Mining, Understanding and Integrating User Preferences in Software Refactoring Using Computational Search, Machine Learning, and Dimensionality Reduction

by

Troh Josselin Dea

**A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Information Systems Engineering)
in the University of Michigan-Dearborn
2017**

Troh Josselin Dea

deatroh@umich.edu

ORCID iD: 0000-0002-1665-4661

DEDICATION

This thesis is in the honor of my family and friends. A special feeling of gratitude to my cousins Theophile Batoua and Gomman Serges Olivier Bagui for his wise advice in difficult time, though they are miles away from me in the Ivory Coast.

I also dedicate this dissertation to my church families. They supported me throughout the process. In particular, I thanks my Lord and Savior Jesus Christ for all the members of Word of Faith International Christian Center (WOFICC) in Southfield, and those at Promise Land Church Ministries in Baltimore. I will always appreciate all the wisdom they have imparted to me as well as their prayers.

Finally, this thesis is in the memories of my mother Blah Wenda Jolie and my father Dea Trogbeu. My mother played both the role of mother and father due to my father's sickness. She beautifully mixed both motherly love and rigor to raise me to who I am today. I really miss her.

ACKNOWLEDGEMENTS

I would like to express my deepest gratitude to my advisor, Dr. Marouane Kessentini, for his excellent guidance, patience, and for providing me with an excellent atmosphere for doing research. This thesis would not have been possible without his support.

I would like to thank all the committee members, Professor William Grosky, Professor Bruce Maxim, Dr. Kiumi Akingbehin, and Dr. Alex (Ya Sha) Yi, for their additional guidance and mentoring throughout the different stages of completing this degree.

I would like to thanks Emily Wang and Don Barbarci for me, in particular, and for all the other international students. They have been a lovely family to me, turning a foreign environment into a joyful home. In time of great obstacles, they were around us with words that gave hope. I thank them from the bottom of my heart.

I would also like to give a special thank to Katharine Markotan at the Department of Computer and Information Science, Professor William Grosky and Professor and Dean Anthony England for their support during my PhD. They listened to me, and did more than their assignments required of them.

Finally, I would like to thank all the members of the SBSE Lab. In particular, I would like to thanks Usman Mansoor, Dilan Sahin and Mohamed Wiem for turning my America into a home country at many occasions.

PREFACE

The research that led to this thesis was performed at Search-Based Software Engineering Laboratory at the Department of Computer and Information Science, University of Michigan-Dearborn, with Prof. Marouane Kessentini as the main advisor.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

Search-Based Software Engineering (SBSE) is a software development practice which focuses on couching software engineering problems as optimization problems using meta-heuristic techniques to automate the search for near optimal solutions to those problems. While SBSE has been successfully applied to a wide variety of software engineering problems, our understanding of the extent and nature of how software engineering problems can be formulated as automated or semi-automated search is still lacking. The majority of software engineering solutions are very subjective and present difficulties to formally define fitness functions to evaluate them. Current studies focus on guiding the search of optimal solutions rather than performing it. It is unclear yet the degree of interaction required with software engineers during the optimization process and how to reduce it. In this work, we focus on search-based software maintenance and evolution problems including software refactoring and software remodularization to improve the quality of systems. We propose to address the following challenges:

- A major challenge in adapting a search-based technique for a software engineering problem is the definition of the fitness function. In most cases, fitness functions are ill-defined or subjective.

- Most existing refactoring studies do not include the developer in the loop to analyze suggested refactoring solutions, and give their feedback during the optimization process. In addition, some quality metrics are cost-expensive leading to cost-expensive fitness functions. Moreover, while quality metrics evaluate the structural improvements of the refactored system, it is impossible to evaluate the semantic coherence of the design without user interactions.

- Finally, several metrics can be dependent and correlated, thus it may be possible to reduce the number of objectives/dimensions when addressing refactoring problems.

To address the above challenges, this work provides new techniques and tools to formulate software refactoring as scalable and learning-based search problem. We proposed novel interactive learning-based techniques using machine learning to incorporate developers knowledge and preferences in the search, resulting in more efficient and cost-effective search-based refactoring recommendation systems. We designed and implemented novel objective reduction SBSE methodologies to support scalable number of objectives. The proposed solutions were empirically evaluated in academic (open-source systems) and industrial settings.

KEYWORDS

Search-based Software Engineering, Software Refactoring, Interactive Optimization, Objective Reduction, Software Maintenance and Evolution, Machine Learning.

CHAPTER 1

Introduction

1.1 Context

Over the last century, the majority of machines used in industry and at home were transformed from mechanical hardware into fully integrated devices with complex software. With this transformation, competitions in all major industries has become very intense, and successful companies must constantly innovate by introducing new types of software or extending existing software with new features. Consequently, maintaining high product quality in such dynamic environment has become very challenging, leading software organizations to create standard processes to guide development teams towards successful design and implementation of software products. However, very often, these processes introduce conflicts between the various software stakeholders. On the one hand, sponsors and managers emphasize on meeting release deadlines and functional quality. On the other hand, software developers and programmers place great value on the structural or non-functional aspects of the software since it has long term impact on their performance and workflows. Thus, these conflict leads to timing and cost pressure on developers, causing them to release software with enormous amount of technical debt [4, 5, 6, 7], a metaphor for design defects introduced in software due to creating quick solutions to meet deadlines. On a long term, these debts leads to high maintenance cost estimated at up to 90% of the total software cost over the product lifecycle [8].

Historically, software maintenance activities were performed using manual operations.

The accuracy of this approach is relative to the expertise of the software engineer assigned to the task. Consequently, a very proficient software engineer results in better maintenance performance and quality while an inexperience engineer would worsen the overall quality of the system. Thus, manual maintenance is subject to the expertise of the engineer. Another problem in manual maintenance, even when performed by experienced software engineers, is the gradual degradation in accuracy for large system due to fatigue and strenuous effort required of the engineers. Moreover, this approach has inherent weaknesses including being excessively time consuming, unreasonable resource usage, lack of scalability, etc.

Within the last two decades, several researches have been conducted to solve software engineering problem (SE) from an automation point of view, and various solutions have been proposed. These solutions are driven by heuristic search algorithms, and are referred to as search-based software engineering (SBSE) methodologies [9]. By contrast with manual techniques, SBSE methodologies evolve a population of solutions to search for the optimal trade-off or Pareto-optimal front. They are able to generate diverse sets of solutions in a single execution of the underlying heuristic algorithms. SBSE methodologies cover the entire software development life-cycle, from requirement development to testing and maintenance. In the field of software maintenance, SBSE techniques have been successfully applied to software restructuring, and shown promising results in terms of accuracy, robustness, and scalability [10].

Refactoring has been established as the primary maintenance techniques for restructuring object-oriented (OO) software systems after they have been deployed. It is defined as the process of improving the internal structure of the software without changing its external behavior [11]. As a methodology that focuses on the improvement of the structure of systems, the main benefits of software refactoring are an increase of the understandability (i.e, maintenability), reusability and extensibility [12]. However, in some cases, refactoring operations aimed at removing code duplication can lead to reduction of maintenance cost such as those related to bug removal.

In general software refactoring is preceded by the detection of design or structural defects. Design defects can be found at both architectural and code source levels. In the literature, they are alternatively referred to as code smells [13], anomalies [14], antipatterns [15] or bad design practices [16]. They are design instances that violate good design practices such as designed patterns [17]. The proliferation of code smell within a software system can make difficult for new team members to understand a software systems. Thus, the presence of code smells has negative impact on the maintenance or evolution of the system. In some cases, codes smells can be responsible of bugs in the systems, and their presence is considered as a trigger for software refactoring. The detection of code smells is out of the scope of this dissertation. In the remainder of the thesis, it is assume that code smells have been identified using one of the existing design defect detection tools [18, 19, 20, 21, 22, 23].

1.2 Problem Statement

Various techniques have been proposed by researchers to address software refactoring problems. The majority of existing techniques can be classified, based on their level of automation, into three categories: (1) manual, (2) semi-automated, and (3) fully-automated.

In manual refactoring, code smells are detected, and corresponding refactoring operations are applied manually by the software engineer. This type of refactoring depends heavily on the expertise of the software engineer in charge of the maintenance activity. However, not only real-life software systems are very large in size, but the software development environment is increasingly dynamic - often spread across multiple countries and teams. In addition, software systems are continually evolving, and/or new features are frequently added to address new requirements. Such complexity in size, environment, and deadlines requires methodologies that are scalable and flexible.

Applications and advances in search-based optimization (SBO) have sparked new directions in software engineering. As results, researchers have proposed several approaches to solving the refactoring problem as search-based optimization problem using heuristic algorithms. This class of refactoring techniques is referred to as search-based software refactoring, and constitutes the bulk of semi- and fully-automated refactoring solutions.

Fully automated refactoring methodologies build on search-based optimizations algorithms, and do not include the domain expert (e.g., software engineer) in the process. The majority of existing automated refactoring techniques were formulated using single-objective optimization. The objective (e.g., quality metrics, number of a particular code smell) drives the quality of the final refactoring solutions. However, the quality of a software is defined by a variety of metrics, both internal and external. Thus, software refactoring problems are naturally multi-objective. Several techniques have been proposed to tailor multi-/many-objective fully-automated refactoring techniques. These refactoring techniques can generate a set of solutions or Pareto front from which the software engineer must choose based on his or her expertise and preferences. However, a large number of objective renders the decision making process very difficult. Furthermore, since fully-automated refactoring techniques work as black box solutions, most software engineers are reluctant to utilize them due to fear of introducing bugs and lack of control over the refactoring process. Today, it remains unclear whether such solutions are used by software organizations.

Semi-automated refactoring techniques seek to combine the expertise of the software engineer and the search capability of SBOs. These methodologies stand between manual and the extreme of fully-automated refactoring techniques, and allow for the integration of the user in the refactoring process. In certain form of semi-automated software refactoring, the user is in charge of manually identifying refactoring opportunities that is to find design with structural problems such as code smells. Then, once the bad code is found, an adaptation of the SBO algorithm performs the refactoring. Finally, the user decides if the solution proposed by the SBO must be applied to the system. Another subclass of semi-automated refactoring is often referred to as interactive software refactoring. Interactive methodologies integrate the user in the loop for specific tasks where he can affect the optimization process. These techniques are particularly important for increasing the confidence of the software engineer regarding the task of the SBO algorithm. However, most existing interactive software refactoring techniques are inefficient, due to requiring the software engineer at every iteration of the SBO algorithm. In addition, due to the strenuous effort placed on the software engineer at each iteration, these interactive techniques carries some of the weakness of the manual methods such as non-scalability to large systems and degradation of the performance of the software engineer with time.

In this context, the primarily goal of this thesis is to propose a set of methods to efficiently address the integration of the user preferences in both semi- and fully-automated search-based refactoring techniques. Secondarily, it addresses the question of decision making in high-dimensional search-based software refactoring. The techniques covered in this thesis can be classified under the broad umbrella of search-based software engineering. In particular, they address search-based software refactoring problems related to (1) Coding context, (2) Evaluation of refactoring activity, and (3) Objective Selection in automated refactoring.

1.2.1 Coding Context

Characterizing high-quality source code requires a large number of quality metric, each one defining a specific dimension of software quality. Researches in software refactoring have provided several techniques to improve the quality of software using these quality metrics. Quality metrics can quantify either system-level quality or or component level quality (e.g., class level quality).

Research Problem 1: A large portion of existing refactoring tools suggests refactorings to improve the overall quality of systems without a concrete prioritization plan

[24, 25]. As a result, the number of refactorings to apply can be large, and developers may spend a long time to select relevant refactorings. Consequently, most developers become reluctant to adopt the proposed refactoring tools.

Research Problem 2: When a large number of refactorings are recommended, manual refactoring becomes error-prone and time-consuming. Murphy-Hill et al., [24] showed that most developers do not use fully automated refactoring techniques because they want to mix refactorings with semantic changes, something that is not permitted by existing methods.

1.2.2 Evaluation of refactorings

The ultimate goal of software refactoring is to improve the quality of the system under maintenance. To this end, most exiting search-based software refactoring solutions are evaluated using software quality metrics. Consequently, at each stage of the execution of the SBO algorithm that drives the search, solutions are selected based on their individual score for every metric included as objective function. Thus, during the decision making process, the domain expert must choose the solution that best matches his preferences based on the values of these quality metrics.

Research Problem 3: Quality metrics are widely used to evaluate software quality by either using them directly as objectives, or using them in rules that characterize design defects. However, there is no general consensus on the definition of these design defects or code-smells, due to various programming behavior and contexts. For example, the definition of a large class can change from one software organization to another. Thus, it is difficult to formalize the definitions of design violations in terms of quality metrics to evaluate the quality of software refactoring solution.

Research Problem 4: The majority of existing refactoring studies do not include the developer - that is the final decision maker (DM) - in the loop to analyze the suggested refactoring solutions and give his or her feedback during the optimization process. In

[26], he authors used an interactive genetic algorithm to estimate the quality of refactoring solutions. However, the DM is required to evaluate every refactoring solution throughout the entire execution of the algorithm, making it fastidious and sometimes impractical.

1.2.3 Objective Selection in Automated Refactoring

The aim of SBSE research is to move software engineering problems from human-based search to machine-based search, using a variety of techniques from the fields of metaheuristic search, operations research and evolutionary computation paradigms. Thus, multi-objective evolutionary algorithms (MOEAs) have been widely applied to address several problems such as the generation of test cases, next release problems, software refactoring, model-driven engineering, etc [9]. In search-based software refactoring, the goal is to find a trade-off between different quality preferences of the developers(e.g., quality metrics). For better performance of MOEAs methodologies, it is desired that the objective functions be conflicting.

Research Problem 5: The different fitness functions are defined and selected by the developers among large number of quality metrics or their combinations. Thus, it is challenging to decide up-front of the execution of the search if these functions are conflicting or not. In most cases, developers use their intuition and expertise to define the fitness functions. Without a rigorous check of the possible correlation between the defined fitness functions, a diverse set of solutions cannot be generated. This is also true if some conflicting measures are aggregated into one fitness function. In other scenario, a large number of non-dominated solutions is generated, if non-conflicting measures are treated as separate fitness functions.

Research Problem 6: Several recent studies consider as many objective as possible using many-objective algorithms [2]. however the visualization of the solutions in the generated Pareto front is a challenge for the developers due to the large number of Pareto front solutions when high number of objectives are considered.

Research Problem 7: The consideration of large number of objectives that are not necessarily conflicting make the execution time of the algorithm long and the search slow. This results from the fact that the multi-objective algorithms behave similar to random search when the number of objectives increases.

1.3 Proposed Research Contributions

The main goal of this thesis is to bridge the gap between fully-automated search-based software refactoring and manual approaches by proposing efficient methodologies that enable the integration of the software domain expert in the search process. By including the domain expert in the search loop, it becomes possible to take advantage of his or her expertise early in the process, thus guiding the search algorithm towards refactoring solutions that take his or her preferences into account.

The proposed methodologies explore various techniques used in computational search, machine learning and statistical tool used in high dimensional data processing. These methodologies were evaluated using empirical studies based on several opensource projects as well as industrial project from research partners. Each proposed techniques can be used separately or in combination with others.

1.3.1 Contribution 1: Context-Based Software Refactoring

To solve research problem 1 and 2, we propose a search-based refactoring approach to find the best solution satisfying two objectives: maximizing the number of refactorings applied to buggy or recently modified classes, and minimizing the number of antipatterns using a set of antipatterns detection rules. We implemented our proposed approach and evaluated it on a set of two industrial systems provided by our industrial partner from the automotive industry. We did the evaluation only on these two systems since it is critical to evaluate the relevance of recommended refactorings by the original developers of the systems.

Statistical analysis of our experiments showed that our proposal performed significantly better than existing search-based refactoring approaches and an existing refactoring tool not based on heuristic search. In our qualitative analysis, we conducted a survey with the software developers who participated in our experiments to evaluate the relevance of the fixed quality violations in their daily development activities

1.3.2 Contribution 2: Interactive Software Refactoring

To solve problem 3 and 4, we model the domain expert preferences using machine learning. A predictive model based on artificial neural network is used to capture the expertise of the software engineer during the learning step. First, we propose a general study on search-based software refactoring. Then, an application of web service remodularization is deduced.

1.3.2.1 Interactive Software Refactoring: General Case

we propose a Genetic Algorithm (GA) based interactive learning algorithm for software refactoring based on Artificial Neural Networks (ANN). We model the decision maker's preferences as a predictive model using ANN to approximate the fitness function for the evaluation of refactoring solutions. The developer is asked to manually evaluate refactoring solutions suggested by a Genetic Algorithm (GA) for a few iterations. Then, these evaluated solutions are used as training set for the ANN, and finally the ANN model is used to evaluate subsequent refactoring solutions in the next iterations. We evaluate our approach on open-source systems using existing benchmark. We report the results on the efficiency and effectiveness of our approach, and compare it to existing refactoring methodologies.

1.3.2.2 Interactive Software Refactoring: Web Service Remodularization

We propose, in this chapter, a Genetic Algorithm (GA)-based interactive learning algorithm for Web services interface modularization based on Artificial Neural Networks (ANN). The proposed approach is based on the important feedback of the user to guide the search for relevant Web services modularization solutions using predictive models. To the best of our knowledge, the use of predictive models has not been used to improve the quality of Web services design. In the proposed approach, we model the user's design preferences using ANN as a predictive model to approximate the fitness function for the evaluation of the Web services modularization solutions. The user is asked to manually evaluate Web services interface modularization solutions suggested by a Genetic Algorithm (GA) for few iterations then these examples are used as a training set for the ANNs to evaluate the solutions of the GA in the next iterations. We evaluated our approach on a set of 82 real-world Web services, extracted from an existing benchmark. Statistical analysis of our experiments shows that our interactive approach performed significantly better than the state-of-the-art modularization techniques in terms of design improvements and fixing design defects in web services.

1.3.3 Contribution 3: Dimensionality Reduction in Many-Objective Search-Based Software Refactoring

We address problem 5-7 using the Principal Component Analysis (PCA) algorithm in conjunction with the well-known multi-objective algorithm NSGA-II [27], adapted to address the software refactoring problem [14, 28, 29]. We start from the hypothesis that there may be correlations among any two or more objectives (e.g. quality metrics) that are used to evaluate refactoring solutions. Our approach, based on the PCA-NSGA-II methodology [30], aims at finding the best and reduced set of objective that represents the quality metrics of interest to the domain expert. A regular NSGA-II algorithm with several objectives is executed for a number of iterations. Then a PCA component analyzes the correlation between the different objectives using the execution traces. The number of objectives may be reduced during the next iterations based on the PCA results. The process is repeated several times until a maximum number of iterations is reached.

We implemented our proposed approach and evaluated it on a set of seven open source systems. Statistical analysis of our experiments showed that dimensionality reduction reduced significantly the number of objectives on several case studies by a minimum of 4 objectives and a maximum of 8 objectives. It also generates a smaller number of non-dominated solutions and lower execution time comparing to existing many-objective refactoring techniques. The results show that the approach outperforms several of existing multi-objective refactoring techniques, where the objectives are not analyzed for possible correlations, based on several evaluation measures such as number of fixed anti-patterns and manual correctness.

CHAPTER 2

Related Work

2.1 Introduction

2.2 Search-Based Software Engineering

2.2.1 Introduction

Search-Based Software Engineering (SBSE) is the field of adapting computational search, mainly from the evolutionary algorithm literature, to solve Software Engineering (SE) problems [31]. Unlike traditional deterministic techniques, meta-heuristic search algorithms are used in SBSE methodologies to find a set of near-optimal or ”good” solutions. Applications of meta-heuristic search to software engineering problems appear as early as 1976 with the work of Miller and Spooner on floating point test data generation [32]. However, the term SBSE was first coined by Harman and Jones in a 2001 paper that proposed search-based optimization (SBO) as a general approach to software engineering [31]. It represents a landmark in the development of search-based software engineering that led to the explosion of the application of SBO to software engineering problems within the following decade. In a 2012 survey on SBSE, Harman et al., found that SBSE has covered all the activities of the software lifecycle, including project planning, testing, maintenance, etc. It was also reported in the same paper that the majority of SBSE studies focus on software testing, with a coverage of up to 54% of the overall publications.

The majority of existing works in SBSE treats software engineering problems as a single- or bi-objective optimization problem [9], where one or two objective functions are to be optimized. These objective functions are often formulated as a set of metrics [33, 21, 34]. Harman et al., argue that a combination of multiple metrics into a single scalar fitness function, using weighted sum, is not necessarily suitable for most software engineering problems [9]. In addition, with increasingly large software systems, most software engineering problems are becoming naturally complex in both size and structure. Consequently, real-life software engineering problems require to find a compromise between many evaluation criteria, and are naturally multi-objective.

2.2.2 Multi-/Many-Objective Search-Based Software Engineering

Solutions to SBSE problems, formulated as optimization problems with more than one objective, use Pareto optimality as an alternative to combining several objectives into one fitness function [2]. Based on the Pareto optimality, a solution x dominates another solution y, if x is not worse than y in all objectives and is strictly better than y in at least one objective. Thus, under Pareto optimality, SBSE techniques find a set of non-dominated solutions. Within a set of solutions that are non-dominated, no solution is either worse or better than another solution. It is the responsibility of the software engineer to select the solution that best matches his or her preferences.

In general, SBSE techniques that use Pareto optimality to search solutions are referred to as multi-objective SBSE techniques. In analytical form, they can be formulated as follows:

$$\begin{cases} \underset{\mathbf{x}}{min} & \mathbf{f}(\mathbf{x}) = [f_1(\mathbf{x}), f_2(\mathbf{x}), \ldots, f_M(\mathbf{x})]^T, M > 3 \\ & g_j(x) \geq 0, \qquad j = 1, \ldots, P \\ & h_k(x) = 0, \qquad k = 1, \ldots, Q \\ & x_i^L \leq x_i \leq x_i^U, \qquad i = 1, \ldots, n \end{cases} \tag{2.1}$$

where $\mathbf{f}(\mathbf{x})$ is the M-dimensional objective vector with $M > 1$, $f_i(\mathbf{x})$ is the $i-$th objective to minimize, $\mathbf{x}$ is the decision variable, P is the number of inequality constraints, Q is the number of equality constraints, and x_i^L and x_i^U are the lower and upper bound of the decision variable x_i, respectively. A decision variable satisfying the $(P + Q)$ constraints is said to be feasible, and the set of all feasible solutions defines the feasible search space denoted Ω.

A well-known weakness of multi-objective Pareto-based methodologies used in SBSE is the performance degradation due to large number of objectives [1]. Specifically, when the number of objectives is very large, most existing algorithms fail to distinguish solutions in the Pareto front. As results, the proportion of best non-dominated solutions become almost equal to 1 - that is the multi-objective algorithm tend to include all the population members into the Pareto front. Figure 2.1 illustrates this degradation and loss of performance. As a direct consequence of the performance degradation, the majority of existing multi-objective SBSE techniques are limited to up to 3 objectives. A recent literature survey, including 51 papers around the use of Pareto optimizers in SBSE, showed that more than 50% of the proposed methodologies address SE problems from only a bi-objective point of view [2]. In the same survey, it was reported that another 30% of the proposed methodologies considers only 3 objectives. Thus, at most 20% of existing multi-objective SBSE methodologies consider more than 4 objectives.

Recently, a number of SBSE techniques that can handle more than 3 objectives have been proposed. These techniques that can handle large number of objectives are referred to as many-objectives SBSE [35, 36]. In the field of SBSE, the work of Mkaouer et al., represents one of the few real-applications of many-objectives search-based optimization techniques. This is due to the difficulty of decision making that arises with very large number of objectives as well as novelty of such methodologies. Concerning the decision making aspect, the work described in this thesis include a technique aimed at reducing the number of objectives after the Pareto front as been identified in order to retain objectives that truly contribute to the distinction of one solution from another.

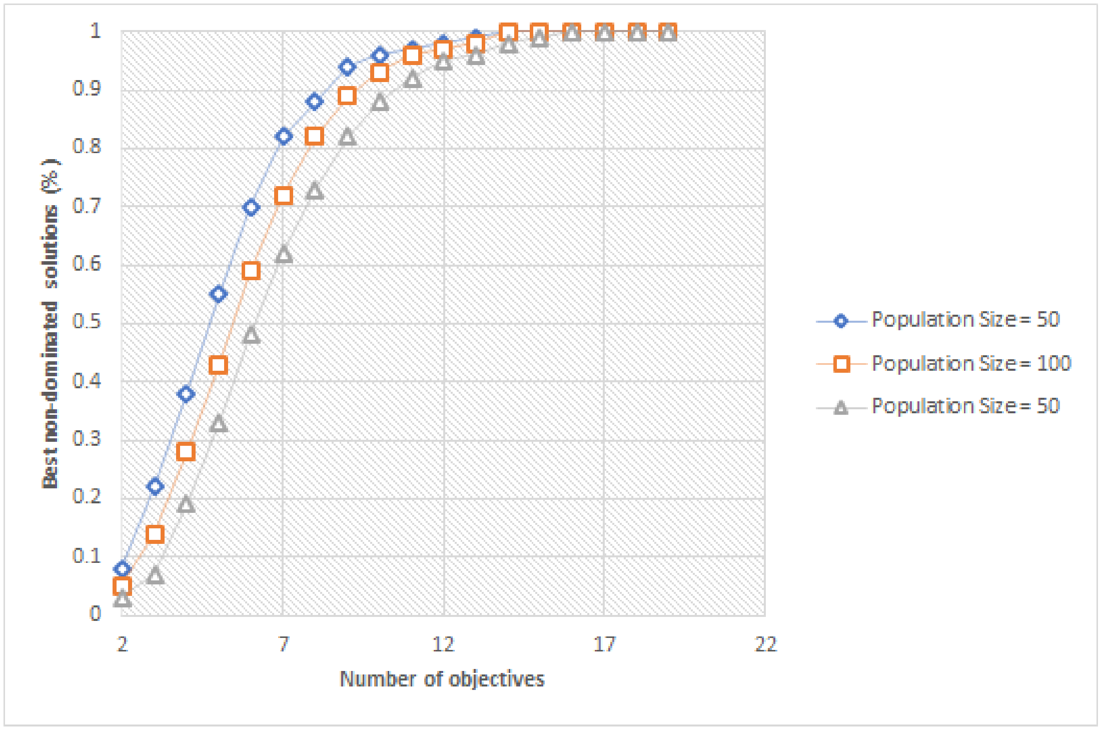


Figure 2.1: Proportion of non-dominated solutions vs. number of objectives, data derived from Deb et al., 2003 [1]

2.2.3 Multi/Many-objective Evolutionary Algorithms in SBSE

In SBSE literarture, multi/many-objective evolutionary algorithms (MOEAs) are the most widely used search-based optimization (SBO) algorithms [9, 2] . Sayyad et al., [2] survey a total of 52 papers on the used of Pareto-based optimization in SBSE, and showed that each surveyed paper uses at least one MOEA.

MOEAs take advantage of the use of various types of heuristic optimizers inspired by the natural process of evolution. Thus, the solutions space of a MOEA algorithms is referred to as *population*, which is itself a set of *individuals* (i.e., solutions to the problem). During the optimization or search process, individuals in the populations undergo a process of change and competition between themselves, resulting in a new population with a better quality. Common change operators encountered in the SBSE literature are the crossover and mutation operators. Figure 2.2 illustrates the crossover operation on two parent individuals as well as the mutation of a single individual.

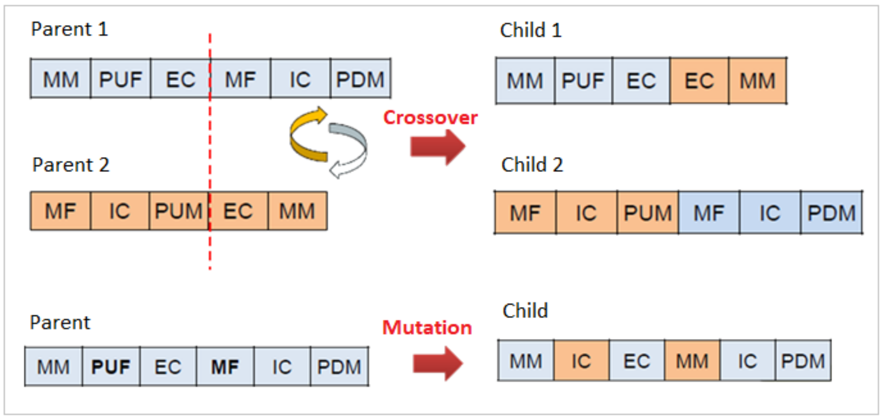


Figure 2.2: Example of crossover and mutation operators on sequence of strings

Among the MOEAs used in SBSE, the genetic algorithm and its variants occupy the first place in usage ranking. In [2], Sayyad et al., showed that the 4 most used multi-objective evolutionary algorithms are NSGA-II [27], MOGA [37], SPEA2 [38], and PAES [39]. This results is also confirmed in the survey by Harman et al., [9]. Table 2.1 shows the usage frequency of these methodologies methodologies. Table 2.1 also shows the very large utilization of the multi-objective genetic agloirthm (MOGA) and its variant NSGA-II in search-based software engineering with a total usage frequencies of 50.7% in the category of multi-algorithms papers and 64.2% in the category of single algorithm paper.

To our knowledge, there is no performance data that justifies the popularity of MOGA

Table 2.1: Usage Frequency of MOEAs in 51 SBSE research papers, Data derived from Sayyad et al., 2013 [2]. 15 paers used more than 1 algorithm and 36 papers used a single algorithm.

MOEAs	Multi-algorithms papers (%)	Single-Algorithm papers (%)
NSGA-II	45.4	53.0
SPEA2	12.0	8.3
MOCell	9.3	0.0
PAES	5.3	0.0
MOGA	5.3	11.2
Others	28.0	22.2

and NSGA-II in SBSE. However, this popularity maybe justified by their availability in state of the art tool such as Matlab®. In this thesis, we use the basic genetic Algorithm (GA), the NSGA-II [27], and the multi-objective simulated annealing (MOSA) [40, 41].

2.2.3.1 The NSGA-II Algorithm

The non-dominated sorting genetic algorithm also known as NSGA-II [27], is a widely used multi-objective evolutionary algorithm in practice [42]. Its performance for solving software engineering problem is well-established comparing to several other algorithms [9]. Algorithm 3 gives a high-level view of the NSGA-II algorithm.

