Balancing Competing Needs of Machine and Human in Search-Based Software Refactoring

by

Mohamed Wiem Mkaouer

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,
2016

Doctoral Committee:

Assistant Professor Marouane Kessentini, Chair
Professor & Koenig Endowed Chair Kalyanmoy Deb, Michigan State University
Dean Anthony England
Professor Mark Harman, University College London, UK
Associate Professor Bruce Maxim
Associate Professor Brahim Medjahed
Professor Qiang Zhu

DEDICATION

This thesis work is dedicated to:

My mother Chadlia

My father Mounir

My sister Amani

My brother Wael

My beautiful family who has been a constant source of support and encouragement during the challenges of graduate school and life. I am truly thankful for having you in my life. This work is the fruit of the hard work of my parents, who have always loved me unconditionally and whose good examples have taught me to work hard for the things that I aspire to achieve.

In the memory of:

My grandfather Houssin

My cousin Khaldoun

My uncle Houssin

My uncle Abdel-Aziz

I miss them every day, they stood up for me since I was a child, believed in me, but they are no longer among us to celebrate this moment with me, God forgives them and have mercy on them

ACKNOWLEDGEMENTS

This journey would not have been possible without the support of my family, professors, and friends. To my family, thank you for encouraging me in all of my pursuits and inspiring me to follow my dreams. I am especially grateful to my parents, who supported me emotionally and financially. I always knew that you believed in me and wanted the best for me. Thank you for teaching me that my job in life was to learn, to be an educator, and to seek knowledge for myself before I can spread it to others

I am very fortunate to have performed my graduate work at a university as collaborative and warm as the University of Michigan-Dearborn. Therefore, there are many people to thank for being a part of my success. I am greatly indebted to Prof. Marouane Kessentini, for guiding me as a researcher, teacher, and a person. I'm grateful to his genuine supervision over the last four years in which he has shown me what it means to be a dedicated and committed to research, he modeled a great teacher and for furthered my thinking about challenges in software quality and challenged my thinking by helping me question assumptions and view issues from multiple perspectives. Simply, He has given endlessly his time, energy, and expertise to pave the path of my Ph.D.

I would like to give special thanks to my dissertation committee that took some of their precious time to review my thesis. I am grateful to Prof. Qiang Zhu, Prof. Brahim Medjahed, Prof. Bruce Maxim, Dean Tony England, Prof. Mark Harman and Prof. Kalyanmoy Deb for their insightful suggestions for improvement, the constructive long discussion I had with them about the future directions I should take in my research.

I also would love to express my deep gratitude to Prof. Grosky for allowing me to be part of the Computer and Information Science department teaching faculty, I really enjoyed teaching several courses and interacting with bright students throughout my Ph.D. journey.

I would like to recognize the SBSE lab members who have shared a part of themselves and their work with me. Over the years, the connections made through several collaborations in our research or in our teaching and grading duties have enriched my life and I look forward to continuing our relationship. For my students, I would like to thank them for sharing their enthusiasm for science and thirst for knowledge.

To my friends and roommates, thank you for listening, offering me advice, and supporting me through this entire process. Special thanks to both my Missouri and Michigan friends along with my friends scattered around the globe and that are too numerous to name, thank you for your thoughts, well wishes/prayers, phone calls, e-mails, texts, visits, and being there whenever I needed a friend. To all the people I have known, thank you for sharing your life stories with me and for your friendship.

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. This work was funded by Search-Based Software Engineering Laboratory in collaboration with Ford Motor Company.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF APPENDICES

ABSTRACT

Successful software must evolve to remain relevant, but this process of evolution can cause the software design to decay and lead to significantly reduced productivity and even canceled projects. Several studies show that developers are postponing software maintenance activities that improve software quality, even while seeking high-quality source code for themselves when updating existing projects. One reason is that time and money pressures force developers to neglect improving the quality of their source code. However, a more fundamental reason is that there is little scientific understanding of how developers restructure/refactor source code for the purpose of improving program quality, which limits the support that researchers can offer developers. In fact, developers often need to make trade-offs between code quality, available resources and delivering a product on time, and such management support is beyond the scope and capability of existing refactoring engines. In addition, existing fully-automated techniques are under-utilized because of the lack of flexibility and worries about introducing bugs. While automation is important, it is essential to understand the points at which human intervention and decision-making should affect the automation process.

To address the above challenges, the main goal of this thesis is to develop a better understanding of how to integrate the preferences of software engineers in the maintenance process. The majority of existing work treats software engineering problems from a single-objective point of view, where the main goal is to maximize or minimize one objective, e.g., correctness, quality, etc. However, most software engineering problems are naturally complex in which many conflicting objectives need to be optimized such as model transformation to transform a model from a source to a target language, software refactoring to improve the quality systems while preserving the behavior, and software remodularization to improve the modularity of systems. The number of objectives to consider for most of software engineering problems is, in general, high (more than three objectives); such problems are termed many-objective. In this

context, the use of traditional multi-objective techniques widely used in software engineering is clearly not sufficient. The main contributions of this thesis can be summarized as follows:

- *Designing, implementing and validating novel scalable many-objective search algorithms for model transformation, software refactoring and software re-modularization*.
- *Optimization under uncertainty:* This contribution extends the formulation of software refactoring as a multi-objective problem to take into account the uncertainties related to the dynamic environment of software development. It enables the generation of robust refactoring solutions without a high loss of quality.
- *Designing, implementing and validating a novel interactive dynamic multi-objective search-based software refactoring algorithm:* The main goal of this contribution is to integrate the developer in the loop when performing software engineering activities such as software refactoring. The developers' feedback is used during the optimization process to converge towards the user's region of interest.

The validation of these contributions on both industrial and open source software systems, in the context of model transformation, software refactoring and software re-modularization, has shown promising results in helping software engineers in the improvement of their software quality while performing their regular maintenance tasks.

KEYWORDS

Search-based Software Engineering, Software Refactoring, Software Remodularization, Model-Driven Engineering, Many-Objective Optimization, Dynamic Interactive Optimization.

Chapter 1: Introduction

1.1 Research Context

Software engineering (SE) problems tend to be complex as they usually evolve ambiguous requirements associated with competing constraints and other external factors. The theoretical analysis of such complex problems in order to linearly solve them is also very hard as the solving process has to take into account an exhaustive list of multiples conditions, heterogeneous decisions and different methodologies to find the optimal solution. To illustrate the complexity of SE problems, let us consider the Software maintenance cycle. Software maintenance is a key artifact in managing the continuous evolution of software. Based on IEEE definition [1], the primary objective of maintenance is to ensure the software quality through its increasing versions and releases demonstrated by evolving user requirements, adding new features, fixing bugs, migrating to new environments and other maintainability practices. The previous decades of software engineering have proven that software maintenance problems tend to be complex, time-consuming and error-prone. Several surveys have been reporting that the highest portion of effort, in terms of manpower, money and time, in software project life cycles is being allocated to maintenance and it can reach up to 90 % of the overall software budget [2]. Therefore, providing new methodologies to preserve software quality while it's being changed has become fundamental to reduce the software maintainability costs.

In this context, the inefficiency of typical software engineering practices has driven the research towards reformulating traditional software engineering challenges as optimization problems and consequently deploying search heuristics to solve them. From the search-based perspective, any SE problem can be defined by a set of conflicting objectives, accompanied multiples constraints and other factors. Instead of targeting an optimal solution, the meta-heuristic search has the capability of generating multiple near-optimal solutions that eventually

represent the best tradeoff between the conflicting objectives while satisfying their constraints. In spite of the idea of considering SE issues as an optimization problem has been present in literature since the early decade of software engineering [3], the concept of Search-Based Software Engineering (SBSE) was coined by Mark Harman in 2001 [4]. It is an approach to software engineering in which artificial intelligent and search algorithms are deployed to identify optimal or near optimal but satisfactory solutions. According to Harman [5], SBSE is attractive, compared with other approaches, as it supports a wide variety of semi-automated, automated and hybrid techniques when facing complex problems with multiple, competing and conflicting objectives. SBSE has been showing promising results when compared to deterministic and exhaustive techniques, in terms of scalability, feasibility, robustness and insight-richness [6]. Since then, Search-based techniques have been widely applied to solve software engineering problems such as in testing, modularization, refactoring, planning, etc. The concept of SBSE will be detailed in the next chapter of this thesis.

1.2 Problem Statement

Most existing approaches to address software engineering problems (i.e., model transformation, design quality, testing, evolution, and comprehension, etc.) from a single objective point of view, where the main goal is to maximize or minimize one objective (e.g., correctness, quality, etc.). However, most SE problems are naturally complex in which many conflicting objectives need to be optimized such as model transformation, design quality, testing, etc. The number of objectives to consider for SE problems is, in general, high (more than three objectives). For example, evaluating the quality of a design after applying refactorings can be performed using a set of quality metrics where each metric can be considered as an objective. In this situation, the use of traditional mono-objective and multi-objective techniques, which are widely used in SBSE, is not sufficient as the context has a high number of objectives.

Recently, researchers have proposed several approaches to tackling many-objective optimization problems (e.g., objective reduction, new preference ordering relations, decomposition, etc.). However, these techniques are not explored yet in SBSE [5]. In this context, this research has been initiated to coin the limitations of the existing techniques due to the high dimensionality of SE problems by integrating many-objective methodologies.

Furthermore, this research aims to bridge the gap between software engineers and SBSE tools by integrating the engineer's preferences and feedback during the search process. To this end, this research will be firstly introducing the formularization of model transformation as a multi-objective optimization problem. Secondly, by highlighting the limitations of the multi-objective optimization, we will be discussing the potential benefits of deploying many-objective optimization in software engineering. Thirdly, we will be addressing the several challenges raised during the validation of the many-objective optimization through designing various algorithms that can be applied in practical software engineering problems associated with mainly (1) model transformation, (2) software remodularization, and (3) software refactoring.

1.2.1 Model Transformation

Model Transformation plays an important role in Model Driven Engineering (MDE) [7]. The research efforts by the MDE community have produced various languages and tools for automating transformations between different modeling environments using mapping rules. These transformation rules can be implemented using general programming languages such as Java or C#, graph transformation languages like AGG [8]and VIATRA [9], or specific languages such as ATL [10] and QVT [11]. Sometimes, transformations are based on invariants that are explicitly defined as pre-conditions and post-conditions and written in languages such as OCL [12].

Research Problem 1: One major challenge is to automate transformations while preserving the quality of the produced models [7]. Thus, the main goal is to reduce the number of possible errors when defining transformation rules. These transformation errors have different causes such as transformation logic (rules) or source/target metamodels. Existing approaches and techniques have been successfully applied to transformation problems with a minimum number of errors. Especially at the model level, correctness is the gold standard characteristic of models: it is essential that the user understands exactly how the target model deviates from fidelity to the source model in order to be able to rely on any results. However, other important objectives are how to minimize the complexity of transformation rules (e.g. the number of rules, number of matching in the same rule) while maximizing the quality of target models to obtain well-designed ones. In fact, reducing rules complexity and improving target models quality are

important to 1) make rules and target models easy to understand and evolve, 2) find transformation errors easily, and 3) generate optimal target models.

Research Problem 2: The majority of existing approaches [7] [8] [13] formulate the transformation problem as a single-objective problem that maximizes rules correctness. In this case, the proposed transformation rules produce target models without errors. However, these rules are sometimes complex (e.g., size) and applying them may generate very large target model. For example, when complex transformation rules are available for mapping from dynamic Unified Modeling Language (UML) models to Colored Petri Nets (CPN), systematically applying them generally results in large PNs [13]. This could compromise the subsequent analysis tasks, which are generally limited by the number of the PNs' states. Obtaining large PNs is not usually related to the size of the source models but to the rules complexity [14]. In addition, it is important to take into consideration the quality of produced target models (e.g. maximizing good design practices by reducing the number of design defects [15] in a generated class diagram from a relational schema). Another category of approaches [16] propose additional steps to minimize complexity, using refactoring operations, after generating transformation rules. However, it is a difficult and fastidious task to modify, evolve and improve the quality of already generated complex rules. To this extent, several open questions should be addressed during the process of generating transformation rules.

1.2.2 Software Remodularization:

Large software systems evolve and become complex quickly, fault-prone and difficult to maintain. In fact, most of the changes during the evolution of systems such as introducing new features or fixing bugs are conducted, in general, within strict deadlines and with a minimal number software developers and engineers. Consequently, these code changes can have a negative impact on the quality of systems design such as the distribution of the classes in packages. To address this issue, one of the widely used techniques is software remodularization, called also software restructuring, which improves the existing decomposition of systems. There has been much work on different techniques and tools for software remodularization [17] [18] [19]. Most of these studies addressed the problem of clustering by finding the best decomposition of a system in terms of modules and not by improving existing modularizations.

In both categories, cohesion and coupling are the main metrics used to improve the quality of existing packages (e.g. modules) by determining which classes need to be grouped in a package.

Research Problem 1: The majority of existing contributions have formulated the restructuring problem as a single-objective problem where the goal is to improve the cohesion and coupling of packages. Even though most of the existing approaches are powerful enough to provide remodularization solutions, some issues still need to be addressed.

Research Problem 2: One of the most important issues is the semantic coherence of the design. The restructured program could improve structural metrics but become semantically incoherent. In this case, the design will become difficult to understand since classes are placed in wrong packages to improve the structure in terms of cohesion and coupling. Also, the number of code changes is not considered when suggesting remodularization solutions; the only aim is to improve the structure of packages independently of the cost of code changes. However, in real-world scenarios, developers prefer, in general, remodularization solutions that improve the structure with a minimum number of changes. It is important to minimize code changes to help developers in understanding the design after applying suggested changes.

Research Problem 3: Existing remodularization studies are also limited to few types of changes mainly move class and split packages [18] [19]. However, refactoring at the class and method levels can improve remodularization solutions such as by moving methods between classes located in different packages. The use of development history can be an efficient aid when proposing remodularization solutions. For example, packages that were extensively modified in the past may have a high probability of being also changed in the future. Moreover, the packages to modify can be similar to some patterns that can be found in the development history, thus, developers can easily recognize and adapt them.

1.2.3 Software Refactoring:

Software Refactoring is defined as the process of improving code after it has been written by changing its internal structure without changing its external behavior [15]. The idea is to reorganize variables, classes and methods to facilitate future adaptations and extensions and

enhance comprehension. This reorganization is used to improve different aspects of software quality such as maintainability, extensibility, reusability, etc. Some modern Integrated Development Environments (IDEs), such as Eclipse, NetBeans, provide semi-automatic support for applying the most commonly used refactorings, e.g., move method, rename class, etc. However, automatically suggesting/deciding where and which refactorings to apply is still a real challenge in Software Engineering. In order to identify which parts of the source code need to be refactored, most existing work relies on the notion of code smells, also called design defects or anti-patterns.

Research Problem 1: Most of the existing refactoring work uses a set of more than 3 to 4 quality metrics to evaluate the quality of software design after applying refactoring operations while quality attributes, as well as design defects, are defined by more than simply 4 metrics. Consequently, more scalable search-based software refactoring approaches will be beneficial to handle such rich objective space. However, the difficulty faced when increasing the number of objectives could be summarized as follows. Firstly, most solutions become equivalent between each other according to the Pareto dominance relation which deteriorates dramatically the search's ability to converge towards the Pareto front and the MOEA behavior becomes very similar to the random search one. Secondly, a search method requires a very high number of solutions (some thousands and even more) to cover the Pareto front when the number of objectives increases, which makes the evolution towards optimal space very expensive.

Research Problem 2: Unlike software bugs, there is no general consensus on how to decide if a particular design violates a quality attribute. An optimal software design is generally described using informal or natural language and relies on a subjective interpretation of developers. Furthermore, different experts can have divergent opinions when identifying possible refactoring opportunities for the same software design. Overall, evaluating the quality of a design is subjective. In addition to the presence of false positives that may create a rejection reaction from development teams, the process of exposing several possible refactoring opportunities, in terms of candidate solutions to execute, asking the developers to understand their impact, and selecting one of them to apply, is long, expensive, and not always profitable. Thus, incorporating

Decision Maker (DM) (designer/expert) preferences can facilitate the best solution selection during the correction process.

Research Problem 3: Fully automated refactoring for large systems involves the application of a lengthy sequence of refactorings at several levels of the source code. This becomes impractical when developers have constraints over some parts of the system that need to remain unchanged. Furthermore, due to the dynamic nature of software development, developers may want to run a subset of refactoring and postpone others. These constraints are not being handled in the classic optimization process, leading to refactoring sequences that lack robustness as their application cannot be accurately determined in practice.

Research Problem 4: Indeed, fully-automated refactoring has several drawbacks as well. It lacks flexibility since developers have to either accept or reject the entire refactoring solution. It fails to consider developer perspective and feedback because suggested refactoring solutions cannot be updated dynamically. It is limited to structural improvements, which leads to infeasible refactoring solutions, and finally, it proposes a long static list of refactorings to be applied but developers may not have enough time to apply all of them. Thus, fully-automated refactoring methods are not useful for floss refactoring where the goal is to maintain good design quality while modifying existing functionality. The developers have to accept the entire refactoring solution even though they prefer, in general, step-wise approaches where the process is interactive and they have total control of the refactorings being applied.

1.3 Proposed Research Contributions

Our primary goal was to explore a higher dimensionality of different SE problems. Problems belonging to different domains, such as model transformation, software refactoring and remodularization, have eventually different formulations (objectives, constraints etc.). The following figure will cluster our contributions according to their application domains.

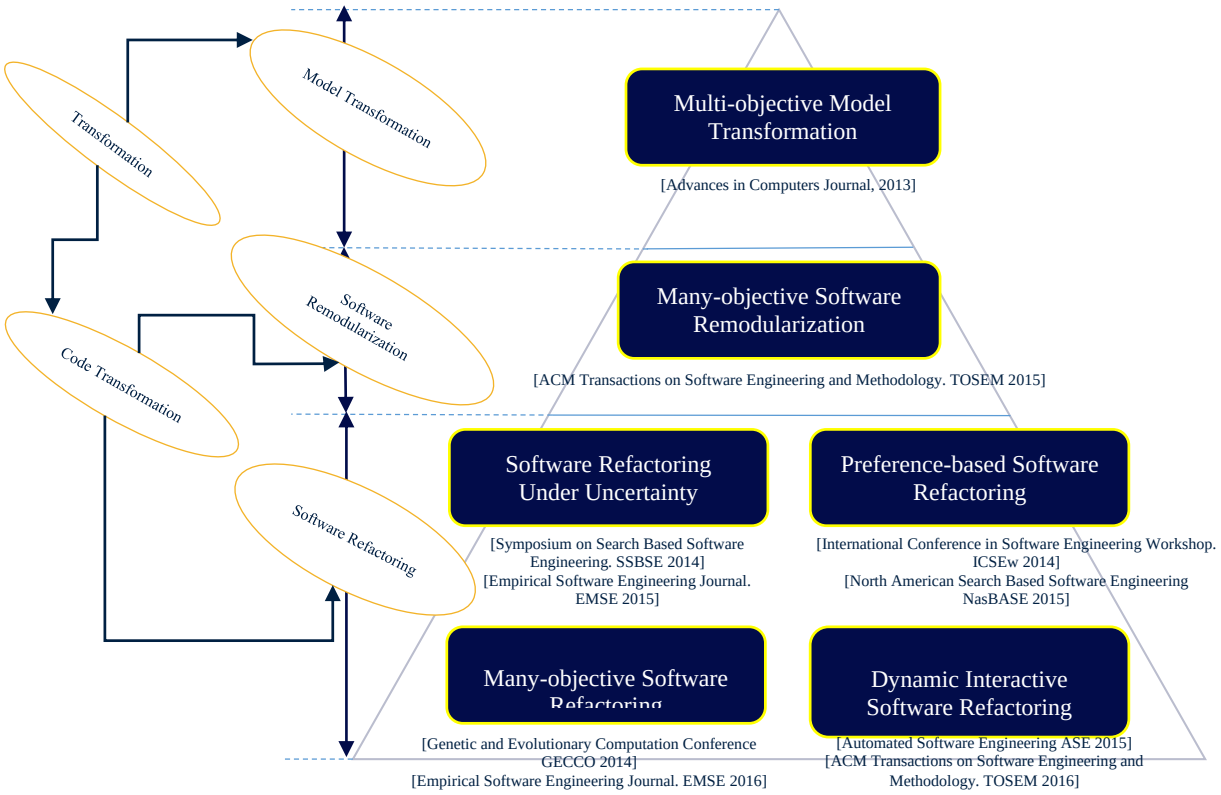


Figure 1.1. Distribution of our contributions based on the application domains.

Indeed, various techniques will be associated with different problems based on their specifications. The validation of these techniques is eventually achieved through rigorous empirical evaluation of their Pareto-optimal solutions: The qualitative and quantitative evaluation will be performed on empirical data collected from several open source and industrial systems. Several comparisons with the related work will be conducted, when possible. Finally, encapsulating all the implemented algorithms into a common framework that can be practically used by software engineers when performing software maintenance and evolution activities in a scalable fashion. In this thesis I have elaborated the following contributions:

1.3.1 Contribution 1: Multi-objective Model Transformation

We introduce a new approach for model transformation using multi-objective optimization. Our proposal does not require to define rules manually, but only to input a set of source models and equivalent target models (without traceability links); it generates well-designed target models/rules without the need to refactor them; it takes in consideration the complexity of the

generated rules; and it can be applied to any source or target metamodels (independent from source and target languages). We used three different transformation mechanisms to evaluate our proposal: class diagrams to relational schemas and vice-versa; and sequence diagrams to colored Petri nets. The generated rules for both mechanisms achieved high-quality scores with a minimal set of rules.

1.3.2 Contribution 2: Many-objective Software remodularization

We introduce a novel formulation of the remodularization problem as a many-objective problem that considers several objectives such as structural improvement, semantic coherence, number of changes and consistency with history of changes. We consider in the contribution the use of new operations, comparing to existing remodularization studies including move method, extract class and merge packages. This work was the subject of an empirical study of our many-objective technique compared to different existing approaches. The obtained results provide evidence to support the claim that our proposal is, in average, more efficient than existing techniques based on a benchmark of four large open source systems and one industrial project.

1.3.3 Contribution 3: High Dimensional and Dynamic Interactive Software refactoring

We focus on the correction of design defects by proposing automated refactoring driven by metaheuristic search and guided by software quality metrics and used subsequently to address the problem of automating design improvement. To overcome the previously stated research problems, we propose several contributions that can be enumerated as follows.

1.3.3.1 Multi-objective Software Refactoring under uncertainty

The work introduces a novel formulation of the refactoring problem as a multi-objective problem that takes into account the uncertainties related to code smell detection and the dynamic environment of software development. To the best of our knowledge, this is the first work to use robust optimization for software refactoring, and the first in SBSE to treat robustness as a helper objective during the search. We report, in the results section, an empirical study of our robust technique as applied to 6 open source systems. We compared our approach to random search, multi-objective particle swarm optimization (MOPSO), and other refactoring techniques. The

results provide evidence to support the claim that our proposal enables the generation of robust refactoring solutions without a high loss of quality using a variety of real-world scenarios.

1.3.3.2 Many-objective Software Refactoring

We propose for the first time a scalable search-based software engineering approach based on NSGA-III where there are 15 different objectives to optimize. Thus, in our approach, automated refactoring solutions will be evaluated using a set of 15 software quality metrics. NSGA-III is a very recent many-objective algorithm proposed by Deb et al. [20]. The basic framework remains similar to the original NSGA-II algorithm [21], with significant changes in its selection mechanism. This contribution represents the first real-world application of NSGA-III and the first scalable work that supports the use of 15 objectives to address a software engineering problem. We implemented our approach and evaluated it on seven large open source systems and found that, on average, more than 92% of code smells were corrected. The statistical analysis of our experiments over 31 runs shows that NSGA-III performed significantly better than two other many-objective techniques (IBEA and MOEA/D), a multi-objective algorithm (NSGA-II) and two mono-objective approaches.

1.3.3.3 Dynamic Interactive Software Refactoring

We propose, for the first time, the use of innovization (innovation through optimization) [22] to analyze and explore the Pareto front interactively with the developers. Our innovization algorithm starts by finding the most frequent refactorings among the set of non-dominated refactoring solutions. Based on this analysis, the suggested refactorings are ranked and suggested to the developer one by one. The developer can approve, modify or reject each suggested refactoring. This feedback is later used to update the ranking of the suggested refactorings. After a number of introduced code changes (e.g. fix bugs, add new requirements, etc.) and interactions with the developers, the search is executed again while taking the feedback as constraints to prune the search space and converge towards a user-preferred region.

We implemented our proposed approach and evaluated it on six open source systems. In addition, we evaluated our proposal on one industrial system provided by our industrial partner, i.e., the Ford Motor Company. Statistical analysis of our experiments over 31 runs showed that our proposal performed significantly better than four existing search-based approaches, and

manual refactorings using Eclipse. The software developers who participated in our experiments confirmed the relevance of the suggested refactoring and the flexibility of the tool in modifying and adapting the suggested refactorings.

1.4 Roadmap

The remainder of this thesis is structured as follows:

Chapter 2 reviews related work on model transformation, software remodularization and software refactoring. Chapter 3 reports our contribution for automating model transformation as a multi-objective optimization problem and which was published in Advances in Computers Journal [23]. Chapter 4 introduces software remodularization as a many-objective optimization that was published in the ACM Transactions on Software Engineering and Methodology [24]. Chapter 5 gathers various contributions related to software refactoring optimization. We present our Empirical Software Engineering journal paper [25] for the many-objective software refactoring. For the optimization under uncertainty, our Empirical Software Engineering journal paper [26] incorporates the robustness in the search process. The third and last part of chapter 5 presents our approach to design the dynamic interactive optimizer. This contribution is illustrated via our ACM Transactions in Software Engineering and Methodology which is under review. Chapter 6 presents the conclusions of this dissertation and outlines some directions for future research

Chapter 2: Related work

This chapter highlights the state of the art of existing work related to our proposed approaches. It starts by defining the concept of software engineering as an optimization problem, enumerates the related work in our areas of interests: (1) model transformation, (2) software remodularization, and (3) software refactoring.

2.1 Search-Based Software Engineering

2.1.1 Introduction

Our research is contributions are all gathered under the umbrella of Search-Based Software Engineering (SBSE). SBSE is defined as the application of search-based approaches to solving optimization problems in software engineering [4]. Once a software engineering task is framed as a search problem, there are numerous approaches that can be applied to solving that problem, from local searches such as exhaustive search and hill-climbing to meta-heuristic searches such as Genetic Algorithms (GAs) and ant colony optimization. All these techniques offer various explorations to the search space under various user-tuned parameters. The search results in one to possibly multiple candidate solutions as the output of the problem being analyzed. The evaluation process is done through a fitness function that, its optimization, guides the search towards an optimal or near-optimal state. Thus, the fitness function is calculated based on the outcome of the solution performance, once executed, to solve the given problem. Such evaluation can be automatically done in software engineering, unlike other engineering disciplines, as it is the unique discipline whose artifacts are virtual. This property is one of the main reasons behind the rapid growth of SBSE in the SE literature and many contributions have been proposed for various problems, mainly in cost estimation, testing, and maintenance.

SBSE is also very generic, by defining a fitness function, a given problem can be tackled by various search based optimization strategies. SBSE becomes of great value when there is a vast number of possible combinations of candidate solutions in the search space that their evolution is guided by a fitness function. A candidate solution can be defined in various ways depending on the problem formulation, it can be a vector, a graph, a set of rules or a sequence of code changes. According to Harman [4], SBSE methodology can be summarized in the following steps:

Problem formulation: For multiple candidate solutions to the same problem, the evaluation of their quality is assessed by a fitness function, which can be defined by the degree of which, it is meeting the expected result for the problem.

Solution representation: The formulation of a given SE problem is achieved by defining a possible solution representation that solves that problem

Solution variation: In each search algorithm, the variation operators play the key role of moving candidate solutions within the search space with the aim of driving them towards optimal solutions. These recombination operators need to be defined respectfully to the solution presentation and their application should derive new solutions with eventually different fitness values. The deployed algorithm has the responsibility to conduct the search and evolve the candidate solutions until stopping criteria are being met.

Consequently, our tackled SE problems will be presented according to the above-mentioned steps. Our problem formulation will be using population-based multi-objective genetic algorithms, where solutions are defined similarly to genes, their reproduction is maintained by crossover and mutation operators along with the repeated calculation of their fitness values to select the best solutions and constitute the next generation. Through the generations, solutions are being guided in the search space using the problem's fitness functions until stopping criteria are being met, and an optimum is found.

Based on recent SBSE survey [5], treating SE problems from a single-objective perspective us insufficient, as most SE problems are naturally complex in which many conflicting objectives

need to be optimized. As a consequence, our formulated problems are defined by many objectives. This formulation is detailed in the following sub-section.

2.1.2 Many-objective Search-Based Software Engineering

Recently, many-objective optimization has attracted much attention in evolutionary multi-objective optimization (EMO) which is one of the most active research areas in evolutionary computation. By definition, a many-objective problem is multi-objective, but with a high number of objectives *M*, namely *M* >3. Analytically, it could be stated as follows:

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

where *M* is the number of objective functions and is *strictly greater* than 3, *P* is the number of inequality constraints, *Q* is the number of equality constraints, x_i^L and x_i^U correspond to the lower and upper bounds of the decision variable x_i (i.e., i^{th} component of *x*). A solution x satisfying the (*P*+*Q*) constraints is said to be feasible and the set of all feasible solutions defines the feasible search space denoted by *Ω*.

In this formulation, we consider a minimization MOP since maximization can be easily transformed to minimization based on the duality principle by negating each objective function. Over the past two decades, several Multi-Objective Evolutionary Algorithms (MOEAs) have been proposed with the hope to work with any number of objectives *M*. Unfortunately, it has been demonstrated that most MOEAs are ineffective in handling many-objective problems. For example, NSGA-II, which is one of the most used MOEAs, compares solutions based on their non-domination ranks. Solutions with higher ranks are emphasized in order to converge to the Pareto front. When *M*>3, there is a high probability that almost all population individuals become non-dominated with each other, resulting in them all being lumped together in a single rank. Thus, NSGA-II is not able to maintain selection pressure in high-dimensional objective spaces.

The difficulty faced when solving a many-objective problem can be summarized as follows. Firstly, most solutions become of equivalent quality to each other according to the Pareto dominance relation which deteriorates *dramatically* the search process' ability to converge towards the Pareto front and the MOEA behavior becomes very similar to random search. Secondly, a search method requires a very high number of solutions (some thousands or even more) to cover the Pareto front when the number of objectives increases. For instance, it has been shown that in order to find a good approximation of the Pareto front for problems involving 4, 5 and 7 objective functions, the number of required non-dominated solutions is about 62 500, 1 953 125 and 1 708 984 375 respectively, which makes the decision-making task very difficult. Thirdly, the objective space dimensionality increases significantly which makes promising search directions very hard to find. Finally, the Pareto front visualization becomes more complicated for the DM, thereby complicating the interpretation of the MOEA's results.

