A User-aware Intelligent Refactoring for Discrete and Continuous Software Integration

by

Vahid Alizadeh

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
2020

Doctoral Committee:

Associate Professor Marouane Kessentini, Chair
Professor William Grosky
Professor Bruce Maxim
Associate Professor Luis Ortiz
Professor Armen Zakarian

M
DEARBORN

ISELab
Intelligent Software Engineering

© Vahid Alizadeh 2020

All Rights Reserved

TABLE OF CONTENTS

LIST OF FIGURES . . . vi

LIST OF TABLES . . . ix

LIST OF ABBREVIATIONS . . . xi

ABSTRACT . . . xii

CHAPTER

I. Introduction . . . 1

1.1 Research Context: Software Refactoring . . . 1
1.2 Problem Statement . . . 3
1.3 Proposed Contributions . . . 4
1.4 Organization of the Dissertation . . . 9

II. State of the Art . . . 10

2.1 Introduction . . . 10
2.2 Background . . . 10
2.2.1 Software Refactoring . . . 10
2.2.1.1 Refactoring Operations . . . 12
2.2.2 Interactive and Dynamic Evolutionary Multi-Objective Optimization . . . 13
2.2.3 Code Quality Metrics . . . 16
2.3 Related Work . . . 19
2.3.1 Manual Refactoring . . . 19
2.3.2 Automated Refactoring . . . 21
2.3.3 Interactive Refactoring . . . 23
2.3.4 Search Based Software Refactoring . . . 25
2.3.5 Refactoring Recommendation . . . 26
2.3.6 Empirical Studies on Refactoring . . . 27
2.3.7 Software Bots . . . 28

2.4 Summary of Systematic Literature Review on Refactoring . . 29

III. Interactive Multi-Objective Refactoring 35

3.1 Introduction and Problem Statement 35
3.2 Approach: Search-based Interactive Refactoring Recommendation 41
3.2.1 Approach Overview 41
3.2.2 Adaptation 43
3.2.2.1 Multi-objective formulation 43
3.2.2.2 Solution representation 45
3.2.2.3 Solution variation 46
3.2.2.4 Solution evaluation 48
3.2.3 Interactive Recommendation of Refactorings 49
3.2.4 Running Example: Illustration on the JVacation System 54
3.2.4.1 Context 54
3.2.4.2 Illustration of the Innovization Component 55
3.2.4.3 Illustration of the Interactive and Dynamic Components 57
3.3 Evaluation 60
3.3.1 Research Questions 61
3.3.2 Validation Methodology 62
3.3.3 Studied Software Projects 68
3.3.4 Study Participants 69
3.3.5 Techniques Studied 70
3.3.5.1 Overview of the Used Techniques 70
3.3.5.2 Parameters Setting 72
3.3.6 Case Studies Summary 73
3.3.7 Results and Discussion 75
3.3.7.1 Statistical Analysis 75
3.4 Threats to Validity 93
3.5 Conclusion 96

IV. Reducing Interactive Refactoring Effort via Clustering-based Search 98

4.1 Introduction 98
4.2 Problem Statement 101
4.3 Approach: Clustering-based Interactive Multi-objective Software Refactoring 103
4.3.1 Overview 103
4.3.2 Phase 1: Multi-Objective Refactoring 104
4.3.2.1 Refactoring Solution Representation . . 105
4.3.2.2 Fitness Functions 108

4.3.2.3 Variation Operators 108
4.3.3 Phase 2: Clustering the Pareto Front of Refactoring Solutions 109
4.3.3.1 Calinski Harabasz (CH) Index 109
4.3.4 Phase 3: Developers Interaction and Preferences Extraction 111
4.3.5 Applying Preference Parameters 115
4.4 Evaluation 116
4.4.1 Research Questions 117
4.4.2 Experimental Setup 117
4.4.3 Statistical Tests and Parameters Setting 120
4.4.4 Results 121
4.5 Threats to Validity 126
4.6 Conclusion 127

V. From Multi-objective to Mono-objective Refactoring via Developers Preference Extraction 128

5.1 Introduction 128
5.2 Motivations 131
5.3 Approach Overview 133
5.3.1 Phase 1: Multi-Objective Refactoring 134
5.3.1.1 Solution Representation 137
5.3.1.2 Fitness Functions 137
5.3.2 Phase 2: Clustering Refactoring Solutions and Extracting Developer Preferences 138
5.3.2.1 Clustering the Pareto-front 138
5.3.2.2 Interaction and Preference Extraction . 140
5.3.3 Phase 3: Preference-based Mono-objective Refactoring 141
5.4 Evaluation 145
5.4.1 Research Questions 145
5.4.2 Experimental Setup 146
5.4.3 Results 151
5.5 Threats to Validity 157
5.6 Conclusion 158

VI. Simultaneous Decision and Objective Space Clustering for Interactive Refactoring 160

6.1 Introduction 160
6.2 Interactive Refactoring Challenges 164
6.3 Approach Overview 168
6.3.1 Phase 1: Multi-Objective Refactoring 168
6.3.1.1 Solution Representation 168
6.3.1.2 Fitness Functions 170

6.3.2 Phase 2: Objective Space Clustering 170
6.3.3 Phase 3: Decision Space Clustering 172
6.3.4 Phase 4: Developer Feedback and Preference Extraction 175
6.4 Evaluation 176
6.4.1 Research Questions 176
6.4.2 Experimental Setup 177
6.4.3 Parameter Setting 180
6.4.4 Results 181
6.5 Threats to Validity 185
6.6 Conclusion 187

VII. Intelligent Refactoring Bot for Continuous Integration 188

7.1 Introduction and Problem Statement 188
7.2 Approach 191
7.2.1 RefBot Parameters Setting 192
7.2.2 Processing a Pull Request 192
7.2.2.1 Calculating Quality Changes 194
7.2.2.2 Optimization Using Refactoring 194
7.2.3 Developer's Interaction 196
7.2.4 Configuration and Customization 197
7.2.5 Running Example 198
7.3 Validation 201
7.3.1 Data Collection 203
7.3.2 Experimental Setting and Data Analysis 207
7.3.3 Results 207
7.4 Threats to Validity 211
7.5 Conclusion 211

VIII. Conclusion 213

8.1 Summary 213
8.2 Future Work 217

BIBLIOGRAPHY 220

LIST OF FIGURES

Figure

1.1	Overview of the contributions of this thesis.	5
2.1	Number of refactoring publications over the last two decades.	31
2.2	Leading refactoring researchers over the last decade based on both publications and citations.	31
2.3	Distribution of refactoring researchers around the world.	32
2.4	Taxonomy of refactoring researches and the number of publications during the past two decade.	33
3.1	Approach overview.	41
3.2	Refactorings recommended by our technique.	51
3.3	Recommended target classes by our technique for a move method refactoring to modify.	52
3.4	Boxplots of G, NF, MC, PR, and RC on all the ten systems based on 30 independent runs. (Continue on the next page.) Label of the methods: **M1** (Our approach)=Interactive+Innovization NSGA-II, **M2**=Innovization NSGA-II, **M3**=Kessentini et al.[1], **M4**=Ouni et al.[2], **M5**=Harman et al.[3], **M6**=O'Keeffe et al.[4], **M7**=Jdeodorant [5]	83
3.5	MC@k results on the different systems with k= 1, 5, 10 and 15.	87
3.6	PR@k results on the different systems with k= 1, 5, 10 and 15.	87
3.7	The median NMR, NRR and NAR results in the different systems.	88
3.8	The average productivity difference (TP) results on the different tasks performed by the three groups using our interactive approach, Ouni et al. [2], Harman et al.[3]	89
3.9	GanttOptions before and after refactoring.	92
4.1	Overview of our proposed IC-NSGA-II approach.	103
4.2	Allowing user to select the desired refactoring operators and fitness functions in our tool	107
4.3	Psuedo-code for Phase 2 of our proposed approach.	110
4.4	Interactive solution charts in our tool.	112
4.5	Interactive solution tables and cluster selection in our tool.	113

4.6 The median manual evaluation scores, MC, on the six systems with 95% confidence level ($\alpha = 5\%$) based on a one-way ANOVA statistical test . . . 122
4.7 Illustration of the refactoring solutions convergence to a region of interest after two rounds of interactions extracted from the experiments on the Gantt Project. . . . 125
5.1 The output of a multi-objective refactoring tool[6] finding trade-offs between QMOOD quality attributes on GanttProject v1.10.2 . . . 133
5.2 Overview of our proposed approach. . . . 134
5.3 The output of phase 2 (Clustering) on GanttProject v1.10.2. . . . 140
5.4 The output of phase 3 (Mono-objective) on GanttProject v1.10.2 system. . . . 145
5.5 Average manual evaluations, MC, on the 7 systems. . . . 152
5.6 The median number of recommended refactorings, NR, of the selected solution on the 7 systems. . . . 153
5.7 The median number of required interactions (accept / reject/ modify / selection), NI, on the 7 systems. . . . 154
5.8 The average execution time, T, in minutes on the 7 systems. . . . 155
5.9 A qualitative example of three executions extracted from our experiments on GanttProject to illustrate the process of converting a multi-objective search into a mono-objective one. . . . 156
6.1 The output of a multi-objective refactoring tool [6] finding trade-offs between QMOOD quality attributes on GanttProject v1.10.2 with clustering only in the objective space. . . . 167
6.2 Example of a refactoring solution proposed by our tool for GanttProject v1.10.2. . . . 169
6.3 Clustering based on code locations (decision space) of the refactoring solutions of one region of interest in the objective space of GanttProject v1.10.2. . . . 172
6.4 Illustration of the clustered solutions in the objective space and the decision space. . . . 174
6.5 Median manual evaluations, MC, on the 7 systems. . . . 183
6.6 The median number of recommended refactorings, NR, of the selected solution on the 7 systems. . . . 184
6.7 The median number of required interactions (accept / reject / modify / selection), NI, on the 7 systems. . . . 185
6.8 The median execution time, T, in minutes on the 7 systems. . . . 186
7.1 The overview of RefBot Pipeline. . . . 192
7.2 Installing RefBot on a repository. . . . 193
7.3 The quality table in solution report page. . . . 198
7.4 The quality bar charts in file report page for all six quality attributes. 199
7.5 The list of refactoring operations recommended for a single file. . . . 200
7.6 The code abstraction of source and target classes after applying a specific refactoring. . . . 200

7.7 The refactoring instructions related to a single file are added to the source code as a marker style. 201
7.8 Median percentage of fixed code smells (NF) on the different pull-requests of the seven systems. 208
7.9 Median quality gain (G) on the different pull-requests of the seven systems. 208

LIST OF TABLES

Table

2.1 List of refactoring operations included in this thesis. 12
2.2 QMOOD design metrics. 18
2.3 QMOOD quality attributes. 19
3.1 Example of a solution representation. 47
3.2 Quality attributes value on the JVacation system. 56
3.3 Three simplified refactoring solutions recommended for JVacation v1.0. 58
3.4 Four different interaction examples with the developer applied on the refactoring solutions recommended for JVacation v1.0. 59
3.5 Summary of the research questions, their goals, defined metrics to answer and analyze them, and the associated tasks to collect data and calculate the metrics. 64
3.6 Statistics of the studied software projects. 69
3.7 Survey organization. 72
3.8 F-value results from one-way ANOVA statistical tests for corresponding software project and metric between different methods. 77
3.9 Effect size values (Eta squared (η^2)) for corresponding software project and metric. 78
3.10 Tukey post hoc analysis results between our method(M1) and others reported by Mean difference and 95% confidence intervals. Label of the methods: **M1** (Our approach)=Interactive+Innovization NSGA-II, **M2**=Innovization NSGA-II, **M3**=Kessentini et al.[1], **M4**=Ouni et al.[2], **M5**=Harman et al.[3], **M6**=O'Keeffe et al.[4], **M7**=Jdeodorant [5]. 79
4.1 Statistics of the studied systems. 118
4.2 Selected programmers. 119
4.3 Median time, in minutes, and number of refactorings proposed by both interactive approaches on the different six systems. 124
5.1 Statistics of the studied systems. 147
5.2 Selected programmers. 149
6.1 Statistics of the studied systems. 178
6.2 Selected participants. 180
7.1 Statistics of the studied systems. 203

7.2 Participants involved in RQ2. 206
7.3 RQ2: Would you apply the proposed refactorings of the generated refactoring pull-request? 209

LIST OF ABBREVIATIONS

NSGAII Non-dominated Sorting Genetic Algorithm

GMM Gaussian Mixture Model

IDE Integrated development environment

SLR systematic literature review

IGA Interactive Genetic Algorithm

GA Classic Genetic Algorithm

EMO Evolutionary Multi-objective Optimization

CI/CD Continous Integration/Continous Development

QMOOD Quality Metrics for Object Oriented Designs

ISE LAB Intelligent Software Engineering Laboratory

ABSTRACT

Successful software products evolve through a process of continual change. However, this process may weaken the design of the software and make it unnecessarily complex, leading to significantly reduced productivity and increased fault-proneness.

Refactoring improves the software design while preserving overall functionality and behavior, and is an important technique in managing the growing complexity of software systems. Most of the existing work on software refactoring uses either an entirely manual or a fully automated approach. Manual refactoring is time-consuming, error-prone and unsuitable for large-scale, radical refactoring. Furthermore, fully automated refactoring yields a static list of refactorings which, when applied, leads to a new and often hard to comprehend design. In addition, it is challenging to merge these refactorings with other changes performed in parallel by developers.

In this thesis, we propose a refactoring recommendation approach that dynamically adapts and interactively suggests refactorings to developers and takes their feedback into consideration. Our approach uses Non-dominated Sorting Genetic Algorithm (NSGAII) to find a set of good refactoring solutions that improve software quality while minimizing the deviation from the initial design. These refactoring solutions are then analyzed to extract interesting common features between them such as the frequently occurring refactorings in the best non-dominated solutions.

We combined our interactive approach and unsupervised learning to reduce the developer's interaction effort when refactoring a system. The unsupervised learning

algorithm clusters the different trade-off solutions, called the Pareto front, to guide the developers in selecting their region of interests and reduce the number of refactoring options to explore.

To reduce the interaction effort, we propose an approach to convert multi-objective search into a mono-objective one after interacting with the developer to identify a good refactoring solution based on their preferences. Since developers may want to focus on specific code locations, the "Decision Space" is also important. Therefore, our interactive approach enables developers to pinpoint their preference simultaneously in the objective (quality metrics) and decision (code location) spaces.

Due to an urgent need for refactoring tools that can support continuous integration and some recent development processes such as DevOps that are based on rapid releases, we propose, for the first time, an intelligent software refactoring bot, called RefBot. Our bot continuously monitors the software repository and find the best sequence of refactorings to fix the quality issues in Continous Integration/Continous Development (CI/CD) environments as a set of pull-requests generated after mining previous code changes to understand the profile of developers.

We quantitatively and qualitatively evaluated the performance and effectiveness of our proposed approaches via a set of studies conducted with experienced developers who used our tools on both open source and industry projects.

CHAPTER I

Introduction

1.1 Research Context: Software Refactoring

A recent study [7] by the US Air Force Software Technology Support Centre (STSC) shows that the code restructuring of several software systems reduced developers' time by over 60% when introducing new features into a restructured architecture.

General Motors (GM) is recalling nearly 4.3 million vehicles in 2017 after discovering a software quality defect of poor modularity in an evolved program in a car controller. It caused performance issues that prevented air bags from deploying in time during a crash [8]. That flaw has already been linked to one death and three injuries.

Clearly, urgently, software engineers need better ways to reduce and manage the growing complexity of software systems and improve their productivity. Refactoring [9, 10, 11] is a technique that improves the design structure while preserving the overall functionality and behavior. Refactoring is a key practice in agile development processes, and is well supported by refactoring tools that are standard with all major IDEs. Refactoring is an extremely important solution to address the challenge of managing software complexity [12, 13, 14], and has experienced tremendous adoption in Object-oriented systems [15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25].

Evolution is a characteristic of software which means modifying the software to adapt new requirements and to incorporate new features. These modifications over time can degrade the software quality and increase the complexity of code leading to higher costs of development and maintenance. Therefore, there is a need of techniques to improve the quality and reduce the complexity of the software. The research area for this purpose is called *restructuring* or in case of an object-oriented environment, *Refactoring*.

Martin Fowler defined Refactoring as "a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior" [9]. This implies that refactoring is a method which reconstruct the code's structure without altering its behavior in order to improve the software quality in terms of maintainability, extensibility, and re-usability. Refactoring typically consists of small steps after each the functionality of the code will be unchanged. Refactoring can be done in various areas of the software: Code, Database, or User interface. However, we aim to focus on code refactoring.

It might be difficult for a developer to be justified to spend time on improvement of a piece of code in order to have the same exact functionality. However, it can be seen as an investment for future developments. Specifically, refactoring is an imperative task on softwares with longer lifespans with multiple developers need to read and understand the codes. Refactoring can improve both the quality of software and the productivity of its developers. Increasing the quality of software is due to decreasing the complexity of it at design and source code level caused by refactoring which is proved by many studies [26, 27]. The long-term effect of refactoring is improving the productivity of developers by increasing two crucial factors, understandability and maintainability of the codes, especially when a new developer join to an existing project. It is shown that refactoring can help to detect, fix, and reduce software bugs and leading to software projects which are less likely to expose bug in development

process [28]. Another study claims that there are some specific kinds of refactoring methods that are very probable to induce bug fixes [29].

Refactoring is a way of removing or reducing the presence of technical debt. Technical debt is a concept analogous to financial credit and it consists of code, design, test, and documentation debts. In software engineering world, it implies extra efforts and costs caused by an improper design or code structure. This can be seen more dramatically in large and long-lived software systems. Technical debt can be managed by increasing awareness, detecting and repaying, and preventing accumulation of it. Refactoring is the best strategy to cope with technical debt before it get out of control. Refactoring is beneficial to keep technical debt low and can be more efficient when it is automated [30].

Critical systems are those in which failure results in significant physical damages, economic disasters, or threats to human life. There are three types of critical systems: safety, mission, and business critical systems. Examples of these systems are automotive industry, spacecraft navigation systems, and banking. Regular changes are inevitable in software-critical systems, therefore refactoring plays a crucial role. It is shown that refactoring can improve the overall security of safety-critical system [31].

1.2 Problem Statement

Software design is a human activity that cannot be fully automated because designers understand the problem domain intuitively and they have targeted design goals in mind. Thus, several studies show that fully automated refactoring does not always lead to the desired architecture [32]. On the other hand, manual refactoring is error-prone, time consuming and not practical for radical changes. Based on interviews that we conducted as part of an NSF I-Corps project, programmers spend an average of 45% of their overall development time manually applying refactoring. Ba-

tory *et al.* [33] presented several case studies where architectural refactoring involved more than 750 refactoring steps and took more than 3 weeks to execute. Thus, it is important to develop intelligent methods to determine when and how to integrate programmer feedback to semi-automate architecture refactoring. We will seek to answer the fundamental scientific question: "What is the minimal guidance that leads automated search to useful and realistic architecture refactoring recommendations?" This will require both incorporating human- and machine- provided refactoring recommendations and human provided "hints".

Lack of reusable refactoring principles within the same project or across projects. Since refactoring cannot be fully automated, the interaction with humans during this process can be repetitive, expensive and tedious. In an interactive refactoring process, developers must evaluate recommended refactoring methods and adapt them to the targeted design. Recent work observes that software refactoring often requires systematic and repetitive changes to different contexts. Why do software engineers spend so much time repeatedly performing the same tedious low-level refactoring tasks? Cai *et al.* found that 24% to 40% of architecture level fixes involve similar changes to numerous locations [34]. A failure to systematically learn refactoring patterns from the interaction data can lead to costly and annoying labor.

1.3 Proposed Contributions

To address the problems mentioned in Subsection 1.2, we propose the following solutions which are organized into five main contributions as it is shown in Fig 1.1.

1. We propose a refactoring recommendation approach that dynamically adapts and interactively suggests refactorings to developers and takes their feedback into consideration. Our approach uses NSGAII to find a set of good refactoring solutions that improve software quality while minimizing the deviation from the

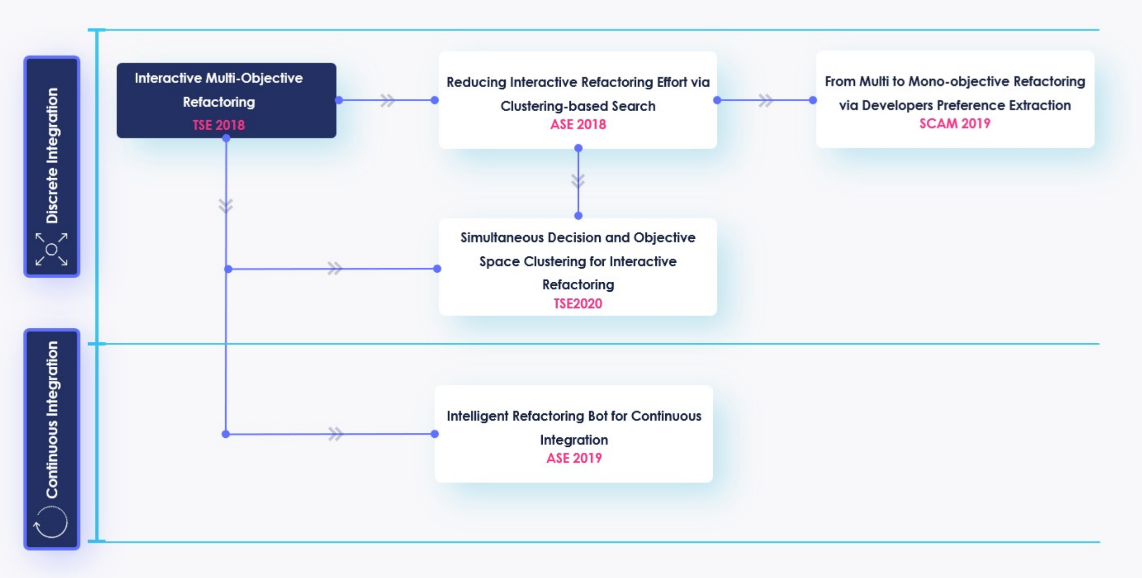


Figure 1.1: Overview of the contributions of this thesis.

initial design. These refactoring solutions are then analyzed to extract interesting common features between them such as the frequently occurring refactorings in the best non-dominated solutions. Based on this analysis, the refactorings are ranked and suggested to the developer in an interactive fashion as a sequence of transformations. The developer can approve, modify or reject each of the recommended refactorings, and this feedback is then used to update the proposed rankings of recommended refactorings. We evaluated our approach on a set of eight open source systems and two industrial projects provided by an industrial partner. Statistical analysis of our experiments shows that our dynamic interactive refactoring approach performed significantly better than four existing search-based refactoring techniques and one fully-automated refactoring tool not based on heuristic search. A paper is accepted and published at the IEEE Transactions in Software Engineering journal **TSE 2018** [1] [6], the tool is licensed to industrial partners, and a patent is approved [35].

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

2. We propose to extend the previous contribution by combining the use of multi-objective and unsupervised learning to reduce the developer's interaction effort when refactoring systems, a big challenge faced by programmers. We generate, first, using multi-objective search different possible refactoring strategies by finding a trade-off between several conflicting quality attributes. Then, an unsupervised learning algorithm clusters the different trade-off solutions, called the Pareto front, to guide the developers in selecting their region of interests and reduce the number of refactoring options to explore. The feedback from the developer, both at the cluster and solution levels, are used to automatically generate constraints to reduce the search space in the next iterations and focus on the region of developer preferences. We selected 14 active developers to manually evaluate the effectiveness our tool on 5 open source projects and one industrial system. An invention disclosure is approved for this work and the results are published at the 33rd IEEE/ACM International Conference on Automated Software Engineering **ASE 2018** [2] [36].

3. we proposed, for the first time, a way to convert multi-objective search into a mono-objective one after interacting with the developer to identify a good refactoring solution based on his preferences. The first step consists of using a multi-objective search to generate different possible refactoring strategies by finding a trade-off between several conflicting quality attributes. Then, an unsupervised learning algorithm clusters the different trade-off solutions, called the Pareto front, to guide the developers in selecting their region of interests and to reduce the number of refactoring options to explore. Finally, the extracted preferences from the developer are used to transform the multi-objective search into a mono-objective one by taking the preferred cluster of the Pareto front as

[2] Alizadeh, V., & Kessentini, M. (2018, September). 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).

the initial population for the mono-objective search and generating an evaluation function based on the weights that are automatically computed from the position of the cluster in the Pareto front. Thus, the developer will just interact with only one refactoring solution generated by the mono-objective search. We selected 32 participants to manually evaluate the effectiveness of our tool on 7 open source projects and one industrial project. The results show that the recommended refactorings are more accurate than the current state of the art. This approach is accepted and published at International Working Conference on Source Code Analysis and Manipulation **SCAM 2019** [3] [37].

4. To give developers more insight about the decision space, we proposed an interactive approach that enables developers to pinpoint their preference simultaneously in the objective (quality metrics) and decision (code location) spaces. Developers may be interested in looking at refactoring strategies that can improve a specific quality attribute, such as extendibility (objective space), but they are related to different code locations (decision space). A plethora of solutions is generated at first using multi-objective search that tries to find the possible trade-offs between quality objectives. Then, an unsupervised learning algorithm clusters the trade-off solutions based on their quality metrics, and another clustering algorithm is applied to each cluster of the objective space to identify solutions related to different code locations. The objective and decision spaces can now be explored more efficiently by the developer, who can give feedback on a smaller number of solutions. This feedback is then used to generate constraints for the optimization process, to focus on the developer's regions of interest in both the decision and objective spaces. The manual validation of selected refactoring solutions by developers confirms that our approach outper-

[3] Alizadeh, V., Fehri, H., & Kessentini, M. Less is More: From Multi-objective to Mono-objective Refactoring via Developer's Knowledge Extraction. In 2019 19th IEEE International Conference on Source Code Analysis and Manipulation (SCAM) (pp. 181-192). IEEE.

forms state of the art refactoring techniques. This work is accepted at the IEEE Transactions in Software Engineering journal **TSE 2020** [4] [38].

5. The adoption of refactoring techniques for continuous integration received much less attention from the research community comparing to root-canal refactoring to fix the quality issues in the whole system. Several recent empirical studies show that developers, in practice, are applying refactoring incrementally when they are fixing bugs or adding new features. There is an urgent need for refactoring tools that can support continuous integration and some recent development processes such as DevOps that are based on rapid releases. Furthermore, several studies show that manual refactoring is expensive and existing automated refactoring tools are challenging to configure and integrate into the development pipelines with significant disruption cost.

   We presented a first attempt to propose an intelligent software refactoring bot, as GitHub app, that can **submit a pull-request to refactor recent code changes**. The salient feature of the proposed bot is that it incorporates interaction support, via our Web app, hence allowing developers to approve or modify or reject the applied code refactoring. The refactoring bot also provides support to explain why the refactorings are applied by quantifying the quality improvements. To evaluate the effectiveness of our technique, we applied it to four open-source and one industrial projects comparing it with state-of-the-art approaches. Our results show promising evidence on the usefulness of the proposed interactive refactoring bot. The participants highlighted the high usability of the bot in terms of easy integration with their development environments with the least configuration effort. An invention disclosure is approved for this work and the results are published at the 34th IEEE/ACM International

[4] Alizadeh, V., Fehri, H., Kessentini, & Kazman, R. (2020). Enabling Decision and Objective Space Exploration for Interactive Multi-Objective Refactoring. IEEE Transactions on Software Engineering.

Conference on Automated Software Engineering **ASE 2019**[5] [39, 40].

We note that the research contributions proposed in this thesis, generated 7 UM inventions, which are selected by the Michigan Translational Research and Commercialization (MTRAC) funding program to commercialize them and we are currently founding a startup with the UM Technology Transfer Office. One of these inventions (interactive refactoring) is selected by the UM Technology Transfer Office among the top 8 inventions of the year in 2019 from over 500 applications.

1.4 Organization of the Dissertation

This thesis is organized as follows: Chapter II introduces the current state of the art and related works to this thesis. Chapter III presents interactive refactoring recommendation approach. Chapter IV discusses our proposed approach to reduce refactoring effort for developers. Chapter V, describes our proposed method to convert multi-objective to mono-objective refactoring problem based on the user's preferences. We present our approach to enable the users to explore decision space of recommended refactorings in VI. Chapter VII describes our refactoring bot for CI/CD. Finally, a summary and future research directions are presented in VIII.

[5] Alizadeh, V., Ouali, M. A., Kessentini, M., & Chater, M. (2019, November). RefBot: Intelligent Software Refactoring Bot. In 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE) (pp. 823-834). IEEE.

CHAPTER II

State of the Art

2.1 Introduction

In this chapter, we cover the necessary background information related to our work followed by an overview of existing studies.

2.2 Background

In this section, we describe the required background to understand the proposed approaches. First, we give an overview about software refactoring. Then, several definitions related to interactive and dynamic multi-objective optimization are described.

2.2.1 Software Refactoring

Refactoring is defined as the process of improving the code after it has been written by changing its internal structure without changing its external behavior. The idea is to reorganize variables, classes and methods to facilitate future adaptations and enhance comprehension. This reorganization is used to improve different aspects of the software quality such as maintainability, extendibility, reusability, etc. Some modern Integrated Development Environments (IDEs), such as Eclipse, Netbeans, provide support for applying the most commonly used refactorings, e.g., move

method, rename class, etc.

In order to identify which parts of the source code need to be refactored, most of the existing work relies on the notion of bad smells (e.g., Fowler's textbook [9]), also called design defects or anti-patterns. Typically, code smells refer to design situations that adversely affect the development of the software. When applying refactorings to fix design defects, software metrics can be used as an overall indication of the quality of the new design. For instance, high intra-class cohesion and low inter-class coupling usually indicate a high-quality system.

Refactoring is one of the most used terms in software development and has played a major role in the maintenance of software for decades. While most developers have an intuitive understanding of the refactoring process, many of us lack a true mastery of this important skill. In this article, we will explore the textbook definition of refactoring, how this definition holds up to the reality of software development, and how we can ensure our codebase is prepared for refactoring. Along the way, we will walk-through an entire set of refactorings, from start to finish, to illustrate the simplicity and importance of this ubiquitous process.

Refactoring is one of the most self-evident processes in software development, but it is surprisingly difficult to perform properly. In most cases, we deviate from strict refactoring and execute an approximation of the process; sometimes, things work out and we are left with cleaner code, but other times, we get snared, wondering where we went wrong. In either case, it is important to fully understand the importance and simplicity of barebones refactoring.

In short, the process of refactoring involves taking small, manageable steps that incrementally increase the cleanliness of code while still maintaining the functionality of the code. As we perform more and more of these small changes, we start to transform messy code into simpler, easier to read, and more maintainable code. It is not a single refactoring that makes the change: It's the cumulative effect of many

Table 2.1: List of refactoring operations included in this thesis.

Refactoring	**Controlling Parameter**
Moving Features Between Objects	
Move Method	Source, Target, Method
Move Field	Source, Target, Attribute
Extract Class	Source, Target, Attributes, Methods
Organizing Data	
Encapsulate Field	Source, Attribute
Simplifying Method Calls	
Decrease Field Security	Source, Attribute
Decrease Method Security	Source, Method
Increase Field Security	Source, Attribute
Increase Method Security	Source, Method
Dealing with Generalization	
Pull Up Field	Source, Target, Attribute
Pull Up Method	Source, Target, Method
Push Down Field	Source, Target, Attribute
Push Down Method	Source, Target, Method
Extract SubClass	Source, Target, Attributes, Methods
Extract SuperClass	Source, Target, Attributes, Methods

small refactorings performed toward a single goal that makes the difference.

2.2.1.1 Refactoring Operations

The refactoring operations considered in the approaches proposed in this thesis cover the most used operations selected from different categories: "Moving features", "Data organizers", "Method calls simplifiers", and "Generalization modifiers". These refactorings are listed in Table 2.1. We selected these refactoring operations because they have the most impact on code quality attributes.

2.2.2 Interactive and Dynamic Evolutionary Multi-Objective Optimization

In this section, we give a brief overview about two important aspects in the Evolutionary Multi-objective Optimization (EMO) [41] paradigm related to the: (1) Interaction with the user and (2) Dynamicity of the problem.

Interacting with the human user means allowing the user to inject his/her preferences into the computational search algorithm and then using these preferences to guide the search process. To express his/her preferences, the user needs some preference modeling tools. The most commonly used ones are [41]:

- *Weights*: Each objective is assigned a weighting coefficient expressing its importance. The larger the weight is, the more important the objective is.

- *Solution ranking*: The user is provided with a sample of solutions (a subset of the current population) and is invited to perform comparisons between pairs of equally-ranked solutions in order to differentiate between solutions that the fitness function regards as equal.

- *Objective ranking*: Pairwise comparisons between pairs of objectives are performed in order to rank the problem's objectives where strong conflict exists between a pair of objectives.

- *Reference point* (also called a goal or an aspiration level vector): The user supplies, for each objective, the desired level that he/she wishes to achieve. This desired level is called aspiration level.

- *Reservation point* (also called a reservation level vector): The user supplies, for each objective, the accepted level that he/she wishes to reach. This accepted level is called reservation level.

- *Trade-off between objectives*: The user specifies that the gain of one unit in one objective is worth degradation in some others and vice versa.

- *Outranking thresholds*: The user specifies the necessary thresholds to design a fuzzy predicate modeling the truth degree of the predicate “solution x is at least as good as solution y.”

- *Desirability thresholds*: The user supplies: (1) an absolutely satisfying objective value and (2) a marginally infeasible objective value. These thresholds represent the parameters that define the desirability functions.

Based on these preference modeling tools, we observe that the goal of a preference-based EMO algorithm is to assign different importance levels to the problem’s objectives with the aim to guide the search towards the Region of Interest (ROI) that is the portion of the Pareto Front that best matches the user preferences. In fact, usually, the user is not interested with the whole Pareto front and thus he/she is searching only for his/her ROI from which the problem’s final solution will be selected. Several preference-based EMO algorithms have been proposed and used to solve real problems such as PI-EMOA [42], iTDEA [43], NOSGA [44], DF-SMS-EMOA [45], just to cite a few. There are several algorithmic challenges that should be overcome such as the preservation of Pareto dominance, the preservation of population diversity, the scalability with the number of objectives, etc.

Until now, the user’s preferences are expressed and handled in the objective space. It is important to highlight that one of the original aspects of our work, as detailed later, is allowing the user (a software developer) to express his/her preferences in the decision space and then handling these preferences to help the user finding the most desired refactoring solution. Moreover, our approach helps the user in eliciting his/her preferences, which is very important for any preference-based EMO algorithm. These preferences are introduced implicitly by moving between the Pareto front of non-

dominated solutions after obtaining feedback from the user about just a few parts of the solution in order to better understand his preferences. This implicit exploration of the Pareto front will be detailed in the next section where we describe the formulation of our refactoring problem.

The incorporation of user preferences may require the handling of dynamicity issues related to the introduced changes to the solution or the input (i.e. the software system). Handling dynamicity in EMO means solving dynamic problems where the objective functions and or the constraints may change over time such due to, for example, the dynamic nature of most of software evolution problems including software refactoring. Applying evolutionary algorithms (EAs) to solve Dynamic Multi-Objective Problems (DMOPs) has received great attention from researchers thanks to the adaptive behavior of evolutionary computation methods. A DMOP consists of minimizing or maximizing an objective function vector under some constraints over time. Its general form is the following[41]:

$$\begin{cases} Min f(x,t) = [f_1(x,t), f_2(x,t), ..., f_M(x,t)]^T & \\ g_j(x,t) \geq 0, & j = 1, ...P; \\ h_k(x,t) = 0, & \text{k=1,...,Q;} \\ x_i^L \leq x_i \leq x_i^U, & \text{i=1,...,n;} \end{cases}$$

where M is the number of objective functions, t is the time instant, P is the number of inequality constraints, Q is the number of equality constraints, X_i^L and x_i^U correspond respectively to the lower and upper bounds of the variable x_i .

A solution x_i satisfying the $(P + Q)$ constraints is said to be *feasible*, and the set of all feasible solutions defines the feasible search space denoted by Ω. In this formulation, we consider a minimization MOP since maximization can be easily turned into minimization based on the duality principle by multiplying each objective function

by -1 and transforming the constraints based on the duality rules.

The resolution of a MOP yields a set of trade-off solutions, called Pareto optimal solutions or non-dominated solutions, and the image of this set in the objective space is called the Pareto front. Hence, the resolution of a MOP consists in approximating the entire Pareto front. In the following, we provide some background definitions related to multi-objective optimization. It is worth noting that these definitions remain valid in the case of DMOPs.

Definition 1: Pareto optimality

A solution $x^* \in \Omega$ is Pareto optimal if $\forall x \in \Omega$ and $I = \{1, ..., M\}$ either $\forall m \in I$ we have $f_m(x) = f_m(x^*)$ or there is at least one $m \in I$ such that $f_m(x) > f_m(x^*)$.

The definition of Pareto optimality states that x^* is Pareto optimal if no feasible vector exists that would improve some objectives without causing a simultaneous worsening in at least one other objective.

Definition 2: Pareto dominance

A solution $u = (u_1, u_2, ..., u_n)$ is said to dominate another solution $v = (v_1, v_2, ..., v_n)$ (denoted by $f(u) \prec f(v)$) if and only if $f(u)$ is partially less than $f(v)$. In other words, $\forall m \in \{1, ..., M\}$ we have $f_m(u) \leq f_m(v)$ and $\exists m \in \{1, ..., M\}$ where $f_m(u) < f_m(v)$.

Definition 3: Pareto optimal set

For a given MOP $f(x)$, the Pareto optimal set is
$P^* = \{x \in \Omega | \neg\exists x' \in \Omega, f(x') \prec f(x)\}$.

Definition 4: Pareto optimal front

For a given MOP $f(x)$ and its Pareto optimal set P^*, the Pareto front is $PF^* = \{f(x), x \in P^*\}$.

2.2.3 Code Quality Metrics

Many studies have utilized structural metrics as a basis for defining quality indicators for a good system design [18, 51]. As an illustrative example, [46] proposed a

set of quality measures, using the ISO 9126 specification, called QMOOD. This model is developed based on international standard for software product quality measurement. QMOOD is a comprehensive way to assess the software quality and includes four levels.

We employed the first two levels known as "Design Quality Attributes" and "Object-oriented Design Properties" to calculate our fitness functions used in this thesis Reusability, Flexibility, Understandability, Functionality, Extendibility, Effectiveness, Complexity, Cohesion, Coupling). Each of these quality metrics is defined using a combination of low-level metrics as detailed in Tables 2.2 and 2.3.

The QMOOD model has been used previously in the area of search-based software refactoring [4], so we use it to estimate the effect of the suggested refactoring solutions on software quality. QMOOD has the advantage that it defines six high-level design quality attributes (reusability, flexibility, understandability, functionality, extendibility, and effectiveness) that can be calculated using 11 lower level design metrics.

Table 2.2: QMOOD design metrics.

Design Metric	**Design Property**	**Description**
Design Size in Classes (DSC)	Design Size	Total number of classes in the design.
Number Of Hierarchies (NOH)	Hierarchies	Total number of "root" classes in the design (*count(MaxInheritenceTree (class)=0)*)
Average Number of Ancestors (ANA)	Abstraction	Average number of classes in the inheritance tree for each class.
Direct Access Metric (DAM)	Encapsulation	Ratio of the number of private and protected attributes to the total number of attributes in a class.
Direct Class Coupling (DCC)	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 ($CAMC$)	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 (MOA)	Composition	Count of number of attributes whose type is user defined class(es).
Measure of Functional Abstraction (MFA)	Inheritance	Ratio of the number of inherited methods per the total number of methods within a class.
Number of Polymorphic Methods (NOP)	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 (CIS)	Messaging	Number of public methods in class.
Number of Methods (NOM)	Complexity	Number of methods declared in a class.

Table 2.3: QMOOD quality attributes.

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 design's allowance 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.
	$0.2*Abstraction+0.2*Encapsulation+0.2*Composition+ 0.2 * Inheritance + 0.2 * Polymorphism$

2.3 Related Work

2.3.1 Manual Refactoring

We start, this section, by summarizing existing manual approaches for software refactoring. In Fowler's book [9] a non-exhaustive list of low-level design problems in source code has been defined. For each type of code smell, a list of possible refactorings is suggested that can be applied by the developers. Du Bois *et al.* [47] start from the

hypothesis that refactoring opportunities correspond to those that improve cohesion and coupling metrics, and use this to perform an optimal distribution of features over classes. They analyze how refactorings manipulate coupling and cohesion metrics, and how to identify refactoring opportunities that improve these metrics. However, this approach is limited to only certain refactoring types and a small number of quality metrics. Murphy-Hill *et al.* [48, 49] proposed several techniques and empirical studies to support refactoring activities. In [49, 50], the authors proposed new tools to assist software developers in applying refactoring such as selection assistant, box view, and refactoring annotation based on structural information and program analysis techniques.

Recently, Ge and Murphy-Hill [51] have proposed a new refactoring tool called GhostFactor that allows the developer to transform code manually, but checks the correctness of the transformation automatically. BeneFactor [52] and WitchDoctor [53] can detect manual refactorings and then complete them automatically. Tahvildari *et al.* [54] also propose a framework of object-oriented metrics used to suggest to the software developer refactoring opportunities to improve the quality of an object-oriented legacy system. Dig [55] proposes an interactive refactoring technique to improve the parallelism of software systems. However, the proposed approach did not consider learning from the developers' feedback and focused on making programs more parallel. Other contributions are based on rules that can be expressed as assertions (invariants, pre- and post-conditions). All these techniques are more concerned around the correctness of manually applied refactorings rather than interactive recommendations.

The use of invariants has been proposed to detect parts of the program that require refactoring [56]. In addition, Opdyke [10] has proposed the definition and use of pre- and post-conditions with invariants to preserve the behavior of the software when applying refactorings. Hence, behavior preservation is based on the verification/sat-

isfaction of a set of pre- and post-condition. All these conditions are expressed as first-order logic constraints expressed over the elements of the program.

To summarize, manual refactoring is a tedious task for developers that involves exploring the software system to find the best refactoring solution that improves the quality of the software and fix design defects.

2.3.2 Automated Refactoring

To automate refactoring activities, new approaches have been proposed [57, 58, 59, 60, 61, 62, 63, 64, 65]. JDeodorant [5] is an automated refactoring tool implemented as an Eclipse plug-in that identifies certain types of design defect using quality metrics and then proposes a list of refactoring strategies to fix them. Search-based techniques [66] are widely studied to automate software refactoring and consider it as an optimization problem, where the goal is to improve the design quality of a system based mainly on a set of software metrics. The majority of existing work combines several metrics in a single fitness function to find the best sequence of refactorings. Seng *et al.* [67] have proposed a single-objective optimization approach using a genetic algorithm to suggest a list of refactorings to improve software quality. The work of O'Keeffe *et al.* [4] uses various local search-based techniques such as hill climbing and simulated annealing to provide an automated refactoring support. They use the QMOOD metrics suite [46] to evaluate the improvement in quality.

Kessentini *et al.* [1] have proposed single-objective combinatorial optimization using a genetic algorithm to find the best sequence of refactoring operations that improve the quality of the code by minimizing as much as possible the number of design defects detected in the source code. Kilic *et al.* [68] explore the use of a variety of population-based approaches to search-based parallel refactoring, finding that local beam search could find the best solutions. Harman *et al.* [10] have proposed a search-based approach using Pareto optimality that combines two quality metrics, CBO

(coupling between objects) and SDMPC (standard deviation of methods per class), in two separate fitness functions. Ouni *et al.* [69] proposed also a multi-objective refactoring formulation that generates solutions to fix code smells. Ó Cinnéide *et al.* [70] have proposed a multi-objective search-based refactoring to conduct an empirical investigation to assess some structural metrics and to explore relationships between them. They have used a variety of search techniques (Pareto-optimal search, semi-random search) guided by a set of cohesion metrics.

The majority of existing multi-objective refactoring techniques propose as output a set of non-dominated refactoring solutions (the Pareto front) that find a good trade-off between the considered maintainability objectives. This leaves it to the software developers to select the best solution from a set of possible refactoring solutions, which can be a challenging task as it is not natural for developers to express their preferences in terms of a fitness functions value. Thus, the exploration of the Pareto front is still performed manually, which limits the use of multi-objective search techniques to address software engineering problems. An intelligent exploration of the Pareto front is required to expand the applicability of multi-objective techniques for search-based software engineering problems.

In summary, developers should accept the entire refactoring solution and existing tools do not provide the flexibility to adapt the suggested solution in existing fully-automated refactoring techniques. Furthermore, existing automated refactoring tools execute the whole algorithm again to suggest new refactorings after a number of code changes are introduced by developers, rather than simply trying to update the proposed solutions based on the new code changes. While automation is important, it is essential to understand the points at which human oversight, intervention, and decision-making should impact on automation. Human developers might reject changes made by any automated programming technique. Especially if they feel that they have little control, there will be a natural reluctance to trust and use the

automated refactoring tool [71].

2.3.3 Interactive Refactoring

Interactive techniques have been generally introduced in the literature of Search-Based Software Engineering and especially in the area of software modularization. Hall *et al.* [72] treated software modularization as a constraint satisfaction problem. The idea of this work is to provide a baseline distribution of software elements using good design principles (e.g. minimal coupling and maximal cohesion) that will be refined by a set of corrections introduced interactively by the designer.

The approach, called SUMO (Supervised Re-modularization), consists of iteratively feeding domain knowledge into the remodularization process. The process is performed by the designer in terms of constraints that can be introduced to refine the current modularizations. Initially, the system begins with generating a module dependency graph from an input system. This dependency is based on the correlation between software elements (coupling between methods, shared attributes etc.). Possible modularizations are then generated from the graph using multiple simulated authoritative decompositions. Then, using a clustering technique called Bunch, an initial set of clusters is generated that serves as an input to SUMO.

The SUMO algorithm provides a hypothesized modularization to the user, who will agree with some relations, and disagree with others. The user's corrections are then integrated into the modularization process, to generate a more satisfactory modularization. The SUMO algorithm does not necessarily rely on clustering techniques, but it can benefit from their output as a starting point for its refinement process.

Bavota *et al.* [73] presented the adoption of single objective interactive genetic algorithms in software re-modularization process. The main idea is to incorporate the user in the evaluation of the generated remodularizations. Interactive Genetic Algorithms (IGAs) extend the Classic Genetic Algorithms (GAs) by partially or entirely

involving the user in the determination of the solution's fitness function. The basic idea of the Interactive GA (IGA) is to periodically add a constraint to the GA such that some specific components shall be put in a given cluster among those created so far. Initially, the IGA evolves similarly to the non-interactive GA.

After a user-defined set of iterations, the individual with the highest fitness value is selected from the population set (in the case of single-objective GA) or from the first front (in the case of multi-objective GA) and presented to the user. After analyzing the current modularization, the user provides feedback in terms of constraints dictating for example, that a specific element needs to be in the same cluster as another one. Although user feedback is important in guaranteeing convergence, it is essential not to overload the user by asking for a decision about all the current relationships between elements, especially for a large system.

Overall, the above existing studies of interactive remodularization are limited to few types of refactoring such as moving classes between packages and splitting packages. Furthermore, the interaction mechanism is based on the manual evaluation of proposed remodularization solutions which could be a time-consuming process. The proposed interactive remdoularization techniques are also based on a mono-objective algorithm and did not consider multiple objectives when evaluating the solutions.

A recent study [74] extended our previous work [75] to propose an interactive search based approach for refactoring recommendations. The developers have to specify a desired design at the architecture level then the proposed approach try to find the relevant refactorings that can generate a similar design to the expected one. In our work, we do not consider the use of a desired design, thus developers are not required to manually modify the current architecture of the system to get refactoring recommendations. Furthermore, developers maybe interested to change the architecture mainly when they want to introduce an extensive number of refactorings that radically change the architecture to support new features.

Several possible levels of interaction are not considered by existing refactoring techniques. It is easy for developers to identify large classes or long methods that should be refactored, but they find it is difficult, in general, to locate a target class when applying a move method refactoring [76]. In addition, existing refactoring tools do not update their recommended refactoring solutions based on the software developer's feedback such as accepting, modifying or rejecting certain refactoring operations.

Furthurmore, None of the above interactive studies considered reducing the interaction effort with developers which is an important step to improve the applicability of refactoring tools as highlighted in the survey with developers.

To address the above-mentioned limitations, we proposed in this proposal, a new way for software developers to refactor their software systems as a sequence of transformations based on different levels of interaction, implicit exploration of non-dominated refactoring solutions and dynamic adaptive ranking of the suggested refactorings.

2.3.4 Search Based Software Refactoring

Search-based techniques [66] are widely studied to automate software refactoring where the goal is to improve the design quality of a system based mainly on a set of software metrics. The majority of existing work combines several metrics in a single fitness function to find the best sequence of refactorings. Seng et al. [67] have proposed a single-objective optimization approach using a genetic algorithm to suggest a list of refactorings to improve software quality. The work of O'Keeffe et al. [4] uses various local search-based techniques such as hill climbing and simulated annealing to provide an automated refactoring support. They use the QMOOD metrics suite [46] to evaluate the improvement in quality. Kessentini et al. [1] have proposed single-objective combinatorial optimization using a genetic algorithm to find

the best sequence of refactoring operations that improve the quality of the code by minimizing as much as possible the number of design defects detected in the source code. Kilic et al. [68] explore the use of a variety of population-based approaches to search-based parallel refactoring, finding that local beam search could find the best solutions. Harman et al. [10] have proposed a search-based approach using Pareto optimality that combines two quality metrics, CBO (coupling between objects) and SDMPC (standard deviation of methods per class), in two separate fitness functions. Ouni et al. [69] proposed also a multi-objective refactoring formulation that generates solutions to fix code smells. Ó Cinnéide et al. [70] have proposed a multi-objective search-based refactoring to conduct an empirical investigation to assess some structural metrics and to explore relationships between them. They have used a variety of search techniques (Pareto-optimal search, semi-random search) guided by a set of cohesion metrics.

The majority of existing multi-objective refactoring techniques propose as output a set of non-dominated refactoring solutions (the Pareto front) that find a good trade-off between the considered maintainability objectives. This leaves it to the software developers to select the best solution from a set of possible refactoring solutions, which can be a challenging task as it is not natural for developers to express their preferences in terms of a fitness functions value. Thus, the exploration of the Pareto front is still performed manually, which limits the use of multi-objective search techniques to address software engineering problems. An intelligent exploration of the Pareto front is required to expand the applicability of multi-objective techniques for search-based software engineering problems as addressed in this proposal.

2.3.5 Refactoring Recommendation

Much effort has been devoted to the definition of approaches supporting refactoring. One representative example is JDeodorant, the tool proposed by Tsantalis and

Chatzigeorgiou [25].Our paper is mostly related to approaches exploiting search-based techniques to identify refactoring opportunities, and our discussion focuses on them since the bot is based on multi-objective refactoring. We point the interested reader to the survey by Bavota *et al.* [77] for an overview of approaches supporting code refactoring.

O'Keeffe and Cinnéide [78] 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. Such a search is guided by a quality evaluation function based on eleven object-oriented design metrics that reflect refactoring goals. Harman and Tratt [3] 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 [3].

The two aforementioned works [78, 3] paved the way to several search-based approaches aimed at recommending refactoring operations [67, 1, 24, 75, 79, 2]. Several other studies proposed refactorings at the model level as well[80, 81, 82, 83, 84, 85, 86, 87]. A representative example of these techniques is the recent work by Alizadeh et al. *et al.* [6], who proposed an interactive multi-criteria code refactoring approach to improve the QMOOD quality metrics while minimizing the number of refactorings. In our approach, we decided to rely on a simpler optimization algorithm by only considering the refactoring of recently changed files in other pull requests rather than the root-canal refactoring approach of Alizadeh et al. *et al.* [6].

2.3.6 Empirical Studies on Refactoring

Empirical studies on software refactoring mainly aim at investigating the refactoring habits of software developers and the relationship between refactoring and code

quality.

We only discuss studies reporting findings relevant to our work. Murphy-Hill *et al.* [71] investigated how developers perform refactorings. Examples of the exploited datasets are usage data from 41 developers using the Eclipse environment and information extracted from versioning systems. Among their several findings, they show that developers often perform *floss refactoring*; namely, they interleave refactoring with other programming activities, confirming that refactoring is rarely performed in isolation. Kim *et al.* [88] present a survey of software refactoring with 328 Microsoft engineers. They show that the major obstacle of adopting many existing refactoring tools is their configuration and painful integration within their pipelines without disturbing developers with their current focus in terms of meeting deadlines and making regular code changes. Those findings stress out the need for refactoring bots that can be adopted for continuous integration without considerable configuration effort.

2.3.7 Software Bots

The design and implementation of software bots are still in its infancy with a significant focus on chatbots. For instance, Lebeuf et al. *et al.* [89, 90] discussed the potential of using chat bots in software engineering and how they can be helpful to increase collaborations between programmers. The authors also proposed a possible classification of potential benefits of using software bots in various domains, especially to improve the productivity of developers.

An extensive empirical study of over 90 software bots was performed by Wessel et al.*et al.* [91] to provide a classification and taxonomy for them. They found that around 21 bots were actually tried on GitHub repositories and the dominant majority are around testing but without providing any code actions or recommendations to developers. The authors found that none of these bots provides explanations of their analysis which reduced the adoption by developers.

Some examples of regression testing bots include Travis CI and the bot designed by Urli et al. [92] to repair bugs. These tools did not open a new pull-request, but they are executed manually by the developers where they can check the recommended patches. Another bot related to quality assessment but not refactoring is Fix-it *et al.* [93]. It is mainly limited to a few types of code changes, mainly targeting dynamic analysis metrics.

Finally, Wyrich et al. *et al.* [94] proposed a vision paper to emphasize the importance of refactoring bots and motivates their potential use in practice. They proposed a prototype, not a complete bot, by running SonarQube to detect code smells. However, the work is still in its initial stage where refactorings are not recommended yet.

2.4 Summary of Systematic Literature Review on Refactoring

Due to the growing complexity of software systems, there has been a dramatic increase and industry demand for tools and techniques on software refactoring in the last ten years, defined traditionally as a set of program transformations intended to improve the system design while preserving the behavior. Refactoring studies are 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 design to include other non-functional requirements (e.g., improve performance, security, etc). Thus, challenges to be addressed by refactoring work are nowadays beyond code transformation to include, but not limited to, scheduling the opportune time to carry refactoring, recommendations of specific refactoring activities, detection of refactoring opportunities and testing the correctness of

applied refactorings. Therefore, the refactoring research efforts are fragmented over several research communities, various domains, and different objectives. To structure the field and existing research results, we provide a systematic literature review and analyzes the results of about 2800 research papers on refactoring covering the last two decades to offer the most scalable and comprehensive literature review of existing refactoring research studies. Based on this survey, we created a taxonomy to classify the existing research, identified research trends and highlighted gaps in the literature and avenues for further research.