NSGA-II starts with a randomly generated initial parent population P_0 of individuals. Then, the crossover and mutation genetic operators are applied to this initial population to create offspring individuals Q_0. Both parent and offsprings are merged into an initial population R_t ($t = 0$ at the first iteration). The resulting population R_t is used by the *fast-non-dominated-sort* of NSGA-II to classify individual solutions into different dominance level. To determine the dominance level of an individual solution x, this solution is compared to every other solution in R_0 until it is found dominated, or not. Based on the Pareto optimality, a solution x dominates another solution y, if x is no worse than y in all objectives and is strictly better than y in at least one objective. In mathematical notation, given a set of objectives functions $f_i, i \in 1..n$ to minimize, x dominates y can be written as follows:

$$\forall i, f_i(y) \leq f_i(x) \qquad \text{and} \qquad \exists j | f_j(y) < f_j(x) \tag{2.2}$$

Upon sorting using the above dominance principle, the individuals in R_0 are assigned to groups of different level of dominance referred to as Pareto fronts. Solutions in the first Pareto front F_0 are assigned dominance level 0, those in the second Pareto front F_1 are assigned dominance level 1, and so on. Part of the good solutions are used in subsequent iterations based on the dominance levels. The next parent population P_{t+1} is formed by adding individuals from successive fronts, starting with front F_0, until the size of P_{t+1} is equal to N. If filling P_{t+1} require to select a subset of individual in the last available front F_L, such selection is based on the crowding distance of each individual solution within the same front F_L [27]. The crowding distance of a non-dominated solution measures the density of solutions surrounding it, and is used to promote diversity within the population. It is estimated by the size of the largest cuboid enclosing a solution in the Pareto front that does not contain any other solution. The front F_L to undergo partial selection is sorted into descending order with respect to the crowding distance, and the first $N - |P_{t+1}|$ elements are chosen. Then, a new offspring population Q_{t+1} is generated from P_{t+1} using, again, the crossover and mutation operators. This process is repeated until a stopping criteria is met.

2.2.3.2 Multi-Objective Simulated Annealing: MOSA

Simulated annealing (SA) is a single-objective optimization technique inspired from the natural process of annealing solids, and was proposed by Kirkpatrick et al., [43]. The physical process of annealing consists in, first, raising the temperature of the metal, then cooling it down to a low-energy, crystalline state. At high temperature, the metal become soft due to movement of particle within it. This high energy state caused the structure or shape of the metal to be changed. After the new shape of the metal has been given it, the

ALGORITHM 1: NSGA-II: high-level view

```
Input: Population size $N$
$P_0$ =Create-initial-population($N$);
$Q_0$ =Generate-offsprings($P_0$);
$t = 0$;
repeat
    $R_t = P_t \cup Q_t$;
    F = fast-non-dominated-sort($R_t$);
    $P_{t+1} = \emptyset$;
    $i = 0$;
    repeat
        Apply crowding-distance-assignment($F_i$);
        $P_{t+1} = P_{t+1} \cup F_i$;
        $i = i + 1$;
    until ($|P_{t+1}| + |F_i| \leq N$);
    Crowding-Distance-Sort($F_i$);
    $P_{t+1} = P_{t+1} \cup F_i\,[N - |P_{t+1}|]$;
    $Q_{t+1}$ =Generate-offsprings($P_{t+1}$);
until (stopping criteria is reached);
```

temperature is lowered causing the particles to be restrain in movement. Thus, the metal crystallize in its new state (i.e., shape). Simulated annealing is inspired by this physical process as a computational model of the real-world system. The basic single-objective simulated annealing algorithm maintains a state and a computational temperature. At initialization, the algorithm start with a high temperature. This initial temperature is then reduced toward zero during the execution of the algorithm, or until a stopping criteria is reached.

At each iteration of the algorithm a solution is perturbed to produce a new solution. A solution is characterized by the energy of the state, as in the physical process. The quality of the initial solution and the perturbed solution are evaluated, using the objective function, and a new state is selected from the two solutions. When the new solution is no worse than the previous solution, the new solution is selected as the state. If the new solution has lower quality than the existing solution, it may be accepted with a probability dependent upon both the current computational temperature and the magnitude of the difference in quality. Algorithm 2 give a high-level view of the single-objective SA.

ALGORITHM 2: Single-objective simulated annealing

```
Input: maximum number of iteration $N$
$s_0 \leftarrow$ generateRandomSolution();
$t_0 \leftarrow$ getRandomTemperature();
$CS \leftarrow$ selectCoolingSchedule();
repeat
    $i \leftarrow 0$;
    repeat
        $s \leftarrow$ generateRandomSolutionFromNeighborhoodStructure();
        $\delta \leftarrow f(s) - f(s_0) \quad //objectivefunction$;
        if $\delta < 0$ then
            $s_0 \leftarrow s$;
        else
            $x \leftarrow$ getRandomValueFromUniformDistribution($0, 1$);
            if $x < \exp(\frac{-\delta}{t})$ then
                $s_0 \leftarrow s$
            end
        end
    until $i = N$;
    $t_0 \leftarrow CS(t)$;
until (stopping criteria is reached);
```

The multi-objective simulated annealing (MOSA) [40] is based on the principle of the single-objective SA. A solution is a vector s, and the objective function is an $n-$dimensional vector $\mathbf{f}(\mathbf{s})$. The details of this algorithm can be found in [40]. We use the MOSA algorithm in chapter 3.

2.2.4 Challenges In Search-based software Engineering

Though there exists diverse classes of SBO algorithms, various survey have demonstrated that Multi-Objective Evolutionary algorithms (MOEAs) are the most used in SBO algorithms in SBSE [9, 2]. Sayyad and Ammar [2] showed in their study that 5 MOEAs are widely used in SBSE, namely NSGA-II, SPEA2, MOCell, PAES, and MOGA. Among these algorithms, NSGA-II represents almost 44%. It is well-known that NSGA-II performs well for number of objectives up to 3. It is also reported in the same survey that many-objective formulations of SE problems represent only 23%. Among 79 formulations of SE problems as MOEAs, only 14 considered number of objectives higher than 3. This limitation of the majority of SBSE techniques to fewer objectives is a well-known issue inherited from MOEAs. It is referred to as the curse of dimensionality, and consists of 3 majors shortcomings:

- *Deterioration of the performance of the MOEA algorithm* in searching towards the Pareto-optimal front. This problem becomes severe as an increase in number of objectives causes a large proportion of the population to become equivalent.
- *High computational cost* for MOEAs with large number of objectives. Realistically, the number of solutions required to accurately compute the Pareto optimal front grows exponentially with the number of objectives. For example, the number of solutions necessary for a good approximation of the Pareto front for a 4-objective problem is about $62,500$ [44]. The computation cost can be distributed over diversity measure estimation, recombination operations, etc.
- *Difficulty of the visualization of the Pareto-optimal front*. This impairs one of the major steps of MOEAs - decision making. Note that MOEAs do not propose a single solution, but a set of good solutions. Therefore, the decision making process is a vital step, where the expert must select one solution from a set of competitive ones. The visualization of the Pareto front becomes a challenge when the number of non-dominated solutions increase dramatically.

Therefore, Researchers have proposed several approaches to alleviate the limitations of MOEAs in number of objective functions:

- *Incorporation of decision maker's preference* is a method whereby the search-based algorithm is guided by the decision maker towards the region of the Pareto front that is of interest (ROI) to him or her [45, 46, 47],. This keeps the MOEA from performing unnecessary computation outside of the decision maker's ROI.

- Another techniques for handling large dimensional Pareto front is the *new preference ordering relations* method [48, 49, 50]. The idea in preference relations is to improve the deterioration of the Pareto dominance by adding additional information such as solution ranking with respect to the different objectives. However, these ranks may not be the best for the decision maker.
- In addition to these two methods, other methods such as *Decomposition* [51, 52], and *the use of predefined multiple targeted search* [53, 54, 55], have been proposed to overcome the adverse effects of high dimensionality in the Pareto front. Decomposition breaks the high-dimensional many-objective problem into sub-problems, and solves these sub-problems using parallelism in the search algorithm. The use of predefined targets - reference points, reference directions - in the objective space allows to guide the search during the optimization process. These techniques have considerably improved the search abilities of MOEAs. However, issues such as high computational cost and the difficulty of visualization still exists.

In chapter 6, we address the issue of objective selection and decision making in high-dimensional search-based software engineering with the particular case of software refactoring.

2.3 Software Refactoring

2.3.1 Introduction

Real-world software systems undergo multiple stages during their lifecyle. First, software are developed and a first version is released to the potential users. Then, ultimately, new requirements are added to the systems that need to be implemented, and/or existing functionalities go through a process of improvement. this process is referred to as software evolution, and it is an activity on which most software organizations allocate a large portion of the total software cost. In addition, this process of evolution causes the software to increase in size and complexity, and often, leads to deviation of the system from its original coding best practices. When these deviations arise, software maintenance activities such as restructuring are triggered by developers to bring the quality of the system to its desired standard value.

In object-oriented design, the process of restructuring software is referred to as software refactoring [56]. It is defined as the process of improving the software code after it has been written by changing its internal structure without changing its external behavior.Thus, refactoring consists in reorganizing variables, classes and methods. The goal of this reorganization is to improve various aspects of software quality such extensibility, reusability, understandability, etc. These features are elements that improve the maintainability of software systems.

The refactoring process can either be manually performed by a software engineer, a combination of manual and automated operations, or fully-automated. These different approaches constitute the three types of existing software refactoring methodologies.

2.3.2 Manual Software Refactoring

In Fowler's book [15] a non-exhaustive list of design problems in source code has been defined. For each type of code smell, a list of possible refactorings (template) is suggested that can be applied by the developers. In another study, Du Bois et al., [57] focus on the detection of refactoring opportunities that may improve cohesion and coupling metrics, and they used them 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., [24] proposed several techniques and empirical studies to support and understand refactoring activities. In [58, 59] , 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 et al., [60] have proposed a new refactoring tool called GhostFactor that allows the developer to transform code manually, and checks the correctness of the transformation automatically. BeneFactor [61] and WitchDoctor [62] can detect manual refactorings and then complete them automatically. Tahvildari et al., [63] 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 [64] proposes a refactoring technique to improve the parallelism of software systems.

Other contributions are based on rules that can be expressed as assertions (invariants, pre- and post-conditions). The use of invariants has been proposed to detect parts of the program that require refactoring [65]. In addition, Opdyke [56] 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/satisfaction 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.

2.3.3 Fully-automated Refactoring

To automate refactoring activities, new approaches have been proposed. JDeodorant [66] 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 [31] 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., [68] 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 [3] to evaluate the improvement in quality.

Kessentini et al., [16] 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., [69] 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., [70, 71] proposed also a multi-objective refactoring formulation that generates solutions to fix code smells. Ó Cinnéide et al., [72] 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. Mkaouer et al., [73, 36, 74] recently proposed to formulated refactoring as a many-objective problem where a high number of objectives are considered during the search process. However, the number of non-dominated refactoring solutions is high and it is challenging to the developers to select the solution based on their preferences. In addition, some of the quality metrics could be correlated and this may reduce the number of objectives for faster convergence.

2.3.4 Semi-automated 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., [75] 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. 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.

Bavota et al., [76] 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 solutions 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.

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.

CHAPTER 3

Context-Based Refactoring Recommendation Approach: Two Industrial Case Studies

3.1 Introduction

Refactoring is a highly valuable solution to reduce and manage the continuously increasing complexity of software systems. In real-world scenario, programmers are "opportunistic" when undertaking a refactoring activity - That is most programmers are interested in improving the quality of the code fragments that they frequently update and/or those related to the planned activities for the next release (fixing bugs, adding new functionalities, etc.). However, a large portion of existing refactoring tools suggests refactorings to improve the overall quality of systems without a concrete prioritization plan [24][25] . This system-wide treatment of refactoring recommendation rise two major issues: (1) A large number of refactoring operations need to be applied, and (2) the scope of suggested refactorings can be outside of the software component under the control of the programmer.

When a large number of refactorings are recommended, manual refactoring becomes error-prone and time-consuming. Murphy-Hill et al., [24] show that most developers do not use fully automated refactoring techniques because they want to mix refactorings with semantic changes, something that is not permitted by existing methods. In addition, when automated refactoring operations affect multiple components of the software, some of the components may be unknown to the specific software engineers who are performing the maintenance activity. The unfamiliarity of these engineers with those components reduces their confidence in applying these system-wide refactorings. As results, most developers are reluctant to use fully-automated refactoring recommendation tools due to fear of introducing bugs or undesired changes.

In the current literature, Search-based refactoring techniques obtained promising results based on the use of mono-objective and multi-objective algorithms to optimize quality metrics [16][68][77][67][10][28][23]. However, most of these techniques explore a large search space of possible solutions and recommend large system-wide sequence of refactorings operations. Thus, these search-based refactoring techniques fail to meet the need of developers, who are more interested in refactoring recently modified entities related to their current tasks (e.g. features update, fixing bugs, etc.) [24]. In addition, Ouni et al., [70, 28] used the history of code changes to deduce refactoring operations from past refactoring activities applied to fragments of codes under maintenance. Their findings shows that the history of code change is an important source of information that can be used to improve the quality of software refactoring on a given system. Furthermore, it is well-established by recent empirical studies that code fragments that trigger refactoring activities are often classes that are directly correlated with discovered bugs.

In this chapter, we propose a profile-based approach for refactoring recommendations to satisfy the following requirements: 1) programmers prefer to improve, mainly, the quality of recently modified code before a new release due to limited resources and time, 2) several empirical studies [66][78][79][80] identified correlation between bugs and refactoring opportunities, and 3) recently introduced refactorings may give an indication of quality issues that should be fixed and show an interest from programmers to refactor these code fragments.

In this chapter, to overcome some of the above-mentioned limitations of existing refactoring methodologies, we propose a profile-based search-based refactoring approach that account for the context of the software programmer. The approach uses the multi-objective simulated annealing (MOSA) [40] to find the best refactoring solution that satisfies two objectives: maximizing the number refactorings applied to buggy or recently modified classes, and minimizing the number of antipatterns [14] using a set of antipatterns detection rules [16]. We implemented our proposed approach and evaluated it on a set of two industrial systems provided by our industrial partner from the automotive industry. We did the evaluation only on these two systems since it is critical to evaluate the relevance of recommended refactorings by the original developers of the systems. Statistical analysis of our experiments showed that our proposal performed significantly better than existing search-based refactoring approaches [68][77] and an existing refactoring tool not based on heuristic search, JDeodorant [66] regarding the relevance and importance of recommended refactorings. In our qualitative analysis, we conducted a survey with the software developers who participated in our experiments to evaluate the relevance of the fixed quality violations in their daily development activities.

The primary contributions of our profile-based approach can be summarized as follows:

1. Meet programmers' requirement to improve the quality of recently modified code before a new release. This includes the optimization of the refactoring cost in time.

2. Recommend refactorings solutions correlated with bug reports,

3. Take into account refactoring operations that were recently applied to the system.

3.2 Approach

The main goal of this approach is to integrate the preference of the developer into the software refactoring. There are two main methods that can be used to integrate the preference of an expert into a process: (1) the active or interactive methods, and (2) The passive or non-interactive methods based on data mining. In the active method, the software engineers, who have the expertise are integrated in the search-loop. Then, using their expertise of the domain, they can accept the solutions that meet their preferences. In addition, the engineers can interact with the search process to bias solution generation towards their preferences. In the second method, the passive techniques, using a repository of previous activities, an automated algorithm can extract the user preferences with respect to software refactoring. Then, using such behavior, new refactorings can be deduced from previous ones.

Our approach uses the passive non-interactive method of user preference integration. It finds the most relevant refactorings for software developers to refactor their systems based on their recent updates of the system. The general structure of our approach is sketched in Figure 3.1.

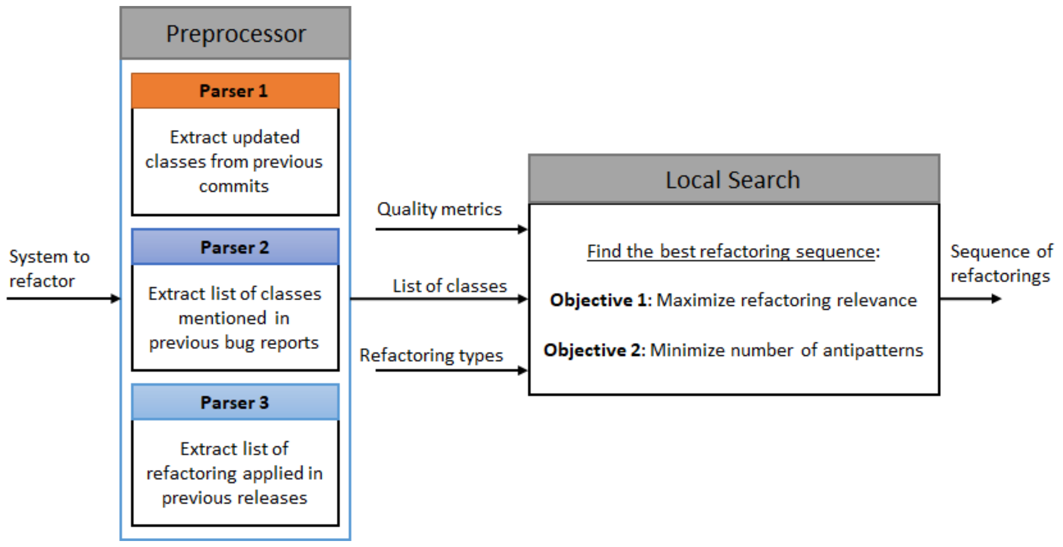

Preprocessor
Parser 1
Extract updated classes from previous commits
Parser 2
Extract list of classes mentioned in previous bug reports
Parser 3
Extract list of refactoring applied in previous releases
System to refactor
Quality metrics
List of classes
Refactoring types
Local Search
Find the best refactoring sequence:
Objective 1: Maximize refactoring relevance
Objective 2: Minimize number of antipatterns
Sequence of refactorings

Figure 3.1: Context-based software refactoring approach overview

Our technique comprises two main components. The first component is the preprocessing component, which rank the list of possible classes to refactor. During the preprocessing phase, three different parsers are executed. The first parser extracts, from the system to refactor, classes that have been recently modified or refactored. Classes mentioned in recent commits reflect the current coding context of the software engineer, and also indicates that they may be important to refactor since they have a high probability of including bugs.

Developers, for example, may introduce bugs because of the complexity of the system and its poor design. In addition, these classes may be candidate for future updated in comparison with stable classes that were not modified for many releases. The second parser extracts a list of classes that were sources of previously reported bugs. Several empirical studies show that correlation exists between buggy classes and poor quality symptoms. Thus, relevant refactorings for the developers context could be identified in these classes based on this pre-processing phase. Finally, the last parser identifies refactoring operations that were applied to previous release of the system. Classes that are refactored recently by the developers but still contain quality issues can be recommended for further refactoring since programmers have already expressed an interest in fixing them. The list of applied refactorings in previous releases are detected using the technique proposed in [23]. The outcome of this first phase is a list of classes that could be refactored based on the three main criteria detailed above.

The outcome of the first phase is used to reduce the search space to find the best refactoring sequence to recommend for developers. A multi-objective search algorithm is used to focus mainly on refactoring, if needed, the classes of the first phase while fixing some other quality issues as well. To this end, a multi-objective simulated annealing algorithm is executed for a number of iterations to find the solutions balancing the two objectives of 1) improving the relevance of recommended refactorings, which corresponds to maximize the number of refactoring recommendation in recently modified or buggy classes and 2) minimizing the number of antipatterns using a set of detection rules defined in the literature [16]. The first objective of the refactorings relevance is based on an average of three different measures of recently modified classes, recent classes mentioned in bug reports and recently refactored classes including incomplete refactoring activities or may need to be further refactored. The formalization of these measures will be described in the next section.

A multi-objective simulated annealing algorithm [40] is selected due to the small search space to explore after the pre-processing phase. A set of semantic constraints is used to check the correctness and feasibility of recommended refactorings based on textual similarities, call graphs and pre/post-conditions. These constraints are described in more details in [77]. The next section will discuss the formalization of our approach and the adaptation of the multi-objective simulated annealing algorithm to our problem.

3.3 Problem Formulation and Solution Approach

Simulated annealing is a local search heuristic inspired by the concept of annealing in metallurgy where metal is heated, raising its energy and relieving it of defects due to its ability to move around more easily [40]. As its temperature drops, the metal's energy drops and eventually it settles in a more stable state and becomes rigid. The local search algorithm of the Simulated Annealing is very suitable for exploring small search spaces. More details about Multi-Objective Simulated Annealing can be found in chapter 2 and related references.

3.3.1 Solution Representation

A solution of our problem is defined as a sequence of refactoring operations involving one or multiple source code fragments of the software to refactor. As described in Table 3.1, the vector-based representation is used to define refactoring sequences. Each dimension of the vector is a refactoring operation, and its index in the vector indicates the order in which it is applied. For every refactoring, pre- and post-conditions are specified to guarantee the correctness of the operation.

The initial population is created by randomly selecting a sequence of operations to a randomly chosen set of code elements, or actors identified in the first phase of search space reduction. 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. In our experiments, we used the

Table 3.1: Example of refactoring solution: A sequence consisting of first randomly generated operations

REF	REFACTORING OPERATIONS
RO_1	MoveMethod(org.apache.xerces.xinclude.XIncludeTextReader, org.apache.xerces.xinclude.XIncludeTextReader, close())
RO_2	MergePackage(org.apache.xerces.xpointer, org.apache.xerces.xs)
RO_3	PullUpMethod(org.apache.html.dom.HTMLTableCaptionElementImpl, org.apache.html.dom.HTMLElementImpl, addEventListener())
RO_4	ExtractInterface(org.apache.xml.serialize.SerializerFactory, apache.xml.serialize.SerializerFactoryInterface)

following list of refactorings: *Extract class (EC), Extract interface (EI), Inline class (IC), Move field (MF), Move method (MM), Push down field (PDF), Push down method (PDM), Pull up field (PUF), Pull up method (PUM), Move class (MC), and Extract method (EM).*

3.3.2 Fitness Functions

The generated solutions are evaluated using two fitness functions. The first fitness function is the number of antipatterns or code smells. This fitness function is to be minimized, and is calculated using equation 3.1.

$$\min f_1(s) = \frac{\text{\# code smells after refactoring}}{\text{\# code smells before refactoring}} \tag{3.1}$$

This function represents the proportion between the number of corrected defects (detected using bad smells detection rules) and the total number of possible defects that can be detected. The detection of defects is based on some metrics-based rules according to which a code fragment can be classified as a design defect or not (without a probability/risk score), i.e., 0 or 1, as defined in the detection rules in [16].

The second fitness function is the refactoring relevance. This fitness is maximized, and its goal to evaluate the refactoring solutions based on their relevance to the developers. Formally, this function is defined as follows:

$$\max f_2(s) = \sum_{i=1}^{n} \frac{\frac{commitf(c_i)+bugreportsf(c_i)}{2}}{n} \tag{3.2}$$

where n is the number of classes to be refactored by the solution S, c is the class that contain at least one code smell and $commitf(c)$ and $bugreportsf(c)$ are respectively the functions to estimate the relevance of the class for refactoring based on previous changes in recent commits and previous bug reports.

The first function $commitf(c)$ checks if a class was recently changed. In fact, a class that was modified recently has a high probability to be refactored comparing to stable classes. Thus, the function compares between the date of the last commit and the last date where the class was modified in the previous commit. If a suggested class was modified in the last commit, then the value of this function is 1. We define this normalized function, normalized in the range of [0, 1] as following:

$$commitf(c) = \frac{1}{commit.date(c) - lastcommit.date + 1} \tag{3.3}$$

The second function $bugreportsf(c)$ counts the number of times a class was fixed to eliminate bugs based on the history of bug reports divided by the maximum number of times that a class in the system was fixed in previous bug reports. In fact, a class that was fixed several times has a high probability of being a buggy class and thus need to be refactored. Formally, this function, normalized between [0,1] is defined as:

$$bugreportsf(c) = \frac{\frac{1}{lastbugreport.date(c)-lastbugreport.date+1} + \frac{NbFixedBugs(reports,c)}{MaxNbFixedBugs(reports)}}{2} \tag{3.4}$$

3.3.3 Change Operators

MOSA is using a mutation operator to generate new solutions. For mutation, we use the bit-string mutation operator that selects one or more refactoring operations (or their controlling parameters) from the solution and replaces them by other ones from the list of possible operations to apply. For an illustration of the mutation operator, refer to figure 2.2.

When applying the change operators, the different pre- and post-conditions are checked to ensure the applicability of the newly generated solutions. We also apply a repair operator to randomly select new refactorings to replace those creating conflicts.

3.4 Evaluation

3.4.1 Research Questions and Evaluation Metrics

To evaluate and compare the performance and relevance of the recommended refactoring by our context-based multi-objective simulated annealing algorithm, we defined the following three research questions:

RQ1: To what extent can our approach recommends relevant refactorings to developers?

RQ2: To what extent can our approach reduces the number of refactorings and the execution time while improving the quality and recommending relevant refactorings compared to existing refactoring techniques?

RQ3: Can our approach be relevant for programmers in practice?

To answer the first research question **RQ1**, we used both qualitative and quantitative evaluations of refactoring solutions recommended by our approach and existing studies. For the quantitative validation, we asked a group of developers from our industrial partner to manually suggest a list of possible refactorings to apply based on the latest release source code of the system to refactor. Then, we used the precision (PR) and recall (RC) measures to evaluate the similarity between the recommended refactorings by our approach and those manually found by the original programmers of the industrial projects. Precision and recall are calculated as follows:

$$\text{RC} = \frac{|\text{set(recommended refactorings)} \cap \text{set(expected refactorings)}|}{|\text{set(expected refactorings)}|} \tag{3.5}$$

$$\text{PR} = \frac{|\text{set(recommended refactorings)} \cap \text{set(expected refactorings)}|}{|\text{set(recommended refactorings)}|} \tag{3.6}$$

Another metric that we considered for the quantitative evaluation is the percentage of fixed antipatterns (NF) by the refactoring solution. The code smells are detected on the new source code after refactoring based on the detection rules provided by [81]. Formally, NF is defined as defined as follows:

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

The detection of antipatterns 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 another metrics the total gain in quality G for each of the considered QMOOD [3] quality attributes q_i before and after refactoring can be easily estimated as:

$$G_{q_i} = q'_i - q_i \tag{3.8}$$

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

Since several good solutions can be relevant, it is important to check the relevance and correctness of recommended refactorings not only by comparing them with one expected solution (quantitative validation). Thus, we performed a qualitative evaluation where we asked the original programmers of the industrial projects to review, manually, if the recommended refactorings are relevant and correct or not from their perspectives. We define the metric Refactoring Relevance (RR) to mean the number of relevant refactorings divided by the total number of suggested refactorings. RR is given by the following equation:

$$\text{RR} = \frac{\text{\# relevant refactorings}}{\text{\# proposed refactorings}} \tag{3.9}$$

To answer **RQ2**, we compared our approach to random search (RS), mono-objective simulated annealing (SA) aggregating both objectives, another multi-objective evolutionary algorithm (NSGA-II) and an existing work based on search algorithms to fully-automate the refactoring recommendation process: OKeeffe and Cinnide [29] and Ouni et al., [82].

OKeeffe and Cinnide proposed a mono-objective formulation to automate the refactoring process by optimizing a set of quality metrics. Ouni et al., [77] proposed a multi-objective refactoring formulation that generates solutions to fix code smells. Both techniques are fully-automated and did not consider the personalization of refactoring recommendations. We have also compared our results with an existing tool, called JDeodorant, not based on heuristic search to fix quality issues by recommending refactorings. JDeodorant implements a set of templates to fix different design violations by providing a generic list of refactorings to apply. 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.

We used the metrics PR, RC, NF, RC and G to perform the comparisons and two new metrics related to the computational time (CT) and the number of refactorings (NR).

To answer **RQ3**, we asked the programmers to answer to a post-study questionnaire to get their opinions and feedback about our personalized refactoring recommendations.

3.4.2 Experimental Setup

To get feedback from the original developers of a system, we considered in our experiments two large industrial projects provided by our industrial partner, from the automotive industry. The first project is a marketing return on investment tool, called MROI, used by the marketing department 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 Company to create the best schedule of orders from the dealers based on many business constraints. This system is also used by the company to find the best configurations of cars based on the requirements of dealers and customers. Software developers have developed several releases of this system at the company over the past 10 years. Due to the high number of changes introduced to this system over the years and its importance, it is critical to ensure that they remain of high quality and minimize the effort required by developers to fix bugs and extend the system in the future. Table 3.2 described the statistics related to the two studied systems.

Our study involved 19 software developers from the company. Participants include 9 original developers of the MROI system and 10 original developers of the JDI one. All the developers who participated in the experiments are expert in Java, quality assurance and testing. The experience of these participants on these areas ranged from 7 to 18 years.

The questionnaire includes five main questions to be answered by the participants. Some of the questions are related to the background of the participants to evaluate their experience and ability to evaluate the results of our technique. Furthermore, we organized a lecture for all the participants about different concepts and examples related to software refactoring then they took six tests about evaluating the relevance of recommended refactorings on code fragments extracted from open source systems.

We formed two groups. Each of the two groups (A and B) is composed of the original developers of each system. We selected the participants of each group based on the collected background information to make sure that both groups have, in average, the same level of expertise with software refactoring and quality assurance. We provided to all the participants the questionnaire, the guidelines about the different steps to perform the experiments, the different used tools and source code of the systems to evaluate. After the first step of the quantitative evaluation, we provided to the participants the list of recommended refactorings by the different tools and asked them to evaluate their relevance and correctness. The participants are not aware of the tools used to get the different results. We counted the votes of the programmers for every of the recommended refactorings then we considered the highest number of votes to evaluate the correctness/relevance of the evaluated operations.

In the first scenario, we asked every participant to manually apply refactorings after reviewing the code of their systems. As an outcome of the first scenario, we estimated the similarity between the suggested refactorings and the expected ones as defined by the programmers.

In the second scenario, we asked the developers to manually evaluate the relevance of every recommended refactoring by our approach. In the third scenario, we collected the opinions of the developers about our tool based on a post-study questionnaire that will be detailed later. The programmers commented on the different evaluated refactorings and these comments/justifications were discussed later with the organizers of the study.

We used different population sizes of the used algorithms to evaluate their performance ranging from 100, 200, 300 and 500 individuals per population.

The maximum number of iterations is $100,000$ evaluations for all the studied systems. We used the Wilcoxon test to compare between the different algorithms considered in our experiments. For each algorithm and project, we use the trial and error strategy to find the good parameters setting. For all the systems and algorithms, the obtained results in our experiments are statistically significant on 30 independent executions using the Wilcoxon rank sum test with a confidence level of 95% ($\alpha < 5\%$).

Table 3.2: Statistical data of the two evaluated industrial projects

Systems	Releases	Avg. # classes	Avg. KLOC	Avg. # code smells	# manual refactorings
JDI	V1.0 - V5.8 (26 releases)	694	252	88	94
MROI	V1.0 - V6.4 (31 releases)	827	269	116	119

3.4.3 Results and Discussions

Results for RQ1. Figure 3.2 summarized the results of our approach of the qualitative evaluation when programmers manually evaluated the relevance and correctness of the recommended refactorings. Most of the solutions recommended by our personalized approach are relevant and correct from the perceptive of the programmers.

