilian Symposium on Software Components, Architectures, and Reuse*, pp. 93–102, 2019.
- [170] A. Peruma, “A preliminary study of android refactorings,” in *2019 IEEE/ACM 6th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pp. 148–149, IEEE, 2019.
- [171] M. Mascheroni and E. Irrazábal, “A design pattern approach for restful tests: A case study,” in *IEEE 12th Colombian Computing Congress*, 2018.

- [172] D. Kermek, T. Jakupić, and N. Vrček, “A model of heterogeneous distributed system for foreign exchange portfolio analysis,” *Journal of Information and Organizational Sciences*, vol. 30, no. 1, pp. 83–92, 2006.
- [173] G. Rodriguez, A. Teyseyre, Á. Soria, and L. Berdun, “A visualization tool to detect refactoring opportunities in soa applications,” in *2017 XLIII Latin American Computer Conference (CLEI)*, pp. 1–10, IEEE, 2017.
- [174] H. Li, S. Thompson, P. Lamela Seijas, and M. A. Francisco, “Automating property-based testing of evolving web services,” in *Proceedings of the ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation*, pp. 169–180, 2014.
- [175] D. Athanasopoulos, A. V. Zarras, G. Miskos, V. Issarny, and P. Vassiliadis, “Cohesion-driven decomposition of service interfaces without access to source code,” *IEEE Transactions on Services Computing*, vol. 8, no. 4, pp. 550–562, 2014.
- [176] M. Kessentini and H. Wang, “Detecting refactorings among multiple web service releases: A heuristic-based approach,” in *2017 IEEE International Conference on Web Services (ICWS)*, pp. 365–372, IEEE, 2017.
- [177] F. Wei, C. Ouyang, and A. Barros, “Discovering behavioural interfaces for overloaded web services,” in *2015 IEEE World Congress on Services*, pp. 286–293, IEEE, 2015.
- [178] K. Fekete, A. Pelle, and K. Csorba, “Energy efficient code optimization in mobile environment,” in *2014 IEEE 36th International Telecommunications Energy Conference (INTELEC)*, pp. 1–6, IEEE, 2014.
- [179] W. B. Langdon, “Genetic improvement of programs,” in *2014 16th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, pp. 14–19, IEEE, 2014.
- [180] H. Wang, A. Ouni, M. Kessentini, B. Maxim, and W. I. Grosky, “Identification of web service refactoring opportunities as a multi-objective problem,” in *2016 IEEE International Conference on Web Services (ICWS)*, pp. 586–593, IEEE, 2016.
- [181] M. Kim, T. Zimmermann, and N. Nagappan, “An empirical study of refactoring challenges and benefits at microsoft,” *IEEE Transactions on Software Engineering*, vol. 40, no. 7, pp. 633–649, 2014.
- [182] S. M. Olbrich, D. S. Cruzes, and D. I. Sjøberg, “Are all code smells harmful? a study of god classes and brain classes in the evolution of three open source systems,” in *2010 IEEE International Conference on Software Maintenance*, pp. 1–10, IEEE, 2010.
- [183] P. S. Kochhar, F. Thung, and D. Lo, “Automatic fine-grained issue report reclassification,” in *2014 19th International Conference on Engineering of Complex Computer Systems*, pp. 126–135, IEEE, 2014.

- [184] G. Bastide, A. Seriali, and M. Oussalah, “Dynamic adaptation of software component structures,” in *2006 IEEE International Conference on Information Reuse & Integration*, pp. 404–409, IEEE, 2006.
- [185] G. Zhang, L. Shen, X. Peng, Z. Xing, and W. Zhao, “Incremental and iterative reengineering towards software product line: An industrial case study,” in *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, pp. 418–427, IEEE, 2011.
- [186] G. Bavota, R. Oliveto, A. De Lucia, G. Antoniol, and Y.-G. Gueheneuc, “Playing with refactoring: Identifying extract class opportunities through game theory,” in *2010 IEEE International Conference on Software Maintenance*, pp. 1–5, IEEE, 2010.
- [187] P. S. Kochhar, Y. Tian, and D. Lo, “Potential biases in bug localization: Do they matter?,” in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pp. 803–814, 2014.
- [188] R. Khatchadourian and H. Masuhara, “Defaultification refactoring: A tool for automatically converting java methods to default,” in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 984–989, IEEE, 2017.
- [189] H. K. Wright, D. Jasper, M. Klimek, C. Carruth, and Z. Wan, “Large-scale automated refactoring using clangmr,” in *2013 IEEE International Conference on Software Maintenance*, pp. 548–551, IEEE, 2013.
- [190] D. Mazinianian and N. Tsantalis, “Migrating cascading style sheets to preprocessors by introducing mixins,” in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pp. 672–683, 2016.
- [191] P. Tonella and M. Ceccato, “Migrating interface implementations to aspects,” in *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.*, pp. 220–229, IEEE, 2004.
- [192] M. Ceccato, “Migrating object oriented code to aspect oriented programming,” 2006.
- [193] D. Majumdar, “Migration from procedural programming to aspect oriented paradigm,” in *2009 IEEE/ACM International Conference on Automated Software Engineering*, pp. 712–715, IEEE, 2009.
- [194] C. Marcos, S. Vidal, E. Abait, M. Arroqui, and S. Sampaoli, “Refactoring of a beef-cattle farm simulator,” *IEEE Latin America Transactions*, vol. 9, no. 7, pp. 1099–1104, 2011.
- [195] R. Khatchadourian and B. Muskalla, “Enumeration refactoring: a tool for automatically converting java constants to enumerated types,” in *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pp. 181–182, 2010.
- [196] E. L. Alves, M. Song, T. Massoni, P. D. Machado, and M. Kim, “Refactoring inspection support for manual refactoring edits,” *IEEE Transactions on Software Engineering*, vol. 44, no. 4, pp. 365–383, 2017.

- [197] Y. Yu, J. Jurjens, and J. Mylopoulos, “Traceability for the maintenance of secure software,” in *2008 IEEE International Conference on Software Maintenance*, pp. 297–306, IEEE, 2008.
- [198] C. Kulkarni, “Notice of violation of iee publication principles a qualitative approach for refactoring of code clone opportunities using graph and tree methods,” in *2016 International Conference on Information Technology (InCITe)-The Next Generation IT Summit on the Theme-Internet of Things: Connect your Worlds*, pp. 154–159, IEEE, 2016.
- [199] A. F. Tappenden, T. Huynh, J. Miller, A. Geras, and M. Smith, “Agile development of secure web-based applications,” *International Journal of Information Technology and Web Engineering (IJITWE)*, vol. 1, no. 2, pp. 1–24, 2006.
- [200] P. M. Cousot, R. Cousot, F. Logozzo, and M. Barnett, “An abstract interpretation framework for refactoring with application to extract methods with contracts,” in *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, pp. 213–232, 2012.
- [201] P. Borba, “An introduction to software product line refactoring,” in *International Summer School on Generative and Transformational Techniques in Software Engineering*, pp. 1–26, Springer, 2009.
- [202] O. Macek and K. Richta, “Application and relational database co-refactoring,” *Computer Science and Information Systems*, vol. 11, no. 2, pp. 503–524, 2014.
- [203] M. S. Feather and L. Z. Markosian, “Architecting and generalizing a safety case for critical condition detection software an experience report,” in *2013 1st International Workshop on Assurance Cases for Software-Intensive Systems (ASSURE)*, pp. 29–33, IEEE, 2013.
- [204] P. Muntean, M. Monperrus, H. Sun, J. Grossklags, and C. Eckert, “Intrepair: Informed repairing of integer overflows,” *IEEE Transactions on Software Engineering*, 2019.
- [205] S. Demeyer, “Refactor conditionals into polymorphism: what’s the performance cost of introducing virtual calls?,” in *21st IEEE International Conference on Software Maintenance (ICSM’05)*, pp. 627–630, IEEE, 2005.
- [206] A. Kumar, A. Sutton, and B. Stroustrup, “Rejuvenating c++ programs through demacrofication,” in *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pp. 98–107, IEEE, 2012.
- [207] A. Kumar, A. Sutton, and B. Stroustrup, “The demacrofier,” in *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pp. 658–661, IEEE, 2012.
- [208] I. Sora, “A meta-model for representing language-independent primary dependency structures.,” in *ENASE*, pp. 65–74, 2012.

- [209] M. F. Zibran, R. K. Saha, M. Asaduzzaman, and C. K. Roy, “Analyzing and forecasting near-miss clones in evolving software: An empirical study,” in *2011 16th IEEE International Conference on Engineering of Complex Computer Systems*, pp. 295–304, IEEE, 2011.
- [210] R. Rolim, “Automating repetitive code changes using examples,” in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 1063–1065, 2016.
- [211] W. S. Evans, C. W. Fraser, and F. Ma, “Clone detection via structural abstraction,” *Software Quality Journal*, vol. 17, no. 4, pp. 309–330, 2009.
- [212] A. Khan, H. A. Basit, S. M. Sarwar, and M. M. Yousaf, “Cloning in popular server side technologies using agile development: An empirical study,” *Pakistan Journal of Engineering and Applied Sciences*, no. 1, 2018.
- [213] T. D. Oyetoyan, R. Conradi, and D. S. Cruzes, “Criticality of defects in cyclic dependent components,” in *2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pp. 21–30, IEEE, 2013.
- [214] M. Gatrell, S. Counsell, and T. Hall, “Empirical support for two refactoring studies using commercial c# software,” in *13th International Conference on Evaluation and Assessment in Software Engineering (EASE) 13*, pp. 1–10, 2009.
- [215] R. K. Saha, M. Asaduzzaman, M. F. Zibran, C. K. Roy, and K. A. Schneider, “Evaluating code clone genealogies at release level: An empirical study,” in *2010 10th IEEE Working Conference on Source Code Analysis and Manipulation*, pp. 87–96, IEEE, 2010.
- [216] A. Derezińska and M. Byczkowski, “Evaluation of design pattern utilization and software metrics in c# programs,” in *International Conference on Dependability and Complex Systems*, pp. 132–142, Springer, 2019.
- [217] Y. A. Liu, M. Gorbovitski, and S. D. Stoller, “A language and framework for invariant-driven transformations,” *ACM Sigplan Notices*, vol. 45, no. 2, pp. 55–64, 2009.
- [218] I. Lanc, P. Bui, D. Thain, and S. Emrich, “Adapting bioinformatics applications for heterogeneous systems: a case study,” *Concurrency and Computation: Practice and Experience*, vol. 26, no. 4, pp. 866–877, 2014.
- [219] L. E. d. S. Amorim, M. J. Steindorfer, S. Erdweg, and E. Visser, “Declarative specification of indentation rules: a tooling perspective on parsing and pretty-printing layout-sensitive languages,” in *Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering*, pp. 3–15, 2018.
- [220] Z. Chen, L. Chen, W. Ma, and B. Xu, “Detecting code smells in python programs,” in *2016 International Conference on Software Analysis, Testing and Evolution (SATE)*, pp. 18–23, IEEE, 2016.