Several studies [95, 96] show that programmers are postponing software maintenance activities that improve software quality, even while seeking high-quality source code for themselves. In fact, the time and monetary pressures force programmers to neglect improving the quality of their source code [13]. 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. To get a deep understanding of the current state of the field and existing research results, we first conducted a systematic literature review (SLR) and analyzed over 2800 research papers on refactoring, spanning the last two decades. This SLR offers the most scalable and comprehensive literature review of refactoring research to date. Based on our SLR, we created a taxonomy to classify the existing research, identified research trends, and highlighted gaps in the literature and avenues for further research. Refactoring is among the fastest growing software engineering research areas, if not the fastest. Figure 2.1 shows the dramatic growth of the refactoring field during the last decade. During just the last three years (2014-2016), over 850 papers were published in the field with an average of 270 papers each year. Over 4990 authors from all over the world contributed to the field of software refactoring. We highlight the most active authors in Figure 2.2, based on both number of publications and citations in the area. As seen in Figure 2.3, most of the active refactoring researchers are located in the

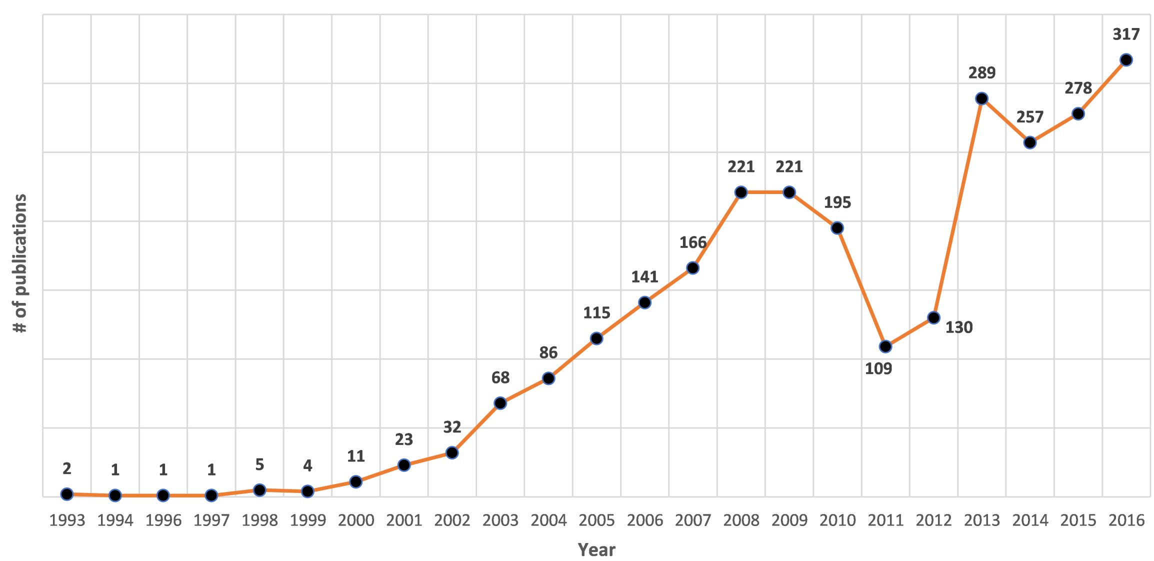


Figure 2.1: Number of refactoring publications over the last two decades.

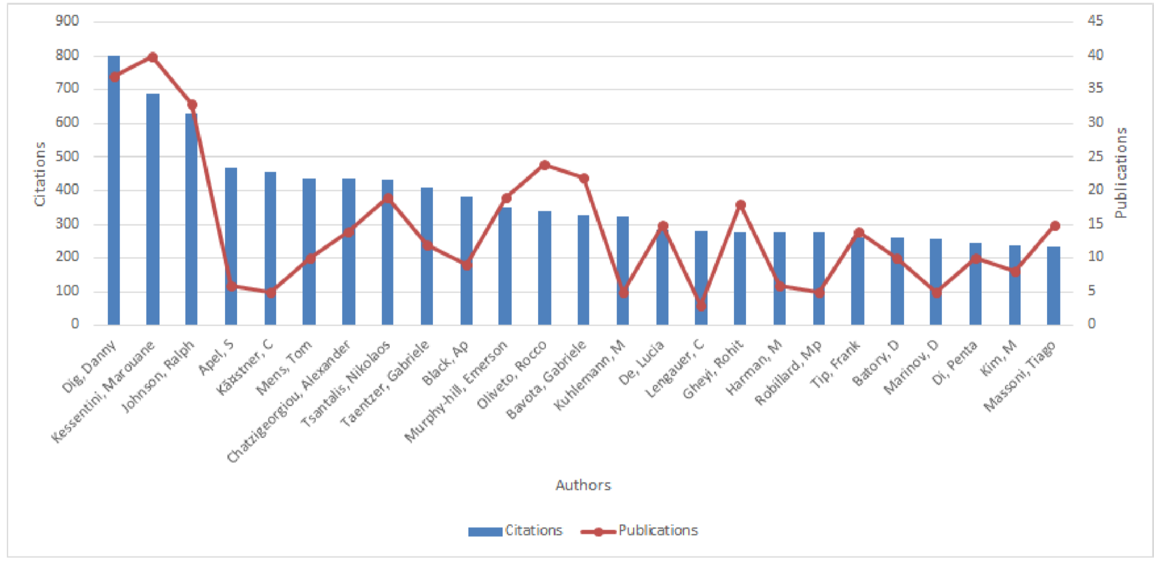


Figure 2.2: Leading refactoring researchers over the last decade based on both publications and citations.

US, thus motivating the proposed infrastructure in US.

Figures 2.4 highlight that refactoring research has expanded significantly since its inception in the early 90s. Refactoring now expands beyond code-restructuring and targets different artefacts (architecture, model, requirements, etc.) [9, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 97], is pervasive in many domains beyond the object-oriented paradigm (cloud computing, mobile, web, etc.)[98, 63, 99, 100, 82, 101, 102,

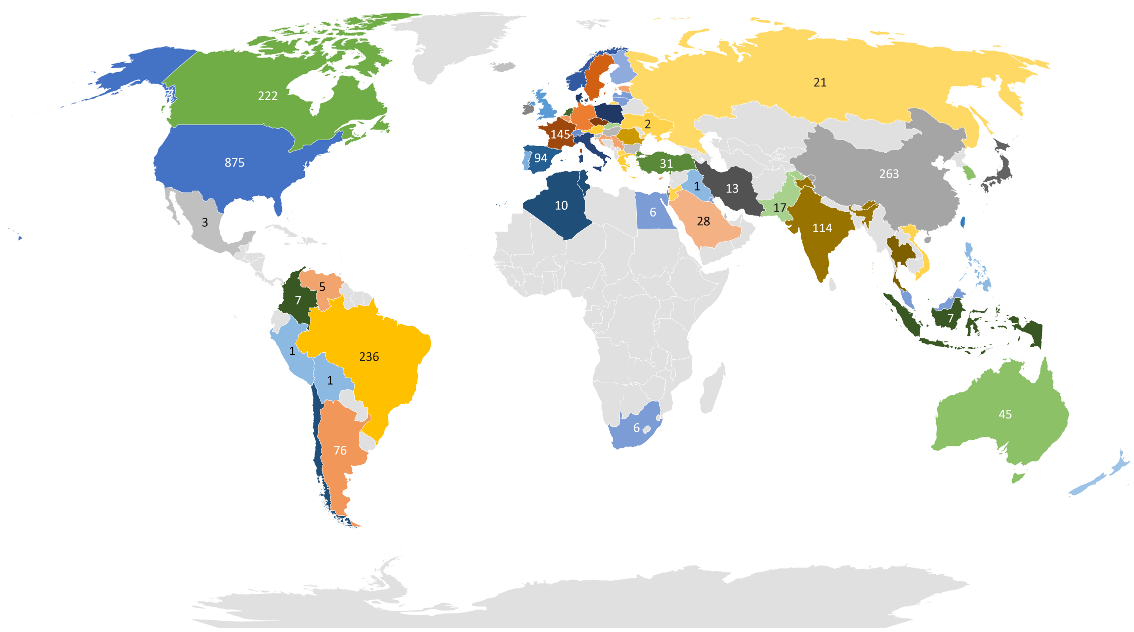


Figure 2.3: Distribution of refactoring researchers around the world.

103, 104, 105], is widely adopted in industrial settings [69,71], and the objectives expand beyond improving design into other nonfunctional requirements (e.g., improve performance, security, etc) [97, 55, 106, 107, 108, 109, 23]. The focus of the refactoring community nowadays goes beyond code transformation to include, but not limited to, scheduling the opportune time to carry refactoring [110, 111, 47, 112], recommending specific refactoring activities [97, 75, 55, 47, 113, 114, 71, 115, 116, 2, 117], inferring refactorings from the code [18, 88], and testing the correctness of applied refactorings [118, 71, 112]. Therefore, the refactoring research efforts are fragmented over several research communities, various domains, and different objectives, motivating the need for a shared infrastructure to promote reuse and collaboration.

Manual refactoring can be challenging and error-prone. Many Integrated development environment (IDE) and software programming tools have implemented refactoring techniques in their products as a recommendation/guideline or partially/fully automated. Based on a survey [119], 38% of developers answered that the refactoring engine of an IDE was used and 7% of them stated that refactoring was done partially

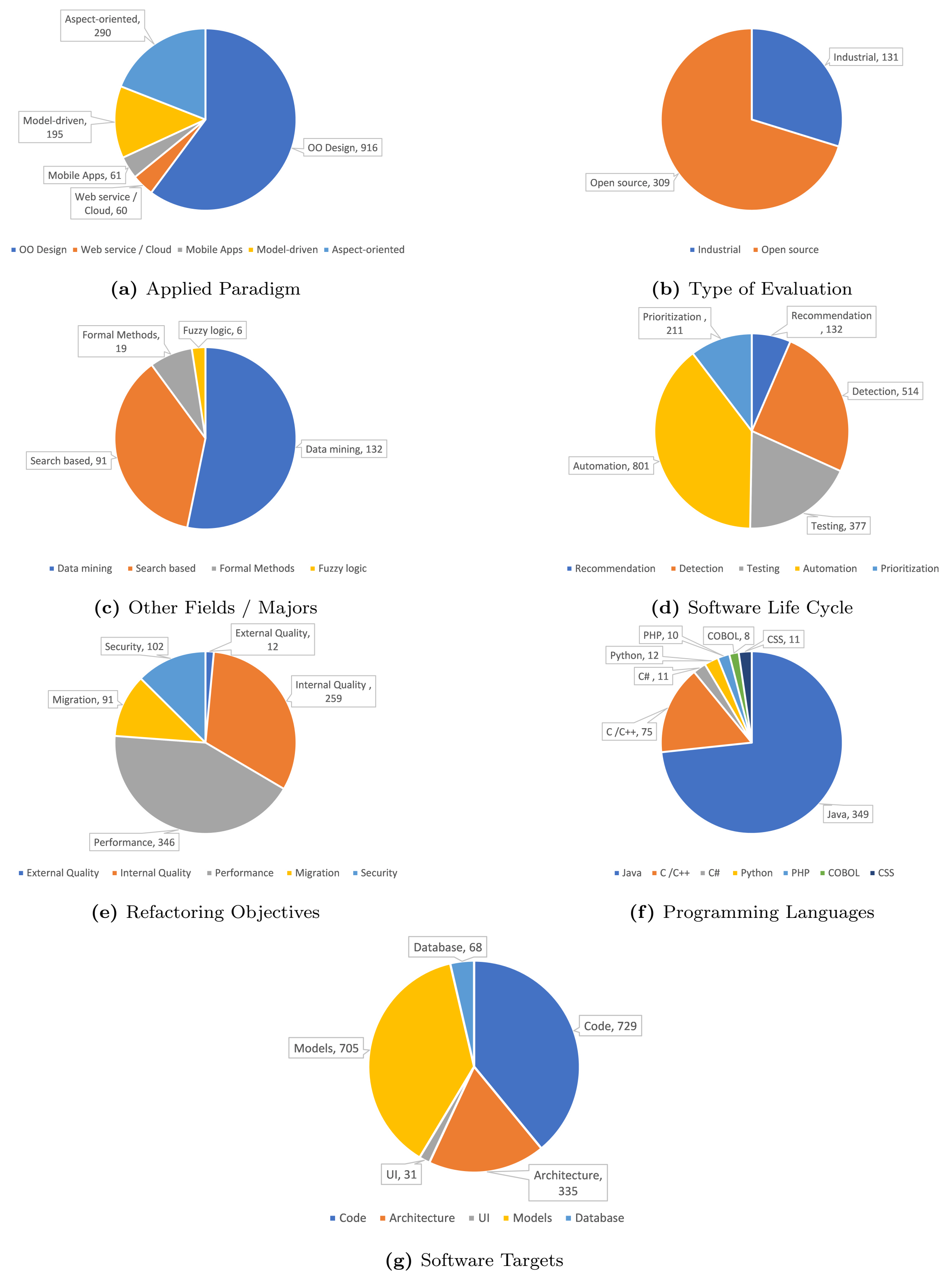


Figure 2.4: Taxonomy of refactoring researches and the number of publications during the past two decade.

automated. The main reasons for developers to do refactoring manually is that they do not trust automated process for complex refactoring techniques or the necessary modification is not supported in their choice of IDE. In another study [71], authors pointed out three factors - awareness, trust, and opportunity- and issues with tool work-flow as the limitations affecting usage of tools for refactoring. Therefore, this study can be useful for people from industry and market to be updated from the latest advancements in refactoring.

CHAPTER III

Interactive Multi-Objective Refactoring

3.1 Introduction and Problem Statement

Successful software products evolve through a process of continual change. However, this process may weaken the design of the software and make it unnecessarily complex, leading to significantly reduced productivity, increased fault-proneness and cost of maintenance, and has even led to projects being canceled. Many studies report that software maintenance activities consume up to 90% of the total cost of a typical software project. It has also been shown that software developers typically spend around 60% of their time in understanding the code they are maintaining [120].

Clearly, software developers need better ways to manage and reduce the growing complexity of software systems and improve their productivity. The standard solution is refactoring, which involves improving the design structure of the software while preserving its functionality [9]. There has been much work done on various techniques and tools for software refactoring [121, 48, 122] and these approaches can be classified into three main categories: *manual*, *semi-automated* and *fully-automated* approaches, as outlined below.

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. It may seem surprising that a developer would es-

chew the use of tools in this way, but Murphy-Hill *et al.* [71] found in their empirical study of the developers usage of the Eclipse refactoring tooling that in almost 90% of cases the developers performed refactorings manually and did not use any automated refactoring tools.

Kim *et al.* [123] confirmed this observation, finding that the interviewed developers from Microsoft preferred to perform refactoring manually in 86% of cases. In spite of its apparent popularity, manual refactoring is very limited however; several studies have shown that 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 [124].

By semi-automated refactoring, we refer to the situation where a developer uses the standard refactoring tooling available in IDEs such as Eclipse and Netbeans to apply the refactorings they deem appropriate. Murphy-Hill *et al.* [71] analyzed data collected from 13,000 Java developers using the Eclipse IDE over a 9-month period, finding that the trivial Rename refactoring accounted for almost 72% of the refactorings performed, while the combination of Rename, Extract Method/Variable and Move accounted for 89.3% of the total number of refactorings performed.

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 [2]. Secondly, it lacks flexibility since the developer has to either accept or reject the entire refactoring solution. Thirdly, it fails to consider the developer 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 applied [49].

In light of the discussion above, we propose an approach to refactoring recommendation that (1) provides refactoring-centric interaction, (2) enables refactoring and development to proceed in parallel and (3) collects 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 to creating a solution that the developer understands and can work with.

We propose that this interaction should be centered on refactorings, which are of direct interest to a developer, rather than code smells or software quality metrics, which have been found not to be strong drivers of the refactoring process in practice [11, 12]. Refactoring and development must be allowed to proceed in parallel, as this is part of test-driven development [125] and the Agile approach to software development in general [126]. Thus the developer can continue to extend the program with new functionality or bug fixes while the refactoring recommendation process executes. Finally, any development carried out is used where possible to improve the refactoring recommendations, e.g., the developer is more likely to value refactorings that affect recently updated code.

Our goal is to present the developer with few refactorings at a time, allowing them to accept / reject/ modify each refactoring as they see it. Thus, developers are not forced to either accept or run the entire refactoring operations or reject them and the developers may not control the number the applied refactorings. In our

approach, the developers can apply operations to the extent that they want. Finding a refactoring solution is a naturally multi-objective problem, so there is not one single "best" solution, rather there is a set of non-dominated solutions, the so-called Pareto front [127].

We use the multi-objective evolutionary algorithm NSGA-II [127] to create the Pareto front, using a fitness function that aims to improve software quality metrics while maintaining design coherence and reducing the number of recommended refactorings. The question we face is how to choose one solution from this front to present to the developer? The traditional approach is to seek a "knee point" on the front, but this ignores the fact that developers have their own refactoring priorities and may prefer a refactoring solution elsewhere on the front. To this end, *we propose, for the first time in search-based software refactoring, the use of innovization (innovation through optimization) [128] to analyze and explore the Pareto front interactively and implicitly with the developer.* Innovization is a technique that seeks interesting commonalities among the solutions of the Pareto front with the aim of developing a deeper understanding of the problem.

Our innovization algorithm starts by finding the most frequently-occurring refactorings among the set of non-dominated refactoring solutions. Based on this analysis, a complete refactoring solution is chosen from the front that best matches the most frequently-occurring refactorings, i.e., one that best represents the entire front in some sense. The recommended refactorings are then ranked and suggested to the developer one by one.

The developer can approve, modify or reject each suggested refactoring. Each such action by the developer is fed back into the search process. For example, if the developer rejects a refactoring, the search process will subsequently avoid this refactoring in creating new solutions. After the software has been changed to some degree, i.e. the developer has changed it by adding new functionality, fixing some bugs

or applying some refactorings and/or has provided feedback by rejecting a number of refactorings, NSGA-II will continue to execute in the new modified context to repair the set of good refactoring solutions based on the updated code and the feedback received from the developer. The feedback received from the developers will be also used as a set of new constraints to consider for the next iterations of NSGA-II. The algorithm will avoid, for example, including rejected refactorings by the developer when generating new solutions or repairing existing ones. However, the algorithm is not based on simply discarding all refactoring suggestions rejected by developer since adding new constraints to reduce the search space may make the current recommended refactoring solutions invalid.

We implemented our proposed approach and evaluated it on a set of eight open source systems and two industrial systems provided by our industrial partner, the Ford Motor Company. Statistical analysis of our experiments showed that our proposal performed significantly better than four existing search-based refactoring approaches [3, 4, 1, 76] and an existing refactoring tool not based on heuristic search, JDeodorant [5]. In our qualitative analysis, we found that the software developers who participated in our experiments confirmed the relevance of the suggested refactorings and the flexibility of the tool in modifying and adapting the suggested refactorings.

This approach is built on our previous work [75] extending it in several ways: (1) the interaction mechanism is improved, we define a new ranking function and different algorithm to repair non-dominated solutions after interactions with developers, (2) ten software applications are studied rather than five, (3) the number of participants in the experiments is doubled from 11 to 22, (4) an entirely new set of experimental results are presented and analyzed in far greater detail, (5) a comparison with a larger set of existing refactoring techniques is included.

It also extends our previous study [2] where we proposed a fully-automated, multi-objective approach to find the best refactoring solutions that improve software quality

metrics and reduce the number of recommended refactorings. In [2], we did not consider any developer interaction (fully-automated approach) and did not update/repair refactoring solutions based on new code changes introduced by developers. A recent study [74] extended our previous work [75] to propose an interactive search based approach for refactoring recommendations. The developers have to specify a desired design at the architecture level then the proposed approach try to find the relevant refactorings that can generate a similar design to the expected one. In our work, we do not consider the use of a desired design, thus developers are not required to manually modify the current architecture of the system to get refactoring recommendations.

The primary contributions of this work can be summarized as follows:

1. We introduce a novel interactive way to refactor software systems using innovization and interactive dynamic multi-objective optimization. The proposed technique supports the adaptation of refactoring solutions based on developer feedback while also taking into account other code changes that the developer may have performed in parallel with the refactoring activity.

2. We propose an implicit exploration of the Pareto front of non-dominated solutions based on our novel interactive approach that can help software developers to use multi-objective optimization for software engineering problems, avoiding the necessity for manual exploration of the Pareto front to find the best trade-off between the objectives.

3. We report the results of an empirical study on an implementation of our approach. The obtained results provide evidence to support the claim that our proposal is more efficient, on average, than existing refactoring techniques based on a benchmark of eight open source systems and two industrial projects. We also evaluate the relevance and usefulness of the suggested refactorings for software developers in improving the quality of their systems.

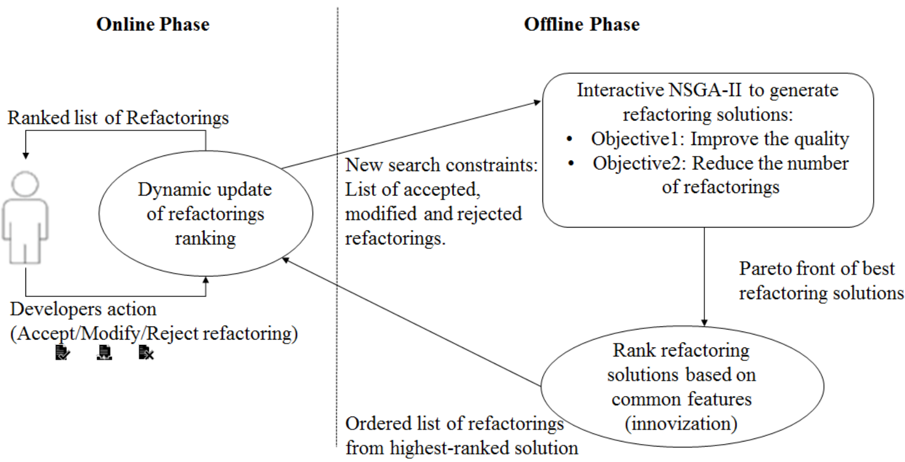


Figure 3.1: Approach overview.

3.2 Approach: Search-based Interactive Refactoring Recommendation

We first detail an overview of our approach and then we provide the details of our problem formulation and the solution approach.

3.2.1 Approach Overview

The goal of our approach is to propose a new dynamic interactive way for software developers to refactor their systems. The general structure of our approach is sketched in Fig. 3.1.

Our technique comprises two main components. The first component is an *offline phase*, executed in the background, when developers are modifying the source code of the system. During this phase, the multi-objective algorithm, NSGA-II, is executed for a number of iterations to find the non-dominated solutions balancing the two objectives of improving the quality, which corresponds to minimizing the number of code smells, maximizing/preserving the semantic coherence of the design and improving the QMOOD (Quality Model for Object-Oriented Design) quality metrics, and the second objective of minimizing the number of refactorings in the proposed solutions.

The output of this first step of the *offline phase* is a set of Pareto-equivalent refactoring solutions that optimizes the above two objectives. The second step of the offline phase explores this Pareto front in an intelligent manner using innovization to rank recommended refactorings based on the common features between the non-dominated solutions. In our adaptation, we assume true the hypothesis that the most frequently occurring refactorings in the non-dominated solutions are the most important ones. Thus, the output of this second step of the offline phase is a set of ranked solutions based on this frequency score. NSGA-II is able to generate not only one good refactoring solution, but a diverse set of non-dominated solutions. This set of refactoring solutions may include specific patterns that make them better and different than dominated (imperfect) refactoring solutions.

To extract these patterns, we used the heuristic of prioritizing the recommendation of refactorings that are the most redundant ones among the non-dominated solutions. To our intuition, it seems very likely that common patterns in the set of non-dominated solutions are very likely to be good patterns. The opposite situation, where some non-dominated solutions share a pattern that in of poor quality, seems highly unlikely, though it could plausibly occur were the poor quality pattern to be an essential enabling feature for another pattern of high quality. While we are only expressing an intuition here, innovization has proven itself to be of value later in the experiments section.

The second component of our approach is an *online phase* to manage the interaction with the developer. It dynamically updates the ranking of recommended refactorings based on the feedback of the developer. This feedback can be to approve/apply or modify or reject the suggested refactoring one by one as a sequence of transformations. Thus, the goal is to guide, *implicitly*, the exploration of the Pareto front to find good refactoring recommendations. Since the ranking is updated dynamically, our interactive algorithm allows the implicit move between non-dominated

solutions of the Pareto front.

After a number of interactions, developers may have modified or rejected a high number of suggested refactorings or have introduced several new code changes (new functionalities, fix bugs, etc.). Whenever the developers stop the refactoring session by closing the suggestions window, the first component of our approach is executed again on the background to update the last set of non-dominated refactoring solutions by continuing the execution of NSGA-II based on the two objectives defined in the first component and also the new constraints summarizing the feedback of the developer. In fact, we consider the rejected refactorings by the developer as constraints to avoid generating solutions containing several already rejected refactorings. This may lead to reducing the search space and thus a fast convergence to better solutions. Of course, the continuation of the execution of NSGA-II takes as input the updated version of the system after the interactions with developers.

The whole process continues until the developers decide that there is no necessity to refactor the system any further.

3.2.2 Adaptation

We describe in the following subsections the details of the various components of our framework.

3.2.2.1 Multi-objective formulation

In our previous work [2], we proposed a fully automated approach, to improve the quality of a system while preserving its domain semantics. It uses multi-objective optimization based on NSGA-II to find the best compromise between code quality improvements and reducing the number of code changes.

In this current work, we introduce the interactive component to our NSGA-II algorithm, which radically changes the process of finding good refactoring solutions

in comparison to our earlier work. We will compare later in the experiments the performance of both algorithms. We present in the following the different adaptation steps of our approach.

We ignored in this new interactive approach two objectives considered in our previous automated refactoring work. These two objectives are used to estimate, preserve and improve the design coherence (semantics) when fully automatically refactoring software systems. The very initial version of our experiments actually added the interaction, dynamic and innovization components at the top of our previous work. However, we found that the user interactions and the constraints learned and generated from it provided the required guidance to avoid "semantics" incoherences. Furthermore, the consideration of a large number of objectives make the execution time much longer to converge towards acceptable solutions since an increase in the number of objectives will increase the number of non-dominated solutions to analyze which is not suitable for interactive optimization algorithms since it will introduce noise in the search. Thus, we considered the textual measures as constraints to satisfy when generating the refactoring solutions rather than an objective to optimize as highlighted later. The users interaction history is sufficient based on our experiments thus we ignored the use of development history in our new interactive approach.

As explained in Algorithm 1, the process starts with a complete execution of a regular NSGA-II algorithm based on the objectives described in the previous section (*offline* phase) then three components are introduced to improve the recommendations: innovization, interactive and dynamic components.

The first iterations of the algorithm identify the Pareto front of the non-dominated refactoring solutions based on the fitness functions that will be discussed later. Then, the innovization component ranks the different non-dominated solutions based on the most common refactoring patterns between them. The different ranked refactorings are presented to the user based on the interactive component. During this interactive

Algorithm 1 Dynamic Interactive NSGA-II at generation t

```
1: Input
2: Sys: system to evaluate, Pt: parent population
3: Output
4: P_{t+1}
5: Begin
6: /* Test if any user interaction occurred in the previous iteration */
7: if UserFeedback = TRUE then
8:     /* Rejected refactoring operations as constraints */
9:     Ct ← GetConstraints();
10:    /* Updated source code after applying changes */
11:    Sys ← GetRefactored − System();
12:    UserFeedback ← FALSE;
13: end if
14: S_t ← ∅, i ← 1;
15: Q_t ← Variation(P_t);
16: R_t ← P_t ∪ Q_t;
17: P_t ← evaluate(P_t, C_t, Sys);
18: (F_1, F_2, ...) ← NonDominatedSort(R_t);
19: repeat
20:    S_t ← S_t ∪ F_i;
21:    i ← i + 1
22: until (|S_t| ≥ N)
23: F_l ← F_i;                                        ▷ //Last front to be included
24: if |S_t| = N then
25:    P_{t+1} ← S_t;
26: else
27:    P_{t+1} ← ∪_{j=1}^{l−1} F_j;
28:    /*Number of points to be chosen from F_l*/
29:    K ← N − |P_{t+1}|;
30:    /*Crowding distance of points in Fl */
31:    Crowding − Distance − Assignment(F_l);
32:    Quick − Sort(F_l);
33:    /*Choose K solutions with largest distance*/
34:    P_{t+1} ← P_{t+1} ∪ Select(F_l, k);
35: end if
36: if t + 1 = Threshold then
37:    UserFeedback ← TRUE;
38:    /* Select and rank the best front */
39:    Rank − Solution(F_1);
40:    Threshold ← Threshold + t + 1;
41: end if
42: End
```

component, the developer may accept or reject or modify the refactoring recommendations. Finally, the last dynamic component uses the interaction data with the user to reduce the search space of possible refactoring solutions and improve the future suggestions by repairing the Pareto front as detailed later.

3.2.2.2 Solution representation

A solution consists of a sequence of n refactoring operations involving one or multiple source code elements of the system to refactor. The vector-based representation

is used to define the refactoring sequence. Each vector's dimension has a refactoring operation and its index in the vector indicates the order in which it will be applied. For every refactoring, pre- and post-conditions are specified to ensure the feasibility of the operation.

The initial population is generated by randomly assigning a sequence of refactorings to a randomly chosen set of code elements, or actors. The type of actor usually depends on the type of the refactoring it is assigned to and also depends on its role in the refactoring operation. An actor can be a package, class, field, method, parameter, statement or variable. Table 2.1 depicts, for each refactoring, its involved actors and its corresponding parameters.

The size of a solution, i.e. the vector's length is randomly chosen between upper and lower bound values. The determination of these two bounds is similar to the problem of bloat control in genetic programming where the goal is to identify the tree size limits. Since the number of required refactorings depends mainly on the size of the target system, we performed, for each target project, several trial and error experiments using the HyperVolume (HV) performance indicator [45] to determine the upper bound after which, the indicator remains invariant. For the lower bound, it is arbitrarily chosen. The experiments section will specify the upper and lower bounds used in this study. Table 3.1 shows an example of a refactoring solution including three operations applied to a simplified version of a solution applied to JVacation v1.0, a Java open-source trip management and scheduling software.

3.2.2.3 Solution variation

In each search algorithm, the variation operators play the key role of moving within the search space with the aim of driving the search towards optimal solutions.

For the crossover, we use the one-point crossover operator. It starts by selecting and splitting at random two parent solutions. Then, this operator creates two child

Table 3.1: Example of a solution representation.

Operation	**Source/entity**	**Target entity**
Move Method	ctrl.booking.BookingController:: handleLodgingViewEvent (java.awt.event.ActionEvent):void	ctrl.booking.LodgingModel
Extract Class	ctrl.booking.SelectionModel:: -flightList + addFlight():void +clearFlight():void	ctrl.booking.FlightList
Move Method	ctrl.booking.BookingController:: createBookings():void	ctrl.CoreModel

solutions by putting, for the first child, the first part of the first parent with the second part of the second parent, and vice versa for the second child. This operator must ensure the respect of the length limits by eliminating randomly some refactoring operations. It is important to note that in multi-objective optimization, it is better to create children that are close to their parents in order to have a more efficient search process.

For mutation, we use the bit-string mutation operator that picks probabilistically one or more refactoring operations from the solutions and replace or modify or delete them. While the crossover operator does not introduce or modify a refactoring of a solution but just the sequence (a swap between refactoring of different solutions), the mutation operator definitely can add or modify or delete a refactoring when applied to any solution of the population. When a mutation operator is applied, the goal is to slightly change the solution for the purpose to probably improve its fitness functions. We used these three operations for the mutation operator that are randomly selected when a mutation is applied to a solution. Thus, the mutation operator can introduce new refactorings by either adding completely new ones or modifying the controlling parameters of an existing refactoring. For example, move method (m1, A, B) could be replaced by $movemethod(m1, A, C)$ or $movemethod(m3, A, B)$ where $m1$, A and B are the controlling parameters of the refactoring move method. Furthermore, the

selection operator allows to regenerate part of the population randomly at every iteration thus new refactoring will be introduced since new solutions are generated during the execution process.

When applying the change operators, the different pre- and post-conditions are checked to ensure the applicability of the newly generated solutions. For example, to apply the refactoring operation *movemethod* a number of necessary pre-conditions should be satisfied such as the method and the source and target classes should exists. A post-condition example is to check that the method exists and was moved to the target class and did not exist anymore in the source class. More details about the adapted pre- and post-conditions for refactorings can be found in [9]. We also apply a repair operator that randomly selects new refactorings to replace those creating conflicts.

3.2.2.4 Solution evaluation

The generated solutions are evaluated using two fitness functions as detailed in the following paragraphs.

Minimize the number of code changes as an objective: The application of a specific suggested refactoring sequence may require an effort that is comparable to that of re-implementing part of the system from scratch. Taking this observation into account, it is essential to minimize the number of suggested operations in the refactoring solution since the designer may have some preferences regarding the percentage of deviation with the initial program design. In addition, most developers prefer solutions that minimize the number of changes applied to their design. Thus, we formally defined the fitness function as the number of modified actors/code elements (packages, classes, methods, attributes) by the generated refactorings solution.

$$f(x) = \sum_{i-1}^{n} \#code_elements(R_i, x) \tag{3.1}$$

where x is the solution to evaluate, n is the number of refactorings in the solution x and $\#code_elements$ is a function that counts the number of modified code elements in a refactoring. Any solution with refactorings being performed on the same code elements will have better (lower) fitness value for this objective. Such a definition of the objective is in favor of code locality since it encourages refactoring the same code fragment, as developers prefer to refactor the specific elements with which they are most familiar [123] instead of applying scattered changes throughout the whole system. The proposed fitness function is different from that employed in our previous work [9] where only the number of applied refactorings are counted. In fact, each refactoring type may have a different impact on the system in terms of number of code changes it engenders, something that can be identified using our new formulation.

Maximize software quality as an objective:

We used Quality Metrics for Object Oriented Designs (QMOOD) and its quality attributes level to define our objective functions. This model and its defenitions are described in Subsection 2.2.3 and Tables 2.2 and 2.3. Therefore, the objective functions are defined as:

$$Quality = \frac{\sum_{i=1}^{6} QA_i(S)}{6} \tag{3.2}$$

3.2.3 Interactive Recommendation of Refactorings

The first step of the interactive component is executed as described in Algorithm 2, to investigate if there are some common principles among the generated non-dominated refactoring solutions.

The algorithm checks if the optimal refactoring solutions have some common features such as identical refactoring operations among most or all of the solutions, and a specific common order/sequence in which to apply the refactorings. Such information will be used to rank the suggested refactorings for developers using the following

Algorithm 2 Rank Refactoring Operation procedure

```
1: Input
2: NS: Non-dominated SolutionSet of the first front
3: Output
4: HM: HashMap of refactorings along with their occurrences.
5: Begin
6: HM ← ∅;
7: /* Compute the number of occurrence of each refactoring operation*/
8: for i = 1 to |NS| do
9:     for each j = 1 to |NS_i| do
10:        /* If a refactoring operation does not exist in the list, add its hash and set its occurrence number to 1*/
11:        if (NS_{i,j} ∉ HM) then
12:            HM ← HM ∪ Hash(NS_{i,j});
13:            HM[Hash(NS_{i,j})] ← 1;
14:            /* If a refactoring operation exists in the list, increment its occurrence number */
15:        else
16:            HM[Hash(NS_{i,j})] ← HM[Hash(NS_{i,j})] + 1;
17:        end if
18:    end for
19: end for
20: End
```

formula:

$$Rank(R_{x,y}) = \frac{\sum_{j=0}^{n} \sum_{i=0}^{size(S_j)} [R_{i,j} = R_{x,y}]}{MAX(\sum_{j=0}^{n} \sum_{i=0}^{size(S_j)} [R_{i,j} = R_{x,y}])} \in [0...1] \quad (3.3)$$

where $R_{x,y}$ is the refactoring operation number x (index in the solution vector) of solution number y, and n is the number of solutions in the front. S_i is the solution of index i. All the solutions of the Pareto front are ranked based on the score of this measure applied to every solution.

Once the Pareto front solutions are ranked, the second step of the interactive step is executed as described in Algorithm 3. The refactorings of the best solution, in terms of ranking, are recommended to the developer based on their order in the vector. Then, the ranking score of the solutions is updated automatically after every feedback (interaction) with the developer. Our interactive algorithm proposes three levels of interaction as described in Fig. 3.2 and Algorithm 3.

The developer can check the ranked list of refactorings and then *apply*, *modify* or *reject* the refactoring. If the developer prefers to modify the refactoring, then our

Algorithm 3 GUF (Get User Feedback) procedure to manage the interactions with the developer (Online Phase)

1: Input
2: *RNS*: Ranked Non-dominated SolutionSet
3: Output
4: *HM*: HashMap of refactorings along with their occurrences.
5: **Begin**
6: $AppliedRefactorings \leftarrow \emptyset$;
7: $RejectedRefactorings \leftarrow \emptyset$;
8: **for** $i = 1$ **to** $|RNS|$ **do**
9: $\quad ref[i] \leftarrow 0$;
10: **end for**
11: /* Main loop to suggest refactorings one by one to the user*/
12: **while** $|RejectedRefactorings < a$ **do**
13: $\quad$ /* Select index of the the solution with highest rank*/
14: $\quad index \leftarrow MaxRank(RNS)$;
15: $\quad d \leftarrow UserDecision(RNS_{index,ref[index]})$;
16: $\quad$ /* If the user has applied or modified the operation*/
17: $\quad$ **if** $(d = True)$ **then**
18: $\qquad AppliedRefactorings \leftarrow AppliedRefactorings \cup RNS_{index,ref[index]}$;
19: $\qquad$ /* If the user has rejected the operation*/
20: $\quad$ **else**
21: $\qquad RejectedRefactorings \leftarrow RejectedRefactorings \cup RNS_{index,ref[index]}$;
22: $\quad$ **end if**
23: $\quad Ref[index] \leftarrow ref[index] + 1$;
24: $\quad$ /* Update solutions indexes */
25: $\quad$ **for** $i = 1$ **to** $|RNS|$ **do**
26: $\qquad UpdateRank(RNS_i; AppliedRefactorings, RejectedRefactorings)$
27: $\quad$ **end for**
28: **end while**
29: **End**

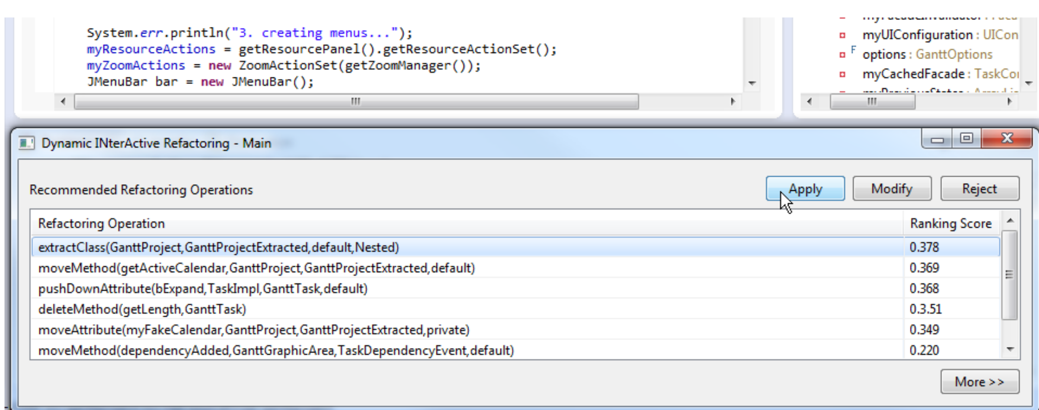


Figure 3.2: Refactorings recommended by our technique.

algorithm can help them during the modification process as described in Fig. 3.3.

In fact, our tool proposes to the developer a set of recommendations to modify the refactoring based on the history of changes applied in the past and the semantic similarity between code elements (classes, methods, etc.). For example, if the developer wants to modify a move method refactoring then, having specified the source

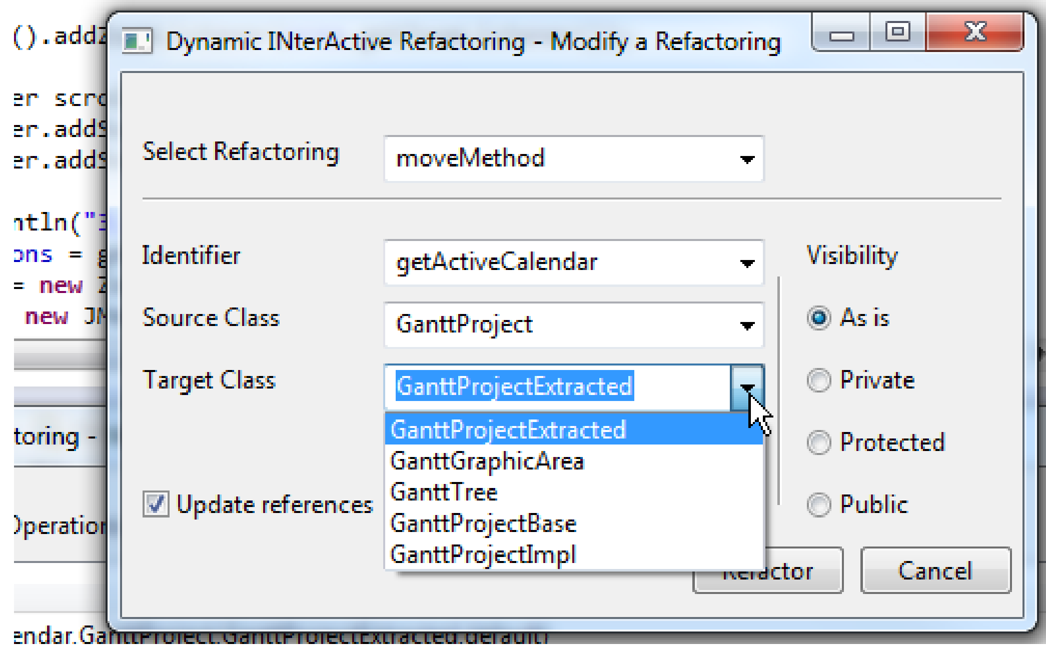


Figure 3.3: Recommended target classes by our technique for a move method refactoring to modify.

method to move, our interactive algorithm automatically suggests a list of possible target classes ranked based on the history of changes and semantic similarity. This is an interesting feature since developers often know which method to move, but find it hard to determine a suitable target class [75]. The same observation is valid for the remaining refactoring types. Another action that the developers can select is to reject/delete a refactoring from the list.

After every action selected by the developer, the ranking is updated based on the feedback using the following formula:

$$
\begin{aligned}
Rank(S_i) = & \sum_{k=1}^{size(S_i)} Rank(R_{k,i}) \\
& + (RO \cap AppliedRefactoringsList) \\
& - (RO \cap RejectedRefactoringsList) \\
& + 0.5 * (RO \cap ModifiedRefactoringsList)
\end{aligned} \tag{3.4}
$$

where S_i is the solution to be ranked, the first component consists of the sum of the ranks of its operations as explained previously and the second component will take the value of 1 if the recommended refactoring operation was applied by the developer, or -1 if the refactoring operation was rejected or 0.5 if it was partially modified by the developer. The recommended refactorings will be adjusted based on the updated ranking score.

It is important to note that we calculate the ranking score for each non-dominated solution using the innovization component and then the solution with the highest score is presented refactoring by refactoring to the developer. In fact, refactorings tend to be dependent on one another thus it is important to ensure the coherence of the recommended solution. After a number of modified or rejected refactorings or several new code changes introduced, the generated Pareto front of refactoring solutions by NSGA-II needs to be updated since the system was modified in different locations.

To check the applicability of the refactorings, we continuously check the pre-conditions of individual refactorings on the version after manual edits. Thus, the ranking of the solutions will change after every interaction. If many refactorings are rejected, the NSGA-II algorithm will continue to execute while taking into consideration all the feedback from developers as constraints to satisfy during the search. The rejected refactorings should not be considered as part of the newly generated solutions and the new system after refactoring will be considered in the input of the next iteration of the NSGA-II.

In the non-interactive refactoring systems, the set of refactorings, suggested by the best-chosen solution, needs to be fully executed in order to reach the solution's promised results. Thus, any changes applied to the set of refactorings such as changing or skipping some of them could deteriorate the resulting system's quality. In this context, the goal of this work is to cope with the above-mentioned limitation by

granting to the developer's the possibility to customize the set of suggested refactorings either by accepting, modifying or rejecting them. The novelty of this work is the approach's ability to take into account the developer's interaction, in terms of introduced customization to the existing solution, by conducting a local search to locate a new solution in the Pareto Front that is nearest to the newly introduced changes. We believe that our approach may narrow the gap that exists between automated refactoring techniques and human intensive development. It allows the developer to select the refactorings that best matches his/her coding preferences while modifying the source code to update existing features.

3.2.4 Running Example: Illustration on the JVacation System

3.2.4.1 Context

To illustrate our interactive algorithm, we consider the refactoring of JVacation *v1.0* [1] , a Java open-source trip management and scheduling software. We asked a developer to update an existing feature by adding one more field (*Premium member ID*) in the personal information form that a user has to fill out when booking a flight.

As JVacation architecture is based on the Model/View/Controller model, adding this extra field would trigger small updates on the View by adding a textbox in the personal information input form. Also the controller that handles the booking process needs to be revised. At the model level, an attribute needs to be added to the class that hosts the booking information. Finally, an update on the database level is needed to save the newly modified booking objects.

To simplify the illustration, we have limited the update to these above-mentioned changes knowing that, in order to completely implement this function, several other updates may be needed in other views and controllers in order to show, for example, the newly added field, as part of the information related to the passengers' records

[1] https://sourceforge.net/projects/jvacation/

for a given flight. We asked the developer to refactor the software system while performing the given task, therefore, the developer has initially launched the plugin that triggered our interactive algorithms. We assisted the developer in only selecting the initial default parameters for the optimization algorithm (such as the minimum and maximum chromosome lengths).

3.2.4.2 Illustration of the Innovization Component

After generating the upfront list of best refactoring solutions, three solutions are selected from the Pareto front that were involved in the interactive session to simplify this running example. Each solution has a fitness score composed of the median of quality improvement calculated based on the structural measures of the refactored system for each solution, and the number of operations within each solution. The previous section describes, these fitness values, for each solution, in terms of quality improvement and refactoring effort compared to the original system values before refactoring. These information is shown in Table 3.2.

One of the classic challenges in multi-objective optimization is the choice of the most suitable solution for the developer. The straightforward solution for this problem would be to manually investigate all solutions, i. e., execute all refactoring operations for each solution and allow the developer to compare between several refactored designs. This task can easily become tedious due to the large number of solutions in the Pareto front.

To facilitate the selection task, decision making support tools can be used to automate the selection of solutions based on the decision maker's preferences. In our context, these preferences can be considered as the packages and classes that the developer is interested in when implementing the requested feature. Thus, another straightforward heuristic would be to automatically shortlist solutions that only refactor entities that are of interest to developers. Unfortunately, this will not necessarily

Table 3.2: Quality attributes value on the JVacation system.

Quality Attribute	Original System	Solution 1	Solution 2	Solution 3
Reusability		(+0.5)	(+0.4)	(+0.5)
	1.74225	1.79225	1.79225	1.79225
Flexibility		(+0.001)	(+0.001)	(+0.001)
	1.82	1.820	1.820	1.820
Understandability		(+0.08)	(+0.07)	(+0.087)
	-4.5408	-4.5398	-4.5398	-4.5398
Functionality		(+0.5)	(+0.6)	(+0.5)
	1.16314	1.21314	1.21314	1.21314
Extendibility		(+0.007)	(+0.012)	(+0.011)
	19.7225	19.7295	19.7300	19.7299
Effectiveness	9.5406	9.5406	9.5406	9.5406
Quality Gain	-	0.198	0.202	0.209
Number of operations	-	11	14	19

reduce drastically the number of preferred solutions especially if the system is small.

To cope with this issue, another interesting idea would be to calculate the overlap between solutions. Still, choosing the most appropriate solution can be challenging as the developer has to manually break the tie between solutions by comparing between their specific refactorings. This comparison may not be straightforward because specific refactorings between to candidate solutions may both be of an interest to the developer, for example, when comparing between solution 1 and solution 2, both solutions contain a move-method operation that agree on moving a function called *getSaluation()* but disagree on the target class.

Since this function belongs to the booking panel, the participating entities are of an interest to the developer, so no choice can be automatically done based on the developer's preferred entities. Moreover, both target classes (respectively *LabelSpinner* and *LabelEdit*), each proposed by one solution, belong to the same package (*gui.components*) and they are semantically close, so the fitness function values can-

not be used to break the tie. In this scenario, only the developer would be qualified to take the decision of either accepting one operation over the other or maybe rejecting both operations. Thus, simply filtering solutions based on the developer's preferred entities may fall short in this kind of scenarios. Furthermore, asking the developer to exhaustively break the tie between shortlisted solutions can become tedious.

In this context, our interactive process differs from simply "filtering" operations based on a given preference as it "learns" from the developer's decision making and dynamically break the tie between Pareto-equivalent solutions by upgrading those with the highest number of successful recommendations (applied refactorings) while penalizing those who contain rejected operations. To illustrate this process, Table 3.3 describes each solution's refactorings along with its rank after the execution of the first step of the interactive algorithm. For the purpose of simplicity, we considered a first fragment of each solution. The solutions are ranked based on Equation 3.3 to identify the most common refactorings between the non-dominated solutions. This is achieved by counting the number of occurrences of operation within the Pareto front solution set, this number will be averaged by the maximum number of occurrences found.

3.2.4.3 Illustration of the Interactive and Dynamic Components

In the interaction part, the recommended refactoring wanted to move a function that defines the trip's starting date to a *LabelCombo* class. The developer thought that moving it to *DateEdit* class makes more sense instead because the return value of the moved function is of type Date and *DateEdit* is semantically closer to the method. So the refactroings were partially modified by the developer and the ranking score of the second solution was increased by 0.5 for *Solution 2* but by 1 for *Solution 3* since it has already a move method operation that suggests moving the same method to the chosen class by the developer, i. e., the applied operation exists in that solution.

Table 3.3: Three simplified refactoring solutions recommended for JVacation v1.0.

Operation	Source entity	Target entity
Solution 1 fitness scores before normalization (0.198, 4)		
Move Method	ctrl.booking.BookingController:: handleLodgingViewEvent(java.awt.event.ActionEvent):void	ctrl.booking.LodgingModel
Extract Class	ctrl.booking.SelectionModel:: - flightList + addFlight():void +clearFlight():void	ctrl.booking.FlightList
Move Method	ctrl.booking.BookingController:: createBookings():void	ctrl.CoreModel
Move Method	gui.panels.booking.bTravelersPanel:: getSalutation():java.lang.String	gui.components.LabelSpinner
Solution 1 Rank		3.960
Solution 2 fitness scores before normalization (0.202, 5)		
Move Method	ctrl.booking.BookingController:: handleLodgingViewEvent(java.awt.event.ActionEvent):void	ctrl.booking.lodgingList
Move Method	gui.panels.maintenance.mLodgingsPanel ::getStart():java.util.Date	gui.components.LabelCombo
Inline Class	ctrl.ModelChangeEvent	ctrl.CoreModel
Extract Class	ctrl.booking.SelectionModel:: - travelerList + addTraveler():void +clearTraveler():void	ctrl.booking.TravelerList
Move Method	gui.panels.booking.bTravelersPanel:: getSalutation():java.lang.String	gui.components.LabelSpinner
Solution 2 Rank		4.064
Solution 3 fitness scores before normalization (0.209, 6)		
Move Method	ctrl.booking.BookingController:: handleLodgingViewEvent(java.awt.event.ActionEvent):void	ctrl.booking.lodgingList
Move Method	gui.panels.maintenance.mLodgingsPanel ::getStart():java.util.Date	gui.components.DateEdit
Extract Class	ctrl.booking.SelectionModel:: - flightList + addFlight():void +clearFlight():void	ctrl.booking.FlightList
Extract Class	ctrl.booking.SelectionModel:: - travelerList + addTraveler():void +clearTraveler():void	ctrl.booking.TravelerList
Inline Class	ctrl.ModelChangeEvent	ctrl.CoreModel
Move Class	Db.factory.DBObjectFactory	db
Solution 3 Rank		3.471

In the third interaction, the recommended refactoring suggests merging two classes *CoreModel* and *ModelChangeEvent*. The first class gathers, for a given customer, all his/her bookings and sums up the total price, since the price may be later on reduced based on the customer's premium number (field to be added) the developer decided to keep the class intact and thus the operation was rejected and so the score of the top *Solution 2* was decreased by 1. The solution with the highest rank is selected for execution and its related operations are shown to the user based on their order in the vector. Table 3.4 summarizes the various interactions between the developer and the suggested refactorings from the three above mentioned solutions when adding the

Table 3.4: Four different interaction examples with the developer applied on the refactoring solutions recommended for JVacation v1.0.

Operation	R1:MoveMethod(ctrl.booking.BookingController::handleLodgingViewEvent:void, ctrl.booking.LodgingList)		
Decision	Applied		
Changes	AppliedRefactoringsList={R1}, RejectedRefactoringsList={}		
SolutionSet	Solution1	Solution2 *	Solution3
Initial rank	3.960	4.064	3.471
Interation1	3.960	5.064 (+1)	4.471 (+1)

Operation	R2:MoveMethod(gui.panels.maintenance.mLodgingsPanel::getStart():java.util.Date, gui.components.LabelCombo)		
Decision	Modified to: R2: MoveMethod(gui.panels.maintenance.mLodgingsPanel::getStart(): java.util.Date,gui.components.DateEdit)		
Changes	AppliedRefactoringsList={R1,R2}, RejectedRefactoringsList={}		
SolutionSet	Solution1	Solution2 *	Solution3
Initial rank	3.960	4.064	3.471
Interation1	3.960	5.064 (+1)	4.471 (+1)
Interation2	3.960	5.564 (+0.5)	5.471 (+1)

Operation	R3:InlineClass(ctrl.ModelChangeEvent,ctrl.CoreModel)		
Decision	Rejected		
Changes	AppliedRefactoringsList={R1,R2}, RejectedRefactoringsList={R3}		
SolutionSet	Solution1	Solution2 *	Solution3
Initial rank	3.960	4.064	3.471
Interation1	3.960	5.064 (+1)	4.471 (+1)
Interation2	3.960	5.564 (+0.5)	5.471 (+1)
Interation3	3.960	4.564 (-1)	5.471

Operation	R4:ExtractClass(ctrl.booking.SelectionModel:-flightList+addFlight():void+clearFlight():void,ctrl.booking.FlightList)		
Decision	Applied		
Changes	AppliedRefactoringsList={R1,R2,R4}, RejectedRefactoringsList={R3}		
SolutionSet	Solution1	Solution2	Solution3 *
Initial rank	3.960	4.064	3.471
Interation1	3.960	5.064 (+1)	4.471 (+1)
Interation2	3.960	5.564 (+0.5)	5.471 (+1)
Interation3	3.960	4.564 (-1)	5.471
Interation4	4.960 (+1)	4.564	6.471 (+1)

new feature.

The first recommended refactoring of the top ranked solution (Solution 2) suggests moving an event function from the controller class of the booking process, since the developer is required to investigate this class and since this function is not called during the booking procedure, moving it out of the class will reduce the number of

investigated functions, so the operation was applied by the developer and accordingly the ranking score was increased by 1 for both *Solutions 2* and *3* since they include this refactoring in their solutions.

Upon the rejection of the third suggested refactoring, the ranking score of *solution 3* has become higher than the one of *solution 2*, this has triggered the fourth recommended operation to be issued from *solution 3* instead. All the refactorings that belong to the intersection between *solution 3* and the lists of applied/rejected refactorings will be skipped during the recommendation process.

For instance, the first and second operation of *solution 3* will be skipped as they have been already applied by the developer, and the third operation will be suggested during the fourth interaction. This operation suggests the extraction of a class from the selection mode of the booking process. Since this refactoring will facilitate the distinction between functions related to the flight from those related to the passengers, the developer has approved the operation. The algorithm will stop recommending new refactorings either on the request of the developer or when the system achieves acceptable quality improvement in terms of reducing the number of design defects and improving quality metrics. These parameters can be specified by the developer or the team manager.

3.3 Evaluation

To evaluate the ability of our refactoring framework to generate good refactoring recommendations, we conducted a set of experiments based on eight open source systems and two industrial projects provided by the IT department at the Ford Motor Company. 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 [129].

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

3.3.1 Research Questions

We defined three categories of research questions to measure the correctness, relevance and benefits of our interactive multi-objective refactoring approach comparing to the state of the art based on several practical scenarios. It is important to evaluate, first, the correctness of the recommended refactoring. Since it is not sufficient to make correct refactoring recommendations, we evaluated the benefits of applying the recommended refactorings in terms of fixing code smells and improving quality attributes. Programmers are not interested, in practice, to apply *all* the correct and useful recommended refactorings due to limited resources thus we evaluated both the relevance of our recommendations and our ranking efficiency from programmers perspective based on several real-world scenarios including productivity and post-study questionnaires. We considered various existing refactoring approaches as a baseline for this proposed interactive refactoring technology to define an accurate estimation of possible improvements.