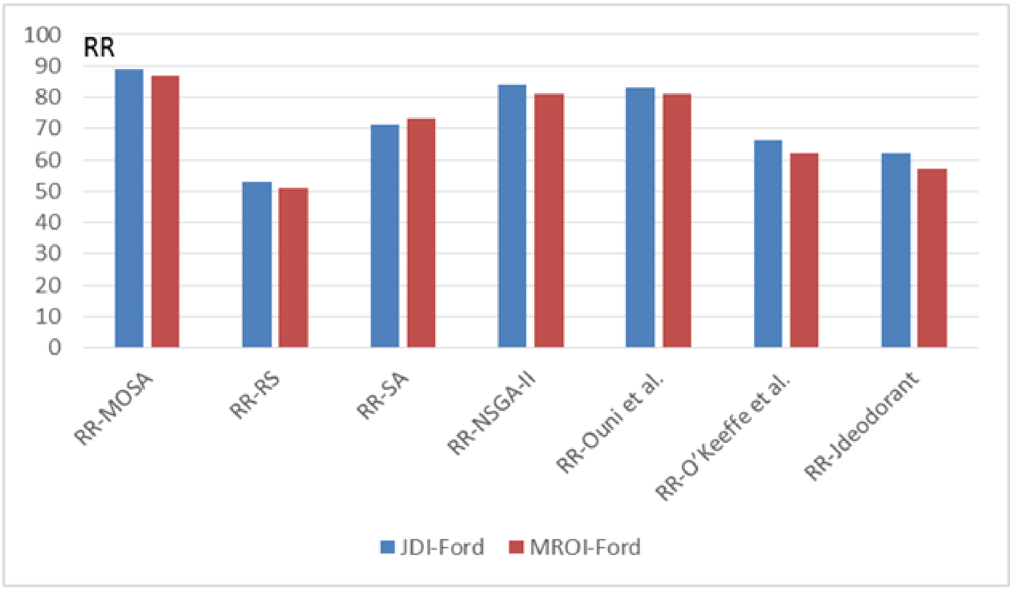


Figure 3.2: Median refactoring relevance (RR) value for 30 executions on the two systems with a 95% confidence level ($\alpha < 5\%$)

On average, for the two studied projects, around 88% of the proposed refactoring operations are found to be useful by the software developers of our experiments. The highest MC score is 89% for the JDI project and the RR score is 87% for the second system MROI. Thus, it is clear the obtained results are not dependent on 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 fixing non-relevant quality issues or introducing design incoherence.

We also compared the proposed refactoring solutions with the ones that are provided manually by the programmers of these industrial systems. Figures 3-4 show that the majority of the proposed refactorings, with an average of 84% in terms of precision and 87% of recall, are equivalent to those manually found by the programmers when trying to refactor the system. The higher score of the recall comparing to the precision can be explained by the fact that our approach proposes a complete list of refactorings comparing to the manually recommended operations by the programmers due to the time-consuming process of code refactoring. Also, 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 quality issues or those related more to find better ways to extend the current design.

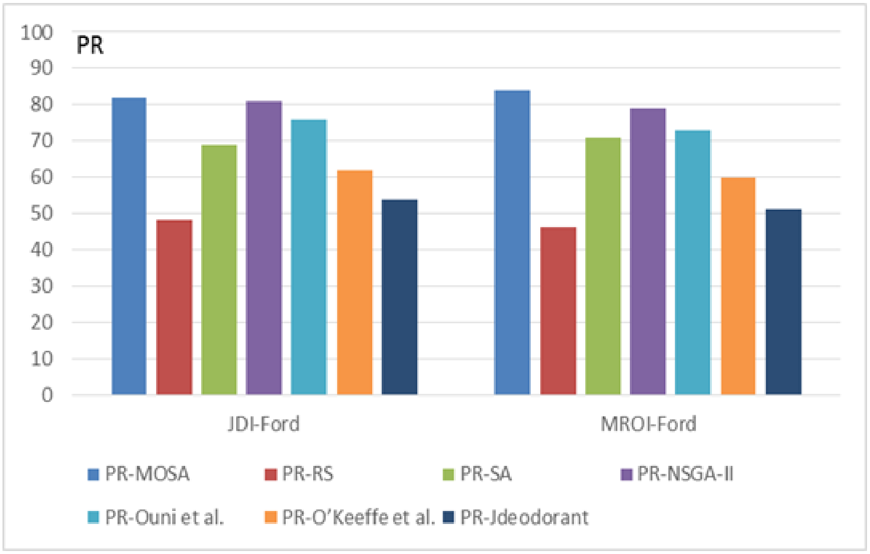


Figure 3.3: Median precision (PR) value for 30 executions on all the two systems with a 95% confidence level ($\alpha < 5\%$)

Figure 3.5 shows that the refactorings recommended by the approach and applied by developers improved the quality metrics value (G) of the two systems. The average quality gain for the two industrial systems was the highest among the systems with more than 0.2. The improvements in the quality gain confirm that the recommended refactorings helped to optimize different quality metrics by fixing the most severe quality issues. Although the average quality gain is lower comparing to one of the existing techniques, it dominates 5 existing techniques. In addition, it is comparable to the best existing method, and is lower due to the much lower number of refactorings recommended by our technique.

Result for RQ2. Figures $3.3-3.8$ confirm the average superior performance of our personalized refactoring approach compared to existing refactoring approaches. Figure 3.3 describes that our approach provides better precision results (PR) than existing approaches having PR scores as high as 79% on average, on the two different systems. The same result is observed for the recall (RC) and quality gain (G) as described in Figure 3.4 and 3.5. However, the quality gain is slightly lower than some of the existing techniques as showed in Figure 3.5. This can be explained by the reason that the main goal of developers is not to fix the maximum number the quality issues detected in the system (which was the goal of most of the existing studies). Also, our approach is based on a multi-objective algorithm to find a trade-off between improving the quality and reducing the number of refactorings. Figure 3.6 clearly shows that our personalized refactoring approach converges much faster to acceptable refactoring solutions comparing to most of the existing studies. For example, the work of Ouni et al., required at least 20 minutes to converge to a good quality of solutions however our approach was able to recommend good refactoring opportunities within 2 minutes. One reason of the low execution time of our approach is the number of recommended refactorings as described in Figure 3.7. To conclude, our interactive approach provides better results, on average, than existing fully-automated refactoring techniques (answer to RQ2).

Results for RQ3. In the first component of the post-study questionnaire, the participants were asked to rate their agreement on a Likert scale from 1 (complete disagreement)

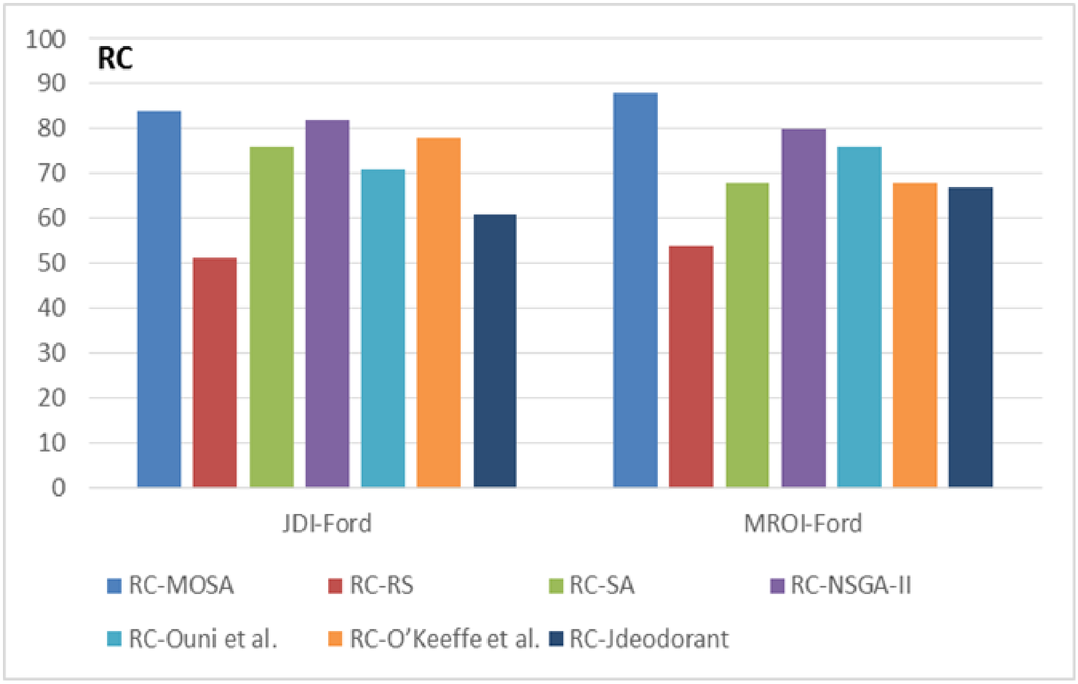


Figure 3.4: Median recall (RC) value for 30 executions on all the two systems with a 95% confidence level ($\alpha < 5\%$)

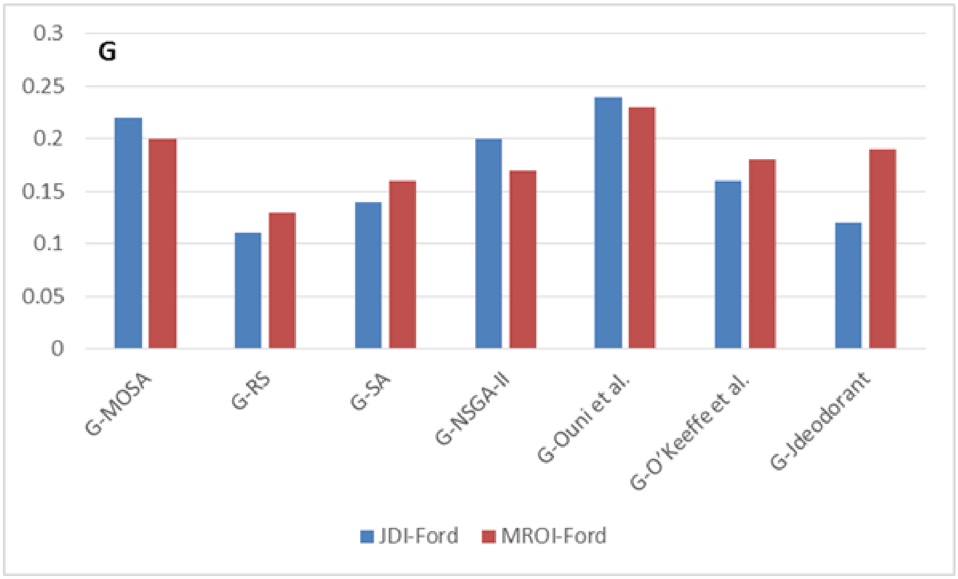


Figure 3.5: Median quality gain (G) value for 30 executions on all the two systems with a 95% confidence level ($\alpha < 5\%$)

to 5 (complete agreement) with the following statements: 1. The proposed personalized refactoring technique is a desirable feature in integrated development environments. 2.

The reduced number of recommended relevant refactorings may help developers performing every-day design, implementation and maintenance activities.

In the second component of the questionnaire, the subjects were asked to specify the possible usefulness of the suggested refactorings to perform some activities such as quality assurance/assessment, regression testing, effort prediction, code inspection, and features extension. In the third part, we asked the programmers about possible improvements of our personalized refactoring tool.

As described in Figure 7, the agreement of the participants was 4.6 and 4.3 for the first and second statements respectively. This confirms the usefulness of our approach for the software developers. Regarding the possible usefulness to perform some activities, the developers agreed that quality assurance/assessment and features extension are the three main activities where the personalized refactorings could be very helpful with an agreement of more than 4.3.

The three other activities of effort prediction, regression testing and code inspection are considered less relevant for our tool with an agreement of around 3.8. The majority of the programmers we interviewed found that the personalized refactorings give interesting quick advice about possible refactoring opportunities to improve the quality and mainly facilitate extending the design of the system to update recently introduced features.

The remaining questions of the post-study questionnaire were about the benefits and also limitations (possible improvements) of our approach. They found that the personalized refactoring technique is much more efficient than the traditional manual and fully-automated techniques. The programmers considered the use of most of existing manual refactoring techniques as a time-consuming process, and it is more relevant to apply refactorings related to their recent development activities. Most of the participants mention that our personalized approach to refactor the code is much faster than analyzing the long list of recommended refactorings by current techniques. The programmers also highlighted that our personalized approach recommended relevant refactorings to continue improving the quality of some code fragments that they started refactoring them in the past.

The participants also suggested some possible improvements to our personalized refactoring approach. Several participants found that it will be very interesting and helpful to integrate to the tool a new functionality to visualize the design before and after refactoring. The developers also proposed to explore the area of impact changes analysis as a complementary step of our technique after applying the recommended refactorings.

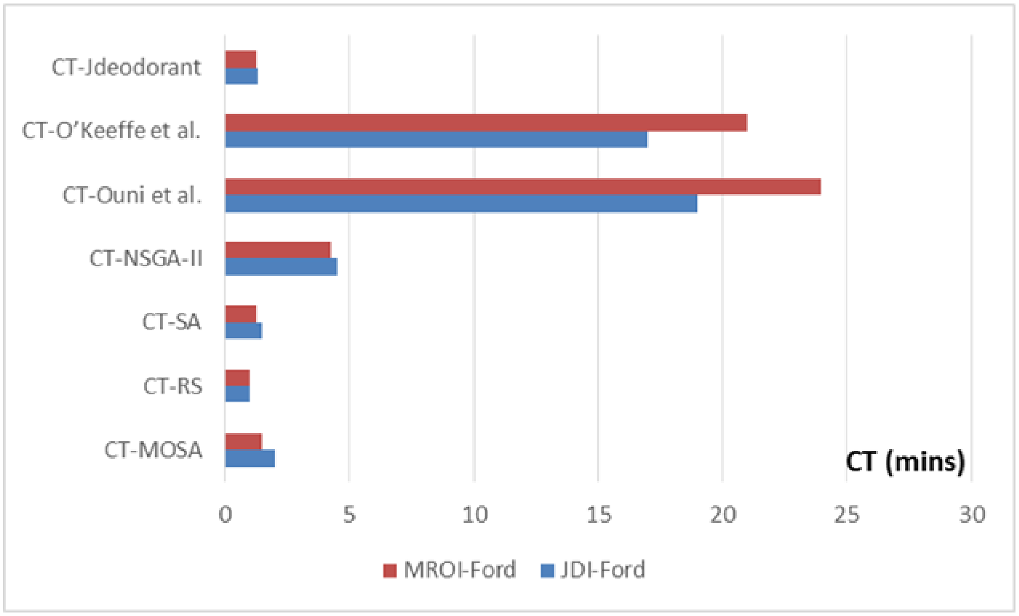


Figure 3.6: Median execution time (CT) value for 30 executions on all the two systems with a 95% confidence level ($\alpha < 5\%$)

3.5 Conclusion

In this chapter, we proposed a personalized search-based refactoring technique that integrate programmer-specific coding preferences based on the history of changes of the system. In dynamic coding environment this refactoring approach allows for recommending refactorings that are relevant to the coding context of the programmer in charge of the maintenance activity. This means, recommended refactoring will affect only the part of the system related to the history of activity provided as input, an thus, avoiding system-wide

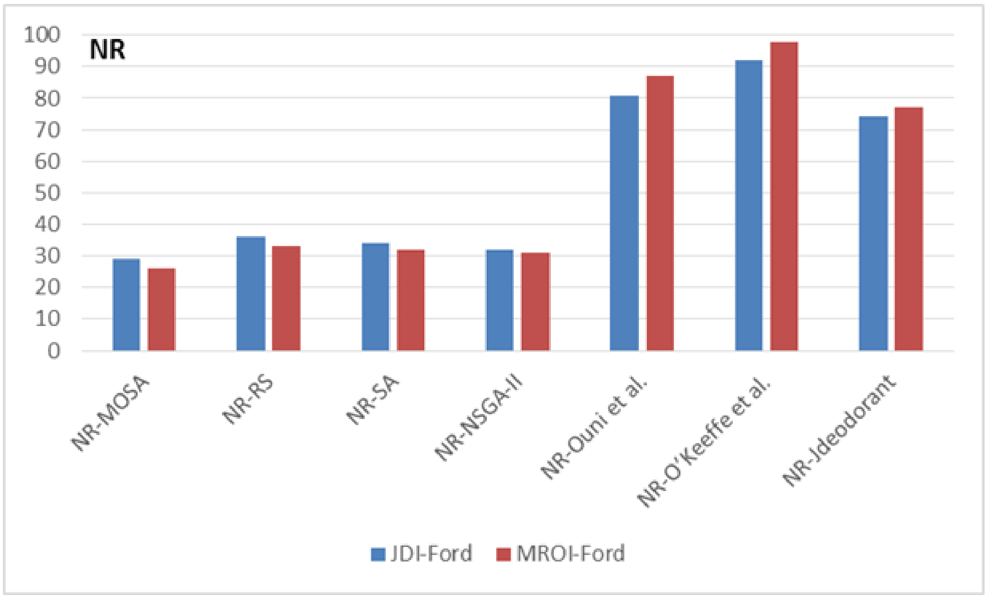


Figure 3.7: Median number of refactorings (NR) for 30 executions on all the two systems with a 95% confidence level ($\alpha < 5\%$)

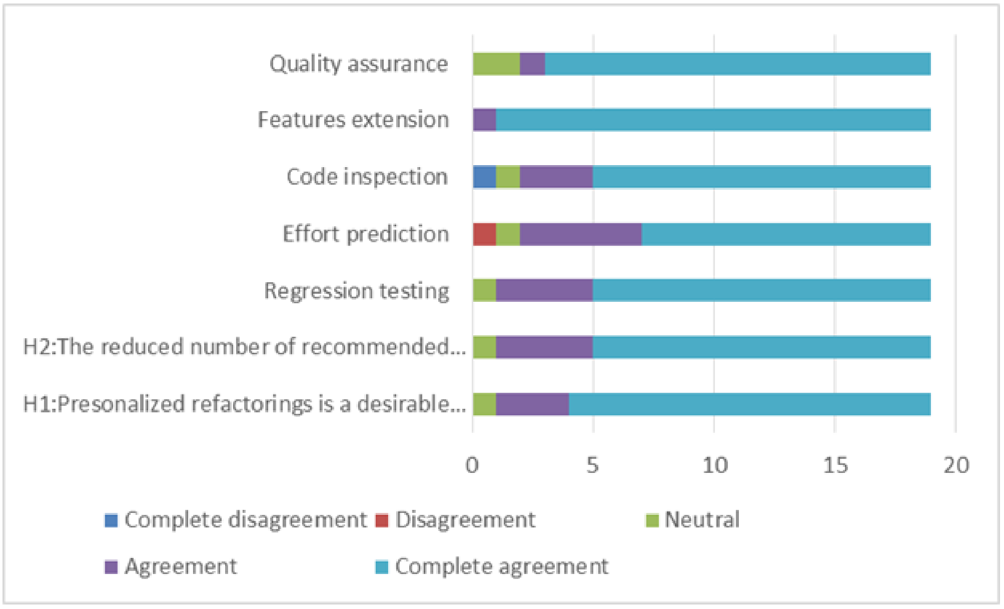


Figure 3.8: Post-study questionnaire results

operations that may jeopardize the quality of the entire system. In addition, this approach is suitable to increase the confidence of the developer since refactoring suggestions based on recent activities will hit his expectation.

The evaluation of the proposed technique shows that it help programmers take advantage of search-based refactoring tools with a reasonable execution time, and recommends a short list of refactorings. This results is a direct consequence of the reduction of the search space to explore, which is made possible by analyzing previous commits and bug reports. In addition, an evaluation of the proposed context-based multi-objective approach was performed on two industrial systems. The results of this evaluation show that the proposed technique outperforms several existing search-based refactoring approaches as well as JDeodorant, an refactoring tool not based on heuristic search. This outperformance was highlighted for comparison measures such as the relevance and correctness of the recommended refactorings.

In order to increase the confidence of programmers in using the proposed methodology, future work may involve the validation of this technique with additional refactoring tools and software systems. In addition, it is possible to extend the type of data used in the history of software change such as the type of code smells frequently corrected in the system.

CHAPTER 4

Interactive Search-based Software Refactoring Using Machine Learning

4.1 Introduction

Large scale software systems exhibit high complexity, and become challenging to maintain over time. In dynamic software engineering environment, where new functionalities are regularly added and new software engineer must adjust to the ongoing development process, both software evolution and maintenance become indispensable.

In general, the success of *timely* software maintenance and evolution activities is a function of the quality of the existing software design, both the architecture and the source code. Most specifically, the majority of difficulties encountered during maintenance operations can be significantly reduced for software systems built around well-known design patterns [17], and possessing easy-to-understand design structures. To achieve these software quality, search-Based Software Engineering (SBSE) techniques have been successfully applied to reformulate software refactoring as a search problem using metaheuristics [9, 10, 70, 67].

In the majority of exiting search-based software engineering methodologies, refactoring solutions are evaluated using software quality metrics as objective functions to guide the search process. Consequently, at each stage of the execution of the SBO algorithm that drives the search, solutions are selected based on their individual score for every metric included as objective function. Thus, during the decision making process, the domain expert must choose the solution that best matches his preferences based on the values of these quality metrics. However, when using these quality metrics to evaluate a software, several issues arise. First, there is no general consensus on the definition of design defects or code-smells, due to various programming behavior and contexts. For example, the definition of a large class can change from one software organization to another. Thus, it is difficult to formalize the definitions of design violations in terms of quality metrics to evaluate the quality of software refactoring solution. Second, the majority of existing refactoring studies do not include the developer - that is the final decision maker - in the loop to analyze the suggested refactoring solutions and give his or her feedback during the optimization process.

Various multi-objective optimization algorithms have been proposed that incorporate some user preferences to steer the search algorithm towards a Pareto front that meets pre-defined criteria [83, 84, 46, 50]. However, these methodologies assume that the decision maker has prior knowledge of possible region of convergence. However, in practice, not only it is difficult to predict the location of the Pareto front, but also can a bad guess lead to poor convergence towards the actual Pareto front. In fact, incorporating bad preference point could lead to locking the search algorithm in a region of the search space that is far from the actual Pareto front.

In [26], the authors used an interactive genetic algorithm to estimate the quality of refactoring solutions. However, the DM is required to evaluate every refactoring solution throughout the entire execution of the algorithm, making it fastidious and sometimes impractical. Third, the calculation of some quality metrics is expensive. Thus, the corresponding fitness function defined to evaluate refactoring solutions can be expensive. Finally, quality metrics can only evaluate the structural improvements of the design after applying the suggested refactoring solutions, but it is difficult to evaluate the semantic coherence of the design without the input of a developer .

In this chapter, we tackle a problem that is faced by the majority of search-based software engineering research: How do we define fitness when the computation and interpretation of fitness is an inherently subjective and aesthetic judgment that can only really be adequately made by human?

To answer this question, we propose a Genetic Algorithm (GA) based interactive learning algorithm for software refactoring based on Artificial Neural Networks (ANN) [85] [86]. We model the DM's preferences as a predictive model using ANN to approximate the fitness function for the evaluation of refactoring solutions. The developer is asked to evaluate manually refactoring solutions suggested by a Genetic Algorithm (GA) for a few iterations. Then, these evaluated solutions are used as training set for the ANN, and finally the ANN model is used to evaluate subsequent refactoring solutions in the next iterations. We evaluate our approach on open-source systems using existing benchmark [74, 20, 87]. We report the results on the efficiency and effectiveness of our approach, and compare it to existing refactoring methodologies [26, 16, 10, 66].

4.2 Refactoring as an Interactive Search-based Learning Problem

4.2.1 Approach Overview

We combine two methodologies to include the decision maker (DM) in the search loop: (1) A genetic Algorithm (GA)-based software refactoring, and (2) A learning system based on the well-known Artificial Neural Network (ANN). The proposed techniques takes as input the software system to refactor, the list of available refactoring operations, the maximum number of iteration (N) and the number of DM interactions (N_{DM}) with the search process. Upon reaching the stop criteria, or exhausting the maximum number of iterations, the system generates the best sequence of refactoring operations that improves the quality of the input software system. We divide the overall approach into two main components, namely, the Interactive Component (IGA) and the leaning module (LGA). Figure 4.1 shows a high-level view of the approach.

The IGA is the point of entry of our interactive software refactoring method. It includes the genetic algorithm (GA), which handle the search process as well as generates the refactoring sequences, and the decision maker (DM). After generating potential refactoring solutions, the IGA proposes them to the DM for evaluation. This evaluation is manually done by the DM for every refactoring solution before the training of the ANN, and takes into account its feasibility, efficiency and quality. After repeating this process of generation and evaluation for N_{DM} iterations, all the refactoring solutions evaluated by the DM are used as training set for the LGA.

The LGA components uses the previously formed training set to learn a ANN predictive model. Then, the predictive model is used in the remaining iterations of the GA to approximate the evaluation of the refactoring solutions. Thus, this approach does not require an explicit definition of a fitness function. In addition, since the techniques uses a data set evaluated by the DM to learn the ANN predictive model, it captures the DM preferences of good and bad refactoring in the model with a preset effort.

4.2.2 The Interactive Genetic Algorithm (IGA)

4.2.2.1 Solution Representation and Evaluation

The first part of the interactive software refactoring technique combines the search capability GAs and the expertise of the decision maker to find and accurately evaluate candidate refactoring solutions. When creating a sequence of refactorings (individuals), a number of pre- and post-conditions that guarantee the feasibility and applicability must be met. Opdyke was the first to introduce a way of formalizing the preconditions that must be imposed before a refactoring can be applied in order to preserve the external behavior of the system [56]. He defined functions which can be used to formalize constraints, similar to the Analysis functions proposed later by Ó Cinnéide[68], and Roberts [88]. Similar to [88], our

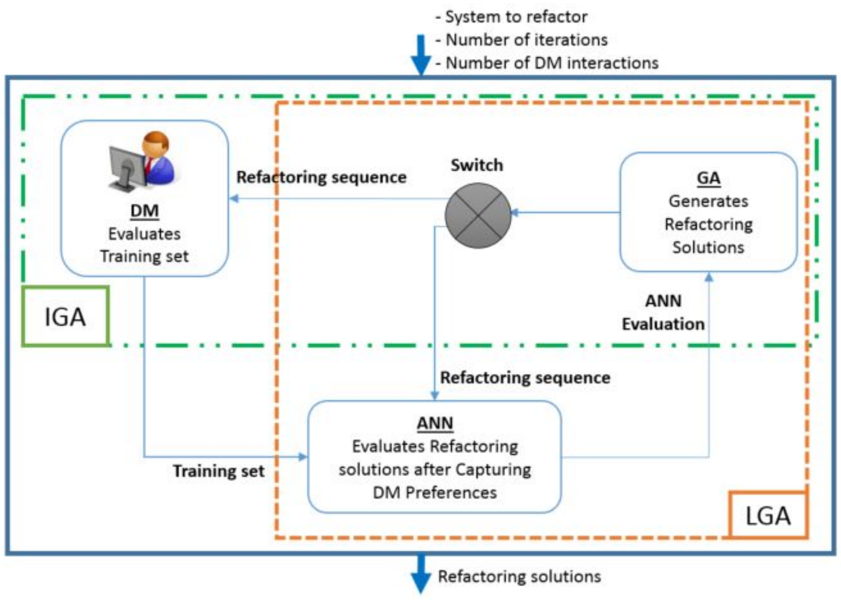


Figure 4.1: High-level view of the Learning-based interactive refactoring technique

method simulates refactorings using pre- and post-conditions that are expressed in terms of conditions on a code model. Our system suggests refactoring recommendations, and does not apply these refactorings automatically, but verifies the applicability of the suggested refactoring operations.

Each refactoring solution in itself is a sequence of refactoring operations such as Move Method (MM), Extract Class (EC), etc. Thus, we adopt a vector representation where each dimension of the vector is a refactoring operation. A typical solution consisting of m refactoring operations will be denoted as $[ro_1, ro_2, \ldots, ro_m]$, where $m > 0$ and ro_i is the i-th refactoring operation. As such, the position of each refactoring operation corresponds to the order of applying it to the system to refactor. In addition, for each refactoring operation, a set of controlling parameters stored in the vector such as actors and roles are randomly picked from the program to be refactored. Actors are expressed in terms of their properties such as number of methods.

After a solution is generated, it is presented to the DM for evaluation. Then, all evaluated solution are used to select the fittest solution to generate offspring solution for another round of evaluation. This process continue until the number of DM interactions (N_{DM}) is reached. Then, the system switches to the learning module for the evaluation of the remaining iterations of the GA.

4.2.2.2 Change Operators

Change operators are used to better explore the solution search space. We use the crossover and mutation operators to recombine the fittest solutions in each generation in order to create new solutions.

For the crossover, we use a single random cut-point crossover. It starts by randomly selecting and splitting two parent solutions to create two offsprings. The first child solution is formed by combining the first part of the first parent with the second part of the second parent. The second child solution is obtained by combining first part of the second parent with the second part of the first parent. This operator enforces the constraints on the solution length by randomly eliminating some refactoring operations. In every generation, each solution will be used as parent in at most one crossover operation.

The mutation operator randomly picks one or more refactoring operations from a sequence, and replaces them with randomly selected refactoring operation from the initial list of possible refactoring operations.

After applying genetic operators, crossover and mutation, we verify the feasibility of the generated sequence of refactoring by checking the pre- and post-conditions. Each refactoring operation that is not feasible due to unsatisfactory preconditions will be removed from the generated refactoring sequence.

4.2.3 The Learning-based Genetic Algorithm (LGA)

4.2.3.1 Overview of Artificial Neural Networks (ANNs)

Research on neural networks began as early as 1943 with the works of McCulloch and Pitts [89]. In their study, they give a mathematical model of the biological neural network in which neural events and the relations among them can be treated by means of propositional logic. Their work led to studies oriented toward the application of neural network to the field of artificial intelligence. Since their work was publish in 1943, many researchers have further the application of the concept to complex computing machine that seek to emulate the working mechanism of biological neurons.

The concept of learning was introduced by Hebb [90, 91], and In 1958, Rosenblatt proposed the first ANN model - the perceptron [92]. It is a simple model of the biological neuron, which takes input signals in a real vector format to produce an output through an associated vector of weights that represents the synaptic connections. Though the perceptron developed earlier is the building block for more complex ANN models, the explosion of researches and applications in the field of artificial neural networks was sparked by the works of Werbos who created the initial version of the backpropagation algorithm [93]. The backpropagation algorithm is a learning method used to train ANNs, and utilizes an optimization algorithm to calculate the weights of each neuron (node) in the network such that the error on the output is minimized. The name derives from the behavior of the network with respect to each input vector. While the input vector propagates forward (feed-forward) through the network from the input layer to the next until it reaches the output, the error incurred propagate backward - thus the backpropagation (of error).