- [221] C. Chapman and K. T. Stolee, “Exploring regular expression usage and context in python,” in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pp. 282–293, 2016.
- [222] J. B. Cabral, B. Sánchez, F. Ramos, S. Gurovich, P. M. Granitto, and J. Vanderplas, “From fats to feets: Further improvements to an astronomical feature extraction tool based on machine learning,” *Astronomy and computing*, vol. 25, pp. 213–220, 2018.
- [223] M. Zhu, F. McKenna, and M. H. Scott, “Openseespy: Python library for the openses finite element framework,” *SoftwareX*, vol. 7, pp. 6–11, 2018.
- [224] M. Furr, J.-h. An, and J. S. Foster, “Profile-guided static typing for dynamic scripting languages,” in *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pp. 283–300, 2009.
- [225] Z. W. Bell, G. G. Davidson, T. M. D’Azevedo, W. Joubert, J. K. Munro Jr, D. R. Patlolla, and B. Vacaliuc, “Python for development of openmp and cuda kernels for multidimensional data,” in *Symposium on Application Accelerators in HPC*, 2011.
- [226] Y. Hu, U. Z. Ahmed, S. Mechtaev, B. Leong, and A. Roychoudhury, “Re-factoring based program repair applied to programming assignments,” in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 388–398, IEEE, 2019.
- [227] C. Wang, S. Hirasawa, H. Takizawa, and H. Kobayashi, “A platform-specific code smell alert system for high performance computing applications,” in *2014 IEEE International Parallel & Distributed Processing Symposium Workshops*, pp. 652–661, IEEE, 2014.
- [228] Ç. Biray and F. Buzluca, “A learning-based method for detecting defective classes in object-oriented systems,” in *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pp. 1–8, IEEE, 2015.
- [229] W. Hasanain, Y. Labiche, and S. Eldh, “An analysis of complex industrial test code using clone analysis,” in *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pp. 482–489, IEEE, 2018.
- [230] D. Mazinianian and N. Tsantalis, “An empirical study on the use of css preprocessors,” in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1, pp. 168–178, IEEE, 2016.
- [231] D. Mazinianian and N. Tsantalis, “Cssdev: refactoring duplication in cascading style sheets,” in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pp. 63–66, IEEE, 2017.
- [232] D. Mazinianian, N. Tsantalis, and A. Mesbah, “Discovering refactoring opportunities in cascading style sheets,” in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 496–506, 2014.

- [233] D. D. Perez and W. Le, “Generating predicate callback summaries for the android framework,” in *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pp. 68–78, IEEE, 2017.
- [234] S. Negara, M. Vakilian, N. Chen, R. E. Johnson, and D. Dig, “Is it dangerous to use version control histories to study source code evolution?,” in *European Conference on Object-Oriented Programming*, pp. 79–103, Springer, 2012.
- [235] M. Bosch, P. Genevès, and N. Layaïda, “Reasoning with style,” in *Twenty-Fourth International Joint Conference on Artificial Intelligence*, 2015.
- [236] H. A. Nguyen, H. V. Nguyen, T. T. Nguyen, and T. N. Nguyen, “Output-oriented refactoring in php-based dynamic web applications,” in *2013 IEEE International Conference on Software Maintenance*, pp. 150–159, IEEE, 2013.
- [237] B. Chen, Z. M. Jiang, P. Matos, and M. Lacaria, “An industrial experience report on performance-aware refactoring on a database-centric web application,” in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 653–664, IEEE, 2019.
- [238] L. Eshkevari, F. Dos Santos, J. R. Cordy, and G. Antoniol, “Are php applications ready for hack?,” in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pp. 63–72, IEEE, 2015.
- [239] J. L. Overbey and R. E. Johnson, “Differential precondition checking: A lightweight, reusable analysis for refactoring tools,” in *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, pp. 303–312, IEEE, 2011.
- [240] J. L. Overbey, R. E. Johnson, and M. Hafiz, “Differential precondition checking: a language-independent, reusable analysis for refactoring engines,” *Automated Software Engineering*, vol. 23, no. 1, pp. 77–104, 2016.
- [241] M. Hills, P. Klint, and J. J. Vinju, “Enabling php software engineering research in rascal,” *Science of Computer Programming*, vol. 134, pp. 37–46, 2017.
- [242] F. Gauthier, D. Letarte, T. Lavoie, and E. Merlo, “Extraction and comprehension of moodle’s access control model: A case study,” in *2011 Ninth Annual International Conference on Privacy, Security and Trust*, pp. 44–51, IEEE, 2011.
- [243] M. Hills and P. Klint, “Php air: Analyzing php systems with rascal,” in *2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, pp. 454–457, IEEE, 2014.
- [244] Y. Yu, Y. Wang, J. Mylopoulos, S. Liaskos, A. Lapouchnian, and J. C. S. do Prado Leite, “Reverse engineering goal models from legacy code,” in *13th IEEE International Conference on Requirements Engineering (RE’05)*, pp. 363–372, IEEE, 2005.

- [245] R. Lämmel and J. Visser, “A strafunski application letter,” in *International Symposium on Practical Aspects of Declarative Languages*, pp. 357–375, Springer, 2003.
- [246] G. M. Rama, “A desiderata for refactoring-based software modularity improvement,” in *Proceedings of the 3rd India software engineering conference*, pp. 93–102, 2010.
- [247] T. Mens and T. Tourwé, “A survey of software refactoring,” *IEEE Transactions on software engineering*, vol. 30, no. 2, pp. 126–139, 2004.
- [248] A. Abadi, R. Ettinger, and Y. A. Feldman, “Fine slicing,” in *International Conference on Fundamental Approaches to Software Engineering*, pp. 471–485, Springer, 2012.
- [249] M. Lillack, C. Bucholdt, and D. Schilling, “Detection of code clones in software generators,” in *Proceedings of the 6th International Workshop on Feature-Oriented Software Development*, pp. 37–44, 2014.
- [250] H. M. Sneed and K. Erdoes, “Migrating as400-cobol to java: a report from the field,” in *2013 17th European Conference on Software Maintenance and Reengineering*, pp. 231–240, IEEE, 2013.
- [251] M. K. Smith and T. Laszewski, “Modernization case study: Italian ministry of instruction, university, and research,” in *Information Systems Transformation*, pp. 171–191, Elsevier, 2010.
- [252] T. Hatano and A. Matsuo, “Removing code clones from industrial systems using compiler directives,” in *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, pp. 336–345, IEEE, 2017.
- [253] T. Gerlitz, Q. M. Tran, and C. Dziobek, “Detection and handling of model smells for matlab/simulink models,” in *MASE@ MoDELS*, pp. 13–22, 2015.
- [254] Z. Zhao, X. Li, L. He, C. Wu, and J. K. Hedrick, “Estimation of torques transmitted by twin-clutch of dry dual-clutch transmission during vehicle’s launching process,” *IEEE Transactions on Vehicular Technology*, vol. 66, no. 6, pp. 4727–4741, 2016.
- [255] K. Aishwarya, R. Ramesh, P. M. Sobarad, and V. Singh, “Lossy image compression using svd coding algorithm,” in *2016 International Conference on Wireless Communications, Signal Processing and Networking (WiSPNET)*, pp. 1384–1389, IEEE, 2016.
- [256] S. Schlesinger, P. Herber, T. Göthel, and S. Glesner, “Proving correctness of refactorings for hybrid simulink models with control flow,” in *International Workshop on Design, Modeling, and Evaluation of Cyber Physical Systems*, pp. 71–86, Springer, 2016.
- [257] S. Makka and B. Sagar, “Simulation of a model for refactoring approach for parallelism using parallel computing tool box,” in *Proceedings of First International Conference on Information and Communication Technology for Intelligent Systems: Volume 2*, pp. 77–84, Springer, 2016.



- [258] V. Pantelic, S. Postma, M. Lawford, M. Jaskolka, B. Mackenzie, A. Korobkine, M. Bender, J. Ong, G. Marks, and A. Wassying, “Software engineering practices and simulink: bridging the gap,” *International Journal on Software Tools for Technology Transfer*, vol. 20, no. 1, pp. 95–117, 2018.
- [259] H. Zhu, Y. Yu, W. Qi, S. Liu, Y. Weng, T. Yuan, and H. Li, “The research on fault restoration and refactoring for active distribution network,” in *2019 Chinese Automation Congress (CAC)*, pp. 4470–4474, IEEE, 2019.
- [260] V. N. Leonenko, N. V. Pertsev, and M. Artzrouni, “Using high performance algorithms for the hybrid simulation of disease dynamics on cpu and gpu.,” in *ICCS*, pp. 150–159, 2015.
- [261] S. Tichelaar, S. Ducasse, S. Demeyer, and O. Nierstrasz, “A meta-model for language-independent refactoring,” in *Proceedings International Symposium on Principles of Software Evolution*, pp. 154–164, IEEE, 2000.
- [262] T. Mens, T. Tourwe, and F. Munoz, “Beyond the refactoring browser: advanced tool support for software refactoring,” in *Sixth International Workshop on Principles of Software Evolution, 2003. Proceedings.*, pp. 39–44, 2003.
- [263] A. Garrido and R. Johnson, “Challenges of refactoring c programs,” in *Proceedings of the international workshop on Principles of software evolution*, pp. 6–14, 2002.
- [264] K. Mens and T. Tourwé, “Delving source code with formal concept analysis,” *Computer Languages, Systems & Structures*, vol. 31, no. 3-4, pp. 183–197, 2005.
- [265] Y. Y. Lee, N. Chen, and R. E. Johnson, “Drag-and-drop refactoring: intuitive and efficient program transformation,” in *2013 35th International Conference on Software Engineering (ICSE)*, pp. 23–32, IEEE, 2013.
- [266] V. U. Gómez, A. Kellens, K. Gybels, and T. D’Hondt, “Experiments with proactive declarative meta-programming,” in *Proceedings of the International Workshop on Smalltalk Technologies*, pp. 68–76, 2009.
- [267] M. Unterholzner, “Improving refactoring tools in smalltalk using static type inference,” *Science of Computer Programming*, vol. 96, pp. 70–83, 2014.
- [268] D. Vainsencher, “Mudpie: layers in the ball of mud,” *Computer Languages, Systems & Structures*, vol. 30, no. 1-2, pp. 5–19, 2004.
- [269] O. Callaú, R. Robbes, É. Tanter, D. Röthlisberger, and A. Bergel, “On the use of type predicates in object-oriented software: The case of smalltalk,” in *Proceedings of the 10th ACM Symposium on Dynamic languages*, pp. 135–146, 2014.
- [270] P. Tesone, G. Polito, L. Fabresse, N. Bouraqadi, and S. Ducasse, “Preserving instance state during refactorings in live environments,” *Future Generation Computer Systems*, 2020.

- [271] V. Arnaoudova and C. Constantinides, “Adaptation of refactoring strategies to multiple axes of modularity: characteristics and criteria,” in *2008 Sixth International Conference on Software Engineering Research, Management and Applications*, pp. 105–114, IEEE, 2008.
- [272] E. Rodrigues Jr, R. S. Durelli, R. W. de Bettio, L. Montecchi, and R. Terra, “Refactorings for replacing dynamic instructions with static ones: the case of ruby,” in *Proceedings of the XXII Brazilian Symposium on Programming Languages*, pp. 59–66, 2018.
- [273] P. Sommerlad, G. Zraggen, T. Corbat, and L. Felber, “Retaining comments when refactoring code,” in *Companion to the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, pp. 653–662, 2008.
- [274] T. Corbat, L. Felber, M. Stocker, and P. Sommerlad, “Ruby refactoring plug-in for eclipse,” in *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, pp. 779–780, 2007.
- [275] R. Chen and H. Miao, “A selenium based approach to automatic test script generation for refactoring javascript code,” in *2013 IEEE/ACIS 12th International Conference on Computer and Information Science (ICIS)*, pp. 341–346, IEEE, 2013.
- [276] H. V. Nguyen, H. A. Nguyen, T. T. Nguyen, and T. N. Nguyen, “Babelref: detection and renaming tool for cross-language program entities in dynamic web applications,” in *2012 34th International Conference on Software Engineering (ICSE)*, pp. 1391–1394, IEEE, 2012.
- [277] K. An and E. Tilevich, “D-goldilocks: Automatic redistribution of remote functionalities for performance and efficiency,” in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 251–260, IEEE, 2020.
- [278] C.-Y. Hsieh, C. Le My, K. T. Ho, and Y. C. Cheng, “Identification and refactoring of exception handling code smells in javascript,” *Journal of Internet Technology*, vol. 18, no. 6, pp. 1461–1471, 2017.
- [279] T. Mendes, M. T. Valente, and A. Hora, “Identifying utility functions in java and javascript,” in *2016 X Brazilian Symposium on Software Components, Architectures and Reuse (SBCARS)*, pp. 121–130, IEEE, 2016.
- [280] N. Van Es, Q. Stievenart, J. Nicolay, T. D’Hondt, and C. De Roover, “Implementing a performant scheme interpreter for the web in asm. js,” *Computer Languages, Systems & Structures*, vol. 49, pp. 62–81, 2017.
- [281] L. Gong, M. Pradel, and K. Sen, “Jitprof: pinpointing jit-unfriendly javascript code,” in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pp. 357–368, 2015.
- [282] A. M. Fard and A. Mesbah, “Jsnose: Detecting javascript code smells,” in *2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pp. 116–125, IEEE, 2013.