Recently, researchers have proposed several solution approaches to tackle many-objective optimization problems. Table 2.1 illustrates a summary of existing many-objective approaches. Firstly, we find the o*bjective reduction approach*, which involves finding the minimal subset of objective functions that are in conflict with each other. The main idea is to study the different conflicts between the objectives. The objective reduction approach attempts to eliminate objectives that are not essential to describe the Pareto-optimal front. Even when the essential objectives are four or more, the reduced representation of the problem has a favorable impact on the search efficiency, computational cost, and decision making. However, although this approach has solved benchmark problems involving up to 20 objectives, its applicability in real world setting is not straightforward and it remains to be investigated since most objectives are usually in conflict with each other in real problems. Secondly, we have the *incorporation of decision maker's preferences*: When the number of objective functions increases, the Pareto optimal approximation would be composed of a huge number of non-dominated solutions. Consequently, the selection of the final alternative would be very difficult for the human decision maker (DM). In reality, the DM is not interested with the whole Pareto front rather than the portion of the front that best matches his/her preferences, called the Region of Interest (ROI). The main idea is to exploit the DM's preferences in order to differentiate between Pareto equivalent solutions so that we can direct the search towards the ROI on problems involving more than 3 objectives.

Preference-based MOEAs have demonstrated several promising results. Thirdly, we find *new preference ordering relations*. Since the Pareto dominance has the ability to differentiate between solutions with the increased of the number of objectives, researchers have proposed several new alternative relations. These relations try to circumvent the failure of the Pareto dominance by using additional information such as the ranks of the particular solution regarding the different objectives and the related population, but may not be agreeable to the decision makers. Fourthly, we have *decomposition*. This technique consists of decomposing the problem into several sub-problems and then solving these sub-problems simultaneously by exploiting the parallel search ability of evolutionary algorithms. The most reputable decomposition-based MOEA is MOEA/D. Finally, we find the u*se of a predefined multiple targeted search*. Inspired by preference-based MOEAs and the decomposition approach, recently, Deb and Jain [20], have proposed a new idea that involves guiding the population during the optimization process based on multiple predefined targets (e.g., reference points, reference direction) in the objective space. This idea has demonstrated very promising results on MOPs involving up to 15 objectives. Table 2.1. Recapitulates the stated many-objective approaches.

Table 2.1. Summary of many-objective approaches.

Approach	Basic idea	Example algorithms	No. of objectives	Real world many-objective application
Objective reduction	Find the minimal subset of conflicting objectives, then eliminate the objectives that are not essential to describe the Pareto optimal front.	1) PCA-NSGA-II 2) PCSEA	10 20	1) Not found 2) Water resource problem
Incorporating decision maker's preferences	Exploit DM's preferences in order to differentiate between Pareto equivalent solutions so that we can direct the search towards the region of interest instead of the whole front.	1) r-NSGA-II 2) PBEA 3) R-NSGA-II	10 10 10	1) Payment scheduling negotiation problem 2) Not found 3) Welded beam design problem
New preference ordering relations	Propose alternative preference relations that are different from the Pareto dominance.	1) Preference Order Ranking–based algorithm 2) Ranking dominance-based algorithm 3) IBEA 4) HypE	8 10 5 20	1) Water distribution problem 2) Not found 3) Software product line management 4) Not found
Decomposition	Decompose the problem into several sub-problems and then solve these subproblems simultaneously by exploiting the parallel search ability of EAs.	1) MOEA/D	5	1) Not found
Use of a predefined multiple	Guide the population during the optimization process based on multiple predefined targets (e.g.,	1) PICEA 2) NSGA-III	10 15	1) Not found 2) Crash-worthiness Design of Vehicles

targeted search	reference points, reference direction) in the objective space.			

The following table summarizes the existing SBSE approaches with the number of objectives being equal or higher than 5.

Table 2.2. Many-objectives approaches applied in software engineering.

Author(s)	Year	Title	Area	Algorithm(s)	Number of objectives
Z. Liu, H. Guo, D. Li, T. Han and J. Zhang [27]	2007	Solving Multi-objective and Fuzzy Multi-attributive Integrated Technique for QoS Aware Web Service Selection	Design Engineering	MOGA	5
H. Wada, P. Champrasert, J. Suzuki and K. Oba [28]	2008	Multiobjective Optimization of SLA- Aware Service Composition	Design Engineering	E^3-MOGA	10
M. Bowman, L. C. Briand and Y. Labiche [29]	2010	Solving the Class Responsibility Assignment Problem in Object-Oriented Analysis with Multi-Objective Genetic Algorithms	Design Engineering	SPEA2	5
T. Kremmel, J. Kubalik and S. Biffl [30]	2011	Software Project Portfolio Optimization with Advanced Multiobjective Evolutionary Algorithms	Management Engineering	mPOEMS,	5
D. Rodríguez, M. Ruiz, J. C. Riquelme and R. Harrison [31]	2011	Multiobjective Simulation Optimisation in Software Project Management	Management Engineering	NSGA-II	5
K. Praditwong, M. Harman and X. Yao [32]	2011	Software Module Clustering as a Multi-Objective Search Problem	Design Engineering	Two-Archive GA	5
M. O. Barros [33]	2012	An Analysis of the Effects of Composite Objectives in Multiobjective Software Module Clustering	Design Engineering	NSGA-II	5
T. E. Colanzi and S.R. Vergilio [34]	2012	Applying Search-Based Optimization to SPL Architectures: Lessons Learned	Design Engineering	NSGA-II	5
F. Sarro, F. Ferrucci and C. Gravino [35]	2012	Single and Multi-Objective GP for Software Development Effort Estimation	Management engineering	MOGP	5
A. S. Sayyad, T. Menzies and H. Ammar [36]	2013	On the Value of User Preferences in Search-Based Software Engineering: A Case Study in Software Product Lines	Requirements Engineering	IBEA	Up to 5
A. S. Sayyad, J. Ingram and T. Menzies [37]	2013	Scalable Product Line Configuration: A Straw to Break the Camel's Back	Requirements Engineering	IBEA	5
S. Kalboussi, S. Bechikh, M. Kessentini and L. Ben Said [38]	2014	Preference-Based Many-Objective Evolutionary Testing Generates Harder Test Cases for Autonomous Agents	Testing Engineering	P-MOET	7
A. Ramírez, J R. Romero and S. Ventura [39]	2014	On the Performance of Multiple Objective Evolutionary Algorithms for Software Architecture Discovery	Design Engineering	SPEA2, NSGA-II, ε-MOEA, MOEA/D, GrEA	6
R. Olaechea, D. Rayside, J. Guo and K. Czarnecki [40]	2014	Comparison of Exact and Approximate Multi-Objective Optimization for Software Product Lines	Requirements Engineering	GIA, IBEA	Up to 7
A. Panichella, F. M. Kifetew, P. Tonella [41]	2015	Reformulating branch coverage as a many-objective optimization problem	Testing Engineering	MOSA	Up to 1213

We notice from the previous table that some software engineering domains such as model-driven engineering and software maintenance have not yet been the subject of many-objective optimization although they are, as we have shown earlier, in need of scalable approaches. This

was our motivation to firstly investigate the applicability of many-objective optimization based on the currently defined problems and secondly adapt novel algorithms to solve them. The next section will briefly introduce two main many-objective techniques that we deployed in our methodology. The following subsections will present the state of the art of our SE areas of interest i.e. model and code transformation to analyze the existing work and expose its limitation in handling high dimensional problems.

2.2 Model transformation

Kleppe et al. [42] have provided the following definition of model transformation. A transformation is the automatic generation of a target model from a source model, according to a transformation definition. A transformation definition is a set of transformation rules that together describe how a model in the source language can be transformed into a model in the target language. A transformation rule is a description of how one or more constructs in the source language can be transformed into one or more constructs in the target language.

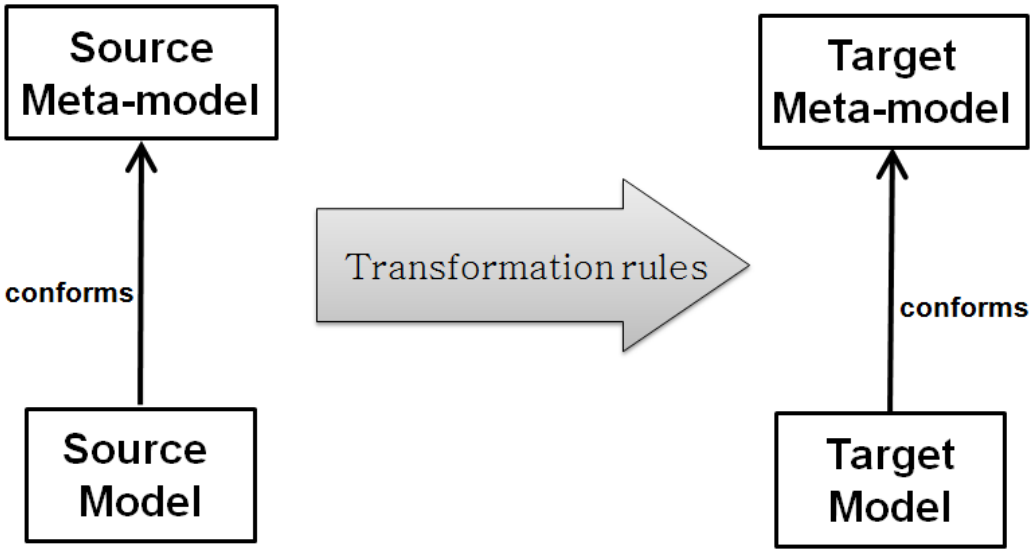


Figure 2.1. Model transformation process.

The closest work to our proposal is model transformation by example (MTBE). The commonalities of the by-example approaches for model transformation can be summarized as follows: All approaches define an example as a triple consisting of an input model and its equivalent output model, and traces between the input and output model elements. These

examples have to be established by the user, preferably in concrete syntax. Then, generalization techniques such as hard-coded reasoning rules, inductive logic, or relational concept analysis or pattern are used to derive model transformation rules from the examples, in a deterministic way that is applicable for all possible input models which have a high similarity with the predefined examples. One conclusion to be drawn from studying the existing by-example approaches is that they use semi-automated rules generation, with the generated rules further refined by the user. In practice, this may be a lengthy process and require a large number of transformation examples to assure the quality of the inferred rules. In this context, the use of search-based optimization techniques can be a preferable transformation approach since it directly generates the target model from the existing examples, without using the rules step. This also leads to a higher degree of automation than in existing by-example approaches. Table 2.3 summarizes existing transformation by-example approaches according to given criteria. The majority of these approaches are specific to exogenous transformation and based on the use of traceability.

Table 2.3. By-example Approaches.

By-example approaches	Exogenous transformation (different languages)	Endogenous transformation (same language)	Traceability	Rules generation
Varrò et al. [9]	X		X	X
Wimmer et al. [43]	X		X	X
Sun et al. [44]		X	X	
Dolques et al. [45]	X		X	X
Langler et al. [46]	X		X	X

As shown in the search-based section, like many other domains of software engineering, MDE is concerned with finding exact solutions to these problems, or those that fall within a specified acceptance margin. Search-based optimization techniques are well-suited for the purpose. For example, when testing model transformations, the use of deterministic techniques can be unfeasible due to the number of possibilities to explore for test case generation, in order to cover all source meta-model elements. However, the complex nature of MDE problems sometimes requires the definition of complex fitness functions. Furthermore, the definition is specific to the problem to solve and necessitate expertise in both search-based and MDE fields. It is thus

desirable to define a generic fitness function, evaluating a quality of a solution that can be applied to various MDE problems with low adaptation effort and expertise.

To tackle these challenges, our contribution combines search-based and by-example techniques. The difference with case-based reasoning approaches is that many sub-cases can be combined to derive a solution, not just the most adequate case. In addition, if a large number of combinations have to be investigated, the use of search-based techniques becomes beneficial in terms of search speed to find the best combination.

2.3 Software remodularization

Large systems such as automotive industry applications have to run and evolve over decades. Most of the industrial systems must evolve and the design is, in general, extended far away the initial structure. Thus, it is mandatory to restructure the program design to reduce the cost of possible future evolutions. To this end, software remodularization is an important component in software maintenance activities.

Object-oriented software modularization consists of regrouping a set of classes C in terms of packages P. Thus, each package P contains a set of classes. Several types of dependencies between packages can be found in the literature. In this work, we use the definition of dependencies between packages defined in [32]. Two main types of dependencies are described: 1- intra-edges dependencies and 2- inter-edges dependencies. The intra-edges include all types of internal dependencies between classes in the same package such as method call, class reference and inheritance. The inter-edges include external dependencies between classes that are not in the same package. As illustrated in Figure 2.2, the system includes 2 packages, 3 intra-edges such as (c1, c3) and 2 inter-edges such as (c3, c4) for package P1.

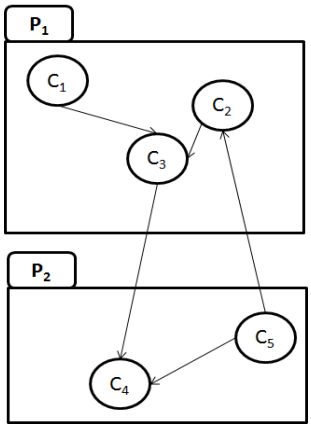


Figure 2.2. The dependency graph including two packages, 3 intra-edges and 2 inter-edges.

There have been several developments in search-based approaches to support the automation of software modularization. The work of Mancoridis et al. [47] was the first search-based approach to address the problem of software modularization using a single-objective optimization. Their initial work, to identify the modularization of a software system, is based hill-climbing to maximize cohesion and minimize coupling. The same technique has been also used in [48] where the authors present Bunch, a tool supporting automatic system decomposition. Subsystem decomposition is performed by Bunch by partitioning a graph of entities and relations in a given source code. To evaluate the quality of the graph partition, a fitness function is used to find the trade-off between interconnectivity (i.e., dependencies between the modules of two distinct subsystems) and intraconnectivity (i.e., dependencies between the modules of the same subsystem), to found out a satisfactory solution. In [49], Harman et al. use a genetic algorithm to improve the subsystem decomposition of a software system. The fitness function to maximize is defined using a combination of quality metrics, e.g., coupling, cohesion, and complexity. Similarly, Seng et al. [50], treated the remodularization task as a single-objective optimization problem using the genetic algorithm. The goal is to develop a methodology for object-oriented systems that, starting from an existing subsystem decomposition, determines a decomposition with better metric values and fewer violations of design principles. Abdeen et al. [18] proposed a heuristic search-based approach for automatically optimizing (i.e., reducing) the dependencies between packages of a software system using simulated annealing. Their optimization technique is based on moving classes between the original packages. Taking inspiration from our previous work, Abdeen et al. have

recently extended their initial work to consider the remodularization task as a multi-objective optimization problem to improve existing packages structure while minimizing the modification amount on the original design. Using NSGA-II, this optimization approach aims at increasing the cohesion and reducing the coupling and cyclic connectivity of packages, by modifying as less as possible the existing packages organization. Praditwong et al. [32] have recently formulated the software clustering problem as a multi-objective optimization problem. Their work aim at maximizing the modularization quality measurement; minimizing the inter-package dependencies; increasing intra-package dependencies; maximizing the number of clusters having similar sizes; and minimizing the number of isolated clusters.

Most of the remodularization approaches in the literature are based on information derived only from structural metrics to modularize/restructure the original package organization. However, this is not enough to produce a semantically coherent design. The first attempt that addresses this problem was by Bavota et al. [51] who proposed an automated, single-objective, approach to split an existing package into smaller but more cohesive ones. The proposed approach analyzes the structural and semantic relationships between classes in a package identifying chains of strongly related classes. The identified chains are used to define new packages with higher cohesion than the original package. This work has been extended in [19], to propose an interactive multi-objective remodularization approach. The proposed Interactive Genetic Algorithms (IGAs) aims at integrating developer's knowledge in a remodularization task. Specifically, the proposed algorithm uses a fitness composed of automatically-evaluated factors (accounting for the modularization quality achieved by the solution) and a human-evaluated factor, penalizing cases where the way remodularization places components into modules is considered meaningless by the developer. One of the limitations of this approach is that, in each generation of the remodularization process, end users should analyze the suggested solution, class-by-class and package-by-package, and provide their feedback. User feedback can be either about classes which should stay together, or not, and/or about small/isolated clusters. This is not always profitable when we deal with industrial size software projects, and it need expert users to suitably drive the optimization process.

The semantic meaningfulness of the recommended restructuration is a fundamental issue when automatically modifying a software design. The first attempt to integrate the semantic coherence of when automatically modifying the software design was in [52]. Similarly to the related work, the remodularization approach uses the combination of semantic and structural information captured in the package and class levels to suggest more meaningful remodularization and better decide how to group together, split, or move, (or not) certain code elements. Furthermore, while automatic remodularization approaches proved to be very effective to increase cohesiveness and reduce coupling of software modules, they do not take into account the history of changes that provide a lot of information that is very useful in automating many software maintenance tasks. One of the characteristics of our approach is that it exploits the change history that represents an effective way to produce more meaningful remodularization. Another issue is that the majority of existing remodularization approaches considers only moving classes or grouping/splitting packages; however, none considered move methods/fields among classes in different packages. Hence, sometimes, it is enough to move only a method or a field between two classes in two different packages to reduce the dependency between them.

To illustrate some of the limitation of the related work, Figure 2 shows a concrete example extracted from GanttProject v1.10.2, a well-known Java open-source project management software. We consider a design fragment containing four packages *net.sourceforge.ganttproject*, *net.sourceforge.ganttproject.document*, *net.sourceforge.ganttproject.gui*, and *net.sourceforge.ganttproject.task*. The largest package in GanttProject v1.10.2 is *net.sourceforge.ganttproject* including more than 40 classes, comparing to all other packages, implementing several features in one package. All the twelve software engineers that we asked in our experiments agreed that *net.sourceforge.ganttproject* is a large package that monopolizes the behavior of a large part of the system.

We consider an example of a remodularization solution that consists of moving some classes from the package *net.sourceforge.ganttproject* to the package *net.sourceforge.ganttproject.document*. This operation can improve the modularization quality by reducing the number of classes/functionalities of the package *net.sourceforge.ganttproject*. However, from the semantics coherence standpoint, all the classes of

net.sourceforge.ganttproject.document are different from those implemented in *net.sourceforge.ganttproject* since they implement a feature to open streams to a project file. Based on the semantic and structural information, using respectively a vocabulary-based similarity, and cohesion/coupling, many other target packages are possible including *net.sourceforge.ganttproject.gui*, and *net.sourceforge.ganttproject.task*. These two packages have almost the same structure based on metrics such as number of classes and their semantic similarity is close to *net.sourceforge.ganttproject* using a vocabulary-based measure, or cohesion and coupling. On the other hand, from previous versions of GanttProject, we recorded that there are some classes (e.g. *TaskManagerImpl*) that have been moved from the package *net.sourceforge.ganttproject* to the package *net.sourceforge.ganttproject.task*. As a consequence, moving from the package *net.sourceforge.ganttproject* to the package *net.sourceforge.ganttproject.task* has higher correctness probability than moving classes between the remaining packages. Thus, some classes can be moved between these two packages such as *GanttTask*, *GanttTaskPropertiesBean* and *GanttTaskRelationship*. The direction of class movement should be taken into account while analyzing the history of changes to find similarities.

Based on these observations, we believe that it is important to consider additional objectives rather than using only structural metrics to improve the automation of software remodularization. However, in most of the existing remodularization work, semantic coherence, code changes, and development history are not considered. Thus, the remodularization process needs a manual inspection by the user to evaluate the meaningfulness/feasibility of proposed changes that mainly improve structural metrics. The inspection aims at verifying if these changes could produce semantic incoherence in the program design. For large systems, this manual inspection is complex, time-consuming and error-prone. Improving the packages structure, minimizing semantic incoherencies, reducing code changes, and keeping consistent with development change history may be conflicting. In some cases, improving the program modularization could provide a design that does not make sense semantically or could change radically the initial design. For these reasons, a good remodularization strategy needs to find a compromise between all of these objectives. In addition, moving classes and splitting packages are not enough code

changes to improve the remodularization of systems. These observations are at the origin of the work described in Chapter 4:

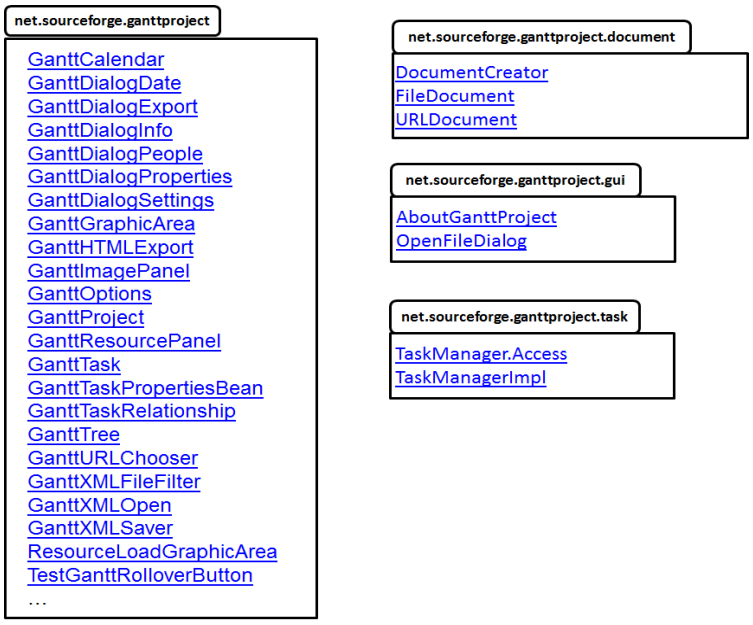


Figure 2.3. Motivating example extracted from GanttProject v1.10.2.

2.4 Software refactoring

Refactoring is defined as the process of improving code after it has been written by changing its internal structure without changing its external behavior. The idea is to reorganize variables, classes and methods to facilitate future adaptations and extensions and enhance comprehension. This reorganization is used to improve different aspects of software quality such as maintainability, extensibility, reusability, etc. Some modern Integrated Development Environments (IDEs), such as Eclipse, NetBeans, provide semi-automatic support for applying the most commonly used refactorings, e.g., move method, rename class, etc. However, automatically suggesting/deciding where and which refactorings to apply is still a real challenge in Software Engineering. In order to identify which parts of the source code need to be refactored, most existing work relies on the notion of code smells, also called design defects or anti-patterns.

The impact of code smells on software systems has been the subject of several studies over the past decade since their first introduction by Fowler [15]. He defined 22 code smells as structural

code flaws that may decrease the overall software quality and serve as indicators of potential issues related to software evolution and maintenance. To cope with these smells, Fowler has introduced a set of 72 Refactoring operations to fix code smells and thus improving the system overall design.

The detection process can either be manual, semi-automated or fully automated. Mäntylä et al. [53] provided an initial formalization of the code smells, in terms of metrics, based on analyzing Fowler's smells description, they studied the correlation between the smells and provided a classification according to their similarity. Mäntylä revealed that the manual detection of smells is dependent on the level of expertise of detection performers, which represents one of the main limitation of this approach. Marinescu et al. [54] presented an automated framework for smells identification using detection strategies which are defined by metric-based rules. Moha et al. [55] presented a semi-automated technique called DECOR. This framework allows subjects to manually suggest their own defects through their description with domain specific language, then the framework automatically searches for smells and visually reports any finding. Most of the above-mentioned work focus mainly on smells specification in order to improve their detection, for the correction step, their proposals were limited to guidance on how to manually manage these smells by suggesting refactoring recommendations according to detected smell's type.

Similarly to the detection's state of the art, refactoring techniques can be either manual, semi-automated or fully automated. Fowler [15] manually linked a set of suggested refactorings (Move Attribute, Extract Class, Move Method etc.) with each identified smell. Van Emden et al. [56] focused on the detection of two code smells related to the Java language and suggested their specific correction. Martin [57] has used patterns to cope with the poor system design in the presence of code smells. Mäntylä proposed refactoring solutions based on developers' opinions and driven by an automatic detection of structural anomalies at the source code level. Counsell et al. [58] refined Fowler's suggested refactorings by prioritizing some refactorings in the execution order. Piveta el al. [59] discussed when refactoring opportunities are eventually needed when detecting bad smells in aspect-oriented software. Meananeatra [60] presented another semi-automated heuristic for refactoring, it generates a graph or refactoring sequences that are

being refined using three main objectives to minimize the number of code smells, the number of refactorings and the number of refactored code elements.

JDeodorant [61] is an automated refactoring tool 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. Du Bois et al. [62] has investigated decreasing coupling and increasing cohesion metrics through the refactoring opportunities and use this to perform an optimal distribution of features over classes. They analyze how refactorings manipulate coupling and cohesion metrics, and how to identify refactoring opportunities that improve these metrics. However, this approach is limited to only some possible refactoring operations with few number of quality metrics. Murphy-Hill et al. [63] proposed several techniques and empirical studies to support refactoring activities. The authors proposed new tools to assist software engineers in applying refactoring such as selection assistant, box view, and refactoring annotation based on structural information and program analysis techniques. Recently, in [64] the authors have proposed a new refactoring tool called GhostFactor that allows the developer to transform code manually but check the correctness of the transformation automatically. BeneFactor [65] and WitchDoctor [66] can detect manual refactorings and then complete them automatically. Tahvildari et al. [67] also proposed a framework of object-oriented metrics used to suggest to the software engineer refactoring opportunities to improve the quality of an object-oriented legacy system. Dig et al. [68] proposed an interactive 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). Kataoka et al. [69] used invariants in the detection and extraction of source code fragments in need of refactoring.

Search-based refactoring represents fully automated refactoring driven by metaheuristic search and guided by software quality metrics and used subsequently to address the problem of automating design improvement. Seng et al. [50] propose a search-based technique that uses a genetic algorithm over refactoring sequences. The employed metrics are mainly related to various class level properties such as coupling, cohesion, complexity and stability. The approach was limited only to the use of one refactoring operation type, namely 'move method'. In contrast to O'Keeffe et al. [70], their fitness function is based on well-known measures of coupling

between program components. Both these approaches use weighted-sum to combine metrics into a fitness function, which is of practical value but is a questionable operation on ordinal metric values. Kessentini et al. [71] also propose a 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 code smells detected using a set of quality metrics. In [72] population-based direct approaches have been used in finding the local beam search for locating best refactoring solutions.

Harman and Tratt were the first to introduce the concept of Pareto optimality to search-based refactoring [6]. They use it to combine two metrics into a fitness function, namely CBO (coupling between objects) and SDMPC (standard deviation of methods per class) and demonstrate that it has several advantages over the weighted-sum approach. More recent work on multi-objective search-based refactoring is the work by Ouni et al. [52] who propose a multi-objective optimization approach to find the best sequence of refactorings using NSGA-II. The proposed approach is based on two objective functions, quality (proportion of corrected code smells) and code modification effort, to recommend a sequence of refactorings that provide the best trade-off between quality and effort.

Chapter 3: Model Transformation

3.1 Introduction

Model Transformation plays an important role in Model Driven Engineering (MDE) [7]. The research efforts by the MDE community have produced various languages and tools for automating transformations between different formalisms using mapping rules. These transformation rules can be implemented using general programming languages such as Java or C#; graph transformation languages like AGG [8] and VIATRA [9]; or specific languages such as ATL [73] and QVT [11]. Sometimes, transformations are based on invariants (pre-conditions and post-conditions specified in languages such as OCL [12]).

One major challenge is to automate transformations while preserving the quality of the produced models [7]. Thus, the main goal is to reduce the number of possible errors when defining transformation rules. These transformation errors have different causes such as transformation logic (rules) or source/target metamodels. Existing approaches and techniques have been successfully applied to transformation problems with a minimum number of errors. Especially at the model level, correctness is the gold standard characteristic of models: it is essential that the user understands exactly how the target model deviates from fidelity to the source model in order to be able to rely on any results. However, other important objectives are how to minimize the complexity of transformation rules (e.g. the number of rules, number of matching in the same rule) while maximizing the quality of target models to obtain well-designed ones. In fact, reducing rules complexity and improving target models quality are important to 1) make rules and target models easy to understand and evolve, 2) find transformation errors easily, and 3) generate optimal target models.

The majority of existing approaches formulate the transformation problem as a single-objective problem that maximizes rules correctness. In this case, the proposed transformation rules produce target models without errors. However, these rules are sometimes complex (e.g., size) and applying them may generate very large target model. For example, when complex transformation rules are available for mapping from dynamic Unified Modeling Language (UML) models to Colored Petri Nets (CPN), systematically applying them generally results in large PNs [13]. This could compromise the subsequent analysis tasks, which are generally limited by the number of the PNs' states. Obtaining large PNs is not usually related to the size of the source models but to the rules complexity [14]. In addition, it is important to take into consideration the quality of produced target models (e.g. maximizing good design practices by reducing the number of bad smells in a generated class diagram from a relational schema). Another category of approaches proposes an additional step to minimize complexity, using refactoring operations, after generating transformation rules [16]. However, it is a difficult and fastidious task to modify, evolve and improve the quality of already generated complex rules.

In this chapter, to overcome some of the above-mentioned limitations, we propose to alternatively view transformation rules generation as a multi-objective problem. We generate solutions matching the source metamodel elements to their equivalent target ones, taking in consideration two objectives: 1) minimizing rules' complexity and 2) maximizing target-models' quality. We start by randomly generating a set of rules, executing them on different source models to generate some target models, and then evaluate the quality of the proposed solution (rules). Of course, during the optimization process, we select only solutions ensuring full correctness (generating correct target models/rules). Correctness is the gold standard characteristic of models: it is essential that the user understands exactly how the target model deviates from fidelity to the source model in order to be able to rely on any results. To ensure the transformation correctness, we used a list of constraints to satisfy when generating target models. For the first objective, it calculates the number of rules and number of matching metamodels in each rule (one-to-one, many-to-one, etc.). For the second objective, we use a set of software quality metrics to evaluate the quality of generated target models. To search for solutions, we selected and adapted, from the existing Multi-Objective Evolutionary Algorithms (MOEAs), the Non-dominated Sorting Genetic Algorithm (NSGA-II) [21]. NSGA-II aims to find a set of

representative Pareto-optimal solutions in a single run. In our case, the evaluation of these solutions is based on the two mentioned conflicting criteria.

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

1. We introduce a new approach for model transformation using multi-objective techniques. Our proposal does not require to define rules manually, but only to input a set of source models and equivalent target models (without traceability links); it generates well-designed target models/rules without the need to refactor them; it takes in consideration the complexity of the generated rules; and it can be applied to any source or target metamodels (independent from source and target languages). However, different limitations are discussed in the discussion section.

2. We report the results of an evaluation of our approach; we used three different transformation mechanisms to evaluate our proposal: class diagrams to relational schemas and vice-versa; and sequence diagrams to colored Petri nets. The generated rules for both mechanisms achieved high-quality scores with a minimal set of rules.