Various models of ANNs have been studied, and more complex than the simple perceptron model have been established. Well-known complex ANN models are feed-forward models [94], recurrent models [95, 96], Radial Basis Functions [97], the self-organizing Maps [98], and the recursive neural networks [99, 100]. In this chapter, we choose the feed-forward ANN predictive model, namely, the multilayer perceptron. We do not give theoretical details of the model since it is out of the scope of this chapter. Rather, the predictive model is used as a black box after externally controllable parameters have been specified. Concerning the capability of feed-forward neural networks in approximating fitness functions, Hornick et al. proves that multilayered feed-forward networks are universal approximators [101]. Consequently, multilayer perceptons have been shown to be good approximators of fitness functions including those used in evolutionary computing [102].

The multilayer perceptron is a feed-forward ANN where all the neurons are fully connected. Our implementations uses three layers: The first layer consists of p input neurons, each being assigned a value $x_{kt}, t \in [1, \ldots, p]$. The second layer is the hidden layer, and is composed of a number of hidden neurons. Then we have the output layer, which has only one neurons that gives the values of the approximated fitness function. We calibrate the network to use the backpropagation (BP) algorithm as its learning method [103]. The BP is an algorithm that iteratively evaluates the output of the ANN against a desired output until the error is small enough to satisfy the performance constraints set by the user. We use a fast version of the BP where the performance is controlled by two parameters: (1) the momentum parameter used to avoid local minima by stabilizing weights, and (2) the learning rate of the underlying gradient optimization method, which allows for fast convergence.

In the following section, we give the details of the setting of the ANN, and the the structure of the training data.

4.2.3.2 Training set and Data Normalization

After the execution of the IGA, the refactoring solutions evaluated by the DM are organized to form the training data set used for the learning module (LGA). To this end, let X_k denote the k-th refactoring solution evaluated during the execution of the interactive component, where $1 \leq k \leq N$ and N is the number of DM interactions. X_k being a sequence of refactoring operation, it is coded as a vector, and each entry corresponds to a refactoring operation or a controlling parameter. Thus, $X_k = [x_{k1}, x_{k2}, \ldots, x_{kt}, \ldots, x_{kp}]$, where $t \in [1, \ldots, p]$ and p is the maximum length of the coded refactoring sequence. For each refactoring sequence, the DM assigns a score y_k in the range $[0 \ldots 1]$ where 0 is the worst score and 1 is assigned to the best refactoring sequence.

Let's denote by $\mathbf{O}$ the matrix of refactoring solutions and by $\mathbf{y}$ the vector of score values associated with the refactoring solutions. Then, $\mathbf{O}$ is an $N \times p$ matrix, and $\mathbf{y}$ is an p-dimensional vector. The training set is noted as $\mathcal{T} = \{\mathbf{O}, \mathbf{y}\}$, where $\mathbf{O} = [X_1, \ldots, X_N]^T$ is the input data and $\mathbf{y} = [y_1, \ldots, y_N]^T$ is the desired output, $\left({}^T\right)$ being the matrix transpose operator.

$$\mathbf{O} = \begin{bmatrix} x_{11} & x_{12} & \ldots & x_{1p} \\ x_{21} & x_{22} & \ldots & x_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ x_{N1} & x_{N2} & \ldots & x_{Np} \end{bmatrix} \quad \mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{bmatrix} \tag{4.1}$$

4.3 Validation

4.3.1 Research Questions

In our study, we assess the performance of our refactoring approach by finding out whether it could generate meaningful sequences of refactorings that fix design defects while reducing the number of code changes, preserving the semantic coherence of the design, and reusing as much as possible a base of recorded refactoring operations applied in the past in similar contexts. Our study aims at addressing the research questions outlined below.

Research question RQ1: To what extent can the proposed approach improve the design quality and propose efficient refactoring solutions?

Research question RQ2: How does the proposed approach perform compared to other existing search-based refactoring approaches and other search algorithms?

Research question RQ3: How does the proposed approach perform comparing to existing approaches not based on heuristic search?

To answer **RQ1**, we use two different validation methods: manual validation and automatic validation to evaluate the efficiency of the proposed refactorings. For the manual validation, we asked groups of potential users of our refactoring tool to evaluate, manually, whether the suggested refactorings are feasible and efficient. We define the metric refactoring efficiency (RE) which corresponds to the number of meaningful refactoring operations over the total number of suggested refactoring operations. For the automatic validation we compare the proposed refactorings with the expected ones using an existing benchmark [74][20][87]. In terms of recall and precision. The expected refactorings are those applied by the software development team to the next software release. To collect these expected refactorings, we use Ref-Finder [104], an Eclipse plug-in designed to detect refactorings between two program versions. Ref-Finder allows us to detect the list of refactorings applied to the current version we use in our experiments to suggest refactorings to obtain the next software version.

To answer **RQ2**, we compare our approach to two other existing search-based refactoring approaches: Kessentini et al. [16] and Harman et al. [10] that consider the refactoring suggestion task using fitness function as a combination of quality metrics (single objective). We also assessed the performance of our proposal with the IGA technique proposed by Ghannem et al. [26] where the developer evaluates all the solutions manually.

To answer **RQ3**, we compared our refactoring results with a popular design defects detection and correction tool JDeodorant [66] that do not use heuristic search techniques in terms precision, recall and RE. 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.

Table 4.1: Studied systems

Systems	Releases	# classes	KLOC
Xerces-J	v2.7.0	991	240
JFreeChart	v1.0.9	521	170
GanttProject	v1.10.2	245	41
AntApache	v1.8.2	1191	255
JHotDraw	v6.1	585	21
Rhino	v1.7R1	305	42

4.3.2 Experimental Settings

The goal of the study is to evaluate the usefulness and the effectiveness of our refactoring tool in practice. We conducted a non-subjective evaluation with potential developers who can use our refactoring tool. Our study involved a total number of 16 subjects. All the subjects are volunteers and familiar with Java development. The experience of these subjects on Java programming ranged from 2 to 15 years including two undergraduate students, four master students, six PhD students, one faculty member, and three junior software developers. Subjects were very familiar with the practice of refactoring.

We used a set of well-known and well-commented open-source Java projects. We applied our approach to six open-source Java projects: Xerces-J, JFreeChart, GanttProject, AntApache, JHotDraw, and Rhino. Xerces-J is a family of software packages for parsing XML. JFreeChart is a powerful and flexible Java library for generating charts. GanttProject is a cross-platform tool for project scheduling. AntApache is a build tool and library specifically conceived for Java applications. JHotDraw is a GUI framework for drawing editors. Finally, Rhino is a JavaScript interpreter and compiler written in Java and developed for the Mozilla/Firefox browser. We selected these 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 4.1 provides some descriptive statistics about these six programs.

To collect the expected refactorings applied to next version of studied systems, we use Ref-Finder [104]. Ref-Finder, implemented as an Eclipse plug-in, can identify refactoring

Table 4.2: Expected refactoring in next release of studied systems collected using Ref-Finder.

Systems	Next Releases	Expected # Refactorings
Xerces-J	v2.8.1	31
JFreeChart	v1.0.11	31
GanttProject	v1.11.2	46
AntApache	v1.8.4	78
JHotDraw	v6.2	27
Rhino	v1.7R4	46

Table 4.3: Parameter setting.

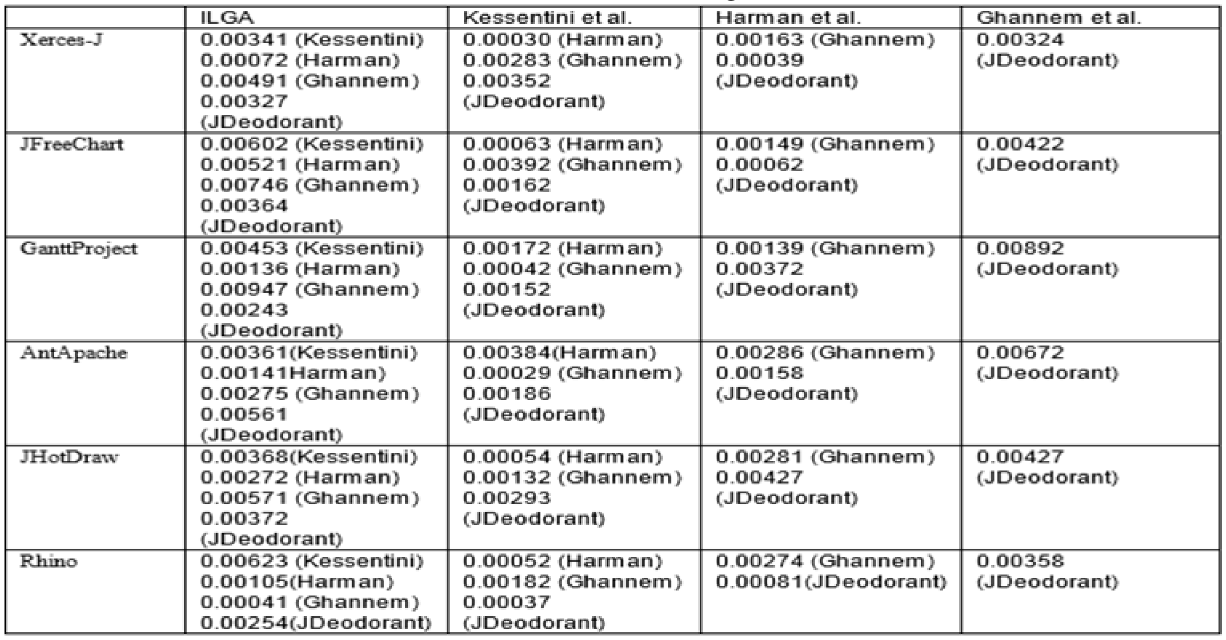
Parameter	Our approach	Ghannem	Harman et al.	Kessentini et al.
Population size	50	50	50	50
Termination criterion	10000	10000	10000	10000
Crossover probability	0.8	0.8	0.8	0.8
Mutation probability	0.2	0.2	0.2	0.2
Individual size	150	150	150	150
Number of interactions	35	90	N/A	N/A
Interaction sample size	4	4	N/A	N/A

operations applied between two releases of a software system. Table 4.2 shows the analyzed versions and the number of refactoring operations, identified by Ref-Finder, between each subsequent couple of analyzed versions, after the manual validation.

In our experiments, we use and compare different refactoring techniques. For each algorithm, to generate an initial population, we start by defining the maximum vector length (maximum number of operations per solution). The vector length is proportional to the number of refactorings that are considered and the size of the program to be refactored. During the creation, the solutions have random sizes inside the allowed range. For all algorithms, we fixed the maximum vector length to 150 refactorings,. We consider a list of 11 possible refactorings to restructure the design of the original program. Table 4.3 presents the parameter setting used in our experiments.

Our approach, like the others search-based approaches (Harman et al., Ghannem et al. and Kessentini et al.), is stochastic by nature, i.e., two different executions of the same algorithm with the same parameters on the same systems generally leads to different sets

Table 4.4: Parameter setting.



	ILGA	Kessentini et al.	Harman et al.	Ghannem et al.
Xerces-J	0.00341 (Kessentini) 0.00072 (Harman) 0.00491 (Ghannem) 0.00327 (JDeodorant)	0.00030 (Harman) 0.00283 (Ghannem) 0.00352 (JDeodorant)	0.00163 (Ghannem) 0.00039 (JDeodorant)	0.00324 (JDeodorant)
JFreeChart	0.00602 (Kessentini) 0.00521 (Harman) 0.00746 (Ghannem) 0.00364 (JDeodorant)	0.00063 (Harman) 0.00392 (Ghannem) 0.00162 (JDeodorant)	0.00149 (Ghannem) 0.00062 (JDeodorant)	0.00422 (JDeodorant)
GanttProject	0.00453 (Kessentini) 0.00136 (Harman) 0.00947 (Ghannem) 0.00243 (JDeodorant)	0.00172 (Harman) 0.00042 (Ghannem) 0.00152 (JDeodorant)	0.00139 (Ghannem) 0.00372 (JDeodorant)	0.00892 (JDeodorant)
AntApache	0.00361(Kessentini) 0.00141Harman) 0.00275 (Ghannem) 0.00561 (JDeodorant)	0.00384(Harman) 0.00029 (Ghannem) 0.00186 (JDeodorant)	0.00286 (Ghannem) 0.00158 (JDeodorant)	0.00672 (JDeodorant)
JHotDraw	0.00368(Kessentini) 0.00272 (Harman) 0.00571 (Ghannem) 0.00372 (JDeodorant)	0.00054 (Harman) 0.00132 (Ghannem) 0.00293 (JDeodorant)	0.00281 (Ghannem) 0.00427 (JDeodorant)	0.00427 (JDeodorant)
Rhino	0.00623 (Kessentini) 0.00105(Harman) 0.00041 (Ghannem) 0.00254(JDeodorant)	0.00052 (Harman) 0.00182 (Ghannem) 0.00037 (JDeodorant)	0.00274 (Ghannem) 0.00081(JDeodorant)	0.00358 (JDeodorant)

of suggested refactorings. To confirm the validity of the results, we executed each of the three algorithm 31 times and tested statistically the differences in terms of precision, recall, and RE. To compare two algorithms based on these metrics, we record the obtained metrics values for both algorithms over 51 runs. After that, we compute the metrics median value for each algorithm. Besides, we execute the Wilcoxon test with a 99% confidence level ($\alpha = 0.01$) on the recorded metrics values using the Wilcoxon MATLAB routine. If the returned p-value is less than 0.01 than, we reject H0 and we can state that one algorithm outperforms the other, otherwise we cannot say anything in terms of performance difference between the two algorithms. In table 4.4, we have performed multiple pairwise comparisons using the Wilcoxon test. Thus, we have to adjust the p-values. To achieve this task, we used Holm method which is reported to be more accurate than the Bonferroni one [105].

As interesting observation from the results that will be detailed in the next section is that the medians are close, the results are statistically different but the effect size which quantifies the difference is small for most of the systems and techniques considered in our experiments.

4.3.3 Results and Discussions

Results for RQ1: To answer **RQ1**, we need to assess the correctness/meaningfulness of the suggested refactorings from developers stand point. We reported the results of our empirical evaluation in Figure 4.2. We found that the majority of the suggested refactorings, with an average of more than 85% of RE, are considered by potential users as feasible, efficient in terms of improving the quality of the design and make sense.

In addition to the empirical evaluation performed manually by developers to evaluate the suggested refactorings, we automatically evaluate our approach without using the feedback of potential users to give more quantitative evaluation to answer **RQ1**. Thus, we compare the proposed refactorings with the expected ones. The expected refactorings are those applied by the software development team to the next software release as described in Table4.2. We use Ref-Finder to identify refactoring operations that are applied between the program version under analysis and the next version. Figures 4.3 and 4.4 summarizes our results. We found that a considerable number of proposed refactorings (an average of more than 80% for all studied systems in terms of recall) are already applied to the next version by software development team which is considered as a good recommendation score, especially that not all refactorings applied to next version are related to quality improvement, but also to add new functionalities, increase security, fix bugs, etc.

To conclude, we found that our approach produces good refactoring based on potential users of our refactoring tool and expected refactorings applied to the next program version.

Results for RQ2: To answer **RQ2**, we evaluate the efficiency of our approach comparing to three existing search-based refactoring contributions Harman et al. [10] Kessentini et al. [16] and Ghannem et al. [26]. In [10], Harman et al. proposed a multi-objective approach that uses two quality metrics to improve (coupling between objects CBO, and standard deviation of methods per class SDMPC) after applying the refactorings sequence.

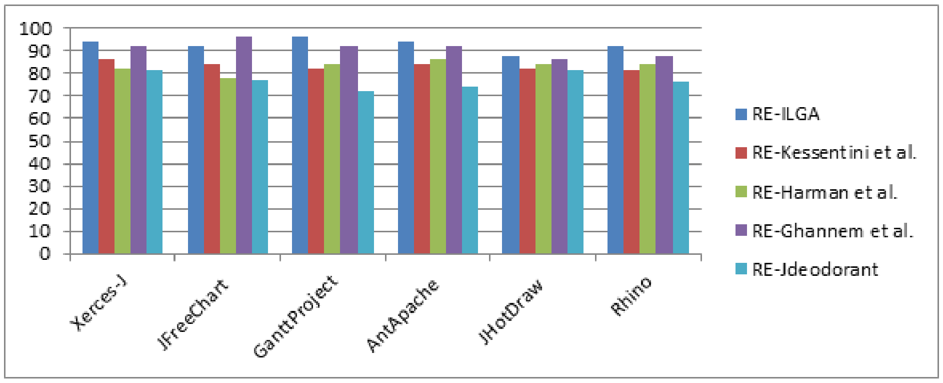


Figure 4.2: RE median values of ILGA, Kessentini et al., Harman et al., Ghannem et al. and JDeodorant over 31 independent simulation runs using the Wilcoxon rank sum test with a 99% confidence level ($\alpha < 1\%$).

In [16], a single-objective genetic algorithm is used to correct defects by finding the best refactoring sequence that reduces the number of defects. In [26], Ghannem et al. proposed an interactive Genetic Algorithm (IGA) for software refactoring where the user manually evaluates the suggested solutions by the GA. The comparison is performed through three metrics of Precision, Recall and RE. Figures $4.2-4.4$ summarize our findings and report the median values of each of our evaluation metrics obtained for 31 simulation runs of all projects.

We found that a considerable number of proposed refactorings (an average of 80% for all studied systems in terms of precision and recall) are already applied to the next version by software development team comparing to other existing approaches having only 65% and 74% for respectively Harman et al. and Kessentini et al. The precision and recall scores of the interactive approach proposed by Ghannem et al. are very similar to our approach (ILGA). However, our proposal requires much less effort and interactions with the designer to evaluate the solutions since the ANN replace the DM after a number of iterations/interactions. The same observations are also valid for RE where developers evaluated manually the best refactoring suggestions on all systems as described in Figure 4.2.

Results for RQ3: JDeodorant uses only structural information/improvements to sug-

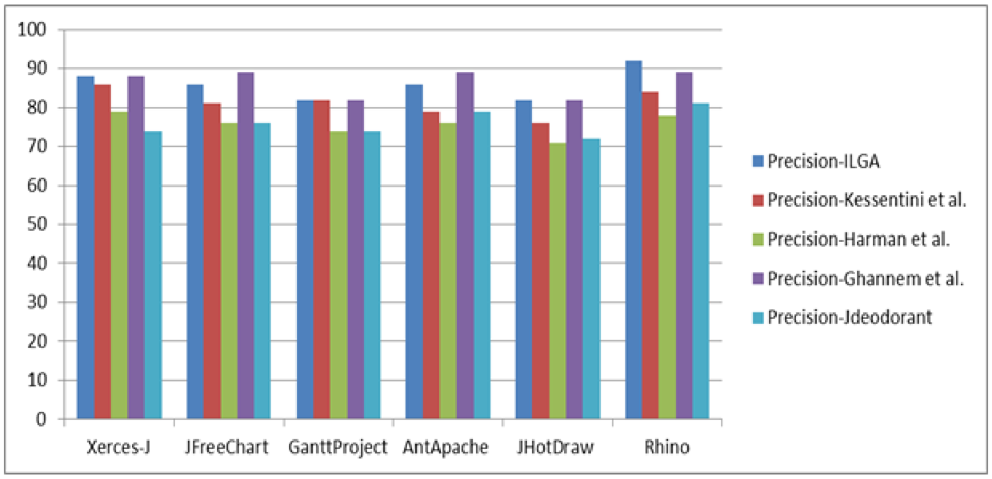


Figure 4.3: Precision median values of ILGA, Kessentini et al., Harman et al., Ghannem et al. and JDeodorant over 31 independent simulation runs using the Wilcoxon rank sum test with a 99% confidence level ($\alpha < 1\%$).

gest refactorings. Figures 4.2−4.4 summarize our finding. It is clear that our proposal outperforms JDeodorant, in average, on all the systems in terms of RE, precision and recall. This is can be explained by the fact that JDeodorant use only structural metrics to evaluate the impact of suggested and do consider the designer preferences and the programming context.

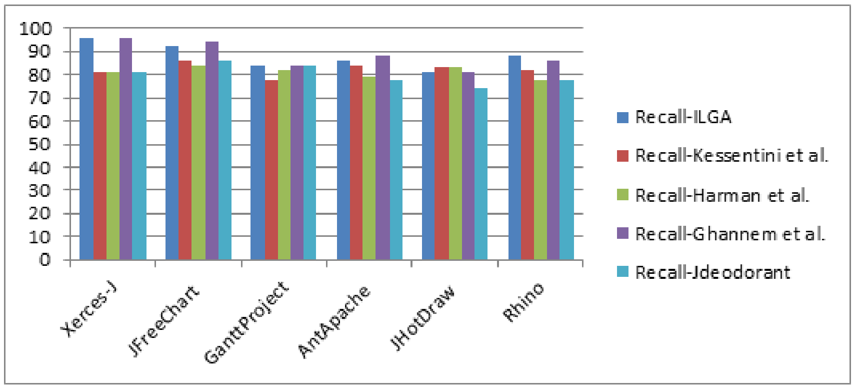


Figure 4.4: Recall median values of ILGA, Kessentini et al., Harman et al., Ghannem et al. and JDeodorant over 31 independent simulation runs using the Wilcoxon rank sum test with a 99% confidence level ($\alpha < 1\%$).

The number of interactions with the designer is a critical parameter of our ILGA approach to estimate the number of training examples required by the ANNs to generate a good predictive model of the fitness function. Figure 4.5 shows that an increase of the number of interactions improve the quality of the results on Xerces however after around 35 interactions (iterations) the precision and recall scores become stable. At each iteration, the designer evaluates 4 refactoring solutions. Thus, we defined the number of interactions empirically in our experiments based on this observation.

Usually in the optimization research field, the most time consuming operation is the evaluation step. Thus, we show how our ILGA algorithm is more efficient than existing search-based approaches from a CPU time (Computational Time) viewpoint. In fact, all the algorithms under comparison were executed on machines with Intel Xeon 3 GHz processors and 8 GB RAM. Figure 4.6 illustrates the obtained average CPU times of all algorithms on the systems. We note that the results presented in this figure were analyzed by using the same previously described statistical analysis methodology. In fact, based on the obtained p-values regarding CPU times, the ILGA is demonstrated to be faster than the remaining techniques as highlighted through Figure 4.6. The ILGA spends approximately the half amount of time required for IGA proposed by Ghannem et al. This observation could be explained by the fact that IGA requires a high number of interactions with the designer to evaluate the solutions which is a time consuming task. We can see that the use of an ANN to generate a surrogate fitness function seems to be an interesting approach to tackle software engineering problems where the individual evaluations are expensive like the case of software refactoring.

4.4 Conclusion

This chapter presented a novel interactive search-based learning refactoring approach that does not require the definition of a fitness function. while search-based software refactoring is a powerful tool to improve the design of existing software, most existing methodologies

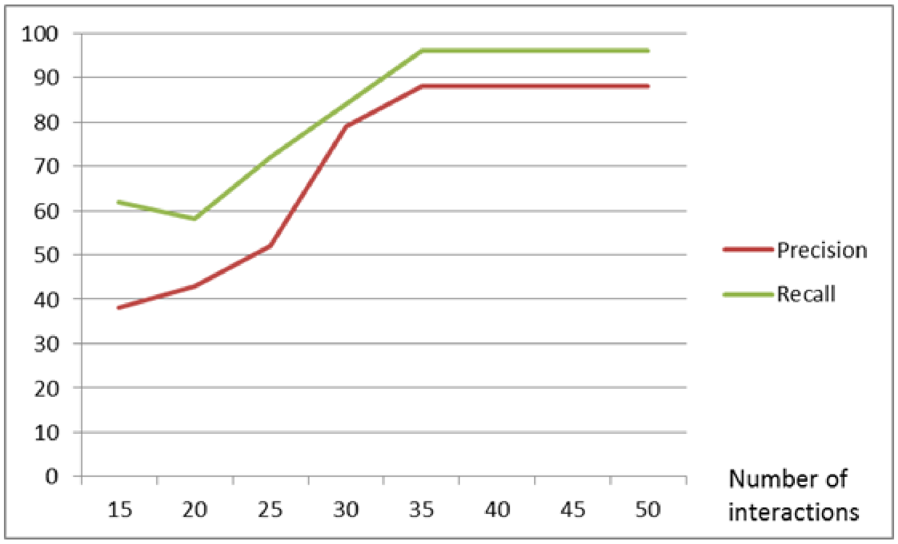


Figure 4.5: Number of interactions versus Precision/Recall on Xerces.

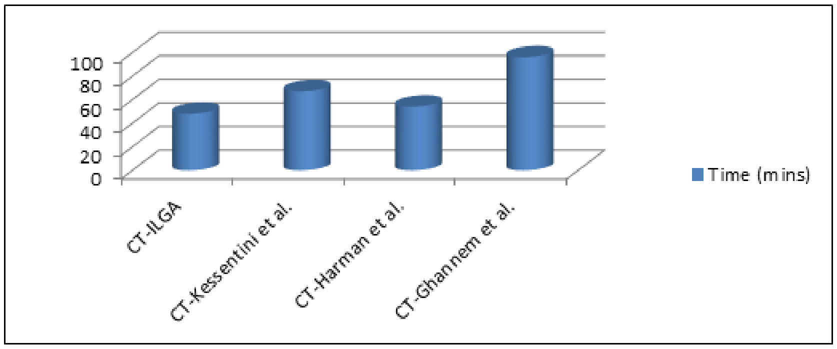


Figure 4.6: Execution time.

are, yet, to gain acceptance from the industrial software engineering world. The current setback in their adoption is due to the fact that automated refactorings recommendations do not take into account the preference of the software engineers. Thus, the main goal of the interactive search-based refactoring approach presented in this chapter is to allow the software engineer to be the decision maker during the search process. The developer is asked to evaluate manually refactoring solutions suggested by a Genetic Algorithm (GA) for few iterations then these examples are used as a training set for the ANNs to evaluate the solutions of the GA in the next iterations.

Another aspect of the proposed approach that was highlighted is the efficiency of integration of the developer in the loop. This is made possible by using a predictive model that capture the preference of the software engineer during a training step, then use it as fitness function for the remaining iteration of the search algorithm.

We evaluated our approach on open-source systems. Our evaluation results provide strong evidence that our technique successfully reduced the initial set of large number of objectives. The results also show that our approach outperforms several of existing multi-objective refactoring techniques based on several evaluation measures such as execution time, number of

xed antipatterns and manual correctness. Also, when existing refactoring techniques are better than our approach, we found that the results were comparable in magnitude.

In future work, we are planning to investigate an empirical study to consider additional systems and larger set of refactoring operations in our experiments. We are also planning to extend our approach to include the detection of refactoring opportunities using our interactive search-based learning approach.

CHAPTER 5

Improving Web Services Design Quality Using Heuristic Search and Machine Learning

5.1 Introduction

One of the important factors for deploying successful and popular services is to provide a well-designed interface to the users that can help them to easily find relevant operations [106]. In fact, the Web services interface. Web services interface could be provided by different service providers such as FedEx, Google, PayPal and Google, and represents the only visible part for the users to select the operations that they want to adopt in their implementation of services-based systems. Thus, the design quality of Web services interface is a critical and an important problem.

The evolution of Web services may have a negative impact on the design quality of the interface by concatenating many non-cohesive operations that are semantically unrelated. The Web services interface design becomes unnecessarily complex for users to find relevant operations to be used in their services-based systems. An example of well-known interface design defect is the God object Web service (GOWS) [19][107]. GOWS implements many operations related to different business and technical abstractions in a single service interface leading to low cohesion of its operations and high unavailability to end users because it is overloaded. Indeed, the modularization process of how operations should be exposed through a service interface can have an impact on the performance, popularity and reusability of the service and it is not a trivial task.

Recently, several studies provided solutions to improve the design of Web service interfaces for the users/subscribers [108][106][19][109][110][80]. However, most of these studies addressed the problem of the detection of design defects of Web services interface based on declarative rule specification and not the correction step to fix these design defects. In these existing techniques, Web services modularization solutions are evaluated based on the use of quality metrics. However, the evaluation of the design quality is subjective and difficult to formalize using quality metrics with the appropriate threshold values due to several reasons.

Several challenges could be discussed around the modularization of Web services interface. First, there is no consensus about the definition of Web services design defects [107][111][9][112], also called antipatterns, due to the various user behaviors and contexts. Thus, it is difficult to formalize the definitions of these design violations in terms of quality metrics then use them to evaluate the quality of a Web service modularization solution. Second, existing studies do not include the user in the loop to analyze the suggested modularization solutions and give their feed-back during the design improvement process. Third, the computational complexity of some Web services quality metrics is expensive thus the defined fitness function to evaluate proposed Web services design changes can be expensive. Fourth, deciding on how to decompose/modularize an interface is subjective and difficult to automate since it is required to integrate the feedback of users during the modularization process Finally, quality metrics can just evaluate the structural improvements of the design after applying the suggested interface changes but it is difficult to evaluate the semantic coherence of the design without an interactive user interpretation.

We propose, in this chapter, a Genetic Algorithm (GA)-based interactive learning algorithm [113] for Web services interface modularization based on Artificial Neural Networks (ANN) [114]. The proposed approach is based on the important feedback of the user to guide the search for relevant Web services modularization solutions using predictive models. To the best of our knowledge, the use of predictive models has not been used to improve the quality of Web services design. In the proposed approach, we are modeling the users design preferences using ANN as a predictive model to approximate the fitness function for the evaluation of the Web services modularization solutions. The user is asked to evaluate manually Web services interface modularization solutions suggested by a Genetic Algorithm (GA) for few iterations then these examples are used as a training set for the ANNs to evaluate the solutions of the GA in the next iterations.

We evaluated our approach on a set of 82 real-world Web services, extracted from an existing benchmark [19][109]. Statistical analysis of our experiments shows that our interactive approach performed significantly better than the state-of-the-art modularization techniques [109][110] in terms of design improvements and fixing design defects. The primary contributions of this chapter can be summarized as follows:

1. The paper introduces a novel way to modularize and improve the design quality of Web services using interactive predictive modeling optimization. The proposed technique supports the adaptation of interface design solutions based on the user without the need to use specific design quality metrics. To the best of our knowledge, we propose the first approach to interactively generate a modularized Web services interface using predictive modeling techniques.

2. The paper reports 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 Web services modularization techniques based on 82 real-world services.

5.2 Web Service Remodulation

The interface of a Web service is described as a WSDL (Web service Description Language) document that contains structured information about the offered operations and their input/output parameters [115][116]. A port Type is a set of abstract operations. Each operation refers to an input message and output messages. The users select the desired operation on their services-based system implementation via the interface by specifying the name of the operations and the required parameters (inputs) and they receive the required outputs without accessing to the source code of these used operations.