The research questions are as follows:

RQ1: Correctness, Relevance and Comparison with State of the Art.

- **RQ1-a: Correctness.** To what extent the results of our approach are similar to the ones proposed by developers compared to fully-automated refactoring techniques?
- **RQ1-b: Benefits–antipatterns correction.** To what extent code smells can be fixed using our approach compared to fully-automated refactoring techniques?
- **RQ1-c: Benefits–improving quality.** To what extent can our approach

improve the overall quality of software systems compared to fully-automated refactoring techniques?

- **RQ1-d: Relevance to programmers.** To what extent can our approach make meaningful recommendations compared to fully-automated refactoring techniques?

RQ2: Interaction Relevance. To what extent can our approach **efficiently rank** the recommended refactorings?

RQ3: Impact based on Practical Scenarios.

- **RQ3-a:** To what extent our approach can improve the **productivity of programmers when fixing bugs** compared to fully-automated refactoring techniques?

- **RQ3-b:** To what extent our approach can improve the **productivity of programmers when adding new features** compared to fully-automated refactoring techniques?

- **RQ3-c:** How do programmers evaluate the **usefulness of our approach (questionnaire)**?

3.3.2 Validation Methodology

To answer the research questions described in the previous section, we give, first, an overview about the adopted validation methodology that include the following tasks:

- **Task-1:** Generate data for baseline methods by using other existing state-of-the-art automated refactoring tools and methods offline. (RQ1a-d)

- **Task-2:** Manually refactor a system. (RQ1a)

- **Task-3:** Use our tool (DINAR) to collect final set of recommendations. (RQ1a-d, RQ2)
- **Task-4:** Rate solutions and recommendations of different methods and tools. (RQ1d, RQ2)
- **Task-5:** Code smells detection after refactoring. (RQ1b)
- **Task-6:** Measure quality metrics after refactoring. (RQ1c)
- **Task-7:** Fix bugs on refactored / unrefactored systems. (RQ3a)
- **Task-8:** Implement features on refactored / unrefactored systems.(RQ3b)
- **Task-9:** Post-study questionnaire. (RQ3c)

For each task, we defined and used different evaluation metrics (Precision, Recall, number of fixed antipatterns, the quality gain, manual correctness, number of modified/rejected/accepted recommendations and execution time) which are described in this section. These metrics are calculated and compared for different refactoring techniques which are applied on a variety of software projects under the specific above scenarios. Table 3.5 shows the summary of the connections between the research questions, metrics and tasks detailed in this section.

In order to have a consistent comparison, we considered the refactoring solutions recommended by our approach after all interactions with the developers (last set of solutions). Therefore, we refer to these sets of refactoring solutions as "our approach results" afterward. To create a baseline, we asked the participants in our study to analyze and apply manually several refactoring types using Eclipse IDE on several code fragments extracted from different systems where most of them correspond to code smells identified in previous studies as worth removing by refactoring [19, 54]. This golden set is defined based on the following two main criteria: 1. Refactorings

Table 3.5: Summary of the research questions, their goals, defined metrics to answer and analyze them, and the associated tasks to collect data and calculate the metrics.

RQ#	RQ Goal	Sub-RQ	Sub-Goal	Metric(s)	Task(s)#
RQ1	Relevant Solutions	RQ1-a	Similarity	RC, PR	1, 2, 3
		RQ1-b	Fixing code smells	NF	1,3,5
		RQ1-c	Overall quality	G	1,3,6
		RQ1-d	Meaningful recommendation	MC	1,3,4
RQ2	Efficient ranking	-		NAR, NRR, NMR, PR@k, MC@k	3, 4
RQ3	Usefulness	RQ3-a	Productivity / fixing bugs	TP	7
		RQ3-b	Productivity /adding features		8
		RQ3-c	questionnaire		9

that fix a design flaw and did not change the behavior or introduce bugs, 2. Refactorings that improve a set of quality metrics (based on the QMOOD model) and did not change the behavior or introduce bugs. We refer to these refactoring solutions as "expected refactorings" afterward.

To answer RQ1, it is important to validate the proposed refactoring solutions from both quantitative and qualitative perspectives. For RQ1-a, we calculated precision and recall scores to compare between refactorings recommended by each approach and those expected based on the participants opinion:

$$RC_{recall} = \frac{Approach\ Solution \cap Expected\ Refactorings}{Expected\ Refactorings} \in [0, 1] \tag{3.5}$$

$$PR_{precision} = \frac{Approach\ Solution \cap Expected\ Refactorings}{Approach\ Solution} \in [0, 1] \tag{3.6}$$

When calculating the precision and recall, we consider a refactoring as a correct recommendation if all the controlling parameters are the same like the expected ones.

For RQ1-b, we considered another quantitative evaluation which is 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 [1].

Formally, NF is defined as:

$$NF = \frac{\#fixed\ code\ smells}{\#code\ smells} \in [0, 1] \tag{3.7}$$

The detection of code smells is very 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 for RQ1-c another metric, G, 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. The average of the six QMOOD attributes were used: reusability, flexibility, understandability, Extendibility, Functionality and effectiveness. All of them are formalized using a set of quality metrics. Hence, the gain for each of the considered QMOOD quality attributes and the average total gain in quality after refactoring can be easily estimated as:

$$G = \frac{\sum_{i=1}^{6} G_{q_i}}{6}\ and\ G_{q_i} = q'_i - q_i \tag{3.8}$$

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

For RQ1-d, we asked the participant in our study, as detailed in Section 4.4, to evaluate, manually, whether the suggested refactorings are feasible and efficient at improving the software quality and achieving their maintainability objectives. We define the metric Manual Correctness (MC) to mean the number of meaningful refactorings divided by the total number of recommended refactorings. The meaningful refactorings are recognized by taking the majority of votes from the developers. This procedure is analogous to the real-world situations based on our own experience with our industrial partners. Therefore, MC is given by the following equation:

$$MC = \frac{\#\ Meaningful\ Refactorings}{\#\ Recommended\ Refactorings} \tag{3.9}$$

To avoid the computation of the MC metric being biased by the developer's feedback, we asked the developers to manually evaluate the correctness of the recommended refactorings of our approach on the systems that they did not refactor using our tool. Therefore, The developers did not evaluate the results of their own results of interactive refactoring but the resultant refactorings recommended on other systems where other developers apllied our approach. The main motivation for the "manual correctness" metric is actually to address the concern that the deviation with the expected refactorings could be just because of the preferences of the developers. The manual correctness metric is evaluated manually on each refactoring one-by-one to check their validity. Thus, we evaluated the results produced by the different tools and we were not limited to the comparison with the expected results. We did the comparison with the expected results to provide an automated way to evaluate the results and avoid the developers being biased by the results of our tool (developers did not know anything about the refactorings suggested by the different tools when they provided their recommendations).

We used the metrics *MC, RC, PR, NF* and G to perform the comparisons and answer respectively RQ1a-d.

We considered some other useful metrics to answer RQ2 that count the percentage of refactorings that were accepted (NAR) or rejected (NRR) or applied with some modifications (NMR). Formally, these metrics are defined as:

$$NAR = \frac{\#\ Accepted\ Refactorings}{\#\ Recommended\ Refactorings} \in [0,1] \tag{3.10}$$

$$NRR = \frac{\#\ Rejected\ Refactorings}{\#\ Recommended\ Refactorings} \in [0,1] \tag{3.11}$$

$$NMR = \frac{\#\ Modified\ Refactorings}{\#\ Recommended\ Refactorings} \in [0,1] \tag{3.12}$$

To answer RQ2, we also evaluated the relevance of the recommended refactorings in the top k where k =*1, 5, 10* and *15* using the following metrics *PR@k* and *MC@k*. We used the same equations defined for RQ1 with the only difference that the considered suggested refactorings are exclusively those located in the top k positions of the ranked list of refactorings at multiple instances after the execution of the innovization component.

To answer RQ3, we aimed to assess how the refactoring actually increases the software quality and productivity in that the effort to fixing bugs (R3-a) or adding new features (R3-b) should reduce after performing the refactorings. We asked the software developers participated in this study to add new features and fix a set of bugs. To avoid that the achieved results might be due to the different levels of ability of the developers groups, we adapted a counter-balanced design where each participant performed two tasks, one on the original system and one on the refactored system. The details of these scenarios will be described later as detailed in Section 4.6. To estimate the impact of the suggested refactorings on the productivity of developers, we defined the following metric TP to measure the time required to perform the same activities on the system with and without refactoring:

$$TP_i = 1 - \frac{\#minutes\ required\ to\ perform\ task\ i\ on\ the\ system\ after\ refactoring}{\#minutes\ required\ to\ perform\ task\ i\ on\ the\ system\ befor\ refactoring} \quad (3.13)$$

We have also compared the productivity results of our approach compared to Kessentini *et al.* [1], Ouni *et al.* [2] and Harman *et al.* [3] to test the hypothesis if better quality of the software may increase the productivity of developers. To answer RQ3-b, we used a post-study questionnaire that collects the opinions of developers on our tool as detailed in the next section.

3.3.3 Studied Software Projects

We used a set of well-known open-source Java projects and two systems from our industrial partner, the Ford Motor Company. We applied our approach to eight open-source Java projects: Xerces-J, JHotDraw, JFreeChart, GanttProject, Apache Ant, Rhino and Log4J and Nutch. Xerces-J is a family of software packages for parsing XML. JFreeChart is a free tool for generating charts. Apache Ant is a build tool and library specifically conceived for Java applications. Rhino is a JavaScript interpreter and compiler written in Java and developed for the Mozilla/Firefox browser. GanttProject is a cross-platform tool for project scheduling. Log4J is a popular logging package for Java. Nutch is an Apache project for web crawling. JHotDraw is a GUI framework for drawing editors.

In order to get feedback from the original developers of a system, we considered in our experiments two large industrial projects provided by our industrial partner, the Ford Motor Company. The first project is a marketing return on investment tool, called MROI, used by the marketing department of Ford to predict the sales of cars based on the demand, dealers' information, advertisements, etc. The tool can collect, analyze and synthesize a variety of data types and sources related to customers and dealers. It was implemented over a period of more than eight years and frequently changed to include and remove new/redundant features.

The second project is a Java-based software system, JDI, which helps the Ford Motor Company to create the best schedule of orders from the dealers based on thousands of business constraints. This system is also used by Ford Motor Company to improve their vehicles sales by selecting the right vehicle configuration to match the expectations of their customers. JDI is highly structured and software developers have developed several versions of it at Ford over the past 10 years. Due to the importance of the application and the high number of updates performed on both systems, it is critical to ensure that they remain of high quality so to reduce the time

Table 3.6: Statistics of the studied software projects.

System	Release	#classes	KLOC	#Code smells	#Applicable Refactorings
Xerces-J	v2.7.0	991	240	61	80
JHotDraw	v6.1	585	21	22	36
JFreeChart	v1.0.9	521	170	51	96
GanttProject	v1.10.2	245	41	60	63
Apache Ant	v1.8.2	1191	255	61	74
Rhino	v1.7R1	305	42	79	50
Log4J	v1.2.1	189	31	27	41
Nutch	v1.1	207	39	39	24
JDI-Ford	v5.8	638	247	83	94
MROI-Ford	V6.4	786	264	97	119

required by developers to introduce new features in the future.

We selected these 10 systems for our validation because they range from medium to large-sized open-source projects, which have been actively developed over the past 10 years, and their design has not been responsible for a slowdown of their developments. Table 3.6 provides some descriptive statistics about these 10 programs.

3.3.4 Study Participants

Our study involved 14 participants from the University of Michigan and 8 software developers from the Ford Motor Company. Participants include 6 master students in Software Engineering, 8 Ph.D. students in Software Engineering and 8 software developers. All the participants are volunteers and familiar with Java development and refactoring. The experience of these participants on Java programming ranged from 2 to 19 years. We carefully selected the participants to make sure that they already applied refactorings during their previous experiences in development.

All the graduate students have already taken at least one position as software engineer in industry for at least two years as software developer and most of them (11 out of 14 students) participated in similar experiments in the past, either as part of a research project or during graduate courses on Software Quality Assurance or Software Evolution offered at the University of Michigan. Furthermore, 6 out the 14 students (the selected master students) are working as full-time or part-time

developers in the software industry.

Participants were first asked to fill out a pre-study questionnaire containing five questions. The questionnaire helped to collect background information such as their role within the company, their programming experience, and their familiarity with software refactoring. In addition, all the participants attended one lecture about software refactoring and passed six tests to evaluate their performance in evaluate and suggest refactoring solutions.

We formed 3 groups. The groups were formed based on the pre-study questionnaire and the test results to ensure that all the groups have almost the same average skill level. We divided the participants into groups according to the studied systems, the techniques to be tested and developers' experience.

Each of the first two groups (A and B) is composed of three masters students and four Ph.D. students. The third group is composed of eight software developers from the Ford Motor company, since they agreed to participate only in the evaluation of their two software systems. It is important to note that the third group formed by the developers from Ford is part of the original developers of the two evaluated systems.

3.3.5 Techniques Studied

3.3.5.1 Overview of the Used Techniques

To answer our research questions from the perspective of evaluating our interactive approach performance against the state-of-the-art refactoring techniques, we compared our approach to four other existing fully-automated search-based refactoring techniques and our multi-objective approach without the interaction component (NSGA-II-Innovization). Studied techniques includes: Kessentini *et al.* [1], O'Keeffe and O' Cinnéide [4], Ouni *et al.* [2] and Harman *et al.* [3] that consider the refactoring suggestion task only from the quality improvement perspective.

Autors in [1], formulate software refactoring as a mono-objective search problem

where the main goal is to fix design defects and improve quality metrics. Also, [4] proposed a mono-objective formulation to automate the refactoring process by optimizing a set of quality metrics. The authors in [2] and [3] proposed a multi-objective refactoring formulation that generates solutions to fix code smells. Both techniques are non-interactive and fully-automated.

We considered in our experiments another popular design defects detection and correction tool JDeodorant [5] that does not use heuristic search techniques. 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. Since JDeodorant just recommends a few types of refactoring with respect to the ones considered by our tool. We restricted, in this case, the comparison to the same refactoring types supported by JDeodorant such as *Move Method*, *Extract Method* and *Extract Class*.

Our approach differs with the above fully-automated techniques in two factors: *innovization* and *interactive features*. Therefore, it is important to evaluate the impact of every factor on the quality of our results. If the innovization makes the largest contribution, which is another fully automated search-based approach, the results cannot support the hypothesis related to the outperformance of interactive refactoring. Thus, we compared our approach to NSGA-II with the innovization feature using the same multi-objective optimization but without the use of the interactive feature.

All these existing techniques are fully-automated and do not provide any interaction with the developers to update their solutions.

Table 3.7 summarizes the survey organization including the list of systems and algorithms evaluated by the groups of participants.

Table 3.7: Survey organization.

Participants groups	Software Projects	Approaches	Tasks
Group A	Xerces-J	Interactive NSGA-II, [4], [2], JDeodorant [5], [1], [3]	-Interactive refactoring -Manual refactoring -Post-study questionnaire -Fixing bugs -Adding features
	JHotDraw		
	JFreeChart		
	GanttProject		
Group B	Apache Ant		
	Rhino		
	Log4J		
	Nutch		
Group C	JDI-Ford	Interactive NSGA-II, [4], [9], JDeodorant [5]	
	MROI-Ford		

3.3.5.2 Parameters Setting

Parameter setting influences significantly the performance of a search algorithm on a particular problem [130]. For this reason, for each algorithm and for each system, we perform a set of experiments using several population sizes: 50, 100, 200, 300 and 500. The stopping criterion was set to 100,000 evaluations for all algorithms in order to ensure fairness of comparison. The other parameters' values were fixed by trial and error and are as follows: crossover probability = 0.8; mutation probability = 0.5 where the probability of gene modification is 0.3; stopping criterion = 100,000 evaluations.

In order to have significant results, for each couple (algorithm, system), we use the trial and error method [131] in order to obtain a good parameter configuration. Trial and error is a fundamental method of problem solving. It is characterized by repeated and varied attempts of algorithm configurations.

Regarding the evaluation of fixed code smells, we focus on the following code smell types: Blob, Spaghetti Code (SC), Functional Decomposition (FD), Feature Envy (FE), Data Class (DC), Lazy Class (LC), and Shotgun Surgery (SS). We choose these code smell types in our experiments because they are the most frequent and hard to fix based on several studies [9, 48]. These design flaws are automatically detected

using the detection rules of our previous work [1] based on genetic programming. We have generated and manually validated, in [1] and several of our other previous studies, a set of metrics-based rules that can automatically detect the different types of code smells considered in our experiments. Table 3.6 reports the number of code smells for each system. Only real design flaws that were manually validated in our previous work [1] are considered in this validation.

The upper and lower bounds on the chromosome length used in this study are set to 10 and 350 respectively. Several SBSE problems including software refactoring are characterized by a varying chromosome length. This issue is similar to the problem of bloat control in genetic programming where the goal is to identify the tree size limits. To solve this problem, we performed several trial and error experiments where we assess the average performance of our algorithm using the hypervolume (HV) performance indicator while varying the size limits between 10 and 500 operations.

3.3.6 Case Studies Summary

Each group of participants received a questionnaire, a manuscript guide to help them to fill the questionnaire, the tools and results to evaluate and the source code of the studied systems as described in the following five scenarios:

In the first scenario, we selected a total of 90 classes from all the systems that include design defects (9 classes to fix per system). Then we asked every participant to manually apply refactorings to improve the quality of the systems by fixing an average of two of these defects. As an outcome of the this scenario, we have a set of "expected refactorings" and we are able to calculate the differences between the recommended refactorings and the expected ones (manually suggested by the developers).

In the second scenario, we asked the developers to evaluate the suggested solutions of our algorithm. We performed a cross-validation between the ratings of each group to avoid the computation of the MC metric being biased by the developer's feedback.

Thus, the developers in each group rated results generated by the other developers in the same group.

In the third scenario, we collected a set of 6 bugs per system from the bug reports of the studied release for every project and asked the groups to fix them based on the refactored and non-refactored version. The tasks are completely different and they are applied to different packages/classes of the same version of the systems. Furthermore, the participants did not know if they are working on the system before or after refactoring. We did not follow as well any specific order when asking the developers to work on the tasks. Only 3 out of the 22 participants worked as part of the experiments on the systems before refactoring and then the systems after refactoring. We adapted a counter-balanced design where we asked every developer to fix 2 bugs on the version before refactoring and then 2 other bugs in the version after refactoring. We selected the bugs that require almost the same effort to fix in terms of number of changes, with an average of 15 changes.

In the fourth scenario, we asked the groups to add two simple features to every system before refactoring, and then two other features on the system after refactoring. All the features require almost the same number of changes to be introduced or deleted with an average of 23 code changes per feature. In the third and fourth scenarios, the bugs to fix and features to add are related to the classes that are refactored by the developers when using our tool.

The participants were asked to justify their evaluation of the solutions and these justifications are reviewed by the organizers of the study (one faculty member, one postdoc, one Ph.D. student and one Master's student). Participants do not know the particular experiment research questions and the used algorithms.

In the fifth scenario, we asked the participants to use our tool during a period of two hours on the different systems and then we collected their opinions based on a post-study questionnaire. To better understand subjects' opinions with regard to

usefulness of our approach in a real setting, the post-study questionnaire was given to each participant after completing the refactoring tasks using our interactive approach and all the techniques considered in our experiments. The questionnaires collected the opinions of the participants about their experience in using our approach compared to manual and fully-automated refactoring tools. We asked participants to rate their agreement on a Likert scale from 1 (complete disagreement) to 5 (complete agreement) with the following statements:

1. The interactive dynamic refactoring recommendations are a desirable feature in integrated development environments (IDEs).

2. The interactive manner of recommending refactorings by our approach is a useful and flexible way to refactor systems compared to fully-automated or manual refactorings.

The remaining questions of the post-study questionnaire were about the benefits and also limitations (possible improvements) of our interactive approach.

3.3.7 Results and Discussion

3.3.7.1 Statistical Analysis

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 experimental study is based on 30 independent simulation runs for each problem instance. 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 one-way ANOVA statistical test with a 95% confidence level ($\alpha = 5\%$) to find out whether our sample results of different approaches are different significantly.

Since one-way ANOVA is an omnibus test, A statistically significant result determines whether three or more group means differ in some undisclosed way in the population.

One-way ANOVA is conducted for the results obtained from each software project to investigate and compare each performance metric (dependent variable) between various studied algorithms (independent variable - groups).
We test the null hypothesis (H_0) that population means of each metric are equal for all methods ($\forall\ Software\ Projects : \mu_{M1}^{metric} = \mu_{M2}^{metric} = ... = \mu_{M7}^{metric}\ where\ metric \in \{G, NF, MC, PR, RC\}$) against the alternative (H_1) that they are not all equal and at least one method population mean is different.

There are some assumptions for one-way ANOVA test which we assessed before applying the test on the data:

Outliers: There were no outliers in the data, as assessed by inspection of a boxplot for values greater than 1.5 box-lengths from the edge of the box.

Normal Distribution: Some of the dependent variables were not normally distributed for each method, as assessed by Shapiro-Wilk's test. However, the one-way ANOVA is fairly robust to deviation from normality. Since the sample size is more than 15 (there are 30 observations in each group) and the sample sizes are equal for all groups (balanced), non-normality is not an issue and does not affect Type I error.

Homogeneity of variances: The one-way ANOVA assumes that the population variances of the dependent variables are equal for all groups of the independent variable. If the variances are unequal, this can affect the Type I error rate. There was homogeneity of variances, as assessed by Levene's test for equality of variances ($p > 0.05$).

The results of one-way ANOVA tests for all pair of software projects and metrics indicates that The group means were statistically significantly different ($p < .0005$) and, therefore, we can reject the null hypothesis and accept the alternative hypothesis which says there is difference in population means between at least two groups. Table

Table 3.8: F-value results from one-way ANOVA statistical tests for corresponding software project and metric between different methods.

Software	**G**	**NF**	**MC**	**PR**	**RC**
ApacheAnt	335.7	224.8	803.9	379.1	757.1
GanttProject	209.6	593.0	1463.2	379.6	1130.4
JDIFord	135.6	320.3	1036.2	917.3	1032.8
JFreeChart	300.1	776.7	494.7	211.9	663.9
JHotDraw	181.7	408.2	1022.6	158.4	663.8
Log4J	297.8	306.2	477.8	617.9	1044.9
MROIFord	189.5	474.8	1260.2	1228.8	1217.2
Nutch	333.7	361.3	408.1	269.9	658.9
Rhino	121.2	606.2	872.8	598.0	702.2
XercesJ	155.0	214.5	598.0	492.3	633.8

3.8 reports the obtained value of F-statistics with the between and within groups degree of freedoms equal to 6 and 203, respectively. In one-way ANOVA, the F-statistic is the ratio of variation between sample means over variation within the samples. The larger value of F-statistics represents the group means are further apart from each other and are significantly different. Also, it shows that the observation within each group are close to the group mean with a low variance within samples. Therefore, a large F-value is required to reject the null hypothesis that the group means are equal. Our obtained F-statistics results are correspond to very small p-values.

One-way ANOVA does not report the size of the difference. Therefore, we calculated Eta squared (η^2) 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 3.9 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. For instance, it says 90% of the total variance of metric G for ApacheAnt software project is accounted for by different algorithms effect, not error or other effects.

Tukey post hoc analysis [132] is carried out in order to find out between which

Table 3.9: Effect size values (Eta squared (η^2)) for corresponding software project and metric.

Software	G	NF	MC	PR	RC
ApacheAnt	0.908	0.869	0.960	0.918	0.957
GanttProject	0.861	0.946	0.977	0.918	0.971
JDIFord	0.789	0.898	0.966	0.962	0.966
JFreeChart	0.899	0.958	0.936	0.862	0.952
JHotDraw	0.843	0.923	0.968	0.824	0.951
Log4J	0.898	0.900	0.934	0.948	0.969
MROIFord	0.839	0.929	0.972	0.971	0.971
Nutch	0.908	0.914	0.923	0.889	0.951
Rhino	0.782	0.947	0.963	0.946	0.954
XercesJ	0.821	0.864	0.946	0.936	0.949

group(s) the significant difference is occurred. Basically, it tests all possible group comparisons. However, we only report the results of comparison of our method and others in Table 3.10. This table represents the point estimate of the difference between each pair of means and is computed from the sample data. Also, it includes the confidence interval showing the difference between population means and is centered on point estimate. If This interval does not include zero, indicates that the difference between the means is statistically significant. The 95% individual confidence level indicates that we can be 95% confident that each interval contains the real difference for that particular comparison. The results shows that all pairwise comparisons between our method and others' for each pair of (software / metric) are statistically significant at the 0.05 level except for G and NF of JFreeChart as their results highlighted in the table of the results. Therefore, the difference between the means of these two metrics,G and NF, for JFreeChart project is 0.

To this end, we used the Vargha-Delaney A measure [133] which is a non-parametric effect size measure. In our context, given the different performance metrics (such as PR, RC, MC, etc.), the A statistic measures the probability that running an algorithm B1 (interactive NSGA-II) yields better performance than running another algorithm B2 (such as[1], [4], [2], etc.). If the two algorithms are equivalent, then A = 0.5. In our experiments, we have found the following results: a) On small and medium scale

Table 3.10: Tukey post hoc analysis results between our method(M1) and others reported by Mean difference and 95% confidence intervals. Label of the methods: **M1** (Our approach)=Interactive+Innovization NSGA-II, **M2**=Innovization NSGA-II, **M3**=Kessentini et al.[1], **M4**=Ouni et al.[2], **M5**=Harman et al.[3], **M6**=O'Keeffe et al.[4], **M7**=Jdeodorant [5].

		|Mean difference | 95% Confidence Interval|									
		G		**NF**		**MC**		**PR**		**RC**	
ApacheAnt	M1-M2	0.10	(0.09,0.12)	0.05	(0.04,0.06)	0.07	(0.06,0.08)	0.09	(0.07,0.10)	0.07	(0.06,0.08)
ApacheAnt	M1-M3	0.15	(0.13,0.17)	0.07	(0.06,0.09)	0.12	(0.11,0.13)	0.14	(0.12,0.15)	0.18	(0.17,0.19)
ApacheAnt	M1-M4	0.12	(0.10,0.14)	0.05	(0.04,0.07)	0.10	(0.09,0.11)	0.12	(0.10,0.13)	0.13	(0.12,0.14)
ApacheAnt	M1-M5	0.21	(0.19,0.23)	0.10	(0.09,0.11)	0.17	(0.16,0.18)	0.13	(0.11,0.14)	0.18	(0.17,0.19)
ApacheAnt	M1-M6	0.16	(0.14,0.18)	0.04	(0.03,0.05)	0.14	(0.13,0.15)	0.12	(0.10,0.13)	0.10	(0.09,0.11)
ApacheAnt	M1-M7	0.18	(0.17,0.20)	0.15	(0.14,0.17)	0.29	(0.28,0.30)	0.23	(0.21,0.24)	0.28	(0.27,0.29)
GanttProject	M1-M2	0.05	(0.03,0.07)	0.02	(0.01,0.03)	0.11	(0.10,0.12)	0.10	(0.08,0.11)	0.03	(0.02,0.04)
GanttProject	M1-M3	0.09	(0.07,0.10)	0.06	(0.05,0.07)	0.15	(0.14,0.16)	0.12	(0.10,0.13)	0.08	(0.07,0.09)
GanttProject	M1-M4	0.07	(0.06,0.09)	-0.04	(-0.05,-0.03)	0.22	(0.21,0.23)	0.12	(0.10,0.13)	0.06	(0.05,0.07)
GanttProject	M1-M5	0.15	(0.13,0.17)	0.17	(0.16,0.18)	0.30	(0.29,0.31)	0.20	(0.19,0.21)	0.29	(0.28,0.30)
GanttProject	M1-M6	0.15	(0.13,0.17)	0.14	(0.13,0.15)	0.26	(0.25,0.27)	0.16	(0.14,0.17)	0.10	(0.09,0.11)
GanttProject	M1-M7	0.12	(0.10,0.14)	0.14	(0.13,0.15)	0.33	(0.32,0.34)	0.18	(0.17,0.19)	0.22	(0.21,0.23)
JDIFord	M1-M2	0.03	(0.01,0.04)	-0.02	(-0.03,-0.01)	0.07	(0.06,0.08)	0.08	(0.07,0.09)	0.06	(0.05,0.07)
JDIFord	M1-M3	-	-	-	-	-	-	-	-	-	-
JDIFord	M1-M4	0.03	(0.01,0.04)	-0.03	(-0.04,-0.02)	0.20	(0.19,0.21)	0.13	(0.12,0.14)	0.15	(0.14,0.16)
JDIFord	M1-M5	-	-	-	-	-	-	-	-	-	-
JDIFord	M1-M6	0.07	(0.05,0.08)	0.10	(0.09,0.11)	0.20	(0.19,0.21)	0.17	(0.16,0.18)	0.06	(0.05,0.07)
JDIFord	M1-M7	0.11	(0.09,0.12)	0.08	(0.07,0.09)	0.25	(0.24,0.26)	0.25	(0.24,0.26)	0.27	(0.26,0.28)
JFreeChart	M1-M2	0.09	(0.07,0.11)	0.02	(0.00,0.03)	0.08	(0.07,0.09)	0.07	(0.06,0.08)	0.12	(0.11,0.13)
JFreeChart	M1-M3	0.12	(0.10,0.14)	0.02	(0.01,0.03)	0.14	(0.13,0.15)	0.12	(0.11,0.13)	0.16	(0.15,0.17)
JFreeChart	M1-M4	0.00	(-0.02,0.02)	0.00	(-0.01,0.01)	0.12	(0.11,0.13)	0.10	(0.09,0.11)	0.14	(0.12,0.15)
JFreeChart	M1-M5	0.14	(0.12,0.16)	0.24	(0.22,0.25)	0.14	(0.13,0.16)	0.15	(0.14,0.16)	0.28	(0.26,0.29)
JFreeChart	M1-M6	0.17	(0.15,0.19)	0.09	(0.08,0.10)	0.20	(0.19,0.22)	0.10	(0.09,0.12)	0.16	(0.15,0.17)
JFreeChart	M1-M7	0.13	(0.11,0.15)	0.15	(0.13,0.16)	0.22	(0.21,0.24)	0.12	(0.11,0.13)	0.24	(0.23,0.25)
JHotDraw	M1-M2	0.02	(0.01,0.03)	0.05	(0.04,0.07)	0.08	(0.07,0.09)	0.04	(0.03,0.05)	0.06	(0.04,0.07)
JHotDraw	M1-M3	0.06	(0.05,0.07)	0.04	(0.03,0.05)	0.16	(0.15,0.17)	0.09	(0.08,0.10)	0.10	(0.09,0.12)
JHotDraw	M1-M4	0.03	(0.02,0.04)	-0.02	(-0.03,-0.01)	0.14	(0.13,0.15)	0.07	(0.06,0.08)	0.09	(0.08,0.10)
JHotDraw	M1-M5	0.08	(0.07,0.09)	0.14	(0.13,0.15)	0.30	(0.29,0.31)	0.12	(0.11,0.13)	0.21	(0.20,0.22)
JHotDraw	M1-M6	0.04	(0.03,0.05)	0.14	(0.13,0.15)	0.24	(0.23,0.25)	0.10	(0.09,0.11)	0.17	(0.16,0.18)
JHotDraw	M1-M7	0.11	(0.10,0.12)	0.08	(0.07,0.09)	0.24	(0.23,0.25)	0.10	(0.09,0.12)	0.24	(0.23,0.25)
Log4J	M1-M2	0.08	(0.07,0.10)	0.06	(0.05,0.07)	0.08	(0.07,0.10)	0.03	(0.01,0.04)	0.06	(0.05,0.07)
Log4J	M1-M3	0.13	(0.12,0.14)	0.13	(0.12,0.14)	0.12	(0.11,0.13)	0.14	(0.12,0.15)	0.22	(0.21,0.23)
Log4J	M1-M4	0.10	(0.09,0.11)	0.06	(0.05,0.07)	0.10	(0.09,0.11)	0.05	(0.03,0.06)	0.08	(0.06,0.09)
Log4J	M1-M5	0.14	(0.13,0.15)	0.15	(0.14,0.16)	0.19	(0.18,0.20)	0.19	(0.17,0.20)	0.21	(0.20,0.22)
Log4J	M1-M6	0.19	(0.18,0.21)	0.13	(0.12,0.14)	0.16	(0.15,0.17)	0.12	(0.11,0.13)	0.19	(0.18,0.20)
Log4J	M1-M7	0.12	(0.10,0.13)	0.15	(0.14,0.16)	0.21	(0.20,0.22)	0.22	(0.21,0.23)	0.31	(0.30,0.32)
MROIFord	M1-M2	0.05	(0.04,0.07)	0.02	(0.01,0.04)	0.08	(0.07,0.09)	0.06	(0.05,0.07)	0.12	(0.11,0.13)
MROIFord	M1-M3	-	-	-	-	-	-	-	-	-	-
MROIFord	M1-M4	0.08	(0.07,0.09)	0.03	(0.02,0.04)	0.16	(0.15,0.17)	0.09	(0.08,0.10)	0.16	(0.15,0.17)
MROIFord	M1-M5	-	-	-	-	-	-	-	-	-	-
MROIFord	M1-M6	0.12	(0.10,0.13)	0.17	(0.16,0.19)	0.21	(0.20,0.22)	0.13	(0.12,0.14)	0.26	(0.25,0.27)
MROIFord	M1-M7	0.13	(0.11,0.14)	0.14	(0.13,0.15)	0.29	(0.28,0.30)	0.31	(0.30,0.32)	0.28	(0.27,0.29)
Nutch	M1-M2	0.07	(0.05,0.08)	0.06	(0.04,0.07)	0.07	(0.06,0.08)	0.04	(0.03,0.05)	0.05	(0.04,0.06)
Nutch	M1-M3	0.14	(0.12,0.16)	0.11	(0.10,0.12)	0.11	(0.10,0.12)	0.08	(0.07,0.09)	0.14	(0.13,0.15)
Nutch	M1-M4	0.10	(0.08,0.12)	0.05	(0.04,0.07)	0.09	(0.08,0.10)	0.08	(0.07,0.09)	0.05	(0.04,0.06)
Nutch	M1-M5	0.20	(0.18,0.22)	0.19	(0.18,0.20)	0.18	(0.17,0.19)	0.12	(0.11,0.13)	0.19	(0.18,0.21)
Nutch	M1-M6	0.14	(0.12,0.16)	0.15	(0.14,0.16)	0.14	(0.13,0.15)	0.06	(0.05,0.07)	0.17	(0.16,0.18)
Nutch	M1-M7	0.17	(0.15,0.19)	0.09	(0.08,0.10)	0.19	(0.18,0.20)	0.16	(0.15,0.17)	0.22	(0.21,0.23)
Rhino	M1-M2	0.06	(0.04,0.08)	0.07	(0.06,0.09)	0.05	(0.03,0.06)	0.04	(0.03,0.05)	0.09	(0.08,0.10)
Rhino	M1-M3	0.08	(0.06,0.10)	0.14	(0.13,0.15)	0.09	(0.08,0.10)	0.06	(0.05,0.07)	0.16	(0.15,0.17)
Rhino	M1-M4	0.07	(0.05,0.09)	0.12	(0.11,0.13)	0.07	(0.06,0.08)	0.05	(0.04,0.06)	0.13	(0.12,0.15)
Rhino	M1-M5	0.13	(0.11,0.15)	0.20	(0.19,0.22)	0.23	(0.21,0.24)	0.22	(0.21,0.23)	0.28	(0.27,0.29)
Rhino	M1-M6	0.08	(0.06,0.10)	0.18	(0.17,0.19)	0.14	(0.13,0.15)	0.12	(0.11,0.13)	0.15	(0.14,0.17)
Rhino	M1-M7	0.11	(0.09,0.13)	0.24	(0.23,0.26)	0.28	(0.27,0.29)	0.17	(0.16,0.18)	0.23	(0.22,0.24)
XercesJ	M1-M2	0.03	(0.02,0.04)	0.02	(0.01,0.03)	0.06	(0.05,0.07)	0.09	(0.08,0.11)	0.08	(0.07,0.09)
XercesJ	M1-M3	0.07	(0.06,0.08)	0.02	(0.01,0.04)	0.11	(0.10,0.12)	0.16	(0.15,0.17)	0.13	(0.12,0.14)
XercesJ	M1-M4	0.04	(0.03,0.05)	-0.02	(-0.03,0.00)	0.08	(0.07,0.09)	0.13	(0.12,0.15)	0.10	(0.09,0.11)
XercesJ	M1-M5	0.12	(0.11,0.13)	0.12	(0.11,0.13)	0.20	(0.19,0.21)	0.19	(0.18,0.21)	0.22	(0.21,0.23)
XercesJ	M1-M6	0.08	(0.07,0.09)	0.08	(0.07,0.10)	0.23	(0.21,0.24)	0.16	(0.15,0.17)	0.20	(0.19,0.21)
XercesJ	M1-M7	0.09	(0.08,0.10)	0.06	(0.05,0.08)	0.17	(0.16,0.18)	0.23	(0.22,0.25)	0.20	(0.19,0.21)

software projects (GanttProject, Rhino, Log4J and Nutch) our approach is better than all the other algorithms based on all the performance metrics with an A effect size higher than 0.94; and b) On large scale software projects (JDI-Ford, MROI-Ford, Apache Ant, Xerces-J, JHotDraw and JFreeChart), our approach is better than all the other algorithms with an A effect size higher than 0.87.

Results for RQ1a: Fig. 3.4 summarizes our findings regarding the obtained precision (PR) and recall (RC) results on the 10 systems. We found that a considerable number of proposed refactorings, with an average of more than 82% and 86% respectively in terms of precision and recall, were already applied by the software development team and suggested manually (expected refactorings). The recall scores are higher than precision ones since we found that the refactorings suggested manually by developers are incomplete compared to the solutions provided by our approach. In addition, we found that the slight deviation with the expected refactorings is not related to incorrect operations but to the fact that the developers were interested mainly in fixing the severest code smells or improving the quality of the code fragments that they frequently modify.

Fig. 3.4 also confirms the out-performance of our interactive refactoring approach comparing to existing fully-automated techniques and since we confirmed a statistically significant difference between the means of metrics, we can say that these better results are not obtained by chance. The precision and recall scores were consistent on all the ten systems which confirm that the results are independent from the size of the systems, number of refactorings and number of code smells. The closest results are those obtained by NSGA-II based on innovization (without interaction) and the multi-objective refactoring approach of Ouni et al. This may confirm that the obtained results are more due to the interaction component of our approach. A detailed qualitative discussion will be presented later in RQ1d.

Results for RQ1b: We evaluated also the ability of our approach to fix several

types of code smell. Fig. 3.4 depicts the percentage of fixed code smells (NF). It is higher than 82% on all the ten systems, which is an acceptable score since developers may reject or modify some refactorings that fix some code smells because they do not consider them very important (their goal is not to fix all code smells in the system) or the current version of the code becomes stable. Some systems, such as Rhino and Gantt, have a higher percentage of fixed code smells with an average of more than 88%. This can be explained by the fact that these systems include a higher number of code smells than others.

However, the percentage of fixed code smells (NF) is slightly lower than some fully-automated refactoring techniques such as [1] and [2]. This is can be explained by the reason that the main goal of developers during the interaction process is not to fix the maximum number the code smells detected in the system (which was the goal of [1] and [2]) thus they rejected or modified some refactorings suggested by our tool. In addition, our approach is based on a multi-objective algorithm to find a trade-off between improving the quality and reducing the number of changes. Therefore, the slight loss in NF is explained by the fact that we are not considering fixing code smells as one of the objectives, and justified by a better improvement in the quality of the refactored system.

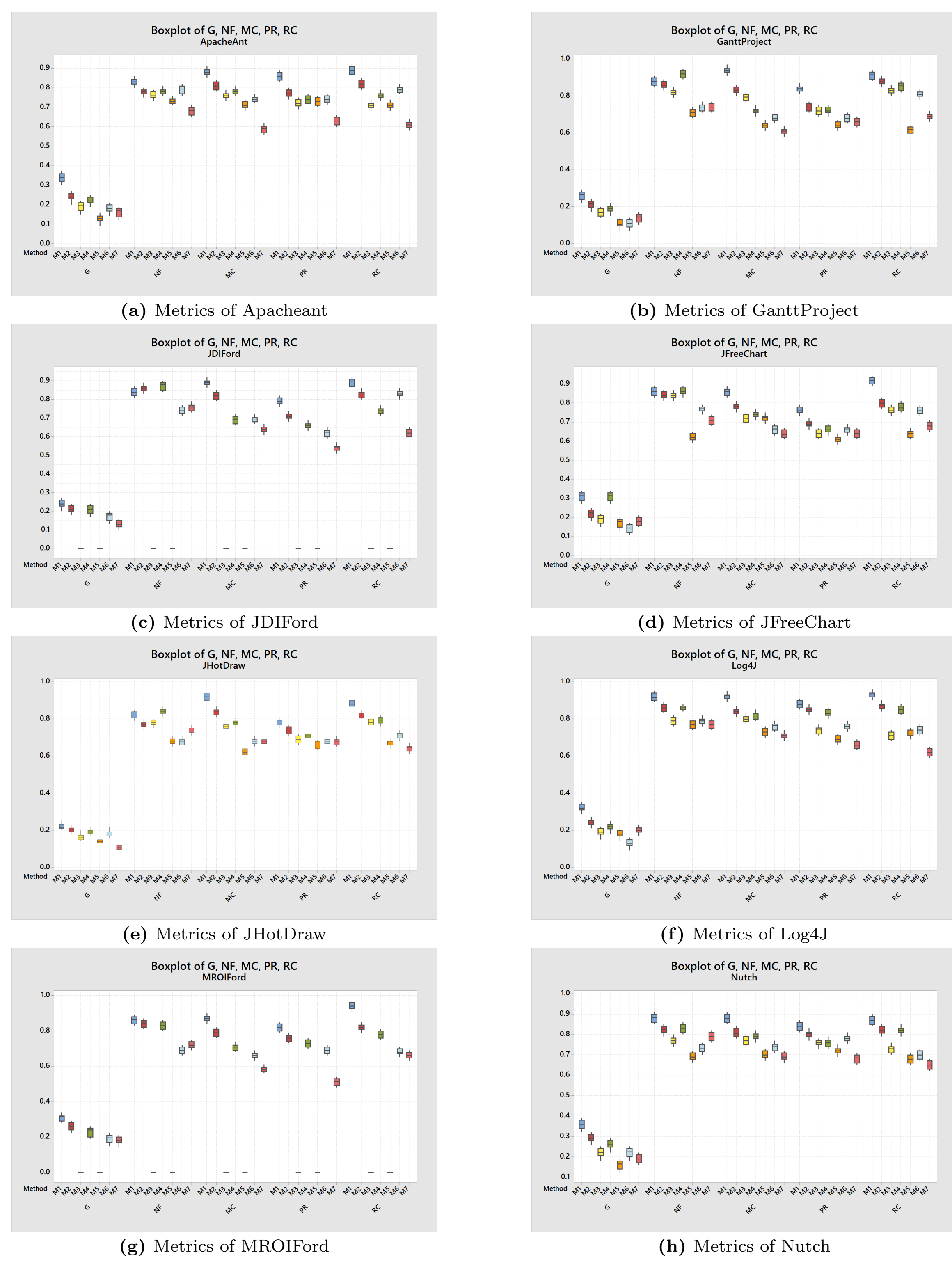
Results for RQ1c: Fig. 3.4 and Table 3.10 show that the refactorings recommended by the approach and applied by developers improved the quality metrics value (G) of the ten systems. For example, the average quality gain for the two industrial systems was the highest among the ten systems with more than 0.3. The improvements in the quality gain confirm that the recommended refactorings helped to optimize different quality metrics. The functionality attribute has the lowest improvement on the different systems. This may be explained by the fact that refactoring is expected to preserve the behavior of existing functionalities. Our interactive approach clearly also outperforms existing fully-automated techniques. One of the

reasons could be related to the fact that the optimization of the quality attributes is considered as part of the fitness functions unlike some of the existing techniques.

Results for RQ1d: We report the results of our empirical qualitative evaluation (MC) in Fig. 3.4. As reported in this figure, the majority of the refactoring solutions recommended by our interactive approach were correct and approved by developers. On average, for all of our ten studied projects, 87% of the proposed refactoring operations are considered as semantically feasible, improve the quality and are found to be useful by the software developers of our experiments. The highest MC score is 93% for the Gantt project and the lowest score is 86% for JFreeChart. Thus, it is clear that the results are independent of the size of the systems and the number of recommended refactorings. Most of the refactorings that were not manually approved by the developers were found to be either violating some post-conditions or introducing design incoherence.

Fig. 3.4 shows that our approach provides significantly higher manual correctness results (MC) than all other approaches having MC scores respectively between 60% and 78%, on average as MC scores on the different systems.

Qualitative Evaluation of RQ1 Results: To provide more qualitative evaluation, we considered some of the feedback that we received from the developers at Ford since they are part of the original developers of these systems. For example, these developers rejected a set of move methods because they believed that these methods should stay in their original class. The original class in this case is responsible for implementing several security constraints (e.g. login information) around database access. The number of security constraints is very high and they were implemented in several methods grouped into one class. Although this set of methods created a blob, the developers assessed that they should stay together because there is a logic behind implementing them in that way, and splitting the behavior may require a redesign of the application.



(a) Metrics of Apacheant

(b) Metrics of GanttProject

(c) Metrics of JDIFord

(d) Metrics of JFreeChart

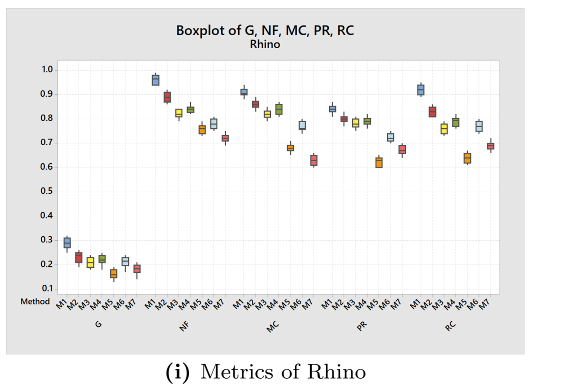
(e) Metrics of JHotDraw

(f) Metrics of Log4J

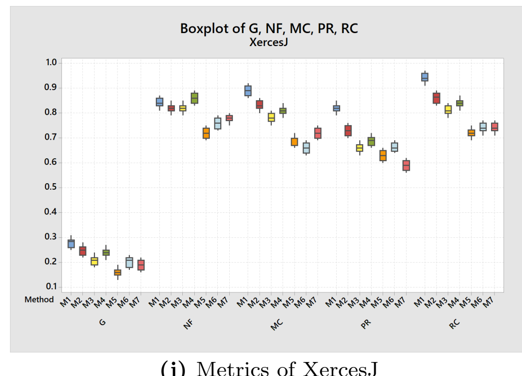
(g) Metrics of MROIFord

(h) Metrics of Nutch

Figure 3.4: Boxplots of G, NF, MC, PR, and RC on all the ten systems based on 30 independent runs. (Continue on the next page.) Label of the methods: **M1** (Our approach)=Interactive+Innovization NSGA-II, **M2**=Innovization NSGA-II, **M3**=Kessentini et al.[1], **M4**=Ouni et al.[2], **M5**=Harman et al.[3], **M6**=O'Keeffe et al.[4], **M7**=Jdeodorant [5]



(i) Metrics of Rhino



(j) Metrics of XercesJ

Figure 3.4: (Continue from the previous page.) Boxplots of G, NF, MC, PR, and RC on all the ten systems based on 30 independent runs. Label of the methods: **M1** (Our approach)=Interactive+Innovization NSGA-II, **M2**=Innovization NSGA-II, **M3**=Kessentini et al.[1], **M4**=Ouni et al.[2], **M5**=Harman et al.[3], **M6**=O'Keeffe et al.[4], **M7**=Jdeodorant [5]

In another case, the developers elected to extract a class that regroups several methods implementing a parser to extract dealer information. However, this refactoring was not recommended by our approach since the methods were located in a small class that did not contain any code smell or quality violation symptoms. Thus, the refactoring applied by the developers was more based on the features implemented in the methods. This refactoring is hard to recommend even with the considered semantics/textual similarity measures since few comments exist in these methods and furthermore their implementation structures look very different. These observations explain the reasons why some the refactorings recommended by our approach was rejected by the developers and also the differences with those that are manually recommended by the developers.

In general, we found that most of the common patterns in the Pareto front are not individual operations, but a short sequence of refactorings. Thus, we believe that most of these patterns are targeting specific quality issues and hence the applied refactorings are not individual operations but small refactoring patterns. This observation was found to be valid when we manually checked the interactive results of our

tool.

A general interesting observation from the experiments is that evolutionary search involves both diversification and convergence, so the question is does innovization emphasize convergence at the cost of sacrificing divergence? We would argue against this, for the following reasons: In the context of our refactoring problem, it is very rare to observe no overlap between non-dominated solutions for several reasons such as the large size of refactoring solutions and the fact that some common quality issues should be fixed (high priority). In fact, at least few quality issues (e.g. code smells) need to be fixed independently from the other objectives. Thus, it is normal to always observe some overlap between the refactoring solutions. Regarding diversification, the ranking of the refactoring solutions is only used after the generation of the Pareto front so this ranking is not part of the fitness function used in the search. The goal is to implicitly explore the front based on the feedback of the developers to identify the region of interest and prioritize the solutions that contain common patterns. We believe that these common patterns distinguish non-dominated solutions from dominated ones. The diversification is not penalized because we do not consider the innovization heuristic as part of the fitness functions but as a post-processing step to prioritize solutions (and not eliminating them).

We compared the results of our approach (M1) and innovization NSGA-II method (M2) in Fig. 3.4 and Table 3.10 in order to contrast the impact of interactivity component. The best solution (at the knee point) based on the innovization feature (without interaction) was evaluated based on all studied metrics. The results confirm that our interactive approach outperforms NSGA-II with the only use of innovation (without interaction) in terms of G, NF, MC, PR, and RC. However, the results of NSGA-II with innovization are better than regular multi-objective refactoring approaches (e.g. Ouni et al., etc.) thus it is clear that the positive results of our approach are due to the combination of the two factors: innovization and interactive features.

The superior performance of our interactive approach can be explained by several factors. First, [1], [4] and [3] use only structural indications (quality metrics) to evaluate the refactoring solutions and thus a high number of refactorings lead to a semantically incoherent design. Our approach reduces the number of semantic incoherencies when suggesting refactorings and during the interaction with the developers. Second, the innovization component improved the quality of the suggested refactoring solutions by using an interactive approach as compared to a regular NSGA-II where the developers need to select one solution from the Pareto front that cannot be updated dynamically. Third, JDeodorant proposes some pre-defined patterns to fix some types of code smells that cannot be sometimes generalized.

To summarize and answer RQ1, the experimentation results confirm that our interactive approach helps the participants to refactor their systems efficiently by finding more relevant refactoring solutions and improve the quality of all the ten systems under study. In addition, our interactive approach provides better results, on average, than all of the existing fully-automated refactoring techniques.

Results for RQ2: We evaluated the ability of our approach to help software developers to find quickly good refactorings based on an efficient ranking of the proposed operations. We compared the MC@k and PR@k where k was varied between 1, 5, 10 and 15 as described in Fig. 3.5 and Fig. 3.6 where show that the lowest MC@1 is 93% and the highest is 100% on the different ten systems confirming that the highest-ranked refactoring was almost always correct and relevant for the developers.

The MC@15 presents the lowest results, which is to be expected since we evaluated the manual correctness of the top 15 recommended refactorings at several interactions and this increases the probability that it contains few irrelevant refactorings. However, the average MC@15 still could be considered acceptable with an average of more than 81%. The same observations are also valid for the PR@k; however the results are a bit lower than for MC@k. The average PR@k results were respectively 94%, 89%,

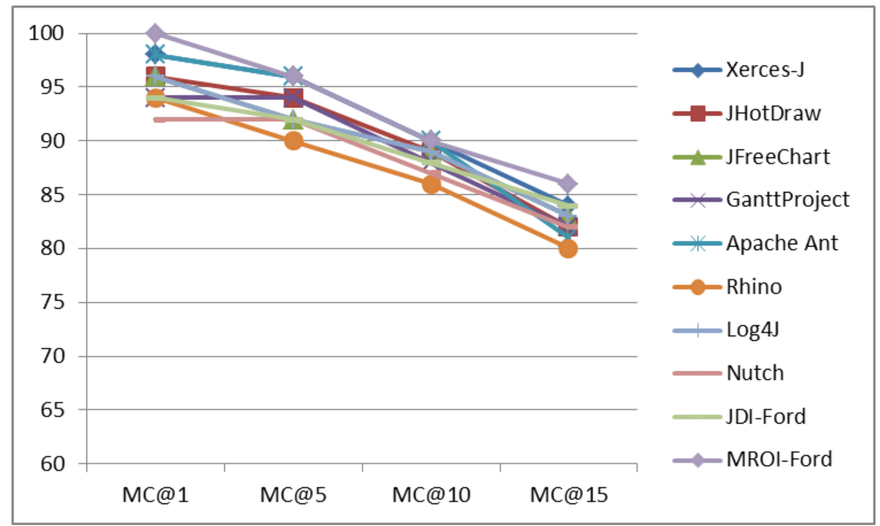


Figure 3.5: MC@k results on the different systems with k= 1, 5, 10 and 15.

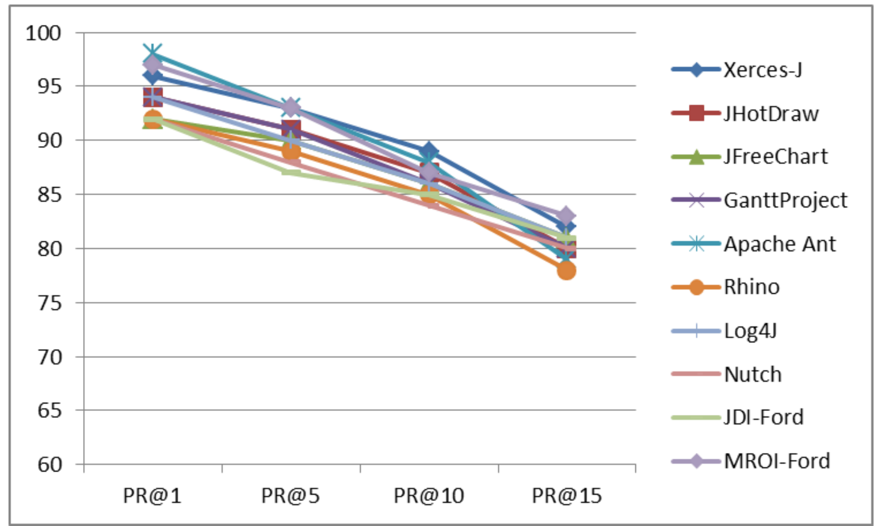


Figure 3.6: PR@k results on the different systems with k= 1, 5, 10 and 15.

84% and 80% for k = 1, 5, 10 and 15. Thus, it is clear that the ranking function used by our interactive approach to explore the Pareto front is efficient.