3.2 Approach

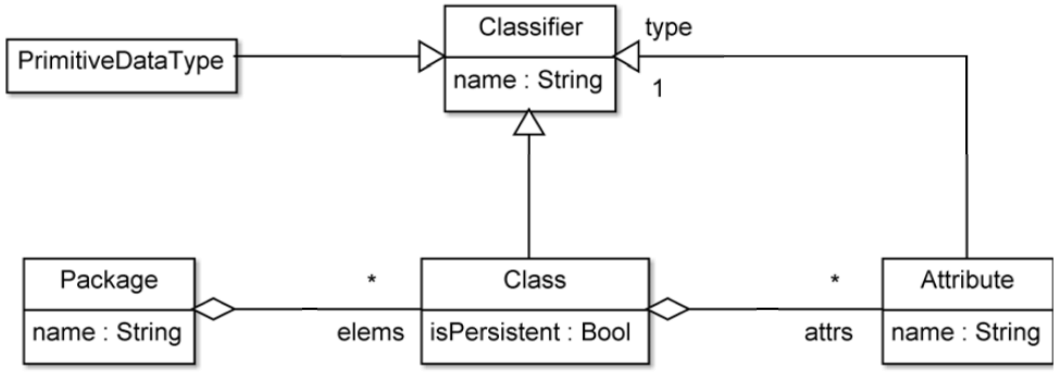
3.2.1 Automated model transformation

A model transformation mechanism takes as input a model to transform, the source model, and produces as output another model, the target model. The source and target models must conform to specific metamodels and, usually, relatively complex transformation rules are defined to ensure this.

We can illustrate this definition of the model transformation mechanism with the case of Class Diagram (CLD) to Relational Schema (RS) transformation. Our choice of CLD-to-RS transformation is motivated by the fact that it is well known and reasonably complex; this allows us to focus on describing the technical aspects of our approach. In the validation section, we show that our approach can also be applied to more complex transformations such as sequence diagrams to CPNs.

Figure 3.1.a shows a simplified metamodel of the UML class diagram, containing concepts like class, attribute, relationship between classes, etc. Figure 3.1.b shows a partial view of the relational schema metamodel, composed of table, column, attribute, etc. The transformation mechanism, based on rules, will then specify how the persistent classes, their attributes and their associations should be transformed into tables, columns and keys.

(a) Class diagram metamodel



(b) Relational schema metamodel

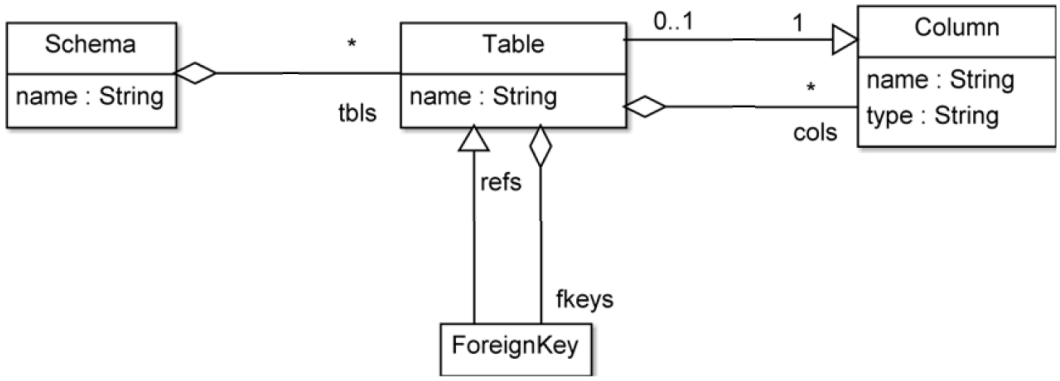
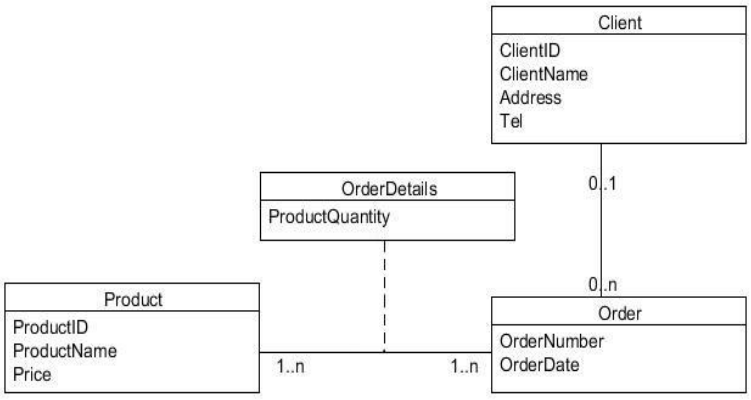


Figure 3.1. Class diagram and relational schema metamodels.

Figure 3.2 shows the example of a source model as class diagram containing 4 classes, and 2 association links. It shows also the associated target model expressed as a RS. 4 classes are mapped to tables (Client, Order, OrderDetails and Product). The two association links become foreign keys. Finally, attributes in subclasses are mapped into columns of the table derived from the parent class. The class diagram to relational schema transformation is used to illustrate our approach described in the rest of this chapter.

(a) Class diagram example



(b) Relational schema diagram example

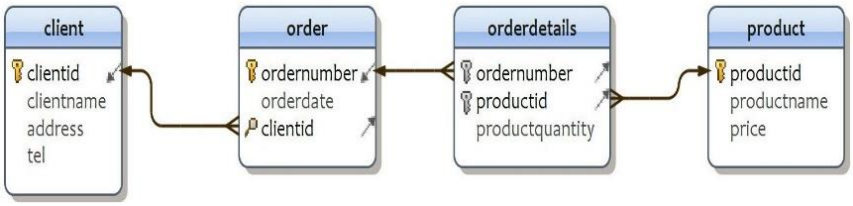


Figure 3.2. Class diagram (source model) with equivalent relational schema diagram (target model).

The general structure of our approach is introduced in Figure 3.3. The following two subsections give more details about our proposals. As described in Figure 3.3, the number of source models and the expected target ones is used to generate the transformation rules. In fact, our approach takes as inputs a set of source models with their equivalent target models, a list of quality metrics and another list of constraints (to ensure transformation correctness) and takes as controlling parameters a list of source and target metamodel elements. Our approach generates a set of rules as output.

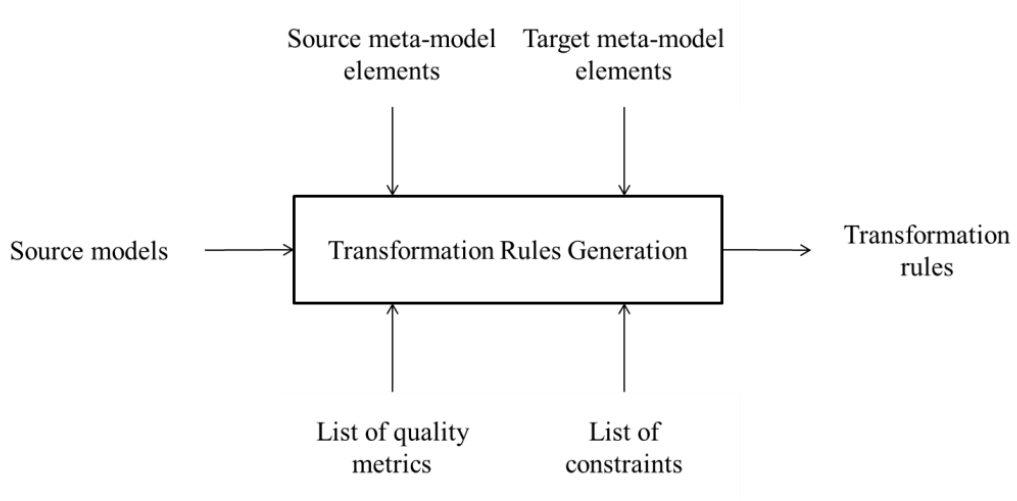


Figure 3.3. Overview of the approach: general architecture.

The rule generation process combines source and target metamodel elements within rule expressions. Some logical expressions (union OR; intersection AND) can be used to combine between metamodel elements. Consequently, a solution to the transformation problem is a set of rules that best transform the source models to target models within the satisfaction of the list of all transformation-constraints. For example, the following rule states that a class is transformed into a table with the same name having a primary key:

R1: IF Class(A) THEN Table(A) AND Column(idA, A, pk).

In this example of a rule, class, table and primary key column correspond to some elements extracted from the source and target metamodels. The first part of the rule contains only elements from the source metamodel. Consequently, the second part of the rule contains only elements from the target metamodel.

To ensure the transformation correctness when generating transformation rules, the idea is that the transformation of source models into target models is coupled with a contract consisting of pre- and post-conditions. Hence, the transformation is tested with a range of source models that satisfy the pre-conditions to ensure that it always yield target models that satisfy the post-conditions. If the transformation produces an output model that violates a post-condition, then the contract is not satisfied and the transformation needs to be corrected. The contract is defined

at the metamodel level and conditions are generally expressed in OCL. We used these constraints as input in our approach.

After ensuring the transformation correctness, our multi-objective optimization process uses two criteria to evaluate the generated solutions. The first criterion consists of minimizing the rules complexity by reducing the number of rules and the number of matching metamodels in each rule. The second criterion consists of maximizing the quality of generated target models based on different quality metrics. Quality metrics provide useful information that helps to assess the level of conformance of a software system to the desired quality such as *evolvability* and *reusability*. For instance, [16] propose different metrics to evaluate the quality of relational schemas such as: Depth of Relational Tree of a table T (DRT(T)) which is defined as the longest referential path between tables, from the table T to any other table in the schema database; Referential Degree of a table T (RD(T)) consists of the number of foreign keys in the table T; Percentage of complex columns PCC(T) metric of a table T; and Size of a Schema (SS) defined as the sum of the tables size (TS) in the schema.

We selected also a set of quality metrics that can be applied on class diagrams as target models. These metrics include: Number of associations (Naccoc): the total number of associations; Number of aggregations (Nagg): the total number of aggregation relationships; Number of dependencies (Ndep): the total number of dependency relationships; Number of generalizations (Ngen): the total number of generalization relationships (each parent-child pair in a generalization relationship); Number of aggregations hierarchies (NAggH): the total number of aggregation hierarchies; Number of generalization hierarchies (NGenH): the total number of generalization hierarchies; Maximum DIT (MaxDIT): the maximum of the DIT (Depth of Inheritance Tree) values for each class in a class diagram. The DIT value for a class within a generalization hierarchy is the longest path from the class to the root of the hierarchy; Number of attributes (NA): the total number of attributes; Number of methods (LOCMETHOD): the total number of methods; and Number of private attributes (NPRIVFIELD): number of private attributes in a specific class.

During the multi-objective optimization process, our approach combines randomly source and target metamodel elements within logical expressions (union OR; intersection AND) to create

rules. In this case, the number *n* of possible combinations is very large. The rule generation process consists of finding the best combination between *m* source metamodel elements and *k* target metamodel elements. In addition, a huge number of possibilities to execute the transformation rules exist (rules execution sequence). In this context, the number *NR* of possible combinations that have to be explored is given by: $NR = ((n+k)!)^m$

This value quickly becomes huge. Consequently, the rule generation process is a combinatorial optimization problem. As any solution must satisfy two criteria (complexity and quality), we propose to consider the search as a multi-objective optimization problem instead of a single-objective one. To this end, we propose an adaptation of the NSGA-II. This algorithm and its adaptation are described in the next sub-section.

3.2.2 NSGA-II adaptation

We adapted NSGA-II to the problem of generating transformation rules, taking in consideration both complexity and models quality dimensions. We consider each one of these criteria as a separate objective for NSGA-II. The pseudo-code for the algorithm is given in Algorithm 3.1.

Algorithm 3.1. High-level pseudo-code for NSGA-II adaptation to our problem.

Input: Source metamodel elements SMM - Target metamodel elements TMM - source models SM
Input: Correctness constraints CC - Quality metrics QM
Output: Near-optimal transformation rules
1: initial_population(P, Max_size)
2: P_0:= set_of(S)
3: S:= set_of(Rules:SMM:TMM)
4: SM:= Source_Models
5: t:=0
6: repeat
7: Q_t:= Gen_Operators(P_t)
8: R_t:=P_t U Q_t
9: for all $S_i \in R_t$ do
10: TM:= execute_rules(Rules, SM);
11: *Correctness*(S_i) := calculate_constraintsCoverage(SM, TM, CC);
12: if (calculate_constraintsCoverage(SM, TM, CC) ==1) then

```
13:              Complexity(Si) := calculate_Complexity(Rules);
14:              QualityModels(Si) := calculate_qualityModels(TM, QM);
15:           else
16:              Complexity(Si) == 0;
17:              QualityModels(Si) == 0;
18:           End if
19:       end for
20:       F:=fast-non-dominated-sort(Rt)
21:       Pt+1 :=Ø
22:       while |Pt+1|<Max_size
23:           Fi := crowding_distance_assignment(Fi)
24:           Pt+1 := Pt+1+Fi
25:       end while
26:       Pt+1 :=Pt+1[0:Max_size]
27:       t:=t+1;
28: until t=max_it
29: best_solution = First_front(Rt)
30: return best_solution
```

As Algorithm 3.1 shows, the algorithm takes as input a set of source and target metamodel elements, and a set of source models and its equivalent target ones. Lines 1–5 construct an initial population based on a specific representation, using the list of metamodels elements given at the inputs. Thus, the initial population stands for a set of possible transformation rules solutions that represent a set of source and target metamodel elements selected and combined randomly. Lines 6–30 encode the main NSGA-II loop whose goal is to make a population of candidate solutions evolves toward the best rules combination, i.e., the one that minimizes as much as possible the number of rules and matching metamodels in the same rule, and maximizes the target models quality by improving quality metrics values. During each iteration *t*, a child population Q_t is generated from a parent generation P_t (line 7) using genetic operators. Then, Q_t and P_t are assembled in order to create a global population R_t (line 8). After that, each solution S_i in the population R_t is evaluated using the two fitness functions, complexity and quality (lines 11-18):

- Complexity function (line 13) calculates the number of rules and matching metamodels in each rule.

- Quality function (line 14) represents the quality score of target models based on a combination of quality metrics.

These two functions take the value 0 if the transformation-correctness is not ensured. The correctness function (line 11) represents the percentage of source/target metamodel constraints that are satisfied by the proposed solution S_i. We consider during the optimization process only solutions that satisfy all correctness constraints.

Once quality and complexity are calculated, solutions are sorted in order to return a list of non-dominated fronts *F* (line 20). When the whole current population is sorted, the next population P_{t+1} will be created using solutions that are selected from sorted fronts *F* (lines 21-26). When two solutions are in the same front, i.e., same dominance, they are sorted by the crowding distance, a measure of density in the neighborhood of a solution. The algorithm terminates (line 28) when it achieves the termination criterion (maximum iteration number). The algorithm returns the best solutions that are extracted from the first front of the last iteration (line 29).

We give more details in the following sub-sections about the representation of solutions, genetic operators, and the fitness functions.

3.2.2.1 Solution representation

An individual is a set of declarative IF – THEN rules. To ease the manipulation of the source and target metamodels and their transformation, the metamodels are described using a set of predicates that correspond to the included element. For example, Figure 3.4 shows the rule interpretation of an individual containing two rules. So, the mapping between predicates *Class (A)* and *Table (A)* indicates that class *A* is transformed into a table with the same name.

Similarly, the mapping between *Association(1,n,1,n,N,A, B)* and *'Table(N) AND Column(idA, N,pfk) AND Column(idB, N,pfk)'* indicates that the association link *N* is transformed into a table with the same name containing two primary-foreign keys *pfk*, *idA* and *idB* which are primary keys respectively in tables *A* and *B*.

Rule 1: Class(A) THEN Table(A)

Rule 2: Association(1,n,1,n,N,A, B). THEN Table(N) AND Column(idA, N,pfk) AND Column(idB, N,pfk).

Figure 3.4. Rule interpretation of an individual.

Consequently, a transformation rule has the following structure:

IF "Combination of source metamodel elements" THEN "Combination of target metamodel elements"

As it is shown in Figure 3.4, the IF clause contains a combination of source metamodel elements. These elements are combined using logic operators (AND, OR). Consequently, THEN clauses highlight the equivalent target metamodel elements. Some other additional rules determine the sequence of applying transformation rules.

One of the most suitable computer representations of rules is based on the use of trees. In our case, the rule interpretation of an individual will be handled by a tree representation which is composed of two types of nodes: terminals and functions. The terminals (leaf nodes of a tree) correspond to source or target metamodel elements. The functions that can be used between these elements correspond to logical operators, which are Union (OR) and Intersection (AND).

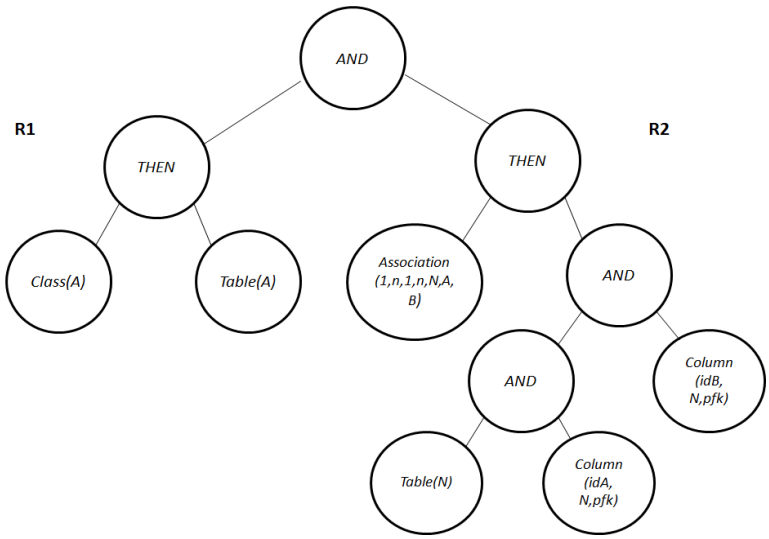


Figure 3.5. Solution representation.

Consequently, the rule interpretation of the individual of Figure 3.4 has the following tree representation of Figure 3.5. The sequence of applying the rules is determined randomly.

3.2.2.2 Generation of an initial population

To generate an initial population, we start by defining the maximum tree length including the number of nodes and levels. Because the individuals will evolve with different tree lengths (structures), we randomly assign for each one:

- One source or target metamodel element to each terminal node
- A logic operator (AND, OR) to each function node

3.2.2.3 Selection and genetic operations

Selection: There are many selection strategies where fittest individuals are allocated more copies in the next generations than the other ones. Thus, to guide the selection process, NSGA-II uses a comparison operator based on a calculation of the crowding distance to select potential individuals to construct a new population P_{t+1}. Furthermore, for our initial prototype, we used Stochastic Universal Sampling (SUS) to derive a child population Q_t from a parent population P_t, in which each individual's probability of selection is directly proportional to its relative overall fitness value (average score of the two fitness values) in the population. We use SUS to select elements from P_t that represents the best elements to be reproduced in the child population Q_t using genetic operators such as mutation and crossover.

Crossover: Two parent individuals are selected, and a sub-tree is picked on each one. Then, the crossover operator swaps the nodes and their relative sub-trees from one parent to the other. Each child thus combines information from both parents.

Figure 7 shows an example of the crossover process. In fact, the rule R1 and a rule R2 are combined to generate two new rules. The right sub-tree of R1 is swapped with the left sub-tree of R2.

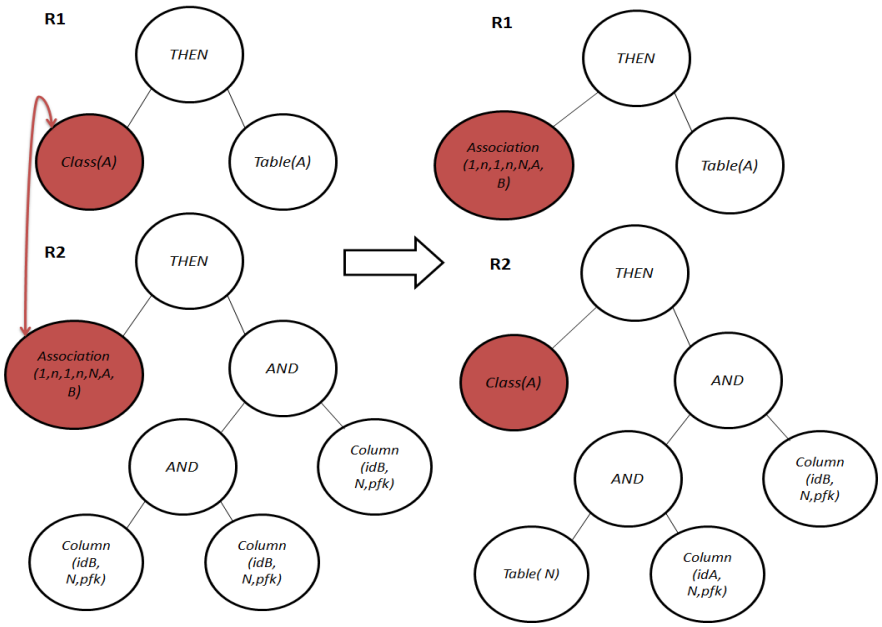


Figure 3.6. Crossover operator.

Mutation: The mutation operator can be applied either to function or terminal nodes. This operator can modify one or many nodes. Given a selected individual, the mutation operator first randomly selects a node in the tree representation of the individual. Then, if the selected node is a terminal (source or target metamodel element), it is replaced by another terminal (another metamodel element).

If the selected node is a function (AND operator, for example), it is replaced by a new function (i.e. AND becomes OR). If a tree mutation is to be carried out, the node and its subtrees are replaced by a new randomly generated sub-tree.

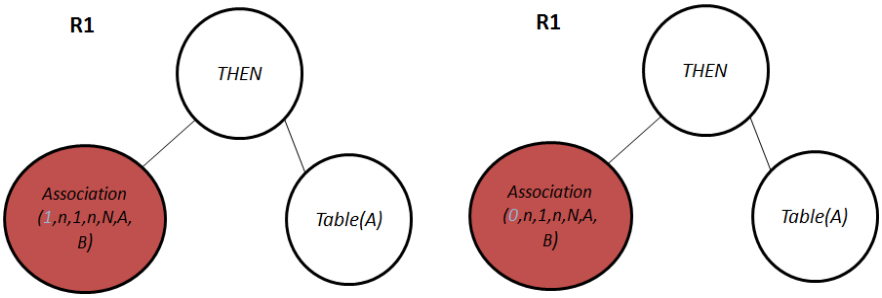


Figure 3.7. Mutation operator.

To illustrate the mutation process, consider again the example that corresponds to a candidate rule. Figure 8 illustrates the effect of a mutation that modifies the metamodel element association link in the rule R1. Thus, after applying the mutation operator the new rule R1 will be:

Rule 1: Association (0,n,0,n,N,A, B). THEN Table(A)

When the cross-over and mutation operators are executed, many pre and post-conditions should be satisfied to ensure that the rules modifications are valid. We specified these conditions for each metamodel element.

1.1.1 Multi-criteria evaluation

In the majority of existing work, the fitness function evaluates a generated solution by verifying its ability to ensure transformation correctness. In our case, in addition to ensuring transformation correctness we define two new fitness functions in our NSGA-II adaptation: (1) rules complexity and (2) target models quality.

To ensure transformation correctness, different constraints are defined manually including two parts: pre and post conditions. The pre-conditions constrain the set of valid models and the post-conditions declare a set of properties that can be expected in the output model. For example, a table should contain at least one primarily key or a foreign key should be a primary key in another table. As described in Figure 4, the transformation correctness constraints are verified before evaluating the rules-complexity and models-quality. If the proposed solution generates correct transformation rules then complexity and quality criterion can be evaluated. Thus, the correctness *C* parameter takes 1 if all constraints are satisfied otherwise 0:

$$C = \begin{cases} 1, \text{if all transformation correctness constraints are satisfied} \\ 0, \text{otherwise} \end{cases}$$

Complexity criterion: In our approach, we define the complexity function, to minimize, as the sum of number of generated rules and number of metamodel elements in each rule:

$$f_1 = c * (n + m) \qquad (3.1)$$

Where *n* is the number of rules to define and *m* is the number of metamodel elements in the same rule. Of course, the complexity function takes 0 if the transformation correctness is not ensured (c =0).

Quality criterion: The quality criterion is evaluated using the fitness function given in Equation (2). The quality value increases when the metrics values (m_i) are in the range of well-designed models thresholds ($m_{i,best\_minOrmax}$). This function, to minimize, returns a real value that represents the difference between good metrics values (expected) and those extracted from the generated target models. The choice of good metrics thresholds is based on our previous works in models quality improvements.

$$f_2 = \sum_{i=0}^{nbMetrics} Min\left(\left|m_{i,\min} - m_i\right|, \left|m_{i,\max} - m_i\right|\right) \qquad (3.2)$$

In this case, the quality of generated target models is maximized when f_2 is minimized. To illustrate the fitness function, we consider that a solution generated contains these four rules:

R1: IF Class(A) THEN Table(A) AND Column(idA,A,pk).

R2: IF Attribute(a,A) THEN Column(a,A,_).

R3: IF Association(0,1,0,n,N,A,B) THEN Column(idA,B,fk).

R4: IF Association(1,n,1,n,N,A, B) THEN Table(N) AND Column(idA,idB,N,pfk).

To evaluate this solution, let's consider the class diagram source model of Figure 3.2.a. After executing this set of four rules, we obtain the relational schema target model of Figure 3.2.b. We consider, for example, that the correctness is ensured based on two constraints: C1) each table should contain, at least, one primary key; and C2) a foreign key in table *A* should be a primary key in another table *B*. To evaluate the design quality of target models, we are using two quality metrics:

- Referential Degree of a table T (RD(T)) consists of the number of foreign keys in the schema: $m_{1,best}$ = (min = 1; max = 3)

- Size of a Schema (SS) defined as the sum of the tables size (TS) in the schema: $m_{2,best}$ = (min = 3; max = 5)

In such scenario, the parameters of the complexity fitness function take the following values: *c=1* since both correctness constraints are satisfied by the target model; *n=4* which corresponds to the number of rules; *m=3+2+2+3 =10 (number of matching metamodels).* Thus, the complexity score of the generated solution is:

$$f_1 = 1 * (4 + 10) = 14$$

Regarding the quality dimension, based on Figure 3.2.b RD and SS take, respectively, the value 3 (0+1+2+0) and 4. Thus, the quality fitness function is defined as follow:

$$f_2 = Min(|1-3|, |3-3|) + Min(|3-4|, |5-4|) = 0 + 1 = 1$$

3.3 Validation

To evaluate the feasibility of our approach, we conducted an experiment with three transformation mechanisms. We start by presenting our research questions. Then, we describe and discuss the obtained results.

3.3.1 Research Questions

Our study addresses two research questions, which are defined here. We also explain how our experiments are designed to address them. The goal of the study is to evaluate the efficiency of our approach for generating correct transformation rules while minimizing the rules-complexity and maximizing the quality of generated target models. The three research questions are then:

- RQ1: To what extent can the proposed approach minimize rules complexity?
- RQ2: To what extent can the proposed approach maximize the quality of generated target models?
- RQ3: To what extent can the proposed multi-objective approach perform comparing to mono-objective search algorithms?

To answer RQ1, we compared the complexity of the generated rules with expected ones defined manually: number of rules and number of elements in each rule.

To answer RQ2, the transformation result is checked for quality using two methods: 1) we calculate the dissimilarity between reference metrics threshold and those related to generated target models, and 2) we evaluate the variation in terms of size between generated target models using NSGA-II and those provided manually by experts.

To answer RQ3, we implemented a mono-objective genetic algorithm where the goal is to generate a minimal set of correct transformation rules (one objective is used which is the complexity). Then, we compared the results to those generated by our NSGA-II approach based on complexity and quality criterions.

3.3.2 Setting

To evaluate the feasibility of our approach, we conducted an experiment on generating rules for: class diagram (CLD) to relational schema (RS) and vice-versa (RS to CLD), and sequence diagram (SD) to colored Petri nets (CPN). We used twelve large class-diagrams with their equivalent relational schemas. The examples were provided by an industrial partner. The size of the CLDs varied from twenty-eight to ninety-two model elements, with an average of fifty-eight. In addition, we collected the transformations of ten sequence diagrams to sequence diagrams from the Internet and textbooks. We ensured by manual inspection that all the transformations are valid. The size of the SDs varied from sixteen to fifty-seven constructs, with an average of thirty-six. The ten sequence diagrams contained many complex fragments: *loop*, *alt*, *opt*, *par*, *region*, *neg* and *ref*.

As described in the previous section, we selected a set of twelve quality metrics for CLD, nine for RS and two for CPN (number of places and transitions). Based on our previous work, we define the thresholds range for each of those metrics. As described previously, we implemented a set of constraints to ensure the correctness of generated target models during the optimization process.

1.2 Results and Discussions

In this subsection we present the answer to each research question, in turn, indicating how the results answer each.

Figure 3.8 shows the rules complexity and target models quality for all the three transformation mechanisms based on the two fitness function values. These two fitness functions to minimize correspond to 1) **Complexity:** the number of rules and matching metamodels in each rule; and 2) **Dissimilarity:** the difference between the solution's calculated metrics values and the reference metrics values; so, decreasing the dissimilarity will increase the solution's quality. For all the transformation mechanisms, different solutions generate well-designed target models with a minimal set of rules.

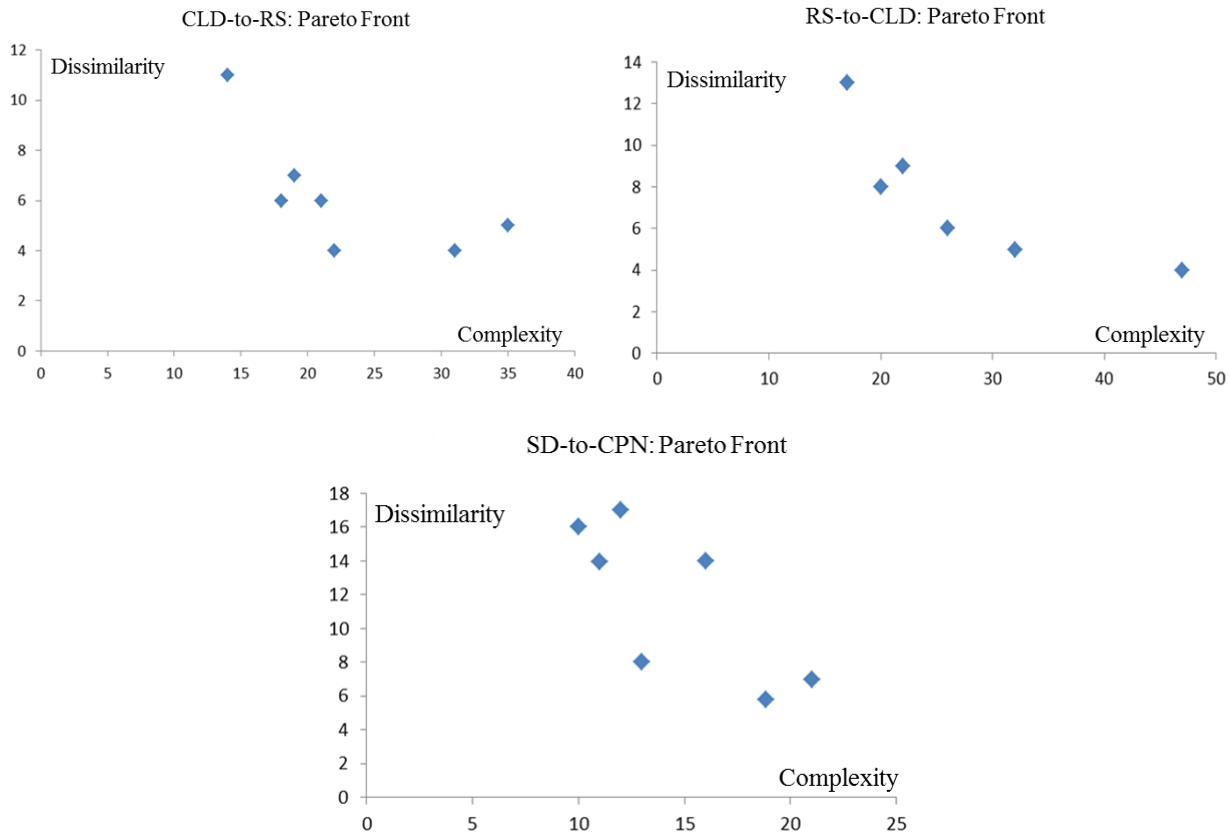


Figure 3.8. Results of the last iteration for the 3 transformations.