Most of existing real-world Web services interface regroup together a high number operations implementing different abstractions such as the Amazon EC2 that contains more than 100 operations in some releases. There are few WSDL design improvement tools [115][116][117] that have emerged to provide basic refactorings on WSDL files however applying these design changes is fully manual and time consuming as discussed in the next section. These interface design changes correspond mainly to Interface Decomposition, Interface Merging (to merge multiple interfaces) and Move Operation (to move an operation between different interfaces).

Web service interface defects are defined as bad design choices that can have a negative impact on the interface quality such as maintainability, changeability and comprehensibility which may impacts the usability and popularity of services [107]. To this end, recent studies defined different types of Web services design defects [19][15]. In our experiments, we focus on the seven following Web service defect types: God object Web service (GOWS): implements a high number of operations related to different business and technical abstractions in a single service. Fine grained Web service (FGWS): is a too fine-grained service whose overhead (communications, maintenance, and so on) outweighs its utility. Chatty Web service (CWS): represents an antipattern where a high number of operations are required to complete one abstraction. Data Web service (DWS): contains typically accessor operations, i.e., getters and setters. In a distributed environment, some Web services may only perform some simple information retrieval or data access operations. Ambiguous Web service (AWS): is an antipattern where developers use ambiguous or meaningless names for denoting the main elements of interface elements (e.g., port types, operations, messages). Redundant PortTypes (RPT): is an antipattern where multiple port Types are duplicated with the similar set of operations. CRUDy Interface (CI): is an antipattern where the design encourages services the RPC-like behavior by declaring create, read, update, and delete (CRUD) operations, e.g., **createX(), readY**(), etc. We choose these defect types in our interactive interface design tool because they are the most frequent and hard to detect [80], cover different interface design issues, due to the availability of defect examples and could be detected using a tool proposed in the literature [19][80][118][119].

Detecting and specifying antipatterns in SOA and Web services is a relatively new area. The first book in the literature was written by Dudney et al. [107] and provides informal definitions of a set of Web service antipatterns. More recently, Rotem-Gal-Oz described the symptoms of a range of SOA antipatterns [106]. Furthermore, Rodriguez et al. [117]provided a set of guidelines for service providers to avoid bad practices while writing WSDLs. Based on some heuristics, the authors detected eight bad practices in the writing of WSDL for Web services.

In [15], the authors presented a repository of 45 general antipatterns in SOA. The goal of this work is a comprehensive review of these antipatterns that will help developers to work with clear understanding of patterns in phases of software development and so avoid many potential problems. Mateos et al. [115][116] have proposed an interesting approach towards generating WSDL documents with less antipatterns using text mining techniques.

In our previous work [19], we proposed a search-based approach based on standard GP to find regularities, from examples of Web service antipatterns, to be translated into detection rules. However, the proposed approach can deal only with Web service interface metrics and cannot consider all Web service antipattern symptoms.

Recently, few studies are proposed to restructure the design of the Web services interface [106][109][110]. We can distinguish two main categories: manual and fully-automated techniques. The manual approaches propose a set of interface design changes that the user can select and execute to split an interface, extract an interface and merge two interfaces

[108]. However, manual refactoring of the interfaces design is a tedious task for developers that involve exploring the whole operations in the interface to find the best refactoring solution that improves the modularity of an interface. In the fully-automated approach, developers should accept the entire design changes solution and existing tools do not provide the flexibility to adapt the suggested solution interactively. In addition, most of these manual and fully-automated techniques focus on fixing design defects rather than the modularity of the interface [19][80].

In the following, we introduce some issues and challenges related to restructuring the design quality of the Web service interfaces. Figure 1 illustrates a fine-grained service that can lead to a system with a poor performance due to an excessive number of calls to one interface regrouping all the operations. Thus, it is critical to fix this issue by creating new port Types that group together the most cohesive operations to decompose the Amazon Simple Notification Service interface.

Overall, there is no consensus on how to decide if a design violates a quality heuristic. In fact, there is a difference between detecting symptoms and asserting that the detected situation is an actual design defect. Another issue is related to the definition of thresholds when dealing with quantitative information. For example, the GOWS defect detection involves information such as the interface size as illustrated in Figure 5.1. Although we can measure the size of an interface, an appropriate threshold value is not trivial to define. An interface considered large by a community of service users could be considered average by others. Thus, it is important to consider the user in the loop when identifying such design violations.

Several possible levels of interaction are not considered by existing Web services interface refactoring techniques. Overall, most of refactoring studies are based on the use of quality metrics as a fitness function to evaluate the quality of the design after applying design changes. However, these metrics can only evaluate the structural improvements. Furthermore, the efficient evaluation of the suggested refactoring from a semantic perspec-

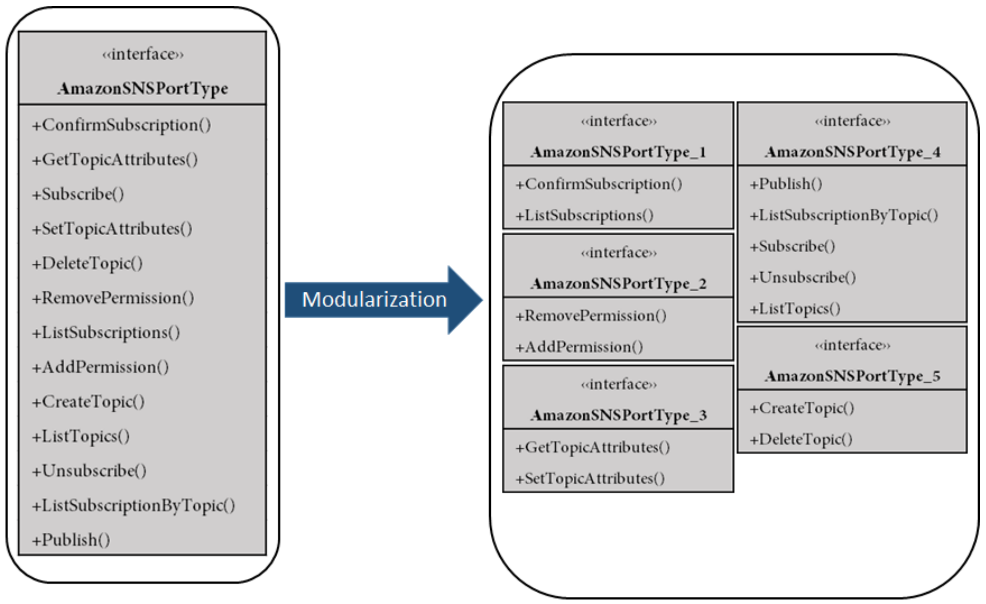


Figure 5.1: Restructuring the design of a Web service Interface example (Amazon Simple Notification Service

tive requires an interaction with the designer. In addition, the symptoms of design defects are difficult to formalize using quality metrics due to the very subjective process to identify them that depends on the programming context and the preferences of developers. Finally, the definition of a fitness function based on quality metrics can be expensive.

To address these challenges, we describe in the next section our approach based on machine learning and heuristic-based techniques to evaluate the Web services modularization solutions without the need to explicitly define a fitness function. This work represents one of the first studies in this area.

5.3 Approach

5.3.1 Approach Overview

As described in Figure 5.2, our approach takes as input the Web services interface to modularize, list of possible operators (decompose a port Type or merge port Types or move operations) and the number of users interactions during the search process. It generates as output the best sequence of design changes/operators that improves the quality of the Web service interface. Our approach is composed of two main components: the interactive component (IGA) and the learning module (LGA).

The algorithm starts first by executing the IGA component where the designer evaluates the modularization solutions manually generated by a genetic algorithm (GA) [113] for a number of iterations. The user evaluates the feasibility and the efficiency/quality of the suggested suggestions one by one since each modularization solution is a sequence of change operator (decompose or merge or move). Thus, the user classifies all the suggested design changes (modules) as good or not one by one based on his preferences and gives the different port Types values between 0 and 1.

After executing the IGA component for a number of iterations, all the evaluated solutions by the user are considered as training set for the second component LGA of the algorithm. The LGA component executes an Artificial Neural Network (ANN)[120] to generate a predictive model to approximate the evaluation of the interface modularization solutions in the next iteration of the GA. Thus, our approach does not require the definition of a fitness function. Alternatively, the LGA incorporates many components to approximate the unknown target function . Those components are the training set, the learning algorithm and the predictive model. For each new sequence of refactoring , the goal of learning is to maximize the accuracy of the evaluation . We applied the ANN as being among the most reliable predictive models, especially, in the case of noisy and incomplete data. Its architecture is chosen to be a multilayered architecture in which all neurons are fully connected; weights of connections have been, randomly, set at the beginning of the training. Regarding the activation function, the sigmoid function is applied [114] as being adequate in the case of continuous data. The network is composed of three layers: the first layer is composed of p input neurons. Each neuron is assigned the value . The hidden layer is composed of a set of hidden neurons. The learning algorithm is an iterative algorithm that allows the training of the network. Its performance is controlled by two parameters. The first parameter is the momentum factor that tries to avoid local minima by stabilizing weights. The second factor is the learning rate which is responsible of the rapidity of the adjustment of weights.

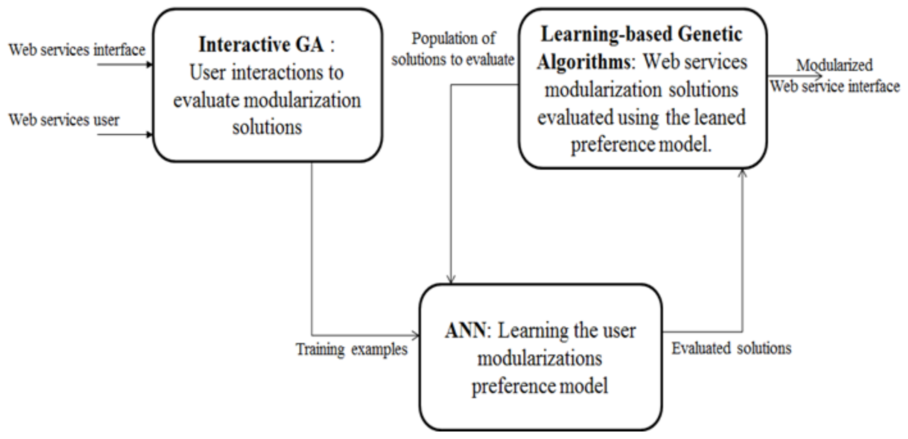

Web services interface
Web services user
Interactive GA : User interactions to evaluate modularization solutions
Population of solutions to evaluate
Learning-based Genetic Algorithms: Web services modularization solutions evaluated using the leaned preference model.
Modularized Web service interface
Training examples
ANN: Learning the user modularizations preference model
Evaluated solutions

Figure 5.2: Approach overview

5.3.2 Algorithm Adaptation

5.3.2.1 Solution Coding

A solution consists of a sequence of n interface change operations assigned to a set of port types. A port type could contain one or many operations but an operation could be assigned to only one port type. A vector-based representation is used to cluster the different operations of the original interface, taken as input from the WSDL file description, into appropriate interfaces, i.e., port types. Figure 5.3 describes an example of 5 operations assigned to two port types.

The initial population is generated by randomly assigning a sequence of operations to a randomly chosen set of port Types. The size of a solution, i.e. the vectors length corresponds to the number of operations of the Web service however the number of port Types is randomly chosen between upper and lower bound values. The determination of these two

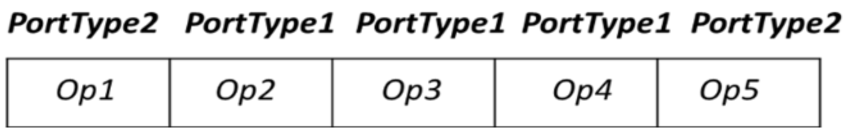


Figure 5.3: Example of a solution representation

bounds is similar to the problem of bloat control in genetic programming [113] where the goal is to identify the tree size limits. The number of required port types depends on the size of the target interface design. Thus, we performed, for each target design, several trial and error experiments using the HyperVolume (HP) [113] performance indicator to determine the upper bound after which, the indicator remains invariant. For the lower bound, it is arbitrarily chosen.

5.3.2.2 Training Set and Data Normalization

Before the learning process, the data used in the training set should be normalized. In our case, we choose to apply the Min-max technique since it is among the most accurate techniques according to [121]. We used the following data representation to the GA-based learning problem using ANN for software refactoring. Let us denote by $\mathcal{T}$ the training set of the ANN. It is composed of a set of couples that represent the refactoring sequence and its evaluation. Let X_k denote the k-th refactoring solution evaluated during the execution of the interactive component, where $1 \leq k \leq N$ and N is the number of DM interactions. X_k being a sequence of refactoring operation, it is coded as a vector, and each entry corresponds to a refactoring operation or a controlling parameter. Thus, $X_k = [x_{k1}, x_{k2}, \ldots, x_{kt}, \ldots, x_{kp}]$, where $t \in [1, \ldots, p]$ and p is the maximum length of the coded refactoring sequence. For each refactoring sequence, the DM assigns a score y_k in the range $[0 \ldots 1]$ where 0 is the worst score and 1 is assigned to the best refactoring sequence.

Let's denote by $\mathbf{O}$ the matrix of refactoring solutions and by $\mathbf{y}$ the vector of score values associated with the refactoring solutions. Then, $\mathbf{O}$ is an $N \times p$ matrix, and $\mathbf{y}$ is an p-dimensional vector. The training set is noted as $\mathcal{T} = \{\mathbf{O}, \mathbf{y}\}$, where $\mathbf{O} = [X_1, \ldots, X_N]^T$ is the input data and $\mathbf{y} = [y_1, \ldots, y_N]^T$ is the desired output, $\left({}^T\right)$ being the matrix transpose operator.

$$\mathbf{O} = \begin{bmatrix} x_{11} & x_{12} & \ldots & x_{1p} \\ x_{21} & x_{22} & \ldots & x_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ x_{N1} & x_{N2} & \ldots & x_{Np} \end{bmatrix} \qquad \mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{bmatrix} \tag{5.1}$$

5.3.2.3 Change Operators

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 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. It is important to note that in multi-objective optimization, it is better to create children that are close to their parents to have a more efficient search process. For mutation, we use the bit-string mutation operator that picks probabilistically one or more modularization operations from its or their associated sequence and replaces them by other ones from the initial list of possible refactorings.

When applying the change operators, different pre- and post-conditions are checked to ensure the applicability of the newly generated solutions such as removing redundant operations or conflicts between operations such as assigning the same operation to two different port types.

5.4 Validation

To evaluate the ability of our Web services modularization framework to generate a good design quality, we conducted a set of experiments based on 82 real-world web services as described in Table 5.1. the obtained results are subsequently statistically analyzed with the aim of comparing our proposal with a variety of existing fully-automated approaches. In this section, we first present our research questions and then describe and discuss the obtained results.

5.4.1 Research Questions

We defined three research questions that address the applicability, performance in comparison to existing fully-automated Web services modularization approaches [109][110], and the usefulness of our approach. The three research questions are as follows:

RQ1: To what extent can our approach recommend relevant Web services design improvements?

RQ2: How does our interactive formulation perform compared to fully-automated Web services restructuring techniques?

RQ3: Can our approach be useful for the users of Web services (the developers of service-based systems)?

To answer these research questions, we considered the best interface design restructuring solutions recommended by our approach. To answer **RQ1**, it is important to validate the proposed modularization solutions on the different Web services highlighted in Table 5.1. We asked a group of developers, as detailed in the next section, to manually modularize the design of the different interfaces considered in our experiments. Then, we calculated precision (PR) and recall (RC) scores to compare between the generated design and the expected one.

When calculating the precision and recall, we consider a two port types are similar if they contain the same operations. We divided the participants in groups to make sure that they do not use our tool on the Web services that they are asked to manually modularize.

$$\text{PR} = \frac{|\text{suggested portTypes} \cap \text{expected portTypes}|}{|\text{suggested portTypes}|} \in [0,1] \tag{5.2}$$

$$\text{RC} = \frac{|\text{suggested portTypes} \cap \text{expected portTypes}|}{|\text{expected portTypes}|} \in [0,1] \tag{5.3}$$

Another metric that we considered for the quantitative evaluation is the percentage of fixed design antipatterns (NF) by the proposed modularization solution. The detection of design antipatterns after applying a modularization solution is performed using the detection rules of our previous work [19]. Formally, NF is defined as

$$\text{NF} = \frac{\text{\# fixed design antipatterns}}{\text{\# design antipatterns}} \in [0,1] \tag{5.4}$$

For the qualitative validation, we asked groups of potential users of our Web services modularization tool to evaluate, manually, whether the suggested interface design modularizations are feasible and efficient at improving the quality of Web services interface design. We define the metric Manual Correctness (MC) to mean the number of meaningful Web services interface refactorings divided by the total number of recommended refactorings by our tool. MC is given by the following equation:

$$\text{NF} = \frac{\text{\# correct modularization operations}}{\text{\# proposed modularization operations}} \in [0,1] \tag{5.5}$$

To answer **RQ2**, we compared our approach to two other existing fully-automated Web services decomposition techniques [109][110]. Ouni et al. [110] proposed an approach to decompose Web services using graph partitioning to improve cohesion. Similarly, Athanasopoulos et al. [109] used a greedy algorithm to decompose the interface based on cohesion as well. All these existing techniques are fully-automated and do not provide any

Table 5.1: Web service statistics

Web Service Provider	#services	#operations (min, max)
FedEx	19	$(13, 36)$
Amazon	16	$(16, 93)$
Yahoo	18	$(11, 41)$
Ebay	12	$(13, 37)$
Microsoft	17	$(11, 59)$

interaction with the developers to update their solutions towards a desired design. We also compared the running time T of the proposed algorithm comparing to fully automated techniques. Thus, we used the metrics PR, RC, T and NF to perform the comparisons.

To answer **RQ3**, we used a post-study questionnaire that collects the opinions of Web service developers on our tool as described in the next section. Thus, we asked these participants to use both our tool and the automated framework proposed by Ouni et al. [19] on different sets of Web services. The participants were asked to make changes, when appropriate, to the final solution of the automated approach of Ouni et al. [19]. Thus, we can check whether the interactive component of the proposed interactive approach makes a real contribution, or whether the same effect can be attained by just fixing the output of the automated remodularization approaches. We measured the time spent by the developers on using our interactive approach and the automated techniques. Then, we compared between the outcomes of the survey questions for both interactive and fully automate techniques.

5.4.2 Experimental Setting

We extracted a set of 82 well-known Web services from an existing benchmark [19][109] as detailed in Table 5.1 . All studied services are widely used in different contexts and provided by Amazon, FedEx, Ebay, Microsoft and Yahoo, five major Web service providers. We selected these Web services for our validation because they range from medium to large-sized interfaces, which have been actively developed and changed over several years. Our study involved 36 participants from the University of Michigan to use and evaluate our tool. Participants include 27 master students in Software Engineering and 9 Ph.D. students

Table 5.2: Survey organization

Groups	Web Services
Group 1	FedEx
Group 2	Amazon
Group 3	Yahoo
Group 4	Ebay
Group 5	Microsoft, Ebay
Group 6	FedEx, Yahoo

in Software Engineering. All the participants are volunteers and familiar with Web services and refactoring in general. The experience of these participants on programming ranged from 3 to 17 years. 19 out of the 36 participants are currently active programmers as well in software industry with a minimum experience of 3 years. Participants were first asked to fill out a pre-study questionnaire containing nine questions. The questionnaire helped to collect background information such as their role within the company, their programming experience, their familiarity with Web services. As part of the Software Quality Assurance graduate course, all the participants attended two lectures about Web services design quality, modularization and passed five tests to evaluate their performance to evaluate and suggest interface design modularization solutions.

As described in Table 5.2, we formed 6 groups. Each of the 6 groups is composed by 6 participants. Table 5.2 summarizes the survey organization including the list of Web services and the algorithms evaluated by each of the groups. The groups were formed based on the pre-study questionnaire and the tests result to make sure that all the groups have almost the same average skills. Consequently, each group of participants who accepted to participate in the study received a questionnaire, a manuscript guide to help them to fill the questionnaire, the tools and results to evaluate the Web services design. Since the application of remodularization solutions is a subjective process, it is normal that not all the developers have the same opinion. In our case, we considered the majority of votes to determine if suggested solutions are correct or not.

We executed three different scenarios. In the first scenario, we asked every participant to manually modularize a set of Web services. As an outcome of the first scenario, we calculated the differences between the recommended modularizations and the expected ones (manually suggested by the users/developers). To evaluate the fixed Web services design antipatterns, we focus on the ones defined in Section 5.2. In the second scenario, we asked the users to manually evaluate the recommended solution by our algorithm. We performed a cross-validation between the groups to avoid the computation of the MC metric being biased by the developers feedback. In the third scenario, we collected their opinions of the participants based on a post-study questionnaire that will be detailed before in this section. The participants were asked to justify their evaluation of the solutions and these justifications are reviewed by the organizers of the study.

Parameter setting influences significantly the performance of a search algorithm. For this reason, for each algorithm and for each Web service, we perform a set of experiments using several population sizes: $20, 30$ and 50. We limited the interaction with the user in our approach to a maximum of 30. The stopping criterion was set to 1000 evaluations for all algorithms to ensure fairness of comparison. The other parameters values were fixed by trial and error and are as follows: (1) crossover probability = 0.5; mutation probability = 0.2 where the probability of gene modification is 0.1. Each algorithm is executed 30 times with each configuration and then the comparison between the configurations is done using the Wilcoxon test. To achieve significant results, for each couple (algorithm, Web service), we use the trial and error method to obtain a good parameter configuration. Since metaheuristic 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 of the automated approaches and the obtained results are statistically analyzed by using the Wilcoxon rank sum test with a 95% confidence level ($\alpha = 5\%$). The latter tests the null hypothesis, H0, that the obtained results of two algorithms are samples from continuous distributions with equal medians, against the alternative that they are not, H1. The p-value of the Wilcoxon test corresponds to the probability of rejecting the null hypothesis H0 while it is true (type I error). A p-value that is less than or equal to α (≤ 0.05) means that we accept H1 and we reject H0. However, a p-value that is strictly greater than α (> 0.05) means the opposite. In fact, for each problem instance, we compute the p-value obtained by comparing existing studies [109][110] results with our approach ones. In this way, we determine whether the performance difference between our technique and one of the other approaches is statistically significant or just a random result. The results presented were found to be statistically significant on 30 independent runs of the fully-automated approaches using the Wilcoxon rank sum test with a 95% confidence level ($\alpha = 5\%$) as detailed in the next section.

5.4.3 Results and Discussions

Results for RQ1. As described in Figures 5.4 and 5.5, we found that a considerable number of proposed port types, with an average of more than 81% in terms of precision and recall on all the 82 Web services, were already suggested manually (expected refactorings) by the users (software development team). The achieved recall scores are slightly higher, in average, than the precision ones since we found that some of the port types suggested manually by developers do not exactly match the solutions provided by our approach. In addition, we found that the slight deviation with the expected port types is not related to incorrect ones but to the fact that different possible modularization solutions could be optimal.

We evaluated the ability of our approach to fix several types of interface design antipatterns and to improve the quality. Figure 5.6 depicts the percentage of fixed code smells (NF). It is higher than 82% on all the Web services, which is an acceptable score since users may not be interested to fix all the antipatterns in the interface. We reported the results of our empirical qualitative evaluation in terms of manual correctness (MC) in Figure 5.7. As reported in Figure 5.7, most of the Web services modularization solutions recommended by our interactive approach were correct and approved by developers. On average, for the different Web services, 88% of the created port types and applied changes to the initial design are considered as correct, improve the quality, and are found to be useful by the software developers of our experiments. Thus, we found that the slight deviation with the expected design is not related to incorrect changes but to the fact that the developers have different scenarios/contexts in using the different operations.

To summarize and answer **RQ1**, the experimentation results con-firm that our interactive approach helps the participants to re-structure their Web service interface design efficiently by finding the relevant portTypes and improve the quality of all the 22 Web services.

Results for RQ2. Figures $5.4-5.7$ confirm the average superior performance of our interactive learning GA approach compared to the two existing fully automated Web service modularization techniques [109][110]. Figure $5.7 showsthatourapproachprovidessignificantlyhigher$
and $5.5. Theoutperformanceofourtechniqueintermsofpercentageoffixeddefects, asdescribedinFi$
can be explained by the fact that the main goal of existing studies is not to mainly fix these defects (not considered in the fitness function by the work of Ouni et al. [110]).

In conclusion, our interactive approach provides better results, on average, than all existing fully-automated Web services modularization techniques.

Results for RQ3. To further analyze the obtained results, we have also asked the participants to take a post-study questionnaire after completing the different validation and tasks using our interactive approach and the two techniques considered in our experiments. The post-study questionnaires collected the opinions of the participants about their experience in using our approach compared to fully-automated tools. The post-study questionnaire 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 interface modularization recommendations using our predictive modeling approach are a desirable feature to improve the quality of Web services interface. (2) The interactive manner of recommending modularization solutions by our GA learning approach is a useful and flexible way to consider the user perspective compared to fully-automated tools.

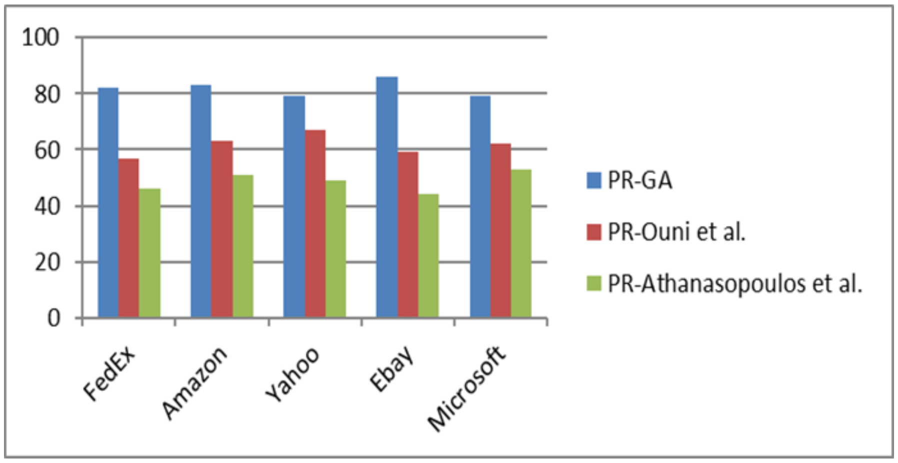


Figure 5.4: Median precision (PR) value over 30 runs on all Web services using the different modularization techniques with a 95% confidence level ($\alpha = 5\%$).

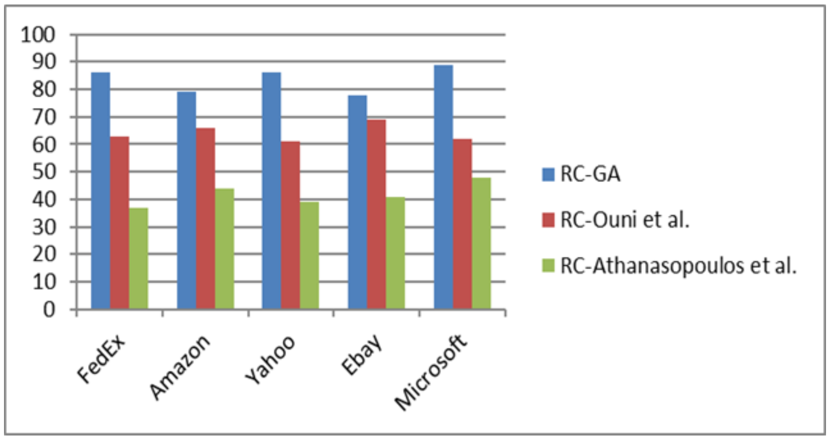


Figure 5.5: Median recall (RC) value over 30 runs on all Web services using the different modularization techniques with a 95% confidence level ($\alpha = 5\%$).

The agreement of the participants was 4.6 and 4.2 for the first and second statements respectively. This confirms the usefulness of our approach for the users of our experiments. The remaining questions of the post-study questionnaire were about the benefits and the limitations (possible improvements) of our interactive approach.

We summarize in the following the feedback of the users. Most of the participants mention that our interactive approach is much faster and easy to use compared to the manual

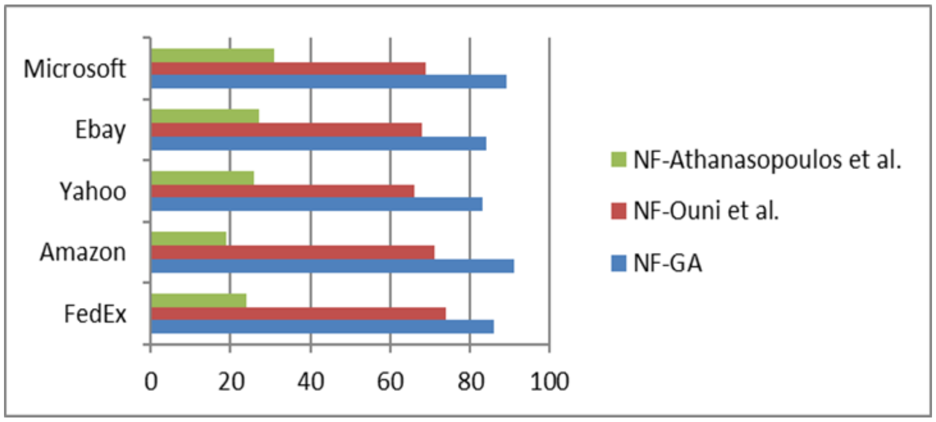


Figure 5.6: Median number of fixed Web service defects (NF) value over 30 runs on all Web services using the different modularization techniques with a 95% confidence level ($\alpha = 5\%$).

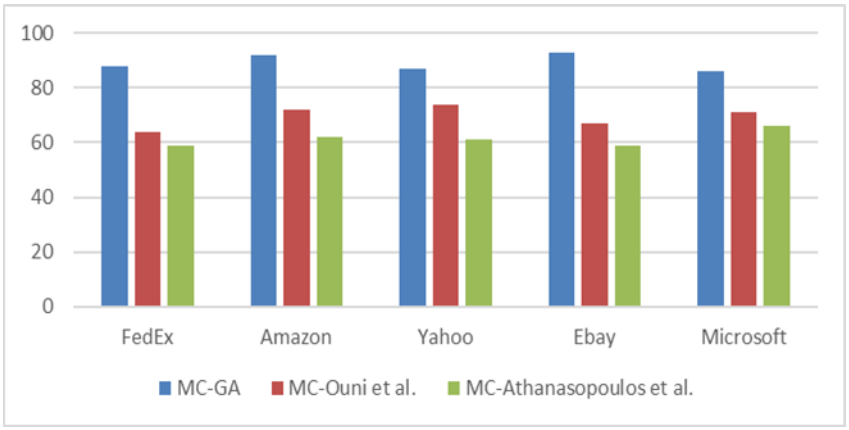


Figure 5.7: Median manual correctness (MC) value over 30 runs on all Web services using the different modularization techniques with a 95% confidence level ($\alpha = 5\%$).

restructuring of the interface since they spent a long time with manual changes to create port types and move operations. Thus, the developers liked the functionality of our tool that helps them to modify a port type based on the recommendations. 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 decompose multiple services into interfaces based on the dependency between them. Another possibly suggested improvement is to consider the users invocation data to restructure the interface.