- [283] C. Schuster, T. Disney, and C. Flanagan, “Macrofication: Refactoring by reverse macro expansion,” in *European Symposium on Programming*, pp. 644–671, Springer, 2016.
- [284] J. Portner, J. Kerr, and B. Chu, “Moving target defense against cross-site scripting attacks (position paper),” in *International Symposium on Foundations and Practice of Security*, pp. 85–91, Springer, 2014.
- [285] G. Ortiz, J. A. Caravaca, A. García-de Prado, J. Boubeta-Puig, *et al.*, “Real-time context-aware microservice architecture for predictive analytics and smart decision-making,” *IEEE Access*, vol. 7, pp. 183177–183194, 2019.
- [286] M. U. Khan, M. Z. Iqbal, and S. Ali, “A heuristic-based approach to refactor cross-cutting behaviors in uml state machines,” in *2014 IEEE International Conference on Software Maintenance and Evolution*, pp. 557–560, IEEE, 2014.
- [287] R. Terra, M. T. Valente, and N. Anquetil, “A lightweight modularization process based on structural similarity,” in *2016 X Brazilian Symposium on Software Components, Architectures and Reuse (SBCARS)*, pp. 111–120, IEEE, 2016.
- [288] M. Bialy, M. Lawford, V. Pantelic, and A. Wassying, “A methodology for the simplification of tabular designs in model-based development,” in *2015 IEEE/ACM 3rd FME Workshop on Formal Methods in Software Engineering*, pp. 47–53, IEEE, 2015.
- [289] P. Langer, M. Wimmer, P. Brosch, M. Herrmannsdörfer, M. Seidl, K. Wieland, and G. Kappel, “A posteriori operation detection in evolving software models,” *Journal of Systems and Software*, vol. 86, no. 2, pp. 551–566, 2013.
- [290] A. T. Sampson, J. M. Bjorndalen, and P. S. Andrews, “Birds on the wall: Distributing a process-oriented simulation,” in *2009 IEEE Congress on Evolutionary Computation*, pp. 225–231, IEEE, 2009.
- [291] Y. Wang, H. Yu, Z. Zhu, W. Zhang, and Y. Zhao, “Automatic software refactoring via weighted clustering in method-level networks,” *IEEE Transactions on Software Engineering*, vol. 44, no. 3, pp. 202–236, 2017.
- [292] A. Ouni, M. Kessentini, M. Ó Cinnéide, H. Sahraoui, K. Deb, and K. Inoue, “More: A multi-objective refactoring recommendation approach to introducing design patterns and fixing code smells,” *Journal of Software: Evolution and Process*, vol. 29, no. 5, p. e1843, 2017.
- [293] H. Wang, M. Kessentini, and A. Ouni, “Bi-level identification of web service defects,” in *International Conference on Service-Oriented Computing*, pp. 352–368, Springer, Cham, 2016.
- [294] A. Ghannem, G. El Boussaidi, and M. Kessentini, “On the use of design defect examples to detect model refactoring opportunities,” *Software Quality Journal*, vol. 24, no. 4, pp. 947–965, 2016.

- [295] B. Amal, M. Kessentini, S. Bechikh, J. Dea, and L. B. Said, "On the use of machine learning and search-based software engineering for ill-defined fitness function: a case study on software refactoring," in *International Symposium on Search Based Software Engineering*, pp. 31–45, Springer, Cham, 2014.
- [296] M. Kessentini, A. Ouni, P. Langer, M. Wimmer, and S. Bechikh, "Search-based meta-model matching with structural and syntactic measures," *Journal of Systems and Software*, vol. 97, pp. 1–14, 2014.
- [297] M. Kessentini, R. Mahaouachi, and K. Ghedira, "What you like in design use to correct bad-smells," *Software Quality Journal*, vol. 21, no. 4, pp. 551–571, 2013.
- [298] A. Ghannem, M. Kessentini, and G. El Boussaidi, "Detecting model refactoring opportunities using heuristic search," in *Proceedings of the 2011 Conference of the Center for Advanced Studies on Collaborative Research*, pp. 175–187, 2011.
- [299] M. Boussaa, W. Kessentini, M. Kessentini, S. Bechikh, and S. B. Chikha, "Competitive coevolutionary code-smells detection," in *International Symposium on Search Based Software Engineering*, pp. 50–65, Springer, Berlin, Heidelberg, 2013.
- [300] E. Erturk and E. A. Sezer, "A comparison of some soft computing methods for software fault prediction," *Expert systems with applications*, vol. 42, no. 4, pp. 1872–1879, 2015.
- [301] C. S. Melo, M. M. L. da Cruz, A. D. F. Martins, T. Matos, J. M. da Silva Monteiro Filho, and J. de Castro Machado, "A practical guide to support change-proneness prediction," *Proceedings of the 21st International Conference on Enterprise Systems*, pp. 269–276, 2019.
- [302] L. Kumar, S. M. Satapathy, and A. Krishna, "Application of smote and lssvm with various kernels for predicting refactoring at method level," in *International Conference on Neural Information Processing*, pp. 150–161, Springer, 2018.
- [303] R. Hill and J. Rideout, "Automatic method completion," in *Proceedings. 19th International Conference on Automated Software Engineering, 2004.*, pp. 228–235, IEEE, 2004.
- [304] S. Haiduc, G. Bavota, A. Marcus, R. Oliveto, A. De Lucia, and T. Menzies, "Automatic query reformulations for text retrieval in software engineering," in *2013 35th International Conference on Software Engineering (ICSE)*, pp. 842–851, IEEE, 2013.
- [305] G. M. Ubayawardana and D. D. Karunaratna, "Bug prediction model using code smells," in *2018 18th International Conference on Advances in ICT for Emerging Regions (ICTer)*, pp. 70–77, IEEE, 2018.
- [306] Z. Aliyu, L. A. Rahim, and E. E. Mustapha, "A combine usability framework for imcat evaluation," in *2014 International Conference on Computer and Information Sciences (ICCOINS)*, pp. 1–5, IEEE, 2014.

- [307] A. Herranz and J. J. Moreno-Navarro, “Formal extreme (and extremely formal) programming,” in *International Conference on Extreme Programming and Agile Processes in Software Engineering*, pp. 88–96, Springer, 2003.
- [308] L. Quan, Q. Zongyan, and Z. Liu, “Formal use of design patterns and refactoring,” in *International Symposium on Leveraging Applications of Formal Methods, Verification and Validation*, pp. 323–338, Springer, 2008.
- [309] J. W. Ko and Y. J. Song, “Graph based model transformation verification using mapping patterns and graph comparison algorithm,” *International Journal of Advancements in Computing Technology*, vol. 4, no. 8, 2012.
- [310] T. Ruhroth and H. Wehrheim, “Model evolution and refinement,” *Science of Computer Programming*, vol. 77, no. 3, pp. 270–289, 2012.
- [311] S. Stepney, F. Polack, and I. Toyn, “Patterns to guide practical refactoring: examples targetting promotion in z,” in *International Conference of B and Z Users*, pp. 20–39, Springer, 2003.
- [312] T. v. Enckevoort, “Refactoring uml models: using openarchitectureware to measure uml model quality and perform pattern matching on uml models with ocl queries,” in *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, pp. 635–646, 2009.
- [313] D. Luciv, D. Koznov, H. A. Basit, and A. N. Terekhov, “On fuzzy repetitions detection in documentation reuse,” *Programming and Computer Software*, vol. 42, no. 4, pp. 216–224, 2016.
- [314] D. Arcelli, V. Cortellessa, and C. Trubiani, “Performance-based software model refactoring in fuzzy contexts,” in *International Conference on Fundamental Approaches to Software Engineering*, pp. 149–164, Springer, 2015.
- [315] C. Wang and S. Kang, “Adfl: An improved algorithm for american fuzzy lop in fuzz testing,” in *International Conference on Cloud Computing and Security*, pp. 27–36, Springer, 2018.
- [316] P. Lerthathairat and N. Prompoon, “An approach for source code classification using software metrics and fuzzy logic to improve code quality with refactoring techniques,” in *International Conference on Software Engineering and Computer Systems*, pp. 478–492, Springer, 2011.
- [317] Z. Avdagic, D. Boskovic, and A. Delic, “Code evaluation using fuzzy logic,” in *Proceedings of the 9th WSEAS International Conference on Fuzzy Systems*, pp. 20–25, World Scientific and Engineering Academy and Society (WSEAS), 2008.
- [318] H. Liu, J. Jin, Z. Xu, Y. Bu, Y. Zou, and L. Zhang, “Deep learning based code smell detection,” *IEEE Transactions on Software Engineering*, 2019.

- [319] Y. Wang, “What motivate software engineers to refactor source code? evidences from professional developers,” in *2009 IEEE International Conference on Software Maintenance*, pp. 413–416, IEEE, 2009.
- [320] J. Grigera, A. Garrido, and G. Rossi, “Kobold: web usability as a service,” in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 990–995, IEEE, 2017.
- [321] 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*, pp. 464–474, 2018.
- [322] M. Ó. Cinnéide and P. Nixon, “A methodology for the automated introduction of design patterns,” in *Proceedings IEEE International Conference on Software Maintenance-1999 (ICSM’99). ‘Software Maintenance for Business Change’ (Cat. No. 99CB36360)*, pp. 463–472, IEEE, 1999.
- [323] X.-L. Yang, D. Lo, X. Xia, Z.-Y. Wan, and J.-L. Sun, “What security questions do developers ask? a large-scale study of stack overflow posts,” *Journal of Computer Science and Technology*, vol. 31, no. 5, pp. 910–924, 2016.
- [324] C. Rosen and E. Shihab, “What are mobile developers asking about? a large scale study using stack overflow,” *Empirical Software Engineering*, vol. 21, no. 3, pp. 1192–1223, 2016.
- [325] M. Linares-Vásquez, B. Dit, and D. Poshyvanyk, “An exploratory analysis of mobile development issues using stack overflow,” in *2013 10th Working Conference on Mining Software Repositories (MSR)*, pp. 93–96, IEEE, 2013.
- [326] S. Beyer and M. Pinzger, “A manual categorization of android app development issues on stack overflow,” in *2014 IEEE International Conference on Software Maintenance and Evolution*, pp. 531–535, IEEE, 2014.
- [327] A. Barua, S. W. Thomas, and A. E. Hassan, “What are developers talking about? an analysis of topics and trends in stack overflow,” *Empirical Software Engineering*, vol. 19, no. 3, pp. 619–654, 2014.
- [328] C. Parnin, C. Treude, L. Grammel, and M.-A. Storey, “Crowd documentation: Exploring the coverage and the dynamics of api discussions on stack overflow,” *Georgia Institute of Technology, Tech. Rep*, 2012.
- [329] “Stack exchange creative commons data now hosted by the internet archive.” <https://stackoverflow.blog/2014/01/23/stack-exchange-cc-data-now-hosted-by-the-internet-archive/>. Accessed: 2019-04-05.
- [330] “Stack exchange data dump.” <https://archive.org/details/stackexchange>. Accessed: 2019-04-05.