As shown in Figure 3.8, NSGA-II converges to Pareto-optimal solutions that are considered as good compromises between quality and complexity. In this figure, each point is a solution with

the complexity score represented in the x-axis and the dissimilarity score (deviation from reference metrics threshold) in the y-axis. The best solutions exist in the middle representing the Pareto-front that minimizes dissimilarity with reference metrics threshold and the rules complexity. The user can choose a solution from this front depending on his preferences in terms of compromise. However, at least for our validation, we need to have only one best solution that will be suggested by our approach. To this end and in order to fully automate our approach, we propose to extract and suggest only one best solution from the returned set of solutions. Equation (3.3) is used to choose the solution that corresponds to the best compromise between Quality and Complexity. Hence, we select the nearest solution to the ideal one in terms of Euclidian distance:

$$bestSol = \underset{i=0}{\overset{n}{Min}} \left(\sqrt{(Dissimilar\ ity[i])^2 + (Complexity[i])^2} \right) \qquad (3.3)$$

where n is the number of solutions in the Pareto front returned by NSGA-II. Since the two objectives of quality and complexity are conflicting/contradicting, the results of Figure 3.8 confirm that a solution which scores better in complexity is better than any other solution which is of lower quality.

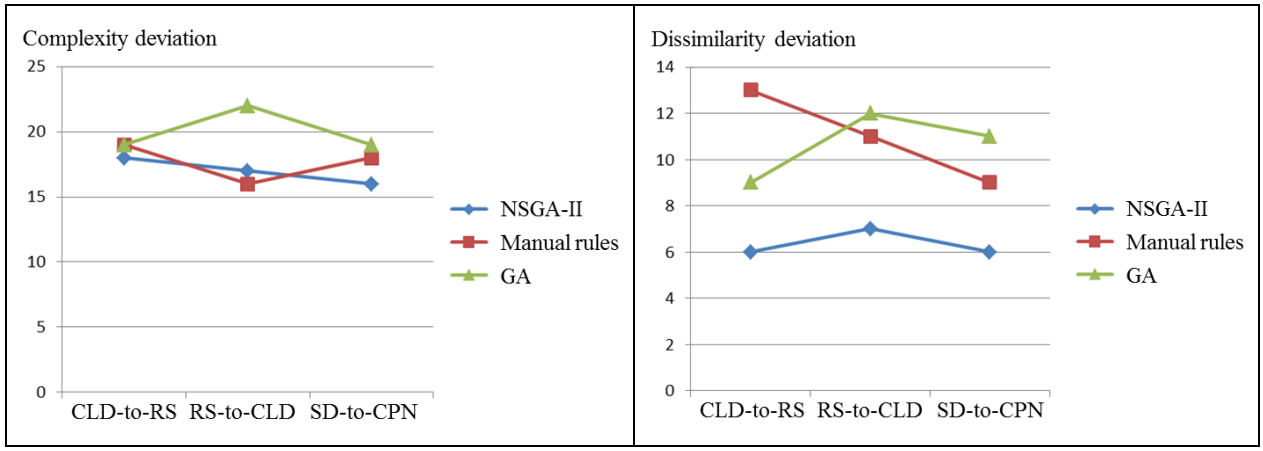


Figure 3.9. Complexity and Dissimilarity comparison between NSGA-II, manually defined rules and a mono-objective Genetic Algorithm (GA).

As described in Figure 3.9, the majority of proposed transformation rules generate good quality of target models with minimal complexity comparing to those provided manually by experts or a mono-objective genetic algorithm. For all the three transformation mechanisms, the dissimilarity of generated target models using NSGA-II is lower than those generated by the manual and genetic algorithm methods which means that NSGA-II's quality is much better than the other

transformation mechanisms. In fact, when experts write rules manually they did not take into consideration, in general, the quality of produced models but only the correctness. Since the mono-objective algorithm considers only correctness when generating transformation rules thus it is evident that NSGA-II performs better in terms of target models quality. The generated rules using NSGA-II are less complex than those generated by an expert for all the three transformation mechanisms. In fact, experts ensure that the rules are correct as the main goal. However, GA provides less complex rules for CLD-to-RS and RS-to-CLD than our NSGA-II algorithm. This is can be explained by the reason that these two transformation mechanisms are not complex. However, with more complex transformation mechanisms, such as SD-to-CPN, it is difficult to obtain a minimal set of rules without specifying complexity as a separate objective in addition to correctness. In addition, based on NSGA-II algorithm we can sacrifice a small complexity decrease to improve the quality of generated target models.

Figure 3.9 shows that, in general, we generate, approximately, the same number of rules for all transformation mechanisms. The number of generated rules is comparable to those provided by our expert in terms of number of matching metamodels. The different generated rules are verified manually and we did not found any errors.

As described in Figure 3.9, the average of quality deviation, from reference metrics values, for all transformed source models is low. This is confirming the good quality of generated target models. After a manual investigation of the results, we found that most of the quality deviations are due to the bad quality of source models to transform. In conclusion, our approach produces good refactoring suggestions both from the point of views of complexity and target models quality.

The generated rules might vary depending on search space exploration since solutions are randomly generated, though guided by a meta-heuristic. To ensure that our results are relatively stable, we compared the results of multiple executions for NSGA-II as shown in Figure 11. We, consequently, believe that our technique is stable since the quality and complexity scores are approximately the same for different executions (each fold).

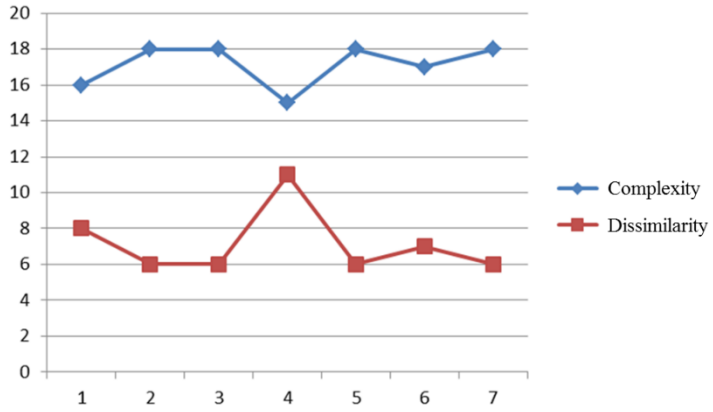


Figure 3.10. An example of seven executions on CLD-to-RS (best solutions).

Since we viewed the maintainability defects correction problem as a combinatorial problem addressed with heuristic search, it is important to contrast the results with the execution time. We executed our algorithm on a standard desktop computer (i7 CPU running at 4 GHz with 4GB of RAM). The execution time for finding the optimal rules with a number of iterations (stopping criteria) fixed to 1000 was less than one hour. This indicates that our approach is reasonably scalable from the performance standpoint. However, the execution time depends on the source and target metamodels.

As described in Table 3.1, we used the CPN-SD transformation mechanism to compare the quality of the generated CPNs using a mono-objective genetic algorithm (minimizing only rules complexity) and multi-objective approach.

Table 3.1. Complexity comparison.

CPN size (mono-objective)	CPN size (multi-objective)	Variation
13	11	15%
22	19	14%
24	24	0%
31	26	17%
36	33	9%
39	29	25%
44	37	16%
52	43	18%
54	46	15%
Average variation		13%

When developing our approach, we conjectured that the multi-objective approach produces CPNs less complex/better quality (in size for example) than the one obtained by a mono-

objective approach. Table 3.1 compares the obtained CPN sizes by using both approaches for the 10 source models to transform.

The size of a CPN is defined by the number of elements. In all cases, a reduction in size occurs when using our multi-objective approach, with an average reduction of 13% in comparison with mono-objective. The obtained results confirm our assumption that systematic application of rules using a mono-objective approach results in larger CPNs.

3.4 Conclusion

In this chapter, we introduced a new multi-objective approach for generating model transformation rules. Our algorithm starts by randomly generating a set of rules, executing them to generate some target models, and then evaluates the complexity by reducing the number of generated rules and the quality of generated target models based on some quality-metric thresholds. Our approach differs from rule-based transformation approaches as it does not require writing rules. To our best knowledge, our proposal represents the first work that uses multi-objective techniques to automated model transformations. It also differs from existing by example approaches by the fact that no traceability links are needed in the examples.

We have evaluated our approach on three transformation mechanisms. The experimental results indicate that the quality of derived target models is comparable and sometimes better than those defined by experts in the base of examples in terms of correctness with a minimal set of rules.

Finally, we discussed some limitations and open research directions related to our proposal. First, all our performance contribution depends on the availability of examples, which could be difficult to collect. However, as we have shown in the experiments, only a few examples are needed to obtain good results. Second, due to the nature of our solution, i.e., an optimization technique, the process could be time-consuming for large models. Furthermore, as we use heuristic algorithms, different executions for the same input could lead to different outputs. This can be a disadvantage for some model-driven engineering applications, e.g. when the model transformation is a deterministic process and the generated target model is unique. Nevertheless,

having different and equivalent output models is close to what happens in the real world where different experts may propose different target models.

Different future work directions can be explored. The application of new search-based techniques like the artificial immune system to model evolution or model refactoring is challenging. We are working on an extension of our first contribution about exogenous transformation by example. The idea is to generate transformation rules from examples using heuristic search. Our approach starts by randomly generating a set of rules, executing them to generate some target models. Then, it evaluates the quality of the proposed solution (rules) by comparing the generated target models to the expected ones in the base of examples. In this case, the search space is large and heuristic-search is needed.

We are actually working to extend our proposal to other problems. A new technique for predicting “buggy” changes, when modifying an existing version of a model, can be proposed. The idea is to classify the changes as clean or not. The Change classification determines whether a new model change is more similar to prior “buggy” or clean changes in the base of examples. In this manner, change classification can predict the existence of “bugs” in models changes.

Furthermore, we are working on transformation composition using examples. We propose a solution based on a music-inspired approach. We draw an analogy between the transformation composition process and finding the best harmony when composing music. Say, for example, that we have a transformation mechanism M1 that transforms formalism T1 into T2, but the meta-model of T2 evolved into T3, after deleting or adding elements. We want to generate new transformation rules that transform T1 into T3. The idea is to compose two transformation mechanisms T1 to T2 and T2 to T3. To this end, we propose to view transformation rules generation as an optimization problem where rules are automatically derived from available examples. Each example corresponds to a source model and its corresponding target model, without transformation traces from T1 to T3. Our approach starts by composing a set of rules (T1 to T2 and T2 to T3), executing them to generate some target models, and then evaluating the quality of the proposed solution (rules) by comparing the generated target models and the expected ones in the base of examples.

Chapter 4: Software Remodularization

4.1 Introduction

Large software systems evolve and become complex quickly, fault-prone and difficult to maintain [48]. In fact, most of the changes during the evolution of systems such as introducing new features or fixing bugs are conducted, in general, within strict deadlines. As a consequence, these code changes can have a negative impact on the quality of systems design such as the distribution of the classes in packages. To address this issue, one of the widely used techniques is software remodularization, called also software restructuring, which improves the existing decomposition of systems.

There has been much work on different techniques and tools for software remodularization [17] [47] [74] [75] [76] [77]. Most of these studies addressed the problem of clustering by finding the best decomposition of a system in terms of modules and not by improving existing modularizations. In both categories, cohesion and coupling are the main metrics used to improve the quality of existing packages (e.g. modules) by determining which classes need to be grouped in a package. In this chapter, we focus on restructuring software design and not on the decomposition of systems to generate an initial coherent object oriented design.

The majority of existing contributions have formulated the restructuring problem as a single-objective problem where the goal is to improve the cohesion and coupling of packages [18] [32] [51]. Even though most of the existing approaches are powerful enough to provide remodularization solutions, some issues still need to be addressed.

One of the most important issues is the semantic coherence of the design. The restructured program could improve structural metrics but become semantically incoherent. In this case, the

design will become difficult to understand since classes are placed in wrong packages to improve the structure in terms of cohesion and coupling. Also, the number of code changes is not considered when suggesting remodularization solutions; the only aim is to improve the structure of packages independently of the cost of code changes.

However, in real-world scenarios, developers prefer, in general, remodularization solutions that improve the structure with a minimum number of changes. It is important to minimize code changes to help developers in understanding the design after applying suggested changes.

Existing remodularization studies are also limited to few types of changes mainly *move class* and *split packages* [18] [19] [51] [78] [79]. However, refactoring at the class and method levels can improve remodularization solutions such as by moving methods between classes located in different packages. The use of development history can be an efficient aid when proposing remodularization solutions [80]. For example, packages that were extensively modified in the past may have a high probability of being also changed in the future. Moreover, the packages to modify can be similar to some patterns that can be found in the development history, thus, developers can easily recognize and adapt them.

In this chapter, we propose a many-objective search-based approach to address the above-mentioned limitations. Search-based software engineering is suitable for the software remodularization problem since the goal is to find the best sequence of operations that can lead to a better-remodularized system. The number of combinations to explore is high, leading to a huge and complex search space. Our many-objective search-based software engineering approach aims at finding the remodularization solution that: 1) Improve the structure of packages by optimizing some metrics such as number of classes per package, number of packages, coupling and cohesion; 2) Improve the semantic coherence of the restructured program. We combine two heuristics to estimate the semantic proximity between packages when moving elements between them (vocabulary similarity, and dependencies between extracted classes from call graphs) and some semantic/syntactic heuristics depending on the change type; 3) Minimize code changes. Compared to existing remodularization studies, we consider new changes that can be related to the package, class and method levels; and 4) Maximize the consistency with development change history. To better guide the search process, recorded code changes that are

applied in the past in similar contexts are considered. We evaluate if similar changes are applied in previous versions of the packages that will be modified by the suggested remodularization solution.

The number of objectives to consider in our problem formulation is high (more than three objectives); such problems are termed many-objective. In this context, the use of traditional multi-objective techniques, e.g., NSGA-II [21], widely used in Search-Based Software Engineering (SBSE) [6] [81], is clearly not sufficient like in our case for the problem of software remodularization. There is a growing need for SBSE approaches that address software engineering problems where a large number of objectives are to be optimized. Recent work in optimization has proposed several solution approaches to tackle many-objective optimization problems [20] [82] [83] using e.g., objective reduction, new preference ordering relations and decomposition. However, these techniques have not yet been widely explored in SBSE [5]. To the best of our knowledge and based on recent SBSE surveys [84], only one work exists proposed by [36] [37] that uses a many-objective approach, IBEA (Indicator-Based Evolutionary Algorithm) [85], to address the problem of software product line creation. However, the number of considered objectives is limited to five.

We propose a scalable search-based software engineering approach based on NSGA-III [20] where there are seven different objectives to optimize. Thus, in our approach, automated remodularization solutions will be evaluated using a set of seven measures as described above. The basic framework remains similar to the original NSGA-II algorithm, with significant changes in its selection mechanism. This work represents one of the first real-world applications of NSGA-III and the first scalable work that supports the use of seven objectives to address and improve software remodularization.

We evaluated our approach on four open source systems and one industrial system provided by our industrial partner Ford Motor Company. We report the results on the efficiency and effectiveness of our approach, compared to the state of the art remodularization approaches [18] [78] [79]. Our results indicate that our approach significantly outperforms, in average, existing approaches in terms of improving the structure, reducing the number code changes, and semantics preservation.

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

1. A novel formulation of the remodularization problem as a many-objective problem that considers several objectives such as structural improvement, semantic coherence, number of changes and consistency with the history of changes.

2. We consider the use of new operations, comparing to existing remodularization studies including move method, extract class and merge packages.

3. This chapter reports the results of an empirical study of our many-objective technique compared to different existing approaches. The obtained results provide evidence to support the claim that our proposal is, in average, more efficient than existing techniques based on a benchmark of four large open source systems and one industrial project.

4. The qualitative evaluation of the results by software engineers at Ford Motor Company and also graduate students confirms the usefulness the suggested remodularization solutions.

4.2 Approach

4.2.1 Approach Overview

Our approach aims at exploring a huge search space to find a set of remodularization solutions on the Pareto front such that in order to optimize any objective further will result in sub-optimizing one or more additional objectives. These remodularization solutions are a sequence of change operations, to restructure packages. The search space is determined not only by the number of possible change combinations but also by the order in which they are applied. A heuristic-based optimization method is used to generate remodularization solutions. We have 7 objectives to optimize: 1) minimize the number of classes per package; 2) minimize the number of packages; 3) maximize package cohesion; 4) minimize package coupling; 5) minimize the number of semantic errors by preserving the way classes are semantically grouped and connected together; 6) minimize code changes needed to apply remodularization solution; and 7) maximize

the consistency with development change history. We consider the remodularization task as a many-objective optimization problem instead of a single-objective one using the new many-objective non-dominated sorting genetic algorithm (NSGA-III) that will be described in Section 4.

Figure 4.1. Approach overview.

The general structure of our approach is sketched in Fig. 3. It takes as input the source code of the program to be restructured, a list of possible Remodularization Operations (ROs) that can be applied, a set of semantic and structural measures, and a history of applied changes to previous versions of the system. Our approach generates as output the optimal sequence of operations, selected from a list of possible ones that improve the structure of packages, minimize code changes needed to apply the remodularization solution, preserve the semantics coherence, and maximize the consistency with development change history. In the following, we describe the formal formulation of these different remodularization objectives to optimize.

4.2.1.1 Remodularization Objectives

We describe in this section the 7 objectives to optimize in our many-objective adaptation to the software remodularization problem.

Structure. Four conflicting objectives are related to improving the structure of packages: 1) number of classes per package (to minimize); 2) number of packages in the system (to minimize); 3) cohesion (to maximize); it corresponds to the number of intra-edges (calls between classes in the same package) as described in Section 2; and 4) coupling (to minimize); it corresponds to the number of inter-edges (class between classes in different packages).

Number of code changes. Table 2 describes the types of ROs that are considered by our approach: Move method, Extract class, Move class, Merge packages, and Extract/Split package. Existing remodularization studies are limited to only two operation types: move class and

split/extract package. We believe that these two operations are not enough to generate good remodularization solutions. In fact, sometimes only part of the class should be moved to another package (e.g. methods) and not the whole class. To apply a remodularization operation we need to specify which actors, i.e., code fragments, are involved in this operation and which roles they play when performing the change. As illustrated in Table 2, an actor can be a package, class, or method and we specify for each operation the involved actors and their roles. It is important to minimize the number of suggested operations in the remodularization solution since the designer can have some preferences regarding the percentage of deviation with the initial program modularization. In addition, most of the developers prefer solutions that minimize the number of changes applied to their design [86].

Table 4.1. Types of remodularization operations.

Type of the operation	Actors	Roles
Move method	class	source class, target class
	method	moved method
Extract class	class	source class, new class
	method	moved methods
Move class	package	source package, target package
	class	moved class
Merge packages	package	source package, target package
Extract/Split package	package	source package, target package
	class	moved class

Similarity with the history of code changes. The idea is to encourage the use of ROs that are similar to those applied to the same code fragments (packages) in the past. To calculate the similarity score between a proposed operation and a recorded code change, we use the following function:

$$SimilarityHistory(RO) = \sum_{j=1}^{n} w_j \qquad (4.1)$$

where n is the number of recorded operations applied to the system in the past, and w_j is a change weight that reflects the similarity between the suggested RO and the recorded code change j. The weight w_j is computed as described in Table 3.

Table 4.2. Similarity scores between remodularization operations applied to similar code fragments.

	Move method	**Extract class**	**Move class**	**Merge packages**	**Extract/Split package**
Move method	$w_j = 2$	$w_j = 1$	$w_j = 1$	$w_j = 0$	$w_j = 0$
Extract class	$w_j = 1$	$w_j = 2$	$w_j = 0$	$w_j = 0$	$w_j = 0$
Move class	$w_j = 1$	$wj = 0$	$w_j = 2$	$w_j = 1$	$w_j = 1$
Merge packages	$w_j = 0$	$wj = 0$	$w_j = 1$	$w_j = 2$	$w_j = 0$
Extract/Split package	$w_j = 0$	$wj = 0$	$w_j = 1$	$w_j = 0$	$w_j = 2$

Semantics. Most of the ROs are simple to implement and it is almost trivial to show that they preserve the behavior. However, until now there is no consensual way to investigate whether a code change can preserve the semantic coherence of the original program/design. To preserve the semantics design, some constraints should be satisfied to ensure the correctness of the applied operations. We distinguish between two kinds of constraints: structural constraints and semantic constraints. Structural constraints were extensively investigated in the literature. Opdyke, for example, defined in [87] a set of pre and post-conditions for a large list of operations to ensure the structural consistency. Developers should check manually all code elements (packages, classes, methods and fields) related to the operation to inspect the semantic relationship between them. We formulate semantics constraints using different measures in which we describe the concepts from a perspective that helps in automating the remodularization task:

Vocabulary-based similarity (VS)

This kind of similarity is interesting to consider when moving methods, or classes or merging packages or extracting packages. For example, when a class has to be moved from one package to another, the operation would make sense if both code elements (source class and target packages) use similar vocabularies [52] [88]. The vocabulary could be used as an indicator of the semantic similarity between different code elements that are involved when performing a remodularization operation. We start from the assumption that the vocabulary of an actor is borrowed from the domain terminology and, therefore, could be used to determine which part of the domain semantics is encoded by an actor. Thus, two code elements could be semantically similar if they use similar vocabularies.

The vocabulary could be extracted from the names of packages, classes, methods, fields, variables, parameters and types. Tokenization is performed using the Camel Case Splitter [88] 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, commits information, and documentation. We calculate the semantic similarity between code elements using information retrieval-based techniques (e.g., cosine similarity). The following equation calculates the cosine similarity between two code elements. 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 code elements *c1* and *c2* is determined as follows:

$$Sim(c1,c2) = \cos(\vec{c1},\vec{c2}) = \frac{\vec{c1}.\vec{c2}}{\| \vec{c1} \| * \| \vec{c2} \|} = \frac{\sum_{i=1}^{n} (w_{i,1} * w_{i,2})}{\sqrt{\sum_{i=1}^{n} (w_{i,1})^2} \sqrt{\sum_{i=1}^{n} (w_{i,2})^2}} \in [0,1] \qquad (4.2)$$

where $\vec{c1} = (w_{1,1},..., w_{n,1})$ is the term vector corresponding to actor *c1* and $\vec{c2} = (w_{1,2},..., w_{n,2})$ is the term vector corresponding to *c2*. The weights $w_{i,j}$ 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 [52] to determine the vocabulary and represent the code elements as term vectors.

Dependency-based similarity (DS)

We approximate domain semantics closeness between code elements starting from their mutual dependencies. The intuition is that code elements that are strongly connected (i.e., having dependency links) are semantically related. As a consequence, ROs requiring semantic closeness between involved code elements are likely to be successful when these code elements are strongly connected. We consider two types of dependency links:

1) *Shared method calls (SMC)* that can be captured from call graphs derived from the whole program using CHA (Class Hierarchy Analysis) [89]. A call graph is a directed graph which represents the different calls (call in and call out) among all methods of the entire program. Nodes represent methods, and edges represent calls between these methods. CHA is a basic call graph that considers class hierarchy information, e.g, for a call *c.m(...)* assume that any *m(...)* is

reachable that is declared in a supertype of the declared type of *c*. For a pair of code elements, shared calls are captured through this graph by identifying shared neighbors of nodes related to each actor. We consider both, shared call-out and shared call-in. The following equations are used to measure respectively the shared call-out and the shared call-in between two code elements c_1 and c_2 (two classes, for example). A shared method call is defined as the average of shared call-in and call-out.

$$\text{sharedCall Out}(c1, c2) = \frac{|\text{callOut}(c1) \cap \text{callOut}(c2)|}{|\text{callOut}(c1) \cup \text{callOut}(c2)|} \in [0,1] \quad (4.3)$$

$$\text{sharedCall In}(c1, c2) = \frac{|\text{callIn}(c1) \cap \text{callIn}(c2)|}{|\text{callIn}(c1) \cup \text{callIn}(c2)|} \in [0,1] \quad (4.4)$$

2) *Shared field access (SFA)* can be calculated by capturing all field references that occur using static analysis to identify dependencies based on field accesses (read or modify). We assume that two software elements are semantically related if they read or modify the same fields. The rate of shared fields (read or modified) between two code elements c_1 and c_2 is calculated according to the following equation. In this equation, $fieldRW(c_i)$ computes the number of fields that may be read or modified by each method of the actor c_i. Thus, by applying a suitable static program analysis to the whole method body, all field references that occur could be easily computed.

$$\text{sharedFiel dsRW}(c1, c2) = \frac{|\text{fieldRW}(c1) \cap \text{fieldRW}(c2)|}{|\text{fieldRW}(c1) \cup \text{fieldRW}(c2)|} \in [0,1] \quad (4.5)$$

Cohesion-based dependency (CD)

The cohesion-based similarity that we propose for software remodularization is mainly used by the extract-class, merge-package and extract-package operations. It is defined to find a cohesive set of classes and methods to be moved to the newly extracted class or package. A new class or package can be extracted from a source class or package by moving a set of strongly related (cohesive) classes and methods from the original class or package to the new class or package. Extracting this set will improve the cohesion of the original package or class and minimize the coupling with the new package/class. Applying the *Extract Package/Class* or *Merge Packages* operation on a specific package/class will result in this class being split (or merged) into

classes/packages. We need to calculate the semantic similarity between the elements in the original package/class to decide how to split or merge the original packages/classes.

We use vocabulary-based similarity and dependency-based similarity to find the cohesive set of code elements. For *move method*, the cohesion matrix is composed of the fields of the source method as lines, and the fields and methods of the target class as columns. For *extract class*, the lines are the fields and methods of the source class and the columns are the methods of the extracted class. For *move class*, the lines of the cohesion matrix are composed of the fields, methods of the class to move, and the columns are the fields and methods of the classes of the target package. Regarding *merge package* and *extract/split package* operations, the lines correspond to the fields and methods of the classes of the source package and the columns are the fields and methods of the classes of the target package. We calculate the similarity between each pair of elements in a cohesion matrix. The cohesion matrix is obtained as follows: For the field-field similarity, we consider the vocabulary-based similarity. For the method-method similarity, we consider both vocabulary and dependency-based similarity. For the method-field similarity, if the method m_i may access (read or write) the field f_j, then the similarity value is 1. Otherwise, the similarity value is 0. For example, the suitable set of methods to be moved to a new class is obtained as follows: we consider the line with the highest average value and construct a set that consists of the elements in this line that have a similarity value that is higher than a certain threshold.

The semantic function for a remodularization operation corresponds to the average (equal importance) of the different three semantic measures described above. As a remodularization solution is a sequence of operations, the overall semantic evaluation of the solution is the average of the semantic values of all the operations composing a solution.

Some code changes contribute to the domain vocabulary of the system but not necessarily all of them. Thus, we are considering semantic coherence and consistency with prior code changes as separate objectives. In addition, the consistency with priori code changes is not mainly related to the domain vocabulary but to the type of operations that are applied to a similar context. Treating the similarity with prior changes as a separate objective can address the problem that developers use sometimes names of code elements that semantically do not make any sense.

Furthermore, we decided to separate semantic coherence and the consistency with prior code changes in two different objectives to give more flexibility to user when selecting the best solution based on their preferences and also to ensure the usefulness of our approach even in the situation when the change history is not available (the user can easily exclude this objective while maintaining the semantic coherence one).

To find a compromise between the seven objectives described in this section, we used a recent many-objective optimization algorithm (NSGA-III) that will be described in the next section.

4.2.2 Software remodularization using NSGA-III

This section shows how the remodularization problem can be addressed using NSGA-III. We first present an overview of the technique then we provide the details of our adaptation to the remodularization problem.

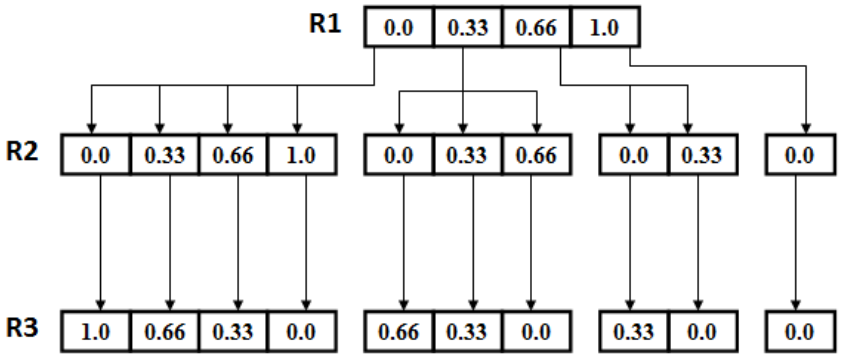
4.2.2.1 NSGA-III

NSGA-III is a recent many-objective algorithm proposed by [20]. The basic framework remains similar to the original NSGA-II algorithm with significant changes in its selection mechanism. Figure 5 gives the pseudo-code of the NSGA-III procedure for a particular generation *t*. First, the parent population P_t (of size N) is randomly initialized in the specified domain, and then the binary tournament selection, crossover and mutation operators are applied to create an offspring population Q_t. Thereafter, both populations are combined and sorted according to their domination level and the best N members are selected from the combined population to form the parent population for the next generation. The fundamental difference between NSGA-II and NSGA-III lies in the way the niche preservation operation is performed. Unlike NSGA-II, NSGA-III starts with a set of reference points Z^r. The set of uniformly distributed reference points is generated using the method of Das and Dennis [90] which is well-detailed and described in [91].

In order to illustrate the reference point generation process, we give in what follows an example of such generation with only three objectives in order to ease the understanding and the visualization of such process. However, this process is generic for any number of objectives M. The Das and Dennis approach for this case generates W reference points on the hyperplane with

a uniform spacing $\delta = 1/p$. We assume for this example that $p = 3$ (i.e., $\delta = 0.33$). The number of reference points is thus $W = C_p^{(M+p-1)}$ which is equal to 10. The following figure describes the reference point set generation mechanism:

The following Figure 4.2 shows an Illustration of the Das and Dennis method for the generation of the reference points by computing $R1$, $R2$, and $R3$ recursively. The table shows the combinations of $R1$, $R2$, and $R3$ components. The figure shows the plotting of the 10 obtained reference points on the hyperplane. For our remodularization problem, we have seven objectives ($M = 7$) and we use $p = 5$. These are the only parameters to adjust for our remodularization problem.