In our evaluation, we considered measuring the time spent by the different developers to

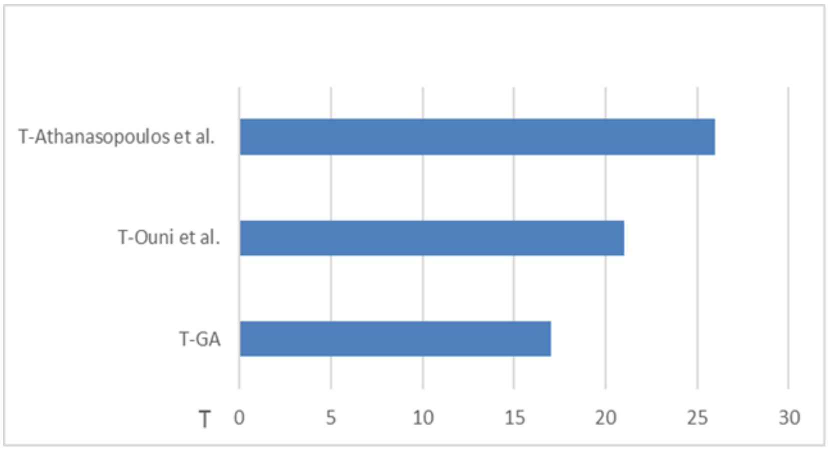


Figure 5.8: Median execution time (T), including user interaction, of the different tools over 30 runs on all Web services with a95% confidence level ($\alpha = 5\%$).

use our interactive tool and automated Web services modularization techniques [109][110]. We allowed the user to fix the solutions proposed by the automated tools to reach an acceptable design. Figure 5.8 shows the average results of the execution time of the different tools per Web service including the interaction time. The developers found that automated techniques generate solutions that require a lot of effort to inspect and manually adjust the proposed design. All developers expressed a high interest in the idea of the interactive tool that can incorporate their preferences by evaluating manually very few solutions. Furthermore, the execution time results confirm that few number of interactions are required with the user and that the generate solutions do not require a lot of changes to meet the developers preferences.

5.5 Conclusion

In this chapter, we presented a novel interactive search-based learning Web services modularization approach that does not require the definition of a fitness function. The user is asked to evaluate manually few modularization solutions suggested by a Genetic Algorithm (GA) for few iterations then these examples are used as a training set for the ANNs to evaluate the solutions of the GA in the next iterations. We evaluated our approach on a benchmark of Web services. We report the results on the efficiency and effectiveness of our approach, compared to existing approaches [109][110].

In future work, we are planning to investigate an empirical study to consider additional Web services and larger set of refactorings in our experiments. We are also planning to extend our approach to include the detection of refactoring opportunities in Web services using our interactive heuristic-based learning approach.

CHAPTER 6

Dimensionality Reduction for Multi-Objective Search-Based Software Refactoring

6.1 Introduction

Software engineering is by nature a search problem, where the goal is to find an optimal or near-optimal solution [31]. This search is often complex with several competing constraints, and conflicting functional and non-functional objectives. The situation can be worse since nowadays successful software are more complex, more critical and more dynamic leading to an increasing need to automate or semi-automate the search process of acceptable solutions for software engineers. As a result, an emerging software engineering area, called Search-Based Software Engineering (SBSE) [9], is rapidly growing.

SBSE is a software development practice that focuses on formulating software engineering problems as optimization problems and utilizing meta-heuristic techniques to discover and automate the search for near optimal solutions to those problems. The aim of SBSE research is to move software engineering problems from human-based search to machine-based search, using a variety of techniques from the fields of metaheuristic search, operations research and evolutionary computation paradigms. SBSE has proved to be a widely applicable and successful approach, with many applications right across the full spectrum of activities in software engineering, from initial requirements, project planning, and cost estimation to regression testing and onward evolution. There is also an increasing interest in search based optimization from the industrial sector, as illustrated by works on testing involving Daimler [122] and Microsoft [123], works on requirements analysis and optimization involving Motorola [124], Ericsson [125] and NASA [126], and works on refactoring involving Ford [29]. The increasing maturity of the field has led to a number of tools for SBSE applications.

Due to the subjective nature of software engineering problems, several conflicting preferences should be considered during the search for near optimal solutions. Thus, multi-objective evolutionary algorithms were widely applied to address several problems such as the generation of test cases, next release problems, software refactoring, model-driven engineering, etc [9]. The goal is to find a trade-off between different preferences of the developers such as coverage measures for testing, quality metrics for refactoring, implementation time and scheduled requirements for the next release problem, etc. The output is a set of diverse non-dominated solutions, called the Pareto front, that cover the possible preferences of the software engineers who select the final solution.

While the use of multi-objective evolutionary computation in software engineering show very promising results [9], several challenges could be discussed. First, the different fitness functions are defined and selected by the developers. Thus, it is challenging to decide up-front of the execution of the search if these functions are conflicting or not. In most cases, developers use their intuition and expertise to define the fitness functions. Without a rigorous check of the possible correlation between the defined fitness functions, a diverse set of solutions cannot be generated, if some conflicting measures are aggregated into one fitness function, or a large number of non-dominated solutions is generated, if non-conflicting measures are treated as separate fitness functions. Second, several recent studies considered as many objective as possible using many-objective algorithms [2]. however the visualization of the solutions in the generated Pareto front is a challenge for the developers due to the large number of Pareto front solutions when high number of objectives are considered. Third, the consideration of high number of objectives that are not necessarily conflicting make the execution time of the algorithm long and the search slow. Finally, the multi-objective algorithms behave similar to random search when the number of objectives increase thus reducing the number of objectives by studying their correlations may insure a better convergence of the algorithm.

In this chapter, we propose to address the above challenges by using the Principal Component Analysis (PCA) algorithm [127, 128] during the execution of a well-known multi-objective algorithm NSGA-II [27], adapted to address the software refactoring problem [11]. Refactoring is a critical task to improve the design structure of a system while preserving its behavior. To automate refactoring activities, new approaches have emerged where search-based techniques have been used [9]. These approaches cast the refactoring problem 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 conflicting quality metrics in a single fitness function to find the best sequence of refactorings [9]. Harman et al. [10] propose 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. Later, several work treated refactoring as a many-objective problem using between eight and fifteen objectives [73, 36, 74] where every objective corresponds to either a quality metric or quality attribute. The majority of these refactoring studies suffer from the challenges discussed above.

In this work, we start from the hypothesis that there may be correlations among any two or more objectives (e.g. quality metrics) that are used to evaluate refactoring solutions. Our approach, based on the PCA-NSGA-II methodology [129, 130], aims at finding the best and reduced set of objective that represents the quality metrics of interest to the domain expert. A regular NSGA-II algorithm with several objectives, including the quality metrics and attributes of the QMOOD model [3], is executed for a number of iterations then a PCA component analyzes the correlation between the different objectives using the execution traces. The number of objectives maybe reduced during the next iterations based on the PCA results. The process is repeated several times until a maximum number of iterations is reached.

We implemented our proposed approach and evaluated it on a set of seven open source systems. Statistical analysis of our experiments showed that our dimensionality reduction reduced significantly the number of objectives on several case studies by a minimum of 4 objectives and a maximum of 8 objectives. It also generates a smaller number of non-dominated solutions and lower execution time comparing to existing many-objective refactoring techniques [73, 36, 74]. The results show that the approach outperforms several of existing multi-objective refactoring techniques [..], where the objectives are not analyzed for possible correlations, based on several evaluation measures such as number of fixed anti-patterns and manual correctness.

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

- The paper introduces a novel dimensionality reduction technique to find the best minimum set of objectives needed to evaluate search based software engineering solutions. The developers needs to just specify several evaluation measures (proprieties of the desired solution) and our technique can automatically identify possible correlations and conflicts between them to find the best set of fitness functions.

- We propose a case study related to software refactoring. The goal is to find the best minimum set of quality metrics and attributes that maybe needed to converge towards good refactoring recommendations.

- The paper reports the results of an empirical study on an implementation of our approach and a comparison with the state of the art refactoring techniques [11]. The obtained results provide evidence to support the claim that our approach significantly reduce the number of needed objectives while generating a diverse set of good refactoring solutions.

6.2 A Dimensionality Reduction Approach for Search-Based Software Refactoring

In this section, we formulate the software refactoring problem as an optimization problem using the PCA-NSGA-II algorithm as described in [130]. Consequently, the section include a review of the NSGA-II algorithm, refactoring solution representation, and a review of the objective selection scheme for the PCA step.

6.2.1 Overview

The general structure of the proposed approach is described in Figure 6.1. The approach takes as inputs a set of quality metrics and attributes, several code refactoring types, and a software system to refactor. The first component consists of a regular execution of NSGA-II during a number of iterations. During this phase, NSGA-II will try to find the non-dominated solutions balancing the initial set containing all the objectives such as improving the quality attributes or metrics of the system, maximizing/preserving the semantic coherence of the design, minimizing the number of refactorings in the proposed solutions, etc.

After a number of iterations, the second component of the algorithm is executed to analyze the execution traces of the first component (solutions and their evaluations), using PCA [127, 128], to check the correlation between the different objectives. When a correlation between two or more objectives is detected, only one of them is selected for future iterations of the first component. Then, the first component is executed again with the new objective set.

The whole process of these two components continue until a maximum number of iterations is reached. A set of non-dominated refacotoring solutions are proposed to the developers with the reduced objectives set to select the best refactorings sequence based on his or her preferences.

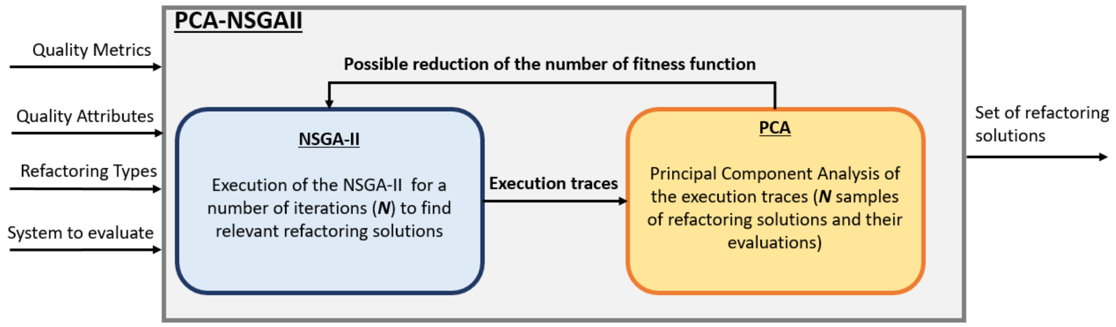

PCA-NSGAII
Quality Metrics
Quality Attributes
Refactoring Types
System to evaluate
Possible reduction of the number of fitness function
NSGA-II
Execution of the NSGA-II for a number of iterations (N) to find relevant refactoring solutions
Execution traces
PCA
Principal Component Analysis of the execution traces (N samples of refactoring solutions and their evaluations)
Set of refactoring solutions

Figure 6.1: Approach overview.

In the next sections, we describe in details the used algorithms and their adaptation.

6.2.2 Multi-Objective Dimensionality Reduction Algorithm

In this section, the two components of our approach are described. First, we describe the first component based on NSGA-II [27]. Then, we discuss how PCA is combined with NSGA-II in our methodology.

6.2.2.1 NSGA-II Overview

The non-dominated sorting genetic algorithm also known as NSGA-II [27], is a widely used multi-objective evolutionary algorithm in practice [42]. Its performance for solving software engineering problem is well-established comparing to several other algorithms [9]. Algorithm 3 gives a high-level view of the NSGA-II algorithm.

NSGA-II starts with a randomly generated initial parent population P_0 of individuals. Then, the crossover and mutation genetic operators are applied to this initial population to create offspring individuals Q_0. Both parent and offspring are merged into an initial population R_t ($t = 0$ at the first iteration). The resulting population R_t is used by the *fast-non-dominated-sort* of NSGA-II to classify individual solutions into different dominance level. To determine the dominance level of an individual solution x, this solution is compared to every other solution in R_0 until it is found dominated, or not. Based on the

Pareto optimality, a solution x dominates another solution y, if x is no worse than y in all objectives and is strictly better than y in at least one objective. In mathematical notation, given a set of objectives functions $f_i, i \in 1..n$ to minimize, x dominates y can be written as follows:

$$\forall i, f_i(y) \leq f_i(x) \qquad \text{and} \qquad \exists j | f_j(y) < f_j(x) \tag{6.1}$$

Upon sorting using the above dominance principle, the individuals in R_0 are assigned to groups of different level of dominance referred to as Pareto fronts. Solutions in the first Pareto front F_0 are assigned dominance level 0, those in the second Pareto front F_1 are assigned dominance level 1, and so on. Part of the good solutions are used in subsequent iterations based on the dominance levels. The next parent population P_{t+1} is formed by adding individuals from successive fronts, starting with front F_0, until the size of P_{t+1} is equal to N. If filling P_{t+1} require to select a subset of individual in the last available front F_L, such selection is based on the crowding distance of each individual solution within the same front F_L [27]. The crowding distance of a non-dominated solution measures the density of solutions surrounding it, and is used to promote diversity within the population. It is estimated by the size of the largest cuboid enclosing a solution in the Pareto front that does not contain any other solution. The front F_L to undergo partial selection is sorted into descending order with respect to the crowding distance, and the first $N - |P_{t+1}|$ elements are chosen. Then, a new offspring population Q_{t+1} is generated from P_{t+1} using, again, the crossover and mutation operators. This process is repeated until a stopping criteria is met.

6.2.3 NSGAII-PCA Combination

Large-dimensional data analysis encounters various challenges as well as some opportunities. With such dataset, in many cases, not all the measured parameters or variable are

ALGORITHM 3: NSGA-II: high-level view

```
Input: Population size $N$
$P_0$ =Create-initial-population($N$);
$Q_0$ =Generate-offsprings($P_0$);
$t = 0$;
repeat
    $R_t = P_t \cup Q_t$;
    $\mathbf{F}$ = fast-non-dominated-sort($R_t$);
    $P_{t+1} = \emptyset$;
    $i = 0$;
    repeat
        Apply crowding-distance-assignment($F_i$);
        $P_{t+1} = P_{t+1} \cup F_i$;
        $i = i + 1$;
    until $(|P_{t+1}| + |F_i| \leq N)$;
    Crowding-Distance-Sort($F_i$);
    $P_{t+1} = P_{t+1} \cup F_i\,[N - |P_{t+1}|]$;
    $Q_{t+1}$ =Generate-offsprings($P_{t+1}$);
until (stopping criteria is reached);
```

essential for describing the process of interest. In the case of many-objectives optimization problem, the increase in the dimensionality of the Pareto-optimal front create different problems as describe above. While many-objective optimization algorithms exist [53, 54] to address the increase in number of objectives, it is important to reduce the dimensions of the original data prior to any modeling or decision making. One such situation is when hidden redundancies exist naturally among the decision parameters. In this section, we are focusing on addressing dimensionality reduction in the objectives space.

Objective space dimensionality reduction approaches assume that given a multi-objective problem with M objectives, there is a subset of the objectives that are correlated. To the best of our knowledge, very few methodologies have been developed for multi-objective evolutionary algorithms towards the reduction of the number of objectives [131, 129, 44, 132, 133].

In [131], Brockhoff and Zitzler proposed a dimensonality reduction method around the Minimum Objective Subset problem (MOSS). The goal is to find the smallest subset of objectives included in the original feasible region $\mathbf{X}$, such that the weak Pareto dominance structure is left unchanged. In [134], they proposed two variants of the MOSS methodology referred to as $\delta-$ MOSS and $k-$ MOSS. Both of the two variant introduce a consideration for error. The the original methodology focused on the definition of the problem, while $\delta-$ MOSS and $k-$ MOSS address practical questions. In particular, they offered two algorithms to implement the MOSS problem: (1) an exact computation of the subset, and an approximate algorithm. While the proposed solution was shown to perform well for very large number of objective against the original PCA methodologies, it is an off-line technique used for decision making after the MOEA have been executed.

Saxena et al. proposed two dimensionality reduction methodologies based on Principal Component Analysis (PCA). Their methodology considers both linear and nonlinear solutions [129, 30]. Procedures were proposed to identify from the whole population the significant principal components and then to reduce the number of objectives. The authors demonstrated that the methodology have some vulnerabilities in finding Pareto-optimal front in a 10-objective problem. In [130], a more robust objectives selection approach was proposed to improve the performance of both non-linear and linear dimensionality reduction. Not only these methodologies can be utilized before and after execution of the MOEA, but the computation of the PCA is straightforward for the multi-objective optimization problem. In this chapter, we apply the linear PCA dimensionality reduction technique to the multi/many-objective software refactoring problem using NSGA-II. In the remainder of this chapter, PCA refers to linear PCA unless specified otherwise.

Principal component Analysis (PCA) is established as the best linear dimensionality reduction techniques, in the mean-square error sense [127]. In various field, it is referred to as the Singular Value Decomposition (SVD) and the Hotelling transform [128]. PCA is a statistical analysis technique used in multi-variate analysis, and seeks to reduce the dimensionality of a given dataset when there is a large number of statistically correlated variables. In essence, PCA finds a few ordered orthogonal linear combinations referred to as the principal components (PCs), which retain the largest variance in the data. The first PC is the linear combination with the largest variance, and so on.

Given equation 2.1, PCA is posed as an eigenvalue-eigenvector problem. The data is recorded over a population of individuals of size N generated and used in the NSGA-II algorithm. This data consists of measurement of all the objective function used in the NSGA-II, and represented as a matrix $\mathbf{F} = (\mathbf{f}_1, \mathbf{f}_2, \ldots, \mathbf{f}_M)^T$. A column $\mathbf{f}_i = [f_{i1}, f_{i2}, \ldots, f_{iN}]^T$ is the vector representing values for the $i-$th objective over the N individuals, and each entry f_{ij} of $\mathbf{f}_i$ is the value of the i-th objective for the j-th individual in the population. In this notation, $(^T)$ is the matrix transpose operator, and M is the number of objectives.

PCA is performed using the correlation or covariance matrix of the standardized dataset $\mathbf{X} = (\mathbf{x_1}, \mathbf{x_2}, \ldots, \mathbf{x_M})^T$. This means each entry $x_{ij} = (f_{ij} - \mu_i)/\sigma_i$, where μ_i and σ_i are the sample mean and standard deviation of $\mathbf{f}_i$, respectively. Consequently, every row of $\mathbf{X}$ centered at zero, and has unit standard deviation.

In the PCA-NSGA-II procedure, the original dataset is not projected onto the principal component. Rather, after the eigenvalue-eigenvector decomposition of the $M \times M$correlation matrix, the essential objective are selected through the analysis of eigenvalues, eigenvectors, and the correlation among objectives. The choice of the correlation matrix is suitable for dataset were the different variable do not have the same scales. The correlation matrix is given by equation 6.2, and algorithm 4 gives an high-level view of the objective reduction procedure.

$$\mathbf{R} = \frac{1}{M}\mathbf{X}\mathbf{X}^T \tag{6.2}$$

First, the eigenvectors $\mathbf{V} = (\mathbf{V}_1, \mathbf{V}_2, \ldots, \mathbf{V}_M)$ and eigenvalues $\mathbf{\Lambda} = [\lambda_1, \lambda_2, \ldots, \lambda_M]$ such that $\lambda_1 \leq \lambda_2 \leq \ldots \leq \lambda_M$. Then, the contribution of each eigenvalue to the overall variance, $e_i = \lambda_i / \sum_{j=1}^{M} \lambda_j$, and the contribution of the i-th objective f_i to all the eigenvectors, $c_i^M = \sum_{j=1}^{M} e_j f_{ij}^2$, are computed. After the contributions of eigenvalues are calculated, the essential (conflicting) objectives are selected as follows:

1. *Find the number N_v of significant eigenvectors.* It is the number of eigenvectors such that the sum of eigenvalue contribution is within a threshold cut T_c - that is $\sum_{j=1}^{N_v} e_j \geq T_c$. Note that the e_js are chosen in descending order of their values, starting with the eigenvalue with the largest contribution.

2. *Find the conflicting objectives for each significant eigenvector.* Given a significant eigenvector $\mathbf{V}_j = [f_{j1}, f_{j2}, \ldots, f_{jM}]^T$, f_{jm} represents the contribution of the m-th objective f_m. Then the objective set of is partitioned into positive set $\mathcal{F}^+ = \{f_m | f_{jm} \geq 0\}$ and negative set $\mathcal{F}^- = \{f_m | f_{jm} < 0\}$. finally, f_p and f_n being the objectives with the highest magnitudes in $\mathcal{F}^+$ and $\mathcal{F}^-$, respectively, the selection of conflicting objective is as follows:

   - if $|f_p| \geq |f_n|$, select f_p and all the objectives in $\mathcal{F}^-$.
   - if $|f_p| < |f_n|$, select f_n and all the objectives in $\mathcal{F}^+$.
   - if $\mathcal{F}^+ = \emptyset$ or $\mathcal{F}^- = \emptyset$, then the two objectives with the highest magnitude are selected.

After eigenanalysis, the set of objectives retained is denoted $\mathcal{F}_e$, the initial set of objective being $\mathcal{F}_o$. The next step is the reduce correlation analysis, which is the analysis of correlation between objectives in $\mathcal{F}_e$. This is intended to eliminate objectives that are strongly correlated among each other. The reduce correlation matrix (RCM) is the same as $\mathbf{R}$, except the rows and columns corresponding to objectives in $\mathcal{F}_o \setminus \mathcal{F}_e$ are removed. For each objective f_i in $\mathcal{F}_e$, an objective f_j in $\mathcal{F}_e$ is identically correlated with it if it satisfies Equation 6.3.

$$\text{sign}(R_{ik}) = \text{sign}(R_{jk}), k = 1, 2, \ldots, M \qquad \text{and} \qquad R_{ij} \geq T_{corr} \tag{6.3}$$

where T_{corr} is the threshold cut that determines significant correlation between two objectives. For each objective in $\mathcal{F}_e$, the identically correlated subset is calculated as $\mathcal{S}_i =$

$\{f_j \in \mathcal{F}_e | f_j$ satisfies Equation 6.3$\}$. Within an identically correlated subset $\mathcal{S}_i$, a selection score is assigned to each objective as $c_i = \sum_{j=1}^{N_v} e_j |f_{ij}|$, and the objective with the highest score is selected. After the RCM analysis, a smaller subset of objective is retained. The set of such objective is denoted $\mathcal{F}_s$.

ALGORITHM 4: Objective Reduction High-level view

```
Input: $\mathbf{V}, \mathbf{\Lambda}, N_v$
Output: $\mathcal{F}_s$
$\mathcal{F}^+ = \emptyset, \mathcal{F}^- = \emptyset, \mathcal{F}_e = \emptyset$;
foreach $V_i, 1 \leq i \leq N_v$ do
    partition-objectives$(\mathcal{F}^+, \mathcal{F}^-)$;
    set-selected-objectives$(\mathcal{F}_e, \mathcal{F}^+, \mathcal{F}^-)$;
end
$\mathcal{F}_s = \emptyset$;
foreach $f_i \in \mathcal{F}_e$ do
    $\mathcal{S}_i$ = compute-identically-correlated-subset$(f_i)$;
    set-selected-objectives$(\mathcal{F}_s, \mathcal{S}_i)$;
end
```

6.2.4 Solution Approach

The previous subsections described a generic operation of the NSGA-II and the PCA procedures used in the PCA-NSGA-II algorithm. In order to use the algorithm on the specific case of software refactoring, it is necessary to defined the following entities upon which the algorithm will operate:

1. Solution representation
2. Fitness functions
3. Genetic operators and solution generation

6.2.4.1 Refactoring Solution Representation

During execution of the NSGA-II, a population consists of a set of N individuals (or refactoring solutions). Each refactoring solution is a sequence of refactoring operations (ROs).

A refactoring operation consists of the actual operation to apply to the system under maintenance, and a set of controlling parameters randomly selected (e.g., actors, roles, etc.), as illustrated in table. We use a vector representation, such that a solution $\mathbf{x}$ can be written as $(RO_0, RO_1, \ldots, RO_{L-1})$, where L is the maximum number of refactoring operations in a solution, and RO_l is the l-th refactoring operation. The order in which the ROs appear in the vector is the same as the order in which they are applied to the software system under refactoring - that is RO_1 is applied first, then RO_2, and so on. The initial population is generated by randomly choosing refactoring operations for each individual in the solution. In addition to the refactoring solution, a set of pre- and post-conditions is specified to ensure the feasibility of the operation. This is in agreement with the work of [15], which defined methods for formalizing preconditions that must be met before a refactoring operation can be applied in order to preserve the behavior of the system. Finally, each refactoring operation involves actors - that is the code elements that are involved or impacted by the operation (e.g., Class, package, method, etc.). In our experiments, we considered the following types of refactorings: Extract class, Extract interface, Inline class, Move field, Move method, Push down field, Push down method, Pull up field, Pull up method, Move class and Extract method.

6.2.4.2 Change Operators

In this work, we use the single-point crossover. It consists in selecting and splitting two parent solutions at a random locations in their respective vector of the refactoring solution. Then, two child solutions are produced from the two parent. For the first child, the first part of the first parent is replaced by the second part of the second parent, and vice versa. Once each child is produced, the constraint on the maximum number of refactoring operation is enforced by randomly removing some refactoring operations.

For the mutation operator, we use the bit-string method. It consists in randomly picking one or more refactoring operators from the individual under mutation, and replacing them with new refactoring operators from the initial list of operation. The process of selection is driven by a probability.

6.2.4.3 Objective functions and solution evaluation

Solution evaluation constitutes a major step in every multi-objective optimization process. Consequently, in our proposed methods, every application of a refactoring sequence is followed by the evaluation of the resulting software system. In this work, we used three categories of objectives:

Quality objectives:

In a first scenario of our experiments, we used the 6 quality attributes that are defined by the QMOOD quality model [3]. QMOOD quality measures are defined using a weighted sum of high-level software metrics detailed in Table **??**. The 6 external quality attributes used are given in Table **??**, and constitutes six objective functions for the NSGA-II step.

In a second scenario of our experiments, we used the 11 quality metrics that compose the 6 quality attributes as separate objectives. The list of these metrics are described in Table **??**.

For both scenarios, we used the following two additional objectives as well:

Number of recommended refactorings objective

This objective, to minimize, corresponds basically to the size of the solution.

Design coherence objective

It aims at approximating the design preservation after applying the suggested refactorings. We used the function that we defined in our previous work [29] to ensure that the refactoring operation preserves design coherence. The vocabulary could be used as an indicator of the semantic/textual similarity between different actors that are involved when performing a refactoring operation. We start from the assumption that the vocabulary of an actor is borrowed from the domain terminology and therefore can be used to determine which part of the domain semantics an actor encodes. Thus, two actors are likely to be

Table 6.1: QMOOD metrics description [3]

Design Metrics	Design Property	Description
Design Size in Class (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.

semantically similar if they use similar vocabularies.

The vocabulary can be extracted from the names of methods, fields, variables, parameters, types, etc. Tokenisation is performed using the Camel Case Splitter [135], which is one of the most used techniques in Software Maintenance tools for the preprocessing of identifiers. A more pertinent vocabulary can also be extracted from comments, commit information, and documentation. We calculate the textual similarity between actors using an information retrieval-based technique, namely cosine similarity, as shown in Equation 6.4.

Table 6.2: QMOOD quality attributes [3]

Quality Attribute	Definition *Computation Equation*
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. $0.12 * Cohesion + 0.22 * Polymorphism +$ $0.22 * Messaging + 0.22 * DesignSize + 0.22 * Hierarchies$
Extendability	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. $0.2 * Abstraction + 0.2 * Encapsulation + 0.2 * Composition +$ $0.2 * Inheritance + 0.2 * Polymorphism$

Each actor is represented as an n-dimensional vector, where each dimension corresponds to a vocabulary term. The cosine of the angle between two vectors is considered as an indicator of similarity. Using cosine similarity, the conceptual similarity between two actors c_1 and c_2 is determined as follows:

$$Sim(c_1, c_2) = Cos(\vec{c_1}, \vec{c_2}) = \frac{\vec{c_1} \cdot \vec{c_2}}{\|\vec{c_1}\| \times \|\vec{c_2}\|} = \frac{\sum_{i=1}^{n} w_{i,1} \times w_{i,2}}{\sqrt{\sum_{i=1}^{n} w_{i,1}^2} \times \sqrt{\sum_{i=1}^{n} w_{i,2}^2}} \tag{6.4}$$

where $\vec{c_1} = (w_{1,1}, ..., w_{n,1})$ is the term vector corresponding to actor c_1 and $\vec{c_2} = (w_{1,2}, ..., w_{n,2})$ is the term vector corresponding to c_2. The weights wi, j can be computed using information retrieval based techniques such as the Term Frequency – Inverse Term Frequency (TF-

IDF) method. We used a method similar to that described in [**?**] to determine the vocabulary and represent the actors as term vectors.

The next section describes the evaluation of our approach based on several open source systems.

6.3 Validation

To evaluate the ability of our dimensionality reduction refactoring framework to generate good solutions and find the minimum set of required objectives, we validated the proposed approach based on six open source systems and one industrial project provided by a partner from health-care industry based on a funded project. The obtained results were statistically analyzed, based on 30 runs for every system, when compared with existing refactoring approaches. In this section, we first present our research questions and then describe and discuss the obtained results.

6.3.1 Research questions

In our study, we assess the performance of our refactoring approach by determining whether it can generate meaningful sequences of refactorings that fix design defects while minimizing the number of code changes, preserving the semantics of the design, and reusing, as much as possible a base of recorded refactoring operations applied in the past in similar contexts. Our study aims at addressing the research questions outlined below.

The first four research questions evaluate the ability of our proposal to find a compromise between the four considered objectives that can lead to good refactoring recommendation solutions.

- **RQ1.**: To what extent can the proposed dimensionality reduction approach recommends useful refactorings?

- **RQ2.**: To what extent does the proposed dimensionality reduction approach reduce the number of objectives, execution time and number of non-dominated solutions while recommending useful refactorings?

- **RQ3.**: How does the proposed dimensionality reduction approach perform compared to other existing search-based refactoring approaches, including mono-, multi- and many-objective studies and an existing technique not based on computational search?