- [331] Y. Liu, Z. Liu, T.-S. Chua, and M. Sun, “Topical word embeddings,” in *Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015.
- [332] R. Alghamdi and K. Alfalqi, “A survey of topic modeling in text mining,” *Int. J. Adv. Comput. Sci. Appl.(IJACSA)*, vol. 6, no. 1, 2015.
- [333] E. Loper and S. Bird, “Nltk: the natural language toolkit,” *arXiv preprint cs/0205028*, 2002.
- [334] V. Balakrishnan and E. Lloyd-Yemoh, “Stemming and lemmatization: a comparison of retrieval performances,” 2014.
- [335] E. Choi, N. Yoshida, R. G. Kula, and K. Inoue, “What do practitioners ask about code clone? a preliminary investigation of stack overflow.,” in *IWSC*, pp. 49–50, 2015.
- [336] M. Fowler, *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018.
- [337] I. D. Baxter, A. Yahin, L. Moura, M. Sant’Anna, and L. Bier, “Clone detection using abstract syntax trees,” in *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*, pp. 368–377, IEEE, 1998.
- [338] H. Gall, K. Hajek, and M. Jazayeri, “Detection of logical coupling based on product release history,” in *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*, pp. 190–198, IEEE, 1998.
- [339] M. Kim, V. Sazawal, D. Notkin, and G. C. Murphy, “An empirical study of code clone genealogies,” in *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pp. 187–196, 2005.
- [340] S. Kim and M. D. Ernst, “Prioritizing warning categories by analyzing software history,” in *Fourth International Workshop on Mining Software Repositories (MSR’07: ICSE Workshops 2007)*, pp. 27–27, IEEE, 2007.
- [341] T. Kamiya, S. Kusumoto, and K. Inoue, “Ccfinder: A multilinguistic token-based code clone detection system for large scale source code,” *IEEE Transactions on Software Engineering*, vol. 28, no. 7, pp. 654–670, 2002.
- [342] N. Moha, Y.-G. Guéhéneuc, L. Duchien, and A.-F. Le Meur, “DECOR: A method for the specification and detection of code and design smells,” *IEEE Transactions on Software Engineering*, vol. 36, no. 1, pp. 20–36, 2009.
- [343] M. Kessentini, S. Vaucher, and H. Sahraoui, “Deviance from perfection is a better criterion than closeness to evil when identifying risky code,” in *25th International Conference on Automated Software engineering (ASE)*, pp. 113–122, 2010.
- [344] R. L. Bocchino Jr, V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian, “A type and effect system for deterministic parallel java,” in *ACM Sigplan Notices*, vol. 44, pp. 97–116, ACM, 2009.

- [345] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson, “Automated detection of refactorings in evolving components,” in *European Conference on Object-Oriented Programming*, pp. 404–428, Springer, 2006.
- [346] A. Ouni, M. Kessentini, and H. Sahraoui, “Search-based refactoring using recorded code changes,” in *17th European Conference on Software Maintenance and Reengineering (CSMR)*, pp. 221–230, 2013.
- [347] N. Tsantalis and A. Chatzigeorgiou, “Identification of move method refactoring opportunities,” *IEEE Transactions on Software Engineering*, vol. 35, no. 3, pp. 347–367, 2009.
- [348] A. Ouni, M. Kessentini, K. Inoue, and M. O. Cinnéide, “Search-based web service antipatterns detection,” *IEEE Transactions on Services Computing*, 2015.
- [349] M. K. M. M. G. Marwa Daagi, Ali Ouni and S. Bouktif, “Web service interface decomposition using formal concept analysis,” in *International Conference on Web Services ICWS2017*, pp. 171–180, IEEE, 2017.
- [350] T. H. Hanzhang Wang, Marouane Kessentini and A. Ouni, “On the value of quality of service attributes for detecting bad design practices,” in *International Conference on Web Services ICWS2017*, pp. 242–251, IEEE, 2017.
- [351] M. Kessentini and H. Wang, “Detecting refactorings among multiple web service releases: A heuristic-based approach,” in *International Conference on Web Services ICWS2017*, pp. 263–272, IEEE, 2017.
- [352] M. K. S. B. Ali Ouni, Marwa Daagi and M. M. Gammoudi, “A machine learning-based approach to detect web service design defects,” in *International Conference on Web Services ICWS2017*, pp. 382–391, IEEE, 2017.
- [353] J. D. Marouane Kessentini, Hanzhang Wang and A. Ouni, “Improving web services desing quality using heuristic search and machine learning,” in *International Conference on Web Services ICWS2017*, pp. 410–419, IEEE, 2017.
- [354] J. Kerievsky, *Refactoring to Patterns*. Pearson Deutschland GmbH, 2005.
- [355] M. Feathers, *Working Effectively with Legacy Code: WORK EFFECT LEG CODE \_p1*. Prentice Hall Professional, 2004.
- [356] D. Dig, “A refactoring approach to parallelism,” *IEEE software*, vol. 28, no. 1, pp. 17–22, 2010.
- [357] M. Kim and D. Notkin, “Discovering and representing systematic code changes,” in *Proceedings of the 31st International Conference on Software Engineering*, pp. 309–319, IEEE Computer Society, 2009.
- [358] Y. Cai, R. Kazman, C. Jaspan, and J. Aldrich, “Introducing tool-supported architecture review into software design education,” in *2013 26th International Conference on Software Engineering Education and Training (CSEET)*, pp. 70–79, IEEE, 2013.



- [359] A. Yamashita and L. Moonen, “Do developers care about code smells? an exploratory survey,” in *20th Working Conference on Reverse Engineering (WCRE)*, pp. 242–251, IEEE, 2013.
- [360] A. Telea and L. Voinea, “Visual software analytics for the build optimization of large-scale software systems,” *Computational Statistics*, vol. 26, no. 4, pp. 635–654, 2011.
- [361] D. Sahin, M. Kessentini, S. Bechikh, and K. Deb, “Code-smell detection as a bilevel problem,” *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 24, no. 1, p. 6, 2014.
- [362] L. Xiao, Y. Cai, and R. Kazman, “Titan: A toolset that connects software architecture with quality analysis,” in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 763–766, 2014.
- [363] Y. Lin and D. Dig, “A study and toolkit of CHECK-THEN-ACT idioms of java concurrent collections,” *Softw. Test., Verif. Reliab.*, vol. 25, no. 4, pp. 397–425, 2015.
- [364] 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 (TOSEM)*, vol. 25, no. 3, pp. 1–53, 2016.
- [365] 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.
- [366] E. resource for C# parallel programmers. July’14, <http://learnparallelism.net>.
- [367] M. Ó. Cinnéide, D. Boyle, and I. H. Moghadam, “Automated refactoring for testability,” in *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on*, pp. 437–443, IEEE, 2011.
- [368] E. Murphy-Hill, C. Parnin, and A. P. Black, “How we refactor, and how we know it,” in *Proceedings of the International Conference on Software Engineering*, pp. 287–297, 2009.
- [369] Y. Lin and D. Dig, “CHECK-THEN-ACT Misuse of Java Concurrent Collections,” in *International Conference on Software Testing, Verification and Validation (ICST)*, pp. 164–173, 2013.
- [370] K. O. Elish and M. Alshayeb, “A classification of refactoring methods based on software quality attributes,” *Arabian Journal for Science and Engineering*, vol. 36, no. 7, pp. 1253–1267, 2011.
- [371] B. Du Bois, P. Van Gorp, A. Amsel, N. Van Eetvelde, H. Stenten, S. Demeyer, and T. Mens, “A discussion of refactoring in research and practice,” *Reporte Técnico. Universidad de Antwerpen, Bélgica*, 2004.

- [372] T. Mens, A. Van Deursen, *et al.*, “Refactoring: Emerging trends and open problems,” in *Proceedings First International Workshop on REFactoring: Achievements, Challenges, Effects (REFACE)*. University of Waterloo, 2003.
- [373] M. Misbhauddin and M. Alshayeb, “Uml model refactoring: a systematic literature review,” *Empirical Software Engineering*, vol. 20, no. 1, pp. 206–251, 2015.
- [374] M. Abebe and C.-J. Yoo, “Trends, opportunities and challenges of software refactoring: A systematic literature review,” *International Journal of Software Engineering and Its Applications*, vol. 8, no. 6, pp. 299–318, 2014.
- [375] A. A. B. Baqais and M. Alshayeb, “Automatic software refactoring: a systematic literature review,” *Software Quality Journal*, pp. 1–44, 2019.
- [376] A. K. Saha, R. K. Saha, and K. A. Schneider, “A discriminative model approach for suggesting tags automatically for stack overflow questions,” in *Proceedings of the 10th Working Conference on Mining Software Repositories*, pp. 73–76, IEEE Press, 2013.
- [377] G. Pinto, W. Torres, and F. Castor, “A study on the most popular questions about concurrent programming,” in *Proceedings of the 6th Workshop on Evaluation and Usability of Programming Languages and Tools*, pp. 39–46, 2015.
- [378] Y. Jin, X. Yang, R. G. Kula, E. Choi, K. Inoue, and H. Iida, “Quick trigger on stack overflow: a study of gamification-influenced member tendencies,” in *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pp. 434–437, IEEE, 2015.
- [379] F. Tian, P. Liang, and M. A. Babar, “How developers discuss architecture smells? an exploratory study on stack overflow,” in *2019 IEEE International Conference on Software Architecture (ICSA)*, pp. 91–100, IEEE, 2019.
- [380] A. Tahir, A. Yamashita, S. Licorish, J. Dietrich, and S. Counsell, “Can you tell me if it smells? a study on how developers discuss code smells and anti-patterns in stack overflow,” in *Proceedings of the 22nd International Conference on Evaluation and Assessment in Software Engineering 2018*, pp. 68–78, 2018.
- [381] G. H. Pinto and F. Kamei, “What programmers say about refactoring tools? an empirical investigation of stack overflow,” in *Proceedings of the 2013 ACM workshop on Workshop on refactoring tools*, pp. 33–36, 2013.
- [382] W. Martin, F. Sarro, Y. Jia, Y. Zhang, and M. Harman, “A survey of app store analysis for software engineering,” *IEEE transactions on software engineering*, vol. 43, no. 9, pp. 817–847, 2016.
- [383] K. Mao, L. Capra, M. Harman, and Y. Jia, “A survey of the use of crowdsourcing in software engineering,” *Journal of Systems and Software*, vol. 126, pp. 57–84, 2017.
- [384] N. Chen, J. Lin, S. C. Hoi, X. Xiao, and B. Zhang, “Ar-miner: mining informative reviews for developers from mobile app marketplace,” in *Proceedings of the 36th International Conference on Software Engineering*, pp. 767–778, ACM, 2014.

- [385] E. Guzman and W. Maalej, “How do users like this feature? a fine grained sentiment analysis of app reviews,” in *2014 IEEE 22nd international requirements engineering conference (RE)*, pp. 153–162, IEEE, 2014.
- [386] C. Iacob and R. Harrison, “Retrieving and analyzing mobile apps feature requests from online reviews,” in *2013 10th working conference on mining software repositories (MSR)*, pp. 41–44, IEEE, 2013.
- [387] S. McIlroy, N. Ali, H. Khalid, and A. E. Hassan, “Analyzing and automatically labelling the types of user issues that are raised in mobile app reviews,” *Empirical Software Engineering*, vol. 21, no. 3, pp. 1067–1106, 2016.
- [388] S. Panichella, A. Di Sorbo, E. Guzman, C. A. Visaggio, G. Canfora, and H. C. Gall, “How can i improve my app? classifying user reviews for software maintenance and evolution,” in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 281–290, IEEE, 2015.
- [389] C. Gao, H. Xu, J. Hu, and Y. Zhou, “Ar-tracker: Track the dynamics of mobile apps via user review mining,” in *2015 IEEE Symposium on Service-Oriented System Engineering*, pp. 284–290, IEEE, 2015.
- [390] A. Bacchelli, M. Lanza, and R. Robbes, “Linking e-mails and source code artifacts,” in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pp. 375–384, 2010.
- [391] T. Dasgupta, M. Grechanik, E. Moritz, B. Dit, and D. Poshyvanyk, “Enhancing software traceability by automatically expanding corpora with relevant documentation,” in *2013 IEEE International Conference on Software Maintenance*, pp. 320–329, IEEE, 2013.
- [392] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, R. Oliveto, M. Di Penta, and D. Poshyvanyk, “Mining energy-greedy api usage patterns in android apps: an empirical study,” in *Proceedings of the 11th Working Conference on Mining Software Repositories*, pp. 2–11, 2014.
- [393] Y. Zhang and D. Hou, “Extracting problematic api features from forum discussions,” in *2013 21st International Conference on Program Comprehension (ICPC)*, pp. 142–151, IEEE, 2013.
- [394] A. Ciurumelea, A. Schaufelbühl, S. Panichella, and H. C. Gall, “Analyzing reviews and code of mobile apps for better release planning,” in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 91–102, IEEE, 2017.
- [395] F. Palomba, M. Linares-Vásquez, G. Bavota, R. Oliveto, M. Di Penta, D. Poshyvanyk, and A. De Lucia, “Crowdsourcing user reviews to support the evolution of mobile apps,” *Journal of Systems and Software*, vol. 137, pp. 143–162, 2018.