R1	R2	R3 = (1-R1-R2)
0.0	0.0	1.0
0.0	0.33	0.66
0.0	0.66	0.33
0.0	1.0	0.0
0.33	0.0	0.66
0.33	0.33	0.33
0.33	0.66	0.0
0.66	0.0	0.33
0.66	0.33	0.0
1.0	0.0	0.0

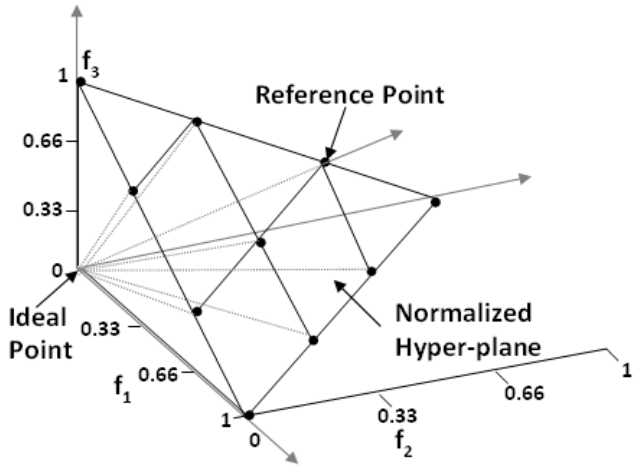


Figure 4.2. 3D plotting of the obtained 10 reference points with p = 3.

After non-dominated sorting, all acceptable front members and the last front F_l that could not be completely accepted are saved in a set S_t. Members in S_t/F_l are selected right away for the next generation. However, the remaining members are selected from F_l such that a desired diversity is maintained in the population. Original NSGA-II uses the crowding distance measure for selecting a well-distributed set of points, however, in NSGA-III the supplied reference points (Z^r) are used to select these remaining members (cf. Figure 4.2). To accomplish this, objective values and reference points are first normalized so that they have an identical range. Thereafter, the orthogonal distance between a member in S_t and each of the reference lines (joining the ideal point, i.e., the vector composed of 7 zero and a reference point) is computed. The member is then associated with the reference point having the smallest orthogonal distance. Next, the niche count ρ for each reference point, defined as the number of members in S_t/F_l that are associated with the reference point, is computed for further processing. The reference point having the minimum niche count is identified and the member from the last front F_l that is associated with it is included in the final population. The niche count of the identified reference point is increased by one and the procedure is repeated to fill up population P_{t+1}.

It is worth noting that a reference point may have one or more population members associated with it or need not have any population member associated with it. Let us denote this niche count as ρ_j for the j-th reference point. We now devise a new niche-preserving operation as follows. First, we identify the reference point set $J_{\min} = \{j: \text{argmin}_j (\rho_j)\}$ having minimum ρ_j. In case of multiple such reference points, one ($j^* \in J_{\min}$) is chosen at random. If $\rho_{j^*} = 0$ (meaning that there is

no associated P_{t+1} member to the reference point j^*), two scenarios can occur. First, there exist one or more members in front F_l that are already associated with the reference point j^*. In this case, the one having the shortest perpendicular distance from the reference line is added to P_{t+1}. The count ρ_{j^*} is then incremented by one. Second, the front F_l does not have any member associated with the reference point j^*. In this case, the reference point is excluded from further consideration for the current generation. In the event of $\rho_{j^*} \geq 1$ (meaning that already one member associated with the reference point exists), a randomly chosen member, if exists, from front F_l that is associated with the reference point F_l is added to P_{t+1}. If such a member exists, the count ρ_{j^*} is incremented by one. After ρ_j counts are updated, the procedure is repeated for a total of K times to increase the population size of P_{t+1} to N.

This following section describes our adaptation of NSGA-III to our remodularization problem. Thus, we define the following adaptation steps: representation of the solutions and the generation of the initial population, evaluation of individuals using the fitness functions, selection of the individuals from one generation to another, generation of new individuals using genetic operators (crossover and mutation) to explore the search space and the normalization of population members.

4.2.2.2 Solution Representation

To represent a candidate remodularization solution (individual), we used a vector representation. Each vector's dimension represents a remodularization operation. Thus, a solution is defined as a sequence of operations applied to different parts of the system to improve its modularization. A randomly generated solution is created as follows. First, we generate the solution length randomly between the lower and upper bounds of the solution length. After that, for each chromosome dimension, we generate a number i between 1 and the total number of possible operations, then we assign the i^{th} operation to the considered dimension. For each operation, the parameters, described in Table 2, are randomly generated from the list of source code elements extracted from the system to remodularize using a parser. An example of a solution is given in Figure 6 applied to the motivating example described in Section 2.

When generating a sequence of operations (individual), it is important to guarantee that they are feasible and that they can be applied. The first work in the literature was proposed by [87]

who introduced a way of formalizing the preconditions that must be imposed before a code change can be applied in order to preserve the behavior of the system. Opdyke created functions which could be used to formalize constraints. These constraints are similar to the Analysis Functions used later by [70] [92] who developed a tool to reduce program analysis. In our approach, we used a system to check a set of simple conditions, taking inspiration from the work proposed by Ó Cinnéide. Our search-based remodularization tool simulates operations using pre and post conditions that are expressed in terms of conditions on a code model. For example, to apply the remodularization operation *MoveClass(GanttTaskRelationship, net.sourceforge.ganttproject, net.sourceforge.ganttproject.task)*, a number of necessary preconditions should be satisfied, e.g., *net.sourceforge.ganttproject* and *net.sourceforge.ganttproject.task* should exist and should be packages; *GanttTaskRelationship* should exist and should be a class and the class *AttrNSImpl* should be implemented in the package *net.sourceforge.ganttproject*. As postconditions, *GanttTaskRelationship, net.sourceforge.ganttproject,* and *net.sourceforge.ganttproject.task* should exist; *GanttTaskRelationship* class should be in the package *net.sourceforge.ganttproject.task* and should not exists anymore in the package *net.sourceforge.ganttproject*.

Algorithm 4.1. Pseudo-code of NSGA-III main procedure at generation *t*.

Input: H structured reference points Z^s, parent population P_t

Output: P_{t+1}

00: **Begin**

01: $S_t \leftarrow \emptyset$, $i \leftarrow 1$;

02: $Q_t \leftarrow$ Variation (P_t);

03: $R_t \leftarrow P_t \cup Q_t$;

04: $(F_1, F_2, ...) \leftarrow$ Non-dominationed_Sort (R_t);

05: **Repeat**

06: $S_t \leftarrow S_t \cup F_i$; $i \leftarrow i+1$;

07: **Until** $|S_t| \geq N$;

08: $F_l \leftarrow F_i$; /*Last front to be included*/

09: **If** $|S_t| = N$ **then**

10: $P_{t+1} \leftarrow S_t$;

11: **Else**

12: $P_{t+1} \leftarrow \bigcup_{j=1}^{l-1} F_j$;

13: /*Number of points to be chosen from F_l*/
$K \leftarrow N - |P_{t+1}|$;

14: /*Normalize objectives and create reference set Z^r*/
Normalize (F^M; S_t; Z^r; Z^s);

/*Associate each member s of S_t with a reference point*/

/*$\pi(s)$: closest reference point*/

15: /*$d(s)$: distance between s and $\pi(s)$*/
$[\pi(s), d(s)] \leftarrow$ Associate (S_t, Z^r);

/*Compute niche count of reference point $j \in Z^r$ */

16: $\rho_j \leftarrow \sum_{s \in S_t / F_l} ((\pi(s) = j) ? 1 : 0)$;

17: /*Choose K members one at a time from F_l to construct P_{t+1}*/

18: Niching (K, ρ_j, $\pi(s)$, $d(s)$, Z^r, F_l, P_{t+1});

19: **End If**

End

Move Class(GanttTaskRelationship, net.sourceforge.ganttproject, net.sourceforge.ganttproject.task)
Extract Class(XGrammarWriter, XGrammarInput, parseInt())
Move Method(normalize(), XGrammarWriter, DTDGrammar)
Extract Package(net.sourceforge.ganttproject, net.sourceforge.ganttproject.dtl, CharacterDataImpl, ChildNode)

Figure 4.3. An example of solution representation.

4.2.2.3 Fitness Functions

Each generated remodularization solution is executed on the system *S*. Once all required data is computed, the solution is evaluated based on the 7 objectives described in Section 4.2.1.1. Based on these values, the remodularization solution is assigned a non-domination rank (as in NSGA-II) and a position in the objective space allowing it to be assigned to a particular reference point based on distance calculation as previously described. As a reminder, the following fitness functions are used: 1) *number of classes per package* (to minimize); 2) *number of packages in the system* (to minimize); 3) *cohesion* (to maximize); 4) *coupling* (to minimize); 5) *Semantics coherence* (to maximize); 6) *number of operations* (to minimize); and 7) *coherence with the history of code changes* (to minimize). The *semantic fitness function* of a solution corresponds to the average of the semantic values of the operations in the vector. The *history of changes fitness function* maximizes the use of ROs that are similar to those applied to the same code fragments in the past. To calculate the similarity score between a proposed remodularization operation and a recorded operation, we use the fitness function described in Section 4.2.1.1.

Normalization of population members. Usually objective functions are incommensurable (i.e., they have different scales). For this reason, we used the normalization procedure proposed by [20] to circumvent this problem. At each generation, the minimal and maximal values for each metric are recorded and then used by the normalization procedure. Normalization allows the population members and with the reference points to have the same range, which is a pre-requisite for diversity preservation.

4.2.2.4 Evolutionary 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. This operator must ensure the respect of the length limits by

eliminating randomly some ROs. As illustrated in Figure 4.4, each child combines some of the operations of the first parent with some ones of the second parent. In any given generation, each solution will be the parent in at most one crossover operation. It is important to note that in many-objective optimization, it is better to create children that are close to their parents in order have a more efficient search process. For this reason, we control the cutting point of the one-point crossover operator by restricting its position to be either belonging to the first third of the operations sequence or belonging to the last third.

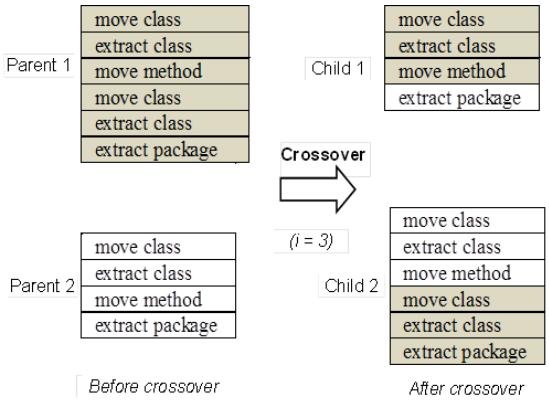


Figure 4.4. Crossover operator.

For mutation, we use the bit-string mutation operator that picks probabilistically one or more operations from its or their associated sequence and replace them by other ones from the initial list of possible operations as described in the running example of Figure 4.5. The number of changes is unknown a priori and depends on the mutation probability. Indeed, each chromosome dimension would be changed according to the mutation probability. For example, for a mutation probability of 0.2, for each dimension, we generate randomly a number x between 0 and 1, if x<0.2 we change the dimension, otherwise not.

Figure 4.5. Mutation operator.

After applying genetic operators (mutation and crossover), we verify the feasibility of the generated sequence of operations by checking the pre and post conditions. Each operation that is

not feasible due to unsatisfied preconditions will be removed from the generated remodularization sequence. The new sequence is considered valid in our NSGA-III adaptation if the number of rejected operations is less than 10% of the total sequence size.

4.3 Validation

In order to evaluate our approach for restructuring systems using NSGA-III, we conducted a set of experiments based on different versions of large open source systems and one industrial project provided by Ford Motor Company. Each experiment is repeated 31 times, and the obtained results are subsequently statistically analyzed with the aim to compare our NSGA-III proposal with a variety of existing approaches [18] [79] [78]. In this section, we first present our research questions and then describe and discuss the obtained results. Finally, we discuss the various threats to the validity of our experiments.

4.3.1 Research Questions

In our study, we assess the performance of our remodularization approach by finding out whether it could generate meaningful sequences of operations that improve the structure of packages while reducing the number of code changes, preserving the semantic coherence of the design, and reusing as much as possible a base of recorded operations applied in the past in similar contexts. Our study aims at addressing the following research questions outlined below. We also explain how our experiments are designed to address these questions. The main question to answer is to what extent the proposed approach can propose meaningful remodularization solutions. To find an answer, we defined the following 7 research questions:

RQ1.1: To what extent can the proposed approach improve the structure of packages in the system?

RQ1.2: To what extent the proposed approach preserve the semantics while improving the packages structure?

RQ1.3: To what extent can the proposed approach minimize the number of changes (size)?

RQ1.4: To what extent the use of recorded changes improves the suggestion of good remodularization solutions?

RQ2: How does the proposed many-objective approach based on NSGA-III perform compared to other many/multi-objective algorithms or a mono-objective approach?

RQ3: How does the proposed many-objective approach based on NSGA-III perform compared to existing remodularization approach not based on heuristic search?

RQ4: Insight. How our many-objective remodularization approach can be useful for software engineers in a real-world setting?

To answer RQ1.1, we validate the proposed remodularization on four medium to large-size open-source systems and one industrial project to evaluate the structural improvements of systems after applying the best solution. To this end, we used the following metrics: *average number of classes per package (NCP), number of packages (NP), number of inter-edges (NIE) and number of intra-edges (NAE).*

To answer RQ1.2, it is important to validate the proposed remodularization solutions from both quantitative and qualitative perspectives. To this end, we use two different validation methods: manual validation and automatic validation of the efficiency of the proposed solutions. For the manual validation, we asked groups of potential users (software engineers) of our remodularization tool to evaluate, manually, whether the suggested operations are feasible and make sense semantically. We define the metric "manual precision" (MP) which corresponds to the number of meaningful operations, in terms of semantic coherence, over the total number of suggested operations. MP is given by the following equation:

$$MP = \frac{|coherent\ operations|}{|proposed\ operations|} \in [0,1] \qquad (4.6)$$

For the automatic validation, we introduce manually several changes on the remodularization of JHotDraw and we evaluate the ability of our approach to generating the initial version of the system (considered as a well-designed system). In fact, JHotDraw is considered as one of the well-designed open source systems and several design patterns are used in its implementation.

Thus, we compare the proposed operations with the expected ones in terms of recall and precision:

$$\mathrm{RE}_{recall} = \frac{|\text{suggested operations}| \cap |\text{expected operations}|}{|\text{expected operations}|} \in [0,1] \quad (4.7)$$

$$\mathrm{PR}_{precesion} = \frac{|\text{suggested operations}| \cap |\text{expected operations}|}{|\text{suggested operations}|} \in [0,1] \quad (4.8)$$

To answer RQ1.3, we evaluate the number of operations (*NO*) suggested by the best remodularization solutions on the different systems.

To answer RQ1.4, we use the metric *MP* to evaluate the effect of the use of recorded operations, applied in the past to similar contexts, on the semantic coherence. Moreover, in order to evaluate the importance of reusing recorded operations in similar contexts, we define the metric "reused operations" (ROP) that calculates the percentage of operations from the base of recorded operations used to generate the optimal remodularization solutions by our proposal. *ROP* is given by the following equation:

$$ROP = \frac{|\text{used operations from the base of recorded operations}|}{|\text{base of recorded operations}|} \in [0,1] \quad (4.9)$$

To answer RQ2, we compared the performance of NSGA-III with two many-objective techniques, MOEA/D and IBEA, and also with a multi-objective algorithm that uses NSGA-II. We used *Inverted Generational Distance (IGD)* to compare between the different algorithms*:* A number of performance metrics for multi-objective optimization have been proposed and discussed in the literature, which aims to evaluate the closeness to the Pareto optimal front and the diversity of the obtained solution set, or both criterion. Most of the existing metrics require the obtained set to be compared against a specified set of Pareto optimal reference solutions. In this study, the inverted generational distance (IGD) is used as the performance metric since it has been shown to reflect both the diversity and convergence of the obtained non-dominated solutions. The IGD corresponds to the average Euclidean distance separating each reference solution from its closest non-dominated one. Note that for each system we use the set of Pareto optimal solutions generated by all algorithms over all runs as reference solutions. In addition to *IGD*, we used the above-described metrics to compare between all the algorithms: *NCP*, *NP*, *NIE*, *NAE*, *MP*, *RE*, and *PR*. We also compared our approach with a multi-objective remodularization technique proposed by [79] where the objectives considered are coupling,

cohesion and number of changes. Since the approach of Abdeen et al. is limited to the use of only one operation (Move Class), we only used the qualitative evaluation based on *MP* for the comparison.

It is important also to determine if considering each conflicting metric as a separate objective to optimize performs better than a mono-objective approach that aggregates several metrics in one objective. The comparison between a many-objective EA with a mono-objective one is not straightforward. The first one returns a set of non-dominated solutions while the second one returns a single optimal solution. In order to resolve this problem, for each many-objective algorithm we choose the nearest solution to the Knee point (i.e., the vector composed of the best objective values among the population members) as a candidate solution to be compared with the single solution returned by the mono-objective algorithm. We compared NSGA-III with an existing mono-objective remodularization approach [18] based on the use of cohesion and coupling aggregated in one fitness function. Since the mono-objective approach is limited to the use of only one operation (Move Class), we only used the qualitative evaluation based on *MP* for the comparison and feedback from software engineers on using both tools.

For RQ3, since it is not sufficient to outperform existing search-based remodularization techniques, we compared our proposal to an existing remodularization technique based on the use of coupling and cohesion [78] and limited to the only use of *Split packages* change. Thus, we compared our proposal using only the qualitative evaluation based on *MP* and feedback from software engineers on using both tools.

For RQ4, we evaluated the benefits of our remodularization tool by several software engineers. To this end, they classify the suggested operations (*IOP*) one by one as interesting or not. The difference with the *MP* metric is that the operations are not classified from a semantic coherence perspective but form a usefulness one:

$$IOP = \frac{|\text{useful operations}|}{|\text{operations}|} \in [0,1] \qquad (4.10)$$

To answer the above research questions, we selected the solution from the set of non-dominated ones providing the maximum trade-off using the following strategy when comparing between the different algorithms (expect the mono-objective algorithm where we select the

solution with the highest fitness function). In order to find the maximal trade-off solution of the multi-objective or many-objective algorithm, we use the trade-off worthiness metric proposed by Rachmawati and Srinivasan [93] to evaluate the worthiness of each non-dominated solution in terms of compromise between the objectives. This metric is expressed as follows:

$$\mu(x_i, S) = \min_{x_j \in S, x_i \neq x_j, x_j \neq x_i} T(x_i, x_j) \quad (4.11)$$

Where

$$T(x_i, x_j) = \frac{\sum_{m=1}^{M} \max\left[0, \frac{f_m(x_j) - f_m(x_i)}{f_m^{\max} - f_m^{\min}}\right]}{\sum_{m=1}^{M} \max\left[0, \frac{f_m(x_i) - f_m(x_j)}{f_m^{\max} - f_m^{\min}}\right]}$$

We note that x_j denotes members of the set of non-dominated solutions S that are non-dominated with respect to x_i. The quantity $\mu(x_i, S)$ expresses the least amount of improvement per unit deterioration by substituting any alternative x_j from S with x_i. We note also that $f_m(x_i)$ corresponds to the m^{th} objective value of solution x_i and $f_m^{\max}/f_m^{\min}$ corresponds to the maximal/minimal value of the m^{th} objective in the population individuals. In the above equations, normalization is performed in order to prevent some objectives being predominant over others since objectives are usually incommensurable in real world applications. In the last equation, the numerator expresses the aggregated improvement gained by substituting x_j with x_i. However, the denominator evaluates the deterioration generated by the substitution.

4.3.2 Software Projects Studied

We used a set of well-known open-source java projects and one project from our industrial partner Ford Motor Company. We applied our approach to four large and medium size open-source java projects: Xerces-J, JFreeChart, GanttProject, and JHotDraw. 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. JHotDraw is a GUI framework for drawing editors. Finally, the industrial project, JDI, is a Java-based software system that helps Ford Motor Company analyze useful information from the past sales of dealerships data and suggests which vehicles to order for their dealer inventories in the future. This system is main key software application used by Ford Motor Company to improve their vehicles sales by selecting the right vehicle configuration to the expectations of customers. JDI is

a highly structured and several versions were proposed by software engineers at Ford during the past 10 years. Due to the importance of the application and the high number of updates performed during a period of 10 years, it is critical to ensure good modularization of JDI to reduce the time required by developers to introduce new features in the future. 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. Table 4.3 Table 4.3. Statistics of the studied systems and solution length limits.provides some descriptive statistics about these six programs. As described in**,** the upper and lower bounds on the chromosome length used in this study are set to 10 and 350 respectively. Several SBSE problems including remodularization are characterized by a varying chromosome length. This issue is similar to the problem of bloat control in genetic programming where the goal is to identify the tree size limits. To solve this problem, we performed several trial and error experiments where we assess the average performance of NSGA-III using the HV (hypervolume) performance indicator while varying the size limits between 10 and 500 operations. Figure 4.6 shows the obtained results which explain our choices described in Table 4.3.

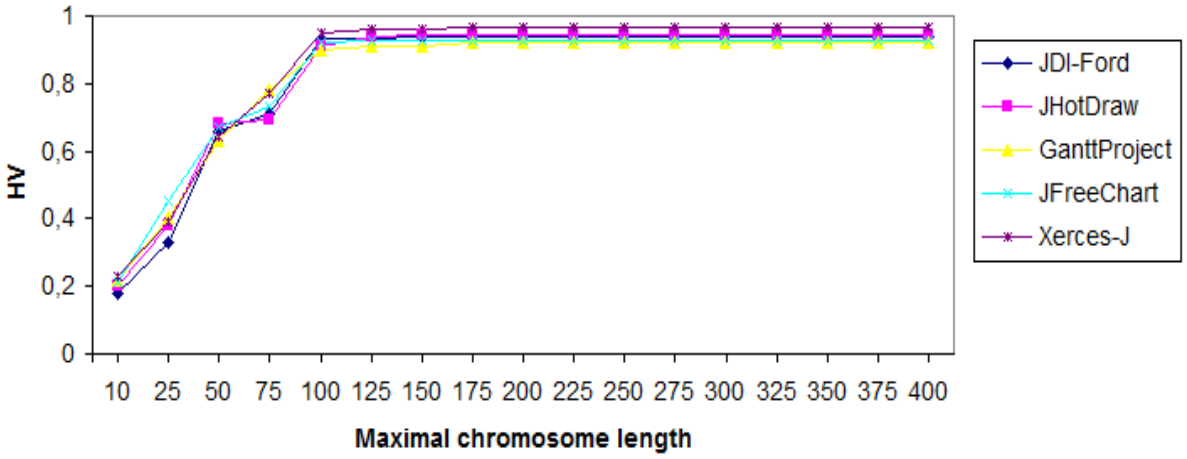


Figure 4.6. Average Performance of NSGA-III the search algorithms using the HV (hypervolume) indicator while varying the size limits between 10 and 500 operations.

Table 4.3. Statistics of the studied systems and solution length limits.

Systems	Release	# classes	KLOC	Solution length limits (Min/Max)
Xerces-J	v2.7.0	991	240	[35 , 350]
JHotDraw	v6.1	585	21	[20 , 250]
JFreeChart	v1.0.9	521	170	[20 , 250]

GanttProject	v1.10.2	245	41	[10 , 150]
JDI-Ford	v5.8	638	247	[25 , 300]

To collect operations applied in previous program versions, we use Ref-Finder [94]. Ref-Finder, implemented as an Eclipse plug-in, can identify operations between two releases of a software system. Table 4.4 shows the analyzed versions and the number of operations, identified by Ref-Finder, between each subsequent couple of analyzed versions, after the manual validation. In our study, we consider only the operation types described in Table 4.4.

Table 4.4. Analyzed versions and operations collection.

Systems	**Collected operation**	
	Previous releases	**# operations**
Xerces-J	v1.4.2 - v2.6.1	52
JFreeChart	v1.0.6 - v1.0.8	63
GanttProject	v1.7 - v1.10.1	81
JHotDraw	v5.1 - v6.0	56
JDI-Ford	v2.4 – v5.6	97

4.3.3 Experimental Setting

The goal of the study is to evaluate the usefulness and the effectiveness of our remodularization tool in practice. We conducted a non-subjective evaluation with potential developers who can use our tool. Indeed, operations should not only improve the structure of packages but should also be meaningful from a developer's point of view in terms of semantic coherence and usefulness.

4.3.3.1 Subjects

Our study involved 13 subjects from the University of Michigan and 2 software engineers from Ford Motor Company. Subjects include 5 master students in Software Engineering, 7 Ph.D. students in Software Engineering, 1 faculty member in Software Engineering, and 2 junior software developers. 4 of them are females and 11 are males. All the subjects are volunteers and familiar with Java development. The experience of these subjects on Java programming ranged from 2 to 16 years. The evaluated solutions by the subjects are those that represent the maximum trade-off between the objective using the trade-off worthiness metric proposed by Rachmawati and Srinivasan as described in Section 4.3.1.

4.3.3.2 Scenario

We designed our study to answer our research questions. The subjects were invited to fill a questionnaire that aims to evaluate our suggested operations. We divided the subjects into six groups according to 1) the number of studied systems 2) the number of remodularization solutions to evaluate, and 3) the number of techniques to be tested.

Table 4.5. Considered solutions for the qualitative evaluation.

Ref. Solution	Algorithm/ Approach	# Objectives	Considered objectives
Solution 1	NSGA-III	2	NCP, NP
Solution 2		3	NCP, NP, COU
Solution 3		4	NCP, NP, COU, COH
Solution 4		5	NCP, NP, COU, COH, SP
Solution 5		6	NCP, NP, COU, COH, SP, NCH
Solution 6		7	NCP, NP, COU, COH, SP, NCH, CHC
Solution 7	IBEA	2	NCP, NP
Solution 8		3	NCP, NP, COU
Solution 9		4	NCP, NP, COU, COH
Solution 10		5	NCP, NP, COU, COH, SP
Solution 11		6	NCP, NP, COU, COH, SP, NCH
Solution 12		7	NCP, NP, COU, COH, SP, NCH, CHC
Solution 13	MOEA/D	2	NCP, NP
Solution 14		3	NCP, NP, COU
Solution 15		4	NCP, NP, COU, COH
Solution 16		5	NCP, NP, COU, COH, SP
Solution 17		6	NCP, NP, COU, COH, SP, NCH
Solution 18		7	NCP, NP, COU, COH, SP, NCH, CHC
Solution 19	Mono-objective Simulate Annealing [18]	1	COU+COH
Solution 20	NSGA-II [79]	3	COU, COH, NCH
Solution 21	Automated re-modularization [78]	2	COU, COH

The number of remodularization solutions to evaluate depends on different objectives combinations: *average number of classes per package (NCP), number of packages (NP), coupling (COU), cohesion (COH), semantics preservation (SP), number of changes (NCH) and coherence with history of changes (CHC)*. For each combination (two, three, four, five, six and seven, objectives), a remodularization solution is suggested to find the best compromise between the considered objectives. Similarly, the solutions of the state-of-the-art works are empirically evaluated in order to compare them to our approach as described in the previous section. Table 6 describes the number of remodularization solutions to be evaluated for each studied system in order to answer our research questions.

As shown in Table 4.5, for each system, 21 remodularization solutions have to be evaluated. Due to the huge number of operations to be evaluated (each solution consists of a set of

operations), we pick at random a subset of up to 10 operations per solution to be evaluated in our study. In Table 4.5, we summarize how we divided subjects into 6 groups in order to cover all remodularization solutions. In addition, as illustrated in Table 4.6, we are using a cross-validation to reduce the impact of subjects on the evaluation. Each subject evaluates different remodularization solutions for three different systems.

Subjects were first asked to fill out a pre-study questionnaire containing five questions. The questionnaire helped to collect background information such as their role within the company, their programming experience, their familiarity with software remodularization. In addition, all the participants attended one lecture about software remodularization and passed five tests to evaluate their performance to evaluate and suggest remodularization solutions. Then, the groups are formed based on the pre-study questionnaire and the test result to make sure that all the groups have almost the same average skills.

The participants were asked to justify their evaluation of the solutions and these justifications are reviewed by the organizers of the study (one faculty member, one postdoc, one Ph.D. student and one master student). In addition, our experiments are not only limited to the manual validation, but also, the automatic validation can verify the effectiveness of our approach.

Subjects were aware that they are going to evaluate the semantic coherence and the usefulness of the operations, but do not know the particular experiment research questions (algorithms used, different objectives used and their combinations). Consequently, each group of subjects who accepted to participate in the study, received a questionnaire, a manuscript guide to help them to fill the questionnaire, and the source code of the studied systems, in order to evaluate 21 solutions (10 operations per solution). The questionnaire is organized in an excel file with hyperlinks to visualize easily the source code of the affected code elements. Subjects are invited to select for each operation one of the possibilities: *"Yes"*, "*No"*, or *"Maybe"* (if not sure) about the semantic coherence and usefulness. 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.

Table 4.6. Survey organization.

Subject groups	Systems	Algorithms / Approaches	Solutions
Group A	GanttProject	NSGA-III IBEA	Solution 1-6 Solution 7-12
	Xerces	MOEA/D, Abdeen et al.2011	Solution 13-18 Solution 19
	JFreeChart	Abdeen et al. 2013, Bavota et al. 2013	Solution 20 Solution 21
Group B	GanttProject	NSGA-III IBEA	Solution 1-6 Solution 7-12
	Xerces	MOEA/D, Abdeen et al.2011	Solution 13-18 Solution 19
	JFreeChart	Abdeen et al. 2013, Bavota et al. 2013	Solution 20 Solution 21
Group C	GanttProject	NSGA-III IBEA	Solution 1-6 Solution 7-12
	Xerces	MOEA/D, Abdeen et al.2011	Solution 13-18 Solution 19
	JFreeChart	Abdeen et al. 2013, Bavota et al. 2013	Solution 20 Solution 21
Group D	GanttProject	NSGA-III IBEA	Solution 1-6 Solution 7-12
	JHotDraw	MOEA/D, Abdeen et al.2011	Solution 13-18 Solution 19
	JDI-Ford	Abdeen et al. 2013, Bavota et al. 2013	Solution 20 Solution 21
Group E	Xerces	NSGA-III IBEA	Solution 1-6 Solution 7-12
	JHotDraw	MOEA/D, Abdeen et al.2011	Solution 13-18 Solution 19
	JDI-Ford	Abdeen et al. 2013, Bavota et al. 2013	Solution 20 Solution 21
Group F	JFreeChart	NSGA-III IBEA	Solution 1-6 Solution 7-12
	JHotDraw	MOEA/D, Abdeen et al.2011	Solution 13-18 Solution 19
	JDI-Ford	Abdeen et al. 2013, Bavota et al. 2013	Solution 20 Solution 21

4.3.3.3 Parameters Tuning

The parameter setting influences significantly the performance of a search algorithm for a particular problem [95]. For this reason, for the search-based algorithm and for each system (cf. Table 4.7), we perform a set of experiments using several population sizes: 62, 100, 150, 180, 140, and 190 for respectively 2, 3, 4, 5, 6 and 7 objectives. The maximum number of generations used is 300, 500, 700, 1000, 1200, and 1400 respectively, for 2, 3, 4, 5, 6 and 7 objectives. 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 operations that are considered and the size of the program to be restructured (cf.