To answer **RQ1.**, we considered both automatic and manual validations to evaluate the usefulness of the proposed refactorings. For the automatic validation we compared the proposed refactorings with the expected ones. The expected refactorings are suggested by developers to fix existing design defects as detailed later.

$$RE_{recall} = \frac{|\ \textit{suggested refactorings} \cap \textit{expected refactorings}\ |}{|\ \textit{expected refactorings}\ |} \in [0,1] \tag{6.5}$$

$$RE_{precision} = \frac{|\ \textit{suggested refactorings} \cap \textit{expected refactorings}\ |}{|\ \textit{suggested refactorings}\ |} \in [0,1] \tag{6.6}$$

For the manual validation, we asked groups of potential users of our tool to manually evaluate 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*) that corresponds to the number of meaningful refactorings divided by the total number of suggested refactorings. MC is given by the following equation:

$$MC_{manualcorrectness} = \frac{|\ \textit{relevant refactorings}\ |}{|\ \textit{suggested refactorings}\ |} \in [0,1] \tag{6.7}$$

To answer **RQ2**, we compared the number of objectives (*NOB*) after the execution of the NSGAII-PCA algorithm. We have also compared the execution time (*T*)and the number of non-dominated solutions (*NS*) to existing mono/multi/many-objective refactoring approaches [...].

To answer **RQ3**, we compared our approach to other existing search-based refactoring approaches: (*i*) Kessentini et al. [16], O'Keeffe et al. [68], Mkaouer et al. [73] and (*ii*) Harman et al. [10]. O'Keeffe and Ó Cinnéide [68] formulate software refactoring as a mono-objective search problem where the main goal is to optimize the different QMOOD attributes aggregated into one fitness function using Simulated Annealing [40]. Kessentini et al. [16] also formulated refactoring suggestion as a single objective problem to reduce as much as possible the number design defects, while Harman et al. formulated refactoring recommendation as multi-objective to find a trade-off between two quality metrics, CBO (coupling between objects) and SDMPC (standard deviation of methods per class). Moreover, we compared the performance of our approach to an existing refactoring work of Ouni et al.[29], a multi-objective approach based on the NSGA-II algorithm, that finds trade-offs between quality improvements, design semantics and effort. In addition, we compared our approach with a many-objective refactoring technique based on NSGA-III proposed by Mkaouer et al. [73] with 8 objectives of quality attributes. Finally, we compared our refactoring results with a popular design defects detection and correction tool JDeodorant [66] that does not use heuristic search techniques in terms of precision, recall and manual correctness. 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.

6.3.2 Experimental setup

6.3.2.1 Subjects

Our study involved 12 participants from the University of Michigan and 5 software developers from our industrial partner. The graduate students include 6 master students in Software Engineering and 6 PhD. students in Software Engineering. All the participants are volunteers and familiar with Java development and refactoring. The experience of these participants on Java programming ranged from 3 to 17 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 three years as software developer and most of them (10 out of 12 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 12 students (the selected master students) are working as full-time or part-time developers in software industry.

6.3.2.2 Studied projects

We applied our approach to a set of six well-known and well-commented industrial open source Java projects: Xerces-J[1], JFreeChart[2], GanttProject[3], Apache Ant[4], JHotDraw[5], and Rhino[6]. Xerces-J is a family of software packages for parsing XML. JFreeChart is a powerful and flexible Java library for generating charts. GanttProject is a cross-platform tool for project scheduling. Apache Ant is a build tool and library specifically conceived for Java applications. JHotDraw is a GUI framework for drawing editors. Finally, Rhino is a JavaScript interpreter and compiler written in Java and developed for the Mozilla/Firefox browser. We selected these 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.

Our industrial partner from the health-care industry has a legacy project developed over 11 years. The system is written in JAVA and its main feature is image reconstruction. Based on a funded project, we collaborated with our partner to provide them efficient refactoring

[1]http://xerces.apache.org/xerces-j/
[2]http://www.jfree.org/jfreechart/
[3]www.ganttproject.biz
[4]http://ant.apache.org/
[5]http://www.jhotdraw.org/
[6]http://www.mozilla.org/rhino/

Table 6.3: Statistical data of the two evaluated programs

Systems	Release	# classes	# design defects	KLOC	# refactorings
Xerces-J	v2.7.0	991	91	240	80
JFreeChart	v1.0.9	521	72	170	96
GanttProject	v1.10.2	245	49	41	63
Apache Ant	v1.8.2	1191	112	255	74
JHotDraw	v6.1	585	25	21	36
Rhino	v1.7R1	305	69	42	50
Industrial P	v6.1	958	146	271	103

recommendations to fix most of their maintainability issues, improve the extendability of their software and provide useful assistance to their developers when updating the code. Five developers from our partner participated in the experiments of the proposed technique in this chapter.

Table 6.3 provides some descriptive statistics about these seven programs.

6.3.2.3 Scenarios

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 (if any), their programming experience, and their familiarity with software refactoring. In addition, all the participants attended one lecture about software refactoring, and passed ten tests to evaluate their performance in evaluating and suggesting refactoring solutions.

We formed 3 groups. Each of the first two groups is composed of three masters students and three PhD. students. The third group is composed of five software developers from our industrial partner, since they agreed to participate only in the evaluation of their software system. The two first 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. Consequently, each group of participants who agreed to participate in the study 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. Since the application of refactoring solutions is a subjective process, it is normal that not all the developers have the same opinion. In our case, we considered the majority of votes to determine if suggested solutions are correct or not. Each participant evaluates different refactoring solutions for the different techniques and systems.

In the first scenario, we selected a total of 70 classes from all the systems that include design defects (10 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 first scenario, we calculated the differences between the recommended refactorings and the expected ones (manually suggested by the developers).

In the second scenario, we asked the developers to manually evaluate the recommended solutions by our algorithm and existing techniques. Our experiments were not limited to only comparisons with expected refactorings. 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). All the recommended refactorings are executed using the Eclipse platform.

6.3.2.4 Parameters setting

In our experiments, we use and compare different mono, multi- and many-objective algorithms. For each algorithm, to generate an initial population, we start by defining the maximum vector length (maximum number of operations per solution). The vector length is proportional to the number of refactorings that are considered, the size of the program to be refactored, and the number of detected design defects. This parameter can be specified by the user or derived randomly from the sizes of the program. For all search algorithms, we fixed the maximum vector length to 200 refactorings, and the population size to 100 individuals (refactoring solutions), and the maximum number of iterations to 10,000 iterations.

Another element that should be considered when comparing the results of the four algorithms is that multi/many-objective algorithms do not produce a single solution like a mono-objective technique, but a set of optimal solutions (non-dominated solutions). The maintenance engineer can choose a solution from them depending on their preferences in terms of compromise. However, at least for our evaluation, we need to select only one solution. Thereafter, and in order to fully automate our approach, we proposed to extract and suggest only one best solution from the returned set of solutions. In our case, we selected the closest solution to the knee-point in terms of Euclidean distance [..].

6.3.2.5 Inferential Statistical Test Methods Used

Our approach is stochastic by nature, i.e., two different executions of the same algorithm with the same parameters on the same systems generally leads to different sets of suggested refactorings. For this reason, our experimental study is performed based on 30 independent simulation runs for each problem instance, and the obtained results are statistically analyzed by using the Wilcoxon rank sum test with a 95% confidence level ($\alpha = 0.05$). The Wilcoxon signed-rank test is a non-parametric statistical hypothesis test used when comparing two related samples to verify whether their population mean-ranks differ or not. In this way, we could decide whether the difference in performance between our approach and the other detection algorithms is statistically significant or just a random result.

6.3.3 Empirical study results and discussions

Results for RQ1: As described in the approach adaptation section, Two different adaptations of the dimensionality reduction approach were considered. The first adaptation used as initial fitness functions the different QMOOD attributes along with the number of refactorings and design semantic objectives. The second adaptation is based on the use of the different quality metrics (not the 6 attributes) of QMOOD as fitness functions in addition to the two other objectives of effort and semantics. Figures 2, 3 and 4 summarize the results for the first research question using the manual correctness (MC), precision (PR) and recall metrics (RE). Figure 2 shows that both of the dimensionality reduction approaches generate good refactoring solutions that are relevant to programmers. On average, for all of our ten studied projects, 84% 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 88% for the Rhino project and the lowest score is 79% for Apache-Ant. Thus, the results are clearly 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. The first dimensionality reduction formulation based on QMOOD attributes outperformed the second one based on the quality metrics for almost all the systems. One of the main reasons could be that the first formulation based on QMOOD already aggregate some of the correlated quality metrics (fitness functions) into attributes.

Since the MC metric is limited to the evaluation of the correctness and not the relevance of the recommended refactorings, we also compared the proposed operations with some expected ones defined manually by the different groups for several code fragments extracted from the different systems. Most of these classes represent some severe code smells detected using the detection rules defined in our previous work [...]. Figures 3 and 4 summarize the obtained results of precision and recall.

We found that a considerable number of proposed refactorings, with an average of

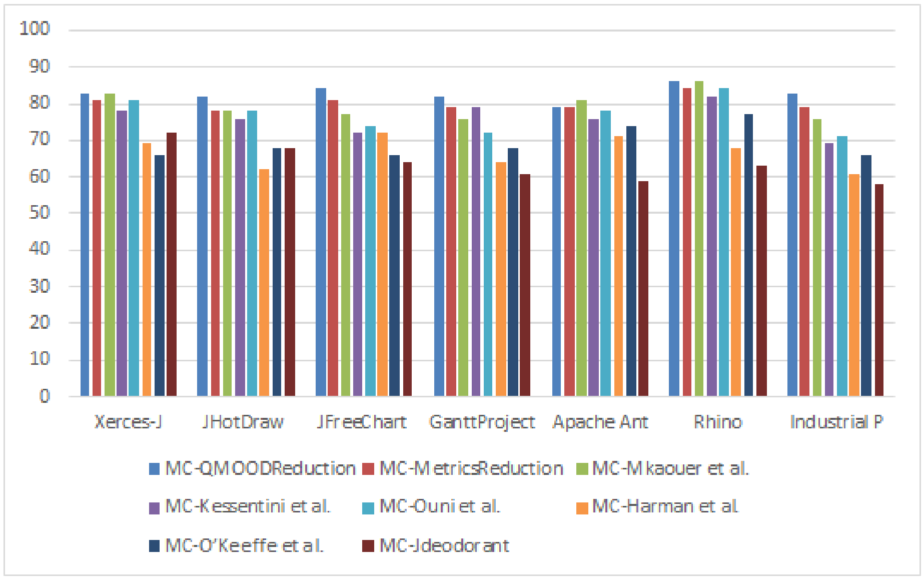


Figure 6.2: Median manual correctness (MC) value over 30 runs on all the systems using the different refactoring techniques with a 95% confidence level ($\alpha < 5\%$).

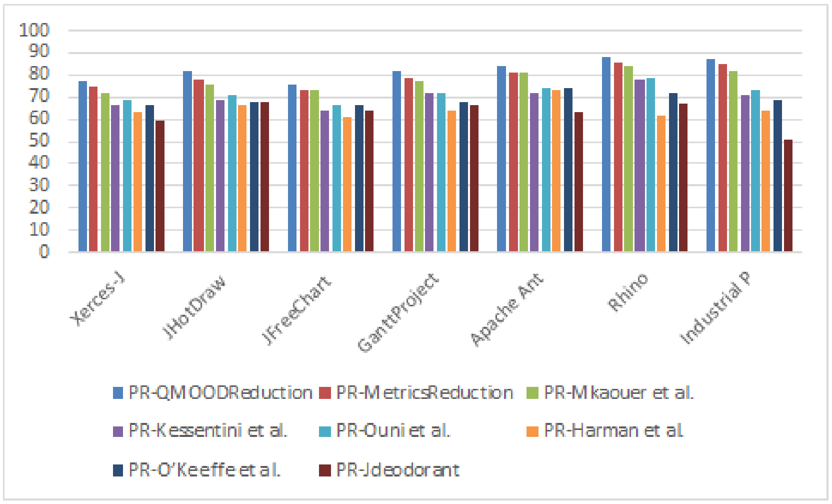


Figure 6.3: Median precision (PR) value over 30 runs on all the systems using the different refactoring techniques with a 95% confidence level ($\alpha < 5\%$).

more than 79% 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 and this

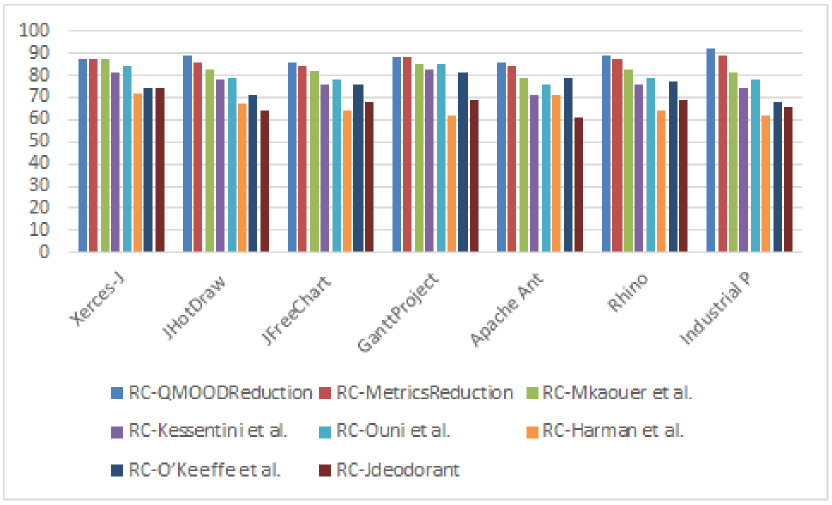


Figure 6.4: Median recall (RC) value over 30 runs on all the systems using the different refactoring techniques with a 95% confidence level ($\alpha < 5\%$).

was confirmed by the qualitative evaluation (MC). 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. Similar to the observations of the manual correctness results, the QMOOD dimensionality reduction formulation has the best results. It is clear that higher number of objectives may require higher number of iterations to converge towards good refactoring solutions. In addition, some non-relevant fitness functions may introduce noise to the search process.

To summarize and answer RQ1, the experimentation results confirm that our dimensionality reduction approach successfully identified relevant refactorings.

Results for RQ2: Figure 5 shows that our approach significantly reduced the number of objectives for both formulations when executed on all the systems. For the first QMOOD adaptation, the number of objectives were reduced to only four that corresponds to the number of changes, semantics preservation, reusability and understandability. The reduced objectives may show the importance of coupling and cohesion when identifying refactoring recommendations since both reusability and understandability are based on these

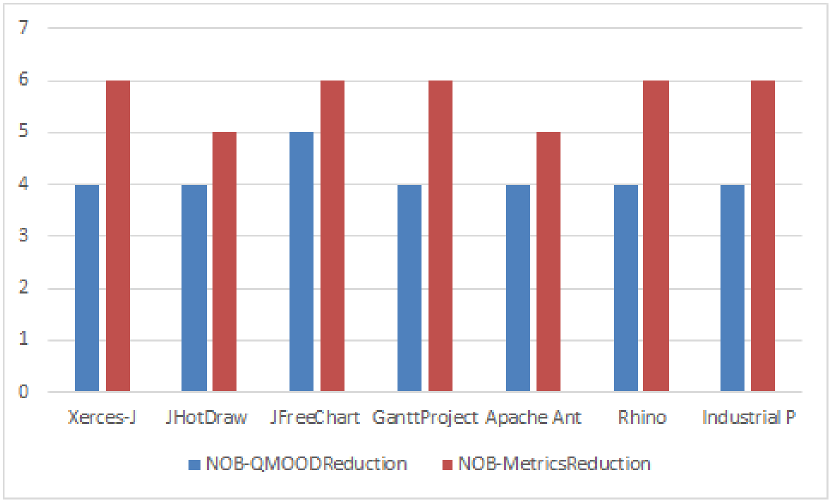


Figure 6.5: Median number of objectives (NOB) value over 30 runs on all the systems.

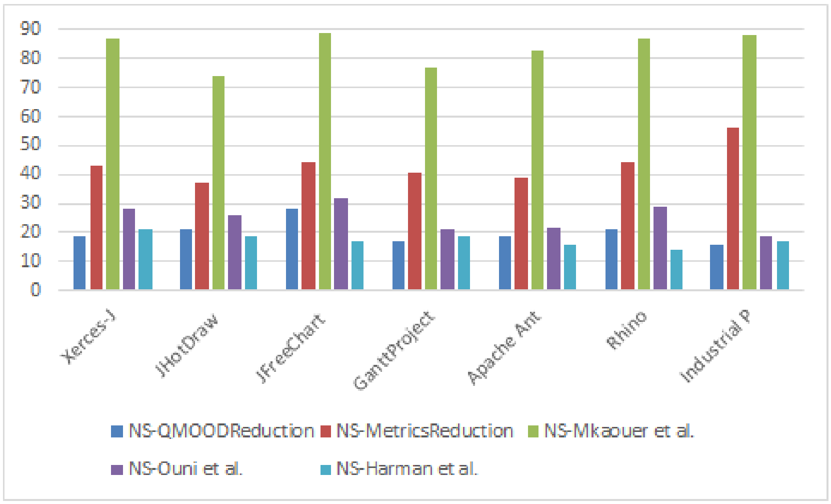


Figure 6.6: Median number of non-dominated solutions (NS) over 30 runs on all the systems using the different refactoring techniques with a 95% confidence level ($\alpha < 5\%$).

metrics. This outcome is also confirmed by the second formulation that selected these metrics among the ones that represented the last set of objectives. Along with the number of changes and design semantic fitness functions, four metrics were selected that number of methods, coupling, cohesion and the direct access metric.

Figure 6 shows that the number of objectives is correlated with the number of non-dominated solutions. In fact, the existing work of Harman et al. has the lowest number of

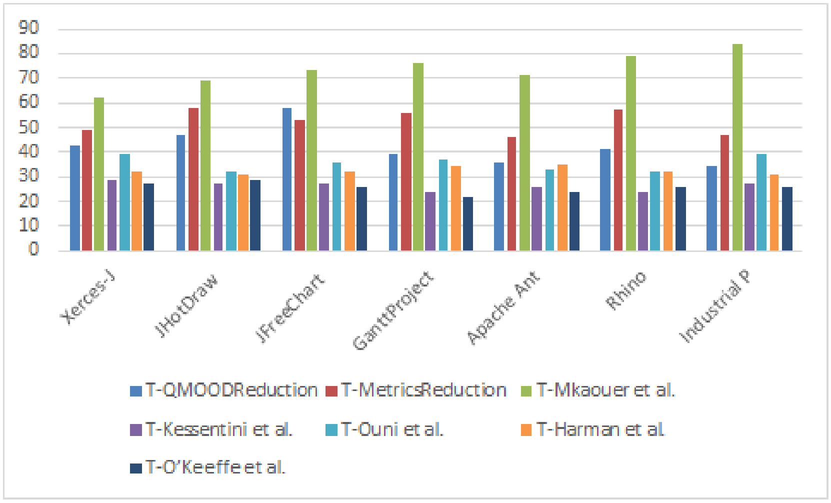


Figure 6.7: Median execution time (T) over 30 runs on all the systems using the different refactoring techniques with a 95% confidence level ($\alpha < 5\%$).

objectives (limited to two) and generated the lowest number of non-dominated solutions on the different systems. The many-objective algorithm of Mkaouer et al. based on NSGA-III has the highest number of non-dominated solutions due to the high number of objectives considered in the search process (8 objectives). It is clear that reducing the number of objectives based on both NSGAII-PCA formulations significantly reduced the number of non-dominated refactoring solutions.

Figure 7 reports the execution time for each of the search algorithms considered in our experiments. As shown in the figure, the execution time of the mono-objective approach, kessentini et al., has the lowest execution time and the highest one is the NSGA-III algorithm. The execution time of the NSGAII-PCA algorithm is slightly higher than the regular NSGA-II and other multi-objective approaches. It is expected that the execution time of the remaining mono-objective approach is almost half the multi-objective ones due to the following reasons: (1) they just considered one objective function and (2) the dimensionality reduction mechanism requires additional time processing, filtering and comparing the identifiers within classes. Since our refactoring problem is not a real time one, the execution time of NSGAII-PCA is considered acceptable by all the programmers of our experiments.

To conclude, the proposed NSGAII-PCA formulation successfully reduced the number of objectives, execution time and the number of non-dominated solutions while generating useful refactoring recommendations.

Results for RQ3: To answer RQ3, we compared our approach to other existing search-based refactoring approaches: (*i*) Kessentini et al. [16], O'Keeffe et al. [68], Mkaouer et al. [73] and (*ii*) Harman et al. [10]. Furthermore, the proposed dimensionality reduction approach is compared to JDeodorant which is a tool not based on heuristic search.

Figure 2, 3 and 4 summarizes the obtained results in terms of precision, recall and manual correctness. It is clear that the mono-objective algorithms (Harman et al., OKeeffe et al. and Kessentini et al.) has lower performance comparing to existing multi-objective and many-objective approaches based on the different evaluation metrics. The two dimensionality reduction adaptations have better manual correctness and precision comparing to all existing approaches. Fig. 2 shows that our approach provides significantly higher manual correctness results (MC) than all other approaches having MC scores respectively between 55% and 84%, on average as MC scores on the different systems. The same observation is valid for the precision and recall as described in Fig. 3 and 4. The recall of the many-objective refactoring technique is better or similar to the dimensionality reduction approaches. This can be explained by the fact that a large number of objectives, such as quality metrics, may generate high number of refactoring recommendations to optimize these objectives.

It is clear that our proposal outperforms also JDeodorant, on average, for all the systems in terms of manual correctness, precision and recall. The comparison is performed based on the types of refactoring supported by JDeodorant. The superiority of our approach over JDeodorant can be explained by the fact that JDeodorant uses only structural metrics to evaluate the impact of suggested refactorings on the detected code smells. However, one of the advantages of JDeodorant is that the suggested refactorings are easier to apply than those proposed by our technique as it provides an Eclipse plugin to suggest and then automatically apply a total of 4 types of refactorings, while the current version of our tool requires to apply refactorings by the developers using the Eclipse IDE with more complex types of refactorings.

To summarize, the proposed PCA-NSGAII formulation outperforms, in average, several of existing refactoring techniques in terms of generating useful refactoring recommendations.

6.4 Conclusion

In this chapter, we proposed a dimensionality reduction approach for multi-objective software refactoring that adjusts the number of considered objectives during the search for near optimal solutions. In most existing automated refactoring techniques, objectives are selected in advance by the software maintenance engineer without knowledge of the outcome of the refactoring. This process is, in general a trial and error approach, and may cause multiple execution of the refactoring tool before reaching the right combination of objectives. In this chapter, the proposed approach allows the user to select as many objectives as possible, and rely on the power of principal component analysis to reduce the number of objectives. Thus, at the last stage, the software engineer is left with objectives that contribute the most to the differentiation of solution, while other correlated objectives are removed during the intermediate optimization steps. To our knowledge, this is the first application of dimensionality reduction to the problem of multi-objective search-based software refactoring.

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 technique successfully reduced the initial set of large number of objectives. The results also show that our approach outperforms several of existing multi-objective refactoring techniques, where the objectives are not analyzed for possible correlations, based on several evaluation measures such as execution time, number of fixed antipatterns and manual correctness.

An interesting direction to further this approach in the future is to consider alternative dimensionality reduction approach. In the current approach, we used linear PCA to perform the analysis of correlation between the objective. Linear PCA consider the second order statistics, and does not include third order and above. Using non-linear PCA such as maximum variant unfolding, and/or spectral decomposition can improve the reduction of the number of objectives by include more statistical information. In addition, we are planning to validate our technique with additional objectives in order to conclude about the general applicability of our methodology. Furthermore, we are planning to adapt our dimensionality reduction approach to others software engineering problems such as test cases generation, next release problem, etc. Another future research direction related to our work is to integrate the developers in the loop when reducing the number of objectives to either select which one to eliminate or revise the fitness function formulation (aggregating the objectives).

CHAPTER 7

Conclusion and Future Work

7.1 Conclusion

Challenges in software refactoring are still fueling researches in the field in order to provide software organizations with realistic refactoring frameworks. Such frameworks are required to take into account the specificity of each organizations in terms of software engineering standards. In addition, successful refactoring tools are still required to go below the wide organization level in order to consider the reality of individual programmers. This last requirement is especially important due to the difficulty of applying system-wide refactoring and the need to increase the confidence of programmers in using proposed refactoring tools. Thus, the main goal of this thesis was to develop techniques to allow the integration of user preferences in search-based software refactoring. In doing so, we aimed at finding refactoring solutions that agree with the preferences of the software engineers in charge of structural maintenance, and ultimately increase their confidence in using the results of the refactoring recommendation tools.

To this end, a set of preference-driven software refactoring techniques has been proposed, and their validity analyzed using various quality measures. The proposed techniques rely on tools borrowed from diverse research fields including computational search, statistical data processing and machine learning. We proposed four (4) major contributions, each of them taking into account a particular aspect of the preference of the software engineer.

These contribution are as follows:

1. **A novel, context-based software refactoring technique.** This is a profile-based search-based refactoring approach that account for the context of the software programmer. The approach draws on the power of computational search using the multi-objective simulated annealing (MOSA) [40] to find the best refactoring solution that takes into account the recent activity of the software engineer.The primary contributions of our profile-based approach can be summarized as follows:

    - Meet programmers' requirement to improve the quality of recently modified code before a new release. This includes the optimization of the refactoring cost in time.
    - Recommend refactorings solutions correlated with bug reports,
    - Take into account refactoring operations that were recently applied to the system.

    Thus, instead of recommending system-wide refactorings, which may be error-prone and time-consuming for large systems, the profile-based techniques constraints the suggested refactoring operation within the domain of code that are affected by the activity of the programmer.

2. **A novel, learning-based interactive search-based software refactoring technique.** This methodology uses a machine learning technique, the artificial neural network (ANN), to overcome the shortcomings of existing interactive search-based software engineering techniques that place a burden on the programmer due to requiring his or her presence for the full duration of the process. By using the ANN predictive model, our approach efficiently capture the coding preference of the programmer via the evaluation of the proposed refactorings. This evaluation by the programmer has the effect of biasing the resulting refactoring recommendations towards his or her

preferences. The main contributions in this techniques the modeling of the preference of the programmer in a machine learned fitness function, which eliminates the need for subjective metric-based fitness functions.

3. **A learning-based interactive web service modularization technique.** This technique is an adaption of our second contribution to the particular problem of web service refactoring. It take into account design consideration that are intrinsic to web service, but build on the same combination of ANN and heuristic search to integrate the user preference into the modularization process. Thus, it can be seen as a particular adaption of the same goal to a different domain.

4. **A novel, many-objective search-based refactoring technique using dimensionality reduction on the Pareto front.** This is a general search-based software engineering techniques that can be applied to any of the field of software engineering, whenever there is a large number of objectives. The main goal of this technique was to alleviate the decision making on the Pareto front by eliminating unnecessary objectives through PCA. This approach is particularly relevant to software refactoring due to the majority of existing techniques being driven by metric-based fitness functions that have been shown to be correlated. The primary contributions are as follows:

    - Help the programmer finds the best minimum set of objectives needed to evaluate search based software engineering solutions. The developers needs to just specify several evaluation measures (proprieties of the desired solution) and our technique can automatically identify possible correlations and conflicts between them to find the best set of fitness functions.
    - Eliminate multiple run of automated refactoring methods due to prior knowledge of important objectives that leads to objective selection by trial and error.
    - We propose a case study related to software refactoring. The goal is to find

the best minimum set of quality metrics and attributes that maybe needed to converge towards good refactoring recommendations.

7.2 Threats to Validity

There are four types of threats that can affect the validity of our experiments as described in this section.

Conclusion validity is concerned with the statistical relationship between the treatment and the outcome. We addressed conclusion threats to validity by performing on average 30 independent simulation runs for each problem instance and statistically analyzing the obtained results using the Wilcoxon rank sum test with a 99% confidence level ($\alpha = 1\%$). 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 for our approaches so that parameters are updated during the execution in order to provide the best possible performance.

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 process that involves randomization makes it highly unlikely that the observed results were caused by anything other than the applied respective search-algorithm, machine learning and dimensionality reduction approaches. The second internal threat is related to the variation of the manual evaluation between the different groups when using our approach and other tools such as JDeodorant. To counteract this, 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 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. We considered in our experiments the majority of votes from the developers. 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.

External validity refers to the generalizability of our findings. In this study, we performed our experiments on seven different widely used open-source systems belonging to different domains and having different sizes. 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. 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 metrics. Future replications of this study are necessary to confirm our findings.

7.3 Future Work

Though programmers are reluctant to adopt refactoring solutions from existing tools, automated and semi-automated refactoring techniques have a well-defined place in the increasingly growing software-dominated industry. Thus, researches that will help remove the barrier between refactoring and software engineers are needed.

A major shortcoming that we identified during our work is the lack of real-life testing of the recommended refactorings. While researchers are aware of the need for testing of the resulting software after refactoring, no real effort has been spent to actually consolidate the two fields, refactoring and testing, in a real-world scenario. To tackle this issue, as a future research, we will investigate the testing of the results of software refactoring as a combined study. While the topic appears challenging, many methods already exists that can be adapted in combination with software refactoring to validate the conservation of the functionality of resulting systems. For example, it is possible to extract models from resulting software systems obtained after refactoring using model extraction methods [136][137][138][139][140][41]. Then, once the model has been extracted from the refactored software, it can be tested using existing model verification techniques.

Another issue that is inherent to the area of software engineering is the tendency of developers to making bad coding choices. As highlighted in this thesis, these choices are often induced by external pressure due to software release milestone. In order to reduce the impact of external pressure on coding habit as well as support for non-experienced programmer, we will investigate in future research the possibility of communicating the available refactoring without intruding on the users of an existing IDE. This will allow for guiding the programmer towards best coding practices without breaking his or her workflow.

Finally, an important lesson learned during our research relates to knowledge transfer across heterogeneous projects. To this end, we would conduct research to answer two major questions:

- To what extent could refactoring be generalized across projects and not only within multiple releases of the same project?
- How to make refactoring independent from specific domains with the aim to reduce the effort required to adjust and adapt existing refactoring tools to new domains?

As an example investigation of these questions, we have successfully applied refactoring solutions used in general software development environment to Web service modularization. The results were discussed in chapter 4 and 5. investigating the answer to the above question will allow us to generalize the concept of cross-domain refactoring in a systematic framework.

BIBLIOGRAPHY

[1] Deb, K., Zope, P., and Jain, A., *Distributed Computing of Pareto-Optimal Solutions with Evolutionary Algorithms*, Springer Berlin Heidelberg, Berlin, Heidelberg, 2003, pp. 534–549.

[2] Sayyad, A. S. and Ammar, H., "Pareto-optimal search-based software engineering (POSBSE): A literature survey," *Proc. 2nd Int. Workshop Realizing Artificial Intelligence Synergies in Software Engineering (RAISE)*, May 2013, pp. 21–27.