- [396] G. Grano, A. Di Sorbo, F. Mercaldo, C. A. Visaggio, G. Canfora, and S. Panichella, “Android apps and user feedback: a dataset for software evolution and quality improvement,” in *Proceedings of the 2nd ACM SIGSOFT International Workshop on App Market Analytics*, pp. 8–11, ACM, 2017.
- [397] A. Di Sorbo, S. Panichella, C. V. Alexandru, C. A. Visaggio, and G. Canfora, “Surf: summarizer of user reviews feedback,” in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pp. 55–58, IEEE, 2017.
- [398] E. Noei, M. D. Syer, Y. Zou, A. E. Hassan, and I. Keivanloo, “A study of the relation of mobile device attributes with the user-perceived quality of android apps,” *Empirical Software Engineering*, vol. 22, no. 6, pp. 3088–3116, 2017.
- [399] J. Zhu, Y. Kang, Z. Zheng, and M. R. Lyu, “A clustering-based qos prediction approach for web service recommendation,” in *Object/Component/Service-Oriented Real-Time Distributed Computing Workshops (ISORCW), 2012 15th IEEE International Symposium on*, pp. 93–98, IEEE, 2012.
- [400] M. Silic, G. Delac, I. Krka, and S. Srbljic, “Scalable and accurate prediction of availability of atomic web services,” *IEEE Transactions on Services Computing*, vol. 7, no. 2, pp. 252–264, 2013.
- [401] J. Zhu, P. He, Q. Xie, Z. Zheng, and M. R. Lyu, “Carp: Context-aware reliability prediction of black-box web services,” in *2017 IEEE International Conference on Web Services (ICWS)*, pp. 17–24, IEEE, 2017.
- [402] Z. Zheng, H. Ma, M. R. Lyu, and I. King, “Qos-aware web service recommendation by collaborative filtering,” *IEEE Transactions on services computing*, vol. 4, no. 2, pp. 140–152, 2010.
- [403] M. Silic, G. Delac, and S. Srbljic, “Prediction of atomic web services reliability for qos-aware recommendation,” *IEEE Transactions on services Computing*, vol. 8, no. 3, pp. 425–438, 2014.
- [404] M. Kessentini, P. Langer, and M. Wimmer, “Searching models, modeling search: On the synergies of sbse and mde,” in *2013 1st International Workshop on Combining Modelling and Search-Based Software Engineering (CMSBSE)*, pp. 51–54, IEEE, 2013.
- [405] A. Ouni, M. Kessentini, S. Bechikh, and H. Sahraoui, “Prioritizing code-smells correction tasks using chemical reaction optimization,” *Software Quality Journal*, vol. 23, no. 2, pp. 323–361, 2015.
- [406] 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, no. 2, pp. 529–552, 2017.
- [407] M. Fleck, J. Troya, M. Kessentini, M. Wimmer, and B. Alkhazi, “Model transformation modularization as a many-objective optimization problem,” *IEEE Transactions on Software Engineering*, vol. 43, no. 11, pp. 1009–1032, 2017.

- [408] Q. Xie, K. Wu, J. Xu, P. He, and M. Chen, “Personalized context-aware qos prediction for web services based on collaborative filtering,” in *International Conference on Advanced Data Mining and Applications*, pp. 368–375, Springer, 2010.
- [409] L. Shao, J. Zhang, Y. Wei, J. Zhao, B. Xie, and H. Mei, “Personalized qos prediction for web services via collaborative filtering,” in *Web Services, 2007. ICWS 2007. IEEE International Conference on*, pp. 439–446, IEEE, 2007.
- [410] Q. Zhang, C. Ding, and C.-H. Chi, “Collaborative filtering based service ranking using invocation histories,” in *Web Services (ICWS), 2011 IEEE International Conference on*, pp. 195–202, IEEE, 2011.
- [411] Z. Zheng, H. Ma, M. R. Lyu, and I. King, “Wsrec: A collaborative filtering based web service recommender system,” in *Web Services, 2009. ICWS 2009. IEEE International Conference on*, pp. 437–444, IEEE, 2009.
- [412] L. Li, M. Rong, and G. Zhang, “A web service qos prediction approach based on multi-dimension qos,” in *Computer Science & Education (ICCSE), 2011 6th International Conference on*, pp. 1319–1322, IEEE, 2011.
- [413] L. Chen, Y. Feng, J. Wu, and Z. Zheng, “An enhanced qos prediction approach for service selection,” in *Services Computing (SCC), 2011 IEEE International Conference on*, pp. 727–728, IEEE, 2011.
- [414] Y. Jiang, J. Liu, M. Tang, and X. F. Liu, “An effective web service recommendation method based on personalized collaborative filtering,” in *2011 IEEE International Conference on Web Services*, pp. 211–218, IEEE, 2011.
- [415] X. Chen, X. Liu, Z. Huang, and H. Sun, “Regionknn: A scalable hybrid collaborative filtering algorithm for personalized web service recommendation,” in *Web Services (ICWS), 2010 IEEE International Conference on*, pp. 9–16, IEEE, 2010.
- [416] M. Zhang, X. Liu, R. Zhang, and H. Sun, “A web service recommendation approach based on qos prediction using fuzzy clustering,” in *Services Computing (SCC), 2012 IEEE Ninth International Conference on*, pp. 138–145, IEEE, 2012.
- [417] J. Ge, Z. Chen, J. Peng, T. Li, and L. Zhang, “Web service recommendation based on qos prediction method,” in *Cognitive Informatics (ICCI), 2010 9th IEEE International Conference on*, pp. 109–112, IEEE, 2010.
- [418] V. Cardellini, E. Casalicchio, V. Grassi, and F. L. Presti, “Flow-based service selection for web service composition supporting multiple qos classes,” in *Web Services, 2007. ICWS 2007. IEEE International Conference on*, pp. 743–750, IEEE, 2007.
- [419] J. El Haddad, M. Manouvrier, G. Ramirez, and M. Rukoz, “Qos-driven selection of web services for transactional composition,” in *2008 IEEE International Conference on Web Services*, pp. 653–660, IEEE, 2008.

- [420] Z. Zheng and M. R. Lyu, “A distributed replication strategy evaluation and selection framework for fault tolerant web services,” in *Web Services, 2008. ICWS’08. IEEE International Conference on*, pp. 145–152, IEEE, 2008.
- [421] D. Romano and M. Pinzger, “Analyzing the evolution of web services using fine-grained changes,” in *Web Services (ICWS), 2012 IEEE 19th International Conference on*, pp. 392–399, IEEE, 2012.
- [422] M. Fokaefs, R. Mikhael, N. Tsantalis, E. Stroulia, and A. Lau, “An empirical study on web service evolution,” in *Web Services (ICWS), 2011 IEEE International Conference on*, pp. 49–56, IEEE, 2011.
- [423] J. M. Rodriguez, M. Crasso, A. Zunino, and M. Campo, “Automatically detecting opportunities for web service descriptions improvement,” in *Conference on e-Business, e-Services and e-Society*, pp. 139–150, Springer, 2010.
- [424] J. Král and M. Zemlicka, “Popular SOA Antipatterns,” in *Computation World: Future Computing, Service Computation, Cognitive, Adaptive, Content, Patterns*, pp. 271–276, 2009.
- [425] N. Moha, F. Palma, M. Nayrolles, B. J. Conseil, Y.-G. Guéhéneuc, B. Baudry, and J.-M. Jézéquel, “Specification and detection of soa antipatterns,” in *Service-Oriented Computing*, pp. 1–16, Springer, 2012.
- [426] F. Palma, N. Moha, G. Tremblay, and Y.-G. Guéhéneuc, “Specification and detection of soa antipatterns in web services,” in *European Conference on Software Architecture*, pp. 58–73, Springer, 2014.
- [427] J. M. Rodriguez, M. Crasso, C. Mateos, and A. Zunino, “Best practices for describing, consuming, and discovering web services: a comprehensive toolset,” *Software: Practice and Experience*, vol. 43, no. 6, pp. 613–639, 2013.
- [428] C. Mateos, J. M. Rodriguez, and A. Zunino, “A tool to improve code-first web services discoverability through text mining techniques,” *Software: Practice and Experience*, vol. 45, no. 7, pp. 925–948, 2015.
- [429] C. Mateos, A. Zunino, S. Misra, D. Anabalón, and A. Flores, “Keeping web service interface complexity low using an oo metric-based early approach,” in *2016 XLII Latin American Computing Conference (CLEI)*, pp. 1–12, IEEE, 2016.
- [430] Z. Li, M. Zhang, Z. Zhu, Y. Chen, A. G. Greenberg, and Y.-M. Wang, “Webprophet: Automating performance prediction for web services,” in *NSDI*, vol. 10, pp. 143–158, 2010.
- [431] M. Tariq, A. Zeitoun, V. Valancius, N. Feamster, and M. Ammar, “Answering what-if deployment and configuration questions with wise,” in *ACM SIGCOMM Computer Communication Review*, vol. 38, pp. 99–110, ACM, 2008.

- [432] S. Chen, K. R. Joshi, M. A. Hiltunen, W. H. Sanders, and R. D. Schlichting, “Link gradients: Predicting the impact of network latency on multitier applications,” in *INFOCOM 2009, IEEE*, pp. 2258–2266, IEEE, 2009.
- [433] N. Munaiah, S. Kroh, C. Cabrey, and M. Nagappan, “Curating github for engineered software projects,” *Empirical Software Engineering*, vol. 22, no. 6, pp. 3219–3253, 2017.
- [434] V. Thakur, M. Kessentini, and T. Sharma, “Qscored: An open platform for code quality ranking and visualization,” in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 818–821, 2020.
- [435] P. Pickerill, J. H. Joshua, O. Mirosław, M. Michał, and S. Mirosław, “Phantom: Curating github for engineered software projects using time-series clustering,” *Empirical Software Engineering*, vol. 25, no. 4, pp. 2897–2929, 2020.
- [436] K. Lochmann, “A benchmarking-inspired approach to determine threshold values for metrics,” *ACM SIGSOFT Software Engineering Notes*, vol. 37, no. 6, pp. 1–8, 2012.
- [437] A. Chatzigeorgiou and E. Stiakakis, “Benchmarking library and application software with data envelopment analysis,” *Software Quality Journal*, vol. 19, no. 3, pp. 553–578, 2011.
- [438] J. P. Correia and J. Visser, “Benchmarking technical quality of software products,” in *2008 15th Working Conference on Reverse Engineering*, pp. 297–300, IEEE, 2008.
- [439] T. Kalibera, J. Lehotsky, D. Majda, B. Repcek, M. Tomcanyi, A. Tomecek, P. Tuma, and J. Urban, “Automated benchmarking and analysis tool,” in *Proceedings of the 1st international conference on Performance evaluation methodolgies and tools*, pp. 5–es, 2006.
- [440] J. Moses, “Benchmarking quality measurement,” *Software Quality Journal*, vol. 15, no. 4, pp. 449–462, 2007.
- [441] H. Gruber, R. Plösch, and M. Saft, “On the validity of benchmarking for evaluating code quality,” *IWSM/MENSURA*, vol. 10, 2010.
- [442] M. Harman and P. McMinn, “A theoretical and empirical study of search-based testing: Local, global, and hybrid search,” *IEEE Transactions on Software Engineering*, vol. 36, no. 2, pp. 226–247, 2009.
- [443] 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*, pp. 49–58, 2012.
- [444] T. Mariani and S. R. Vergilio, “A systematic review on search-based refactoring,” *Information and Software Technology*, vol. 83, pp. 14–34, 2017.