Table 4.3). A higher number of operations in a solution do not necessarily mean that the results will be better. Ideally, a small number of operations should be sufficient to provide a good trade-off between the fitness functions. This parameter can be specified by the user or derived randomly from the sizes of the program and the used operations list. During the creation, the solutions have random sizes in the allowed range. Each algorithm is executed 31 times with each configuration and then the comparison between the configurations is done based on IGD using the Wilcoxon test. In order to have significant results, for each couple (algorithm, system), we use the trial and error method in order to obtain a good parameter configuration. Since we are comparing different search algorithms, we classify parameters into common parameters and specific parameters.

Table 4.7 depicts the important common parameters. For MOEA/D, the neighborhood size is set to 20. For IBEA, the scaling parameter κ is set to 0.01. For the SA of [18], the start and stop temperatures are set respectively to 22.8 and 1.0 using a geometrical cooling scheme with a cooling rate of 0.9975 and the number of local search iterations is set to 15. It is important to note that all heuristic algorithms have the same termination criterion for each experiment (same number of evaluations) in order to ensure fairness of comparisons.

Table 4.7. The setting of common parameters.

Number of objectives	Number of reference points (for NSGA-III and MOEA/D)	Population size	Number of generations	Crossover rate	Mutation rate
2	62	100	300	0.9	0.1
3	100	150	500	0.9	0.1
4	150	182	700	0.9	0.1
5	180	174	1000	0.8	0.2
6	140	186	1200	0.8	0.2
7	190	193	1400	0.8	0.2

4.3.3.4 Statistical Tests

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 performed based on 31 independent simulation runs for each problem instance and the obtained results are statistically analyzed by using the Wilcoxon rank sum test [96] with a 99% confidence level (α = 1%). The latter verifies the null hypothesis H0 that the obtained results of the 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 (≤ 0.01) means that we accept H1 and we reject H0. However, a p-value that is strictly greater than α (> 0.01) means the opposite. In fact, for each problem instance, we compute the p-value obtained by comparing NSGA-II, IBEA, MOEA/D and mono-objective search results with NSGA-III ones. In this way, we determine whether the performance difference between NSGA-III and one of the other approaches is statistically significant or just a random result.

4.3.4 Results

4.3.4.1 Results for RQ1.1

Table 4.8 summarizes the results of median values of the structural metrics over 31 independent simulation runs after applying the proposed operations by the remodularization solution selected using the knee-point strategy. The results of Table 4.8 are based on the consideration of all the 7 objectives for the many-objective algorithms, thus, the order is not important and has no impact on the results. The software engineer can select the best solution based on his preferences (fitness function values) and programming behavior from the non-dominated (trade-off) set of solutions.

As described in Table 4.8, we found that NSGA-III algorithm provides similar structural improvements the other techniques in terms of an average number of classes per package (NCP), cohesion (NIE) and coupling (NAE). However, the number of packages (NP) in the system after applying NSGA-III solutions is slightly higher than all NP values proposed by the best solutions to most of the remaining algorithms in most of the cases except Xerces-J. This can be explained by the fact that decreasing the number of classes per package will automatically increase the number of packages. In addition, the extract class operation created new classes that increased the average number of classes per package.

The structural improvement scores of multi-objective and mono-objective algorithms are very close to those produced by many-objective algorithms, especially NSGA-III. This is an interesting result confirming that our NSGA-III can find very good compromises between 7 objectives that are similar and sometimes outperforms those that are produced by existing

approaches using only structural and semantic objectives. We believe that improving the structure of packages it is a difficult and very important objective to reach. We consider that NSGA-III performance in terms of improving the structure similar to existing approaches is a very interesting result since the main goal of this work is to improve the structure while preserving the domain semantics which not well-considered by the remaining approaches.

Table 4.8. Average number of classes per package (NCP), number of packages (NP), number of inter-edges (NIE), number of intra-edges (NAE) and the deviation (delta with the initial design) median values of NSGA-III, IBEA, MOEA/D, SA, NSGA-II and Bavota et al. over 31 independent simulation runs. A "+" symbol at the i^{th} position in the sequence of signs presented below the instance names means that the NSGA-III algorithm metric median value is statistically different from the i^{th} algorithm one. A "-" symbol at the i^{th} position in the sequence of signs means the opposite (e.g., for Xerces-J, NSGA-III is not statistically different from IBEA, however, it is statistically different from the other algorithms).

System	Approach	NCP	dev.NCP	NP	dev.NP	NIE	dev.NIE	NAE	Dev.NAE
Xerces-J (-++++)	NSGA-III	19	-4	46	+5	316	-69	432	+72
	IBEA	21	-2	44	+3	328	-57	411	+51
	MOEA/D	18	-5	56	+15	352	-33	397	+37
	SA Abdeen et al. 2011	24	+1	42	+1	314	-71	441	+81
	NSGA-II Abdeen et al.2011	18	-5	56	+15	333	-52	422	+62
	Bavota et al. 2013	18	-5	56	+15	302	-83	453	+93
JFreeChart (+++++)	NSGA-III	14	-6	38	+8	286	-71	384	+69
	IBEA	16	-4	36	+6	304	-53	392	+77
	MOEA/D	16	-4	36	+6	314	-43	383	+86
	SA Abdeen et al. 2011	13	-7	42	+12	291	-66	396	+81
	NSGA-II Abdeen et al.2011	13	-7	42	+12	301	-56	3786	+71
	Bavota et al. 2013	11	-9	47	+17	278	-79	398	+94
GanttProject (--+++)	NSGA-III	14	-3	18	+9	259	-68	294	+81
	IBEA	12	-5	21	+12	247	-80	304	+91
	MOEA/D	14	-3	18	+9	259	-68	291	+78
	SA Abdeen et al. 2011	12	-5	21	+12	238	-89	329	+116
	NSGA-II Abdeen et al.2011	13	-4	19	+10	244	-83	323	+111
	Bavota et al. 2013	12	-6	21	+12	236	-91	331	+118
JHotDraw (+-++-)	NSGA-III	16	-8	37	+14	391	-83	425	+84
	IBEA	18	-6	33	+10	404	-70	418	+77
	MOEA/D	16	-8	37	+14	391	-83	433	+92
	SA Abdeen et al. 2011	14	-10	42	+19	384	-90	439	+98
	NSGA-II Abdeen et al.2011	15	-9	40	+17	388	-86	435	+94
	Bavota et al. 2013	11	-13	52	+29	378	-96	445	+104
JDI-Ford (++--+)	NSGA-III	14	-4	46	+21	301	-67	412	+76
	IBEA	14	-4	46	+21	324	-44	422	+86
	MOEA/D	16	-2	39	+16	308	-60	391	+55
	SA Abdeen et al. 2011	13	-5	52	+27	297	-71	421	+85
	NSGA-II Abdeen et al.2011	14	-4	48	+23	304	-64	414	+78
	Bavota et al. 2013	13	-5	52	+27	294	-74	424	+88

4.3.4.2 Results for RQ1.2

To answer RQ1.2, we need to assess the correctness/meaningfulness of the suggested remodularization solutions from a developers' standpoint. To this end, we reported the results of our empirical qualitative evaluation in Figure 4.7 (MP). As reported in Figure 4.7, the majority of the suggested solutions by NSGA-III improve significantly the structure (RQ1.1) while preserving the semantic coherence much better than all existing approaches. On average, for all of our five studied systems, 88% of proposed operations are considered as semantically feasible and do not generate semantic incoherence by the software engineers. This score is significantly higher than the ones of the NSGA-II and SA approaches having respectively between 51% and 70%, in average, of MP scores on the different systems. However, the performance of the IBEA, MOEA/D and Bavota et al. are close to the performance of our NSGA-II approach in terms of semantic preservation with respectively an average of 84%, 83% and 81% of MP. This can be explained by the fact that semantic measures are considered by these approaches.

Thus, our many-objective approach reduces the number of semantic incoherencies when suggesting ROs. To sum up, our approach performs clearly better for semantics preservation with the cost of a slight degradation of structural improvements compared to the other approaches. This slight loss in the structure (RQ1.1) is largely compensated by the significant improvement of the semantic coherence.

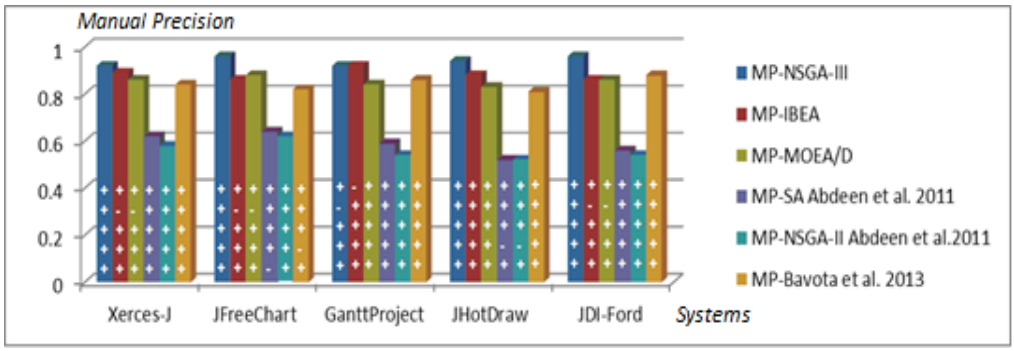


Figure 4.7. Qualitative evaluation of the remodularization solutions (semantics). A "+" symbol at the ith position in the sequence of signs means that the algorithm metric median value is statistically different from the ith algorithm one. A "-" symbol at the ith position in the sequence of signs means the opposite. Sequences of "+" and "-" Signs should be read from top to bottom. The column referring to the system under analysis should be left out of the count in the sequence.

In addition to the empirical evaluation, we automatically evaluate our approach without using the feedback of potential users to give a more quantitative evaluation to answer RQ1.2. Thus, we compare the proposed operations with some expected ones. The expected operations are the inverse of those applied manually by the software engineers who participated in our experiments to modify an initial version of JHotDraw. The participants of our experiments did not introduce changes randomly to JHotDraw. We selected the packages that are clearly very well designed based on a manual inspection using also the following quality metrics: number of classes per package, cohesion of the package, number of lines of code per package, average depth of inheritance tree of classes per package, average number of methods per package and coupling of the package. Thus, the expected refactorings are those that can generate the initial version of these modified well-designed packages. The developers considered only move class and split/extract package operations when modifying the original version of JHotDraw. Thus, we are not only considering JHotDraw without a manual inspection of the best well-designed packages to modify. The total number of introduced changes is 68. Four developers from the University of Michigan subjects worked together to introduce the changes on the same copy of JHotDraw system. We use Ref-Finder to identify operations that are applied to the program version under analysis and the next version. Figure 4.8 summarizes our findings. We found that a considerable number of proposed operations by NSGA-III (an average of 85% in terms of precision and recall) are already applied in the next version by a software development team comparing to other existing mono-objective and multi-objective remodularization approaches having only less than 65% as precision and recall. However, we found that the remodularization solutions proposed by the other many-objective algorithms IBEA and MOEA/D, and Bavota et al. have close scores to NSGA-III with respectively an average of 80%, 75% and 81% of precision. The same observation is valid for the recall.

In conclusion, our approach produces good remodularization suggestions in terms of structural improvements, semantic coherence, and code changes reduction from the point of view of 1) potential users of our tool and 2) expected operations applied to the next program version.

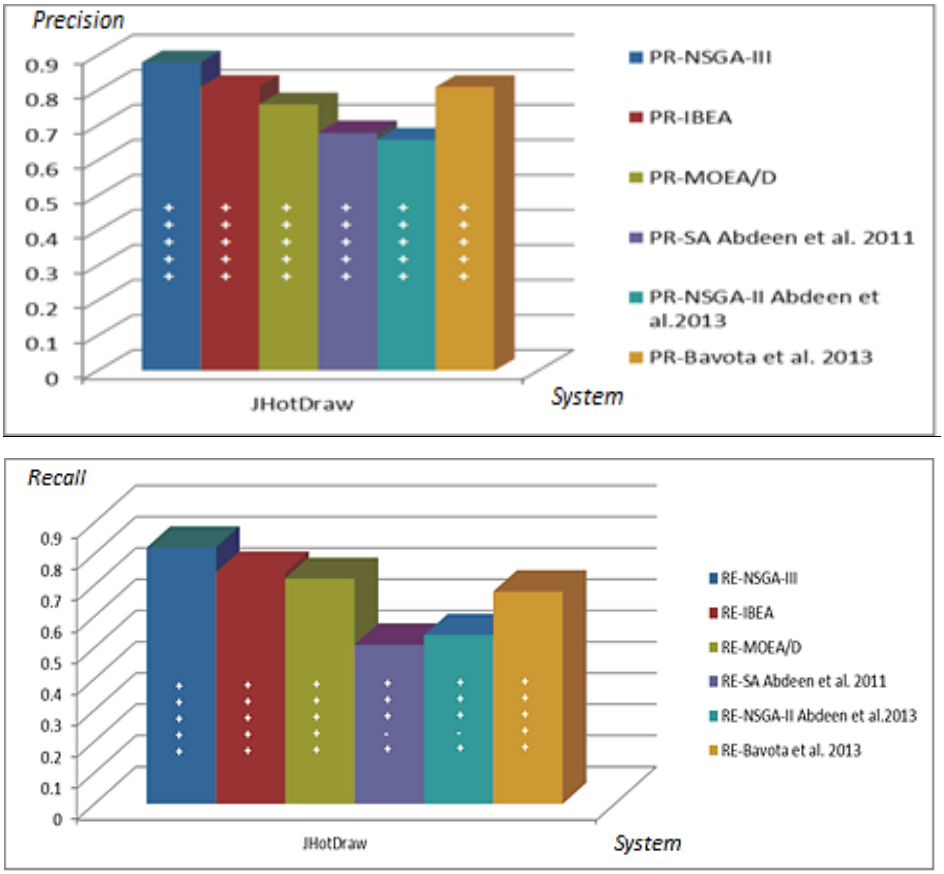


Figure 4.8. Quantitative evaluation (precision and recall) of the remodularization solutions (semantics). A "+" symbol at the ith position in the sequence of signs means that the algorithm metric median value is statistically different from the ith algorithm one. A "-" symbol at the ith position in the sequence of signs means the opposite. Signs should be read from top to bottom. The column referring to the system under analysis should be left out of the count in the sequence.

4.3.4.3 Results for 1.3

To answer RQ1.3, we evaluate the Number of Operations (NO) suggested by the best remodularization solutions on the different systems. Figure 4.9 presents the code changes scores (NO) needed to apply the suggested remodularization solutions for each many-objective or multi-objective algorithm. We found that our approach succeeded in suggesting solutions that do not require high code changes (an average of only 64 operations) comparing to other many-objective (IBEA, MOEA/D) and multi-objective (NSGA-II) algorithms having respectively an average of 72, 71 and 79 for all studied systems. We did not compare the number of changes suggested by our proposal with existing work since they are limited to only two types of changes (move class and split packages).

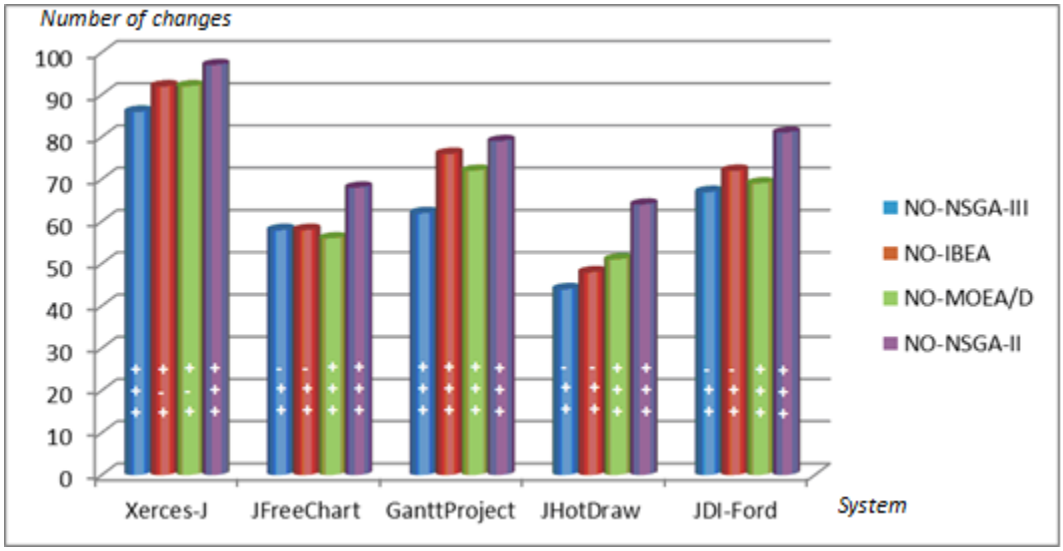


Figure 4.9. Average number of operations. A "+" symbol at the ith position in the sequence of signs means that the algorithm metric median value is statistically different from the ith algorithm one. A "-" symbol at the ith position in the sequence of signs means the opposite. Signs should be read from top to bottom. The column referring to the system under analysis should be left out of the count in the sequence.

4.3.4.4 Results for RQ1.4

To answer RQ1.4, we evaluated the results of our approach compared to other approaches that do not use the history of changes. As described in the previous sections, our NSGA-III approach outperforms clearly existing work including Abdeen et al. 2011, Abdeen et al. 2013 and Bavota et al. 2013 that are not based on the use of the history of changes. This is a good indication that the recorded operations contribute significantly to provide good solutions. In fact, the use of the history of changes is a helper objective to improve the semantic coherence of suggested remodularization solutions. It is also important to note that the SP metrics has the most positive impact on the results of our approach.

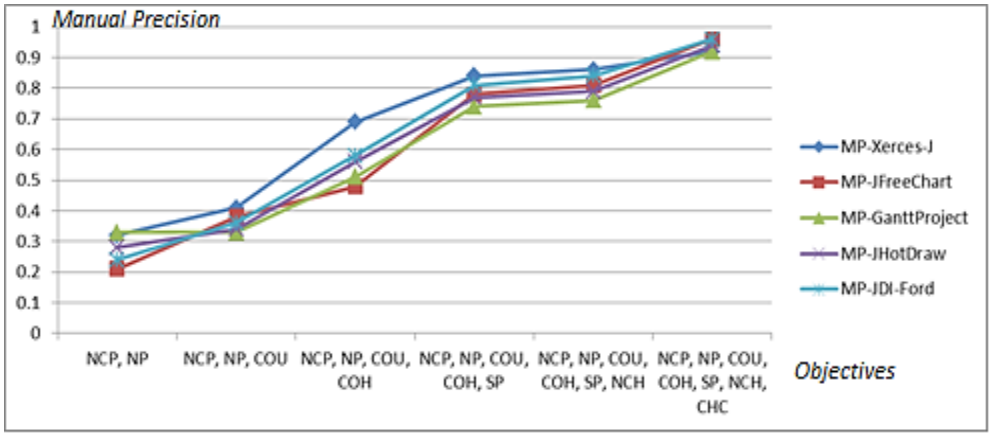


Figure 4.10. The impact of different combinations of objectives on the remodularization solutions (MP).

We conducted also a more quantitative evaluation to investigate the effects of the use of recorded operations, on the semantic coherence (MP). To this end, we compare the MP score with and without using recorded operations. We present in Figure 4.10 the results of different combinations of our seven objectives. The order is important only when we considered the lower number of objectives (Figure 4.10) to evaluate the improvement of the solutions quality if a higher number of objectives are used. The first studies on software remodularization used only the structure then the semantics and after that the effort and history of changes. Our work combined them together in one approach using many-objective techniques. As presented in Figure 4.10, the best MP scores are obtained when the recorded code changes are considered. Moreover, we found that the optimal remodularization solutions found by our approach are obtained with a considerable percentage of reused operations history (ROP) (more than 75% as shown in Figure 4.11). Thus, the obtained results support the claim that recorded operations applied in the past are useful to generate coherent and meaningful remodularization solutions.

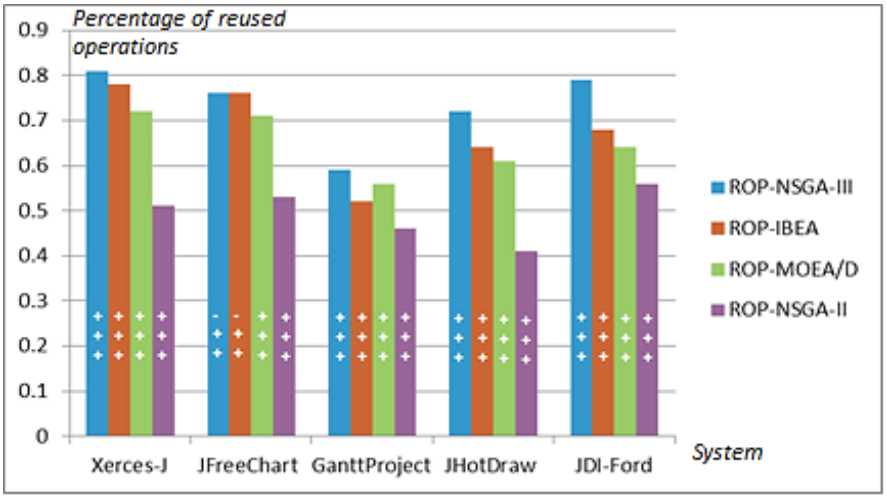


Figure 4.11. Percentage of recorded operations that are used by the best remodularization. A "+" symbol at the ith position in the sequence of signs means that the algorithm metric median value is statistically different from the ith algorithm one. A "-" symbol at the ith position in the sequence of signs means the opposite solutions. Signs should be read from top to bottom. The column referring to the system under analysis should be left out of the count in the sequence.

4.3.4.5 Results for RQ2

In the previous sections, we compared our NSGA-III proposal with one mono-objective technique and one existing multi-objective technique based on NSGA-II. Thus, we focus on the comparison between our NSGA-III adaption and two other many-objective algorithms IBEA and MOEA/D using the same adaptation. Table 10 shows the median IGD values over 31 independent runs for all algorithms under comparison. All the results were statistically significant on the 31 independent simulations using the Wilcoxon rank sum test with a 99% confidence level ($\alpha < 1\%$). For the 3-objective case, we see that NSGA-III and NSGA-II present similar results and that NSGA-III provides slightly better results than IBEA and MOEA/D. For the 5-objective case, NSGA-III strictly outperforms NSGA-II and gives similar results to those of the two other multi-objective algorithms. For the 7-objective case, NSGA-III is better than NSGA-II, IBEA and MOEA/D. Additionally, IBEA seems to be slightly better than MOEA/D. It is worth noting that for problems instances with more 3 objectives, NSGA-II performance is dramatically degraded, which is simply denoted by the ~ symbol. The performance of NSGA-III could be explained by the interaction between: (1) Pareto dominance-based selection and (2) reference point-based selection, which is the distinguishing feature of NSGA-III compared to other existing many-objective algorithms.

Table 4.9. Median IGD values on 31 runs (best values are in bold). ~ means a large value that is not interesting to show. The results were statistically significant on 31 independent runs using the Wilcoxon rank sum test with a 99% confidence level (α < 1%).

System	*M*	MaxGen	NSGA-III	IBEA	MOEA/D	NSGA-II
Xerces-J	3	250	**9.861×10^{-4}**	9.864×10^{-4}	9.863×10^{-4}	9.862×10^{-4}
	5	500	**7.799×10^{-3}**	7.875×10^{-3}	7.878×10^{-3}	8.991×10^{-3}
	7	750	**8.013×10^{-3}**	8.372×10^{-3}	8.368×10^{-3}	~
JHotDraw	3	250	**2.477×10^{-3}**	2.478×10^{-3}	2.478×10^{-3}	**2.477×10^{-3}**
	5	500	**4.193×10^{-3}**	4.201×10^{-3}	4.206×10^{-3}	4.533×10^{-3}
	7	750	**5.536×10^{-3}**	5.801×10^{-3}	5.796×10^{-3}	~
JFreeChart	3	250	**3.744×10^{-4}**	3.747×10^{-4}	3.746×10^{-4}	3.746×10^{-4}
	5	500	**4.578×10^{-4}**	4.602×10^{-4}	4.609×10^{-4}	5.042×10^{-4}
	7	750	**6.099×10^{-4}**	6.208×10^{-4}	6.193×10^{-4}	~
GanttProject	3	250	**5.112×10^{-3}**	5.115×10^{-3}	5.116×10^{-3}	**5.112×10^{-3}**
	5	500	**6.701×10^{-3}**	6.802×10^{-3}	6.801×10^{-3}	6.997×10^{-3}
	7	750	**7.823×10^{-3}**	8.068×10^{-3}	8.044×10^{-3}	~
JDI-Ford	3	250	**6.229×10^{-4}**	6.232×10^{-4}	6.231×10^{-4}	6.231×10^{-4}
	5	500	**6.608×10^{-4}**	6.682×10^{-4}	6.686×10^{-4}	6.887×10^{-4}
	7	750	**6.984×10^{-4}**	7.305×10^{-4}	7.299×10^{-4}	~

Figure 4.12 illustrates the value path plots of all algorithms the 7-objective remodularization problem on JDI-Ford, one of the largest system used in our experiments. Similar observations were made in the remaining systems but are omitted due to space considerations. All quality metrics were normalized between 0 and 1 and all are to be minimized. We observe that NSGA-III and MOEA/D present the best convergence since their non-dominated solution sets are the closest to the ideal point, i.e., the vector composed of 7 zero. However, NSGA-III presents better diversity than all algorithms under comparisons including MOEA/D since its non-dominated solutions have a better spread varying approximately in [0, 0.88] which is not the case for MOEA/D (varying in [0, 0.8]). Besides, MOEA/D seems to have a better convergence than IBEA. However, NSGA-II is unable to progress in terms of convergence as its non-dominated solutions are so far from the ideal vector, and even it diversity is so reduced which may explain the stagnation of its evolutionary process. We conclude that although NSGA-II is the most famous multi-objective algorithm in SBSE, it is not adequate for problems involving over 3 objectives. Based on the results we obtained, it appears that NSGA-III is a very good candidate solution for tackling many-objective SBSE problems.

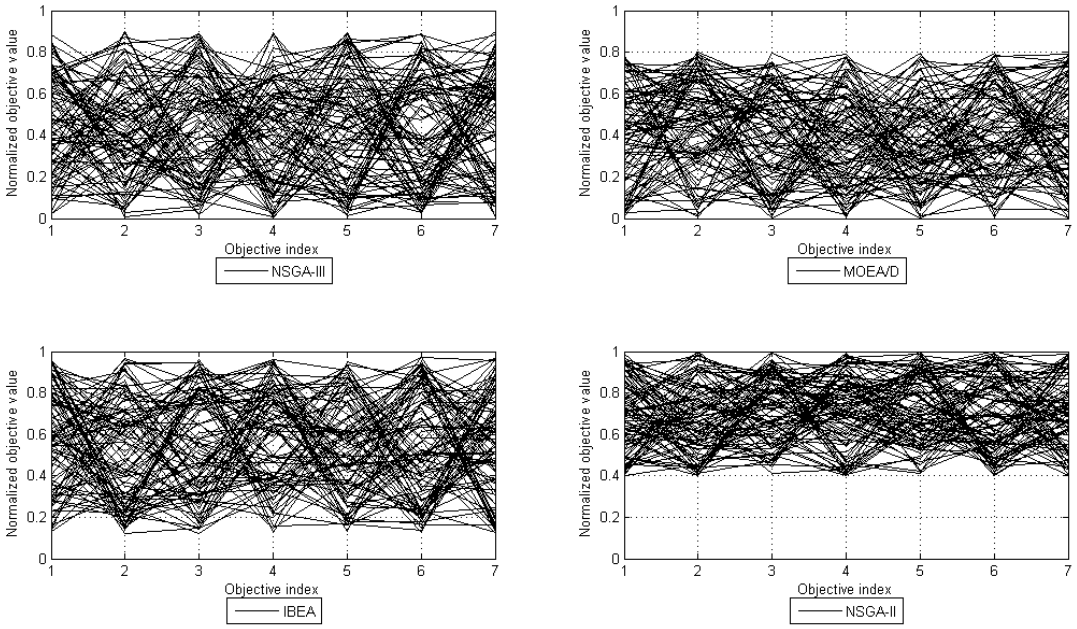


Figure 4.12. Value path plots of non-dominated solutions obtained by NSGA-III, MOEA/D, IBEA and NSGA-II during the median run of the 7-objective remodularization problem on JDI-Ford.

The results of Figure 4.12 are not sufficient to show that the 7 objectives are conflicting. Table 11 presents the results of studying the conflict relation between each pair of objectives. In fact, for each experiment (each plot), we execute a mono-objective GA minimizing the objective shown on the line label and we study the behavior of the other objective shown in the column label by recording its values at the beginning (generation 0) and at the end (generation 300) of the evolutionary process. For example, for the first plot (NCP, NP), we observe that the minimization of NCP causes the maximization of NP, thus NCP and NP are conflicting. The opposite phenomenon could be seen for the plot (NP, CHC). Indeed, the minimization of NP makes CHC decreasing too. We conclude that NP and CHC are in support (non-conflicting). To sum up, there are some pairs of objectives that are in conflict and some others that are in support. For this reason, we should verify whether there are some redundant objectives or not [97], i.e., whether there are some objectives that could be omitted while preserving the Pareto dominance order. Based on Table 4.10 results, we draw Table 4.11 which illustrates for each objective *i*, the objectives that are in conflict with it. We observe from this table that each objective *i* has a set of conflicting objectives that is different from all other objectives' ones, which means that we cannot omit any objective when comparing between any pair of solutions. Indeed, a redundant

objective could appear if two non-conflicting objectives have the same set of conflicting objectives which is not the case for our remodularization problem. Thus, we can say that the latter well involves 7 objectives to be optimized simultaneously without omitting anyone of them.

Table 4.10. Conflict study between objectives.