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

[4] Cunningham, W., "The WyCash Portfolio Management System," *Addendum to the Proceedings on Object-oriented Programming Systems, Languages, and Applications (Addendum)*, OOPSLA '92, ACM, New York, NY, USA, 1992, pp. 29–30.

[5] Marinescu, R., "Assessing technical debt by identifying design flaws in software systems," *IBM Journal of Research and Development*, Vol. 56, No. 5, Sept. 2012, pp. 9:1–9:13.

[6] Brown, N., Cai, Y., Guo, Y., Kazman, R., Kim, M., Kruchten, P., Lim, E., MacCormack, A., Nord, R., Ozkaya, I., Sangwan, R., Seaman, C., Sullivan, K., and Zazworka, N., "Managing Technical Debt in Software-reliant Systems," *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*, FoSER '10, ACM, New York, NY, USA, 2010, pp. 47–52.

[7] Suryanarayana, G., Samarthyam, G., and Sharma, T., *Refactoring for Software Design Smells: Managing Technical Debt*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st ed., 2014.

[8] Abran, A. and Nguyen-Kim, H., "Measurement of the Maintenance Process from a Demand-based Perspective," *Software Maintenance: Research and Practice*, 1993, pp. 64–90.

[9] Harman, M., Mansouri, S. A., and Zhang, Y., "Search-based Software Engineering: Trends, Techniques and Applications," *ACM Comput. Surv.*, Vol. 45, No. 1, Dec. 2012, pp. 11:1–11:61.

[10] Harman, M. and Tratt, L., "Pareto Optimal Search Based Refactoring at the Design Level," *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation*, GECCO '07, ACM, New York, NY, USA, 2007, pp. 1106–1113.

[11] Mens, T. and Tourwe, T., "A survey of software refactoring," *IEEE Transactions on Software Engineering*, Vol. 30, No. 2, Feb. 2004, pp. 126–139.

[12] Liu, H., Guo, X., and Shao, W., "Monitor-Based Instant Software Refactoring," *IEEE Transactions on Software Engineering*, Vol. 39, No. 8, Aug 2013, pp. 1112–1126.

[13] Fenton, N. and Bieman, J., *Software metrics: a rigorous and practical approach*, CRC Press, 2014.

[14] Brown, W., Malveau, R., III, H. M., and Mowbray, T., *Anti-Patterns: Refactoring Software, Architectures, and Projects in Crisis*, John Wiley & Sons, Inc, New York, Chichester, Weinheim, 1998.

[15] Fowler, M., Beck, K., Brant, J., Opdyke, W., and Roberts, D., *Refactoring - improving the design of existing code*, Addison-Wesley, 1st ed., 1999.

[16] Kessentini, M., Kessentini, W., Sahraoui, H., Boukadoum, M., and Ouni, A., "Design Defects Detection and Correction by Example," *Proc. IEEE 19th Int. Conf. Program Comprehension*, June 2011, pp. 81–90.

[17] Gamma, E., Helm, R., Johnson, R., and Vlissides, J., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[18] Ouni, A., Gaikovina Kula, R., Kessentini, M., and Inoue, K., "Web Service Antipatterns Detection Using Genetic Programming," *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, GECCO '15, ACM, New York, NY, USA, 2015, pp. 1351–1358.

[19] Ouni, A., Kessentini, M., Inoue, K., and Cinnéide, M. O., "Search-based web service antipatterns detection," *IEEE Transactions on Services Computing*, 2015.

[20] Moha, N., Gueheneuc, Y. G., Duchien, L., and Meur, A. F. L., "DECOR: A Method for the Specification and Detection of Code and Design Smells," *IEEE Transactions on Software Engineering*, Vol. 36, No. 1, Jan. 2010, pp. 20–36.

[21] Marinescu, R., "Detection strategies: metrics-based rules for detecting design flaws," *Proc. 20th IEEE Int. Conf. Software Maintenance*, Sept. 2004, pp. 350–359.

[22] Mantyla, M., Vanhanen, J., and Lassenius, C., "A taxonomy and an initial empirical study of bad smells in code," *Proc. Int. Conf. Software Maintenance ICSM 2003*, Sept. 2003, pp. 381–384.

[23] ben Fadhel, A., Kessentini, M., Langer, P., and Wimmer, M., "Search-based detection of high-level model changes," *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, IEEE, 2012, pp. 212–221.

[24] Murphy-Hill, E., Parnin, C., and Black, A. P., "How We Refactor, and How We Know It," *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, IEEE Computer Society, Washington, DC, USA, 2009, pp. 287–297.

[25] Alkhazi, B., Ruas, T., Kessentini, M., Wimmer, M., and Grosky, W. I., "Automated Refactoring of ATL Model Transformations: A Search-based Approach," *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*, MODELS '16, ACM, New York, NY, USA, 2016, pp. 295–304.

[26] Ghannem, A., Boussaidi, G., and Kessentini, M., "Model Refactoring Using Interactive Genetic Algorithm," *Symposium on Search-Based Software Engineering*, 2013, pp. 96–110.

[27] Deb, K., Pratap, A., Agarwal, S., and Meyarivan, T., "A fast and elitist multiobjective genetic algorithm: NSGA-II," *IEEE Transactions on Evolutionary Computation*, Vol. 6, No. 2, April 2002, pp. 182–197.

[28] Ouni, A., Kessentini, M., Sahraoui, H., and Boukadoum, M., "Maintainability defects detection and correction: a multi-objective approach," *Automated Software Engineering*, 2013, pp. 1–33.

[29] Ouni, A., Kessentini, M., Sahraoui, H., Inoue, K., and Deb, K., "Multi-Criteria Code Refactoring Using Search-Based Software Engineering: An Industrial Case Study," *ACM Trans. Softw. Eng. Methodol.*, Vol. 25, No. 3, June 2016, pp. 23:1–23:53.

[30] Saxena, D. K. and Deb, K., "Non-linear Dimensionality Reduction Procedures for Certain Large-Dimensional Multi-objective Optimization Problems: Employing Correntropy and a Novel Maximum Variance Unfolding," *Evolutionary Multi-Criterion Optimization*, Jan. 2007.

[31] Harman, M. and Jones, B., "Search-Based Software Engineering," *Information and Software Technology*, Vol. 43, No. 14, Dec. 2001, pp. 833–839.

[32] Miller, W. and Spooner, D. L., "Automatic Generation of Floating-Point Test Data," *IEEE Transactions on Software Engineering*, Vol. SE-2, No. 3, Sept. 1976, pp. 223–226.

[33] Harman, M. and Clark, J., "Metrics are fitness functions too," *Proc. 10th Int. Symp. Software Metrics*, Sept. 2004, pp. 58–69.

[34] Mkaouer, M. W., Kessentini, M., Bechikh, S., Deb, K., and Ó Cinnéide, M., "Recommendation System for Software Refactoring Using Innovization and Interactive

Dynamic Optimization," *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, ACM, New York, NY, USA, 2014, pp. 331–336.

[35] Li, B., Li, J., Tang, K., and Yao, X., "Many-Objective Evolutionary Algorithms: A Survey," *ACM Comput. Surv.*, Vol. 48, No. 1, Sept. 2015, pp. 13:1–13:35.

[36] Mkaouer, W., Kessentini, M., Shaout, A., Koligheu, P., Bechikh, S., Deb, K., and Ouni, A., "Many-Objective Software Remodularization Using NSGA-III," *ACM Trans. Softw. Eng. Methodol.*, Vol. 24, No. 3, May 2015, pp. 17:1–17:45.

[37] Fonseca, C. M. and Fleming, P. J., "Genetic Algorithms for Multiobjective Optimization: FormulationDiscussion and Generalization," *Proceedings of the 5th International Conference on Genetic Algorithms*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993, pp. 416–423.

[38] Zitzler, E., Laumanns, M., and Thiele, L., "SPEA2: Improving the strength pareto evolutionary algorithm," *Evolutionary Methods for Design, Optimization and Control with Applications to Industrial Problems*, Athens, Greece, 2001, pp. 95–100.

[39] Knowles, J. and Corne, D., "The pareto archived evolution strategy: A new baseline algorithm for pareto multiobjective optimisation," *Evolutionary Computation, 1999. CEC 99. Proceedings of the 1999 Congress on*, Vol. 1, IEEE, 1999, pp. 98–105.

[40] Ulungu, E., Teghem, J., Fortemps, P., and Tuyttens, D., "MOSA method: a tool for solving multiobjective combinatorial optimization problems," *Journal of multicriteria decision analysis*, Vol. 8, No. 4, 1999, pp. 221.

[41] Kessentini, M., Bouchoucha, A., Sahraoui, H., and Boukadoum, M., "Example-based sequence diagrams to colored petri nets transformation using heuristic Search," *Modelling Foundations and Applications*, 2010, pp. 156–172.

[42] Lygoe, R. J., Cary, M., and Fleming, P. J., "A Many-objective Optimisation Decision-making Process Applied to Automotive Diesel Engine Calibration," *Proceedings of the 8th International Conference on Simulated Evolution and Learning*, SEAL'10, Springer-Verlag, Berlin, Heidelberg, 2010, pp. 638–646.

[43] Kirkpatrick, S., Gelatt, C. D., Vecchi, M. P., et al., "Optimization by simulated annealing," *science*, Vol. 220, No. 4598, 1983, pp. 671–680.

[44] López Jaimes, A., Coello Coello, C. A., and Chakraborty, D., "Objective Reduction Using a Feature Selection Technique," *Proceedings of the 10th Annual Conference on Genetic and Evolutionary Computation*, GECCO '08, ACM, New York, NY, USA, 2008, pp. 673–680.

[45] Deb, K. and Sundar, J., "Reference Point Based Multi-objective Optimization Using Evolutionary Algorithms," *Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation*, GECCO '06, ACM, New York, NY, USA, 2006, pp. 635–642.

[46] Thiele, L., Miettinen, K., Korhonen, P. J., and Molina, J., “A Preference-based Evolutionary Algorithm for Multi-objective Optimization,” *Evol. Comput.*, Vol. 17, No. 3, Sept. 2009, pp. 411–436.

[47] Said, L. B., Bechikh, S., and Ghedira, K., “The r-Dominance: A New Dominance Relation for Interactive Evolutionary Multicriteria Decision Making,” *IEEE Transactions on Evolutionary Computation*, Vol. 14, No. 5, Oct. 2010, pp. 801–818.

[48] Zitzler, E. and Knzli, S., “Indicator-based selection in multiobjective search parallel problem solving from nature,” *Parallel Problem Solving from Nature-PPSN*, Vol. VIII, Springer, Berlin Heidelberg, Sept. 2004, pp. 832–842.

[49] Kukkonen, S. and Lampinen, J., “Ranking-Dominance and Many-Objective Optimization,” *Proc. IEEE Congress Evolutionary Computation*, Sept. 2007, pp. 3983–3990.

[50] di Pierro, F., Khu, S. T., and Savic, D. A., “An Investigation on Preference Order Ranking Scheme for Multiobjective Evolutionary Optimization,” *IEEE Transactions on Evolutionary Computation*, Vol. 11, No. 1, Feb. 2007, pp. 17–45.

[51] Bader, J. and Zitzler, E., “Hype: An Algorithm for Fast Hypervolume-based Many-objective Optimization,” *Evol. Comput.*, Vol. 19, No. 1, March 2011, pp. 45–76.

[52] Zhang, Q. and Li, H., “MOEA/D: A Multiobjective Evolutionary Algorithm Based on Decomposition,” *IEEE Transactions on Evolutionary Computation*, Vol. 11, No. 6, Dec 2007, pp. 712–731.

[53] Deb, K. and Jain, H., “An Evolutionary Many-Objective Optimization Algorithm Using Reference-Point-Based Nondominated Sorting Approach, Part I: Solving Problems With Box Constraints,” *IEEE Transactions on Evolutionary Computation*, Vol. 18, No. 4, Aug. 2014, pp. 577–601.

[54] Jain, H. and Deb, K., “An Evolutionary Many-Objective Optimization Algorithm Using Reference-Point Based Nondominated Sorting Approach, Part II: Handling Constraints and Extending to an Adaptive Approach,” *IEEE Transactions on Evolutionary Computation*, Vol. 18, No. 4, Aug. 2014, pp. 602–622.

[55] Wang, R., Purshouse, R. C., and Fleming, P. J., “Preference-Inspired Coevolutionary Algorithms for Many-Objective Optimization,” *IEEE Transactions on Evolutionary Computation*, Vol. 17, No. 4, Aug. 2013, pp. 474–494.

[56] Opdyke, W. F., *Refactoring: A Program Restructuring Aid in designing Object-Oriented Application Frameworks*, Ph.D. thesis, University of Illinois at Urbana-Champaign, 1992.

[57] Du Bois, B., Demeyer, S., and Verelst, J., “Refactoring ” Improving Coupling and Cohesion of Existing Code,” *Proceedings of the 11th Working Conference on Reverse Engineering*, WCRE ’04, IEEE Computer Society, Washington, DC, USA, 2004, pp. 144–151.

[58] Murphy-Hill, E. and Black, A. P., "Breaking the Barriers to Successful Refactoring: Observations and Tools for Extract Method," *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, ACM, New York, NY, USA, 2008, pp. 421–430.

[59] Murphy-Hill, E. and Black, A. P., "Programmer-Friendly Refactoring Errors," *IEEE Trans. Softw. Eng.*, Vol. 38, No. 6, Nov. 2012, pp. 1417–1431.

[60] Ge, X. and Murphy-Hill, E., "Manual Refactoring Changes with Automated Refactoring Validation," *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, ACM, New York, NY, USA, 2014, pp. 1095–1105.

[61] Ge, X. and Murphy-Hill, E., "BeneFactor: A Flexible Refactoring Tool for Eclipse," *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, OOPSLA '11, ACM, New York, NY, USA, 2011, pp. 19–20.

[62] Foster, S. R., Griswold, W. G., and Lerner, S., "WitchDoctor: IDE Support for Real-time Auto-completion of Refactorings," *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, IEEE Press, Piscataway, NJ, USA, 2012, pp. 222–232.

[63] Tahvildari, L. and Singh, A., "Categorization of object-oriented software metrics," *2000 Canadian Conference on Electrical and Computer Engineering. Conference Proceedings. Navigating to a New Era (Cat. No.00TH8492)*, Vol. 1, 2000, pp. 235–239 vol.1.

[64] Dig, D., "A Refactoring Approach to Parallelism," *IEEE Softw.*, Vol. 28, No. 1, Jan. 2011, pp. 17–22.

[65] Kataoka, Y., Notkin, D., Ernst, M. D., and Griswold, W. G., "Automated Support for Program Refactoring Using Invariants," *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*, ICSM '01, IEEE Computer Society, Washington, DC, USA, 2001, pp. 736–.

[66] Fokaefs, M., Tsantalis, N., Stroulia, E., and Chatzigeorgiou, A., "JDeodorant: Identification and Application of Extract Class Refactorings," *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, ACM, New York, NY, USA, 2011, pp. 1037–1039.

[67] Seng, O., Stammel, J., and Burkhart, D., "Search-based Determination of Refactorings for Improving the Class Structure of Object-oriented Systems," *Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation*, GECCO '06, ACM, New York, NY, USA, 2006, pp. 1909–1916.

[68] O'Keeffe, M. and Ó Cinnéide, M., "Search-based Refactoring for Software Maintenance," *J. Syst. Softw.*, Vol. 81, No. 4, April 2008, pp. 502–516.

[69] Kilic, H., Koc, E., and Cereci, I., "Search-Based Parallel Refactoring Using Population-Based Direct Approaches," *Search Based Software Engineering*, Jan. 2011.

[70] Ouni, A., Kessentini, M., Sahraoui, H., and Hamdi, M. S., "The Use of Development History in Software Refactoring Using a Multi-objective Evolutionary Algorithm," *Proceedings of the 15th Annual Conference on Genetic and Evolutionary Computation*, GECCO '13, ACM, New York, NY, USA, 2013, pp. 1461–1468.

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

[72] Ó Cinnéide, M., Tratt, L., Harman, M., Counsell, S., and Hemati Moghadam, I., "Experimental Assessment of Software Metrics Using Automated Refactoring," *Proceedings of the ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '12, ACM, New York, NY, USA, 2012, pp. 49–58.

[73] Mkaouer, M. W., Kessentini, M., Bechikh, S., Cinnide, M., and Deb, K., "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, Dec. 2016, pp. 2503.

[74] Mkaouer, M. W., Kessentini, M., Bechikh, S., Deb, K., and Ó Cinnéide, M., "High Dimensional Search-based Software Engineering: Finding Tradeoffs Among 15 Objectives for Automating Software Refactoring Using NSGA-III," *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation*, GECCO '14, ACM, New York, NY, USA, 2014, pp. 1263–1270.

[75] Hall, M., Walkinshaw, N., and McMinn, P., "Supervised software modularisation," *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, Sept 2012, pp. 472–481.

[76] Bavota, G., Carnevale, F., De Lucia, A., Di Penta, M., and Oliveto, R., "Putting the Developer In-the-loop: An Interactive GA for Software Re-modularization," *Proceedings of the 4th International Conference on Search Based Software Engineering*, SSBSE'12, Springer-Verlag, Berlin, Heidelberg, 2012, pp. 75–89.

[77] Ouni, A., Kessentini, M., Sahraoui, H., and Boukadoum, M., "Maintainability Defects Detection and Correction: A Multi-Objective Approach. J. of Autmated Software Engineering," 2012.

[78] Piveta, E. K., Hecht, M., Moreira, A., Pimenta, M. S., Araújo, J., Guerreiro, P., and Price, R. T., "Avoiding Bad Smells in Aspect-Oriented Software." *SEKE*, 2007, p. 81.

[79] Ghannem, A., El Boussaidi, G., and Kessentini, M., "On the use of design defect examples to detect model refactoring opportunities," *Software Quality Journal*, Vol. 24, No. 4, 2016, pp. 947–965.

[80] Wang, H., Kessentini, M., and Ouni, A., "Bi-level identification of web service defects," *International Conference on Service-Oriented Computing*, Springer, 2016, pp. 352–368.

[81] Marinescu, C., Marinescu, R., Mihancea, P. F., and Wettel, R., "iPlasma: An integrated platform for quality assessment of object-oriented design," *In ICSM (Industrial and Tool Volume*, Citeseer, 2005.

[82] Meananeatra, P., "Identifying refactoring sequences for improving software maintainability," *Proc. 27th IEEE/ACM Int. Conf. Automated Software Engineering 2012*, Sept. 2012, pp. 406–409.

[83] Bechikh, S., Kessentini, M., Said, L. B., and Ghédira, K., "Chapter four-preference incorporation in evolutionary multiobjective optimization: A survey of the state-of-the-art," *Advances in Computers*, Vol. 98, 2015, pp. 141–207.

[84] Kalboussi, S., Bechikh, S., Kessentini, M., and Said, L. B., "Preference-based many-objective evolutionary testing generates harder test cases for autonomous agents," *International Symposium on Search Based Software Engineering*, Springer, 2013, pp. 245–250.

[85] Funahashi, K.-I., "On the approximate realization of continuous mappings by neural networks," *Neural Networks*, Vol. 2, No. 3, 1989, pp. 183 – 192.

[86] Yang, S., Ting, T., Man, K., and Guan, S.-U., "Investigation of Neural Networks for Function Approximation," *Procedia Computer Science*, Vol. 17, 2013, pp. 586 – 594.

[87] Palomba, F., Bavota, G., Penta, M. D., Oliveto, R., Lucia, A. D., and Poshyvanyk, D., "Detecting bad smells in source code using change history information," *Proc. 28th IEEE/ACM Int. Conf. Automated Software Engineering (ASE)*, Nov. 2013, pp. 268–278.

[88] Roberts, D. B., *Practical Analysis for Refactoring*, Ph.D. thesis, University of Illinois, 1999.

[89] McCulloch, W. S. and Pitts, W. H., "A Logical Calculus of Ideas Immanent in Nervous Activity," *Bulletin of Mathematical Biophysics*, Vol. 5, 1943, pp. 115–133.

[90] Hebb, D., *The Organization of Behavior*, Wiley & Sons, New York, 1949.

[91] Shaw, G. L., *Donald Hebb: The Organization of Behavior*, Springer Berlin Heidelberg, Berlin, Heidelberg, 1986, pp. 231–233.

[92] Rosenblatt, F., "The Perceptron: A Probabilistic Model for Information Storage and Organization in The Brain," *Psychological Review*, Vol. 62, 1958, pp. 386–408.

[93] Werbos, P., *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences*, Ph.D. thesis, Harvard University, 1974.

[94] Rumelhart, D. E., Hinton, G. E., and Williams, R. J., "Parallel Distributed Processing: Explorations in the Microstructure of Cognition," Vol. 1, chap. Learning Internal Representations by Error Propagation, MIT Press, Cambridge, MA, USA, 1986, pp. 318–362.

[95] Sejnowski, T. and Rosenberg, C., "Parallel networks that learn to pronounce English text," *Complex System*, Vol. 1, 1987, pp. 145–168.

[96] Waibel, A., "Modular Construction of Time-delay Neural Networks for Speech Recognition," *Neural Comput.*, Vol. 1, No. 1, March 1989, pp. 39–46.

[97] bin Huang, G., yu Zhu, Q., and kheong Siew, C., "Extreme learning machine: Theory and applications," *Neurocomputing*, Vol. 70, 2006, pp. 489–501.

[98] Kohonen, T., *Self-Organizing Maps*, Vol. 30 of *Springer Series in Information Sciences*, Springer, Berlin, Heidelberg, 2001.

[99] Sperduti, A. and Starita, A., "Supervised neural networks for the classification of structures," *IEEE Transactions on Neural Networks*, Vol. 8, No. 3, May 1997, pp. 714–735.

[100] Frasconi, P., Gori, M., and Sperduti, A., "A general framework for adaptive processing of data structures," *IEEE Transactions on Neural Networks*, Vol. 9, No. 5, Sept. 1998, pp. 768–786.

[101] Hornik, K., Stinchcombe, M., and White, H., "Multilayer Feedforward Networks Are Universal Approximators," *Neural Netw.*, Vol. 2, No. 5, July 1989, pp. 359–366.

[102] Willmes, L., Back, T., Jin, Y., and Sendhoff, B., "Comparing neural networks and Kriging for fitness approximation in evolutionary optimization," *Proc. Congress Evolutionary Computation CEC '03*, Vol. 1, Dec. 2003, pp. 663–670 Vol.1.

[103] Schmidhuber, J., "Deep Learning in Neural Networks: An Overview," *Neural Networks*, Vol. 61, 2015, pp. 85–117.

[104] Prete, K., Rachatasumrit, N., Sudan, N., and Kim, M., "Template-based reconstruction of complex refactorings," *Software Maintenance (ICSM), 2010 IEEE International Conference on*, IEEE, 2010, pp. 1–10.

[105] Jensen, A. C. and Cheng, B. H., "On the Use of Genetic Programming for Automated Refactoring and the Introduction of Design Patterns," *Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation*, GECCO '10, ACM, New York, NY, USA, 2010, pp. 1341–1348.

[106] Romano, D. and Pinzger, M., "Analyzing the evolution of web services using fine-grained changes," *Web Services (ICWS), 2012 IEEE 19th International Conference on*, IEEE, 2012, pp. 392–399.

[107] Dudney, B., Asbury, S., Krozak, J. K., and Wittkopf, K., *J2EE antipatterns*, John Wiley & Sons, 2003.

[108] Perepletchikov, M., Ryan, C., and Frampton, K., "Cohesion metrics for predicting maintainability of service-oriented software," *Quality Software, 2007. QSIC'07. Seventh International Conference on*, IEEE, 2007, pp. 328–335.

[109] Athanasopoulos, D., Zarras, A. V., Miskos, G., Issarny, V., and Vassiliadis, P., "Cohesion-driven decomposition of service interfaces without access to source code," *IEEE Transactions on Services Computing*, Vol. 8, No. 4, 2015, pp. 550–562.

[110] Ouni, A., Salem, Z., Inoue, K., and Soui, M., "SIM: an automated approach to improve web service interface modularization," *Web Services (ICWS), 2016 IEEE International Conference on*, IEEE, 2016, pp. 91–98.

[111] Mansoor, U., Kessentini, M., Maxim, B. R., and Deb, K., "Multi-objective code-smells detection using good and bad design examples," *Software Quality Journal*, Vol. 25, No. 2, June 2017, pp. 529.

[112] Kessentini, M., Mahaouachi, R., and Ghedira, K., "What You Like in Design Use to Correct Bad-smells," *Software Quality Journal*, Vol. 21, No. 4, Dec. 2013, pp. 551–571.

[113] Mitchell, M., *An Introduction to Genetic Algorithms*, MIT Press, Cambridge, MA, USA, 1998.

[114] Jayalakshmi, T. and Santhakumaran, A., "Statistical Normalization and Back Propagationfor Classification," *International Journal of Computer Theory and Engineering*, Vol. 3, No. 1, 2011, pp. 89.

[115] Coscia, J. L. O., Mateos, C., Crasso, M., and Zunino, A., "Avoiding wsdl bad practices in code-first web services," *Proceedings of the 12th Argentine Symposium on Software Engineering (ASSE2011)-40th JAIIO*, 2011, pp. 1–12.

[116] Mateos, C., Crasso, M., Zunino, A., and Coscia, J. O., "Avoiding WSDL bad practices in code-first web services," *SADIO Electronic Journal of Informatics and Operational Research*, Vol. 11, No. 1, 2012, pp. 31–48.

[117] Crasso, M., Rodriguez, J. M., Zunino, A., and Campo, M., "Revising WSDL Documents: Why and How," *IEEE Internet Computing*, Vol. 14, No. 5, Sept 2010, pp. 48–56.

[118] Almhana, R., Mkaouer, W., Kessentini, M., and Ouni, A., "Recommending Relevant Classes for Bug Reports Using Multi-objective Search," *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, ACM, New York, NY, USA, 2016, pp. 286–295.

[119] Sahin, D., Kessentini, M., Wimmer, M., and Deb, K., "Model transformation testing: a bi-level search-based software engineering approach," *Journal of Software: Evolution and Process*, Vol. 27, No. 11, 2015, pp. 821–837.

[120] Idri, A., Khoshgoftaar, T. M., and Abran, A., "Can neural networks be easily interpreted in software cost estimation?" *Fuzzy Systems, 2002. FUZZ-IEEE'02. Proceedings of the 2002 IEEE International Conference on*, Vol. 2, IEEE, 2002, pp. 1162–1167.

[121] Perepletchikov, M., Ryan, C., and Tari, Z., "The Impact of Service Cohesion on the Analyzability of Service-Oriented Software," *IEEE Trans. Serv. Comput.*, Vol. 3, No. 2, April 2010, pp. 89–103.

[122] Harman, M., Hassoun, Y., Lakhotia, K., McMinn, P., and Wegener, J., "The Impact of Input Domain Reduction on Search-based Test Data Generation," *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE '07, ACM, New York, NY, USA, 2007, pp. 155–164.

[123] XIE, T., TILLMANN, N., HALLEUX, P. D., , and SCHULTE, W., "Fitness-Guided path exploration in dynamic symbolic execution," Tech. Rep. MSR-TR-2008-123, Microsoft Researchg, Sept. 2008.

[124] Baker, P., Harman, M., Steinhofel, K., and Skaliotis, A., "Search based approaches to component selection and prioritization for the next release problem," *22nd IEEE International Conference on Software Maintenance (ICSM06*, IEEE, 2006, pp. 176–185.

[125] Zhang, Y., Alba, E., Durillo, J. J., Eldh, S., and Harman, M., "Today/future importance analysis," *Proceedings of the 12th annual conference on Genetic and evolutionary computation*, ACM, 2010, pp. 1357–1364.

[126] Feather, M. S., Kiper, J. D., and Kalafat, S., "Combining heuristic search, visualization and data mining for exploration of system design space," *14th Annual International Symposium of the Council on Systems Engineering (INCOSE04*, 2004.

[127] Jackson, J., *A Users Guide to Principal Components*, John Wiley and Sons, New York, 1991.

[128] Hotelling, H., "Analysis of a complex of statistical variables into principal components," *Journal of Educational Psychology*, Vol. 24, No. 6, Sept. 1933, pp. 417–441.

[129] Deb, K. and Saxena, D., "Searching for Pareto-optimal solutions through dimensionality reduction for certain large-dimensional multi-objective optimization problems," *2006 IEEE Congress on Evolutionary Computation (CEC'2006)*, IEEE, July 2006, pp. 3353–3360.

[130] Saxena, D. K., Duro, J. A., Tiwari, A., Deb, K., and Zhang, Q., "Objective Reduction in Many-Objective Optimization: Linear and Nonlinear Algorithms," *IEEE Transactions on Evolutionary Computation*, Vol. 17, No. 1, Feb. 2013, pp. 77–99.

[131] Brockhoff, D. and Zitzler, E., "Are All Objectives Necessary? On Dimensionality Reduction in Evolutionary Multiobjective Optimization," *Parallel Problem Solving from Nature - PPSN IX*, Springer, Jan. 2006.

[132] Wang, H. and Yao, X., "Objective reduction based on nonlinear correlation information entropy," *Soft Computing*, Vol. 20, No. 6, June 2016, pp. 2393.

[133] Zhou, C., Zheng, J., Li, K., and Lv, H., "Objective Reduction Based on the Least Square Method for Large-Dimensional Multi-objective Optimization Problem," *Proc. Fifth Int. Conf. Natural Computation*, Vol. 4, Aug. 2009, pp. 350–354.

[134] Brockhoff, D. and Zitzler, E., "Dimensionality Reduction in Multiobjective Optimization: The Minimum Objective Subset Problem," *Operations Research Proceedings 2006*, Springer, Jan. 2007.

[135] Corazza, A., Di Martino, S., and Maggio, V., "LINSEN: An efficient approach to split identifiers and expand abbreviations," *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, IEEE, 2012, pp. 233–242.

[136] Cánovas Izquierdo, J. L. and García Molina, J., "Extracting models from source code in software modernization," *Software and Systems Modeling*, 2012, pp. 1–22.

[137] Corbett, J. C., Dwyer, M. B., Hatcliff, J., Laubach, S., Pasareanu, C. S., Zheng, H., et al., "Bandera: Extracting finite-state models from Java source code," *Software Engineering, 2000. Proceedings of the 2000 International Conference on*, IEEE, 2000, pp. 439–448.

[138] Eisner, C., "Formal verification of software source code through semi-automatic modeling," *Software and Systems Modeling*, Vol. 4, No. 1, 2005, pp. 14–31.

[139] Holzmann, G. J. and H Smith, M., "Software model checking: Extracting verification models from source code," *Software Testing, Verification and Reliability*, Vol. 11, No. 2, 2001, pp. 65–79.

[140] Kessentini, M., Langer, P., and Wimmer, M., "Searching models, modeling search: On the synergies of SBSE and MDE," *Proceedings of the 1st International Workshop on Combining Modelling and Search-Based Software Engineering*, IEEE Press, 2013, pp. 51–54.