Considering three other metrics NAR (percentage of accepted refactorings), NMR (percentage of modified refactorings) and NRR (percentage of rejected refactorings), we seek to evaluate the efficiency of our interactive approach to rank the refactorings. We recorded these metrics using a feature that we implemented in our tool to record all the actions performed by the developers during the refactoring sessions. Fig. 3.7

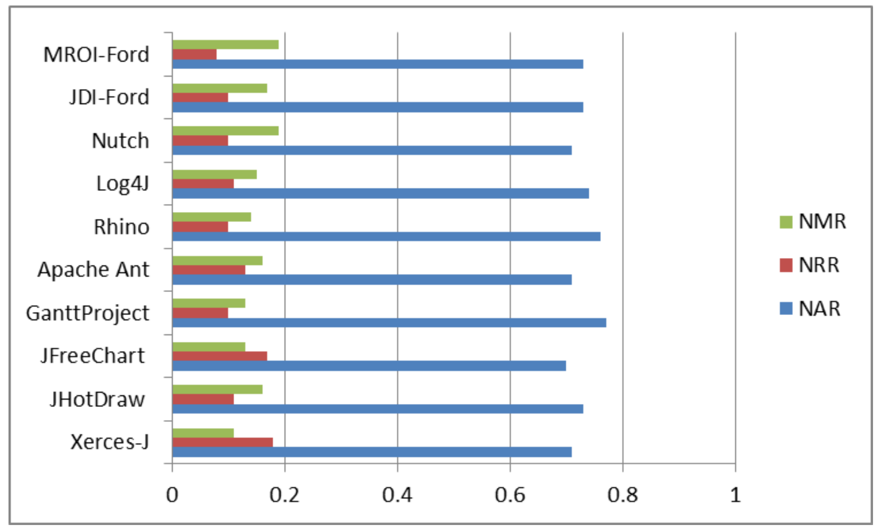


Figure 3.7: The median NMR, NRR and NAR results in the different systems.

shows that, on average, more than 71% of the recommended refactorings were applied by the developers. In addition, an average of 17% of the recommended refactorings were modified by the developers, while 12% of the suggested refactorings were rejected by the developers. Thus, it is clear that our recommendation tool successfully suggested a good set of refactorings to apply.

To conclude, our approach efficiently ranked the recommended refactorings and helped software developers to quickly find good refactorings recommendations.

Results for RQ3a: Fig. 3.8 shows that the time is reduced by 61% and 57% to finalize respectively the two tasks of fixing bugs when programmers worked on the refactored program using our interactive approach. These results outperform the productivity improvements obtained when programmers worked on similar tasks of fixing bugs of the refactored programs by Ouni et al. [2] and Harman et al. [3]. For Ouni et al., the productivity improvements are between 41% and 37% while Harman et al. [3] are between 33% and 31%. The results are correlated with the quality improvements of the evaluated programs, as discussed in the previous sections. Thus, a better quality of the software may increase the productivity of programmers when fixing bugs.

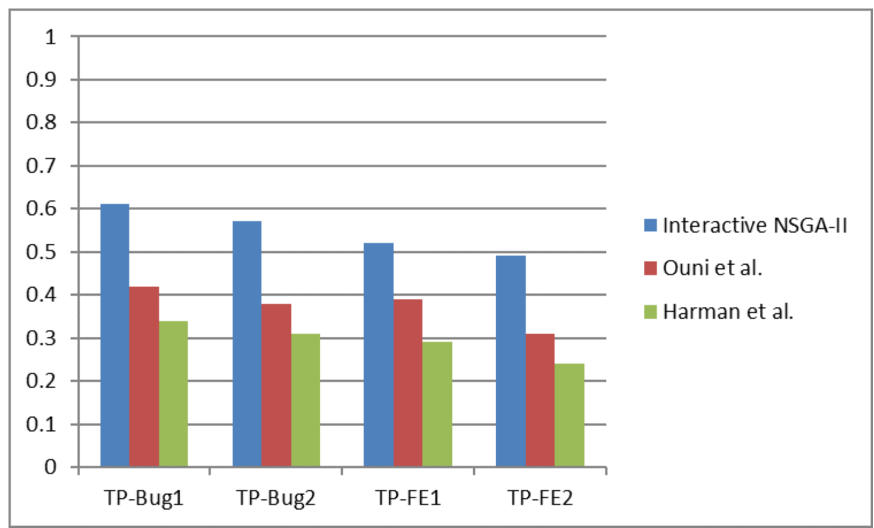


Figure 3.8: The average productivity difference (TP) results on the different tasks performed by the three groups using our interactive approach, Ouni et al. [2], Harman et al.[3]

Results for RQ3b: Similar results to RQ3a are obtained for the tasks of adding new features. Fig. 3.8 shows that the time is reduced by 51% and 48% to finalize respectively the two tasks of adding new features when programmers worked on the refactored program using our interactive approach. These results outperform the productivity improvements obtained when programmers worked on similar tasks of adding features of the refactored programs by Ouni et al. [2] and Harman *et al.* [3]. For Ouni et *et al.* , the productivity improvements are between 38% and 31% while Harman *et al.* [3] are between 29% and 23%. The results are correlated with the quality improvements of the evaluated programs. Thus, a better quality of the software may increase the productivity of programmers when adding new features. Overall, the productivity gain when programmers worked on adding new features is lower than the one observed for fixing bugs. This could be related to the fact that the complexity of adding new features was higher than fixing bugs and the locations where refactorings are introduced.

The metric (TP) to measure the time to perform the different bugs fixing and adding new features task on the systems before and after refactoring included the ex-

ecution time of the different (interactive and fully-automated) refactoring techniques to generate the new systems after refactoring. While the execution time of our interactive approach is slightly higher than fully-automated approaches with an average of 6 minutes comparing to Ouni *et al.* and Harman *et al.* on the different systems used in both scenarios, the overall time that developers spent to perform the new tasks is much lower when working on the new systems after refactoring based on our approach comparing to the state of the art. Thus, the extra manual effort required by our approach is compensated by higher productivity and better accuracy of the results. We believe that the slightly higher execution time by our interactive approach comparing to fully automated search-based refactoring despite the extra-manual effort is explained by the fact that the user feedback can reduce dramatically the search space to converge toward better recommendations. Furthermore, the efficient ranking of refactorings to be inspected by programmers help a lot in reducing the interaction time. Finally, we want to highlight that programmers spend considerable time evaluating long list of refactoring recommendations after the execution of fully-automated approaches which is comparable to the manual interaction effort required during the execution of our interactive approach.

In the following, we describe a qualitative example to illustrate the observed time reduction when updating a feature on the refactored code. The scenario consists of modifying the existing loading and saving of CSV files feature in Gant. The updated feature will enable the modification of colors used in the charts to highlight specific project tasks to match different priorities (e.g. red for high priority task, green for low priority task, etc.) then modify the current CSV format to support the use of colors in the Gantt chart.

To implement this feature, several methods have to be modified that append to different classes before refactoring. The main class related to this feature is GanttOptions that includes 68 methods and highly coupled with 14 classes which can

be considered as a blob. Our interactive refactoring tool proposed a sequence of 29 refactorings to be applied to this class and some related classes (CSVOptions and UIConfiguration). The sequence of refactorings included Extract class, Move field, Move method, PushDown field, PushDown method and Extract method that refactored the GanttOptions as illustrated in Fig. 3.9.

The new version of GanttOptions contained only 43 methods and several methods and fields were moved from/to CSVOptions and UIConfiguration. Thus, the developers introduced less number of changes to update the methods related to changing the colors of the chart tasks and the format of the CSV files since they were cohesively moved to GanttOptions after refactorings rather than being distributed between CSVOptions and UIConfiguration. These refactorings not only reduced the number of changes but also improved the coupling and cohesion within these classes since other methods and fields were moved from CSVOptions which reduced as well the time for developers to identify the relevant methods and fields to modify to integrate the new features.

Results for RQ3c: The post-study questionnaire results show the average agreement of the participants was 4.8 and 4.3 based on a Likert scale for the first and second statements (discussed in section 4.6), respectively. This confirms the usefulness of our approach for the software developers considered in our experiments.

We summarize in the following the feedback of the developers. Most of the participants mention that our interactive approach is faster than manual refactoring since they spent a long time with manual refactoring to find the locations where refactorings should be applied. For example, developers spend time when they decide to extract a class to find the methods to move to the newly created class or when they want to move a method then it takes time to find the best target class by manual exploration of the source code. Thus, the developers liked the functionality of our tool that helps them to modify a refactoring and finding quickly the right parameters

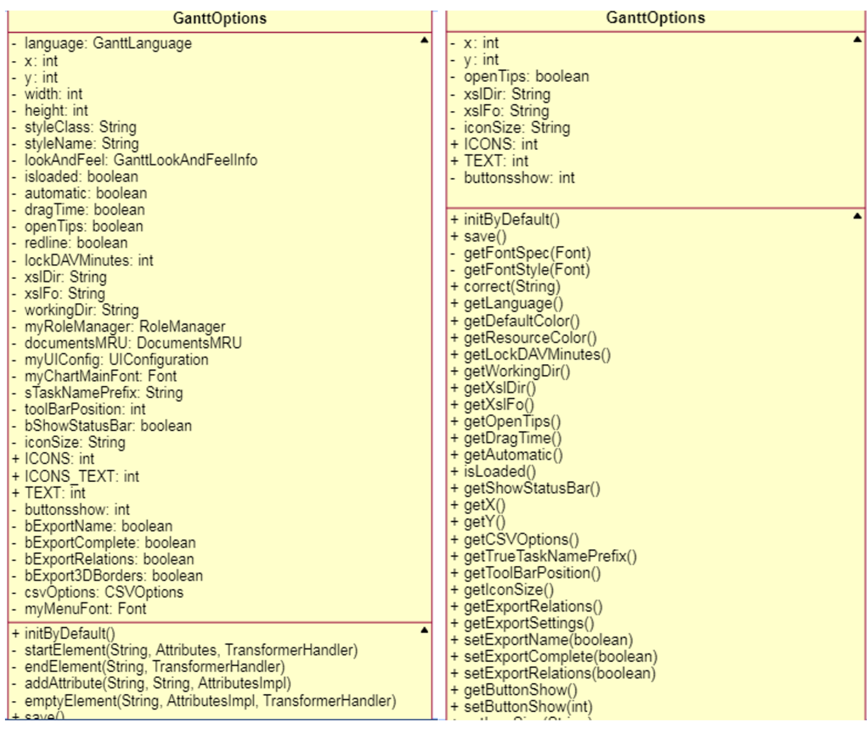


Figure 3.9: GanttOptions before and after refactoring.

based on the recommendations.

Our interactive algorithm automatically suggests a list of possible target classes ranked based on the history of changes and semantic similarity. Furthermore, refactorings may affect several locations in the source code, which is a time-consuming task to perform manually, but they can perform it instantly using our tool.

The participants found our tool helpful for both *floss refactoring*, to maintain a good quality design and also for *root canal refactoring* to fix some quality issues such as code smells. The developers justify their conclusions by the following interesting observations about our tool: a) the list of recommended refactorings helps them to choose the desired refactoring very quickly, b) our tool offers them the possibility to modify the source code (to add new functionality) while doing refactoring since the list of recommendations is updated dynamically. So developers can switch between both activities: refactoring and modifying the source code to modify existing functionalities. c) our tool allows developers to access all the functionality of the IDE

(e.g., Eclipse). d) the suggested refactorings by our interactive tool can fix code smells (root canal refactoring) or improve some quality metrics (floss canal refactoring) due to the use of the multi-objective approach.

Another important feature that the participants mention is that our interactive approach allows them to take the advantages of using multi-objective optimization for software refactoring without the need to learn anything about optimization and exploring explicitly the Pareto front to select one "ideal" solution. The implicit exploration of the Pareto front in an interactive fashion represents an important advantage of our tool along with the dynamic update of the recommended list of refactoring using innovization. In fact, the developers found a lot of difficulties using the multi-objective tool of [116] to explore the Pareto front to find a good refactoring solution. In addition, they did not appreciate the long list of refactoring suggested by [116] since they want to take control of modifying and rejecting some refactorings. In addition, the validation of this long list of refactorings is time-consuming. Thus, they appreciate that our tool suggests refactoring one by one and update the list based on the feedback of developers.

The participants also suggested some possible improvements to our interactive approach. Some participants believe that it will be very helpful to extend the tool by adding a new feature to apply automatically some regression testing techniques to generate test cases to test applied refactorings. Another possibly suggested improvement is to use some visualization techniques to evaluate the impact of applying a refactoring sequence.

3.4 Threats to Validity

There are four types of threats that can affect the validity of our experiments. We consider each of these in the following paragraphs.

Conclusion validity is concerned with the statistical relationship between the treat-

ment and the outcome. We addressed conclusion threats to validity by performing 30 independent simulation runs for each problem instance and statistically analyzing the obtained results using the Wilcoxon rank sum test with a 95% confidence level ($\alpha = 5\%$). However, the parameter tuning of the different optimization algorithms used in our experiments creates another internal threat that we need to evaluate in our future work. The parameters' values used in our experiments are found by trial-and-error, which is commonly used in the SBSE community. However, it would be an interesting perspective to design an adaptive parameter tuning strategy [134] for our approach so that parameters are updated during the execution in order to provide the best possible performance. In addition, our multi-objective formulation treats the different types of refactoring with the same weight in terms of complexity when calculating one of the fitness functions. However, some refactoring types can be more complex than others to apply by developers.

Internal validity is concerned with the causal relationship between the treatment and the outcome. We dealt with internal threats to validity by performing 30 independent simulation runs for each problem instance. This makes it highly unlikely that the observed results were caused by anything other than the applied multi-objective approach. The second internal threat is related to the variation of correctness and speed between the different groups when using our approach and other tools such as JDeodorant. 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. 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. The results obtained by the developers from Ford and those by the graduate students are consistent. The evaluated open source and industrial systems provided similar conclusions in our experiments. The industrial systems are mainly evaluated by the original developers and the results are still consistent with the open source systems.

Construct validity is concerned with the relationship between theory and what is observed. To evaluate the results of our approach, we selected solutions at the knee point when we compared our approach with fully-automated refactoring approaches, but the developers may select a different solution based on their preferences to give different weights to the objectives when selecting the best refactoring solution. The different developers involved in our experiments may have divergent opinions about the recommended refactorings in terms of correctness and readability. We considered in our experiments the majority of votes from the developers. We selected the "majority of votes" as the technique to aggregate the date since it is similar to real-world situations. 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. The architects discuss several possibilities to refactor the current architecture and they will decide the best one based on the majority. We adopted this strategy for our experiments to simulate real-world scenarios. 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 examples of refactorings already evaluated with arguments and justification. For the fatigue threat, we did not limit the time to fill the questionnaire and we also sent the questionnaires to the participants by email and gave them the required time to complete each of the required tasks. We believe that one of the principal strengths of our approach is the interaction component with the developer since many aspects of software quality are subjective and impossible to formalize precisely using quality

metrics alone. The interaction with the developer (i.e., developer feedback) can help to improve the refactoring recommendations, by critically augmenting the objective metric values with subjective developer insight. However, a better fitness function may indeed reduce the interaction effort. Thus, the use of the QMOOD model in a fitness function can be considered as a possible threat since the use of quality metrics to solutions' evaluation is subjective.

External validity refers to the generalizability of our findings. In this study, we performed our experiments on eight different widely used open-source systems belonging to different domains and having different sizes, and two industrial projects. However, we cannot assert that our results can be generalized to other applications, and to other practitioners. Future replications of this study are necessary to confirm our findings. Further empirical studies are also required to deeply evaluate the performance of the interactive NSGA-II using the same problem formulation. The first threat is the limited number of participants and evaluated systems, which externally threatens the generalizability of our results. In addition, our study was limited to the use of specific refactoring types and types of code smell. Future replications of this study are necessary to confirm our findings.

3.5 Conclusion

We proposed an interactive recommendation tool for software refactoring that dynamically adapts and suggests refactorings to developers based on their feedback and introduced code changes. Our interactive approach allows developers to benefit from search-based refactoring tools without explicitly involving any knowledge about optimization and multi-objective optimization algorithms. In fact, the exploration of the non-dominated refactoring solutions is implicitly performed based on the interaction with the developers. The feedback received from the developers is used to reduce the search space and converge to better solutions. To evaluate the effectiveness of our

tool, we conducted a human study on a set of software developers who evaluated the tool and compared it with the state-of-the-art refactoring techniques. Our evaluation results provide strong evidence that our tool improves the applicability of software refactoring, and proposes a novel way for software developers to refactor their systems interactively.

Future work involves validating our technique with additional refactoring types, programming languages and code smell types in order to conclude about the general applicability of our methodology. Furthermore, we only focused on the recommendation of refactorings. We plan to extend the approach by automating the test and verification of applied refactorings. In addition, we will consider the importance of code smells during the correction step using previous code changes, class complexity, etc. Another future research direction related to our work is to build an interactive software engineering framework that applies a similar approach to other software engineering problems such as the next release problem.

The exploration of Pareto front is a very challenging problem, and this work is the first to apply an interactive approach on a large number of Pareto optimal refactoring solutions. Thus, several extensions could be proposed to make the interaction with the users better and less time-consuming including the use of machine learning which is part of our future work.

CHAPTER IV

Reducing Interactive Refactoring Effort via Clustering-based Search

4.1 Introduction

As projects evolve, developers frequently postpone necessary system restructuring, known as refactoring [9], in the rush to deliver a new release until a crisis happens. When that occurs it often results in substantially degraded system performance, perhaps an inability to support new features, or even in terminally broken system architecture. Thus, refactoring received much attention during the last two decades to propose solutions that can manage the growing complexity of software systems nowadays.

Most existing studies focus on either manual or fully automated code-level refactoring. The manual support, integrated into modern IDEs such as Eclipse, NetBeans, and Visual Studio [113, 114, 71, 23, 33, 115, 97, 116, 75, 47, 2], consists of helping developers to apply refactorings based on automated routines that can check a list of pre- and post-conditions but they have to specify manually which types of refactoring to be applied, such as extract class or move method, and where. The fully automated techniques try to identify refactoring opportunities and which refactorings to apply using static and dynamic analysis, and the history of changes. However, design re-

structuring is a human activity that cannot be fully automated because developers understand the problem domain intuitively and they have targeted design goals in mind. Thus, several empirical studies show that fully automated refactoring does not always lead to the desired architecture [113, 107, 34, 74]. Furthermore, manual refactoring is error-prone, time consuming and not practical for radical changes. For instance, Batory et al. [115] presented several case studies where refactoring involved more than 750 refactoring steps on one project and took more than 3 weeks to execute.

Recently, few approaches have been proposed to interactively evaluate refactoring recommendations using search-based software engineering [75, 74, 135]. The developers can provide a feedback about the refactored code and introduce manual changes to some of the recommendations. However, this interactive process can be repetitive, expensive, and tedious since developers must evaluate recommended refactorings, and adapt them to the targeted design especially in large systems where the number of possible strategies can grow exponentially. Thus, we seek, in this work, to answer the fundamental scientific question: "What is the minimal guidance that leads automated search to useful and realistic refactoring recommendations?"

We propose an interactive approach combining the use of multi-objective search, based on NSGA-II [127] and unsupervised learning to reduce the developer's interaction effort when refactoring systems. We generate, first, using multi-objective search different possible refactoring strategies by finding a trade-off between several conflicting quality attributes. Then, an unsupervised learning algorithm clusters the different trade-off solutions, called the Pareto front, to guide the developers in selecting their region of interests and reduce the number of refactoring options to explore. The feedback from the developer, both at the cluster and solution levels, are used to automatically generate constraints to reduce the search space in the next iterations and focus on the region of developer preferences. For instance, the developer can select the most relevant cluster of solutions, called region of interest, based on his preferences

and the multi-objective search will reduce the space of possible solutions, in the next iterations, by generating constraints from the interaction data such as eliminating part of the code (e.g classes or methods) that are not relevant for refactoring to the programmer.

We selected 14 active developers to manually evaluate the effectiveness our tool on 5 open source projects and one industrial system. The results show that the participants found their desired refactorings faster and more accurate than the current state of the art.

The primary contributions of this chapter can be summarized as follows:

1. This chapter introduces, for the first time, an approach combining multi-objective search and machine learning to guide developers in their decision making process. The proposed technique supports the adaptation of refactoring solutions based on developer feedback and learning automatically their preferences from the interaction data.

2. We propose an intelligent exploration of the Pareto front of non-dominated solutions by grouping them into different clusters, based on the similarities between the solutions and their impact on the code, that can summarize to the developer the main options to explore to refactor their systems rather than evaluating a large number of possible strategies.

3. The project reports the results of an empirical study on an implementation of our approach. The obtained manual evaluation results provide evidence to support the claim that our proposal is more efficient, on average, than existing refactoring techniques based on a benchmark of six open source systems and one industrial project in terms of the relevance of recommended refactorings and reducing the refactoring effort.

4.2 Problem Statement

While successful tools and approaches for refactoring have been proposed, several challenges are still to be addressed to expand the adoption of refactoring tools in practice.

To investigate the challenges associated with current refactoring tools, a survey was conducted, as part of an NSF I-Corps project, with 127 professional developers at 38 medium and large companies including eBay, Amazon, Google, IBM, and others. 112 of these interviews were conducted face-to-face. As an outcome of these interviews, the following challenges were identified:

- **Challenge 1: The refactorings effort required by existing approaches and tools.** 83% of the interviewed developers confirmed that they were reluctant to use existing automated refactoring tools because those detect, in general, hundreds of code level quality issues such as anti-patterns but without specifying from where to start or how they are dependent on each others, nor are there any clear benefits such as an impact on the system's quality. During the interviews, 86% of developers confirmed that they want better refactoring tools to give them better understanding of design preferences rather than asking developers to manually inspect a large list of recommendations covering the whole system. A developer said "We need better solutions of refactoring tasks that can reduce the current time-consuming manual work of evaluating a large number of refactorings. Automated tools provide refactoring solutions that are hard and costly to repair because they did not consider our design needs and hard to assess their impact." This argument is consistent with empirical studies performed by Kim et al. [115].

- **Challenge 2: Lack of visualization support to estimate the impact of recommended refactorings.** 69 out of the 112 participants highlighted in the interviews that it is hard to understand the impact of suggested refactorings on the system and they have to look manually at the code before and after refactoring.

Determining which anti-pattern should be refactored and how is never a pure technical problem in practice. Instead, high-level refactoring decisions have to take into account trade-offs between code quality, available resources and expected effort. Furthermore, 53 participants mentioned that several refactoring ”paths” are discussed between architects to determine the best solution to restructure the current architecture or code. However, most of existing refactoring tools and approaches just recommend only one sequence of refactorings to apply.

- **Challenge 3: It is difficult for developers to express their preferences upfront.** Based on our extensive experience working on licensing refactoring research prototypes to industry, developers always have a concern on expressing their preferences upfront as an input for a tool to guide refactoring suggestions. They prefer to get insights from some generated refactoring solutions then decide which quality attributes they want to improve. However, several of existing refactoring tools fail to consider the developer 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 applied.

- **Challenge 4: Lack of refactoring tools that can learn from developers interaction.** High-level refactorings are usually systematic and repetitive in different contexts, involving similar changes to numerous locations [136]. If these repetitive high-level changes can be learned, abstracted, and automated, a large amount of maintenance effort could be saved.

To address all the above challenges, we describe in the next section our interac-

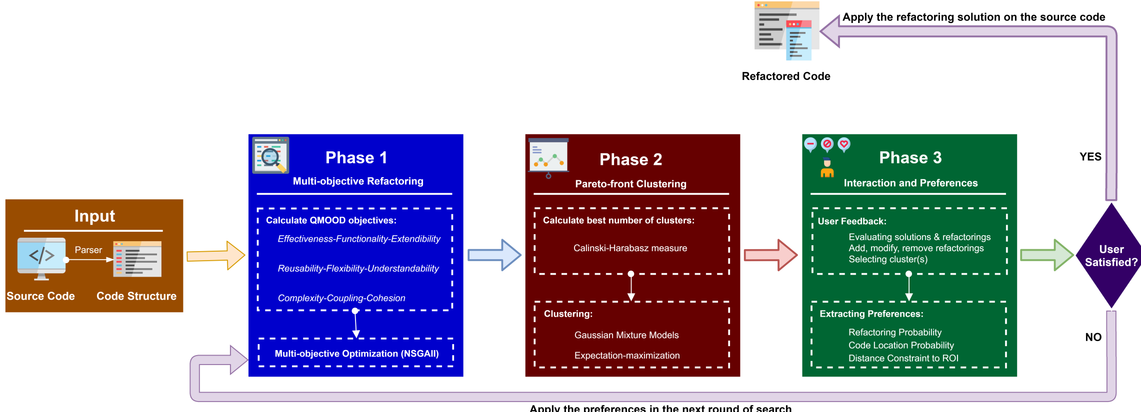


Figure 4.1: Overview of our proposed IC-NSGA-II approach.

tive clustering-based multi-objective refactoring approach then we will explain in the validation section how they are addressed by evaluating the proposed approach and tool with active developers.

4.3 Approach: Clustering-based Interactive Multi-objective Software Refactoring

In this section, we describe an overview of our proposed approach and its components. Then, we provide the details of each component.

4.3.1 Overview

The general structure of our approach is sketched in Fig. 4.1. In the following, we describe the different main components of our approach. Our technique comprises three main components. First, we extract the source code structure information using a dedicated parser. Then, we calculate software design quality metrics and use them as the fitness functions for a multi-objective search algorithm. The results of this phase is a set of Pareto-optimal solutions that can find a trade-off between different quality attributes.

After the generation of the first Pareto front, a number of clusters is selected using

the Calinski and Harabaz score [137]. Then, we fit a Gaussian Mixture Model (GMM) with the number of selected clusters. The GMM parameters are optimized by the Expectation-Maximization algorithm. Then a solution with maximum density will be identified as the cluster representative (center). In the last phase, the user can visualize the clusters of solutions and interact with our tool by evaluating solutions, modifying refactorings and selecting the desired cluster. We extract the user preferences from these activities and consider them in the next round of iterations to converge towards the user's region of interest. This loop can continue until the user is satisfied and a refactoring solution is selected to apply on the source code.

4.3.2 Phase 1: Multi-Objective Refactoring

Discovering a refactoring solution can be a challenging task since a large search space needs to be explored. This large search space is the result of the number of refactoring operations and the importance of their order and combination. To explore this search space, we propose an adaptation of the non-dominated sorting genetic algorithm (NSGA-II) [127] to interactively find a trade-off between multiple quality attributes.

A multi-objective optimization problem can be formulated in the following form:

$$\begin{cases} Minimize & F(x) = (f_1(x), F_2(x), ..., f_M(x)), \\ Subject\ to & x \in S, \\ & S = \{x \in R^m : h(x) = 0, g(x) \geq 0\}; \end{cases}$$

where S is the set of inequality and equality constraints and the functions f_i are *objective* or *fitness* functions. In multi-objective optimization, the quality of a solution is recognized by dominance. The set of feasible solutions that are not dominated by any other solution is called *Pareto-optimal* or *Non-dominated* solution set.

NSGA-II is a multi-objective evolutionary algorithm operating on a population of

candidate solutions which are evolved toward the Pareto-optimal solution set. NSGA-II uses an explicit diversity-preserving strategy together with an elite-preservation strategy [127]. The complexity of NSGA-II is at most $O(MN^2)$ where M and N are the number of objectives and the population size, respectively.

As described in Algorithm 1, the first iteration of the process begins with a complete execution of adapted NSGA-II to our refactoring recommendation problem based on the fitness functions that will be discussed later. At the beginning, a random population of encoded refactoring solutions, P_0, is generated as the initial parent population. Then, the children population, Q_0, is created from the initial population using crossover and mutation. Parent and children populations are combined together to form R_0. Finally, a subset of solutions is selected from R_0 based on the crowding distance and domination rules. This selection is based on elitism which means keeping the best solutions from the parent and child population. Elitism does not allow an already discovered non-dominated solution to be removed. This process is continued until the stopping criteria is satisfied.

The results of the first execution of search algorithm are a set of non-dominated solutions that will be clustered and then updated by the users. After this interactions phase, the multi-objective search algorithm will continue to run using the new constraints generated at the cluster and solution levels.

4.3.2.1 Refactoring Solution Representation

A refactoring solution is represented as a vector consists of an ordered sequence of multiple refactoring operations. Each refactoring operation includes a refactoring action and its specific controlling parameters. The refactoring types considered in our experiments are: Move Method, Move Field, Extract Class, Encapsulate Field, Pull Up Field, Pull Up Method, Push Down Field, Push Down Method, Extract SubClass, Extract SuperClass. These refactoring operations are described in Table 2.1.

Algorithm 1: Interactive Clustering-based NSGA-II (IC-NSGA-II)

```
      Input  : Population Size (N), Source Code
      Output: Recommended Pareto-optimal Solutions
1     UserPreferences ← ∅ ;        /* Initiate Preference Parameters */
2     while ¬ The user is satisfied do
phase1    begin Multi-objective Refactoring
4             P_1 ← InitializePopulation(N,UserPreferences);         /* User
               preferred random population */
5             EvaluateObjectives(P_1,UserPreferences);
6             FastNonDominatedSort(P_1);
7             Q_1 ← SelectCrossoverMutate(P_1,UserPreferences);
8             while ¬StoppingCondition() do
9                 EvaluateObjectives(Q_1,UserPreferences); /* User preferred
                   evaluation */
10                R_t ← P_1 ∪ Q_1;
11                Fronts=FastNonDominatedSort(R_t);
12                P_{t+1} ← ∅;
13                i ← 1 ;
14                while |P_{t+1}| + |Front_i| ≤ N do
15                    CrowdingDistanceAssign(Front_i);
16                    P_{t+1} ← P_{t+1} ∪ Front_i;
17                    i ← i + 1;
18                SortByRankAndDistance(Front_i);
19                P_{t+1} ← P_{t+1} ∪ Front_i[1 : (N − |P_{t+1}|)];
20                Q_{t+1} ← SelectCrossoverMutate(P_{t+1},UserPreferences) ;
                   /* Customized GA Operator */
21                t = t + 1;
22            RecommendedSolutions ← Q_{t+1};
phase2    begin Pareto Front Clustering
24            GMMClustering (RecommendedSolutions);       /* Described in
               Algorithm 2 */
25            ClustersCenter ();
phase3    begin Interaction and User Preference
27            GetUserFeedBack (Clusters,Centers) ;         /* Described in
               Algorithm 3 */
28            UserPreferences ← ExtractPreferences ();
29    Return RecommendedSolutions;
```

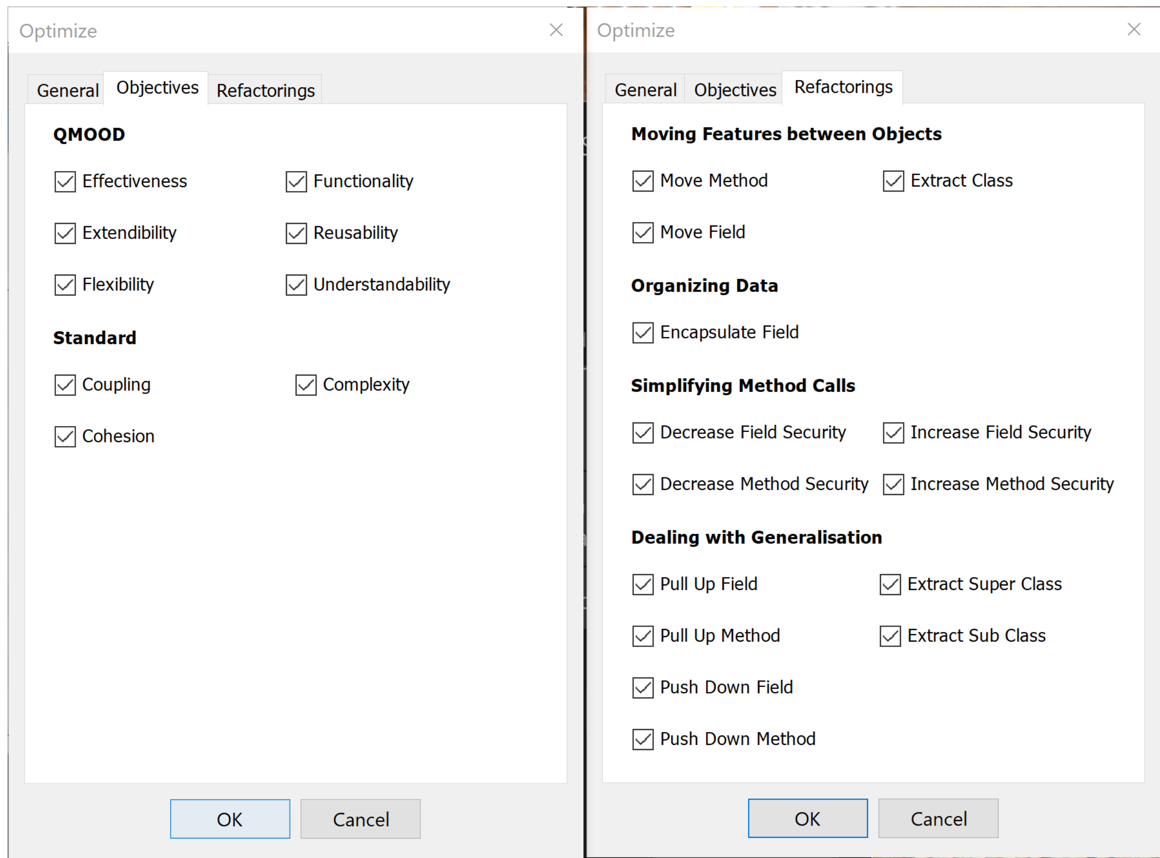


Figure 4.2: Allowing user to select the desired refactoring operators and fitness functions in our tool

Refactoring operations are created or modified randomly during the population initialization or mutation. Also, the size of a solution vector which is the number of included refactoring operation is randomly selected between lower and upper bound values. Therefore, it is important to investigate the feasibility of a solution and its operations using related pre- and post-conditions [10]. These conditions ensure that the program will not break while the behavior is preserved by the refactoring. Our tool allows the user to select the desired refactoring operations to be included in the process as it is shown in Figure 4.2.

4.3.2.2 Fitness Functions

The fitness or objective function evaluates a candidate solution and calculates its goodness degree to the considered problem. In order to measure the influence of a refactoring solution on the software project, we utilized QMOOD [46] which is described in Subsection 2.2.3 and 2.2 and 2.3.

The relative change of the quality metric after applying the refactoring solution is considered as the fitness function and can be expressed as:

$$FitnessFunction_i = \frac{QM_i^{after} - QM_i^{before}}{QM_i^{before}} \tag{4.1}$$

where QM_i^{before} and QM_i^{after} are the value of the quality metric i before and after applying a refactoring solution, respectively.

4.3.2.3 Variation Operators

Variation operators help to navigate through the search space and to maintain a good diversity in the population. There are three variation operators used in the optimization algorithm known as selection, crossover, and mutation.

- **Selection**: Parent selection is a crucial step which directly affects the convergence rate. We used "Roulette Wheel Selection". The idea is to divide a circular wheel and assign the pies to each individual based on its fitness value. Therefore, the more fitted individual has a higher chance to be selected.

- **Corssover**: The process of combining parents in order to generate new offsprings is called parent crossover. We utilized "Single Point Crossover" operator for this mean. In this operator, a random crossover point is chosen and then the two sides of the parents are swapped to produce new children.

- **Mutation**: A small random modification in solution individual is named muta-

tion. This process aid to keep diversity in the population. However, by assigning a low probability to this operator, we avoid a random search. We employed "Bit Flip Mutation" with which a random refactoring operation is selected and replaced with another randomly selected available refactoring operation.

4.3.3 Phase 2: Clustering the Pareto Front of Refactoring Solutions

The goal of this phase is to reduce the effort to investigate the solutions in Pareto-optimal front. We try to group the solutions based on their fitness function values without filtering or removing any of them. In this way, the solutions can be categorized based the similarity among them in the objectives space. Then, a representative solution is identified from each partition to recommend to the decision maker (center of the cluster). For this purpose we used clustering analysis technique. Clustering is one of the most important and popular unsupervised learning problems in Machine Learning. It helps to find a structure in a set of unlabelled data in a way that the data in each cluster are similar together while they are dissimilar to the data in other clusters.

One of the challenges in cluster analysis is to define the optimal number of clusters. Therefore, we need cluster validity index as a measure of clustering performance. Different partitions is computed and the ones that fits the data better are selected. The procedure of Phase 2 is illustrated in Algorithm 2.

4.3.3.1 Calinski Harabasz (CH) Index

Calinski Harabasz (CH) Index is an internal clustering validation measure based on two criteria: compactness and separation [137]. CH evaluates the clustering results based on the average sum of squares between and within clusters and it defines as

Algorithm 2: Pareto-front Clustering

```
    Input  : Pareto-front solutions (S)
    Output: Labeled solutions (LS),
            Clusters Representative Solution (CR)
1  begin Calculate best number of clusters-K
2      for i ← 2 to 10 do
3          LS = GMMClustering (i, S);
4          Score_i=CalinskiHarabaszIndex(LS);
5      K ← MaxScoreIdx();
6  begin GMMClustering (K,S)
7      μ_k, Σ_k, π_k ← Initialize-K-Gaussian();
       /* Expectation-Maximization                                   */
8      while ¬ converge do
9          γ(s_nk) ← Expectation();
10         μ_k, Σ_k, π_k ← Maximization();
11         EvaluateLikelihood();
12     foreach s_n ∈ S do
           /* assigning cluster labels                               */
13         L_n ← MaxResponsibilityIdx(s_n);
       /* Find Clusters Representative                               */
14     foreach Cluster C_k do
15         CR_k ← MaxDensity(s_nk ∈ C_k)
16 Return LS, CR;
```

Figure 4.3: Psuedo-code for Phase 2 of our proposed approach.

follows:

$$CH = \frac{(N-K)}{(K-1)} \frac{\sum_{k=1}^{K} |c_k| \; dist(\overline{c_k}, \overline{S})}{\sum_{k=1}^{K} \sum_{s_i \in c_k} dist(s_i, \overline{c_k})} \tag{4.2}$$

where $dist(a, b)$ is the Euclidean distance, $\overline{c_k}$ and $\overline{S}$ are the cluster and global centroids, respectively.

The first step in Pareto-front clustering is to execute the clustering process with different number of components and to compute CH score for each. The best number of clusters (K) is defined as the one that achieves the highest CH score.

Gaussian Mixture Model (GMM) is a probabilistic model-based clustering algorithm with which a mixture of k Gaussian distributions is fitted on the data. GMM is soft-clustering approach in which each data point is assigned a degree that it belongs to each of the clusters. The parameters that need to fit are Mean (μ_k), Co-variance (Σ_k), and Mixing coefficient (π_k).

GMM clustering begins by random initiation of parameters for K components. Then, Expectation-Maximization (EM) algorithm [138] is employed for parameter estimation. EM is an iterative process to train the parameters and has two steps. In the expectation step, an assignment score to each Gaussian distribution, called "responsibility" or "membership weight", is determined for each solution point as follow:

$$\gamma(z_{nk}) = \frac{\pi_k \mathcal{N}(s_n|\mu_k, \Sigma_k)}{\sum_{i=1}^{K} \pi_i \mathcal{N}(s_n|\mu_i, \Sigma_i)} \tag{4.3}$$

The responsibility coefficient will be used later for preference extraction step. In the maximization step, the parameters of each Gaussian are updated using the computed responsibility coefficients. Lastly, "Likelihood", the probability that the data S was generated by the fitted Gaussian mixture, is computed. After the convergence of EM, each solution is labeled appropriately. Furthermore, in order to find a representative member of each cluster, we measure the corresponding density for each solution and select the solution with the highest density value.

4.3.4 Phase 3: Developers Interaction and Preferences Extraction

Our tool presents the results of clustering-based multi-objective refactoring in a user-friendly way via interactive colored graphical charts and tables as shown in Figure 4.4 and Figure 4.5.

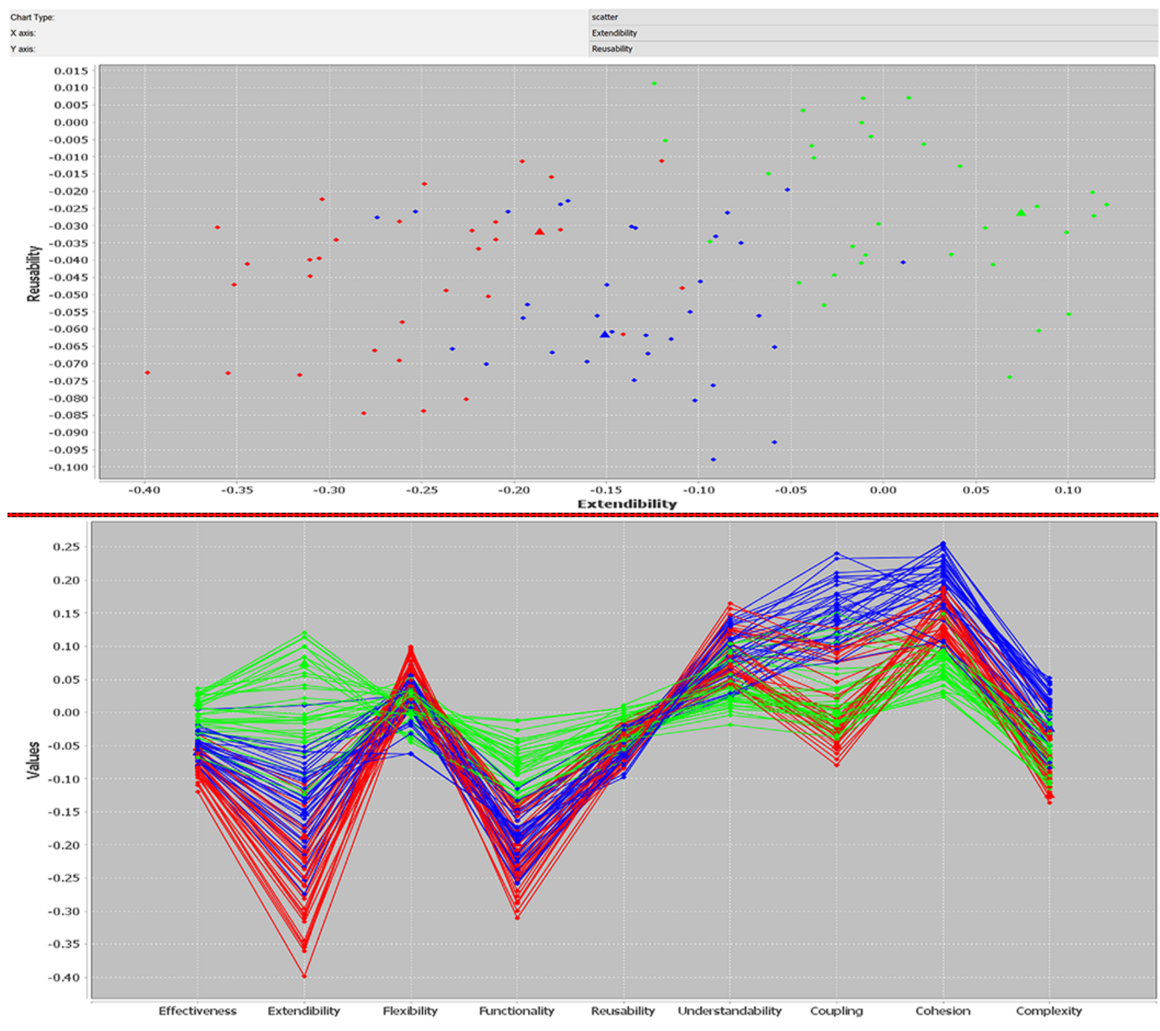


Figure 4.4: Interactive solution charts in our tool.

The developer has the ability to explore the recommended solutions and clusters efficiently and discover the shared underlying characteristics of the solutions in a cluster at a glance. The user may only investigate the cluster's center solution or search further and examine the solutions inside a cluster of interest. Every refactoring operation can be evaluated by the programmer. As described in Algorithm 3, We translate each evaluation feedback to a continuous score in the range of [-1,1].

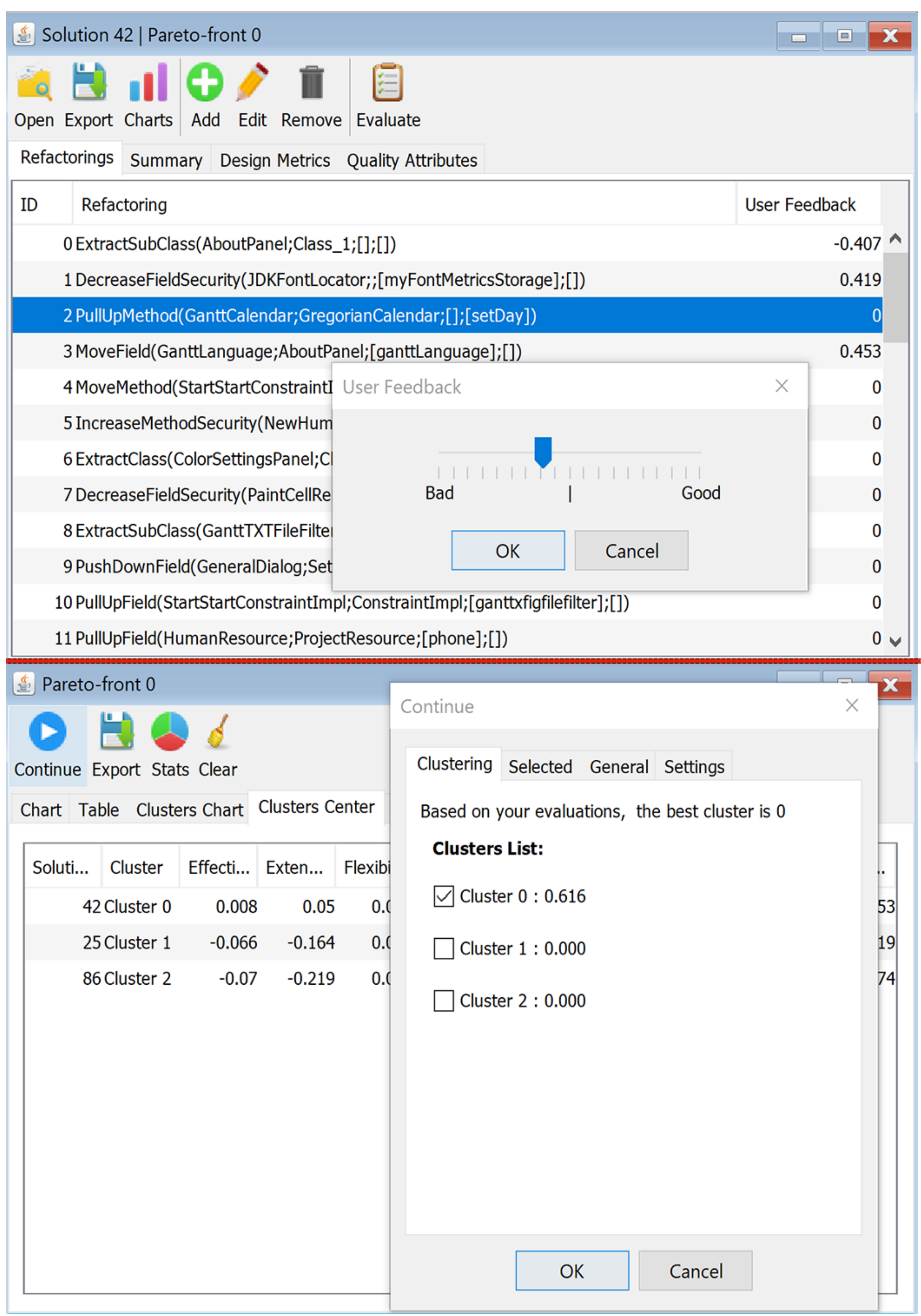


Figure 4.5: Interactive solution tables and cluster selection in our tool.

Algorithm 3: Interaction and User Preferences

```
Input   : Labeled solutions (LS)
Output: Preferred Cluster (PC),
        Preference Parameters=[
        CWP(Classes Weighted Probability,
        RWP(Refactorings Weighted Probability),
        RS(Reference Solution)]
begin User Interaction and Feedback
    while ¬ interaction is done do
        Feedback_i ← UserEvaluation(Ref_i);
        V_i ← Score(Feedback_i);
    /* SOLUTIONS AND CLUSTERS SCORE                                  */
    Score_{s_i} ← Average(V_i ∈ s_i);
    Score_{c_k} ← Average(Score_{s_i} ∈ c_k);
    PC ← cluster with Max score;
begin User Preference Extraction
    /* REPRESENTATIVE SOLUTION AS REFERENCE                          */
    RS ← CR_{PC};
    foreach [ref_i, cl_i] ∈ PC do
        RWP_p ← AverageWeightedFreq(ref_p);
        CWP_q ← AverageWeightedFreq(cl_q);
Return PC, Preference Parameters[];
```

The user can interact with the tool at the solution level by accepting / rejecting / modifying specific refactoring or the cluster level by specifying a specific cluster as the region of interest. After the interaction is done and the user decides to continue to the next round, the score of each solution and cluster are computed. Solution score ($Score_{s_i}$) is defined as the average of all refactoring operations score exists in the solution vector. Similarly, Cluster score ($Score_{c_k}$) is calculated as the average of all solutions score assigned to the cluster. Then, the cluster achieved the highest score among all clusters is considered as the user preferred partition in Pareto-front space from which the preference parameters will be extracted.

The next step of phase 3 of our proposed approach is to extract user preference parameters from the interaction step. We consider the representative solution of the

preferred cluster as the reference point. Then, we compute the weighted probability of refactoring operations (RWP) and target classes of the source code (CWP). Note that only the name of refactoring action without its associated controlling parameters is matched. Assuming the selected cluster's index is j, these parameters can be computed as follow:

$$RWP_p = \frac{\sum_{s_i \in c_j} \gamma_{ij} \times (|r_p \in s_i|)}{\sum_{r_m \in Ref} \sum_{s_i \in c_j} \gamma_{ij} \times (|r_m \in s_i|)} \tag{4.4}$$

$$CWP_q = \frac{\sum_{s_i \in c_j} \gamma_{ij} \times (|cl_q \in s_i|)}{\sum_{cl_m \in Cls} \sum_{s_i \in c_j} \gamma_{ij} \times (|cl_m \in s_i|)} \tag{4.5}$$

where s_i is the solution vector, γ_{ij} is the membership coefficient of solution i to the cluster j, r is refactoring action, Ref is the set of all refactoring operations, and Cls is the set of all classes in the source code.

At this point, if the user satisfied with the recommended refactoring solution(s), they can be applied on the source code, otherwise, we consider the extracted preferences in the next round of optimization which is detailed in the next subsection.

4.3.5 Applying Preference Parameters

If the user decides to continue the search process, then the preference parameters will be applied during the execution of different components of multi-objective optimization as described in the following:

- *Preference-based initial population*: The solutions from preferred clusters will make up the initial population of next iteration as a means of customized search starting point. In this way, we initiate the search from the region of interest rather than randomly. New solutions need to be generated to fill and achieve the pre-defined population size. Instead of random creation of the refactoring

operations (refactoring action and target class) based on a unify probability distribution, we utilize RWP and CWP as a probability distribution.

- *Preference-based mutation*: For this operator, similarly, if a solution is selected to mutate, we give a higher chance to refactoring operations of interest to replace the chosen one based on the probability distribution RWP.

- *Preference-based selection*: the selection operator tends to filter the population and assign higher chance to the more valuable ones based on their fitness values. In order to consider the user preferences in this process, we adjusted this operator to include closeness to the reference solution as an added measure of being a valuable individual of the population. That means the chance of selection is related to both fitness values and distance to the region of interest as:

$$Chance(s_i) \propto \frac{1}{dist(s_i, CR_j)}, Fitness(s_i) \tag{4.6}$$

  where $dist()$ indicates Euclidean distance and CR_j is the representative solution of cluster j.

The above-mentioned customized operators aid to keep the stochastic nature of the optimization process and at the same time take the user preferred refactoring and target code locations (classes) into account. This oppose to simple post-filtering and limiting the population to the individuals of region of interest.

4.4 Evaluation

In this section, we first present our research questions and validation methodology followed by experimental setup. Then, we describe and discuss the obtained results. The data of our experiments including a tool demo and the complete statistical results can be found in the following link [139].

4.4.1 Research Questions

We defined three main research questions to measure the correctness, relevance and benefits of our interactive clustering-based multi-objective refactoring tool comparing to existing approaches that are based on interactive mutli-objective search [75], fully automated multi-objective search (Ouni et al.) [2] and fully automated deterministic tool not based on heuristic search (JDeodorant) [5].

The research questions are as follows:

- **RQ1: Refactorings relevance.** To what extent can our approach make meaningful recommendations compared to existing refactoring techniques?
- **RQ2: Interactive clustering relevance.** To what extent can our clustering-based approach **efficiently** reduce the interaction effort?
- **RQ3: Impact.** How do programmers evaluate the **usefulness of our tool** (questionnaire)?

4.4.2 Experimental Setup

To address the different research questions, we used the six systems in Table 4.1. We selected these six systems because of their size, have been actively developed over the past 10 years and extensively analyzed by the competitive tools considered in this work. UTest[1] is a project of our industrial partner used for identifying, reporting and fixing bugs. We selected that system for our experiments since three programmers of that system agreed to participate in the experiments and they are very knowledgeable about refactoring since they are part of the maintenance team. Table 4.1 provides information about the size of the subject systems (in terms of number of classes and KLOC).

[1]Company anonymized for double-blind.

Table 4.1: Statistics of the studied systems.

System	Release	#Classes	KLOC
ArgoUML	v0.3	1358	114
JHotDraw	v7.5.1	585	25
GanttProject	v1.11.1	245	49
UTest	v7.9	357	74
Apache Ant	v1.8.2	1191	112
Azureus	v2.3.0.6	1449	117

To answer RQ1, we asked a group of 14 active programmers to identify and manually evaluate the relevance of the best refactorings sequence that they found using four tools. These tools are our IC-NSGA-II approach, an existing interactive multi-objective refactoring tool [75] (without the clustering feature) and two fully-automated refactoring tools by the means of Ouni *et al.* [2] and JDeodorant [5]. Ouni *et al.* [2] proposed a multi-objective refactoring formulation based on NSGA-II that generates a solution to maximize the design coherence and refactorings reuse from previous releases. JDeodorant [5] is an Eclipse plugin to detect bad smells and apply refactorings. 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 these refactorings. Mkaouer *et al.* [75] proposed a tool for interactive multi-objective refactoring but the interactions were limited to the refactorings (accept/reject) and there is no clustering of the Pareto front or learning mechanisms from the interaction data. We used these three competitive tools to evaluate the benefits of the clustering feature in helping developers identifying relevant refactorings.

We preferred not to use the antipatterns and internal quality indicators as proxies for estimating the refactorings relevance since we the developers manual evaluation already includes the review of the impact of suggested changes on the quality. Furthermore, not all the refactorings that improve any quality attributes are relevant to the developers, which is one of the main motivations of this work. The only rigorous way to evaluate our the relevance of our tool is the manual evaluation of the results by active developers.

Table 4.2: Selected programmers.

System	#Subjects	Avg. Prog. Exp.	Avg. Refactoring Exp.
ArgoUML	4	10	High
JHotDraw	4	11.5	Very High
Azureus	4	9	Medium
GanttProject	4	10.5	High
UTest	7	13.5	Very High
Apache Ant	4	12	Very High

Participants were first asked to fill out a pre-study questionnaire containing five questions. The questionnaire helped to collect background information such as their role within the company, their programming experience, and their familiarity with software refactoring. In addition, 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 4.2 including their programming experience, familiarity with refactoring, etc. Each participant was asked to assess the meaningfulness of the refactorings recommended after using two out of the four tools on two different systems to avoid the 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 are related to the participants from the industrial partner where only two out of the three agreed to evaluate an additional system to UTest while the third only reviewed the refactoring recommendations on the industrial software. Thus, the total number of evaluations of the different tools is 27. We assigned the tasks to the participants according to the studied systems, the techniques to be tested and developers' experience. Each of the four tools has been evaluated at least one time on every of the six systems.