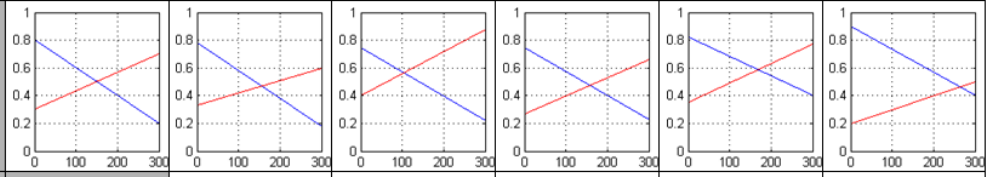
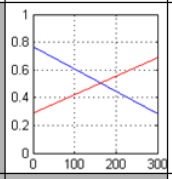
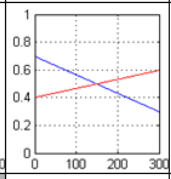
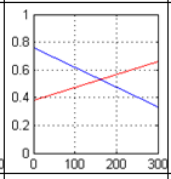
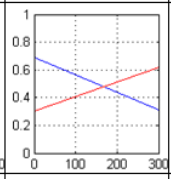
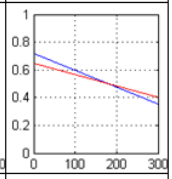
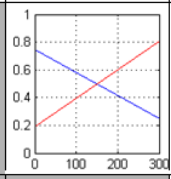
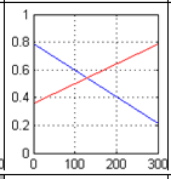
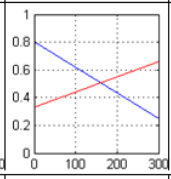
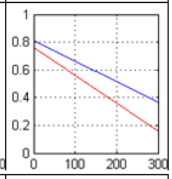
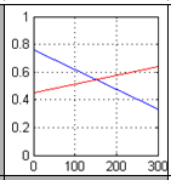
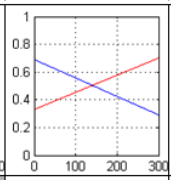
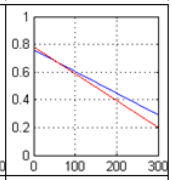
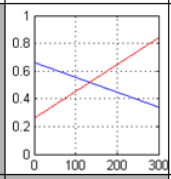
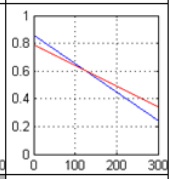
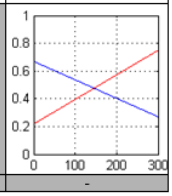
	NCP	NP	COU	COH	SP	NCH	CHC
NCP	-						
NP	-	-					
COU	-	-	-				
COH	-	-	-	-			
SP	-	-	-	-	-		
NCH	-	-	-	-	-	-	
CHC	-	-					-

Table 4.11. Existing conflict between objectives.

Objective *i*	Conflicting objectives with *i*
NCP	NP, COU, COH, SP, NCH.
NP	COU, NCP, COH, SP, NCH.
COU	NCP, NP, COH, SP, NCH.
COH	NCP, NP, COU, SP, NCH.
SP	NCP, NP, COU, COH, NCH.
NCH	NCP, NP, COU, COH, SP, CHC.
CHC	NCP, NCH.

The Wilcoxon rank sum test allows verifying whether the results are statistically different or not. However, it does not give any idea about the difference magnitude. The effect size could be computed by using the Cohen's *d* statistic [98]. The effect size is considered: (1) small if $0.2 \leq d < 0.5$, (2) medium if $0.5 \leq d < 0.8$, or (3) large if $d \geq 0.8$. For all experiments, we obtained a large difference between NSGA-III/IBEA/MOEA/D results and NSGA-II ones for the cases of 5 and 7 objectives using all the evaluation metrics. The same difference is small for the case of 3 objectives. However, when comparing NSGA-III against MOEA/D and IBEA, we have found the following results: a) On small and medium scale Software systems (JFreeChart and GanttProject) NSGA-III is better that MOEA/D and IBEA on most systems with a medium effect size; b) On large scale Software systems (Xerces-J and JDI-Ford), NSGA-III is better than MOEA/D and IBEA on most systems with a small effect size.

When using optimization techniques, the most time-consuming operation is the evaluation step. Thus, we studied the execution time of all many/multi-objective algorithms used in our experiments. Figure 4.13 shows the evolution of the execution time of the different algorithms on the JDI-Ford system, one of the largest systems in our experiments. The results show that the execution time grows linearly with respect to the number of objectives. It is clear from this figure, that all the algorithms have similar running times for the 3-objective cases. However, for a higher number of objectives NSGA-III is faster than IBEA. This observation could be explained by the computational effort required to compute the contribution of each solution in terms of hypervolume. In comparison to MOEA/D, MOEA/D is slightly faster than NSGA-III since it does not make use of non-dominated sorting.

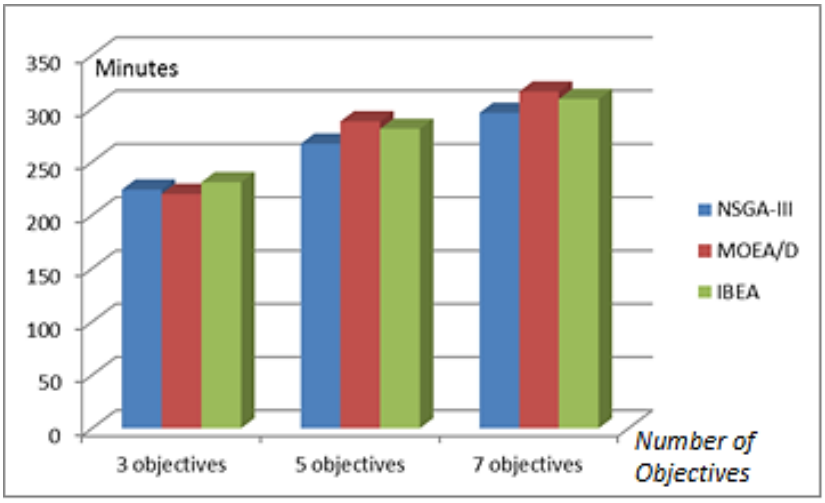


Figure 4.13. Computational time of the different used many-objective remodularization algorithms.

To further evaluate the scalability of the performance of our many-objective approach on systems of increasing size, we executed our remodularization tool on the Eclipse system without assessing the quality of the results. Eclipse is an open source integrated development environment (IDE) written in Java and widely used to develop applications. We considered four versions of Eclipse that contains more than 3 MLOCs. Figure 4.14 describes the execution time of our many-objective approach on 4 different versions of Eclipse using the 7 objectives. We believe that an execution time of 8 hours is still acceptable and reasonable even with the considered huge system to remodularize. Developers can execute our tool overnight then evaluate the results and work next day on the new system.

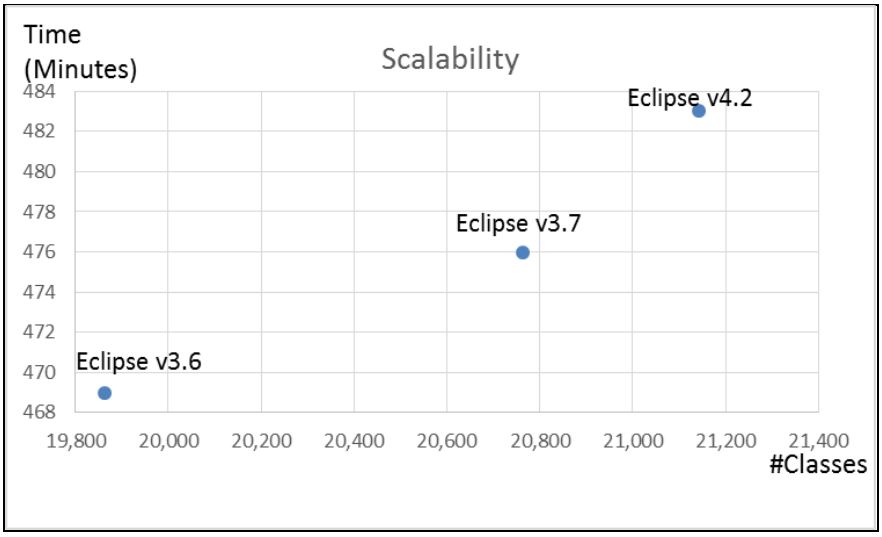


Figure 4.14. Scalability of our remodularization tool tested on Eclipse.

4.3.4.6 Results for RQ3

We compared the results of our proposal with an existing non-search-based work of Bavota et al. [78] that eventually relies on the use of coupling and cohesion along with semantic measures. In addition, the user needs to give as an input the different packages to restructure. We first note that it (like mono-objective approaches also) provides only one remodularization solution, while NSGA-III generate a set of non-dominated solutions. In order to make meaningful comparisons, we select the best solution for NSGA-III using a knee point strategy. For Bavota et al. study, we use the best solution corresponding to the median observation on 31 runs. The results from the 31 runs are depicted in Figure 4.7 and Figure 4.8, and Table 4.8. It can be seen that NSGA-III provides better results than Bavota et al. in all systems as discussed in the previous sections. As described in Figure 4.15, the NSGA-III outperforms it mainly due the use of the history of changes as a helper objective for the semantic measures. In addition, Bavota et al. work is limited to few types of operations and do not consider all the operation types supported by our approach.

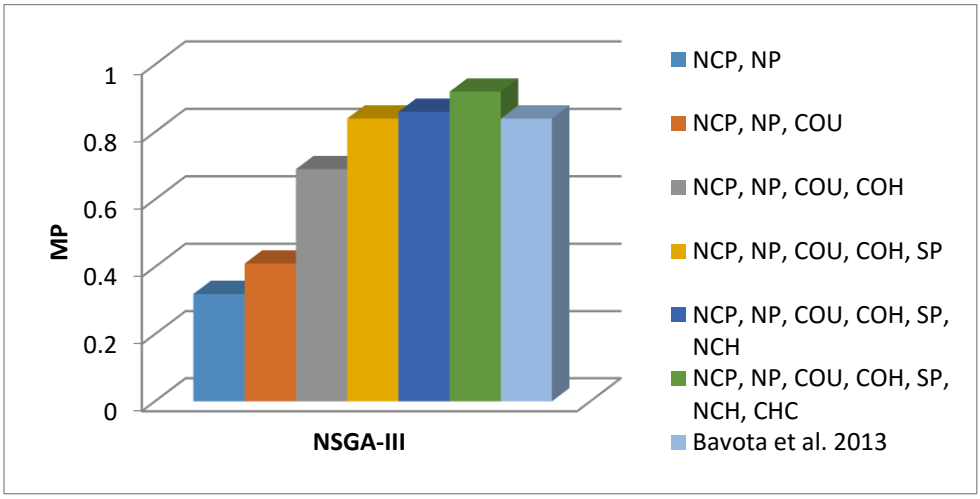


Figure 4.15. A comparison between Bavota et al. 2013 and NSGA-III based on the qualitative evaluation (MP).

4.3.4.7 Results for RQ4

We asked the software engineers involved in our experiments to evaluate the usefulness of the suggested ROs to apply one by one. In fact, sometimes these operations can improve the structure and preserve the semantics but developers will consider them as not useful due to many reasons such as some packages are not used/updated anymore or includes some features that are not important. Figure 4.16 shows that NSGA-III clearly outperforms existing work by suggesting useful remodularization operations for developers. This is can be explained mainly by the use of the history of recorded changes when suggested remodularization solutions. In fact, the use of the history of changes can help our technique to identify which packages are widely updated. In addition, several recent empirical studies showed that repetitive changes are common during the development of systems [89]. We found, in our experiments, that several patterns of changes (applied to different code locations) are repetitive. For example, move methods are in general applied after extract class since several methods are moved to the newly created class. A similar observation is valid for extract package and move class. Thus, the use of the history of changes can guide the search for good remodularization solutions based on the reuse of patterns identified from the history of changes.

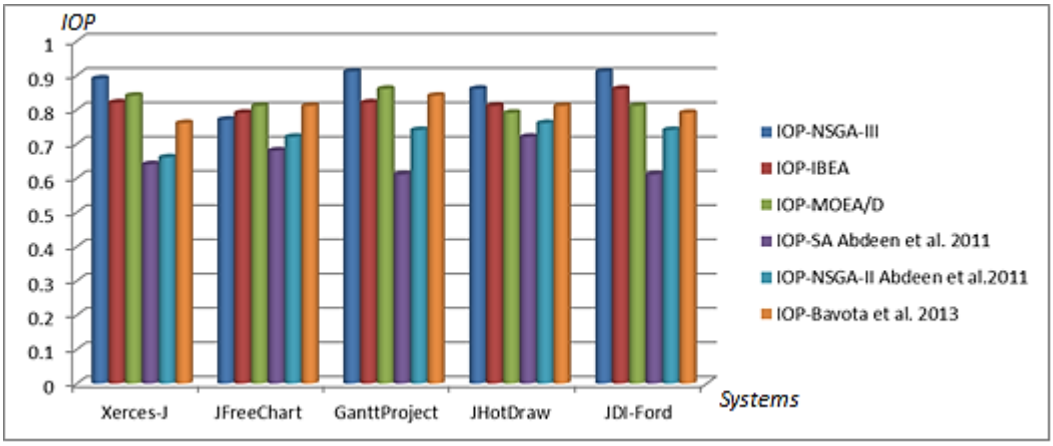


Figure 4.16. Qualitative evaluation of the suggested remodularization solutions in terms of usefulness.

Another feature that the software engineers, involved in our experiments, found it interesting is the use of several types of ROs. Figure 4.17 describes the distribution of the operations types used by the best solutions in all the system. It is clear that the three most important ones are move method, move class and extract/split packages. The software engineers found the idea very useful of moving methods between classes located in different packages or extracting a class then moving it to another class instead of moving the whole initial class to a new package. Sometimes, it is enough to move only a method from a class to another class in order to improve the cohesion of a package or decrease coupling between packages. However, existing remodularization work are limited to only two types of operations (move class and split packages).

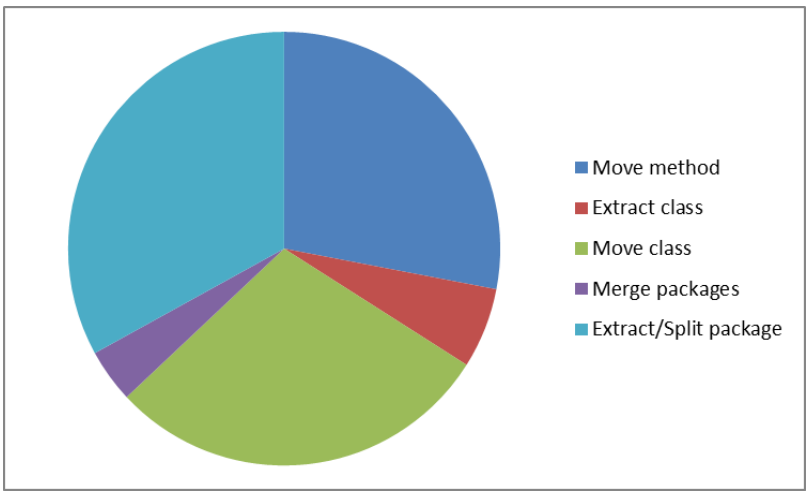


Figure 4.17. Distribution of the types of suggested remodularization operations.

During the survey, the software engineers confirm that the main limitation related to the use of NSGA-III for software remodularization is the high number of equivalent solutions. However, found the idea of the use of the Knee point as described previously useful to select a good solution. We will investigate in our future work different other techniques to select the region of interest based on the preferences of developers.

4.4 Conclusion

In this chapter, we introduced a new scalable search-based software engineering approach for software remodularization based on NSGA-III. We address several challenges of existing software remodularization techniques that are limited to mainly the use of coupling and cohesion, and few types of operations (move class and split package). Our proposal aims at finding the remodularization solution that improves the structure of packages by optimizing some metrics such as number of classes per package, number of packages, coupling and cohesion; improve the semantic coherence of the restructured program; minimize code changes, and maximize the consistency with development change history.

We evaluated our approach on four open source systems and one industrial system provided by our industrial partner Ford Motor Company. We report the results on the efficiency and effectiveness of our approach, compared to the state of the art remodularization approaches. Our results indicate that our approach significantly outperforms, in average, existing approaches based on a quantitative and qualitative evaluation. All the results were statistically significant on the 31 independent simulations using the Wilcoxon rank sum test with a 99% confidence level ($\alpha < 1\%$) where more than 92% of code-smells were fixed on the different open source systems.

As part of the future work, we plan to work on adapting NSGA-III to additional software engineering problems and we will perform more comparative studies on larger open source systems. Furthermore, we will investigate the impact of different parameter settings on the quality of our results. Nevertheless, this extensive study has shown a direction using NSGA-III to handle as many as 7 objectives in the context of solving software engineering problems and would remain as one of the first studies in which such a large number of objectives have been considered. Weighting certain objectives can be an interesting future work direction to integrate

the preferences of developers during the search process. We are also planning to consider only the remodularization of the modified packages after the last release based on analysis of the history of the code changes. Finally, we plan to extend the use of our modularization approach by additional experts to generalize the obtained results.

Chapter 5: Software Refactoring

5.1 Multi-objective Optimization under Uncertainty

5.1.1 Introduction

Software evolution is an essential component of software development process. It is mainly intended to keep the software system up to the user's requirements by regularly performing a set of development activities linked to adding new features, fixing reported bugs, migrating to different environments and platforms, and other function related tasks. Eventually. As a software system evolves, the code-related changes tend to degrade the system's structure as the evolution focuses mainly on the incorporation of required features and the correction of errors on the expense of the deterioration of the system's design. Since these code changes have become inevitable as they constitute a key role in agile methodologies, strategies need to be adapted in order to preserve the software architecture's value over changes. Therefore, software maintenance gathers software change control and management strategies that aim to maintain the software quality during its evolution. On the other hand, the new functionalities, incorporated during the evolution of a software system, exhibit a growth of its size and complexity and so it becomes harder to maintain. As a consequence, the cost of maintenance and evolution activities comprises more than 80% of total software costs. In addition, it has been shown that software maintainers spend around 60% of their time in understanding the code [99]. To facilitate maintenance tasks, one of the widely used techniques is *refactoring* which can be defined as the restructuration of the system's architecture with the intention of improving its internal design while preserving the overall external behavior of the software [15].

Software refactoring, as a concept, has been introduced in the early nineties by Obdyke [87] and became a key artifact of the agile development processes such as Extreme Programming

(XP). Fowler [15] has identified refactoring opportunities within code fragments and provides a refactoring operations catalog that can be applied to enhance the code's structure of code while preserving its semantics. There has been much work on different refactoring techniques and tools [50] [62] [71] [86]. The vast majority of these techniques identify key symptoms that characterize the code to refactor using a combination of quantitative, structural, and/or lexical information and then propose different possible refactoring solutions, for each identified segment of code. In order to find out which parts of the source code need to be refactored, most of the existing work rely on the notion of design defects or code smells. Originally coined by Fowler, the generic term code smell refers to structures in the code that suggest the possibility of refactoring. Once code smells have been identified, refactorings need to be proposed to resolve them. Several automated refactoring approaches are proposed in the literature and most of them are based on the use of software metrics to estimate quality improvements of the system after applying refactorings.

Most of existing approaches propose refactoring solutions without a consideration of the severity and importance of detected refactoring opportunities to fix. In fact, both severity and importance are difficult to define and estimate. The estimation scores of these factors can change during the time due to the highly dynamic nature of software development. The importance and severity of code fragments can be different after new commits introduced by developers. Thus, it is important to consider the uncertainty related to these two factors when recommending refactoring solutions. In addition, the definition of severity and importance is very subjective and depends on the developers' perception.

In this work, we take into account two dynamic aspects as follows:

- *Code Smell Severity*: Once a list of code smells is detected, the correction techniques treat these detected defects equally by suggesting which refactorings could be applied to the code in order to eliminate, or at least reducing them. Whereas, the effects of a defect in terms of potential introduction of faults may vary depending on the type of the code smell [100]. Also, many studies have been investigating the impact of each defect type on the maintenance effort [101]. Thus, we consider a severity level assigned to a code smell type by a developer based on his prior knowledge and his

preference on prioritizing the correction of a specific type of code smell among others. This is the severity level assigned to a code smell type by a developer. It usually varies from developer to developer, and indeed, a developer's assessment of smell severity will change over time as well.

- *Code Smell Class Importance*: This is the importance of a class that contains a code smell, where importance refers to the number and size of the features that the class supports. A code smell with large class importance will have a greater detrimental impact on the software. Again, this property will vary over time as software requirements change [5] and classes are added/deleted/extracted.

We believe that the uncertainties related to the class importance and the code smell severity need to be taken into consideration when suggesting a refactoring solution. To this end, we introduce a novel representation of the code refactoring problem, based on *robust* optimization [102] [103] that generates robust refactoring solutions by taking into account the uncertainties related to code smell severity and the importance of the class that contains the code smell. Our robustness model is based on the well-known multi-objective evolutionary algorithm NSGA-II proposed by Deb et al. [83] and considers possible changes in class importance and code smell severity by generating different scenarios at each iteration of the algorithm. In each scenario, the detected code smell to be corrected is assigned a severity score and each class in the system is assigned an importance score. In our model, we assume that these scores change regularly due to reasons such as developers' evolving perspectives on the software or new features and requirements being implemented or any other code changes that could make some classes/code smells more or less important. Our multi-objective approach aims to find the best trade-off between three objectives to maximize: quality improvements (number of fixed code smells), severity and importance of refactoring opportunities to be fixed (e.g. code smells).

The primary contributions of this work are as follows:

- A novel formulation of the refactoring problem as a multi-objective problem that takes into account the uncertainties related to code smell detection and the dynamic environment of software development. We extended our previous work [104] by considering severity and importance of refactoring opportunities as separate objectives.

In addition, we are considering additional types of code smells to fix and extended our validation to 8 open source systems and one industrial project.

- The Validation of an empirical study of our robust NSGA-II technique as applied different medium and large size systems. We compared our approach to random search, multi-objective particle swarm optimization (MOPSO) [105], search-based refactoring [70] [52] and a non-search-based refactoring tool [106]. The results provide evidence to support the claim that our proposal enables the generation of robust refactoring solutions without a high loss of quality using a variety of real-world scenarios.

5.1.2 Approach

5.1.2.1 Robust Optimization

In dealing with optimization problems, including software engineering ones, most researchers assume that the parameters of the problem are exactly known in advance. Unfortunately, this is an idealization often not the case in a real-world setting. Additionally, uncertainty can change the effective values of some parameters with respect to nominal values. For instance, when handling the knapsack problem (KP), which is one of the most studied combinatorial problems [103], we can face such a problem. The KP problem requires one to find the optimal subset of items to put in a knapsack of capacity C in order to maximize the total profit while respecting the capacity C. The items are selected from an item set where each item has its own weight and its own profit. Usually, the KP's input parameters are not known with certainty in advance. Consequently, we should search for *robust* solutions that are *immune* to small perturbations in terms of input parameter values. In other words, we prefer solutions whose performance levels do not significantly degrade due to small perturbations in one or several input parameters such as item weights, item profits and knapsack capacity for a KP. As stated by Beyer et al. [103], uncertainty is unavoidable in real problem settings; therefore, it should be taken into account in every optimization approach in order to obtain robust solutions. Robustness of an optimal solution can usually be discussed from the following two perspectives: (1) the optimal solution is insensitive to small perturbations in terms of the decision variables and/or (2) the optimal solution is insensitive to small variations in terms of environmental parameters.

Figure 5.1 illustrates the robustness concept with respect to a single decision variable named *x*. Based on the *f* (*x*) landscape, we have two optima: *A* and *B*. We remark that solution *A* is very sensitive to local perturbation of the variable *x*. A very slight perturbation of *x* within the interval [2, 4] can make the optimum *A* unacceptable since its performance *f*(*A*) would dramatically degrade. On the other hand, small perturbations of the optimum *B*, which has a relatively lower objective function value than *A*, within the interval [5,7] hardly affects the performance of solution *B* (i.e., *f*(*B*)) at all. We can say that although solution *A* has a better quality than solution *B*, solution *B* is more *robust* than solution *A*. In an uncertain context, the user would probably prefer solution *B* to solution *A*. This choice is justified by the performance of *B* in terms of robustness. It is clear from this discussion robustness has a price, called *robustness price or cost*, since it engenders a *loss in optimality*. This loss is due to preferring the robust solution *B* over the non-robust solution *A*. According to Figure 1, this loss is equal to $abs(f(B) - f(A))$. Several approaches have been proposed to handle robustness in the optimization field in general and more specifically in design engineering. These approaches can be classified as follows [102]:

- *Explicit averaging*: Assuming *f*(*x*) to be the fitness function of solution *x*, the basic idea is to generate a weighted average for the fitness value in the neighborhood $B_\delta(x)$ of solution *x* with an uncertainty distribution $p(\delta)$. The fitness function then becomes: $\int_{\delta \in B_\delta(x)} p(\delta) f(x+\delta) d\delta$

However, because in the robustness term, case disturbances can be chosen deliberately, variance reduction techniques can be applied, allowing a more accurate estimation with fewer samples.

- *Implicit averaging*: The basic idea is to compute an expected fitness function based on the fitness values of some solutions residing within the neighborhood of the considered solution. Beyer et al. [107] noted that it is important to use a large population size in this case. In fact, given a fixed number of evaluations per generation, it was found that increasing the population size yields better results than multiple random sampling.

- *Use of constraints*: The difference between the weighted average fitness value and the actual fitness value at any point can be restricted to lie within a pre-defined threshold in order to yield more robust solutions [108]. Such an optimization process will prefer robust solutions to

the problem and it then depends on the efficacy of the optimization algorithm to find the highest quality robust solution.

• *Multi-objective formulation*: The core idea in the multi-objective approach is to use an additional helper objective function that handles robustness related to uncertainty in the problem's parameters and/or decision variables. Thus, we can solve a mono-objective problem by means of a multi-objective approach by adding robustness as a new objective to the problem at hand. The same idea could be used to handle robustness in a multi-objective problem; however, in this case, the problem's dimensionality would increase. Another reason to separate fitness from robustness is that using the expected fitness function as a basis for robustness is not sufficient in some cases [109]. With expected fitness as the only objective, positive and negative deviations from the true fitness can cancel each other in the neighborhood of the considered solution. Thus, a solution with high fitness variance may be wrongly considered to be robust. For this reason, it may be advantageous to consider expected fitness and fitness variance as separate additional optimization criteria, which allows searching for solutions with different trade-offs between performance and robustness.

Robustness has a cost in terms of the loss in optimality, termed the *price of robustness*, or sometimes *robustness cost*. This is usually expressed as the ratio between the gain in robustness and the loss in optimality. Robustness handling methods have been successfully applied in several engineering disciplines such as scheduling, electronic engineering and chemistry (cf. [103] for a comprehensive survey).

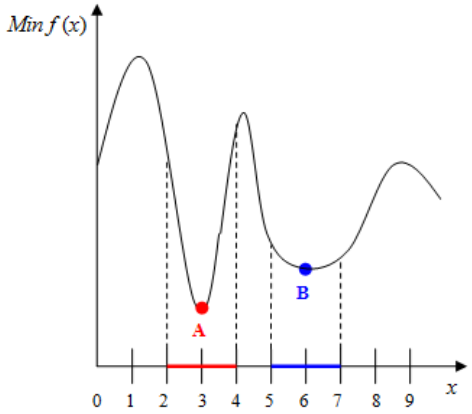


Figure 5.1. Illustration of the robustness concept under uncertainty related to the decision variable x. Solution B is more robust than solution A.

5.1.2.2 Problem Formulation

The refactoring problem, from the search-based perspective, includes the exploration of a set of candidate solutions in order to determine the best one whose sequence of refactorings best satisfies the fitness function(s). A refactoring solution is a set of refactoring operations where the goal of applying the sequence to a software system *S* is typically to minimize the number of design defects in *S*. As outlined in the Introduction, in a real-world setting code smell severity and class importance are not certainties. A refactoring sequence that resolves the smells that one developer rates as severe may not be viewed as effective by another developer with a different outlook on smells. Similarly, a refactoring sequence that fixes the smells in a class that is subsequently deleted in the next commit is not of much value [110].

We propose a robust formulation of the refactoring problem that separates class importance and smells severity into two different objectives. Consequently, we have three objective functions to be maximized in our problem formulation: (1) the quality of the system to refactor, i.e., minimizing the number of code smells, and the robustness of the refactoring solutions in relation to uncertainty in both (2) severity level of the code smells and (3) the importance of the classes that contain the code smells. Analytically speaking, the formulation of the robust refactoring problem can be stated as follows:

$$
\begin{aligned}
&\textit{Maximize} \\
&\begin{cases}
f_1(x,S) = NCCS(x,S)/NDCS(S) \\
f_2(x,S) = \sum_{i=1}^{NCCS(x,S)} SmellSeverity(CCS_i, x, S) \\
f_3(x,S) = \sum_{i=1}^{NCCS(x,S)} ClassImportance(CCS_i, x, S)
\end{cases} \\
&\textit{subject to} \quad x = (x_1, ..., x_n) \in X
\end{aligned}
\tag{5.1}
$$

where X is the set of all legal refactoring sequences starting from S, x_i is the i^{th} refactoring in the sequence x, $NCCS(x,S)$ is the *Number of Corrected Code Smells* after applying the refactoring solution x on the system S, $NDCS(S)$ is the *Number of Detected Code-Smells* prior to the application of solution x to the system S, CCS_i is the i^{th} Corrected Code Smell, $SmellSeverity(CCS_i, x, S)$ is the severity level of the i^{th} corrected code smell related to the

execution of *x* on *S*, and *ClassImportance*(*CCS*$_i$, *x*, *S*) is the importance of the class containing the i^{th} code smell corrected by the execution of *x* on *S*.

The smell's severity level is a numeric quantity, varying between 0 and 1, assigned by the developer to each code smell type (e.g., blob, spaghetti code, functional decomposition, etc.). We define the class importance of a code smell as follows:

$$Importance\ (CCS_i, x, S) = \frac{(NC\ /\ MaxNC\ (S)) + (NR\ /\ MaxNR\ (S)) + (NM\ /\ MaxNM\ (S))}{3} \qquad (5.2)$$

such that *NC*/*NR*/*NM* correspond respectively to the *Number of Comments/Relationships/Methods* related to the *CCS*$_i$ and *MaxNC*/*MaxNR*/*MaxNM* correspond respectively to the *Maximum Number of Comments/Relationships/Methods* of any class in the system *S*. There are of course many ways in which class importance could be measured, and one of the advantages of the search-based approach is that this definition could be easily replaced with a different one. We used in our approach the widely used metrics by existing literature to estimate the importance and severity of code smells [111]. In fact, few empirical studies were performed with software engineers to evaluate the severity and importance of several types of code smells [111] [112]. We are taking these metrics and the severity scores as input for our approach.

In summary, the basic idea behind this work is to maximize the resistance of the refactoring solutions to perturbations in the severity levels and class importance of the code smells while maximizing simultaneously the number of corrected code smells. These three objectives are in conflict with each other since the quality of the proposed refactoring solution usually decreases when the environmental change (smell severity and/or class importance) increases. In addition, severe quality issues may exist in code fragments that are not important for developers. Thus, the goal is to find a good compromise between these three conflicting objectives. This compromise is directly related to robustness cost, as discussed above. In fact, once the three objectives trade-off front is obtained, the developer can navigate through this front in order to select his/her preferred refactoring solution. This is achieved through sacrificing some degree of solution quality while gaining in terms of robustness and smell severity and importance. In fact, robustness is inversely proportional to the severity and class importance, and this is how we

make our algorithm robust. The approach is to use a multi-objective search algorithm to explore the trade-off that can be obtained by ignoring the user's wishes to varying degrees. The results show this to be the case; quality can be most improved when importance and severity objective values are lower. For example, a refactoring applied to a class that is then deleted; the instantaneous quality improvement may be high, but in the project's timeline it is minimal. This makes a solid case for so-called "robust" optimization which seeks to identify good solutions that are resilient against changes in the developer's priorities.