- [445] M. O’Keeffe and M. Ó. Cinnéide, “A stochastic approach to automated design improvement,” in *Proceedings of the 2nd international conference on Principles and practice of programming in Java*, pp. 59–62, Computer Science Press, Inc., 2003.
- [446] A. Ouni, M. Kessentini, H. Sahraoui, and M. S. Hamdi, “Search-based refactoring: Towards semantics preservation,” in *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pp. 347–356, IEEE, 2012.
- [447] A. Ouni, M. Kessentini, and H. Sahraoui, “Search-based refactoring using recorded code changes,” in *2013 17th European Conference on Software Maintenance and Reengineering*, pp. 221–230, IEEE, 2013.
- [448] A. Ouni, M. Kessentini, H. Sahraoui, and M. S. Hamdi, “The use of development history in software refactoring using a multi-objective evolutionary algorithm,” in *Proceedings of the 15th annual conference on Genetic and evolutionary computation*, pp. 1461–1468, ACM, 2013.
- [449] 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)*, (Fortaleza, Brazil), pp. 74—83, ACM, 2017.
- [450] 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 *Proceedings of 13th the ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM ’19)*, (Porto de Galinhas, Brazil), pp. 1–11, IEEE, 2019.
- [451] 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.
- [452] H. Melton and E. Tempero, “Identifying refactoring opportunities by identifying dependency cycles,” in *Proceedings of the 29th Australasian Computer Science Conference (ACSC ’06)*, (Australia), pp. 35–41, ACM, 2006.
- [453] N. Yoshida, Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue, “On refactoring support based on code clone dependency relation,” in *Proceedings of the 11th IEEE International Software Metrics Symposium (METRICS ’05)*, (Como, Italy), pp. 10–pp, IEEE, 2005.
- [454] V. P. L. Oliveira, E. F. Souza, C. Le Goues, and C. G. Camilo-Junior, “Improved crossover operators for genetic programming for program repair,” in *International Symposium on Search Based Software Engineering*, pp. 112–127, Springer, 2016.
- [455] F.-l. Zhu, H.-w. Deng, F. Li, and S.-g. Cheng, “Improved crossover operators and mutation operators to prevent premature convergence,” *Sci Technol Eng*, vol. 10, no. 6, pp. 1540–1542, 2010.



- [456] M. A. Abido and A. Elazouni, “Improved crossover and mutation operators for genetic-algorithm project scheduling,” in *2009 IEEE Congress on Evolutionary Computation*, pp. 1865–1872, IEEE, 2009.
- [457] G. Fraser and A. Arcuri, “The seed is strong: Seeding strategies in search-based software testing,” in *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pp. 121–130, IEEE, 2012.
- [458] C. Grothoff, J. Palsberg, and J. Vitek, “Encapsulating objects with confined types,” *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 29, no. 6, p. 32, 2007.
- [459] P. Bouillon, E. Großkinsky, and F. Steimann, “Controlling accessibility in agile projects with the access modifier modifier,” in *International Conference on Objects, Components, Models and Patterns*, pp. 41–59, Springer, 2008.
- [460] A. Müller, “Bytecode analysis for checking java access modifiers,” in *Work in Progress and Poster Session, 8th Int. Conf. on Principles and Practice of Programming in Java (PPPJ 2010), Vienna, Austria*, 2010.
- [461] F. Steimann and A. Thies, “From public to private to absent: Refactoring java programs under constrained accessibility,” in *European Conference on Object-Oriented Programming*, pp. 419–443, Springer, 2009.
- [462] C. Zoller and A. Schmolitzky, “Measuring inappropriate generosity with access modifiers in java systems,” in *2012 Joint Conference of the 22nd International Workshop on Software Measurement and the 2012 Seventh International Conference on Software Process and Product Measurement*, pp. 43–52, IEEE, 2012.
- [463] K. Kobori, M. Matsushita, and K. Inoue, “Evolution analysis for accessibility excessiveness in java,” in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pp. 83–90, IEEE, 2015.
- [464] S. A. Vidal, A. Bergel, C. Marcos, and J. A. Díaz-Pace, “Understanding and addressing exhibitionism in java empirical research about method accessibility,” *Empirical Software Engineering*, vol. 21, no. 2, pp. 483–516, 2016.
- [465] S. Vidal, A. Bergel, J. A. Díaz-Pace, and C. Marcos, “Over-exposed classes in java: An empirical study,” *Computer Languages, Systems & Structures*, vol. 46, pp. 1–19, 2016.
- [466] A. Agrawal and R. Khan, “Assessing impact of cohesion on security-an object oriented design perspective,” *Pensee*, vol. 76, no. 2, 2014.
- [467] A. Agrawal and R. Khan, “Role of coupling in vulnerability propagation,” *Software Engineering*, vol. 2, no. 1, pp. 60–68, 2012.

- [468] B. Alshammari, C. Fidge, and D. Corney, “Security metrics for object-oriented class designs,” in *2009 Ninth International Conference on Quality Software*, pp. 11–20, IEEE, 2009.
- [469] W. Wang, K. R. Mahakala, A. Gupta, N. Hussein, and Y. Wang, “A linear classifier based approach for identifying security requirements in open source software development,” *Journal of Industrial Information Integration*, 2018.
- [470] J. L. Wright, M. McQueen, and L. Wellman, “Analyses of two end-user software vulnerability exposure metrics (extended version),” *Information Security Technical Report*, vol. 17, no. 4, pp. 173–184, 2013.
- [471] A. K. Srivastava and S. Kumar, “An effective computational technique for taxonomic position of security vulnerability in software development,” *Journal of Computational Science*, vol. 25, pp. 388–396, 2018.
- [472] B. Alshammari, C. Fidge, and D. Corney, “Security metrics for object-oriented class designs,” in *2009 Ninth International Conference on Quality Software*, pp. 11–20, IEEE, 2009.
- [473] I. Chowdhury, B. Chan, and M. Zulkernine, “Security metrics for source code structures,” in *Proceedings of the fourth international workshop on Software engineering for secure systems*, pp. 57–64, ACM, 2008.
- [474] K. Maruyama and T. Omori, “A security-aware refactoring tool for java programs,” in *Proceedings of the 4th Workshop on Refactoring Tools*, pp. 22–28, ACM, 2011.
- [475] B. Alshammari, C. Fidge, and D. Corney, “Assessing the impact of refactoring on security-critical object-oriented designs,” in *2010 Asia Pacific Software Engineering Conference*, pp. 186–195, Nov 2010.
- [476] B. Alshammari, C. Fidge, and D. Corney, “Security metrics for object-oriented class designs,” in *2009 Ninth International Conference on Quality Software*, pp. 11–20, Aug 2009.
- [477] G. McGraw and E. W. Felten, *Securing Java: Getting Down to Business with Mobile Code*. New York, NY, USA: John Wiley & Sons, Inc., 1999.
- [478] S. Ghaith and M. Ó Cinnéide, “Improving software security using search-based refactoring,” in *Search Based Software Engineering* (G. Fraser and J. Teixeira de Souza, eds.), (Berlin, Heidelberg), pp. 121–135, Springer Berlin Heidelberg, 2012.
- [479] S. Ghaith and M. Ó. Cinnéide, “Improving software security using search-based refactoring,” in *International Symposium on Search Based Software Engineering*, pp. 121–135, Springer, 2012.
- [480] 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.

- [481] M. O’Keeffe and M. O. Cinnéide, “Search-based refactoring: An empirical study,” *Journal of Software Maintenance and Evolution*, vol. 20, no. 5, pp. 345–364, 2008.
- [482] 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*, pp. 49–58, ACM, 2012.
- [483] J. Jürjens, *Secure systems development with UML*. Springer Science & Business Media, 2005.
- [484] M. Fowler, *Refactoring: Improving the Design of Existing Programs*. Addison-Wesley Professional, 1 ed., 1999.
- [485] G. Suryanarayana, G. Samarthyan, and T. Sharma, *Refactoring for software design smells: managing technical debt*. Morgan Kaufmann, 2014.
- [486] T. Sharma, “Designitejava (enterprise),” Sept. 2019. <http://www.designite-tools.com/designitejava>.
- [487] T. Sharma, P. Singh, and D. Spinellis, “An empirical investigation on the relationship between design and architecture smells,” *Empirical Software Engineering (EMSE)*, Aug. 2020.
- [488] T. Sharma, M. Fragkoulis, and D. Spinellis, “House of cards: Code smells in open-source c# repositories,” in *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pp. 424–429, 2017.
- [489] W. H. Brown, R. C. Malveau, and T. J. Mowbray, “Antipatterns: refactoring software, architectures, and projects in crisis,” *John Wiley and Sons*, 1998.
- [490] B. Dudley, J. Krozak, K. Wittkopf, S. Asbury, and D. Osborne, *J2EE Antipatterns*. John Wiley; Sons, Inc., 2003.
- [491] J. L. Ordiales Coscia, C. M. Mateos Diaz, M. P. Crasso, and A. O. Zunino Suarez, “Anti-pattern free code-first web services for state-of-the-art java wsdl generation tools,” *International Journal of Web and Grid Services*, vol. 9, no. 2, pp. 107–126, 2013.
- [492] C. Mateos, M. Crasso, A. Zunino, and J. L. O. Coscia, “Detecting wsdl bad practices in code-first web services,” *International Journal of Web and Grid Services*, vol. 7, no. 4, p. 357, 2011.
- [493] J. L. O. Coscia, C. Mateos, M. Crasso, and A. Zunino, “Refactoring code-first web services for early avoiding wsdl anti-patterns: Approach and comprehensive assessment,” *Science of Computer Programming*, vol. 89, pp. 374–407, 2014.
- [494] J. Kral and M. Zemlicka, “The most important service-oriented antipatterns,” in *Software Engineering Advances, 2007. ICSEA 2007. International Conference on*, pp. 29–29, IEEE, 2007.

- [495] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, “A fast and elitist multiobjective genetic algorithm: Nsga-ii,” *IEEE transactions on evolutionary computation*, vol. 6, no. 2, pp. 182–197, 2002.
- [496] 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.
- [497] M. W. Mkaouer, M. Kessentini, M. Ó. Cinnéide, S. Hayashi, and K. Deb, “A robust multi-objective approach to balance severity and importance of refactoring opportunities,” *Empirical Software Engineering*, vol. 22, no. 2, pp. 894–927, 2017.
- [498] R. Shatnawi and W. Li, “An empirical assessment of refactoring impact on software quality using a hierarchical quality model,” *International Journal of Software Engineering and Its Applications*, vol. 5, no. 4, pp. 127–149, 2011.
- [499] J. Kerievsky, *Refactoring to patterns*. Pearson Deutschland GmbH, 2005.
- [500] M. Kessentini, H. Sahraoui, and M. Boukadoum, “Example-based model-transformation testing,” *Automated Software Engineering*, vol. 18, no. 2, pp. 199–224, 2011.
- [501] M. Kessentini, M. Wimmer, H. Sahraoui, and M. Boukadoum, “Generating transformation rules from examples for behavioral models,” in *Proceedings of the Second International Workshop on Behaviour Modelling: Foundation and Applications*, p. 2, ACM, 2010.
- [502] S. Kalboussi, S. Bechikh, M. Kessentini, and L. B. Said, “Preference-based many-objective evolutionary testing generates harder test cases for autonomous agents,” in *International Symposium on Search Based Software Engineering*, pp. 245–250, Springer, Berlin, Heidelberg, 2013.
- [503] U. Mansoor, M. Kessentini, P. Langer, M. Wimmer, S. Bechikh, and K. Deb, “Momm: Multi-objective model merging,” *Journal of Systems and Software*, vol. 103, pp. 423–439, 2015.
- [504] M. Hüttermann, “Beginning devops for developers,” in *DevOps for Developers*, pp. 3–13, Springer, 2012.
- [505] G. G. Claps, R. B. Svensson, and A. Aurum, “On the journey to continuous deployment: Technical and social challenges along the way,” *Information and Software technology*, vol. 57, pp. 21–31, 2015.
- [506] L. Villarroel, G. Bavota, B. Russo, R. Oliveto, and M. Di Penta, “Release planning of mobile apps based on user reviews,” in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pp. 14–24, IEEE, 2016.