To answer RQ2, we measured the time (T) that developers spent to identify the best refactoring strategies based on their preferences and the number of refactorings (NR). Furthermore, we qualitatively evaluated the impact of the interactions with the users on the Pareto front to better converge towards a "region of interests" reflecting

their preferences. For this research question, we decided to limit the comparison to only the interactive multi-objective work of Mkaouer et al. [75] since it is the only one that offers interaction with the users and it will help us understand the real impact of the clustering feature (not supported by [75]) on the refactoring recommendations and interaction effort.

To answer RQ3, we asked the participants to use our tool during a period of two hours on the different systems and then we collected their opinions based on a post-study questionnaire. To better understand subjects' opinions with regard to usefulness and usablility of our approach in a real setting, the post-study questionnaire was given to each participant after completing the refactoring tasks using our interactive approach and all the techniques considered in our experiments. The questionnaires collected the opinions of the participants about their experience in using our tool compared to existing manual, interactive and fully-automated refactoring techniques.

4.4.3 Statistical Tests and Parameters Setting

We used one-way ANOVA statistical test with a 95% confidence level ($\alpha = 5\%$) to find out whether our sample results of different approaches are different significantly. Since one-way ANOVA is an omnibus test, A statistically significant result determines whether three or more group means differ in some undisclosed way in the population. One-way ANOVA is conducted for the results obtained from each software project to investigate and compare each performance metric (dependent variable) between various studied algorithms (independent variable). We test the null hypothesis (H_0) that population means of each metric are equal for all methods against the alternative (H_1) that they are not all equal and at least one method population mean is different.

One-way ANOVA does not report the size of the difference. Therefore, we calculated the Vargha-Delaney A measure [133] 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).

A detailed description of the statistical tests results can be found in this link [139].

Parameter setting influences significantly the performance of a search algorithm on a particular problem [130]. For this reason, for each algorithm and for each system, we perform a set of experiments using several population sizes: 50, 100, 150, 200, 250 and 30. The stopping criterion was set to 100,000 evaluations for all search algorithms in order to ensure fairness of comparison (without counting the number of interactions since it is part of the users decision to reach the best solution based on his preferences). The other parameters' values were fixed by trial and error and are as follows: crossover probability = 0.6; mutation probability = 0.5 where the probability of gene modification is 0.4.

In order to have significant results, for each couple (algorithm, system), we use the trial and error method [131] in order to obtain a good parameter configuration.

4.4.4 Results

Results for RQ1: Refactorings relevance. We report the results of our empirical qualitative evaluation (MC) in Figure 4.6 based on the manual checking of the best solutions identified by each tool. As reported in this figure, the majority of the refactoring solutions recommended by our interactive clustering-based approach were correct and validated by the participants on the different systems. On average, for all of our ten studied projects, 86% of the proposed refactoring operations are considered as semantically feasible, improve the quality and are found to be useful by the software developers of our experiments. The remaining approaches have an average of 70%, 63% and 52% respectively for Mkoauer et al. (interactive multi-

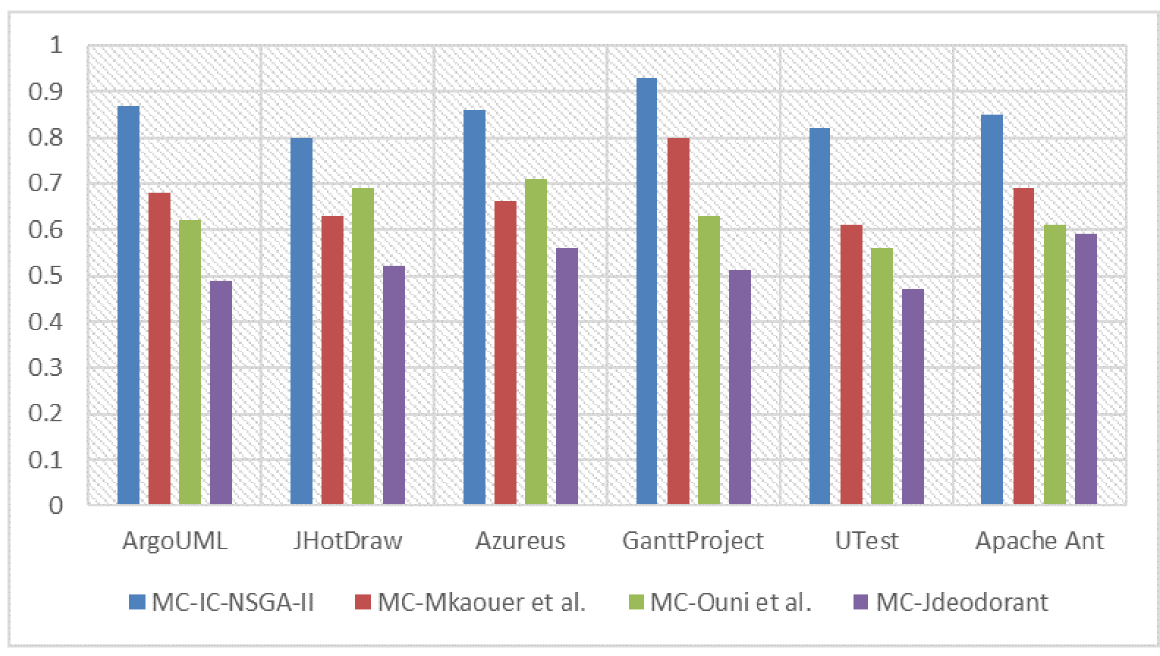


Figure 4.6: The median manual evaluation scores, MC, on the six systems with 95% confidence level ($\alpha = 5\%$) based on a one-way ANOVA statistical test

objective approach), Ouni et al. (fully automated multi-objective approach) and JDeodorant (deterministic non-search based approach).The highest MC score is 93% for the Gantt project and the lowest score is 80% for JHotDraw. Thus, it is clear that the results are independent of the size of the systems and the number of recommended refactorings as detailed in RQ2 as well. Both of the interactive tools outperformed fully-automated ones which shows the importance of integrating the human in the loop when refactoring a system. Furthermore, it is clear that adding the clustering feature to enable the developers to select a region of interests based on which quality objectives they want to prioritize and what refactoring solutions they partially liked.

A qualitative analysis of the results show that several interactions with the developers helped to reduce the search space by avoiding the refactorings that were rejected by them and their location. We found that the best final refactoring solutions identified by the developers after several interactions with our tool cannot be recommended by the remaining approaches. In fact, all these solutions are obtained either after 1) eliminating refactorings applied to specific code locations not relevant

to the programmers' context (something that cannot be learned with the interaction component) or 2) emphasizing specific cluster that prioritizes some objectives and penalizes others. For instance, the developers from the industrial partner found several of the refactorings that are recommended by Ouni et al. and JDeodorant as non relevant, while they could be correct, because it may refactor a stable code or classes that are not of their interest to be refactored.

All the results based on the MC metric on the different systems were statistically significant with 95% of confidence level. Regarding the effect size, we found that our approach is better than all the other algorithms with an A effect size higher than 0.92 for ArgoUML, GanttProject, UTest and Apache Ant; and an A effect size higher than 0.83 for JHotDraw and Azureus.

Results for RQ2: Interactive clustering relevance. Table 4.3 summarizes the time, in minutes, and the number of refactorings in the most relevant solution found using our tool, IC-NSGA-II, and the interactive approach of Mkaouer et al. [75]. All the participants spent less time to find the most relevant refactorings on the different systems comparing to Mkaouer et al. [75]. For instance, the average time is reduced by over 60% for the case of Apache Ant from 147 minutes to just 51 minutes. The time includes the execution of IC-NSGA-II and the different phases of interaction until that the developer is satisfied with a specific solution. It is clear as well that the time reduction is not correlated with the number of recommended refactorings. For instance, the deviation between IC-NSGA-II and Mkaouer et al. for Apache Ant in terms of number of recommended refactorings is limited to 9 (26 vs 35) but the time reduction is almost 100 minutes. However, it is clear that our approach reduced as well the number of recommended refactorings comparing to Mkaouer et al. while increasing the manual correctness as described in RQ1. The highest number of refactorings was observed on the industrial system with 52 refactorings using IC-NSGA-II and 75 refactorings using Mkaouer et al. This could be explained by the

Table 4.3: Median time, in minutes, and number of refactorings proposed by both interactive approaches on the different six systems.

Systems	IC-NSGA-II (T,NR)		Mkaouer et al. (T,NR)	
ArgoUML	100	29	124	34
JHotDraw	25	27	67	52
Azureus	70	24	125	35
GanttProject	36	30	86	39
UTest	46	52	83	75
Apache Ant	51	26	147	35

fact that the original developers can better understand the possible relevance of the recommended refactorings comparing the remaining participants' evaluation on the open source systems.

Figure 4.7 shows a qualitative example extracted from our experiments using IC-NSGA-II on the Gantt project with a population size of 100 based on three phases of interactions. After the generation of the Pareto front, the clustering feature identified three main different clusters for the two objectives selected by the developer (extendibility and effectiveness). During the first phase, the developer selected the cluster with id 0 as the preferred one after exploring several refactoring solutions in that cluster including the center of the cluster. Thus, the next iterations of IC-NSGA-II prioritized that "region of interest" so more refactoring options were generated around the previously selected cluster. Then, since the user selected again a cluster maximizing these two objectives (cluster with id 1) more refactoring options in the next iterations until that a good refactoring sequence is selected.

Results for RQ3: Impact. We summarize in the following the feedback of the developers based on the post-study questionnaire. 12 out the 14 participants mention that our interactive clustering-based refactoring tool is faster and much easier to use than the interactive multi-objective tool of Mkaouer et al. [75] to identify quickly relevant refactorings based on their interests. For instance, the comment of one participant is the following : *"I believe the addition of the clustering algorithm really helped identify a solution quicker. It was difficult to decide between similar*

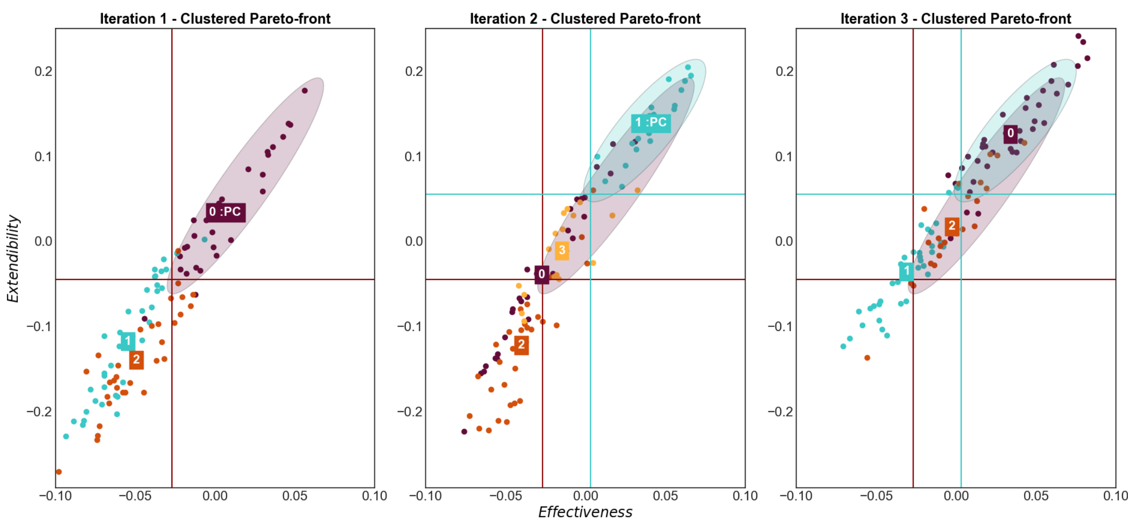


Figure 4.7: Illustration of the refactoring solutions convergence to a region of interest after two rounds of interactions extracted from the experiments on the Gantt Project.

refactoring solutions using the non-clustering version of the tool. The cluster centers helped focus the attention to just a few solutions, which were easy to choose between." A similar observation is valid when comparing our tool to the fully-automated multi-objective refactorings tool of Ouni et al. [2] where 9 out of the 14 participants highlighted the difficulty to select one relevant refactoring solution from a large set of non-dominated solutions and without offering any flexibility to update them. One example of received comments is *"The main advantage of this tool is instead of looking so many refactoring solutions manually this tool helps us to find the best solution based on objective selecting the center of the different clusters which provide the good refactoring recommendations."*

All the developers mentioned the high usability of the tool and the different options that are offered comparing to deterministic tools like JDeodorant. In addition, they did not appreciate a lot the long list of refactoring suggested by Ouni et al. and JDeodorant since they want to take control of modifying and rejecting some refactorings. In addition, the validation of this long list of refactorings is time-consuming. Thus, they appreciate that our tool suggests refactoring one by one and update the

list based on the feedback of developers. 13 participants commented on the minimum effort required to understand the impact of the proposed refactorings on the quality and to identify a relevant solution using the clusters comparing all the three remaining tools: *"Refactoring with clustering reduces the time of the analysis of the objectives. It keeps the similar type of classes or patterns in the same cluster and dissimilar patterns in another cluster."* All the participants found as well our tool helpful for both *floss refactoring*, to maintain a good quality design and also for *root canal refactoring* to fix some quality issues such as code smells.

4.5 Threats to Validity

Conclusion validity. The parameter tuning of the different optimization algorithms used in our experiments creates another internal threat that we need to evaluate in our future work. The parameters' values used in our experiments are found by trial-and-error [131]. However, it would be an interesting perspective to design an adaptive parameter tuning strategy [134] for our approach so that parameters are updated during the execution in order to provide the best possible performance.

Internal validity. The variation of correctness and speed between the different groups when using our approach and other tools such as JDeodorant. 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. 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.

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.

External validity. The first threat is the limited number of participants and evaluated systems, which externally threatens the generalizability of our results. In addition, our study was limited to the use of specific refactoring types. Future replications of this study are necessary to confirm our findings.

4.6 Conclusion

We proposed an interactive clustering-based recommendation tool for software refactoring that reduces the effort of improving the quality of software systems. The exploration of the non-dominated refactoring solutions is implicitly performed based on the interaction with the developers. The feedback received from the developers and the clustering of non-dominated refactoring solutions are used to reduce the search space and converge to better solutions.

To evaluate the effectiveness of our tool, we conducted an evaluation with 14 software developers who evaluated the tool and compared it with the state-of-the-art refactoring techniques. Our evaluation results provide strong evidence that our tool improves the applicability of software refactoring, and proposes a novel way for software developers to refactor their systems interactively with reasonable effort. Future work involves validating our technique with additional refactoring types, programming languages and programmers in order to conclude about the general applicability of our methodology. Furthermore, we only focused on the recommendation of refactorings. We plan to extend the interactive clustering-based approach to others related software maintenance problems such as regression testing and bugs localization. We will also work on making the refactoring recommendations more personalized based on the profile of programmers by learning their preferences.

CHAPTER V

From Multi-objective to Mono-objective Refactoring via Developers Preference Extraction

5.1 Introduction

Software restructuring, or refactoring [9], is critical to improve software quality and developers' productivity, but can be complex, expensive, and risky [96, 140, 141]. A recent study [142] shows that developers are spending over 50% of their time struggling with existing code (e.g. understanding, restructuring, etc.) rather than creating new code. As projects evolve, developer,s in the rush to deliver a new release, frequently postpone necessary refactorings until a crisis happens [95]. When that occurs, it often results in substantially degraded system performance, perhaps an inability to support new features, or even in a terminally broken system architecture and significant losses.

While code-level refactoring, such as *Move-Method, Pullup-Method*, etc, is widely studied and well-supported by tools [113, 114, 71, 23, 33, 115, 97, 116, 75, 47, 2], **understanding the *refactoring rationale***, or the preferences of developers, is still lacking and yet not well supported. In our recent survey, supported by an NSF I-Corps [1] project, with 127 developers at 38 medium and large companies (Google, eBay, IBM,

[1] https://www.nsf.gov/news/special_reports/i-corps

Amazon, etc.), 84% of face-to-face interviewees confirmed that most of the existing automated refactoring tools detect and recommend hundreds of code-level issues (e.g. anti-patterns and low quality metrics/attributes) and refactorings but do not specify where to start or how they can be relevant for their context and preferences. This observation is consistent with another recent study [74]. Furthermore, refactoring is a human activity that cannot be fully automated and requires developers' insight to accept, modify, or reject some of these recommendations because the developers understand the problem domain intuitively and may have a clear target design in mind. Several studies reveal that automated refactoring does not always lead to the desired architecture even when the quality issues are well detected, due to the subjective nature of software design [79, 2, 143, 97, 144, 75, 145]. However, manual refactoring can be error-prone and time-consuming [71, 146].

Few studies have been proposed, recently, to interactively evaluate refactoring recommendations by developers [75, 74, 135, 6, 36]. The developers can provide feedback about the refactored code and introduce manual changes to some of the recommendations. However, this interactive process can be expensive since developers must evaluate a large number of possible refactoring strategies/solutions and eliminate irrelevant ones. Both interactive and automated refactoring approaches have to deal with a big challenge to consider many quality attributes for the generation of refactoring solutions. Thus, refactoring studies either aggregated these quality metrics to evaluate possible code changes or treated them separately to find trade-offs [79, 74, 135, 2, 143, 97, 145, 70]. However, it is challenging to define upfront the weights for the quality objectives since developers are not able to express them upfront. Furthermore, the number of possible trade-offs between quality objectives is large which makes developers reluctant to look at many refactoring solutions due to the time-consuming and confusing process.

In this chapter, we propose an approach that takes advantage of both existing

categories of refactoring work. Thus, we propose, for the first time, a way to convert multi-objective search into a mono-objective one after few interactions with the developer. The first step consists of using a multi-objective search, based on the evolutionary algorithm NSGA-II [127], to generate a diverse set of refactoring strategies by finding a trade-off between several conflicting quality attributes. Then, an unsupervised learning algorithm clusters the different trade-off solutions, called the Pareto front, to guide the developers in selecting their region of interests and reduce the number of refactoring options to explore. Finally, the extracted preferences from the developer are used to transform the multi-objective search into a mono-objective one by taking the preferred cluster of the Pareto front as the initial population for the mono-objective search and generating an evaluation function based on the weights that are automatically calculated from the center of the preferred cluster in the Pareto front. Therefore, the developer will just interact with only one refactoring solution generated by the mono-objective search.

Our approach is taking the advantages of mono-objective search, multi-objective search, clustering and interactive computational intelligence. Multi-objective algorithms are powerful in terms of diversifying solutions and finding trade-offs between many objectives but generate many solutions as an output. The clustering and interactive algorithms are useful in terms of extracting developers' knowledge and preferences. Mono-objective algorithms are the best in terms of optimization power once the evaluation function is well-defined and generate only one solution as an output. We selected 32 active developers to manually evaluate the effectiveness of our tool on 6 open source projects and one industrial system. The results show that the participants found their desired refactorings faster and more accurate than the current state of the art. A tool demo of our interactive refactoring tool of this chapter and an appendix containing all the details of the experiments can be found in the following link [147].

5.2 Motivations

While successful tools for refactoring have been proposed, several challenges are still to be addressed to expand the adoption of refactoring tools in practice. To investigate the challenges associated with current refactoring tools, we conducted a survey, as part of an NSF I-Corps project, with 127 professional developers at 38 medium and large companies including eBay, Amazon, Google, IBM, and others. 112 of these interviews were conducted face-to-face.

The question we encounter most during our industrial collaborations in refactoring is *"We agree that this is a problem, but what should we do?"* Although code-level anti-patterns can largely be automated, higher-level refactoring —such as redistributing functionality into different components, decoupling a large code base into smaller modules, redesigning to a design pattern— requires abstractions determined by human architects. In these cases, the architect usually has a desired design in mind as the refactoring target, and the developer needs to conduct a series of low-level refactorings to achieve this target. Without explicit guidance about which path to take, such refactoring tasks can be demanding: It took a software company several weeks to refactor the architecture of a medium-size project (40K LOC) [36]. Several books [95, 96, 9] on refactoring legacy code and workshops on technical debt [148] present the substantial costs and risks of large-scale refactorings. For example, Tokuda and Batory [149] presented two case studies where architectural refactoring involved more than 800 steps, estimated to take more than 2 weeks.

Prior work [150] shows that even semi-automated tools for lower-level refactorings have been underutilized. Given that fully automatic refactoring usually does not lead to the desired architecture and that a designer's feedback should be included, we propose *an interactive architecture refactoring recommendation system* to integrate higher-level abstractions from humans with lower-level refactoring automation. Over 77% of the interviewees reported that the refactorings they perform do not match

the capabilities of low-level transformations supported by existing tools, and 86% of developers confirmed that they need better design guidance during refactoring: *"We need better solutions of refactoring tasks that can reduce the current time-consuming manual work. Automated tools provide refactoring solutions that are hard and costly to repair because they did not consider our design needs."*

Based on our extensive experience working on licensing refactoring research prototypes to industry, developers always have a concern on expressing their preferences upfront as an input for a tool to guide refactoring suggestions. They prefer to get insights from some generated refactoring solutions then decide which quality attributes they want to improve. However, several existing refactoring tools fail to consider the developer 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 applied. Determining which quality attribute should be improved and how is never a pure technical problem in practice. Instead, high-level refactoring decisions have to take into account the trade-offs between code quality, available resources, project schedule, time-to-market, and management support. Based on our survey, it is very challenging to aggregate quality objectives into one evaluation function to find good refactoring solutions since developers are not able, in general, to express their preferences upfront. Figure 5.1 shows an example of a Pareto front of non-dominated refactoring solutions improving the QMOOD quality attributes of a GanttProject generated using an existing tool [6]. QMOOD is one of the widely accepted software quality models in industry based on our previous collaborations with industry and recent studies [6, 36, 12, 151, 152].

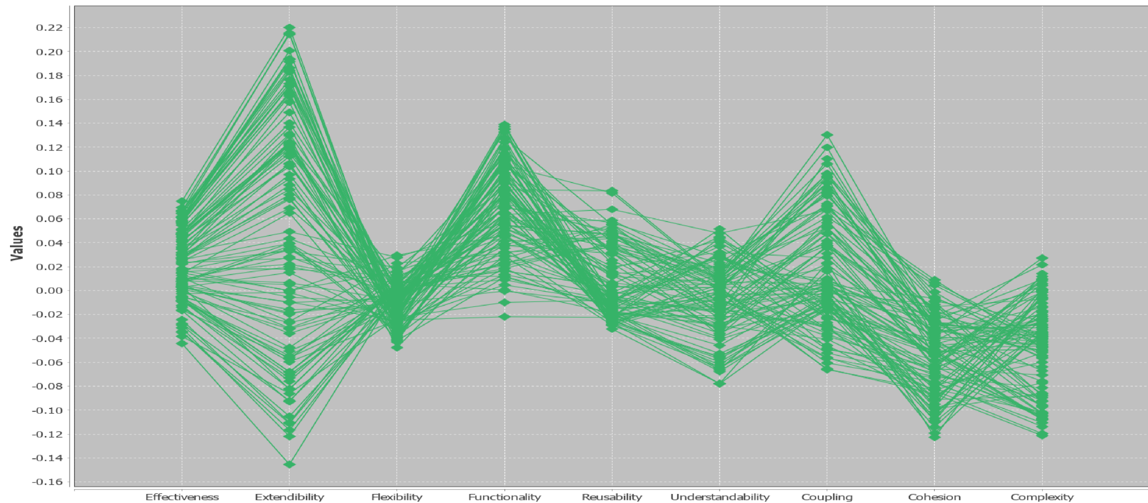


Figure 5.1: The output of a multi-objective refactoring tool[6] finding trade-offs between QMOOD quality attributes on GanttProject v1.10.2

While developers were interested to give a feedback for some of the refactoring solutions but they expected to see only one refactoring solution in the future after this interaction. This means after the first round of optimization and evaluation, the developer wants to have a single personalized solution. The extraction of developers' knowledge from the interaction data is beyond the scope of existing refactoring tools. Furthermore, existing search-based software engineering approaches did not explore converting multi-objective into mono-objective search after knowledge extraction. While multi-objective search algorithms are known to be good in diversifying solutions but they cannot beat well-formulated mono-objective search algorithms in terms of the optimization power.

5.3 Approach Overview

Our proposed approach includes three main phases. First, we use multi-objective optimization to find a set of non-dominated refactoring solutions capable of improving the quality of the software. Second, we cluster these solutions and obtain the center of each cluster to reduce the exploration effort of the Pareto-front by the decision

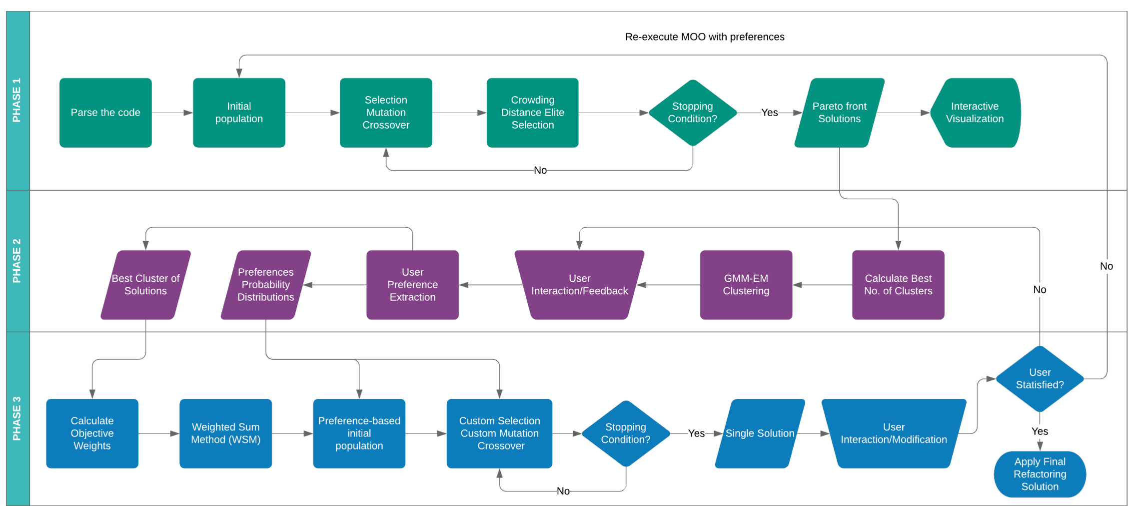


Figure 5.2: Overview of our proposed approach.

maker. Third, we extract automatically the preferences and utilize them to transform the multi-objective problem to a mono-objective one after the user's interaction and evaluation of the recommended refactoring solutions. Finally, the output of the mono-objective search is a single solution fitting to the user's expectations and preferences then the developer can interact with that solution if needed and continue the execution of the mono-objective algorithm until selecting a final refactoring solution. The pseudo code of our algorithm is described in the appendix [147]. In the following, we will explain, in details, the steps of our proposed technique which its overview is depicted in Fig 5.2

5.3.1 Phase 1: Multi-Objective Refactoring

Search-based software refactoring techniques need to investigate a large possible refactoring space which is the result of the variety of the refactoring operations as well as a combinatorial combinations of code locations, attributes, and methods.

Considering the goals and objectives of refactoring a software, this challenging task can be formulated as a multi-objective optimization problem as described in Subsection 2.2.2.

In multi-objective optimization, the quality of an optimal solution is determined by dominance. The set of feasible solutions that are not dominated with respect to each other is called *Pareto-optimal* or *Non-dominated* set.

We proposed an adaptation of the non-dominated sorting genetic algorithm (NSGA-II) [127] to interactively find refactoring solutions with a trade-off between multiple quality attributes in 3.2.

NSGA-II is a multi-objective evolutionary algorithm operating on a population of candidate solutions which are evolved toward the Pareto-optimal solution set. This method uses an explicit diversity-preserving strategy together with an elite-preservation strategy [127]. The complexity of NSGA-II is at most $O(MN^2)$ where M and N are the number of objectives and the population size, respectively.

Our proposed algorithm is described in Algorithm 4. We begin with an execution of the adapted NSGA-II [127] based on the encoded refactoring operations and quality objectives. A set of solutions, P_0, is formed as the initial population. Then, using the change operators, the offspring population, Q_0, is produced. Finally, a subset of solutions is selected from R_0 set which is the union of initial and offspring populations. This selection is based on domination rules and crowding distance where it guarantees that the already discovered non-dominated solution to be kept for the future generation. This process is iterated until the stopping criteria are met.

The result of the first phase of our approach, as it is shown in the Figure 5.1, is a set of Pareto-optimal refactoring solutions. In the following subsections, we briefly summarize the adaptation of multi-objective search to the software refactoring problem.

Algorithm 4: Interactive Preference-based Multi-objective to Mono-objective Refactoring (IPMM)

```
Input  : Population Size (N), Source Code
Output: Recommended Pareto-optimal Solutions
1   UserPreferences ← ∅ ;        /* Initiate Preference Parameters */
2   while ¬ The user is satisfied do
phase1  begin Multi-objective Refactoring
4           P_1 ← InitializePopulation(N);
5           EvaluateObjectives(P_1);
6           FastNonDominatedSort(P_1);
7           Q_1 ← SelectCrossoverMutate(P_1);
8           while ¬StoppingCondition() do
9               EvaluateObjectives(Q_1);
10              R_t ← P_1 ∪ Q_1;
11              Fronts=FastNonDominatedSort(R_t);
12              P_{t+1} ← ∅;
13              i ← 1 ;
14              while |P_{t+1}| + |Front_i| ≤ N do
15                  CrowdingDistanceAssign(Front_i);
16                  P_{t+1} ← P_{t+1} ∪ Front_i;
17                  i ← i + 1;
18              SortByRankAndDistance(Front_i);
19              P_{t+1} ← P_{t+1} ∪ Front_i[1 : (N − |P_{t+1}|)];
20              Q_{t+1} ← SelectCrossoverMutate(P_{t+1}) ;
21              t = t + 1;
22          ParetoFront + = Q_{t+1};
phase2  begin Pareto Front Clustering and User Interaction
24          GMMClustering (ParetoFront);   /* Described in Section 3.3
             */
25          ClustersCenter ();
26          GetUserFeedBack (Clusters,Centers);
27          UserPreferences ← ExtractPreferences ();
phase3  begin Preference-based Mono-objective Optimization
29          W ← CalculateObjectivesWeight(UserPreferences);
             /* Described in Section 3.4 and Algorithm 1 */
30          RecommendedSolution ←
             MonoObjectiveOptimization(W, UserPreferences);
31  Return RecommendedSolution;
```

5.3.1.1 Solution Representation

We encode a refactoring solution as an ordered vector of multiple refactoring operations. Each operation is defined by an action (*e.g.* move method, extract class, etc.) and its specific controlling parameters (e.g. source and target classes, attributes, methods, etc.). We considered a set of the most important and widely used refactorings in our experiments: Extract Class/SubClass/SuperClass/Method, Move Method/Field, PullUp Field/Method, PushDown Field/Method, Encapsulate Field and Increase/Decrease Field/Method Security. These refactoring operations are described in Table 2.1.

During the process of population initialization or mutation operation of the algorithm, the refactoring operation and its parameters are formed randomly. Therefore, due to the random nature of the process, it is crucial to evaluate the feasibility of a solution meaning to preserve the software behavior without breaking it. This evaluation is based on a set of specific pre- and post-conditions for each refactoring operation [10].

The length of an encoded refactoring vector is selected randomly from a predefined range.

5.3.1.2 Fitness Functions

The fitness function is the essential aspect of an optimization problem where we strive to find the best quality value for the given fitness function. It is used to evaluate the goodness of a candidate solution in terms of maximization or minimization. There are two crucial factors for a fitness function: discrimination degree between individuals of a population and calculation speed.

We used the QMOOD [46] as a means of estimating the effect of a refactoring operation on the quality of a software.

This model is described in Subsection 2.2.3 and 2.2 and 2.3. We considered the

relative change of six quality attributes (Understandability, Functionality, Reusability, Effectiveness Flexibility, Extendibility) after applying a refactoring solution as the fitness function.

5.3.2 Phase 2: Clustering Refactoring Solutions and Extracting Developer Preferences

One of the most challenging and tedious tasks for the user during every multi-objective optimization process is the decision making. Since many Pareto-optimal solutions are offered, it is up to the user to select among them which requires exploration and evaluation of the Pareto-front solutions.

The main goal of this step is to cluster and categorize the solutions based on their similarity in the objective space. These clusters of solutions help the user to have an overview of the possible existing options. Therefore, this technique gives the user a more clear initial step of exploration where she can initiate the interaction by evaluating each cluster center or representative member. Based on our previous refactoring collaborations with industry, developers are always highlighting the time consuming and confusing process to deal with the large population of Pareto-front solutions: "where should I start to find my preferred solution?". This observation is valid for various SBSE applications using multi-objective search [36].

5.3.2.1 Clustering the Pareto-front

Clustering is an unsupervised learning method to discover a meaningful underlying structure and pattern between a set of unlabelled data. It puts the data into groups where the similarity of the data points within each group is maximized while keeping a minimized similarity between the groups.

Determining the optimal number of clusters is a fundamental issue in clustering techniques. One of the methods to overcome this issue is to optimize a criterion where

we try to minimize or maximize a measure for the different number of clusters formed on the data set. For this purpose, we utilized Calinski Harabasz (CH) Index which is an internal clustering validation measure based on two criteria: compactness and separation [137]. CH assesses the clustering outcomes based on the average sum of squares between and within clusters. Therefore, we execute the clustering algorithm on the Pareto-front solutions with a various number of components as the input. The CH score is calculated for each execution, and the result with the highest CH score is recognized as the optimal way of clustering our data.

After determining the best number of clusters, we employed a probabilistic model-based clustering algorithm called "Gaussian Mixture Model" (GMM). GMM is a soft-clustering method using a combination of Gaussian distributions with different parameters fitted on the data. The parameters are the number of distributions, Mean, Co-variance, and Mixing coefficient. The optimal values for these parameters are estimated using Expectation-Maximization (EM) algorithm [138]. EM trains the variables through two steps iterative process.
After the convergence of EM, the membership degree of each solution to a fitted Gaussian or cluster is kept for preference extraction step. Furthermore, in order to find a representative member of each cluster, we measure the corresponding density for each solution and select the solution with the highest density value.

The line chart of Pareto-front solutions after clustering is shown in Figure 5.3. Compared to the original chart in Figure 5.1, the color of each line indicates its cluster and the solutions marked with triangles are the cluster representative member.

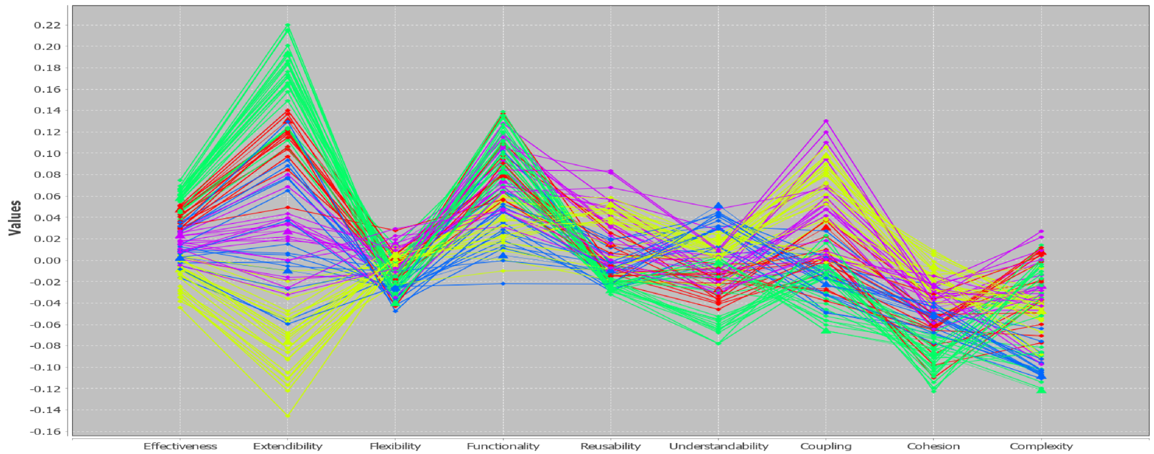


Figure 5.3: The output of phase 2 (Clustering) on GanttProject v1.10.2.

5.3.2.2 Interaction and Preference Extraction

The results of multi-objective refactoring after clustering are presented to the user in various interactive tables and charts alongside with extensive analysis to explain and guide the process of decision making. These explanations are automatically generated using statistical analysis and investigating the content of the solutions and clusters.

The explanations of Pareto-front assist the user to gain a vibrant picture of the available options, costs, and benefits. Furthermore, by clustering similar solutions, it requires less effort to initiate the exploration and finally making a decision.

The user may begin to evaluate the cluster center solutions or expand the search to the other solutions in the cluster. The interaction can be performed at the cluster, solution, and refactoring operation levels depending on the user's desire. The feedback is quantified to a continuous score in the range of [-1,1].

The developer can evaluate a solution by modifying its refactoring operations (edit, add, delete, re-order) or just rate the whole solution or cluster. After the developers interaction, Solution score ($Score_{s_i}$) and Cluster score ($Score_{c_k}$) are computed as the average score of operations in a solution and the average score of solutions in a cluster,

respectively.

The cluster of solutions with the highest score is considered as the region of interest in the solution space. It indicates the preferred objectives, code locations, and refactoring operations. For instance, if the solutions in the selected cluster tend to emphasize on improving Extendibility by applying mostly Generalization category of refactoring operations on certain packages or classes of the software, we consider these factors as the user preferences in the execution of the next phase of our approach.

For this purpose, we compute the weighted probability of refactoring operations (RWP) and target classes of the source code (CWP) as follow:

$$RWP_p = \frac{\sum_{s_i \in c_j} \gamma_{ij} \times (|r_p \in s_i|)}{\sum_{r_m \in Ref} \sum_{s_i \in c_j} \gamma_{ij} \times (|r_m \in s_i|)} \tag{5.1}$$

$$CWP_q = \frac{\sum_{s_i \in c_j} \gamma_{ij} \times (|cl_q \in s_i|)}{\sum_{cl_m \in Cls} \sum_{s_i \in c_j} \gamma_{ij} \times (|cl_m \in s_i|)} \tag{5.2}$$

where j is the index of selected cluster, s_i is the solution vector, γ_{ij} is the membership weight of solution i to the cluster j, r is refactoring action, Ref is the set of all refactoring operations, and Cls is the set of all classes in the source code.

5.3.3 Phase 3: Preference-based Mono-objective Refactoring

One of the main contributions of this chapter is the ability to convert a multi-objective algorithm into a mono-objective one after interacting with the developer to extract his preferences and knowledge. Mono-objective algorithms are known to be the best in terms of optimization but require that the fitness function should be well defined based on the decision maker's preferences. The Multi-objective Evolutionary Algorithm used in Phase 1 might not provide high-quality solutions in the region of interest of the developer because of the high dimensionality nature of the problem and the need to find trade-offs. Therefore, it is important to consider the user preferences

extracted in Phase 2.

The goal of this phase is to use the preferences extracted from the developer after the multi-objective optimization to transform the problem into a single objective optimization problem by aggregating objectives according to the user's preferences. This transformation gives the decision maker a single solution in the region of interest. Consequently, our proposed approach is a combination of all three categories of preference-based search where the preferences are expressed after the first evolutionary process, then they are incorporated to guide the single objective optimization.

One way to convert a multi-objective optimization problem to a mono-objective problem and achieve a single solution is called the Weighted Sum Method (WSM). In this method, the single preference fitness function is computed as a linear weighted sum of multiple objectives. The main drawback of the WSM method is that it needs the weights parameters to be given. Fortunately, in our case, those parameters are computed automatically from the decision maker preferences of the interactive optimization process (preferred cluster) in the objectives space (quality attributes). Thus, the weight of one or more objectives can get the value 0 (or almost) if the selected cluster by the developer penalized them while favoring other objectives. Also, the WSM is not computationally expensive unlike the other scalarization methods. Therefore, the optimization problem can be formulated as:

$$
\begin{aligned}
&Minimize && PF(\boldsymbol{X}) = \sum_{i=1}^{M} \omega_i f_i(x), \\
&Subject\ to && \boldsymbol{X} \in S, \\
& && \omega_i \geq 0; \sum_{i=1}^{M} \omega_i = 1;
\end{aligned}
$$

where $PF(X)$ is the single scalar preference function, and weights ω_i reflects the a priori preferences of the user over the objectives. The weights are a tool to steer

```
Algorithm 5: Preference-based Mono-objective Optimization
Input  : Preferences (P),
         Preferred Cluster (PC),
         Cluster Center (CC)
Output: RecommendedSolution

begin Calculating Objective's Weight
    NormalizeAll(PC);
    W_i ← NormalizeUnitSum(CC);
begin Mono-objective Optimization
    initialPopulation ← PC;
    if size(initialPopulation) ¡ N then
        initialPopulation + = fillPopulation();
    while ¬ stoppingCondition() do
        customSelection();
        Crossover();
        customMutation();
        fitness ← weightedSum(f_i, w_i);
        evaluate(fitness);
    RecommendedSolution ← getFittest();
Return RecommendedSolution;
```

the search along the Pareto-front into a direction determined by the user. This way, the decision maker is offered a single solution that corresponds to his interests and reduces on him the burden of having to go through multiple solutions.

In order to solve the converted mono-objective problem, we adopted a standard Genetic Algorithm (GA). To adapt the GA algorithm to our refactoring context, we use the same solution representation and quality fitness functions as reported in phase 1. Algorithm 5 explains the steps of this phase.

We begin by normalizing the values of each fitness function separately for all solutions in the preferred cluster. Then, we pick the center of the cluster and normalize this solution's fitness values. We use the result as the aggregation weights in WSM where the condition $\sum_{i=1}^{M} \omega_i = 1$ is satisfied. Therefore, we assign the importance of the objectives accordingly based on the intuition and preferences of the user.

The obtained single fitness function is employed to evaluate the solutions in the

execution of adapted GA. We consider the preferences extracted in the previous phase, to customize the components of GA via Preference-based initial population generation and Preference-based Mutation/Selection operators.

Instead of generating the initial population randomly, we acquire the user preferred cluster as the elite set of solution from which the search process is initiated. Thus, we do not generate solutions randomly for the mono-objective GA but we take the solutions in the preferred cluster as the initial population thus we do not lose the knowledge extracted from the developer. Since the number of solutions in the preferred cluster might be less than the required size, we form new individuals to fill the gap. The new solutions are produced based on CWP and RWP probability distribution. It means, for each new solution, we pick the operation and its target class attribute from a distribution aligned with the preferences of the user.

The preference probability distribution for code locations and refactoring operations are used during the mutation process similarly.

The selection operator which is used to keep the most valuable solutions of the population is customized to consider the distance of a solution to the region of interest. Therefore, being closer to the preferences and having higher fitness value are both measured to be factors of selecting an elite solution. Finally, the solutions are evaluated via the preference function aggregated from multiple objectives. When the stopping condition is satisfied, the single optimal solution is recommended to the user. Similar to Phase 1, the user can interact with this solution via editing/adding/removing the refactoring operations.

If the developer is still not satisfied, he can proceed with the search process in two ways: 1) going back to Phase 2 and selecting another cluster. 2) returning to Phase 1 and executing the multi-objective optimization again where, in this time, the approach is customized to accommodate the prior knowledge of the preferences. The result of Phase 3 is represented in Figure 5.4. As it is shown, at this step, the user

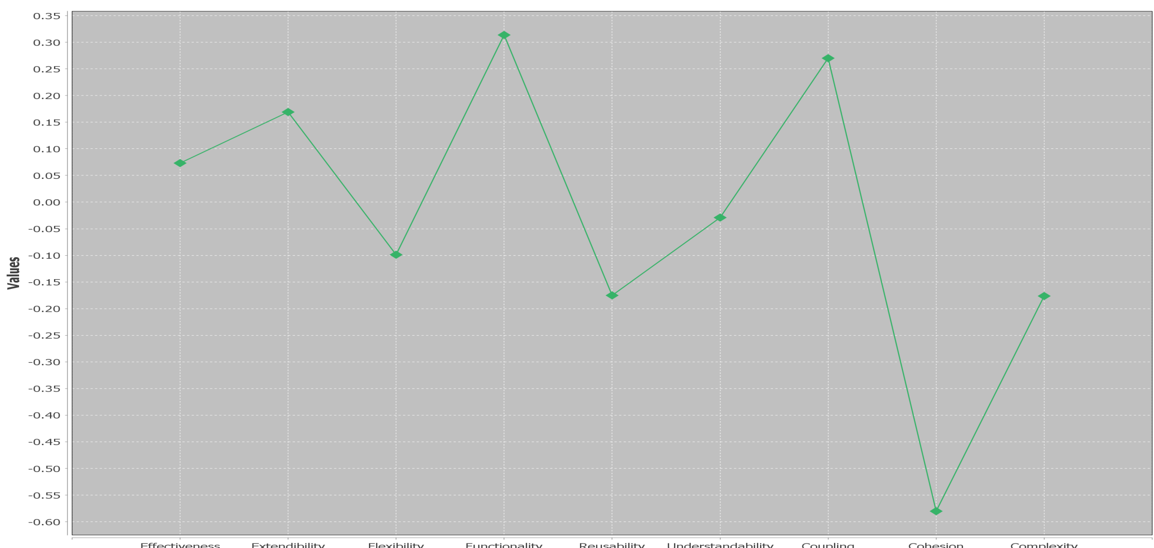


Figure 5.4: The output of phase 3 (Mono-objective) on GanttProject v1.10.2 system.

is required to only interact with one customized solution where it takes shorter effort and time and produces less confusion.

5.4 Evaluation

5.4.1 Research Questions

We defined three main research questions to measure the correctness, relevance and benefits of our interactive clustering-based multi-objective refactoring tool comparing to existing approaches that are based on interactive multi-objective search [75], fully automated multi-objective search (Ouni et al.) [2] and fully automated deterministic tool not based on heuristic search (JDeodorant) [5]. A tool demo of our interactive refactoring tool and supplementary appendix materials (questionnaire, setup of the experiments, statistical analyses, and detailed results) can be found in our study's website [2]. The appendix includes:(a) Study-steps; (b) Pre/Post-study-questionnaires (QMOOD, experience, comments, etc.); (c) Parameters-tuning;(d)

[2]Demo and supplementary appendix materials can be found in the following link: https://sites.google.com/view/scam2019

Box-plots/statistical-tests to give more details than the median.

The research questions are as follows:

- **RQ1: Benefits.** To what extent can our approach make relevant recommendations for developers compared to existing refactoring techniques?

- **RQ2: The relevance of developers' knowledge extraction.** To what extent can our approach reduce the interaction effort, comparing to existing refactoring techniques, while quickly identifying relevant refactoring recommendations?

- **RQ3: Tool usefulness.** How do developers evaluate the relevance of our tool in practice (post-study survey)?

5.4.2 Experimental Setup

We considered a total of seven systems summarized in Table 5.1 to address the above research questions. We selected these seven systems because of their size, have been actively developed over the past 10 years and extensively analyzed by the competitive tools considered in this work. UTest[3] is a project of our industrial partner used for identifying, reporting and fixing bugs. We selected that system for our experiments since five developers of that system agreed to participate in the experiments and they are very knowledgeable about refactoring (they are part of the maintenance team). Table 5.1 provides information about the size of the subject systems (in terms of number of classes and KLOC).

To answer RQ1, we asked a group of 32 participants to identify and manually evaluate the relevance of the refactoring solutions that they selected using four other tools. The first tool is an existing interactive multi-objective refactoring approach proposed by Mkaouer et al. *et al.* [75, 6] but the interactions were limited to the

[3] Company anonymized.

Table 5.1: Statistics of the studied systems.

System	**Release**	**#Classes**	**KLOC**
ArgoUML	v0.3	1358	114
JHotDraw	v7.5.1	585	25
GanttProject	v1.10.2	241	48
UTest	v7.9	357	74
Apache Ant	v1.8.2	1191	112
Azureus	v2.3.0.6	1449	117
JFreeChart	v1.0.9	521	170

refactorings (accept/reject) and there is no clustering of the Pareto front or learning mechanisms from the interaction data. The second tool is an interactive clustering based multi-objective approach proposed by Alizadeh et al. *et al.* [36] however they did not consider the developers' knowledge extraction neither the use of mono-objective search to directly converge towards one refactorings solution after extracting developers preferences. The comparison with these tools will help us evaluating the main new contribution of this chapter related to converting multi-objective to a mono-objective one after extracting the developers' preferences from exploring the clusters and the Pareto front. We have also compared our IMMO approach to two fully-automated refactoring tools by means of Ouni *et al.* [2] and JDeodorant [5]. Ouni *et al.* [2] proposed a multi-objective refactoring formulation based on NSGA-II that generates a solution to maximize the design coherence and refactorings reuse from previous releases. JDeodorant [5] is an Eclipse plugin to detect bad smells and apply refactorings. 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 these refactorings. We used these two competitive tools to evaluate the benefits of the interaction feature in helping developers identifying relevant refactorings especially with the preferences extraction feature and the mono-objective search.

We preferred not to use the antipatterns and internal quality indicators as proxies for estimating the refactorings relevance since the developers manual evaluation already includes the review of the impact of suggested changes on the quality. Fur-

thermore, not all the refactorings that improve any quality attributes are relevant to the developers, which is one of the main motivations of this work. The only rigorous way to evaluate the relevance of our tool is the manual evaluation of the results by active developers. This manual evaluation score, MC, consists of the number of relevant refactorings identified by the developers over the total number of refactorings in the selected solution.

Unlike fixing bugs, refactoring is a very-subjective activity and there is no unique solution to refactor a code/design thus it is very difficult to construct a gold-standard for large-systems which makes calculating the recall very challenging. Does the deviation from an expected refactoring solution means that the recommendation is wrong or simply another way to refactor the code? The context of our work is related to "incremental" refactoring rather than the rare "root canal" refactoring where developers will look at the whole architecture/system to make major refactorings. In this context of incremental refactoring, the main factor is the precision. In addition, developers can check via our tool the impact of the refactoring solutions on the overall code quality using many attributes. Thus, they continue to interactively evaluate and apply refactorings until that they are satisfied in terms of improving the quality attributes that they consider them concerning. Our tool enables the developers to evaluate the current quality of the system then tuning the search algorithm to focus on specific locations of the code based on their needs. With the current large-size of the systems, it is unrealistic to look for all possible refactoring strategies targeting the whole project which is not also the scope of this chapter(root-canal refactoring).

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. Although the vast majority of participants are already familiar with refactoring as part of their job and graduate studies, all the participants

Table 5.2: Selected programmers.

System	#Subjects	Avg. Prog. Exp.	Avg. Refactoring Exp.
ArgoUML	5	7.5	Very High
JHotDraw	5	8	Very High
Azureus	5	9.5	High
GanttProject	5	7	High
UTest	5	15.5	Very High
Apache Ant	5	9	Very High
JFreeChart	5	7	Very High

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 5.2, 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 up-to two out of the five tools on up-to two different systems to avoid the 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 are related to the five participants from the industrial partner where they agreed to evaluate only the industrial software. We assigned the tasks to the participants according to the studied systems, the techniques to be tested and developers' experience. Each of the five tools has been evaluated at least one time on each of the seven systems. 3 out of 32 participants were asked to refactor two projects to ensure that all the seven projects are refactored using the five different tools. To mitigate the training threat, the counter-balanced design ensured that these three participants: (1) did not evaluate the same system using two different tools; (2) did not evaluate the same tool more than one time (even on different projects) and(3) did not evaluate the same type of technique more than one time. Thus, if the participant used a multi-objective tool, then he/she will evaluate JDeodorant (deterministic) on another project.

To answer RQ2, we measured the time (T) that developers spent to identify the best refactoring strategies based on their preferences and the number of refactorings

(NR). Furthermore, we evaluated the number of interactions (NI) required on the Pareto front comparing to the one required once the mono-objective search is executed. This evaluation will help to understand if we efficiently extracted the developer preferences after the Pareto-front interactions. For this research question, we decided to limit the comparison to only the interactive multi-objective work of Mkaouer et al. [75, 6] and Alizadeh et al. [36] since they are the only ones offering interaction with the users and it will help us understand the real impact of the knowledge extraction and mono-objective features (not supported by existing studies) on the refactoring recommendations and interaction effort.

To answer RQ3, we collected the opinions of participants based on a post-study questionnaire. To better understand subjects' opinions with regard to usefulness and usability of our approach in a real setting, the post-study questionnaire was given to each participant after completing the refactoring tasks using our approach and all the techniques considered in our experiments. The questionnaires collected the opinions of the participants about their experience in using our tool compared to the remaining tools used in these experiments and their past experience.

The stopping criterion was set to 100,000 evaluations for all search algorithms in order to ensure fairness of comparison (without counting the number of interactions since it is part of the users' decision to reach the best solution based on his/her preferences). The mono-objective search was limited to 10,000 evaluations after the interactions with the user. The other parameters' values are as follows for both the multi-objective and mono-objective algorithms: crossover probability = 0.4; mutation probability = 0.7 where the probability of gene modification is 0.5. 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's values have been experimented.

5.4.3 Results

Results for RQ1: Benefits. Figure 5.5 summarizes the manual validation results of our IMMO approach comparing to the state of the art as evaluated by the participants. It is clear from the overall results that interactive approaches generated much more relevant refactorings to the programmers comparing to the automated tools of Ouni et al. and JDeodorant. Among the interactive approaches, IMMO outperformed the existing interactive approaches of Mkaouer et al. and Alizadeh et al. which may confirm the importance of extracting the developers' preferences and the performance of mono-objective search in terms of optimization when the fitness function is well-defined based on knowledge extraction from the user. On average, for all of our seven studied projects, 89% of the proposed refactoring operations are considered to be useful by the software developers of our experiments. The remaining approaches have an average of 83%, 71%, 67%, and 56% respectively for Alizadeh et al. (interactive with clustering), Mkaouer et al. (interactive multi-objective approach), Ouni et al. (fully automated multi-objective approach) and JDeodorant (deterministic non-search based approach). The highest MC score is 96% for the Azureus project, and the lowest score is 86% for JHotDraw. The participants were not guided on how to interact with the systems, and they mainly looked to the source code to understand the impact of recommended refactorings.

When comparing manually the results of the different tools, we found that automated refactorings generate a lot of false positive and noise of developers. Both Ouni et al. and JDeodroant tools recommended a large number of refactorings comparing the interactive tools where several of them are not interesting for the context of the developers thus they reject them even if they are correct. For instance, the developers of the industrial partner rejected several recommendations from these automated tools simply because they are related to a stable code or code fragments out of their interests. The majority of them will not change a code out of their ownership as well.

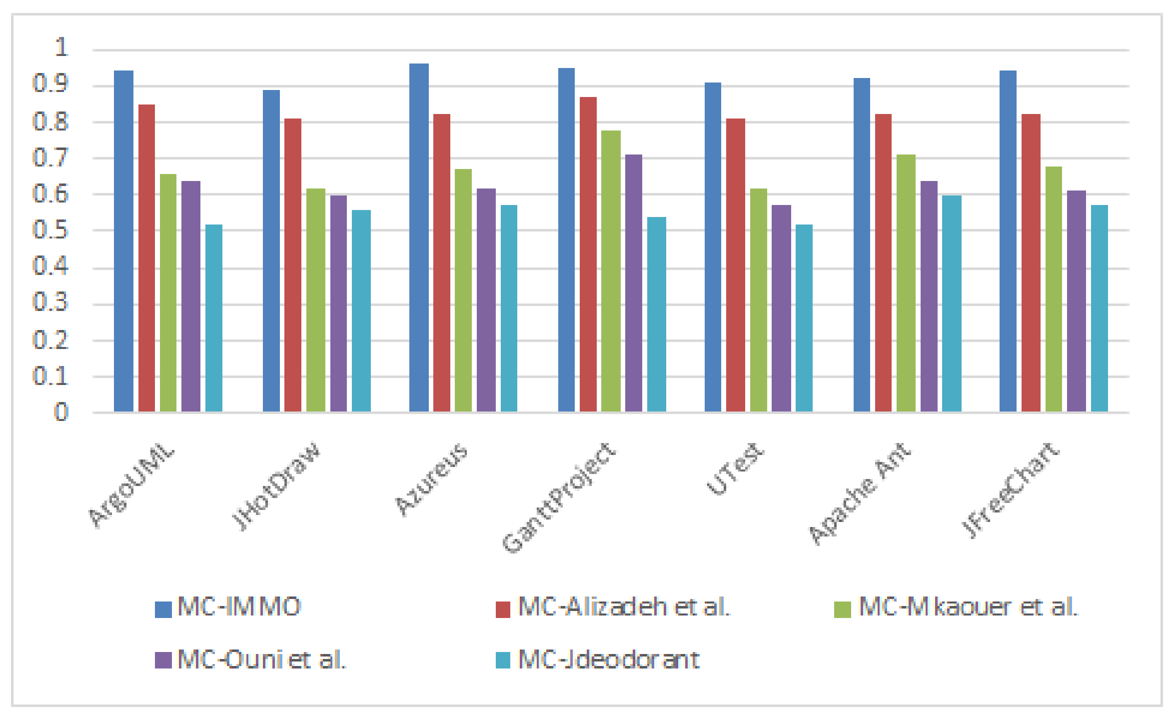


Figure 5.5: Average manual evaluations, MC, on the 7 systems.