- [507] A. Di Sorbo, S. Panichella, C. V. Alexandru, J. Shimagaki, C. A. Visaggio, G. Canfora, and H. C. Gall, “What would users change in my app? summarizing app reviews for recommending software changes,” in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 499–510, ACM, 2016.
- [508] R. K. Saha, M. Lease, S. Khurshid, and D. E. Perry, “Improving bug localization using structured information retrieval,” in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 345–355, IEEE, 2013.
- [509] A. N. Lam, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, “Bug localization with combination of deep learning and information retrieval,” in *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, pp. 218–229, IEEE, 2017.
- [510] T.-D. B. Le, F. Thung, and D. Lo, “Will this localization tool be effective for this bug? mitigating the impact of unreliability of information retrieval based bug localization tools,” *Empirical Software Engineering*, vol. 22, no. 4, pp. 2237–2279, 2017.
- [511] J.-E. J. Tevis and J. A. Hamilton, “Methods for the prevention, detection and removal of software security vulnerabilities,” in *Proceedings of the 42nd annual Southeast regional conference*, pp. 197–202, ACM, 2004.
- [512] D. M. Chess, “Security issues in mobile code systems,” in *Mobile agents and security*, pp. 1–14, Springer, 1998.
- [513] C. Gao, J. Zeng, M. R. Lyu, and I. King, “Online app review analysis for identifying emerging issues,” in *Proceedings of the 40th International Conference on Software Engineering, ICSE ’18, (New York, NY, USA)*, pp. 48–58, ACM, 2018.
- [514] L. AlSumait, D. Barbará, and C. Domeniconi, “On-line lda: Adaptive topic models for mining text streams with applications to topic detection and tracking,” in *2008 eighth IEEE international conference on data mining*, pp. 3–12, IEEE, 2008.
- [515] P. K. Goyal and G. Joshi, “Qmood metric sets to assess quality of java program,” in *2014 International Conference on Issues and Challenges in Intelligent Computing Techniques (ICICT)*, pp. 520–533, IEEE, 2014.
- [516] J. Bayuk and A. Mostashari, “Measuring systems security,” *Systems Engineering*, vol. 16, no. 1, pp. 1–14, 2013.
- [517] C. Abid, S. T. Kessentini, Marouane, and K. Rick, “Study appendix,” 2021. URL: <https://sites.google.com/view/tosem2021>.
- [518] M. Lu and P. Liang, “Automatic classification of non-functional requirements from augmented app user reviews,” in *Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering*, pp. 344–353, 2017.

- [519] S. Panichella, A. Di Sorbo, E. Guzman, C. A. Visaggio, G. Canfora, and H. C. Gall, “Ardoc: App reviews development oriented classifier,” in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 1023–1027, 2016.
- [520] Y. Man, C. Gao, M. R. Lyu, and J. Jiang, “Experience report: Understanding cross-platform app issues from user reviews,” in *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*, pp. 138–149, IEEE, 2016.
- [521] P. M. Vu, T. T. Nguyen, H. V. Pham, and T. T. Nguyen, “Mining user opinions in mobile app reviews: A keyword-based approach (t),” in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 749–759, IEEE, 2015.
- [522] S. Bird, E. Loper, and E. Klein, *Natural Language Processing with Python*. O’Reilly Media Inc., 2009. URL: <http://www.nltk.org>, last accessed on 2020-02-25.
- [523] D. Pagano and W. Maalej, “User feedback in the appstore: An empirical study,” in *2013 21st IEEE international requirements engineering conference (RE)*, pp. 125–134, IEEE, 2013.
- [524] P. Lam, E. Bodden, O. Lhoták, and L. Hendren, “The soot framework for java program analysis: a retrospective,” in *Cetus Users and Compiler Infrastructure Workshop (CETUS 2011)*, vol. 15, p. 35, 2011.
- [525] Y. Tang, F. Zhao, Y. Yang, H. Lu, Y. Zhou, and B. Xu, “Predicting vulnerable components via text mining or software metrics? an effort-aware perspective,” in *2015 IEEE International Conference on Software Quality, Reliability and Security*, pp. 27–36, IEEE, 2015.
- [526] R. Scandariato, J. Walden, A. Hovsepian, and W. Joosen, “Predicting vulnerable software components via text mining,” *IEEE Transactions on Software Engineering*, vol. 40, no. 10, pp. 993–1006, 2014.
- [527] J. Walden, J. Stuckman, and R. Scandariato, “Predicting vulnerable components: Software metrics vs text mining,” in *2014 IEEE 25th international symposium on software reliability engineering*, pp. 23–33, IEEE, 2014.
- [528] R. Artusi, P. Verderio, and E. Marubini, “Bravais-pearson and spearman correlation coefficients: meaning, test of hypothesis and confidence interval,” *The International journal of biological markers*, vol. 17, no. 2, pp. 148–151, 2002.
- [529] H. Akoglu, “User’s guide to correlation coefficients,” *Turkish journal of emergency medicine*, vol. 18, no. 3, pp. 91–93, 2018.
- [530] A. Vargha and H. D. Delaney, “A critique and improvement of the cl common language effect size statistics of mcgraw and wong,” *Journal of Educational and Behavioral Statistics*, vol. 25, no. 2, pp. 101–132, 2000.

- [531] P. Guo, R. Peterson, P. Paukstelis, and J. Wang, “Cloud-based life sciences manufacturing system: Integrated experiment management and data analysis via amazon web services,” in *INFORMS International Conference on Service Science*, pp. 149–159, Springer, 2019.
- [532] A. P. Kalogeras, J. Gialelis, C. Alexakos, M. Georgoudakis, and S. Koubias, “Vertical integration of enterprise industrial systems utilizing web services,” in *IEEE International Workshop on Factory Communication Systems, 2004. Proceedings.*, pp. 187–192, IEEE, 2004.
- [533] J. Jung, B. Song, K. Watson, and T. Usländer, “Design of smart factory web services based on the industrial internet of things,” in *Proceedings of the 50th Hawaii International Conference on System Sciences*, 2017.
- [534] A. Greenberg, J. Hamilton, D. A. Maltz, and P. Patel, “The cost of a cloud: research problems in data center networks,” *ACM SIGCOMM computer communication review*, vol. 39, no. 1, pp. 68–73, 2008.
- [535] A. Nasridinov, J.-Y. Byun, and Y.-H. Park, “A qos-aware performance prediction for self-healing web service composition,” in *2012 Second International Conference on Cloud and Green Computing*, pp. 799–803, IEEE, 2012.
- [536] R. Mohanty, V. Ravi, and M. R. Patra, “Web-services classification using intelligent techniques,” *Expert Systems with Applications*, vol. 37, no. 7, pp. 5484–5490, 2010.
- [537] A. Ouni, R. G. Kula, M. Kessentini, T. Ishio, D. M. German, and K. Inoue, “Search-based software library recommendation using multi-objective optimization,” *Information and Software Technology*, vol. 83, pp. 55–75, 2017.
- [538] E. Al-Masri and Q. H. Mahmoud, “The QWS dataset.” URL: <https://qwsdata.github.io>.
- [539] R. Agrawal, R. Srikant, *et al.*, “Fast algorithms for mining association rules,” in *Proc. 20th int. conf. very large data bases, VLDB*, vol. 1215, pp. 487–499, 1994.
- [540] S. Brin, R. Motwani, and C. Silverstein, “Beyond market baskets: Generalizing association rules to correlations,” *Acm Sigmod Record*, vol. 26, no. 2, pp. 265–276, 1997.
- [541] H. Toivonen *et al.*, “Sampling large databases for association rules,” in *VLDB*, vol. 96, pp. 134–145, 1996.
- [542] M. J. Zaki, S. Parthasarathy, M. Ogihara, and W. Li, “Parallel algorithms for discovery of association rules,” *Data mining and knowledge discovery*, vol. 1, no. 4, pp. 343–373, 1997.
- [543] W. L. J. H. J. Pei *et al.*, “Cmar: Accurate and efficient classification based on multiple class-association rules,” *ICDM-2004*, 2001.

- [544] M. Ilayaraja and T. Meyyappan, “Mining medical data to identify frequent diseases using apriori algorithm,” in *2013 International Conference on Pattern Recognition, Informatics and Mobile Engineering*, pp. 194–199, IEEE, 2013.
- [545] S. Sathyadevan, S. Gangadharan, *et al.*, “Crime analysis and prediction using data mining,” in *2014 First International Conference on Networks & Soft Computing (IC-NSC2014)*, pp. 406–412, IEEE, 2014.
- [546] A. Methaila, P. Kansal, H. Arya, P. Kumar, *et al.*, “Early heart disease prediction using data mining techniques,” *Computer Science & Information Technology Journal*, pp. 53–59, 2014.
- [547] A. Savasere, E. R. Omiecinski, and S. B. Navathe, “An efficient algorithm for mining association rules in large databases,” tech. rep., Georgia Institute of Technology, 1995.
- [548] S. Tomović and P. Stanišić, “Cross validation method in frequent itemset mining,” in *CECIIS-2011*, 2011.
- [549] D. F. Ransohoff, “Rules of evidence for cancer molecular-marker discovery and validation,” *Nature Reviews Cancer*, vol. 4, no. 4, p. 309, 2004.
- [550] P. D. Adams, N. S. Pannu, R. J. Read, and A. T. Brünger, “Cross-validated maximum likelihood enhances crystallographic simulated annealing refinement,” *Proceedings of the National Academy of Sciences*, vol. 94, no. 10, pp. 5018–5023, 1997.
- [551] B. Ray, D. Posnett, V. Filkov, and P. Devanbu, “A large scale study of programming languages and code quality in github,” in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 155–165, 2014.
- [552] L. Prechelt, “An empirical comparison of seven programming languages,” *Computer*, vol. 33, no. 10, pp. 23–29, 2000.
- [553] T. Sharma and D. Spinellis, “A survey on software smells,” *Journal of Systems and Software*, vol. 138, pp. 158–173, 2018.
- [554] N. Moha, Y.-G. Guéhéneuc, L. Duchien, and A.-F. Le Meur, “Decor: A method for the specification and detection of code and design smells,” *IEEE Transactions on Software Engineering*, vol. 36, no. 1, pp. 20–36, 2009.
- [555] F. Palomba, A. Panichella, A. De Lucia, R. Oliveto, and A. Zaidman, “A textual-based technique for smell detection,” in *2016 IEEE 24th international conference on program comprehension (ICPC)*, pp. 1–10, IEEE, 2016.
- [556] F. Palomba, G. Bavota, M. Di 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, 2014.



- [557] 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.
- [558] F. Pecorelli, F. Palomba, D. Di Nucci, and A. De Lucia, “Comparing heuristic and machine learning approaches for metric-based code smell detection,” in *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, pp. 93–104, IEEE, 2019.
- [559] F. Pecorelli, D. Di Nucci, C. De Roover, and A. De Lucia, “A large empirical assessment of the role of data balancing in machine-learning-based code smell detection,” *Journal of Systems and Software*, vol. 169, p. 110693, 2020.
- [560] 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, IEEE, 2015.
- [561] A. J. Mooij, J. Ketema, S. Klusener, and M. Schuts, “Reducing code complexity through code refactoring and model-based rejuvenation,” in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 617–621, IEEE, 2020.
- [562] F. Pecorelli, F. Palomba, F. Khomh, and A. De Lucia, “Developer-driven code smell prioritization,” in *Proceedings of the 17th International Conference on Mining Software Repositories*, pp. 220–231, 2020.
- [563] N. Sae-Lim, S. Hayashi, and M. Saeki, “Context-based code smells prioritization for refactoring,” in *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, pp. 1–10, IEEE, 2016.
- [564] M. I. Azeem, F. Palomba, L. Shi, and Q. Wang, “Machine learning techniques for code smell detection: A systematic literature review and meta-analysis,” *Information and Software Technology*, vol. 108, pp. 115–138, 2019.
- [565] E. V. de Paulo Sobrinho, A. De Lucia, and M. de Almeida Maia, “A systematic literature review on bad smells—5 w’s: which, when, what, who, where,” *IEEE Transactions on Software Engineering*, 2018.
- [566] F. A. Fontana, V. Ferme, M. Zanoni, and A. Yamashita, “Automatic metric thresholds derivation for code smell detection,” in *2015 IEEE/ACM 6th International Workshop on Emerging Trends in Software Metrics*, pp. 44–53, IEEE, 2015.
- [567] F. A. Fontana and M. Zanoni, “Code smell severity classification using machine learning techniques,” *Knowledge-Based Systems*, vol. 128, pp. 43–58, 2017.
- [568] M. K. T. S. F. P. Chaima Abid, Thiago do Nascimento Ferreira, “Study appendix,” 2021. URL: <https://sites.google.com/view/tse2021benchmarking/home>.