Furthermore, they were not interested to blindly change anything in the code just to improve quality attributes. Comparing to the remaining interactive approaches, we found that some of the refactoring solutions of IMMO will never be proposed by Mkaouer et al. or Alizadeh et al. since they are emphasizing specific objectives than others. In fact, one of the main challenges of multi-objective search is the noise introduced by sacrificing some objectives and trying to diversify the solutions. Thus, the use of mono-objective search when the preferences of the user are extracted is powerful both in terms of interaction and optimization. The mono-objective search helped to focus on specific code locations and quality attributes rather than wasting the optimization power on multiple objectives. To conclude, our IMMO approach outperformed the four remaining refactoring approaches in terms of recommending relevant refactoring solutions for developers (RQ1).

Results for RQ2: The relevance of developers' knowledge extraction. Figures 5.6, 5.7 and 5.8 give an overview about the number of refactorings of the

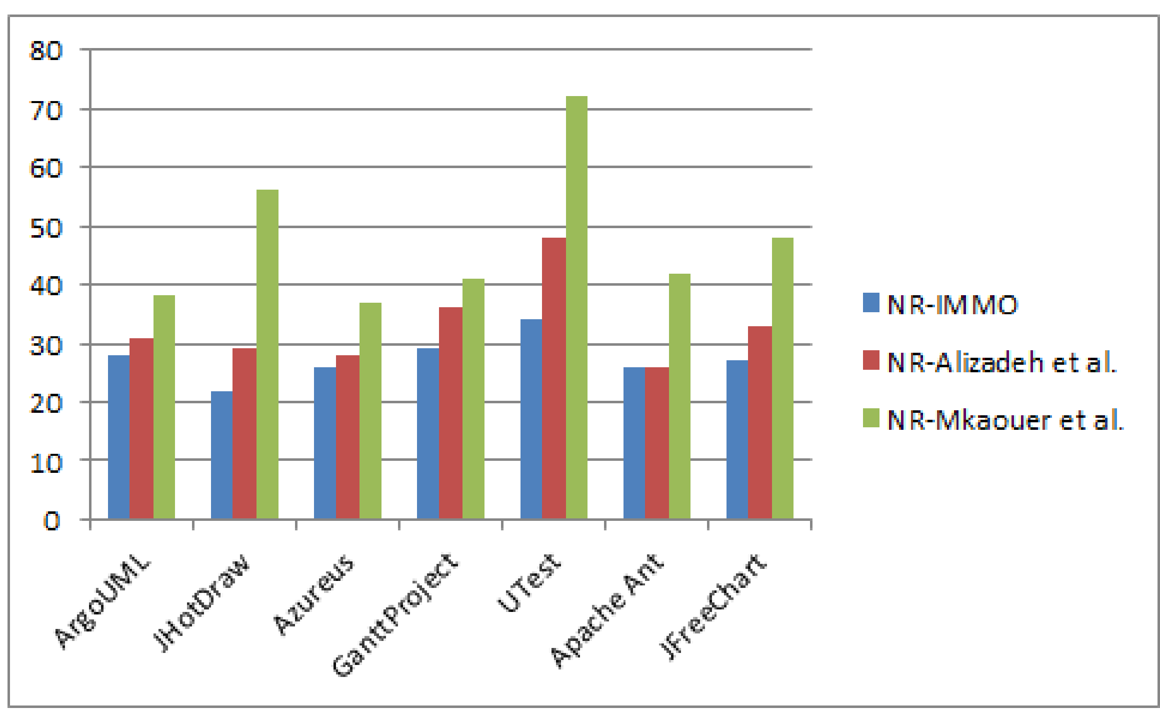


Figure 5.6: The median number of recommended refactorings, NR, of the selected solution on the 7 systems.

selected solution, number of required interaction and the time, in minutes, using our tool, the interactive clustering approach of Alizadeh et al., and the interactive multi-objective approach of Mkaouer et al. Based on the results of Figure 5.6, it is clear that our approach significantly reduced the number of recommended refactorings comparing to both other interactive approaches while increasing the manual correctness as described in RQ1. The highest number of refactorings was observed on the industrial system with 34 refactorings using IMMO, 48 using Alizadeh et al. and 72 refactorings using Mkaouer et al. It may be explained by the size and the quality of this system along with the fact that it was evaluated by some of the original developers of UTest. The lower number of recommended refactorings using IMMO comparing to interactive approaches is mainly related to the elimination of the noise in multi-objective search to handle multiple quality attributes and the extraction of developers preferences. It is normal to see fewer refactorings when the search space is reduced which was the case of IMMO.

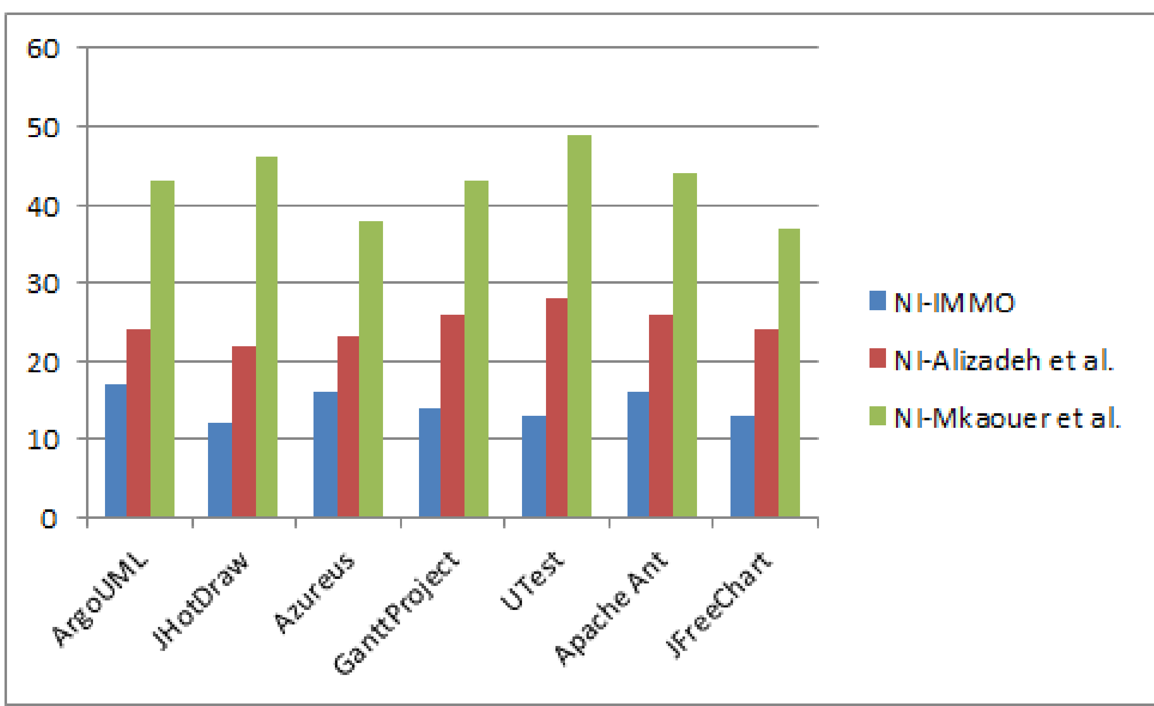


Figure 5.7: The median number of required interactions (accept / reject/ modify / selection), NI, on the 7 systems.

Figure 5.7 shows that IMMO required much fewer developer interactions than the remaining interactive approaches. For instance, only 13 interactions to modify, reject and select refactorings were observed on JFreeChart using our approach while 24 and 37 interactions were needed respectively for Vahid et al. and Mkaouer et al. The reduction of the number of interactions are mainly due to the move from multi-objective to mono-objective search after one round of interactions since the developers will not deal anymore with a set of solutions in the front but only one.

The participants also spent less time to find the most relevant refactorings on the different systems compared to the remaining interactive approaches. For instance, the average time is reduced by over 65% comparing to Mkaouer et al. for the case of JHotDraw (from 62 minutes to just 21 minutes). The time includes the execution of the multi-objective and mono-objective search (if any), the clustering (if any) and the different phases of interaction until the developer is satisfied with a specific solution. The drop of the execution time is mainly explained by the fast execution of the

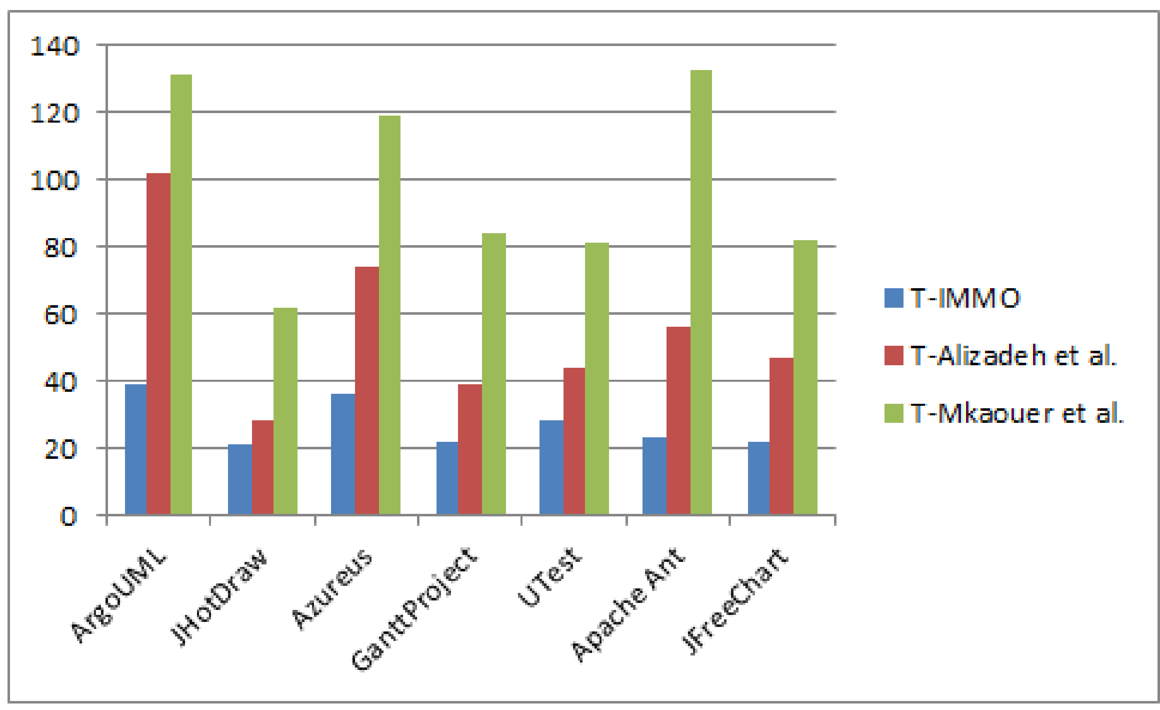


Figure 5.8: The average execution time, T, in minutes on the 7 systems.

mono-objective search and the reduced search space after the interactions with the developers.

Figure 5.9 shows a qualitative example extracted from our experiments using IMMO on the GanttProject based on the four interaction phases. After the generation of the Pareto front, the clustering algorithm of the non-dominated refactoring solutions identified three different main clusters for the two objectives selected by the developer (extendibility and effectiveness). During the first phase, the developer selected the cluster with id 0 as the preferred one after exploring several refactoring solutions in that cluster including mainly the solution located at the center of the cluster. Thus, the next phase took the solutions in the id 0 cluster and generated an initial population for the mono-objective genetic algorithm, and the center of the selected cluster was used to generate the weights for the fitness function. The output of the mono-objective search is one refactoring solution (instead of many solutions like the multi-objective search) that optimize better the selected objectives than all the

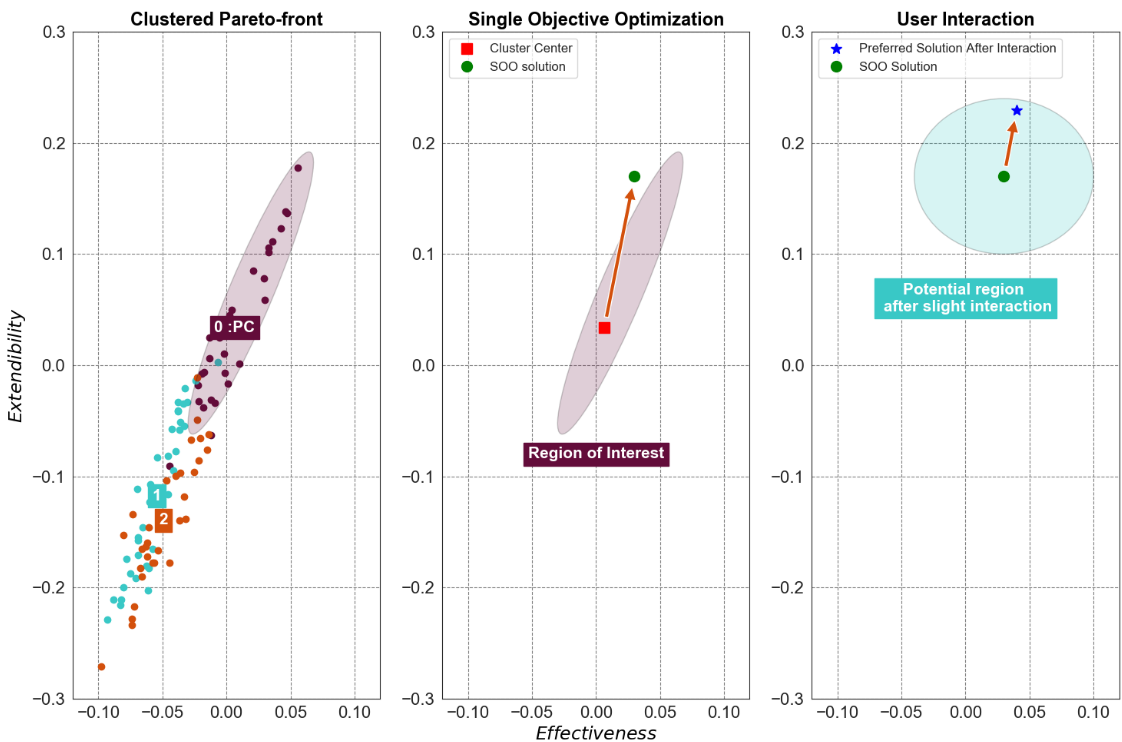


Figure 5.9: A qualitative example of three executions extracted from our experiments on GanttProject to illustrate the process of converting a multi-objective search into a mono-objective one.

solutions in the preferred cluster. Finally, the interactions with the user (accept/reject/modify some refactorings) on that solution helped to converge towards a better final solution by continuing the execution of the mono-objective search.

Results for RQ3: Impact. We did a post-study questionnaire to collect the feedback of the developers about the different evaluated refactoring tools. We found that 26 out the 32 participants highlighted that they preferred IMMO comparing to the remaining tools because of mainly the ability to interact with one solution (instead of a front) and the fast improvement of the refactoring results after just a few interactions. One of the participants submitted the following message: *"It is really great to see only refactoring solutions meeting my needs after just a couple of interactions!"*.

21 out the 32 participants appreciated the combination of multi-objective and

mono-objective search algorithms. They found that multi-objective search was useful to get some insights about several possible strategies to improve the code then the mono-objective powerful in generating better solutions based on their feedback. For instance, one of the developers commented the following: *"I had no idea about the beginning from where to start but looking to the first set of recommendations and their code impact, I had a clear idea on what quality metrics I need to target then it was easy to just give feedback to only one strategy (solution)."* 29 out the 32 participants found that the major refactoring suggestions of both Ouni et al. and JDeodorant hard to evaluate and understand. They found the lack of interactions as a main limitation since they have to accept or reject the whole refactoring suggestions and it is difficult to estimate their impacts. The participants noticed, in the survey, that they were satisfied with the the considered quality attributes and refactoring types by our tool. They did not suggest to add new types of refactoring or quality attribute.

5.5 Threats to Validity

Conclusion validity. 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. Another conclusion threat is the number of interactions with the developers since we did not force them to use the same interaction effort which may sometimes explain the out-performance of our approach. However, the participants were given the same maximum amount of time to use the tool (limited to 3 hours).

Internal validity. The variation of correctness and speed between the different groups 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 groups 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.

External validity. The first threat is the limited number of participants and evaluated systems, which externally threatens the generalizability of our results. In addition, our study was limited to the use of specific refactoring types and quality attributes. Furthermore, we mainly evaluated our approach using NSGA-II and GA algorithms, but other state-of-the-art metaheuristic algorithms can be used. Future replications of this study are necessary to confirm our findings.

5.6 Conclusion

In this chapter, we proposed a novel approach to extract developers' knowledge and preferences to find good refactoring recommendations. We combined the use of multi-objective search, clustering, mono-objective search and users interaction in our approach. To evaluate the effectiveness of our tool, we conducted an evaluation with 32 software developers who evaluated the tool and compared it with the state-of-the-art refactoring techniques. Our evaluation results provide strong evidence that our tool improves the relevance of recommended refactoring, helped developers to quickly find relevant refactorings and successfully extracted developers' knowledge and preferences.

As part of our future work, we are planning to evaluate our approach on further projects and a more extensive set of participants. We will also adapt our approach

to address other problems requiring developer interactions such as bugs localization.

The exploration of the non-dominated refactoring solutions is implicitly performed based on the interaction with the developers. The feedback received from the developers and the clustering of non-dominated refactoring solutions is used to reduce the search space and converge to better solutions. Once the developer selected the right cluster(s) and provided sufficient feedback interactions, the multi-objective search is converted into a mono-objective one by selecting the solutions in the preferred cluster as the initial population and using the center of the cluster to generate the weights for the fitness function.

CHAPTER VI

Simultaneous Decision and Objective Space Clustering for Interactive Refactoring

6.1 Introduction

With the ever-growing size and complexity of software projects, there is a high demand for efficient refactoring [9] tools to improve software quality, reduce technical debt, and increase developer productivity. However, refactoring software systems can be complex, expensive, and risky [96, 140, 141]. A recent study [142] 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 to fully automated techniques [113, 114, 71, 23, 33, 115, 97, 116, 75, 47, 2]. While these tools are successful in generating correct code refactorings, developers are still reluctant to adopt these refactorings. This reluctance is due to the tools' poor consideration of context and developer preferences when finding refactorings[23, 144, 71, 136]. In fact, the preferences of developers ranging from quality improvements to code locations, are still not well supported by existing tools and a large number of refactorings are recommended, in general, to fix the

majority of the quality issues in the system.

In our recent survey, supported by an NSF I-Corps project, with 127 experienced developers in software maintenance at 38 medium and large companies (Google, eBay, IBM, Amazon, etc.) [6, 36], 84% of face-to-face interviewees confirmed that most of the existing automated refactoring tools detect and recommend hundreds of code-level issues (e.g., antipatterns and low quality metrics/attributes) and refactorings. However, these tools do not specify where to start or how they relate to a developer's context (e.g., the recently changed files) and preferences in terms of quality targets. This observation is consistent with another recent study [74]. Furthermore, refactoring is a human activity that cannot be fully automated and requires a developer's insight to accept, modify, or reject recommendations because developers understand their problem domain and may have a clear target design in mind. Several studies reveal that automated refactoring does not always lead to the desired architecture even when quality issues are properly detected, due to the subjective nature of software design choices [79, 2, 143, 97, 144, 75, 145]. However, manual refactoring is often error-prone and time-consuming [71, 146].

Several studies have been proposed recently to have developers interactively evaluate refactoring recommendations [75, 74, 135, 6, 36]. The developers provide feedback about the refactored code and may introduce manual changes to some of the recommendations. However, this interactive process can be expensive since developers must evaluate a large number of possible refactorings and eliminate irrelevant ones. Both interactive and automated refactoring approaches have to deal with the challenge of considering many quality attributes for the generation of refactoring solutions. One of the most commonly used quality attributes are the ones of the QMOOD model including reusablitiy, extensibility, effectiveness, etc [46]. QMOOD was empirically validated by many studies, based one hundreds of open source and industry projects, to ensure that they are associated with the qualities they are supposed to measure

and that they are also conflicting [153, 79, 154].

Refactoring studies have either aggregated these quality metrics to evaluate possible code changes or treated them separately to find trade-offs [79, 74, 135, 2, 143, 97, 145, 70]. However, it is challenging to define weights upfront for the quality objectives since developers are often unable to express. Furthermore, the number of possible trade-offs between quality objectives is large, which makes developers reluctant to look at many refactoring solutions—a time-consuming and confusing process. The closest work to this study of Alizadeh et al. [36, 6] shows that even the clustering of non-dominated refactoring solutions based on quality metrics will still generate a considerable number of refactorings to explore. Developers, in practice, combine the use of quality metrics and code locations/files to target when deciding which refactoring to apply. However, existing refactoring tools are not enabling the interactive exploration of both quality metrics and code locations during the refactoring process to identify relevant solutions. The search is beyond just filtering the refactorings but how can the algorithm finds better recommendations after understanding the preferences of the users and giving them a good understanding on how the refactorings are distributed if they are interested to improve specific quality objectives.

In this chapter, we propose an interactive approach that combines multi-objective search, interactive optimization, and unsupervised learning to reduce developer effort in exploring both objective spaces (quality attributes) and decision spaces (files). As a first step, a multi-objective search algorithm, based on NSGA-II [127], is executed to find a compromise between the multiple conflicting quality objectives and generates a set of non-dominated refactoring solutions. Then, an unsupervised clustering algorithm clusters the different trade-off solutions based on their quality metrics. Finally, another clustering algorithm is applied to each cluster of the objective space based on the code locations where the refactorings are recommended. The developer can interact with our tool by exploring both the decision and objective spaces to

identify relevant refactorings based on their preferences quickly. Thus, the developers can focus on their regions of interest in both the objective and decision spaces. The developers are, in general, first concerned about improving specific quality attributes then they will look for the refactorings that best target the files related to their current interests and ownership [146, 144]. Thus, we followed this pattern in our approach by clustering first the objective space then we showed the developers the distribution of the refactorings into different decision space clusters for their prefered objective space cluster.

Our approach takes advantage of multi-objective search, clustering, and interactive computational intelligence. Multi-objective algorithms are powerful in terms of diversifying solutions and finding trade-offs between many objectives but generate many solutions. The clustering and interactive algorithms are useful in terms of extracting developers knowledge and preferences. Existing search-based software engineering techniques are mainly limited to objective space exploration without considering the decision space.

To evaluate our approach, we selected active developers to manually evaluate the effectiveness of our tool on 5 open source projects and one industrial system. Our results show that the participants found their desired refactorings faster and more accurately than the current state of the art of refactoring tools. This confirms our hypothesis that the second level of clustering (decision space) can help developers to quickly find relevant refactorings based on their preferences in terms of both quality objectives to improve and the location of these changes. A video demo of our interactive refactoring tool can be found at [155].

The main contributions of this chapter can be summarized as follows:

1. To the best of our knowledge, the chapter introduces one of the first search-based software engineering techniques that enables the interactive exploration of the objective and decision spaces while existing work focus only on the objec-

tive space. Our approach is not about a simple filtering of the refactorings based on the locations/files or a clustering of the Pareto front based on the locations. We enabled programmers to interactively navigate between both objective and decision spaces to understand how the refactorings are distributed if they are interested to improve specific quality objectives. Then, our approach can generate even more relevant suggestions after extracting that knowledge from the exploration of the Pareto front.

2. Our contribution is beyond the adoption of an existing metaheuristic technique to refactoring. The proposed approach includes a novel algorithm to enable the exploration of both decision and objective spaces by combining two level of clustering algorithms with multi-objective search.

3. We implemented and validated our framework on a variety of open source and industrial projects. The results support the hypothesis that the combination of both the objective and decision spaces significantly improved the refactoring recommendations.

The remainder of this chapter is structured as follows. Section 6.2 presents the challenges in interactive refactoring. Section 6.3 describes our approach, while the results obtained from our experiments are presented and discussed in Section 6.4. Threats to validity are discussed in Section 6.5. Finally, in Section 6.6, we summarize our conclusions and present some ideas for future work.

6.2 Interactive Refactoring Challenges

Refactoring is a human activity that is hard to automate due to its subjective nature and the high dependency on context. While successful tools for refactoring have been created, several challenges are still to be addressed to expand the adoption of refactoring tools in practice. To investigate the challenges associated with current

refactoring tools, we conducted a survey, as part of an NSF I-Corps project, with 127 professional developers at 38 medium and large companies including eBay, Amazon, Google, IBM, and others [36, 6]. All these developers had a minimum of 11 years of experience in software maintenance tasks and especially refactoring. 112 face-to-face meetings were conducted based on semi-structured interviews to understand the challenges that developers are facing with existing refactoring tools.

From these interviews and our extensive industry collaboration, we learned that architects usually have a desired design in mind as a refactoring target, and developers need to conduct a series of low-level refactorings to achieve this target. Without guiding developers, such refactoring tasks can be demanding: it took one software company several weeks to refactor the architecture of a medium-size project (40K LOC) [36]. Several books [95, 96] on refactoring legacy code and workshops on technical debt [148] present the substantial costs and risks of large-scale refactorings. For example, Tokuda and Batory [149] proposed different case studies with over 800 applied refactorings, estimated to take more than 2 weeks.

The majority of the interviewees emphasized that root-canal refactoring to restructure the whole system is rare and they are mainly interested in refactoring files that they own rather than files owned by their peers. However, most existing refactoring tools do not offer a capability of integrating their preferences, in terms of which files they may want to refactor, and purely rely on potential quality improvements. Fully automated refactoring usually do not lead to the desired architecture, and a designer's feedback should be considered. Moreover, prior work [150] shows that even some semi-automated tools are underutilized by developers. Over 77% of our interviewees reported that the refactorings they perform do not match the capabilities of low-level transformations supported by existing tools, and 86% of developers confirmed that they need better design guidance during refactoring: *"We need better solutions of refactoring tasks that can reduce the current time-consuming manual*

work. Automated tools provide refactoring solutions that are hard and costly to repair because they did not consider our design needs."

Based on our previous experience on licensing refactoring research prototypes to industry, developers always have a concern about expressing their preferences up front as an input to guide refactoring suggestions. They prefer to get insights from some generated refactoring solutions then decide which quality attributes they want to improve. Even worse, these preferences are not limited to just the quality metrics and their improvements but also where these refactorings will be applied. However, many existing refactoring tools fail to consider the developer perspective, and the developer has no opportunity to provide feedback on the refactoring solution 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 applied. Determining which quality attribute should be improved, and how, is never a purely technical problem in practice. Instead, high-level refactoring decisions have to take into account the trade-offs between code quality, available resources, project schedule, time-to-market, and management support.

Based on our survey, it is challenging to aggregate quality objectives into one evaluation function to find good refactoring solutions since developers are not able, in general, to express their preferences upfront. While recent advances on refactoring proposed tools support multiple preferences of developers based on multi-objective search, these tools still require the user to navigate through many solutions. Figure 6.1 shows an example of a Pareto front of non-dominated refactoring solutions improving the QMOOD [143] quality attributes of a Gantt Project generated using an existing tool [6]. QMOOD is a widely accepted software quality model, based

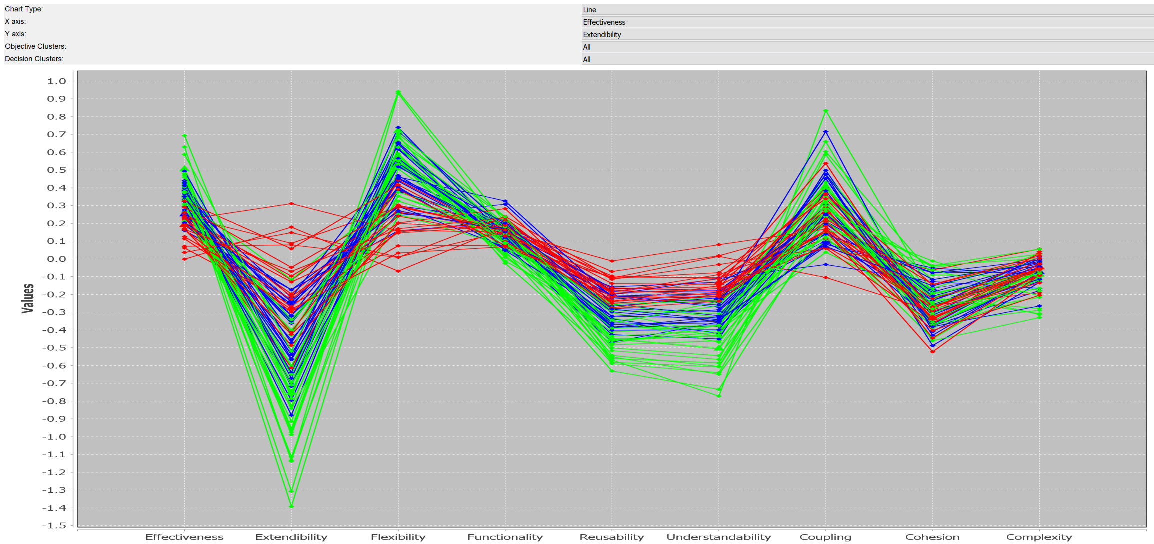


Figure 6.1: The output of a multi-objective refactoring tool [6] finding trade-offs between QMOOD quality attributes on GanttProject v1.10.2 with clustering only in the objective space.

on our collaborations with industry and existing studies [6, 36, 12, 151, 152, 79, 75]. While developers were interested in giving feedback for some refactoring solutions, they still find the interaction process time-consuming. Even when refactoring solutions are clustered based on the quality objectives, as shown in Figure 5.1, the number of solutions to be checked by developers can be substantial. Thus, they want to know how different the solutions are within the same objective space. It may be possible to find more than one refactoring solution that offers the same level of quality improvements but by refactoring different code locations/files. Existing refactoring techniques do not, however, enable developer interaction based on both the decision space and objective space; that is the main challenge of this chapter. For instance, the objective space exploration can help developers focusing on their targeted design quality improvements then the decision space can help them to focus on files they are owning or related to their current tasks or interests.

6.3 Approach Overview

Our proposed approach is composed of four major steps. In the first step, a multi-objective search algorithm is executed to find a set of non-dominated solutions between different conflicting quality objectives of QMOOD [46]. Then, the second step clusters these solutions based on these quality attributes. We call this procedure "objective space clustering". The third step takes, as input, every cluster identified in the objective space and execute another unsupervised learning algorithm to cluster the solutions based on their code locations. Hence, we call this "decision space clustering". Finally, developers can interactively choose among the clustered solutions to find a compromise that suits their preferences in both the decision and objective spaces. For instance, developers may select a cluster that corresponds to their quality improvement preferences. Then, the decision space clustering will identify the most diverse solutions in that cluster that are refactoring different code locations but still provide the same level of quality improvement.

The next sections will explain in further detail the steps of our methodology.

6.3.1 Phase 1: Multi-Objective Refactoring

The search for a refactoring solution requires the exploration of a large search space to find trade-offs between 6 different quality objectives. The multi-objective optimization problem can be formulated mathematically as described in background 2.2.2.

In the following subsections, we briefly summarize the adaptation of multi-objective search to the software refactoring problem.

6.3.1.1 Solution Representation

We encode a refactoring solution as an ordered vector of multiple refactoring operations. Each operation is defined by an action (e.g., move method, extract

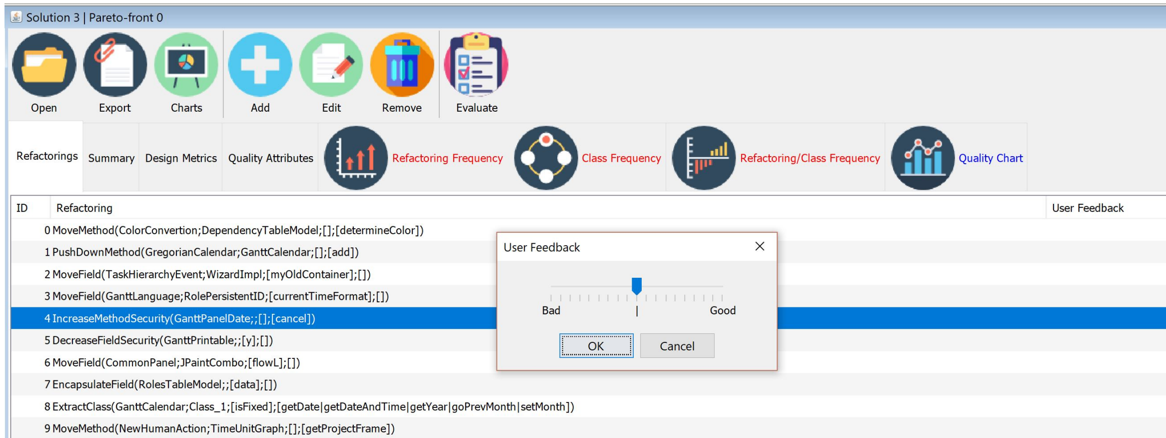


Figure 6.2: Example of a refactoring solution proposed by our tool for GanttProject v1.10.2.

class, etc.) and its specific controlling parameters (e.g., source and target classes, attributes, methods, etc.). We considered a set of the most important and widely used refactorings in our experiments: Extract Class/SubClass/SuperClass/Method, Move Method/Field, PullUp Field/Method, PushDown Field/Method, Encapsulate Field and Increase/Decrease Field/Method Security. These refactoring operations are described in Table 2.1.

We selected these refactoring operations because they have the most impact on QMOOD quality attributes. During the process of population initialization or a mutation operation of the algorithm, the refactoring operation and its parameters are formed randomly. Due to the random nature of this process, it is crucial to evaluate the feasibility of a solution meaning to preserve the software behavior without breaking it. This evaluation is based on a set of specific pre- and post-conditions for each refactoring operation as described in [10]. Figure 6.2 shows an example of a concrete refactoring solution proposed by our approach for GanttProject v1.10.2, including several refactorings applied to different code locations.

6.3.1.2 Fitness Functions

The fitness function is the essential aspect of an optimization problem where we strive to find the best quality value for the given fitness function. It is used to evaluate the goodness of a candidate solution in terms of maximization or minimization. There are two crucial factors for a fitness function: discrimination degree between individuals of a population and calculation speed.

We used the QMOOD [46] as a means of estimating the effect of a refactoring operation on the quality of a software. This model is described in Subsection 2.2.3 and 2.2 and 2.3.

6.3.2 Phase 2: Objective Space Clustering

One of the most challenging and tedious tasks for a user during any multi-objective optimization process is decision making. Since many Pareto-optimal solutions are offered, it is up to the user to select among them, which requires exploration and evaluation of the Pareto-front solutions.

The goal of this step is to cluster and categorize solutions based on their similarity in the objective space. These clusters of solutions help give the user an overview of the options. Therefore, this technique gives the users more explicit initial exploration steps where they can initiate the interaction by evaluating each cluster center or representative member. Based on our previous refactoring collaborations with industry, developers are always highlighting the time-consuming and confusing process to deal with the large population of Pareto-front solutions: ”where should I start to find my preferred solution?”. This observation is valid for many Search-based software engineering (SBSE) applications using multi-objective search [36].

Clustering is an unsupervised learning method to discover meaningful underlying structures and patterns among a set of unlabelled data. It puts the data into groups where the similarity of the data points within each group is maximized while

minimizing the similarity between groups.

Determining the optimal number of clusters is a fundamental issue in clustering techniques. One method to overcome this issue is to optimize a criterion where we try to minimize or maximize a measure for the different number of clusters formed on the data set. For this purpose, we used the Calinski Harabasz (CH) Index, which is an internal clustering validation measure based on two criteria: compactness and separation [137]. We selected the CH index due to the small size of the number of solutions to cluster (our data), and it is known to provide quick clustering solutions with acceptable quality for similar problems. CH assesses the clustering outcomes based on the average sum of squares between individual clusters and within clusters. Therefore, we execute the clustering algorithm on the Pareto-front solutions with various numbers of components as input. The CH score is calculated for each execution, and the result with the highest CH score is recognized as the optimal clustering.

After determining the best number of clusters, we employ a probabilistic model-based clustering algorithm called "Gaussian Mixture Model" (GMM). GMM is a soft-clustering method using a combination of Gaussian distributions with different parameters fitted on the data. The parameters are the number of distributions, Mean, Co-variance, and Mixing coefficient. The optimal values for these parameters are estimated using the Expectation-Maximization (EM) algorithm [138]. EM trains the variables through a two-step iterative process.

After the convergence of EM, the membership degree of each solution to a fitted Gaussian or cluster is kept for the preference extraction step. Furthermore, to find a representative member of each cluster, we measure the corresponding density for each solution and select the solution with the highest density.

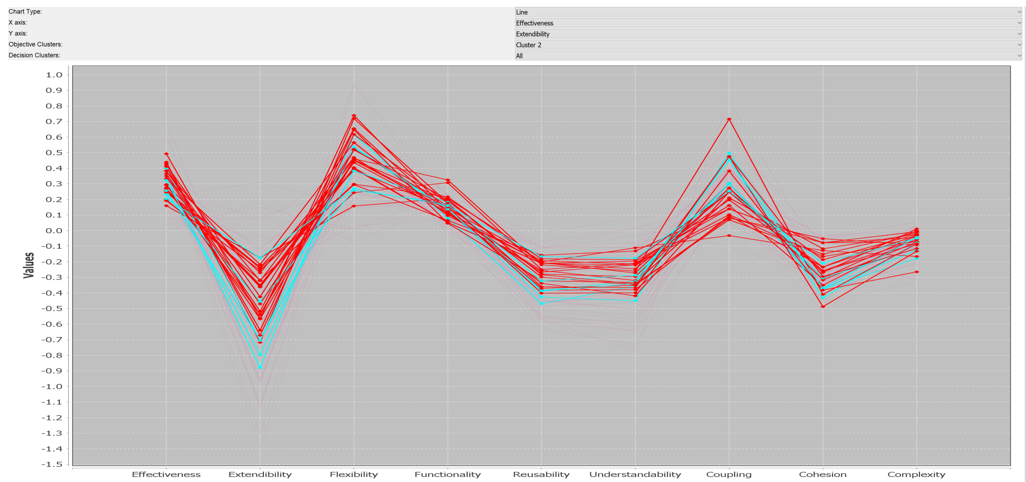


Figure 6.3: Clustering based on code locations (decision space) of the refactoring solutions of one region of interest in the objective space of GanttProject v1.10.2.

6.3.3 Phase 3: Decision Space Clustering

Our approach gives developers the ability to pinpoint their preferences in a different space than the optimization space related to the location of refactorings. After selecting a preferred objective space cluster, the developer may want to see "what are the most diverse solutions within that region of interests". In other words, the clustering in the decision space will show developers the refactoring solutions that improve the quality at the same level (within the same objective space cluster) but targeting different parts of the systems. To do this, we group the solutions by their similarity in the decision space and present them to the developer as depicted in Figure 6.3 where only two clusters were found in the decision space. In each of these two clusters, the solutions composing it are introducing refactorings into similar locations with comparable impact on the different quality attributes. These solutions in the decision space are clustered based on the refactoring locations and their frequency.

As Algorithm 6 represents, to get an optimal grouping of solutions in the decision space of where refactorings are applied, we use a procedure similar to the one used in the objective space with additional pre-processing steps to project the solutions

Algorithm 6: Decision-Space Clustering

```
Input   : Pareto-Front Solutions, Clusters
Output: Labeled Pareto-Front Solutions (LS)
begin Projection Operator
2   RefactoredClasses ← GetRefactoredClasses (ParetoFront) ;
3   ProjectedParetoFront(PS) ←ExtractFrequency
     (RefactoredClasses,ParetoFront)
begin Calculate best number of clusters-K
    for i ← 2 to 10 do
6       LS = GMMClustering (i,PS);
7       Score_i = CalinskiHarabaszIndex (LS)
    K ← MaxScoreIdx ();
begin GMMClustering (K,PS)
    μ_k, Σ_k, π_k ← Initialize-K-Gaussian (); /*Expectation-Maximization
    while ¬converge do
        γ(s_nk) ← Expectation ();
        μ_k, Σ_k, π_k ← Maximization ();
        EvaluateLikelihood ();
    foreach s_n ∈ S do
16      //assigning cluster labels
        L_n ← MaxResponsibilityIdx ();
18 Return LS;
```

on the decision space. We define a projection operator based on the frequency of changes to the classes by the refactorings and their locations (refactored files). Since refactoring operations affect classes differently, where some make changes only at the same class level while others have a source class and a target class, we only count source classes in our work to have a consistent representation for all vectors and to create a new representation for the refactoring vector in the decision space. In this new domain space, the solutions are represented as vectors of integers where the refactored classes are the dimensions of the space, and the values are the number of refactoring operations for that class. The projection operator is used for the entire Pareto-front and enables having two different representations of the same solution set.

The main contribution of our work is enabling the exploration of a diverse set

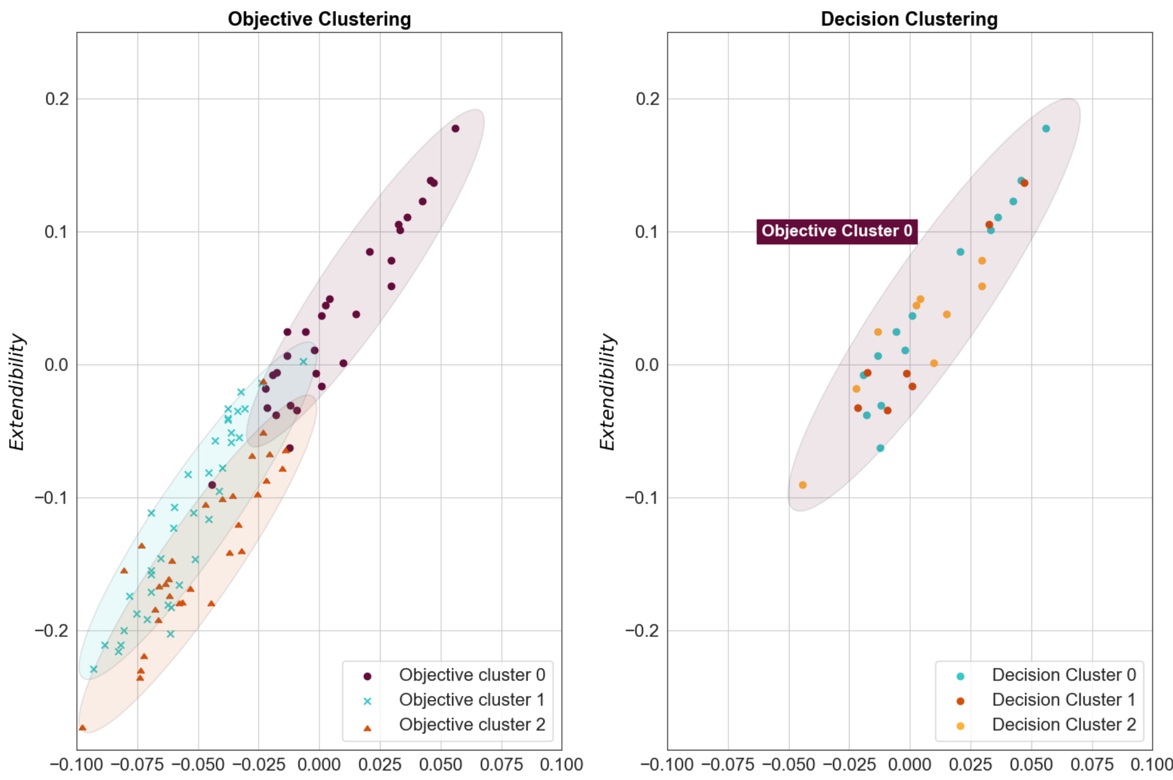


Figure 6.4: Illustration of the clustered solutions in the objective space and the decision space.

of refactoring solutions within the same objectives space. This amounts to having multiple solutions that are neighbors in the objective space but completely different in the decision space. To do this, we go through all the clusters determined in the previous step and then use the GMM clustering algorithm with the same steps described above to group similar solutions in the decision space. Thus, developers can improve the code toward their preferred objectives while only refactoring the parts of the code that interest them.

Figure 6.4 shows an example of our approach using the Bi-Clustering NSGA-II algorithm where after generating the Pareto-front for the effectiveness and extendibility objectives, the developer can select a cluster in the objective space for further exploration. Then, a developer can explore the clusters and observe that within this cluster, there are three different clusters in the decision space. The region of interest can be

highlighted, and the developer can select solutions that correspond to their interest to create further constraints for the optimization process to converge to the desired optimum.

6.3.4 Phase 4: Developer Feedback and Preference Extraction

The results of the Bi-Space clustering algorithm are presented to developers in the form of an interactive chart where they can visualize the cluster of their choice in the objective and decision spaces. This presentation helps them get a complete picture of the diversity of the refactoring solutions and the various compromises they may offer. Our goal is to minimize the effort spent by developers to interact with the system and select a final set of refactorings.

Looking at the solutions, developers can evaluate every solution based on their preferences. The granularity offered by our representation enables developers to make evaluations at the cluster level (selecting one or more clusters in the objective space), solution level (selecting solutions within a chosen cluster) and refactoring level (choosing to accept, reject, or modifying some refactorings within the chosen solution as shown in Figure 6.2.). The score obtained reflects developer preferences and serves to determine their region of interest.

At the solution level, the developer is capable of inspecting every refactoring and modifying it. Refactoring operations can be added, deleted, modified, or re-ordered. The information collected afterward is used to calculate a score at the solution level by averaging the scores for every refactoring, and at the cluster levels by averaging the scores of the solutions.

In this way, we can characterize the developer's region of interest as the cluster with the highest score. Information about the preferred classes, refactorings, and quality metrics is extracted and used to create constraints on the optimization process. Therefore, the search becomes guided in both the decision and the objective spaces,

and we can converge on a developer's preferred solution faster.

For this purpose, we compute the weighted probability of refactoring operations (RWP) and target classes of the source code (CWP) as follow:

$$RWP_p = \frac{\sum_{s_i \in c_j} \gamma_{ij} \times (|r_p \in s_i|)}{\sum_{r_m \in Ref} \sum_{s_i \in c_j} \gamma_{ij} \times (|r_m \in s_i|)} \tag{6.1}$$

$$CWP_q = \frac{\sum_{s_i \in c_j} \gamma_{ij} \times (|cl_q \in s_i|)}{\sum_{cl_m \in Cls} \sum_{s_i \in c_j} \gamma_{ij} \times (|cl_m \in s_i|)} \tag{6.2}$$

where j is the index of selected cluster, s_i is the solution vector, γ_{ij} is the membership weight of solution i to the cluster j, r is refactoring action, Ref is the set of all refactoring operations, and Cls is the set of all classes in the source code.

6.4 Evaluation

6.4.1 Research Questions

We defined two main research questions to measure the correctness, relevance, and benefits of our decision and objective space interactive clustering-based refactoring (DOIMR) tool comparing to existing approaches that are based on interactive clustering-based refactoring only in the objectives space (Alizadeh et al.) [36], interactive multi-objective search (Mkaouer et al.) [75, 6], fully automated multi-objective search (Ouni et al.) [2] and fully automated deterministic tool not based on heuristic search (JDeodorant) [5]. A tool demo of our tool and supplementary appendix materials (questionnaire, setup of the experiments, statistical analyses, and detailed results) can be found in our study's website [1].

The research questions are as follows:

- **RQ1:** Does our approach make more relevant recommendations for developers,

[1] A demo and supplementary appendix materials can be found at the following link: https://sites.google.com/view/tse2020decision

as compared to existing refactoring techniques?

- **RQ2:** Does our approach significantly reduce the number of relevant refactoring recommendations and the user interaction effort, as compared to existing interactive refactoring approaches?

6.4.2 Experimental Setup

We considered a total of seven systems, summarized in Table 6.1, to address the above research questions. We selected these seven systems because they are of reasonable size, have been actively developed over the past 10 years, and have been extensively analyzed by the other tools considered in this work. UTest[2] is a project of our industrial partner used for identifying, reporting, and fixing bugs. We selected that system for our experiments since five developers of that system agreed to participate in the experiments, and they are very knowledgeable about refactoring—they are part of the maintenance team. Table 6.1 provides information about the size of the subject systems (in terms of number of classes and KLOC).

To answer RQ1, we asked a group of 35 participants to manually evaluate the relevance of the refactoring solutions that they selected using four other tools. The first tool of Alizadeh et al. is an approach based on only objective clustering of the Pareto front [36], using the interactive multi-objective search. The second tool is an interactive multi-objective refactoring approach proposed by Mkaouer et al. *et al.* [75, 6], but the interactions were limited to the refactorings (accept/reject) and there is no clustering of the Pareto front or learning mechanisms from the interaction data. Thus, the comparison with these tools will help us to evaluate our main contribution that is built on the top of existing multi-objective refactoring algorithms: the combined use of decision and objective space exploration for interactive refactoring. We have also compared our DOIMR approach to two fully-automated refactoring tools: Ouni

[2]Company anonymized for double-blind.

Table 6.1: Statistics of the studied systems.

System	Release	#Classes	KLOC
ArgoUML	v0.3	1358	114
JHotDraw	v7.5.1	585	25
GanttProject	v1.10.2	241	48
UTest	v7.9	357	74
Apache Ant	v1.8.2	1191	112
Azureus	v2.3.0.6	1449	117
JFreeChart	v1.0.9	521	170

et al. [2] and JDeodorant [5]. Ouni *et al.* [2] proposed a multi-objective refactoring formulation based on NSGA-II that generates a solution to maximize the design coherence and refactoring reuse from previous releases. JDeodorant [5] is an Eclipse plugin to detect bad smells and apply refactorings. 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 those refactorings. We used these two tools to evaluate the relative benefits of our interactive features in helping developers identifying relevant refactorings.

We preferred not to use measures such as anti-patterns or internal quality indicators as proxies for estimating the relevance of refactorings since the developers' manual evaluation already includes a review of the impact of suggested changes on the quality. Furthermore, not all the refactorings that improve quality attributes are relevant to the developers, which is one of the main motivations of this work. The only rigorous way to evaluate the relevance of our tool is the manual evaluation of the results by active developers. This manual evaluation score, MC, consists of the number of relevant refactorings identified by the developers over the total number of refactorings in the selected solution. Due to the subjective nature of refactoring and the large size of considered systems, it is almost impossible to estimate the recall. There is no unique solution to refactor a code/design; thus, it is challenging to construct a gold-standard for large-systems, which makes calculating the recall very

challenging.

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. The list of questions of all the questionnaires and the obtained results can be found in the online appendix. Although the vast majority of participants were already familiar with refactoring as part of their jobs and graduate studies, all the participants attended a two-hour lecture on refactoring by the organizers of the experiments. The details of the selected participants can be found in Table 6.2, including their programming experience in years, familiarity with refactoring, etc. These participants were recruited based on our networks and previous collaborations with 4 industrial partners. They all had a minimum of 6 years experience post-graduation and were working as active programmers with strong backgrounds in refactoring, Java, and software quality metrics.

Each participant was asked to assess the meaningfulness of the refactorings recommended after using up to two of the five tools on up to two different systems, to avoid a training threat to validity. The participants not only evaluated 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 five participants from the industrial partner, where they agreed to evaluate only their industrial software. We assigned tasks to the participants according to the studied systems, the techniques to be tested and developers' experience. Each of the five tools has been evaluated at least one time on each of the seven systems.

To answer RQ2, we measured the time (T) that developers spent to identify the best refactoring strategies based on their preferences and the number of refactorings (NR). Furthermore, we evaluated the number of interactions (NI) required on the Pareto front for all interactive refactoring approaches. This evaluation will help to

Table 6.2: Selected participants.

System	#Subjects	Prog. Exp. (Years)[Avg-Min-Max]	Avg. Refactoring Exp.
ArgoUML	5	[7.5 - 6 - 8.5]	Very High
JHotDraw	5	[8 - 6.5 - 9]	Very High
Azureus	5	[9.5 - 7.5 - 11.5]	High
GanttProject	5	[7 - 6 - 8.5]	High
UTest	5	[15.5 - 13 - 19.5]	Very High
Apache Ant	5	[9 - 6 - 12.5]	Very High
JFreeChart	5	[7 - 6 - 9.5]	Very High

understand if we efficiently reduced the interaction effort. For this research question, we decided to limit the comparison to only the interactive multi-objective work of Mkaouer et al. [75, 6] and Alizadeh et al. [36] since they are the only ones offering interaction with the users, and it will help us understand the real impact of the decision space exploration (not supported by existing studies) on the refactoring recommendations and interaction effort.

6.4.3 Parameter Setting

It is well known that many parameters compose computational search and machine learning algorithms. Parameter setting is one of the longest standing grand challenges of the field. We have used one of the most efficient and popular approaches for parameter setting of evolutionary algorithms, which is Design of Experiments (DoE). 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's values have been experimented.

The stopping criterion was set to 100,000 evaluations for all optimization and search algorithms to ensure fairness of comparison (without counting the number of interactions since it is part of the users' decision to reach the best solution based on their preferences).

The parameters of the multi-objective algorithm are as follows: crossover proba-

bility = 0.7; mutation probability = 0.4, where the probability of gene modification is 0.5. Furthermore, we used the maximum number of iterations = 1000 and convergence threshold = 0.0001 for the GMM clustering phase.

6.4.4 Results

Results for RQ1. Figure 6.5 summarizes the manual validation results of our DOIMR approach compared to the state of the art, as evaluated by the participants. It is clear from the results that interactive approaches generated much more relevant refactorings, as compared with the automated tools of Ouni et al. and JDeodorant. Among the interactive approaches, DOIMR outperformed the other interactive approaches of Mkaouer et al. and Alizadeh et al. which supports the idea that information that the developer used from the decision space, such the code locations where refactorings were applied and the refactorings frequency, was helpful. On average, for all of our seven studied projects, 91% of the proposed refactoring operations were considered to be useful by the subjects. The remaining approaches have an average of 83%, 71%, 67%, and 56% respectively for Alizadeh et al. (interactive with objective space clustering), Mkaouer et al. (interactive multi-objective approach), Ouni et al. (fully automated multi-objective approach) and JDeodorant (deterministic non-search-based approach). The highest MC score is 100% for the Azureus and Gantt projects, and the lowest score is 91% for the industrial system UTest. This lowest score can be explained by the fact that the participants are very knowledgeable about the evaluated system. The participants were not guided on how to interact with the systems, and they mainly looked at the source code to understand the impact of recommended refactorings.

We found that automated refactorings generate a lot of false positives. Both the Ouni et al. and JDeodorant tools recommended a large number of refactorings compared to the interactive tools, and many of them are not interesting for the context

of the developers, and so the developers reject these refactorings, even though they may be correct. For instance, the developers of the industrial partner rejected several recommendations from these automated tools simply because they were related to stable code or code fragments outside of their interests. The majority of them will not change code out of their ownership as well. Furthermore, they were not interested to blindly change anything in the code just to improve quality attributes. Compared to the remaining interactive approaches, we found that some of the refactoring solutions of DOIMR will never be proposed by Mkaouer et al. or Alizadeh et al. since they are selected because of their extensive refactoring on specific code fragments that developers may found essential to improve their quality based on the features included in these classes. In fact, one of the main challenges of multi-objective search is the noise introduced by sacrificing some objectives and trying to diversify the solutions. Thus, the decision space exploration can help the developers know the most diverse refactoring solutions among one preferred cluster in the objective space. Thus, developers did not waste time on evaluating refactoring solutions that are similar but related to entirely different code files.

To conclude, our DOIMR approach outperformed the four other refactoring approaches in terms of recommending relevant refactoring solutions for developers (RQ1).

Results for RQ2. Figures 6.6, 6.7, and 6.8 give an overview of the number of refactorings for the selected solution, number of required interactions, and the time, in minutes, using our tool, the interactive clustering approach of Alizadeh et al., and the interactive multi-objective approach of Mkaouer et al. Based on the results of Figure 6.6, it is transparent that our approach significantly reduced the number of recommended refactorings compared to the other interactive approaches while increasing the manual correctness as described in RQ1. The highest number of refactorings was observed on the industrial system with 32 refactorings using DOIMR, 48 using Alizadeh et al. and 72 refactorings using Mkaouer et al. This result may be explained

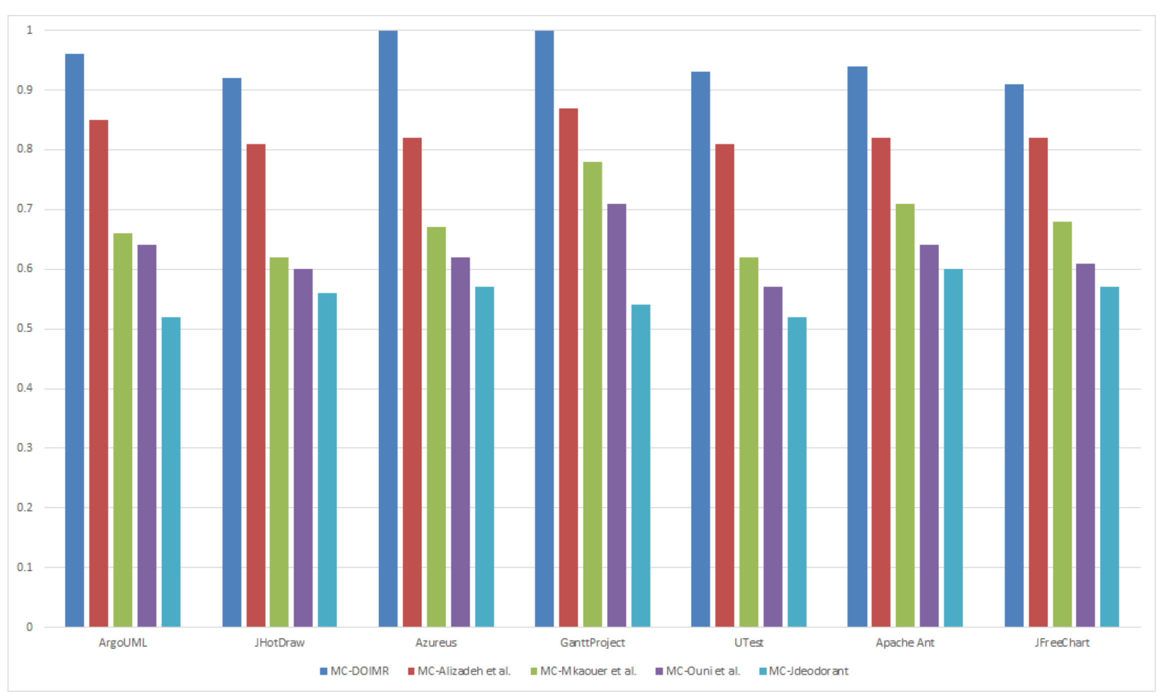


Figure 6.5: Median manual evaluations, MC, on the 7 systems.

by the size and the quality of this system along with the fact that it was evaluated by some of the original developers of UTest. The lower number of recommended refactorings using DOIMR, compared to the other interactive approaches, is related to the elimination of the noise in multi-objective search not only in terms of objectives but the relevant code locations to be refactored (decision space). It is normal to see fewer refactorings when the search space is reduced to a smaller number of files, which was the case of DOIMR.

Figure 6.7 shows that DOIMR required far fewer developer interactions than the other interactive approaches. For instance, only 13 interactions were required to modify, reject and select refactorings on Azureus using our approach, while 23 and 38 interactions respectively were needed for Alizadeh et al. and Mkaouer et al. The reduction of the number of interactions is mainly due to the smaller number of solutions to explore, after the selection of a preferred cluster in both the objective and decision spaces.

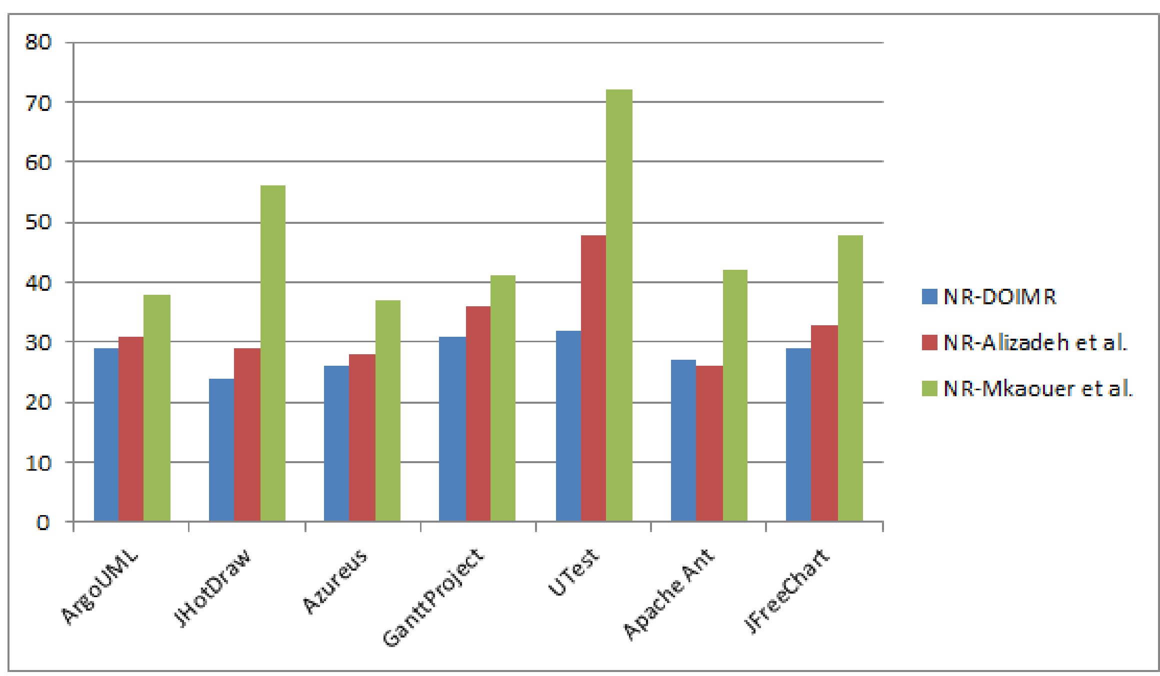


Figure 6.6: The median number of recommended refactorings, NR, of the selected solution on the 7 systems.

The participants also spent less time to find the most relevant refactorings on the various systems compared to the other interactive approaches, as described in Figure 6.8. The execution time of our approach includes the execution of the multi-objective search, both clusterings, and the different phases of interaction until the developer is satisfied with a specific solution. The execution time of Alizadeh et al. included all the steps of multi-objective search, the objective space clustering, and the interactions while Mkaouer et al. included the multi-objective search and the user interactions. Thus, it is natural that the main differences in the execution time can be observed in the interaction effort. The average time of our approach is reduced by over 40 minutes (70%) compared to Mkaouer et al. for the case of JHotDraw. The reduction of the execution time is mainly explained by the rapid exploration of fewer solutions after looking mainly to the most diverse (different) solutions in the decision space of the preferred cluster in the objective space. In fact, our DOIMR tool has more components (clustering at both objective and decision spaces) than

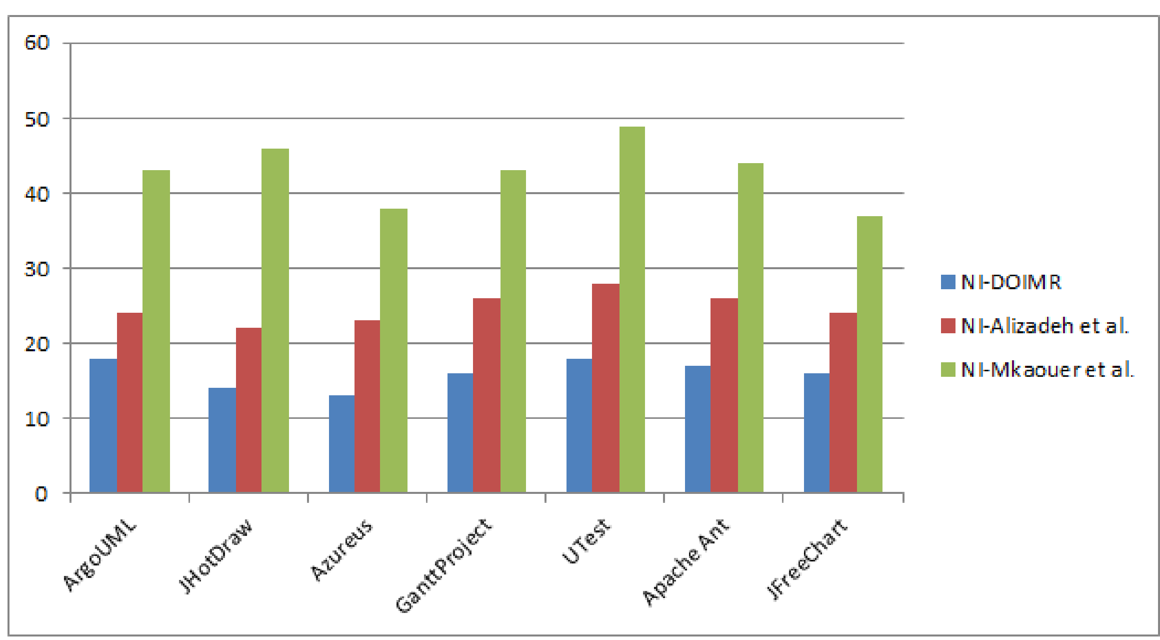


Figure 6.7: The median number of required interactions (accept / reject / modify / selection), NI, on the 7 systems.

Alizadeh et al. and Mkaouer et al. but the clustering at both spaces significantly reduced the most time-consuming step (user interactions) since the clusterings, and multi-objective search algorithms are quick and executed in few minutes (between 2 and 4 minutes).

6.5 Threats to Validity

Conclusion validity. The parameter tuning used in our experiments creates an internal threat that we need to evaluate in our future work. We have used one of the most efficient and popular approaches for parameter setting of evolutionary algorithms, which is Design of Experiments (DoE). Each parameter has been uniformly discretized in some intervals. Values from each interval have been tested for our application. Finally, we chose the best values for all parameters. Hence, a reasonable set of parameter values have been studied. Another conclusion threat is the number of interactions with the developers since we did not force them to use the same max-

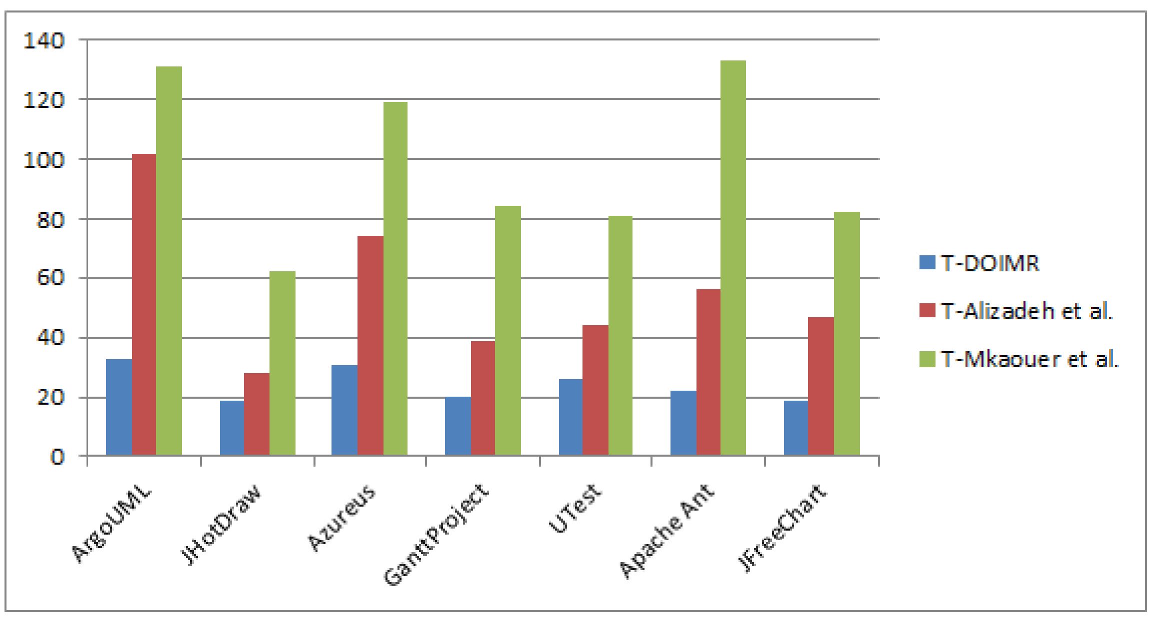


Figure 6.8: The median execution time, T, in minutes on the 7 systems.

imum number of interactions which may sometimes explain the out-performance of our approach. However, the participants were given the same amount of time to use the tool (limited to three hours).

Internal validity. The variation of correctness and speed between the different groups when using our approach and other tools can be an internal threat since the participants have different levels of experience. To counteract this, we assigned the developers to different groups according to their programming experience to reduce the gap between the groups, and we also adopted a counter-balanced design. Regarding the selected participants, we took precautions to ensure that our participants represented a diverse set of software developers with experience in refactoring, and also that the groups formed had similar average skill sets in terms of refactoring area.

Construct validity. The developers involved in our experiments may have had divergent opinions about the relevance of the recommended refactorings, which may impact our results. However, some of the participants are the original programmers of the industrial system, which may reduce the impact of this threat. Unlike fixing bugs, refactoring is a subjective process, and there is no unique refactor solution; thus, it is

difficult to construct a gold-standard for large systems which makes calculating recall challenging. Does the deviation from an expected refactoring solution mean that the recommendation is wrong or simply another way to refactor the code?

External validity. The first threat is the limited number of participants and evaluated systems, which threatens the generalizability of our results. Besides, our study was limited to the use of specific refactoring types and quality attributes. Furthermore, we mainly evaluated our approach using classical algorithms such as NSGA-II, but other existing metaheuristics can be used. Future replications of this study are necessary to confirm our findings.

6.6 Conclusion

In this chapter, we presented a novel way to enable interactive refactoring by combining the exploration of quality improvements (objective space) and refactoring locations (decision space). Our approach helped developers to quickly explore the Pareto front of refactoring solutions that can be generated using multi-objective search. The clustering of the decision space helped the developers identify the most diverse refactoring solutions among ones located within the same cluster in the objective space, improving some desired quality attributes. To evaluate the effectiveness of our tool, we conducted an evaluation with human subjects who evaluated the tool and compared it with the state-of-the-art refactoring techniques. Our evaluation results provide evidence that the insights from both the decision and objective spaces helped developers to quickly express their preferences and converge towards relevant refactorings that met the developers' expectations.

Future work idea can be automatically learning from user interactions for fast convergence to good refactoring solutions. Besides, the experiments can be expanded with more systems and participants.

CHAPTER VII

Intelligent Refactoring Bot for Continuous Integration

7.1 Introduction and Problem Statement

Refactoring, defined as a set of program transformations intended to improve the system design while preserving the desired behaviour, is becoming a critical software maintenance activity, especially with the growing complexity of software systems [156]. A recent study by the US Air Force Software Technology Support Center (STSC) shows that restructuring the code of a large project reduced developers' time by over 60% when introducing new features. However, refactoring is expensive. Developers take an average of 6 weeks to refactor the design of medium-size projects (around 30K LOC) [74]. There has been much work done on various techniques and tools for software refactoring [157, 158, 159, 4, 160] and these approaches can be classified into three main categories: *manual*, *semi-automated* and *fully-automated* approaches.

In manual refactoring, the developers refactor with no tool support except the execution part, identifying the parts of the program that require attention and performing all aspects of the code transformation by hand. It may seem surprising that a developer would eschew the use of tools in this way, but Murphy-Hill et al. [71]

found in their empirical study of the developers' usage of the Eclipse refactoring tooling that in almost 90% of cases the developers performed refactorings manually and did not use automated refactoring tools. Kim et al. [88] confirmed this observation, finding that the interviewed developers from Microsoft preferred to perform refactoring manually in 86% of cases. Despite its apparent popularity, manual refactoring is very limited. However, several studies have shown that manual refactoring is error-prone, time-consuming, not scalable and not practical for extensive application of refactorings to fix major quality issues [122, 161, 158]. Although developers are doing refactorings manually, the surveys confirmed that they are not frequently refactoring their code because of the above limitations.

In fully-automated refactoring, developers provide their code as input, and the tool will provide refactoring recommendations automatically [157]. The majority of existing automated refactoring tools assume that developers want to fix code smells [162, 163, 164]. 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. First, 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, and they lose any control of the introduced code changes. Second, it lacks flexibility since the developer has to either accept or reject the entire refactoring solution. In fact, developers intentions may not be, most of the time, fixing code smells or the majority of them. Third, it fails to consider the developer 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 applied [25]. Finally, one of the significant limitations of existing automated refactoring tools is the high configuration effort required to integrate them into the current development pipeline of the team/company. In fact, several companies are now using continuous integration and DevOps, which make the adoption of current automated refactoring tools very challenging.

Recently, few interactive refactoring techniques were proposed [75, 79, 36, 6]. They provide to the developers the flexibility to approve or reject the recommended refactoring that can improve the quality. However, this interaction process is time-consuming, and developers get frustrated from providing feedback on files that are out of their interests/ownership or navigating through many refactoring recommendations/strategies to improve several quality metrics.

To address all the above challenges, we propose the first attempt to design and build an intelligent refactoring bot as a GitHub app that can be easily integrated into any project repository on GitHub. The bot can be customized to monitor the quality in the repository after some pull-requests repeatedly or automatically executed when the quality analysis shows a significant decrease. The bot analyzes the files changed during that pull request(s) to identify refactoring opportunities using a set of quality attributes then it will find the best sequence of refactorings to fix the quality issues if any. The bot recommends all these refactorings through an automatically generated pull-request. The developer, whenever available without interrupting the development pipeline, can review the recommendations, and their impacts in a detailed report and select the code changes that he wants to keep or ignore. After this review, the developer can close and approve the merge of the bot's pull request. We quantitatively and qualitatively evaluated the performance and effectiveness of RefBot by a survey conducted with experienced developers who used the bot on both open source and industry projects.

The primary contributions of this chapter can be summarized as follows:

1. This chapter introduces a novel way to refactor software systems using autonomous intelligent software bots but still considering developers interaction to review the generated pull-request.

2. We propose an implementation of the refactoring bot as a Git app that can be quickly adopted in a continuous integration environment or DevOps process.

3. The chapter reports the results of an empirical study on an implementation of our approach. The obtained results provide evidence to support the claim that, on average, our bot is more efficient than existing automated refactoring techniques based on a benchmark of six open source systems and one industrial project. This chapter also evaluates the relevance and usefulness of the suggested refactorings for software developers in improving the quality of the modified files in several pull-request.

The remainder of this chapter is structured as follows. Section 7.2 describes our intelligent refactoring bot, while the results obtained from our experiments are presented and discussed in Section 7.3. Threats to validity are discussed in Section 7.4. Finally, in Section 7.5, we summarize our conclusions and present some ideas for future work.

7.2 Approach

We developed the "Refactoring Bot" (RefBot) as a GitHub App using which the workflow can be automated, and the developers can integrate the bot easily to any repository of their interest. The overview of the Refactoring bot is shown in Figure 7.1.

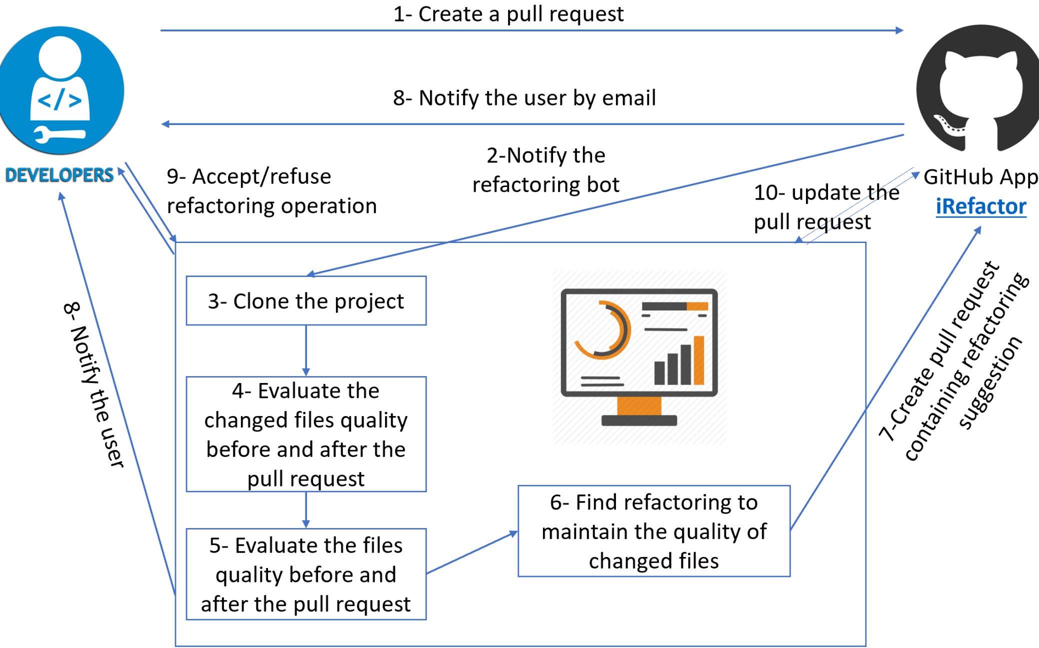


Figure 7.1: The overview of RefBot Pipeline.

7.2.1 RefBot Parameters Setting

The first step of utilizing the Refactoring bot is to install its GitHub application on organizations or user accounts and to set up the appropriate permissions. As the installation page in Figure 7.2 shows, the user can select the repositories. Therefore, RefBot is granted access to the specific repositories via the GitHub API. RefBot has read and write permissions to "Pull Requests" and "WebHook", and also is subscribed to "Pull Requests" and its related "reviews and comments" events.

After this step, RefBot automatically sets up a web-hook for the developer's profile which means the permitted activities on the selected repositories will be posted as JSON-formatted payloads to the designated external server.

7.2.2 Processing a Pull Request

RefBot continuously monitors the actions performed on the repository by checking the subscribed payloads delivered to its server. In our current configuration, opening a new pull request action triggers the RefBot's workflow.

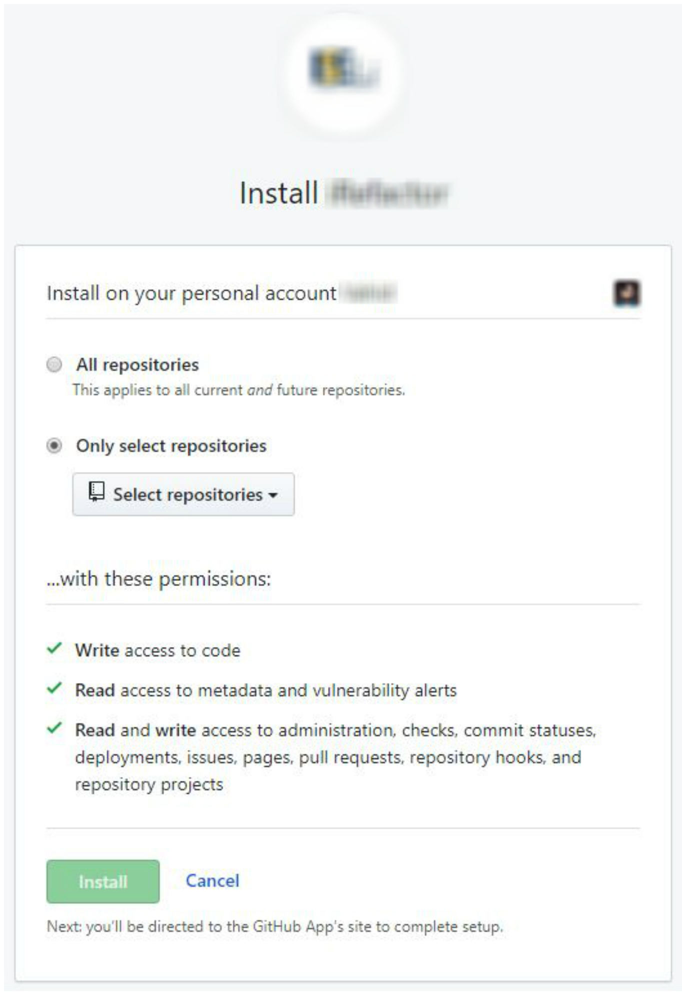


Figure 7.2: Installing RefBot on a repository.

First, the commits in the pull request are compared to the commit at the point where the branch is created to extract the list of all files changed by the pull request. Then, two versions of the files, before and after the pull request, are downloaded to the external server for further processing and modifications.

By processing only the changed files by the pull request, we ensure that the developers are provided with the reports and refactorings limited to the codes they recently modified. This feature facilitates the evaluation of recommended refactorings and is aligned with the idea of maintaining/improving the code quality in the continuous development process.

7.2.2.1 Calculating Quality Changes

The RefBot analyses the code quality of the extracted files. For this purpose, we adopted QMOOD quality assessment methodology, which is a hierarchical model for object-oriented designs [46]. This model is described in Subsection 2.2.3 and 2.2 and 2.3.

QMOOD model comprises of four levels from which we utilized the first level, Design Quality Attributes, to measure code quality changes of the pull request.

It is shown that QMOOD metrics model is highly effective in predicting software defects in both traditional and iterative (like agile) software development processes [165].

Since the QMOOD metrics are not limited to a specific range, it is difficult for the user to interpret their values. Therefore, we built a software quality benchmark dataset consisting of the quality metrics calculated for over 100 open-source and industrial software projects. Then, to summarize all six quality attributes, we defined a super metric called Total Quality Index (TQI) as the linear summation of the metrics.

Finally, we compared the quality metrics and TQI of a new project/file with the range of the benchmark and assigned a quality label (A, B, C, and D) based on the quartile of a value.

This method facilitates the analysis of quality reports and gives meaning to the metrics in terms of the quality level (low/high) of software compared to other standard projects.

7.2.2.2 Optimization Using Refactoring

Finding a refactoring solution can be a challenging task since a huge search space requires to be explored. This search space is the outcome of the number of refactoring operations and the importance of their order and combination. To search this space,

we employed an adaptation of the NSGAII [127] to discover a trade-off between multiple quality attributes.

NSGA-II is a multi-objective evolutionary algorithm operating on a population of candidate solutions that are evolved toward the Pareto-optimal solution set. NSGA-II uses an explicit diversity-preserving strategy together with an elite-preservation strategy. [127].

A refactoring solution is designed as a vector that consists of an ordered sequence of multiple refactoring operations. Each refactoring operation includes a refactoring action and its specific controlling parameters. The refactoring operations considered in RefBot cover the most used operations selected from different categories: "Moving features", "Data organizers", "Method calls simplifiers", and "Generalization modifiers". These refactorings are listed in Table 2.1.

Refactoring operations are created or modified randomly during the population initialization or mutation. Also, the size of a solution vector which is the number of included refactoring operation is randomly selected between lower and upper bound values. Therefore, it is crucial to examine the feasibility of a solution using related pre-conditions and post-conditions [10]. These conditions ensure that the program will not break while the behaviour is preserved by the refactoring.

To evaluate a candidate refactoring solution, a fitness function is defined to estimate its goodness. In order to measure the impact of a refactoring solution on the software project, we utilized six QMOOD quality attributes. The relative change of each quality attribute after applying the refactoring solution to the software system is considered as the fitness function and is expressed as:

$$FitnessFunction_i = \frac{AQM_i^{after}(CC) - AQM_i^{before}(CC)}{AQM_i^{before}(CC)} \tag{7.1}$$

where AQM_i^{before} and AQM_i^{after} are the averages of the quality metric i before and after applying a refactoring solution over all changed classes CC, respectively.

By defining the fitness function in this way, we aim to find the solutions capable of improving the quality attributes of the pull request.

Additionally, we constraint the search process to the solutions in which at least a "class" controlling parameter is in the set of changed files in the pull request. For this purpose, we modified a variation operator of the search algorithm called "Selection Operator". Variation operators help to navigate through the search space and to maintain a good diversity in the population. Parent selection is a crucial step that directly affects the convergence rate. We added the controlling parameter constraint to the selection process.

After the execution of the refactoring search algorithm is finished, the instruction of applying each refactoring operation is added to the related files as a distinctive marker format similar to the Git conflict marker. Finally, RefBot creates a new pull request to introduce the changes to the repository.

7.2.3 Developer's Interaction

One of the main advantages of RefBot is to include the developer in the refactoring process loop. When the internal workflow of RefBot on a pull request is completed, the developer is notified by email and also via GitHub checks API in the same page of the pull request. These notifications contain a link to the report page of the pull request where the users can analyze the results and give feedback to the recommended refactorings.

There are three levels of reports generated for each pull request and provided for the user:

- *Solution Report:* contains the quality history of the pull request and the impact of the recommended solution on the changed files.
- *File Report:* includes the list of refactorings applied to the selected file and the

detailed quality history and impact of refactoring.

- *Refactoring Report:* represents the instruction of a single refactoring and the high-level code abstraction of source and target classes which are transformed by the operation.

Analyzing these simple yet effective reports give the ability of swift detection of required improvements based on individual preferences.

The developer can interact with the refactoring results of RefBot with three actions. Each refactoring can be "rejected", "applied with a code marker", or "applied automatically".

By rejecting a refactoring, it is not considered in the pull request. Applying with a code marker adds the refactoring instruction as a marker inside the related files. Therefore, the developer can manually implement the required changes. Last, applying automatically, gives permission to RefBot to change and apply the refactorings to the source code itself.

The reason we have both manual and automated refactoring is that sometimes the developers prefer to take control of the refactoring process and the changes in the structure of their code either for the whole software or a specific set of important classes/files.

When the developer is satisfied with the feedback, he/she can update the previously created RefBot's pull request.

RefBot can be combined with continuous integration tools like TravisCI, Jenkins, or CircleCI to identify the problems that may occur during the automated refactoring by running integration tests.

7.2.4 Configuration and Customization

RefBot is highly customizable in terms of setting its internal workflow parameters and execution management.

#	Actions	File Relative Path	TQI Before	TQI Grade Before	TQI After	TQI Grade After	TQI Refactoring	TQI Grade Refactoring	Number Of refactoring ↓	Not Reviewed refactoring
38	View	NettyMessagi...	13.618	A	5.508	B	5.107	B	12	12
21	View	AtomixAgent.j...	4.051	B	1.182	C	5.728	B	7	7
1	View	NettyMessagi...	1.912	C	0.544	C	0.501	C	1	1
29	View	AtomixConfigT...	2.717	B	2.047	C	4.608	B	1	1
0	View	DefaultPrimar...	2.767	B	2.757	B	2.758	B	0	0
2	View	PartitionedDist...	7.619	A	7.609	A	7.610	A	0	0
3	View	RaftServiceCo...	4.746	B	4.765	B	4.766	B	0	0
4	View	AtomixCluster...	-2.416	D	-0.333	D	-0.332	D	0	0
5	View	RaftSession.ja...	5.366	B	5.361	B	5.362	B	0	0
6	View	RaftSessionInv...	3.837	B	4.104	B	4.104	B	0	0

Figure 7.3: The quality table in solution report page.

Sometimes a developer is not willing to be disturbed for every new pull request. Therefore, RefBot can be configured to monitor the repository at a specific time interval or even can be triggered manually for a specific pull request.

Furthermore, users can enable/disable different refactoring types and quality attributes. In this way, they can control the optimization process and limit the search to the refactoring operations they are willing to apply and to the quality attributes they prefer to improve.

Additional materials such as the default parameter settings for NSGA-II and video demo of RefBot can be found at this publication's web page [1].

7.2.5 Running Example

In this section, to illustrate the process of RefBot and its performance in refactoring a pull request, we provide a running example on a real open-source software system.

We considered a pull request from "atomix" software repository and manually triggered RefBot to process it. Figure 7.3 represents part of the file quality table in the solution report page, which is generated for the selected pull request. It shows the TQI grade for the changed files before and after creating the pull request alongside

[1] https://sites.google.com/view/ase2019refbot

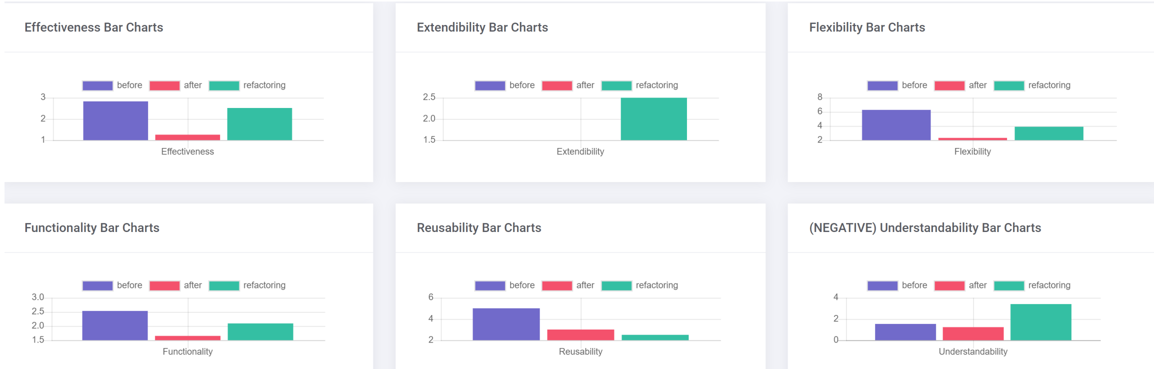


Figure 7.4: The quality bar charts in file report page for all six quality attributes.

with the impact of the recommended refactoring solution on the quality. As an example, the quality of the second file is degraded from 4.05 (B) to 1.18 (C). The solution which RefBot found for the pull request contains seven refactoring operations applied to this file. These refactorings could improve the file quality to 5.72 (B).

The user can view the detailed report page for each file. The bar charts in the file report page are provided in Figure 7.4. It shows the quality changes after the pull request and the refactoring solution impact for each of the six quality attributes, individually. We can observe that the recommended refactoring solution improves 5 out of 6 quality attributes for the file compared to the pull request quality.

Another section in the file report page is shown in Figure 7.5. It lists the refactoring operations from the recommended solution which have a controlling parameter applied to the selected file. The developer can interact with this list and reject or apply (code mark/auto options are as a popup window) each of the refactorings.

Additionally, the developer can further investigate each of the refactorings by viewing the refactoring report page. Figure 7.6 represents the abstract code changes after applying the selected refactoring on the source and target classes. This report can facilitate the decision making of users and help them to understand the changes in the structure introduced by a specific refactoring.

When a developer completes the interaction and analysis, the pull request is up-

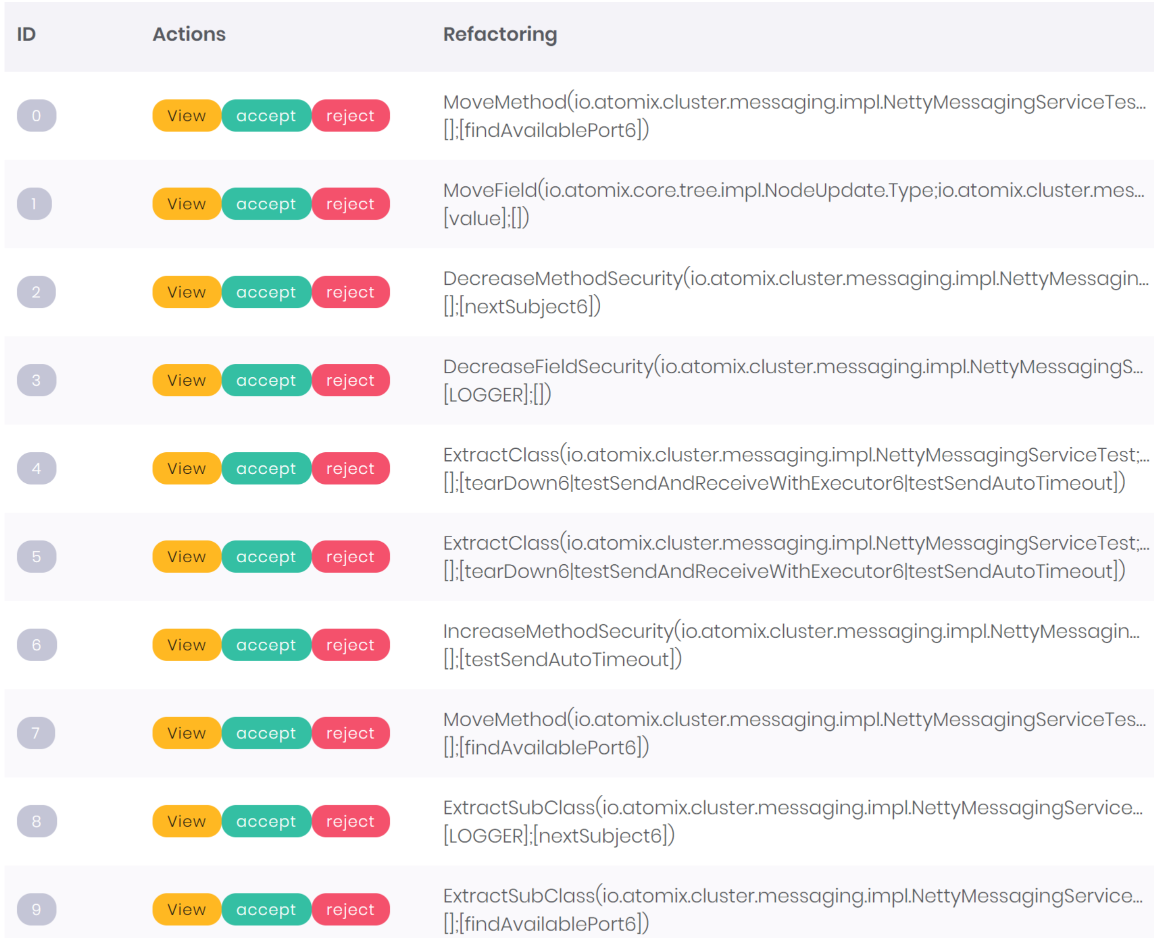


Figure 7.5: The list of refactoring operations recommended for a single file.

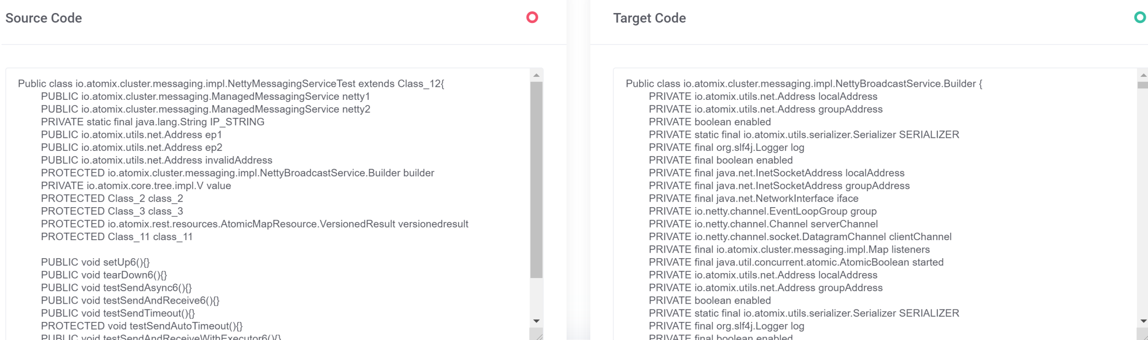


Figure 7.6: The code abstraction of source and target classes after applying a specific refactoring.

dated in the software repository, including the feedbacks on the refactorings. For any refactoring that applied as a code marker, the instructions are added to the top of the related files. Figure 7.7 depicts an example of the format of these markers.

```
1  +->RefactoringNumber->309848<-
   MoveField(io.atomix.core.map.impl.AbstractAtomicMapService.MapEntryValue.Type.TransactionScope.IteratorContext;io.a
   tomix.agent.AtomixAgent;[sessionId];[])
2  +->RefactoringNumber->309852<-ExtractSuperClass(io.atomix.agent.AtomixAgent;Class_1;[];[createParser4])
3  +->RefactoringNumber->309857<-ExtractSuperClass(io.atomix.agent.AtomixAgent;Class_4;[];
   [buildAtomix4|buildRestService4|parseArgs4|parseMemberId6])
4  +->RefactoringNumber->309863<-ExtractSuperClass(io.atomix.agent.AtomixAgent;Class_7;[];[main6])
5  +->RefactoringNumber->309864<-ExtractSuperClass(io.atomix.agent.AtomixAgent;Class_8;[];[createLogger4])
6  +->RefactoringNumber->309865<-ExtractSuperClass(io.atomix.agent.AtomixAgent;Class_9;[];[parseAddress6])
7  +->RefactoringNumber->309866<-ExtractSuperClass(io.atomix.agent.AtomixAgent;Class_10;[];[createConfig4])
8  +<-endRefactoring marker->
9   /*
10   * Copyright 2017-present Open Networking Foundation
11   *
```

Figure 7.7: The refactoring instructions related to a single file are added to the source code as a marker style.

7.3 Validation

We define three categories of research questions to evaluate RefBot and compare it to state-of-the-art techniques for automated refactoring:

- **RQ1: Quality improvement.** *To what extent can our refactoring bot improve the quality of software systems as compared to existing automated refactoring techniques?* In RQ1, we use the internal quality attributes [46] and code smells as proxies to assess the quality improvement brought by the refactoring operations generated by the RefBot for a set of selected pull-requests on different systems. We compare the performance of our approach (MO-MFO) with two, state-of-the-art, refactoring techniques: Ouni *et al.* [2] and JDeodorant [5]. Ouni *et al.* [2] proposed an automated multi-objective refactoring formulation based on NSGA-II using an aggregation of quality metrics while reducing the number of refactorings. JDeodorant [5] is an Eclipse plugin able to detect code smells and automatically recommend refactorings to fix them. JDeodorant is not based on the use of heuristics search. As JDeodorant supports a lower number of refactoring types with respect to the ones we considered, we restrict our

comparison with it to these refactorings. We have also limited the comparison to the changed files in the pull-requests.

- **RQ2: Refactoring meaningfulness.** *Are the refactoring recommendations produced by the RefBot meaningful from a developer's point of view? How do they compare with those generated by existing automated refactoring techniques?* Using antipatterns or internal quality indicators as proxies for code quality (as we do in RQ1) has substantial limitations. For this reason, in RQ1, we survey 25 developers asking for their opinion about the meaningfulness of the refactorings recommended by our technique and by the automated refactoring competitive technique [2]. In RQ2, we do not compare with JDeodorant since we preferred to focus on the most similar competitive technique in the literature to better study the advantages brought by the refactoring bot. The main substantial difference between RefBot and the approach by Ouni *et al.* [2] is indeed the interactive and incremental approach of the refactoring bot to focus on pull-requests.

- **RQ3: Industrial validation.** *To what extent can RefBot support of refactoring in a real-world continuous integration setting?* We integrated a beta version of Refbot into a previously licensed refactoring tool and asked one of our industrial partners to use it for a limited period of 3 business days (with six developers involved) on their regular pull-request after installing the bot on their repository. During this period, we checked the ability of RefBot to select relevant refactorings for the recent pull-requests introduced by the programmers during their daily activities.

The *context* of our study is represented by the seven systems in Table 7.1. We selected these seven systems for our validation because they range from medium to large-size projects and have been actively developed over the past 10 years. JDI[2] is

[2]Company anonymized for double-blind.

Table 7.1: Statistics of the studied systems.

System	Release	#classes	#smells	KLOC
Xerces-J	v2.7.0	991	91	240
JHotDraw	v7.5.1	585	25	21
JFreeChart	v1.0.18	521	72	170
GanttProject	v1.11.1	245	49	41
JDI	v5.8	638	88	247
Apache Ant	v1.8.2	1191	112	255
Rhino	v1.7.5	305	69	42

an industrial project for which 6 of the developers involved in the JDI maintenance agreed to take part in our experiments.

Table 7.1 provides information about the size of the subject systems (in terms of the number of classes and KLOC), and the number of code smells affecting them as detected with the rules defined in [97].

7.3.1 Data Collection

We present the data collection and analysis process grouped by research question category.

To address **RQ1**, we calculated *NF* as the percentage of code smells fixed by the refactoring solutions generated by the three considered approaches, over the total number of code smells which are affecting recent pull-requests of the subject systems. We selected the latest ten pull-requests for each of the open-source systems while a total of 8 pull-requests were opened during the three business days of the RefBot trial by our industrial partner. The detection of code smells before/after applying a refactoring solution was performed with the rules defined in [97]. The considered code smells are *Blob, Feature Envy (FE), Data Class (DC), Spaghetti Code (SC), Functional Decomposition (FD), and Shotgun Surgery (SS).*

Since the concept of code smell is very subjective (*i.e.*, different developers may have different opinions on whether a code component is smelly or not) [166], we also use more objective metrics to assess the quality of the refactorings generated by

the experimental approaches. We adopted the G metric based on *QMOOD* [46] that estimates the quality improvement of the system by comparing the quality before and after refactoring independently from the number of fixed design defects. Six quality factors are considered by *QMOOD*: reusability, flexibility, extendibility, functionality, understandability and effectiveness. All of them are formalized using a set of quality metrics. Hence, the total gain in quality G for each of the considered *QMOOD* quality attributes q_i before and after refactoring can be estimated as:

$$G = \frac{\sum_{i=1}^{6} G_{q_i}}{6} \ \textit{where}\ G_{q_i} = q_i' - q_i \tag{7.2}$$

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

To answer **RQ2** we asked 25 developers to evaluate the meaningfulness of the refactorings recommended by RefBot and by the approach of Ouni *et al.* [2] for pull-requests on the seven subject systems. Before explaining the study design for RQ2, it is important to remember that both the experimental techniques generate output sequences of refactoring operations that make sense when considered together rather than when looking at them in isolation. However, it is not an option to ask a developer to assess the meaningfulness of all the refactoring operations generated for a given system. For this reason, we started by filtering for each system the sequences of refactoring operations impacting the files of a set of pull-requests to make a fair comparison between both tools. Then, the developers manually evaluated the outcomes of both tools for each pull-request.

Each participant was then asked to assess the meaningfulness of the sequences of refactoring operations. Since on six of the seven systems (all but JDI) we involved external developers (*i.e.*, professional developers who did not take part in the development of the subject system), we made sure that each participant only evaluated refactoring sequences recommended by the two competitive techniques on one specific

system (*e.g.* JHotDraw). The rationale for such a choice is that an external developer would need time to acquire a system's knowledge by inspecting its code, and we did not want participants to comprehend the code from four different systems since this would introduce a strong tiring effect in our study.

To answer **RQ3**, the six developers of the JDI project evaluated the refactoring sequences generated for that system, since here we wanted to exploit their experience as original developers of the system. They used RefBot, as a beta version tool, during a period of 3 days instead of a refactoring tool that we licensed to their company in the past. Our industrial partner was motivated to try out RefBot since they are interested in upgrading their current quality assessment tool to another one that can support DevOps like our RefBot. They also expressed a concern about the lack of customization and high configuration effort/training required by existing automated refactoring tools.

To support such a complex experimental design, we built a Java Web-app that automatically assigns the refactored pull-requests to be evaluated to the developers. The Web-app showed each participant one sequence of refactoring operations on a single page, providing the developer with (i) the list of refactorings (*e.g.* move method m_i to class C_j, then push down field f_k to subclass C_j, *etc.*), (ii) the code of the classes impacted by the sequence of refactorings, and (iii) the complete code of the system subject of the refactoring with the description of the opened pull-request and the generated refactoring pull-request by the refactoring bot. The web page showing the refactoring sequence asked participants the question *Would you apply the proposed refactorings?* with a choice between *no* (*i.e.*, the refactoring sequence is not meaningful), *maybe* (*i.e.*, the refactoring sequence is meaningful, but the quality improvement it brings does not justify changing the code), or *yes* (*i.e.*, the refactoring sequence is meaningful and should be implemented). Moreover, participants were allowed to leave a comment justifying their assessment (this was optional). The

Table 7.2: Participants involved in RQ2.

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

Web-app was also in charge of:

Balancing the evaluations per system. We made sure that each system received roughly the same number of participants evaluating the different refactored pull-requests (files associated/modified by these pull-requests) by the two approaches.

Keeping track of the time spent by participants in the evaluation of each refactoring sequence/refactoring pull-request. The time spent by participants was counted in seconds since the moment the Web-app showed the refactoring on the screen to the moment in which the participant submitted their assessment. This feature was done to remove participants from our data set who did not spend a reasonable amount of time in evaluating the refactorings. We consider less than 60 seconds a reasonable threshold to remove noise (*i.e.*, we removed all evaluation sessions in which the participant spent less than 60 seconds in analyzing a single refactoring sequence).

Collecting demographic information about the participants. We asked their programming experience (in years) overall and in Java, and a self-assessment of their refactoring experience (from very low to very high).

Table 7.2 shows the participants involved in our study and how they were distributed in the evaluation of the refactoring sequences generated on the seven systems.

For the three days industrial validation, we integrated a routine in our RefBot to record all the actions of the 6 developers including the number of applied and rejected

refactorings, number of selected test cases, the introduced code changes and commit messages.

7.3.2 Experimental Setting and Data Analysis

For each algorithm and each system, we performed a set of experiments using several population sizes: 50, 100, 200, and 300. Then, we specified the maximum chromosome length (maximum number of operations/test cases per solution). The resulting vector length is proportional to the number of refactorings that are considered, and the size of the program to refactor. Based on those considerations, the upper and lower bounds on the chromosome length were set to 10 and 350, respectively. The stopping criterion was set to 10,000 fitness evaluations for all algorithms to ensure fairness. In order to have significant results, for each couple (algorithm, system), we use the trial and error method [134] for parameter configuration.

Concerning RQ2, we report the percentage of refactoring sequences assessed with a *no*, *maybe*, or *yes* by developers for each treatment (*i.e.*, RefBot and Ouni system [2]). Then, we discuss interesting comments left by developers when justifying their assessment.

7.3.3 Results

RQ1: Quality improvement. Figures 7.8 and 7.9 provide the percentage of fixed code smells (NF) and the quality gain (G) based on the *QMOOD* model, respectively. The average NF on the seven systems is 91% with peaks of ∼96% for JHotDraw and GanttProject.

The recommended refactorings also improved the G metric values (Figure 7.9) of the seven systems. The average quality gain for the Rhino system was the highest among the seven systems with 0.43. The improvement in the quality gain shows that the recommended refactorings help to optimize different quality metrics. Besides,

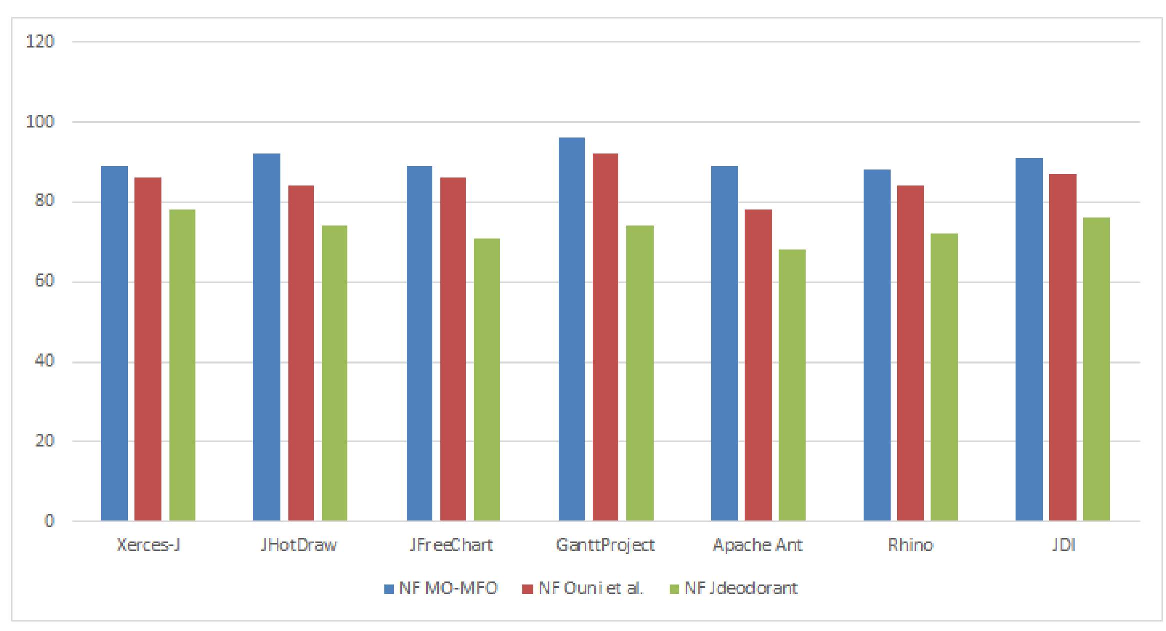


Figure 7.8: Median percentage of fixed code smells (NF) on the different pull-requests of the seven systems.

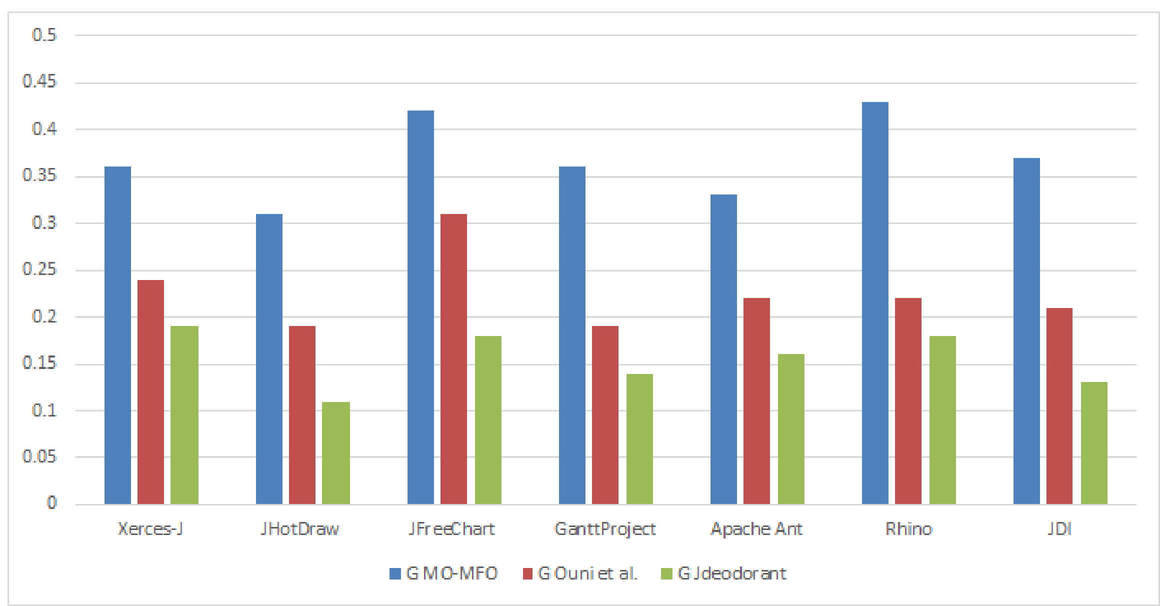


Figure 7.9: Median quality gain (G) on the different pull-requests of the seven systems.

the performance of RefBot is superior as compared to the competitive refactoring techniques [2, 5], even though the difference in terms of fixed code smells is not that marked (Figure 7.8). This latter result is also due to the fact that RefBot does not only recommend refactoring operations aimed at removing code smells it also focuses

Table 7.3: RQ2: Would you apply the proposed refactorings of the generated refactoring pull-request?