- [569] S. Wold, K. Esbensen, and P. Geladi, "Principal component analysis," *Chemometrics and intelligent laboratory systems*, vol. 2, no. 1-3, pp. 37–52, 1987.
- [570] F. J. Massey Jr, "The kolmogorov-smirnov test for goodness of fit," *Journal of the American statistical Association*, vol. 46, no. 253, pp. 68–78, 1951.
- [571] P. J. Rousseeuw, "Silhouettes: a graphical aid to the interpretation and validation of cluster analysis," *Journal of computational and applied mathematics*, vol. 20, pp. 53–65, 1987.
- [572] L. Zhu, B. Ma, and X. Zhao, "Clustering validity analysis based on silhouette coefficient [j]," *Journal of Computer Applications*, vol. 30, no. 2, pp. 139–141, 2010.
- [573] N. Rahmah and I. S. Sitanggang, "Determination of optimal epsilon (eps) value on db-scan algorithm to clustering data on peatland hotspots in sumatra," in *IOP conference series: earth and environmental science*, vol. 31, p. 012012, IOP Publishing, 2016.
- [574] Nist and E. Aroms, *NIST Special Publication 800-53 Revision 3 Recommended Security Controls for Federal Information Systems and Organizations*. Paramount, CA: CreateSpace, 2012.
- [575] S. Planning, "The economic impacts of inadequate infrastructure for software testing," *National Institute of Standards and Technology*, 2002.
- [576] W. Suryn, A. Abran, and A. April, "Iso/iec square. the second generation of standards for software product quality," *IASTED International Conference on Software Engineering and Applications SEA*, 2003.
- [577] T. A. Linden, "Operating system structures to support security and reliable software," *ACM Computing Surveys (CSUR)*, vol. 8, no. 4, pp. 409–445, 1976.
- [578] A. Adewumi, S. Misra, and N. Omoregbe, "Evaluating open source software quality models against iso 25010," in *2015 IEEE International Conference on Computer and Information Technology; Ubiquitous Computing and Communications; Dependable, Autonomic and Secure Computing; Pervasive Intelligence and Computing*, pp. 872–877, IEEE, 2015.
- [579] M. Fowler, *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018.
- [580] K. Maruyama and K. Tokoda, "Security-aware refactoring alerting its impact on code vulnerabilities," in *2008 15th Asia-Pacific Software Engineering Conference*, pp. 445–452, Dec 2008.
- [581] K. Maruyama and T. Omori, "A security-aware refactoring tool for java programs," *Proceedings of the 4th Workshop on Refactoring Tools*, pp. 22–28, 2011.
- [582] 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.

- [583] J. Benesty, J. Chen, Y. Huang, and I. Cohen, “Pearson correlation coefficient,” in *Noise reduction in speech processing*, pp. 1–4, Springer, 2009.
- [584] S. P. Coy, B. L. Golden, G. C. Runger, and E. A. Wasil, “Using experimental design to find effective parameter settings for heuristics,” *Journal of Heuristics*, vol. 7, no. 1, pp. 77–97, 2001.
- [585] M. Harman, S. A. Mansouri, and Y. Zhang, “Search-based software engineering: Trends, techniques and applications,” *ACM Comput. Surv.*, vol. 45, pp. 11:1–11:61, Dec. 2012.
- [586] R. Mo, Y. Cai, R. Kazman, L. Xiao, and Q. Feng, “Decoupling level: a new metric for architectural maintenance complexity,” in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pp. 499–510, IEEE, 2016.
- [587] M. Reif, M. Eichberg, B. Hermann, and M. Mezini, “Hermes: assessment and creation of effective test corpora,” in *Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, pp. 43–48, 2017.
- [588] M. Reif, M. Eichberg, B. Hermann, J. Lerch, and M. Mezini, “Call graph construction for java libraries,” in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 474–486, 2016.
- [589] M. A. Cusumano, “Who is liable for bugs and security flaws in software?,” *Communications of the ACM*, vol. 47, no. 3, pp. 25–27, 2004.
- [590] I. V. Krsul, *Software vulnerability analysis*. Purdue University West Lafayette, IN, 1998.
- [591] N. Nikiforakis, L. Invernizzi, A. Kapravelos, S. Van Acker, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna, “You are what you include: Large-scale evaluation of remote javascript inclusions,” in *Proceedings of the 2012 ACM Conference on Computer and Communications Security, CCS ’12*, pp. 736–747, ACM, 2012.
- [592] J. Han and Y. Zheng, “Security characterisation and integrity assurance for software components and component-based systems,” in *Proceedings of 1998 Australasian Workshop on Software Architectures, Melbourne*, pp. 83–89, 1998.
- [593] V. B. Livshits and M. S. Lam, “Finding security vulnerabilities in java applications with static analysis,” in *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14, SSYM’05*, pp. 18–18, USENIX Association, 2005.
- [594] V. Haldar, D. Chandra, and M. Franz, “Dynamic taint propagation for java,” in *Proceedings of the 21st Annual Computer Security Applications Conference, ACSAC ’05*, pp. 303–311, IEEE Computer Society, 2005.
- [595] K. Huang, J. Zhang, W. Tan, and Z. Feng, “Shifting to mobile: Network-based empirical study of mobile vulnerability market,” *IEEE Transactions on Services Computing*, 2016.

- [596] M. W. Mkaouer, M. Kessentini, S. Bechikh, and M. Ó. Cinnéide, “A robust multi-objective approach for software refactoring under uncertainty,” in *International Symposium on Search Based Software Engineering*, pp. 168–183, Springer, Cham, 2014.
- [597] N. Tsantalis and A. Chatzigeorgiou, “Ranking refactoring suggestions based on historical volatility,” in *2011 15th European Conference on Software Maintenance and Reengineering*, pp. 25–34, IEEE, 2011.
- [598] N. Zazworka, C. Seaman, and F. Shull, “Prioritizing design debt investment opportunities,” in *Proceedings of the 2nd Workshop on Managing Technical Debt*, pp. 39–42, ACM, 2011.
- [599] M. Fokaefs, N. Tsantalis, E. Stroulia, and A. Chatzigeorgiou, “Jdeodorant: identification and application of extract class refactorings,” in *2011 33rd International Conference on Software Engineering (ICSE)*, pp. 1037–1039, IEEE, 2011.
- [600] “Cve vulnerability data.” <https://www.cvedetails.com/>.
- [601] 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.
- [602] A. Arcuri and L. Briand, “A practical guide for using statistical tests to assess randomized algorithms in software engineering,” in *2011 33rd International Conference on Software Engineering (ICSE)*, pp. 1–10, IEEE, 2011.
- [603] 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.
- [604] G. Bavota, A. D. Lucia, M. D. 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.
- [605] N. Tsantalis and A. Chatzigeorgiou, “Identification of move method refactoring opportunities,” *IEEE Transactions on Software Engineering*, vol. 35, no. 3, pp. 347–367, 2009.
- [606] Anonymous Authors(s), “Study appendix,” 2021. <https://sites.google.com/view/asedependency>.
- [607] 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.
- [608] M. O’Keeffe and M. O. Cinnéide, “A stochastic approach to automated design improvement,” in *Proceedings of the 2nd International Conference on Principles and Practice of Programming in Java (PPPJ ’03)*, (Kilkenny City, Ireland), pp. 59–62, ACM, 2003.

- [609] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.
- [610] F. Ferrucci, M. Harman, J. Ren, and F. Sarro, “Not going to take this anymore: multi-objective overtime planning for software engineering projects,” in *2013 35th International Conference on Software Engineering (ICSE)*, pp. 462–471, IEEE, 2013.
- [611] H. Meunier, E.-G. Talbi, and P. Reininger, “A multiobjective genetic algorithm for radio network optimization,” in *Proceedings of the 2000 Congress on Evolutionary Computation. CEC00 (Cat. No. 00TH8512)*, vol. 1, pp. 317–324, IEEE, 2000.
- [612] D. A. Van Veldhuizen and G. B. Lamont, “Multiobjective evolutionary algorithm research: A history and analysis,” tech. rep., Citeseer, 1998.
- [613] E. Zitzler and L. Thiele, “Multiobjective evolutionary algorithms: a comparative case study and the strength pareto approach,” *IEEE transactions on Evolutionary Computation*, vol. 3, no. 4, pp. 257–271, 1999.
- [614] X. Zhang, Y. Tian, and Y. Jin, “A knee point-driven evolutionary algorithm for many-objective optimization,” *IEEE Transactions on Evolutionary Computation*, vol. 19, no. 6, pp. 761–776, 2014.
- [615] J. Koehler and A. Owen, “‘computer experiments’,” *Handbook of Statistics*, vol. 13, pp. 261–308, 1996.
- [616] R. Terra, M. T. Valente, K. Czarnecki, and R. S. Bigonha, “Recommending refactorings to reverse software architecture erosion,” in *2012 16th European Conference on Software Maintenance and Reengineering*, pp. 335–340, IEEE, 2012.
- [617] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk, “Detecting bad smells in source code using change history information,” in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 268–278, IEEE, 2013.
- [618] M. Mohan and D. Greer, “A survey of search-based refactoring for software maintenance,” *Journal of Software Engineering Research and Development*, vol. 6, no. 1, p. 3, 2018.
- [619] M. Hall, N. Walkinshaw, and P. McMinn, “Supervised software modularisation,” in *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pp. 472–481, IEEE, 2012.
- [620] W. Afzal, R. Torkar, and R. Feldt, “A systematic review of search-based testing for non-functional system properties,” *Information and Software Technology*, vol. 51, no. 6, pp. 957–976, 2009.
- [621] V. Toğan and A. T. Daloğlu, “An improved genetic algorithm with initial population strategy and self-adaptive member grouping,” *Computers & Structures*, vol. 86, no. 11–12, pp. 1204–1218, 2008.

- [622] Y. Deng, Y. Liu, and D. Zhou, “An improved genetic algorithm with initial population strategy for symmetric tsp,” *Mathematical Problems in Engineering*, vol. 2015, 2015.
- [623] S. M. Elsayed, R. A. Sarker, and D. L. Essam, “Ga with a new multi-parent crossover for solving ieeec2011 competition problems,” in *2011 IEEE congress of evolutionary computation (CEC)*, pp. 1034–1040, IEEE, 2011.
- [624] N. Tsantalis, M. Mansouri, L. Eshkevari, D. Mazinianian, and D. Dig, “Accurate and efficient refactoring detection in commit history,” in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pp. 483–494, IEEE, 2018.
- [625] A. authors, “Study appendix,” 2020. URL: <https://sites.google.com/view/tse2020xsbr>.
- [626] P. D. McNicholas, T. B. Murphy, and M. O’Regan, “Standardising the lift of an association rule,” *Computational Statistics & Data Analysis*, vol. 52, no. 10, pp. 4712–4721, 2008.
- [627] E.-G. Talbi, *Metaheuristics: from design to implementation*, vol. 74. John Wiley & Sons, 2009.
- [628] F. Wilcoxon, S. Katti, and R. A. Wilcox, “Critical values and probability levels for the wilcoxon rank sum test and the wilcoxon signed rank test,” *Selected tables in mathematical statistics*, vol. 1, pp. 171–259, 1970.
- [629] J. T. Richardson, “Eta squared and partial eta squared as measures of effect size in educational research,” *Educational Research Review*, vol. 6, no. 2, pp. 135–147, 2011.
- [630] M. Harman, S. A. Mansouri, and Y. Zhang, “Search-based software engineering: Trends, techniques and applications,” *ACM Computing Surveys (CSUR)*, vol. 45, no. 1, pp. 1–61, 2012.