Approach	no	maybe	yes
RefBot	4/68 (5%)	11/68 (16%)	53/68 (77%)
Ouni *et al.* [2]	29/83 (34%)	41/83 (49%)	13/83 (15%)

on refactoring classes not affected by code smells but were changed during recent pull-requests. For example, in a manual investigation of the refactorings recommended by RefBot for JFreeChart, we found that 17 of the impacted classes do not exhibit any criticality as indicated by code smells and they were still improved in terms of quality attributes.

RQ2: Refactoring meaningfulness. Table 7.3 summarizes the manual refactoring evaluation results obtained from the 25 participants. Note that there is a slight deviation between the total number of refactorings evaluated by the two approaches (68 *vs* 83) since we did not consider for the data analysis the evaluations in which participants spent less than 60 seconds to assess the meaningfulness of the refactoring sequence under analysis and also the approach of Ouni et al. tends to generate much more refactorings on the analyzed files from the pull-requests.

The analysis of the quality by the Refactoring Bot improved the relevance of the recommended refactorings compared to the fully automated multi-objective approach. Indeed, the percentage of meaningful recommendations (*i.e.*, the sum of the *maybe* and *yes* answers) is much better for RefBot comparing to Ouni et al. (94% for RefBot and 66% for Ouni *et al.*). The percentage of refactorings that participants believe must be applied (*i.e.*, *yes* answers) is significantly higher for Refbot as well (77% *vs* 15%).

By looking at the comments left by participants when justifying their assessment, four out of the six original developers of the JDI system highlighted in their comments for three refactoring sequences that they found the refactorings relevant because it is improving the modularity of a class that they frequently modify in all the most recent

pull-requests. For example, one of the developers wrote in a comment: *"That is a very good recommendation, I spent days working on this class recently there, so I like this move method very much and extract sub-class. It will improve the reusability a lot as highlighted by the explanations of the bot"*. We found this comment as important qualitative evidence of the value of our refactoring bot in terms of analyzing the recently closed pull-requests to identify changed files and fix the identified quality issues in these files.

RQ3: Industry validation. Figures 7.8 and 7.9 summarize the results of deploying our RefBot during 3 business days to our industrial partner on the JDI repository. The six developers used the bot as part of their daily programming activities instead of a previously licensed refactoring tool. The tool was deployed as a Git app that connects automatically to a private GitHub repository whenever some code changes are introduced by the developers to check for refactorings and generate a new pull-request for the review of developers.

Overall, the achieved results confirm the effectiveness of our bot to generate efficient refactoring pull-requests. We found that the developers approved 9 out of 11 refactoring pull-requests generated by the bot during the three days. For the two remaining pull-requests, we found that a total of 7 out of 11 refactorings were approved. The achieved results confirm the basic intuition behind this work, showing that developers are more motivated to apply refactorings when the tool is easy to integrate within their development pipeline. The six developers also confirmed that they feel more comfortable in applying refactorings due to the high level of control proposed by the bot to review the generated pull-request which gives them more confidence and trust to the tool. This may explain the reason why a good number of recommended refactorings were applied.

7.4 Threats to Validity

Our refactoring bot mainly focuses on the recent pull-requests, but developers may have different priorities based on their current context. However, the developers can modify the configuration of our bot to focus on commits, branches, specific files or developers' contributions. Another internal threat is related to the used quality attributes since developers may want to express different preferences than QMOOD, or they want to tune them based on their needs or how critical is the code.

Construct validity is concerned with the relationship between theory and what is observed. To evaluate the results of our approach, we selected a set of pull-requests when comparing with other techniques, but may perform better on other pull-requests where the quality of them are different.

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

7.5 Conclusion

We presented a first attempt to propose an intelligent software refactoring bot, as GitHub app, that can submit a pull-request to refactor recent code changes. The salient feature of the proposed bot is that it incorporates interaction support, via our Web app, hence allowing developers to approve or modify or reject the applied code refactoring. The refactoring bot also provides support to explain why the refactorings are applied by quantifying the quality improvements. To evaluate the effectiveness of our technique, we applied it to four open-source and one industrial projects compar-

ing it with state-of-the-art approaches. Our results show promising evidence on the usefulness of the proposed interactive refactoring bot. The participants highlighted the high usability of the bot in terms of easy integration with their development environments with the least configuration effort.

Future work will involve validating our technique with additional refactoring types, programming languages, quality issues and participation from practitioners to investigate the general applicability of the proposed methodology.

CHAPTER VIII

Conclusion

Refactoring is nowadays widely adopted in the industry because bad design decisions can be very costly and extremely risky. On the one hand, automated refactoring does not always lead to the desired design. On the other hand, manual refactoring is error-prone, time-consuming and not practical for radical changes. Thus, recent research trends in the field focused on integrating developers feedback into automated refactoring recommendations because developers understand the problem domain intuitively and may have a clear target design in mind. However, this interactive process can be repetitive, expensive, and tedious since developers must evaluate recommended refactorings, and adapt them to the targeted design especially in large systems where the number of possible strategies can grow exponentially.

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 refactoring approaches are discussed.

8.1 Summary

In **Chapter I** and **Chapter II**, we defined the problem and the challenges of code refactoring, the contributions of this thesis, required background (including multi-objective optimization, software refactoring, code quality, *etc.*), and state-of-the-art

and related works to our approaches.

In **Chapter III**, we proposed a refactoring recommendation approach that dynamically adapts and interactively suggests refactorings to developers and takes their feedback into consideration. Our approach uses NSGAII to find a set of good refactoring solutions that improve software quality while minimizing the deviation from the initial design. These refactoring solutions are then analyzed to extract interesting common features between them such as the frequently occurring refactorings in the best non-dominated solutions.

Based on this analysis, the refactorings are ranked and suggested to the developer in an interactive fashion as a sequence of transformations. The developer can approve, modify or reject each of the recommended refactorings, and this feedback is then used to update the proposed rankings of recommended refactorings. After a number of introduced code changes and interactions with the developer, the interactive NSGA-II algorithm is executed again on the new modified system to repair the set of refactoring solutions based on the new changes and the feedback received from the developer. We evaluated our approach on a set of eight open source systems and two industrial projects provided by an industrial partner. Statistical analysis of our experiments shows that our dynamic interactive refactoring approach performed significantly better than four existing search-based refactoring techniques and one fully-automated refactoring tool not based on heuristic search.

In **Chapter IV**, we proposed an interactive approach combining the use of multi-objective and unsupervised learning to reduce the developer's interaction effort when refactoring systems. We generate, first, using multi-objective search different possible refactoring strategies by finding a trade-off between several conflicting quality attributes. Then, an unsupervised learning algorithm clusters the different trade-off solutions, called the Pareto front, to guide the developers in selecting their region of interests and reduce the number of refactoring options to explore. The feedback

from the developer, both at the cluster and solution levels, are used to automatically generate constraints to reduce the search space in the next iterations and focus on the region of developer preferences. We selected 14 active developers to manually evaluate the effectiveness our tool on 5 open source projects and one industrial system. The results show that the participants found their desired refactorings faster and more accurate than the current state of the art.

Refactoring studies either aggregated quality metrics to evaluate possible code changes or treated them separately to find trade-offs. For the first category of work, it is challenging to define upfront the weights for the quality objectives since developers are not able to express them upfront. For the second category of work, the number of possible trade-offs between quality objectives is large which makes developers reluctant to look at many refactoring solutions.

Therefore, in **Chapter V**, we proposed, for the first time, a way to convert multi-objective search into a mono-objective one after interacting with the developer to identify a good refactoring solution based on his preferences. The first step consists of using a multi-objective search to generate different possible refactoring strategies by finding a trade-off between several conflicting quality attributes. Then, an unsupervised learning algorithm clusters the different trade-off solutions, called the Pareto front, to guide the developers in selecting their region of interests and to reduce the number of refactoring options to explore. Finally, the extracted preferences from the developer are used to transform the multi-objective search into a mono-objective one by taking the preferred cluster of the Pareto front as the initial population for the mono-objective search and generating an evaluation function based on the weights that are automatically computed from the position of the cluster in the Pareto front. Thus, the developer will just interact with only one refactoring solution generated by the mono-objective search. We selected 32 participants to manually evaluate the effectiveness of our tool on 7 open source projects and one industrial project. The

results show that the recommended refactorings are more accurate than the current state of the art.

Due to the conflicting nature of quality measures, there are always multiple refactoring options to fix quality issues. Thus, interaction with developers is critical to inject their preferences. While several interactive techniques have been proposed, developers still need to examine large numbers of possible refactorings, which makes the interaction time-consuming. Furthermore, existing interactive tools are limited to the "objective space" to show developers the impacts of refactorings on quality attributes. However, the "decision space" is also important since developers may want to focus on specific code locations.

To give developers more insight about the decision space, in **Chapter VI**, we proposed an interactive approach that enables developers to pinpoint their preference simultaneously in the objective (quality metrics) and decision (code location) spaces. Developers may be interested in looking at refactoring strategies that can improve a specific quality attribute, such as extendibility (objective space), but they are related to different code locations (decision space). A plethora of solutions is generated at first using multi-objective search that tries to find the possible trade-offs between quality objectives. Then, an unsupervised learning algorithm clusters the trade-off solutions based on their quality metrics, and another clustering algorithm is applied to each cluster of the objective space to identify solutions related to different code locations. The objective and decision spaces can now be explored more efficiently by the developer, who can give feedback on a smaller number of solutions. This feedback is then used to generate constraints for the optimization process, to focus on the developer's regions of interest in both the decision and objective spaces. The manual validation of selected refactoring solutions by developers confirms that our approach outperforms state of the art refactoring techniques.

Finally, **Chapter VII** is dedicated to our Refactoring Bot. The adoption of

refactoring techniques for continuous integration received much less attention from the research community comparing to root-canal refactoring to fix the quality issues in the whole system. Several recent empirical studies show that developers, in practice, are applying refactoring incrementally when they are fixing bugs or adding new features. There is an urgent need for refactoring tools that can support continuous integration and some recent development processes such as DevOps that are based on rapid releases. Furthermore, several studies show that manual refactoring is expensive and existing automated refactoring tools are challenging to configure and integrate into the development pipelines with significant disruption cost.

Therefore, in **Chapter VII**, we proposed, for the first time, an intelligent software refactoring bot, called RefBot. Integrated into the version control system (e.g. GitHub), our bot continuously monitors the software repository, and it is triggered by any "open" or "merge" action on pull requests. The bot analyzes the files changed during that pull request to identify refactoring opportunities using a set of quality attributes then it will find the best sequence of refactorings to fix the quality issues if any. The bot recommends all these refactorings through an automatically generated pull-request. The developer can review the recommendations and their impacts in a detailed report and select the code changes that he wants to keep or ignore. After this review, the developer can close and approve the merge of the bot's pull request. We quantitatively and qualitatively evaluated the performance and effectiveness of RefBot by a survey conducted with experienced developers who used the bot on both open source and industry projects.

8.2 Future Work

While code-level refactoring has been widely studied and is well supported by tools, understanding refactoring rationale, or why developers should apply recommended refactorings, is less well understood. Without a rigorous understanding of

the rationale for refactoring, existing refactoring recommendation tools will continue to suffer from a high false-positive rate and limited relevance for developers. If, however, refactoring rationale can be identified automatically, this can be used to guide refactoring recommendations to be more purposeful and less ad hoc.

Moreover, once these refactorings have been applied, it is time-consuming for developers to manually document them. However, most existing approaches to automatic generation of documentation focus on functional changes, which are easier to generate from code changes.

Some future works direction can be summarized as follows:

1. **Analyzing Refactoring Rationale:**

   We need to understand and characterize real-world refactoring rationale. This will guide future research on refactoring by better understanding: (1) developer intentions when refactoring, (2) potential inconsistencies between developer intentions and actual refactorings, and (3) when and how developers document their refactorings. The primary challenges of this research include collecting representative documented refactorings; designing taxonomies and finding the keywords that are suitable for characterizing refactoring rationale; and mining large repositories, bug reports and communications data, to identify the kinds of refactorings that are actually successful in targeting developer intentions.

   We have to answer "***How do programmers refactor source code?***". This question targets the physical process that programmers follow to refactor the source code. In our previous works, we used the history of changes of several open source and industrial systems to understand how programmers identify refactoring opportunities and fix these detected quality issues. We propose extending this research to search for patterns that are commonly used in eye movements, and mouse/keyboard cursor strikes when refactoring the source-code. These patterns will highlight the areas of code that are important, as

well as areas that are less important, for program refactoring. Currently, the identification of refactoring opportunities and recommendation of useful refactorings are not very well understood. The majority of program refactoring tools rely on assumptions based on structural quality metrics and fully automated recommendations, but recent studies strongly suggest that these tools make incorrect assumptions.

2. **Enabling Context-Driven Refactoring Recommendations:**
   We need to determine how the generated knowledge from previous step can aid us in finding relevant refactorings based on context. Without guidance on which path to take, refactoring choices can be challenging for developers. Given that fully automated refactoring rarely meets developer needs, we can develop an interactive refactoring recommendation system via natural language support. For this purpose, finding refactoring recommendations based on up-front developer preferences is crucial. Based on our current research results, preferences can be extracted from the history of code changes, bug reports, communication data, commit messages, and pull-request descriptions.

The above mentioned future line of works demonstrate the vast potentiality of the intelligent software refactoring, which needs to be further evaluated. Our research team in **Intelligent Software Engineering Laboratory (ISE LAB)** will continue investigating the possible applications and techniques that can improve the performance and viability of our interactive and intelligent software refactoring tools.

BIBLIOGRAPHY

[1] M. Kessentini, W. Kessentini, H. A. Sahraoui, M. Boukadoum, and A. Ouni, "Design defects detection and correction by example," in *The 19th IEEE International Conference on Program Comprehension, ICPC 2011, Kingston, ON, Canada, June 22-24, 2011*, pp. 81–90, IEEE Computer Society, 2011.

[2] A. Ouni, M. Kessentini, H. A. Sahraoui, K. Inoue, and K. Deb, "Multi-criteria code refactoring using search-based software engineering: An industrial case study," *ACM Trans. Softw. Eng. Methodol.*, vol. 25, no. 3, pp. 23:1–23:53, 2016.

[3] M. Harman and L. Tratt, "Pareto optimal search based refactoring at the design level," in *Genetic and Evolutionary Computation Conference, GECCO 2007, Proceedings, London, England, UK, July 7-11, 2007* (H. Lipson, ed.), pp. 1106–1113, ACM, 2007.

[4] M. K. O'Keeffe and M. Ó. Cinnéide, "Search-based refactoring for software maintenance," *J. Syst. Softw.*, vol. 81, no. 4, pp. 502–516, 2008.

[5] M. Fokaefs, N. Tsantalis, E. Stroulia, and A. Chatzigeorgiou, "Jdeodorant: identification and application of extract class refactorings," in *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011* (R. N. Taylor, H. C. Gall, and N. Medvidovic, eds.), pp. 1037–1039, ACM, 2011.

[6] 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.

[7] L. Kelion, "Airbus a400m plane crash linked to software fault." http://www.bbc.com/news/technology-32810273. (accessed Apr. 1, 2020).

[8] CBC-News, "General motors recalling 4.3 million vehicles for airbag defect." http://www.cbc.ca/news/business/general-motors-recall-airbag-software-1.3755030, Apr. 2020. (accessed Apr. 1, 2020).

[9] M. Fowler, *Refactoring - Improving the Design of Existing Code.* Addison Wesley object technology series, Addison-Wesley, 1999.

[10] W. F. Opdyke, *Refactoring Object-Oriented Frameworks.* PhD thesis, USA, 1992.

[11] W. G. Griswold, *Program Restructuring as an Aid to Software Maintenance.* PhD thesis, USA, 1992. UMI Order No. GAX92-03258.

[12] L. Xiao, Y. Cai, R. Kazman, R. Mo, and Q. Feng, "Identifying and quantifying architectural debt," in *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016* (L. K. Dillon, W. Visser, and L. Williams, eds.), pp. 488–498, ACM, 2016.

[13] O. Bjuhr, K. Segeljakt, M. Addibpour, F. Heiser, and R. Lagerström, "Software architecture decoupling at ericsson," in *2017 IEEE International Conference on Software Architecture Workshops, ICSA Workshops 2017, Gothenburg, Sweden, April 5-7, 2017*, pp. 259–262, IEEE Computer Society, 2017.

[14] F. A. Fontana, R. Roveda, M. Zanoni, C. Raibulet, and R. Capilla, "An experience report on detecting and repairing software architecture erosion," in *13th Working IEEE/IFIP Conference on Software Architecture, WICSA 2016, Venice, Italy, April 5-8, 2016*, pp. 21–30, IEEE Computer Society, 2016.

[15] I. D. Baxter, A. Yahin, L. M. de Moura, M. Sant'Anna, and L. Bier, "Clone detection using abstract syntax trees," in *1998 International Conference on Software Maintenance, ICSM 1998, Bethesda, Maryland, USA, November 16-19, 1998*, pp. 368–377, IEEE Computer Society, 1998.

[16] H. C. Gall, K. Hajek, and M. Jazayeri, "Detection of logical coupling based on product release history," in *1998 International Conference on Software Maintenance, ICSM 1998, Bethesda, Maryland, USA, November 16-19, 1998*, pp. 190–197, IEEE Computer Society, 1998.

[17] 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, 2005, Lisbon, Portugal, September 5-9, 2005* (M. Wermelinger and H. C. Gall, eds.), pp. 187–196, ACM, 2005.

[18] S. Kim and M. D. Ernst, "Prioritizing warning categories by analyzing software history," in *Fourth International Workshop on Mining Software Repositories, MSR 2007 (ICSE Workshop), Minneapolis, MN, USA, May 19-20, 2007, Proceedings*, p. 27, IEEE Computer Society, 2007.

[19] T. Kamiya, S. Kusumoto, and K. Inoue, "Ccfinder: A multilinguistic token-based code clone detection system for large scale source code," *IEEE Trans. Software Eng.*, vol. 28, no. 7, pp. 654–670, 2002.

[20] N. Moha, Y. Guéhéneuc, L. Duchien, and A. L. Meur, "DECOR: A method for the specification and detection of code and design smells," *IEEE Trans. Software Eng.*, vol. 36, no. 1, pp. 20–36, 2010.

[21] M. Kessentini, S. Vaucher, and H. A. Sahraoui, "Deviance from perfection is a better criterion than closeness to evil when identifying risky code," in *ASE 2010, 25th IEEE/ACM International Conference on Automated Software Engineering, Antwerp, Belgium, September 20-24, 2010* (C. Pecheur, J. Andrews, and E. D. Nitto, eds.), pp. 113–122, ACM, 2010.

[22] R. L. B. 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 *Proceedings of the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2009, October 25-29, 2009, Orlando, Florida, USA* (S. Arora and G. T. Leavens, eds.), pp. 97–116, ACM, 2009.

[23] D. Dig, C. Comertoglu, D. Marinov, and R. E. Johnson, "Automated detection of refactorings in evolving components," in *ECOOP 2006 - Object-Oriented Programming, 20th European Conference, Nantes, France, July 3-7, 2006, Proceedings* (D. Thomas, ed.), vol. 4067 of *Lecture Notes in Computer Science*, pp. 404–428, Springer, 2006.

[24] A. Ouni, M. Kessentini, and H. A. Sahraoui, "Search-based refactoring using recorded code changes," in *17th European Conference on Software Maintenance and Reengineering, CSMR 2013, Genova, Italy, March 5-8, 2013* (A. Cleve, F. Ricca, and M. Cerioli, eds.), pp. 221–230, IEEE Computer Society, 2013.

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

[26] K. Stroggylos and D. Spinellis, "Refactoring-does it improve software quality?," in *Proceedings of the 5th International Workshop on Software Quality, WoSQ@ICSE 2007, Minneapolis, Minnesota, USA, May 20, 2007*, p. 10, IEEE Computer Society, 2007.

[27] A. Kaur and M. Kaur, "Analysis of code refactoring impact on software quality," in *MATEC Web of Conferences*, vol. 57, p. 02012, EDP Sciences, 2016.

[28] W. Ma, L. Chen, Y. Zhou, and B. Xu, "Do we have a chance to fix bugs when refactoring code smells?," in *International Conference on Software Analysis, Testing and Evolution, SATE 2016, Kunming, China, November 3-4, 2016*, pp. 24–29, IEEE, 2016.

[29] G. Bavota, B. D. Carluccio, A. D. Lucia, M. D. Penta, R. Oliveto, and O. Strollo, "When does a refactoring induce bugs? an empirical study," in *12th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2012, Riva del Garda, Italy, September 23-24, 2012*, pp. 104–113, IEEE Computer Society, 2012.

[30] W. Tracz, "Refactoring for software design smells: Managing technical debt by girish suryanarayana, ganesh samarthyam, and tushar sharma," *ACM SIGSOFT Software Engineering Notes*, vol. 40, no. 6, p. 36, 2015.

[31] B. Alshammari, C. J. Fidge, and D. Corney, "Assessing the impact of refactoring on security-critical object-oriented designs," in *17th Asia Pacific Software Engineering Conference, APSEC 2010, Sydney, Australia, November 30 - December 3, 2010* (J. Han and T. D. Thu, eds.), pp. 186–195, IEEE Computer Society, 2010.

[32] M. Drozdz, D. G. Kourie, B. W. Watson, and A. Boake, "Refactoring tools and complementary techniques," in *2006 IEEE/ACS International Conference on Computer Systems and Applications (AICCSA 2006), March 8-11, Dubai/Sharjah, UAE*, pp. 685–688, IEEE Computer Society, 2006.

[33] D. S. Batory, J. N. Sarvela, and A. Rauschmayer, "Scaling step-wise refinement," *IEEE Trans. Software Eng.*, vol. 30, no. 6, pp. 355–371, 2004.

[34] Y. Cai and K. J. Sullivan, "Modularity analysis of logical design models," in *21st IEEE/ACM International Conference on Automated Software Engineering (ASE 2006), 18-22 September 2006, Tokyo, Japan*, pp. 91–102, IEEE Computer Society, 2006.

[35] M. Kessentini, V. Alizadeh, and M. W. Mkaouer, "Interactive and dynamic search based approach to software refactoring recommendations," Oct. 17 2019. US Patent App. 16/386,551.

[36] V. Alizadeh and M. Kessentini, "Reducing interactive refactoring effort via clustering-based multi-objective search," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018* (M. Huchard, C. Kästner, and G. Fraser, eds.), pp. 464–474, ACM, 2018.

[37] V. Alizadeh, H. Fehri, and M. Kessentini, "Less is more: From multi-objective to mono-objective refactoring via developer's knowledge extraction," in *19th International Working Conference on Source Code Analysis and Manipulation, SCAM 2019, Cleveland, OH, USA, September 30 - October 1, 2019*, pp. 181–192, IEEE, 2019.

[38] V. Alizadeh, H. Fehri, M. Kessentini, and R. Kazman, "Enabling decision and objective space exploration for interactive multi-objective refactoring.," *IEEE Transactions on Software Engineering*, 2020.

[39] V. Alizadeh, M. A. Ouali, M. Kessentini, and M. Chater, "Refbot: Intelligent software refactoring bot," in *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*, pp. 823–834, IEEE, 2019.

[40] S. Rebai, O. B. Sghaier, V. Alizadeh, M. Kessentini, and M. Chater, "Interactive refactoring documentation bot," in *19th International Working Conference on Source Code Analysis and Manipulation, SCAM 2019, Cleveland, OH, USA, September 30 - October 1, 2019*, pp. 152–162, IEEE, 2019.

[41] M. Farina, K. Deb, and P. Amato, "Dynamic multiobjective optimization problems: test cases, approximations, and applications," *IEEE Trans. Evolutionary Computation*, vol. 8, no. 5, pp. 425–442, 2004.

[42] K. Deb, A. Sinha, P. J. Korhonen, and J. Wallenius, "An interactive evolutionary multiobjective optimization method based on progressively approximated value functions," *IEEE Trans. Evolutionary Computation*, vol. 14, no. 5, pp. 723–739, 2010.

[43] I. Karahan and M. Köksalan, "A territory defining multiobjective evolutionary algorithms and preference incorporation," *IEEE Trans. Evolutionary Computation*, vol. 14, no. 4, pp. 636–664, 2010.

[44] E. Fernández, E. Lopez, S. Bernal, C. A. C. Coello, and J. Navarro, "Evolutionary multiobjective optimization using an outranking-based dominance generalization," *Comput. Oper. Res.*, vol. 37, no. 2, pp. 390–395, 2010.

[45] T. Wagner and H. Trautmann, "Integration of preferences in hypervolume-based multiobjective evolutionary algorithms by means of desirability functions," *IEEE Trans. Evolutionary Computation*, vol. 14, no. 5, pp. 688–701, 2010.

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

[47] B. D. Bois, S. Demeyer, and J. Verelst, "Refactoring – improving coupling and cohesion of existing code," in *11th Working Conference on Reverse Engineering, WCRE 2004, Delft, The Netherlands, November 8-12, 2004*, pp. 144–151, IEEE Computer Society, 2004.

[48] E. R. Murphy-Hill and A. P. Black, "Refactoring tools: Fitness for purpose," *IEEE Software*, vol. 25, no. 5, pp. 38–44, 2008.

[49] E. R. Murphy-Hill and A. P. Black, "Programmer-friendly refactoring errors," *IEEE Trans. Software Eng.*, vol. 38, no. 6, pp. 1417–1431, 2012.

[50] E. R. Murphy-Hill and A. P. Black, "Breaking the barriers to successful refactoring: observations and tools for extract method," in *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008* (W. Schäfer, M. B. Dwyer, and V. Gruhn, eds.), pp. 421–430, ACM, 2008.

[51] X. Ge and E. R. Murphy-Hill, "Manual refactoring changes with automated refactoring validation," in *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014* (P. Jalote, L. C. Briand, and A. van der Hoek, eds.), pp. 1095–1105, ACM, 2014.

[52] X. Ge and E. R. Murphy-Hill, "Benefactor: a flexible refactoring tool for eclipse," in *Companion to the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011* (C. V. Lopes and K. Fisher, eds.), pp. 19–20, ACM, 2011.

[53] S. R. Foster, W. G. Griswold, and S. Lerner, "Witchdoctor: IDE support for real-time auto-completion of refactorings," in *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland* (M. Glinz, G. C. Murphy, and M. Pezzè, eds.), pp. 222–232, IEEE Computer Society, 2012.

[54] L. Tahvildari and K. Kontogiannis, "A metric-based approach to enhance design quality through meta-pattern transformation," in *7th European Conference on Software Maintenance and Reengineering (CSMR 2003), 26-28 March 2003, Benevento, Italy, Proceedings*, pp. 183–192, IEEE Computer Society, 2003.

[55] D. Dig, "A refactoring approach to parallelism," *IEEE Software*, vol. 28, no. 1, pp. 17–22, 2011.

[56] Y. Kataoka, M. D. Ernst, W. G. Griswold, and D. Notkin, "Automated support for program refactoring using invariants," in *2001 International Conference on Software Maintenance, ICSM 2001, Florence, Italy, November 6-10, 2001*, pp. 736–743, IEEE Computer Society, 2001.

[57] A. Ghannem, M. Kessentini, and G. El-Boussaidi, "Detecting model refactoring opportunities using heuristic search," in *Center for Advanced Studies on Collaborative Research, CASCON '11, Toronto, ON, Canada, November 7-10, 2011* (J. W. Ng, C. Couturier, M. Litoiu, and E. Stroulia, eds.), pp. 175–187, IBM / ACM, 2011.

[58] M. Kessentini, R. Mahouachi, and K. Ghédira, "What you like in design use to correct bad-smells," *Software Quality Journal*, vol. 21, no. 4, pp. 551–571, 2013.

[59] A. Ghannem, G. El-Boussaidi, and M. Kessentini, "Model refactoring using examples: a search-based approach," *J. Softw. Evol. Process.*, vol. 26, no. 7, pp. 692–713, 2014.

[60] M. Kessentini, A. Ouni, P. Langer, M. Wimmer, and S. Bechikh, "Search-based metamodel matching with structural and syntactic measures," *J. Syst. Softw.*, vol. 97, pp. 1–14, 2014.

[61] 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 *Search-Based Software Engineering - 6th International Symposium, SSBSE 2014, Fortaleza, Brazil, August 26-29, 2014. Proceedings* (C. L. Goues and S. Yoo, eds.), vol. 8636 of *Lecture Notes in Computer Science*, pp. 31–45, Springer, 2014.

[62] 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.

[63] H. Wang, M. Kessentini, and A. Ouni, "Bi-level identification of web service defects," in *Service-Oriented Computing - 14th International Conference, IC-SOC 2016, Banff, AB, Canada, October 10-13, 2016, Proceedings* (Q. Z. Sheng, E. Stroulia, S. Tata, and S. Bhiri, eds.), vol. 9936 of *Lecture Notes in Computer Science*, pp. 352–368, Springer, 2016.

[64] A. Ouni, M. Kessentini, M. Ó. Cinnéide, H. A. Sahraoui, K. Deb, and K. Inoue, "MORE: A multi-objective refactoring recommendation approach to introducing design patterns and fixing code smells," *J. Softw. Evol. Process.*, vol. 29, no. 5, 2017.

[65] V. Alizadeh, M. Kessentini, and B. R. Maxim, "Refactoring support for variability-intensive systems," in *Software Engineering for Variability Intensive Systems - Foundations and Applications* (I. Mistrík, M. Galster, and B. R. Maxim, eds.), pp. 275–294, Auerbach Publications / Taylor & Francis, 2019.

[66] M. Harman, S. A. Mansouri, and Y. Zhang, "Search-based software engineering: Trends, techniques and applications," *ACM Comput. Surv.*, vol. 45, no. 1, pp. 11:1–11:61, 2012.

[67] O. Seng, J. Stammel, and D. Burkhart, "Search-based determination of refactorings for improving the class structure of object-oriented systems," in *Genetic and Evolutionary Computation Conference, GECCO 2006, Proceedings, Seattle, Washington, USA, July 8-12, 2006* (M. Cattolico, ed.), pp. 1909–1916, ACM, 2006.

[68] H. Kiliç, E. Koc, and I. Cereci, "Search-based parallel refactoring using population-based direct approaches," in *Search Based Software Engineering - Third International Symposium, SSBSE 2011, Szeged, Hungary, September 10-12, 2011. Proceedings* (M. B. Cohen and M. Ó. Cinnéide, eds.), vol. 6956 of *Lecture Notes in Computer Science*, pp. 271–272, Springer, 2011.

[69] A. Ouni, M. Kessentini, H. A. Sahraoui, and M. S. Hamdi, "Search-based refactoring: Towards semantics preservation," in *28th IEEE International Conference on Software Maintenance, ICSM 2012, Trento, Italy, September 23-28, 2012*, pp. 347–356, IEEE Computer Society, 2012.

[70] M. Ó. Cinnéide, L. Tratt, M. Harman, S. Counsell, and I. H. Moghadam, "Experimental assessment of software metrics using automated refactoring," in *2012 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM '12, Lund, Sweden - September 19 - 20, 2012* (P. Runeson, M. Höst, E. Mendes, A. A. Andrews, and R. Harrison, eds.), pp. 49–58, ACM, 2012.

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

[72] M. Hall, N. Walkinshaw, and P. McMinn, "Supervised software modularisation," in *28th IEEE International Conference on Software Maintenance, ICSM 2012, Trento, Italy, September 23-28, 2012*, pp. 472–481, IEEE Computer Society, 2012.

[73] G. Bavota, F. Carnevale, A. D. Lucia, M. D. Penta, and R. Oliveto, "Putting the developer in-the-loop: An interactive GA for software re-modularization," in *Search Based Software Engineering - 4th International Symposium, SSBSE 2012, Riva del Garda, Italy, September 28-30, 2012. Proceedings* (G. Fraser and J. T. de Souza, eds.), vol. 7515 of *Lecture Notes in Computer Science*, pp. 75–89, Springer, 2012.

[74] Y. Lin, X. Peng, Y. Cai, D. Dig, D. Zheng, and W. Zhao, "Interactive and guided architectural refactoring with search-based recommendation," in *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016* (T. Zimmermann, J. Cleland-Huang, and Z. Su, eds.), pp. 535–546, ACM, 2016.

[75] 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 *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014* (I. Crnkovic, M. Chechik, and P. Grünbacher, eds.), pp. 331–336, ACM, 2014.

[76] A. Ouni, M. Kessentini, H. A. Sahraoui, and M. S. Hamdi, "The use of development history in software refactoring using a multi-objective evolutionary algorithm," in *Genetic and Evolutionary Computation Conference, GECCO '13, Amsterdam, The Netherlands, July 6-10, 2013* (C. Blum and E. Alba, eds.), pp. 1461–1468, ACM, 2013.

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

[78] M. K. O'Keeffe and M. Ó. Cinnéide, "A stochastic approach to automated design improvement," in *Proceedings of the 2nd International Symposium on*

Principles and Practice of Programming in Java, PPPJ 2003, Kilkenny City, Ireland, June 16-18, 2003 (J. F. Power and J. Waldron, eds.), vol. 42 of *ACM International Conference Proceeding Series*, pp. 59–62, ACM, 2003.

[79] W. Mkaouer, M. Kessentini, A. Shaout, P. Koligheu, S. Bechikh, K. Deb, and A. Ouni, "Many-objective software remodularization using NSGA-III," *ACM Trans. Softw. Eng. Methodol.*, vol. 24, no. 3, pp. 17:1–17:45, 2015.

[80] M. Fleck, J. Troya, M. Kessentini, M. Wimmer, and B. Alkhazi, "Model transformation modularization as a many-objective optimization problem," *IEEE Trans. Software Eng.*, vol. 43, no. 11, pp. 1009–1032, 2017.

[81] A. Ouni, R. G. Kula, M. Kessentini, T. Ishio, D. M. Germán, and K. Inoue, "Search-based software library recommendation using multi-objective optimization," *Inf. Softw. Technol.*, vol. 83, pp. 55–75, 2017.

[82] A. Ouni, R. G. Kula, M. Kessentini, and K. Inoue, "Web service antipatterns detection using genetic programming," in *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2015, Madrid, Spain, July 11-15, 2015* (S. Silva and A. I. Esparcia-Alcázar, eds.), pp. 1351–1358, ACM, 2015.

[83] A. Ouni, M. Kessentini, S. Bechikh, and H. A. Sahraoui, "Prioritizing code-smells correction tasks using chemical reaction optimization," *Software Quality Journal*, vol. 23, no. 2, pp. 323–361, 2015.

[84] M. Kessentini, M. Wimmer, H. A. 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, Paris, France, June 14, 2010* (M. Aksit, E. Kindler, E. E. Roubtsova, and A. T. McNeile, eds.), p. 2, ACM, 2010.

[85] A. ben Fadhel, M. Kessentini, P. Langer, and M. Wimmer, "Search-based detection of high-level model changes," in *28th IEEE International Conference on Software Maintenance, ICSM 2012, Trento, Italy, September 23-28, 2012*, pp. 212–221, IEEE Computer Society, 2012.

[86] M. Kessentini, H. A. Sahraoui, M. Boukadoum, and M. Wimmer, "Search-based design defects detection by example," in *Fundamental Approaches to Software Engineering - 14th International Conference, FASE 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings* (D. Giannakopoulou and F. Orejas, eds.), vol. 6603 of *Lecture Notes in Computer Science*, pp. 401–415, Springer, 2011.

[87] M. Kessentini, A. Bouchoucha, H. A. Sahraoui, and M. Boukadoum, "Example-based sequence diagrams to colored petri nets transformation using heuristic search," in *Modelling Foundations and Applications, 6th European Conference, ECMFA 2010, Paris, France, June 15-18, 2010. Proceedings* (T. Kühne,

B. Selic, M. Gervais, and F. Terrier, eds.), vol. 6138 of *Lecture Notes in Computer Science*, pp. 156–172, Springer, 2010.

[88] M. Kim, T. Zimmermann, and N. Nagappan, "An empirical study of refactoringchallenges and benefits at microsoft," *IEEE Trans. Software Eng.*, vol. 40, no. 7, pp. 633–649, 2014.

[89] C. Lebeuf, M. D. Storey, and A. Zagalsky, "Software bots," *IEEE Software*, vol. 35, no. 1, pp. 18–23, 2018.

[90] C. Lebeuf, M. D. Storey, and A. Zagalsky, "How software developers mitigate collaboration friction with chatbots," *CoRR*, vol. abs/1702.07011, 2017.

[91] M. S. Wessel, B. M. de Souza, I. Steinmacher, I. S. Wiese, I. Polato, A. P. Chaves, and M. A. Gerosa, "The power of bots: Characterizing and understanding bots in OSS projects," *PACMHCI*, vol. 2, no. CSCW, pp. 182:1–182:19, 2018.

[92] S. Urli, Z. Yu, L. Seinturier, and M. Monperrus, "How to design a program repair bot?: insights from the repairnator project," in *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2018, Gothenburg, Sweden, May 27 - June 03, 2018* (F. Paulisch and J. Bosch, eds.), pp. 95–104, ACM, 2018.

[93] V. Balachandran, "Fix-it: An extensible code auto-fix component in review bot," in *13th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2013, Eindhoven, Netherlands, September 22-23, 2013*, pp. 167–172, IEEE Computer Society, 2013.

[94] M. Wyrich and J. Bogner, "Towards an autonomous bot for automatic source code refactoring," in *Proceedings of the 1st International Workshop on Bots in Software Engineering, BotSE@ICSE 2019, Montreal, QC, Canada, May 28, 2019* (E. Shihab and S. Wagner, eds.), pp. 24–28, IEEE / ACM, 2019.

[95] M. C. Feathers, "Working effectively with legacy code," in *Extreme Programming and Agile Methods - XP/Agile Universe 2004, 4th Conference on Extreme Programming and Agile Methods, Calgary, Canada, August 15-18, 2004, Proceedings* (C. Zannier, H. Erdogmus, and L. Lindstrom, eds.), vol. 3134 of *Lecture Notes in Computer Science*, p. 217, Springer, 2004.

[96] J. Kerievsky, "Refactoring to patterns," in *Extreme Programming and Agile Methods - XP/Agile Universe 2004, 4th Conference on Extreme Programming and Agile Methods, Calgary, Canada, August 15-18, 2004, Proceedings* (C. Zannier, H. Erdogmus, and L. Lindstrom, eds.), vol. 3134 of *Lecture Notes in Computer Science*, p. 232, Springer, 2004.

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

[98] A. Ouni, M. Kessentini, K. Inoue, and M. Ó. Cinnéide, "Search-based web service antipatterns detection," *IEEE Trans. Serv. Comput.*, vol. 10, no. 4, pp. 603–617, 2017.

[99] H. Wang, A. Ouni, M. Kessentini, B. R. Maxim, and W. I. Grosky, "Identification of web service refactoring opportunities as a multi-objective problem," in *IEEE International Conference on Web Services, ICWS 2016, San Francisco, CA, USA, June 27 - July 2, 2016* (S. Reiff-Marganiec, ed.), pp. 586–593, IEEE Computer Society, 2016.

[100] H. Wang, M. Kessentini, and A. Ouni, "Prediction of web services evolution," in *Service-Oriented Computing - 14th International Conference, ICSOC 2016, Banff, AB, Canada, October 10-13, 2016, Proceedings* (Q. Z. Sheng, E. Stroulia, S. Tata, and S. Bhiri, eds.), vol. 9936 of *Lecture Notes in Computer Science*, pp. 282–297, Springer, 2016.

[101] M. Daagi, A. Ouni, M. Kessentini, M. M. Gammoudi, and S. Bouktif, "Web service interface decomposition using formal concept analysis," in *2017 IEEE International Conference on Web Services, ICWS 2017, Honolulu, HI, USA, June 25-30, 2017* (I. Altintas and S. Chen, eds.), pp. 172–179, IEEE, 2017.

[102] H. Wang, M. Kessentini, T. Hassouna, and A. Ouni, "On the value of quality of service attributes for detecting bad design practices," in *2017 IEEE International Conference on Web Services, ICWS 2017, Honolulu, HI, USA, June 25-30, 2017* (I. Altintas and S. Chen, eds.), pp. 341–348, IEEE, 2017.

[103] M. Kessentini and A. Ouni, "Detecting android smells using multi-objective genetic programming," in *4th IEEE/ACM International Conference on Mobile Software Engineering and Systems, MOBILESoft@ICSE 2017, Buenos Aires, Argentina, May 22-23, 2017*, pp. 122–132, IEEE, 2017.

[104] A. Ouni, M. Daagi, M. Kessentini, S. Bouktif, and M. M. Gammoudi, "A machine learning-based approach to detect web service design defects," in *2017 IEEE International Conference on Web Services, ICWS 2017, Honolulu, HI, USA, June 25-30, 2017* (I. Altintas and S. Chen, eds.), pp. 532–539, IEEE, 2017.

[105] M. Kessentini, H. Wang, J. T. Dea, and A. Ouni, "Improving web services design quality using heuristic search and machine learning," in *2017 IEEE International Conference on Web Services, ICWS 2017, Honolulu, HI, USA, June 25-30, 2017* (I. Altintas and S. Chen, eds.), pp. 540–547, IEEE, 2017.

[106] M. Kim and D. Notkin, "Discovering and representing systematic code changes," in *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*, pp. 309–319, IEEE, 2009.

[107] Y. Cai, R. Kazman, C. Jaspan, and J. Aldrich, "Introducing tool-supported architecture review into software design education," in *26th International Conference on Software Engineering Education and Training, CSEE&T 2013, San Francisco, CA, USA, May 19-21, 2013* (T. Cowling, S. Bohner, and M. A. Ardis, eds.), pp. 70–79, IEEE, 2013.

[108] A. F. Yamashita and L. Moonen, "Do developers care about code smells? an exploratory survey," in *20th Working Conference on Reverse Engineering, WCRE 2013, Koblenz, Germany, October 14-17, 2013* (R. Lämmel, R. Oliveto, and R. Robbes, eds.), pp. 242–251, IEEE Computer Society, 2013.

[109] A. Telea and L. Voinea, "Visual software analytics for the build optimization of large-scale software systems," *Computational Statistics*, vol. 26, no. 4, p. 635, 2011.

[110] D. Sahin, M. Kessentini, S. Bechikh, and K. Deb, "Code-smell detection as a bilevel problem," *ACM Trans. Softw. Eng. Methodol.*, vol. 24, no. 1, pp. 6:1–6:44, 2014.

[111] 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, (FSE-22), Hong Kong, China, November 16 - 22, 2014* (S. Cheung, A. Orso, and M. D. Storey, eds.), pp. 763–766, ACM, 2014.

[112] Y. Lin and D. Dig, "A study and toolkit of CHECK-THEN-ACT idioms of java concurrent collections," *Softw. Test. Verification Reliab.*, vol. 25, no. 4, pp. 397–425, 2015.

[113] M. Kim, M. Gee, A. Loh, and N. Rachatasumrit, "Ref-finder: a refactoring reconstruction tool based on logic query templates," in *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2010, Santa Fe, NM, USA, November 7-11, 2010* (G. Roman and A. van der Hoek, eds.), pp. 371–372, ACM, 2010.

[114] R. Marinescu, "Detection strategies: Metrics-based rules for detecting design flaws," in *20th International Conference on Software Maintenance (ICSM 2004), 11-17 September 2004, Chicago, IL, USA*, pp. 350–359, IEEE Computer Society, 2004.

[115] J. Kim, D. S. Batory, D. Dig, and M. Azanza, "Improving refactoring speed by 10x," in *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016* (L. K. Dillon, W. Visser, and L. Williams, eds.), pp. 1145–1156, ACM, 2016.

[116] A. Ouni, M. Kessentini, H. A. Sahraoui, and M. Boukadoum, "Maintainability defects detection and correction: a multi-objective approach," *Autom. Softw. Eng.*, vol. 20, no. 1, pp. 47–79, 2013.

[117] M. D'Ambros, A. Bacchelli, and M. Lanza, "On the impact of design flaws on software defects," in *Proceedings of the 10th International Conference on Quality Software, QSIC 2010, Zhangjiajie, China, 14-15 July 2010* (J. Wang, W. K. Chan, and F. Kuo, eds.), pp. 23–31, IEEE Computer Society, 2010.

[118] M. Ó. Cinnéide, D. Boyle, and I. H. Moghadam, "Automated refactoring for testability," in *Fourth IEEE International Conference on Software Testing, Verification and Validation, ICST 2012, Berlin, Germany, 21-25 March, 2011, Workshop Proceedings*, pp. 437–443, IEEE Computer Society, 2011.

[119] D. Silva, N. Tsantalis, and M. T. Valente, "Why we refactor? confessions of github contributors," in *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016* (T. Zimmermann, J. Cleland-Huang, and Z. Su, eds.), pp. 858–870, ACM, 2016.

[120] W. H. Brown, R. C. Malveau, H. W. cCormick, and T. J. Mowbray, *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. USA: John Wiley Sons, Inc., 1st ed., 1998.

[121] 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.

[122] S. Negara, N. Chen, M. Vakilian, R. E. Johnson, and D. Dig, "A comparative study of manual and automated refactorings," in *ECOOP 2013 - Object-Oriented Programming - 27th European Conference, Montpellier, France, July 1-5, 2013. Proceedings* (G. Castagna, ed.), vol. 7920 of *Lecture Notes in Computer Science*, pp. 552–576, Springer, 2013.

[123] M. Kim, T. Zimmermann, and N. Nagappan, "A field study of refactoring challenges and benefits," in *20th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-20), SIGSOFT/FSE'12, Cary, NC, USA - November 11 - 16, 2012* (W. Tracz, M. P. Robillard, and T. Bultan, eds.), p. 50, ACM, 2012.

[124] L. Yang, T. Kamiya, K. Sakamoto, H. Washizaki, and Y. Fukazawa, "Refactoringscript: A script and its processor for composite refactoring," in *The 26th International Conference on Software Engineering and Knowledge Engineering, Hyatt Regency, Vancouver, BC, Canada, July 1-3, 2013* (M. Reformat, ed.), pp. 711–716, Knowledge Systems Institute Graduate School, 2014.

[125] K. L. Beck, *Test-driven Development - by example*. The Addison-Wesley signature series, Addison-Wesley, 2003.

[126] M. Fowler, J. Highsmith, *et al.*, "The agile manifesto," *Software Development*, vol. 9, no. 8, pp. 28–35, 2001.

[127] K. Deb, S. Agrawal, A. Pratap, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: NSGA-II," *IEEE Trans. Evolutionary Computation*, vol. 6, no. 2, pp. 182–197, 2002.

[128] K. Deb and A. Srinivasan, "Innovization: innovating design principles through optimization," in *Genetic and Evolutionary Computation Conference, GECCO 2006, Proceedings, Seattle, Washington, USA, July 8-12, 2006* (M. Cattolico, ed.), pp. 1629–1636, ACM, 2006.

[129] V. Alizadeh, "An interactive and dynamic search-based approach to software refactoring recommendations." http://kessentini.net/tse18, Apr. 2018. (accessed Apr. 1, 2020).

[130] A. Arcuri and G. Fraser, "Parameter tuning or default values? an empirical investigation in search-based software engineering," *Empirical Software Engineering*, vol. 18, no. 3, pp. 594–623, 2013.

[131] R. Jackson, C. Carter, and M. Tarsitano, "Trial-and-error solving of a confinement problem by a jumping spider, portia fimbriata," *Behaviour*, vol. 138, no. 10, pp. 1215–1234, 2001.

[132] G. Keppel and T. Wickens, "Simultaneous comparisons and the control of type i errors," *Design and analysis: A researcher's handbook*, pp. 111–130, 2004.

[133] 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.

[134] A. Arcuri and L. C. Briand, "A practical guide for using statistical tests to assess randomized algorithms in software engineering," in *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011* (R. N. Taylor, H. C. Gall, and N. Medvidovic, eds.), pp. 1–10, ACM, 2011.

[135] G. Bavota, A. D. Lucia, A. Marcus, R. Oliveto, and F. Palomba, "Supporting extract class refactoring in eclipse: The ARIES project," in *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland* (M. Glinz, G. C. Murphy, and M. Pezzè, eds.), pp. 1419–1422, IEEE Computer Society, 2012.

[136] Y. Cai and K. J. Sullivan, "A formal model for automated software modularity and evolvability analysis," *ACM Trans. Softw. Eng. Methodol.*, vol. 21, no. 4, pp. 21:1–21:29, 2012.

[137] T. Caliński and J. Harabasz, "A dendrite method for cluster analysis," *Communications in Statistics-theory and Methods*, vol. 3, no. 1, pp. 1–27, 1974.

[138] R. A. Redner and H. F. Walker, "Mixture densities, maximum likelihood and the em algorithm," *SIAM review*, vol. 26, no. 2, pp. 195–239, 1984.

[139] V. Alizadeh, "Reducing interactive refactoring effort via clustering-based multi-objective search." https://sites.google.com/view/ase2018, Apr. 2018. (accessed Apr. 1, 2020).

[140] R. Kazman, Y. Cai, R. Mo, Q. Feng, L. Xiao, S. Haziyev, V. Fedak, and A. Shapochka, "A case study in locating the architectural roots of technical debt," in *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 2* (A. Bertolino, G. Canfora, and S. G. Elbaum, eds.), pp. 179–188, IEEE Computer Society, 2015.

[141] S. J. Carrière, R. Kazman, and I. Ozkaya, "A cost-benefit framework for making architectural decisions in a business context," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2, ICSE 2010, Cape Town, South Africa, 1-8 May 2010* (J. Kramer, J. Bishop, P. T. Devanbu, and S. Uchitel, eds.), pp. 149–157, ACM, 2010.

[142] "The developer Coefficient." https://stripe.com/reports/developer-coefficient-2018, Apr. 2020. (accessed Apr. 1, 2020).

[143] 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.

[144] M. Kessentini, T. J. Dea, and A. Ouni, "A context-based refactoring recommendation approach using simulated annealing: two industrial case studies," in *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2017, Berlin, Germany, July 15-19, 2017* (P. A. N. Bosman, ed.), pp. 1303–1310, ACM, 2017.

[145] I. H. Moghadam and M. Ó. Cinnéide, "Automated refactoring using design differencing," in *16th European Conference on Software Maintenance and Reengineering, CSMR 2012, Szeged, Hungary, March 27-30, 2012* (T. Mens, A. Cleve, and R. Ferenc, eds.), pp. 43–52, IEEE Computer Society, 2012.

[146] D. Dig, J. Marrero, and M. D. Ernst, "Refactoring sequential java code for concurrency via concurrent libraries," in *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*, pp. 397–407, IEEE, 2009.

[147] V. Alizadeh, "Less is More: From Multi-Objective to Mono-Objective Refactoring." https://sites.google.com/view/scam2019, Apr. 2020. (accessed Apr. 1, 2020).

[148] "2015 ieee 7th international workshop on managing technical debt (mtd)." http://www.sei.cmu.edu/community/td2015/, Oct. 2015.

[149] L. Tokuda and D. S. Batory, "Evolving object-oriented designs with refactorings," in *The 14th IEEE International Conference on Automated Software Engineering, ASE 1999, Cocoa Beach, Florida, USA, 12-15 October 1999*, p. 174, IEEE Computer Society, 1999.

[150] E. R. Murphy-Hill and A. P. Black, "Why don't people use refactoring tools?," in *1st Workshop on Refactoring Tools, WRT 2007, in conjunction with 21st European Conference on Object-Oriented Programming, July 30 - August 03, 2007, Berlin, Germany, Proceedings* (D. Dig, ed.), pp. 60–61, 2007.

[151] R. Moser, W. Pedrycz, and G. Succi, "A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction," in *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008* (W. Schäfer, M. B. Dwyer, and V. Gruhn, eds.), pp. 181–190, ACM, 2008.

[152] L. C. Briand, J. Wüst, S. V. Ikonomovski, and H. Lounis, "Investigating quality factors in object-oriented designs: An industrial case study," in *Proceedings of the 1999 International Conference on Software Engineering, ICSE' 99, Los Angeles, CA, USA, May 16-22, 1999* (B. W. Boehm, D. Garlan, and J. Kramer, eds.), pp. 345–354, ACM, 1999.

[153] 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.

[154] M. O'Keeffe and M. Ó. Cinnéide, "Search-based refactoring: an empirical study," *Journal of Software Maintenance*, vol. 20, no. 5, pp. 345–364, 2008.

[155] V. Alizadeh, "Enabling decision and objective space exploration for interactive multi-objective refactoring." https://sites.google.com/view/tse2020decision, Apr. 2020. (accessed Apr. 1, 2020).

[156] R. S. Arnold, *Software change impact analysis*. IEEE Computer Society Press, 1996.

[157] T. Mens and T. Tourwé, "A survey of software refactoring," *IEEE Trans. Software Eng.*, vol. 30, no. 2, pp. 126–139, 2004.

[158] E. Mealy, D. A. Carrington, P. A. Strooper, and P. Wyeth, "Improving usability of software refactoring tools," in *18th Australian Software Engineering Conference (ASWEC 2007), April 10-13, 2007, Melbourne, Australia*, pp. 307–318, IEEE Computer Society, 2007.

[159] M. W. Mkaouer, M. Kessentini, S. Bechikh, K. Deb, and M. Ó. Cinnéide, "High dimensional search-based software engineering: finding tradeoffs among 15 objectives for automating software refactoring using NSGA-III," in *Genetic and*

Evolutionary Computation Conference, GECCO '14, Vancouver, BC, Canada, July 12-16, 2014 (D. V. Arnold, ed.), pp. 1263–1270, ACM, 2014.

[160] J. Simmonds and T. Mens, "A comparison of software refactoring tools," *Programming Technology Lab*, 2002.

[161] X. Ge, Q. L. DuBose, and E. R. Murphy-Hill, "Reconciling manual and automatic refactoring," in *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland* (M. Glinz, G. C. Murphy, and M. Pezzè, eds.), pp. 211–221, IEEE Computer Society, 2012.

[162] M. Vakilian, N. Chen, S. Negara, B. A. Rajkumar, B. P. Bailey, and R. E. Johnson, "Use, disuse, and misuse of automated refactorings," in *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland* (M. Glinz, G. C. Murphy, and M. Pezzè, eds.), pp. 233–243, IEEE Computer Society, 2012.

[163] G. Szoke, C. Nagy, L. J. Fülöp, R. Ferenc, and T. Gyimóthy, "Faultbuster: An automatic code smell refactoring toolset," in *15th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2015, Bremen, Germany, September 27-28, 2015* (M. W. Godfrey, D. Lo, and F. Khomh, eds.), pp. 253–258, IEEE Computer Society, 2015.

[164] F. Palomba, G. Bavota, M. D. Penta, F. Fasano, R. Oliveto, and A. D. Lucia, "On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation," *Empirical Software Engineering*, vol. 23, no. 3, pp. 1188–1221, 2018.

[165] H. M. Olague, L. H. Etzkorn, S. Gholston, and S. Quattlebaum, "Empirical validation of three software metrics suites to predict fault-proneness of object-oriented classes developed using highly iterative or agile software development processes," *IEEE Trans. Software Eng.*, vol. 33, no. 6, pp. 402–419, 2007.

[166] F. Palomba, G. Bavota, M. D. Penta, R. Oliveto, and A. D. Lucia, "Do they really smell bad? A study on developers' perception of bad code smells," in *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014*, pp. 101–110, IEEE Computer Society, 2014.