In our robust formulation, we introduce *perturbations/variations* in the severity and importance scores of the code smells and classes at every iteration of our NSGA-II algorithm. As described later in the experiments section, the severity scores of 0.8, 0.6, 0.4, 0.3, 0.5, 0.3, and 0.2 of the different types of code represent just the initial values but these values slightly change at each iteration (a slight random increase or decrease of these scores using a variation rate parameter). These variations correspond to some artificially created changes in the environment (new code changes introduced, etc.) or priorities change or different opinions of developers about the importance or severity of the classes.

5.1.2.3 The Solution Approach

Solution representation. To represent a candidate solution (individual/chromosome), we use a vector-based representation which is widely adopted in the literature. According to this encoding, a solution is an array of refactoring operations where the order of their execution is accorded by their positions in the array. The standard approach of pre- and post-conditions is used to ensure that the refactoring operation can be applied while preserving program behavior. For each refactoring operation, a set of controlling parameters (e.g., actors and roles as illustrated in Table 1) is randomly picked from the program to be refactored. Assigning randomly a sequence of refactorings to certain code fragments generates the initial population. To apply a refactoring operation we need to specify which actors, i.e., code fragments, are involved/impacted by this refactoring and which roles they play to perform the refactoring operation. An actor can be a package, class, field, method, parameter, statement or variable. Table 1 depicts, for each refactoring, its involved actors and its role.

Table 5.1. Refactoring types and their involved actors and roles.

Refactorings	Actors	Roles
Move method	class	source class, target class
	method	moved method
Move field	class	source class, target class
	field	moved field
Pull up field	class	sub classes, super class
	field	moved field
Pull up method	class	sub classes, super class
	method	moved method
Push down field	class	super class, sub classes
	field	moved field
Push down method	class	super class, sub classes
	method	moved method
Inline class	class	source class, target class
Extract class	class	source class, new class
	field	moved fields
	method	moved methods
Move class	package	source package, target package
	class	moved class
Extract interface	class	source classes, new interface
	field	moved fields
	method	moved methods

Solution variation. For 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. This operator must respect the refactoring sequence length limits by eliminating randomly some refactoring operations if necessary. For mutation, we use the bit-string mutation operator that picks probabilistically one or more refactoring operations from its or their associated sequence and replaces them by other ones from a list of possible refactorings. These two variation operators have already demonstrated good performance when tackling the refactoring problem [50] [86].

Solution evaluation. Each refactoring sequence in the population is executed on the system *S*. For each sequence, the solution is evaluated based on the three objective functions defined in the previous section. Since we are considering a three objectives formulation, we use the concept of Pareto optimality to find a set of compromise (Pareto-optimal) refactoring solutions.

By definition, a solution *x* Pareto-dominates a solution *y* if and only if *x* is at least as good as *y* in all objectives and strictly better than *y* in at least one objective. The fitness of a particular solution in NSGA-II corresponds to a couple (*Pareto Rank*, *Crowding distance*). In fact, NSGA-II classifies the population individuals (of parents and children) into different layers, called non-

dominated fronts. Non-dominated solutions are assigned a rank of 1 and then are discarded temporary from the population. Non-dominated solutions from the truncated population are assigned a rank of 2 and then discarded temporarily. This process is repeated until the entire population is classified with the domination metric.

After that, a diversity measure, called *crowding distance*, is assigned front-wise to each individual. The crowding distance is the average side length of the cuboid formed by the nearest neighbors of the considered solution. Once each solution is assigned its Pareto rank, based on refactoring quality and robustness to change in terms of class importance and smell severity levels, in addition to its crowding distance, mating selection and environmental selection are performed. This is based on the crowded comparison operator that favors solutions having better Pareto ranks and, in the case of equal ranks, it favors the solution having larger crowding distance. In this way, convergence towards the Pareto optimal bi-objective front (quality, robustness) and diversity along this front are emphasized simultaneously. The basic iteration of NSGA-II consists in generating an offspring population (of size N) from the parent one (of size N too) based on variation operators (crossover and mutation) where the parent individuals are selected based on the crowded comparison operator. After that, parents and children are merged into a single population R of size $2N$. The parent population for the next generation is composed of the best non-dominated fronts. This process continues until the satisfaction of a stopping criterion. The output of NSGA-II is the last obtained parent population containing the best of the non-dominated solutions found. When plotted in the objective space, they form the Pareto front from which the developer will select his/her preferred refactoring solution.

5.1.3 Validation

5.1.3.1 Design of the Empirical Study

This section describes our empirical study including the research questions to address, the systems examined, evaluation metrics considered in our experiments and the statistical tests results. In addition, we compared different refactoring algorithms quantitatively and qualitatively across several subject systems.

We defined six research questions that address the applicability, performance in comparison to existing refactoring approaches, and the usefulness of our robust multi-objective refactoring. The six research questions are as follows:

RQ1: Search Validation. To validate the problem formulation of our approach, we compared our NSGA-II formulation with Random Search. If Random Search outperforms an intelligent search method thus we can conclude that our problem formulation is not adequate. Since outperforming a random search is not sufficient, the next four questions are related to the comparison between our proposal and the state-of-the-art refactoring approaches.

RQ2.1: How does NSGA-II perform compared to other multi-objective algorithms? It is important to justify the use of NSGA-II for the problem of refactoring under uncertainties. We compare NSGA-II with another widely used multi-objective algorithm, MOPSO (Multi-Objective Particle Swarm Optimization), using the same adaptation (fitness functions). In addition, we compared our approach to our previous robust refactoring study (based on only 2 objectives) [26].

RQ2.2: How do robust, multi-objective algorithms perform compared to mono-objective Evolutionary Algorithms? A multi-objective algorithm provides a trade-off between the two objectives where the developers can select their desired solution from the Pareto-optimal front. A mono-objective approach uses a single fitness function that is formed as an aggregation of both objectives and generates as output only one refactoring solution. This comparison is required to ensure that the refactoring solutions provided by NSGA-II and MOPSO provide a better trade-off between quality and robustness than a mono-objective approach. Otherwise, there is no benefit to our multi-objective adaptation.

RQ2.3: How does NSGA-II perform compare to existing search-based refactoring approaches? Our proposal is the first work that treats the problem of refactoring under uncertainties. A comparison with existing search-based refactoring approaches [71] [70] is helpful to evaluate the cost of robustness of our proposed approach.

RQ2.4: How does NSGA-II perform compared to existing refactoring approaches not based on the use of metaheuristic search? While it is very interesting to show that our proposal

outperforms existing search-based refactoring approaches, developers will consider our approach useful, if it can outperform other existing refactoring tools [106] that are not based on optimization techniques.

RQ3: Developers' evaluation of the recommended refactorings. Can our robust multi-objective approach be useful for software engineers in a real-world setting? In a real-world problem involving uncertainties, it is important to show that a robustness-unaware methodology drives the search to non-robust solutions that are sensitive to variation in the uncertainty parameters. However when robustness is taken into account, a more robust and insensitive solution is found. Some scenarios are required to illustrate the importance of robust refactoring solutions in a real-world setting: a) exploring the trade-off of robust and qualitative solutions, and b) asking what developers think of the results.

Table 5.2. Software studied in our experiments.

Systems	Release	#Classes	#Smells	KLOC
Xerces-J	v2.7.0	991	82	240
JFreeChart	v1.0.9	521	73	170
GanttProject	v1.10.2	245	56	41
ApacheAnt	v1.8.2	1191	91	255
JHotDraw	v6.1	585	33	21
Rhino	v1.7R1	305	74	42
Log4J	v1.2.1	189	64	31
Nutch	v1.1	207	72	39
JDI-Ford	v5.8	638	88	247

Software Projects Studied. In our experiments, we used a set of well-known and well-commented open-source Java projects. We applied our approach to eight large and medium-sized open source Java projects: Xerces-J [113], JFreeChart [114], GanttProject [115], ApacheAnt [116], JHotDraw [117], Rhino [118], Log4J [119], Nutch [120] and JDI-Ford. 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. ApacheAnt is a build tool and library specifically conceived for Java applications. JHotDraw is a GUI framework for drawing editors. Rhino is a JavaScript interpreter and compiler written in Java and developed for the Mozilla/Firefox browser. Nutch is an open source Java implementation of a search engine. Log4j is a Java-based logging utility. Table 2 provides some descriptive statistics about these eight programs. We selected these systems for our validation because they

range from medium to large-sized open source projects that have been actively developed over the past 10 years, and include a large number of code smells. In addition, these systems are well studied in the literature and their code smells have been either detected, analyzed manually [71] [70] or using an existing detection tool [54]. In these corpuses, the four following code smell types were identified manually (that are described in Section 2): Blob, Feature Envy (FE), Data Class (DC), Spaghetti Code (SC), Functional Decomposition (FD), Lazy Class (LC) and Long Parameter List (LPL). We chose these code smell types in our experiments because they are the most frequent ones detected and corrected in recent studies in existing corpuses.

We performed also a small industrial case study, based on one industrial project JDI-Ford. It is a Java-based software system that helps, our industrial partner, the Ford Motor Company, analyzes useful information from the past sales of dealerships data and suggests which vehicles to order for their dealer inventories in the future. This system is the main key software application used by the Ford Motor Company to improve their vehicle sales by selecting the right vehicle configuration to the expectations of customers. Several versions of JDI were proposed by software engineers at Ford during the past 10 years. Due to the importance of the application and the high number of updates performed during a period of 10 years, it is critical to making sure that all the JDI releases are within a good quality to reduce the time required by developers to introduce new features in the future. The software engineers at Ford manually evaluated the suggested refactorings based on their knowledge of the system since they are some of the original developers.

Evaluation Metrics Used. When comparing two mono-objective algorithms, it is usual to compare their best solutions found so far during the optimization process. However, this is not applicable when comparing two multi-objective evolutionary algorithms since each of them gives as output a set of non-dominated (Pareto equivalent) solutions. For this reason, we defined the following metrics:

−*Hypervolume* (*IHV*) corresponds to the proportion of the objective space that is dominated by the Pareto front approximation returned by the algorithm and delimited by a reference point. Larger values for this metric mean better performance. The most interesting features of this indicator are its Pareto dominance compliance and its ability to capture both convergence and

diversity. The reference point used in this study corresponds to the nadir point of the Reference Front (*RF*), where the Reference Front is the set of all non-dominated solutions found so far by all algorithms under comparison.

−*Inverse Generational Distance* (*IGD*) is a convergence measure that corresponds to the average Euclidean distance between the Pareto front Approximation *PA* provided by the algorithm and the reference front *RF*. The distance between *PA* and *RF* in an *M*-objective space is calculated as the average *M*-dimensional Euclidean distance between each solution in *PA* and its nearest neighbor in *RF*. Lower values for this indicator mean better performance (convergence).

−*Contribution* (*IC*) corresponds to the ratio of the number of non-dominated solutions the algorithm provides to the cardinality of *RF*. Larger values for this metric mean better performance.

In addition to these three multi-objective evaluation measures, we used these other metrics mainly to compare between mono-objective and multi-objective approaches defined as follows:

−*Quality: number of Fixed Code-Smells* (*FCS*) is the number of code smells fixed after applying the best refactoring solution.

−*Severity of fixed Code-Smells* (*SCS*) is defined as the sum of the severity of fixed code smells:

$$SCS(S) = \sum_{i=1}^{k} SmellSeverity(d_i) \qquad (5.3)$$

where *k* is the number of fixed code smells and *SmellSeverity* corresponds to the severity (value between 0 and 1) assigned by the developer to each code smell type (blob, spaghetti code, etc.). In our experiments, we use these severity scores 0.8, 0.6, 0.4, 0.3, 0.5, 0.3, and 0.2 respectively for Blob, Spaghetti Code (SC), Functional Decomposition (FD), Lazy Class (LC), Feature Envy (FE), Data Class (DC) and Long Parameter List (LPL). We introduce “perturbations/variations” in the severity and importance scores of the code smells and classes at every iteration of our NSGA-II algorithm. Thus, these severity scores of the different types of code represent just the initial values but these values slightly change at each iteration (a slight random increase or decrease of these scores using a variation rate parameter between -0.2 and +0.2).

−*Importance of fixed Code-Smells* (*ICS*) is defined using three metrics (number of comments, number of relationships and number of methods) as follows:

$$ICS(S) = \sum_{i=1}^{k} \text{importance}(d_i) \quad (5.4)$$

where *importance* is as defined in Section 5.1.2.2.

−*Correctness of the suggested Refactorings (CR)* is defined as the number of semantically correct refactorings divided by the total number of manually evaluated refactorings.

−*Computational time* (*ICT*) is a measure of efficiency employed here since robustness inclusion may cause the search to use more time in order to find a set of Pareto-optimal trade-offs between refactoring quality and solution robustness.

When we compared the different algorithms to NSGA-II, we used the original initial weights (before the perturbation).

Subjects. Our study involved 6 subjects from the University of Michigan and 4 software engineers from Ford Motor Company. Subjects include 1 master student in Software Engineering, 4 Ph.D. students in Software Engineering, 1 faculty member in Software Engineering, 3 junior software developers and 1 senior projects manager. All the subjects are familiar with Java development, software maintenance activities including refactoring. The experience of these subjects on Java programming ranged from 3 to 17 years. All the graduate students have an industrial experience of at least 2 years with large-scale object-oriented systems. The 6 subjects from the University of Michigan evaluated the refactoring results on the open source systems. The 4 software engineers from Ford evaluated the refactoring results only on the JDI-Ford system. They were selected, as part of a project funded by Ford, based on having similar development skills, their motivations to participate in the project and their availability. They are part of the original developers' team of the JDI system.

Parameter tuning and setting. An often-omitted aspect in metaheuristic search is the tuning of algorithm parameters. In fact, parameter setting influences significantly the performance of a search algorithm on a particular problem. For this reason, for each multi-objective algorithm and for each system (cf. Table 5.3), we performed a set of experiments using several population

sizes: 50, 100, 200, 500 and 1000. The stopping criterion was set to 250,000 fitness evaluations for all algorithms in order to ensure fairness of comparison. Each algorithm was executed 51 times with each configuration and then the comparison between the configurations was performed based on *IHV*, *IGD* and *IC* using the Wilcoxon test. Table 5.3 reports the best configuration obtained for each couple (algorithm, system). For the mono-objective EA, we adopted the same approach using best fitness value criterion since multi-objective metrics cannot be used for single-objective algorithms. The best configurations are also shown in Table 3.

The MOPSO used in this work is the Non-dominated Sorting PSO (NSPSO) proposed by Li [105]. The other parameters' values were fixed by trial and error and are as follows: (1) crossover probability = 0.8; mutation probability = 0.5 where the probability of gene modification is 0.3; stopping criterion = 250,000 fitness evaluations. For MOPSO, the cognitive and social scaling parameters c_1 and c_2 were both set to 2.0 and the inertia weighting coefficient w decreased gradually from 1.0 to 0.4. Since refactoring sequences usually have different lengths, we authorized the length n of number of refactorings to belong to the interval [10, 250].

Table 5.3. Best population size configurations.

System	NSGA-II	MOPSO	Mono-EA
Xerces-J	1000	1000	1000
JFreeChart	500	200	500
GanttProject	100	100	100
ApacheAnt	1000	1000	1000
JHotDraw	200	200	200
Rhino	100	200	200
Log4J	200	200	100
Nutch	100	200	150
JDI-Ford	500	600	460

Approaches in comparison. We compared our approach with different existing studies. For the mono-objective approaches, Kessentini et al. [71] used the Genetic Algorithm (GA) to find the best sequence of refactoring that minimizes the number of code smells while O'Keeffe and Ó Cinnéide [70] used different mono-objective algorithms to find the best sequence of refactorings that optimize a fitness function composed of a set of quality metrics. Kessentini et al. [71] and O'Keeffe et al. [70], where uncertainties are not taken into account. We also implemented a mono-objective GA where one fitness function is defined as an average of the three objectives (quality, severity and importance). For the multi-objective algorithms, in Ouni et al. [86], the

authors ask a set of developers to fix manually the code smells in a number of open source systems including those that we are considering in our experiments. They proposed a multi-objective approach to maximize the number of fixed defects and minimize the number of refactorings. We also compared our approach with our previous SSBSE2014 robust refactoring study (based on only 2 objectives) [104] as described in the introduction. In addition, we compared our proposal to a popular design defects detection and correction tool JDeodorant [106] that does not use heuristic search techniques. The current version of JDeodorant is implemented as an Eclipse plug-in that identifies some types of design defects using quality metrics and then proposes a list of refactoring strategies to fix them.

5.1.3.2 Results Analysis

This section describes and discusses the results obtained from the different research questions of Section 4.1. 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 performed based on 51 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 ($\alpha = 5\%$). The latter verifies the null hypothesis H_0 that the obtained results of two algorithms are samples from continuous distributions with equal medians, as against the alternative that they are not, H_1. The p-value of the Wilcoxon test corresponds to the probability of rejecting the null hypothesis H_0 while it is true (type I error). A p-value that is less than or equal to α (≤ 0.05) means that we accept H_1 and we reject H_0. 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 of random search, MOPSO and mono-objective search results, our previous multi-objective work [104] with NSGA-II ones. In this way, we could decide whether the outperformance of NSGA-II over one of each of the others (or the opposite) is statistically significant or just a random result. The inference on the best result is done on the basis of ranking through multiple pair-wise tests.

The Wilcoxon signed-rank test allows verifying whether the results are statistically different or not. However, it does not give any idea about the difference in magnitude. To this end, we used the *Vargha and Delaney's A* statistics which is a non-parametric effect size measure [121].

In our context, given the different performance metrics (such as FCS, SCS and ICS), the *A* statistics measures the probability that running an algorithm B_1 (our robust multi-objective algorithm) yields better performance than running another algorithm B_2 (such as random search and MOPSO). If the two algorithms are equivalent, then $A = 0.5$. In our experiments, we have found the following results: a) On small and medium scale Software systems (JFreeChart and GanttProject) NSGA-II is better than all the other algorithms based on all the performance metrics with an *A*-effect size higher than 0.9; b) On large scale Software systems (Xerces-J and JDI-Ford), NSGA-II is better than all the other algorithms with a an *A*-effect size higher than 0.83.

Results for RQ1: Comparison between NSGA-II and Random Search.

To answer the first research question RQ1 an algorithm was implemented where refactorings were randomly applied at each iteration. The obtained Pareto fronts were compared for statistically significant differences with NSGA-II using IHV, IGD and IC.

We do not dwell long in answering the first research question, **RQ1** that involves comparing our approach based on NSGA-II with the random search. The remaining research questions will reveal more about the performance, insight, and usefulness of our approach.

Table 5.4 confirms that NSGA-II and MOPSO are better than random search based on the three quality indicators IHV, IGD and IC on all six open source systems. The Wilcoxon rank sum test showed that in 51 runs both NSGA-II and MOPSO results were significantly better than random search. We conclude that there is empirical evidence that our multi-objective formulation surpasses the performance of random search thus our formulation is adequate (this answers RQ1).

Table 5.4. The significantly best algorithm among random search, NSGA-II and MOPSO (No sign. diff. means that NSGA-II and MOPSO are significantly better than random, but not statistically different).

Project	***IC***	***IHV***	***IGD***
Xerces-J	NSGA-II	NSGA-II	NSGA-II
JFreeChart	NSGA-II	NSGA-II	NSGA-II
GanttProject	MOPSO	No sign. diff.	MOPSO
ApacheAnt	NSGA-II	NSGA-II	NSGA-II
JHotDraw	NSGA-II	NSGA-II	NSGA-II
Rhino	No sign. diff.	NSGA-II	No sign. diff.
Log4J	NSGA-II	NSGA-II	NSGA-II

Nutch	No sign. diff.	No sign. diff.	NSGA-II
JDI-Ford	NSGA-II	NSGA-II	NSGA-II

Results for RQ2: Comparison with State of the Art Refactoring Approaches.

In this section, we compare our NSGA-II adaptation to the current, state-of-the-art refactoring approaches. To answer RQ2.1, we implemented a widely used multi-objective algorithm, namely multi-objective particle swarm optimization (MOPSO) and we compared NSGA-II and MOPSO using the same quality indicators used in RQ1. In addition, we used boxplots to analyze the distribution of the results and discover the knee point (best trade-off between the objectives). Furthermore, we compare our proposal with our previous formulation [104] based on two objectives using the *FCS*, *SCS*, *ICS*, *CR*, and *ICT*. To answer RQ2.2 we implemented a mono-objective Genetic Algorithm where one fitness function is defined as an average of the three objectives (quality, severity and importance). The multi-objective evaluation measures (*IHV*, *IGD* and *IC*) cannot be used in this comparison thus we considered the five metrics *FCS*, *SCS*, *ICS*, *CR*, and *ICT* defined in Section 4.3. To answer RQ2.3 we compared NSGA-II with two existing search-based refactoring approaches, Kessentini et al. [71] and O'Keeffe et al. [70], where uncertainties are not taken into account. We considered the same metrics used to answer RQ2.2. To answer RQ2.4, we compared our proposal to a popular design defects detection and correction tool JDeodorant that does not use heuristic search techniques. We compared the results of this tool with NSGA-II using *FCS*, *SCS*, *ICS*, *CR*, and *ICT* since only one refactoring solution can be proposed and not a set of "non-dominated" solutions. To answer the above research questions, we selected the solution from the set of non-dominated ones providing the maximum trade-off using the following strategy when comparing the different algorithms (expect the mono-objective algorithm where we select the solution with the highest fitness function or the JDeodorant tool). In order to find the maximal trade-off solution of the multi-objective or many-objective algorithm, we use the trade-off worthiness metric proposed by Rachmawati and Srinivasan [93] to evaluate the worthiness of each non-dominated solution in terms of compromise between the objectives.

Results for RQ2.1: Comparison with Multi-Objective Approaches.

To answer the second research question, RQ2.1, we compared NSGA-II to another widely used multi-objective algorithm, MOPSO, using the same adapted fitness function. Table 4 shows the overview of the results of the significance tests comparison between NSGA-II and MOPSO. NSGA-II outperforms MOPSO in most of the cases: 20 out of 27 experiments (74%). MOPSO outperforms the NSGA-II approach only in GanttProject, which is the smallest open source system considered in our experiments, having the lowest number of legal refactorings available, so it appears that MOPSO's search operators make a better task of working with a smaller search space. In particular, NSGA-II outperforms MOPSO in terms of IC values in 4 out 6 experiments with one 'no significant difference' result. Regarding IHV, NSGA-II outperformed MOPSO in 6 out of 9 experiments, where only two cases were not statistically significant, namely GanttProject and Nutch. For IGD, the results were similar as for IC.

A more qualitative evaluation is presented in Figure 5.2. Boxplots using the quality measures (a) IC, (b) IHV, and (c) IGD applied to NSGAII and MOPSO. illustrating the box plots obtained for the multi-objective metrics on the different projects. For almost all problems the distributions of the metrics values for NSGA-II have smaller variability than for MOPSO. This fact confirms the effectiveness of NSGA-II over MOPSO in finding a well-converged and well-diversified set of Pareto-optimal refactoring solutions.

Next, we use all four metrics FCS, SCS, ICS and ICT to compare four robust refactoring algorithms: our NSGA-II adaptation with the three objectives, MOPSO, our previous work based on NSGA-II with two objectives.

The results from 51 runs are depicted in Table 5.5. For FCS, the number of fixed code smells using NSGA-II is better than MOPSO in all systems except for GanttProject and also, the FCS score for NSGA-II is better than mono-EA in 100% of cases. We have the same observation for the SCS and ICS scores where NSGA-II outperforms MOPSO at least 88% of cases. Even for GanttProject, the number of fixed code smells using NSGA-II is very close to those fixed by MOPSO. The execution time of NSGA-II is invariably lower than that of MOPSO with the same number of iterations, however, the execution time required by Mono-EA is lower than both NSGA-II and MOPSO. The NSGA-II with the two objectives has a lower execution time than our adaptation with three objectives which normal. The different algorithms are using different

change operators and the number of objectives. In addition, our robust algorithm is using a perturbation function which not used by the non-robust algorithms. Thus, it is normal that the execution time is different. However, it is not a very important factor since refactoring recommendation is not a real-time problem

In conclusion, we answer RQ2.2 by concluding that the results obtained in Table 5.5 (a) confirm that both multi-objective formulations are adequate and that NSGA-II outperforms MOPSO in most of the cases.

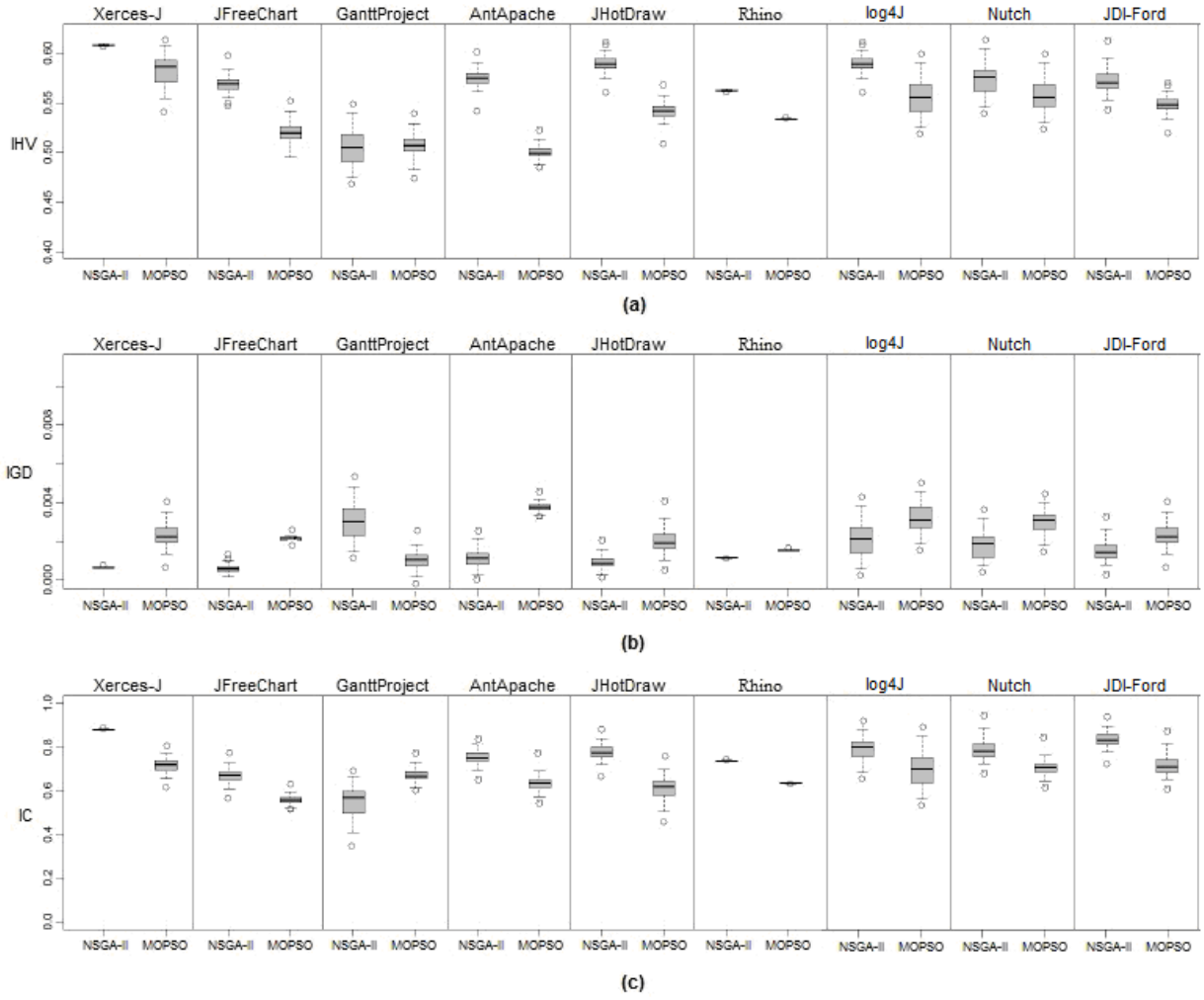


Figure 5.2. Boxplots using the quality measures (a) IC, (b) IHV, and (c) IGD applied to NSGAII and MOPSO.

We first note that the mono-EA provides only one refactoring solution while NSGA-II and MOPSO generate a set of non-dominated solutions. In order to make meaningful comparisons,

we select the best solution for NSGA-II and MOPSO using a knee point strategy as described previously. The knee point corresponds to the solution with the maximal trade-off between quality and robustness, i.e., a small improvement in either objective induces a large degradation in the other. Hence moving from the knee point in either direction is usually not interesting for the developer. Thus, for NSGA-II and MOPSO, we select the knee point from the Pareto approximation having the median IHV value. We aim by this strategy to ensure fairness when making comparisons against the mono-objective EA. For the latter, we use the best solution corresponding to the median observation on 51 runs. We use the trade-off "worth" metric proposed by Rachmawati et al. to find the knee point [93]. This metric estimates the worthiness of each non-dominated refactoring solution in terms of the trade-off between quality and robustness. After that, the knee point corresponds to the solution having the maximal trade-off "worthiness" value.

Table 5.5. FCS, SCS and ICS median values of 51 independent runs: (a) Robust Algorithms, and (b) Non-Robust algorithms.

Systems	NSGA-II (3 Obj)				MOPSO				NSGA-II (2 Obj)				Mono-EA			
	FCS	SCS	ICS	ICT	FCS	SCS	ICS	ICT	FCS	SCS	ICS	ICT	FCS	SCS	ICS	ICT
Xerces-J	74/82	42.6	49.2	1h56	69/82	34.6	46.8	1h48	61/82	29.6	39.6	1h32	54/82	21.8	32.8	1h08
JFreeChart	61/73	39.8	37.8	2h02	60/73	39.4	36.1	2h21	51/73	24.8	31.2	1h29	42/73	19.9	23.6	1h06
GanttProject	42/56	31.4	29.6	1h59	40/56	30.1	27.1	2h09	32/56	21.1	22.4	1h22	24/56	18.7	17.6	1h01
ApacheAnt	83/91	44.6	46.4	2h18	84/91	44.9	46.4	2h22	69/91	32.6	38.6	1h42	53/91	26.4	31.2	1h11
JHotDraw	28/33	14.3	18.7	1h49	25/33	12.2	16.2	1h46	16/33	11.1	10.3	1h34	12/33	06.3	07.1	1h04
Rhino	62/74	42.8	32.3	1h56	63/74	43.2	33.1	1h59	58/74	33.1	28.4	1h39	41/74	21.4	19.5	1h12
Log4J	53/64	39.5	29.4	1h38	50/64	35.9	27.3	1h52	41/64	28.9	23.1	1h14	32/64	19.4	18.9	1h01
Nutch	66/72	41.1	36.8	1h49	61/72	39.2	31.6	1h58	49/72	30.2	21.8	1h19	43/72	24.8	19.1	1h02
JDI-Ford	74/88	43.8	42.1	1h56	72/88	42.9	40.2	2h12	64/88	37.3	28.4	1h24	61/88	32.6	21.1	1h04

Systems	Kessentini et al.'11				O'Keeffe et al.'08				JDedorant			
	FCS	SCS	ICS	ICT	FCS	SCS	ICS	ICT	FCS	SCS	ICS	ICT
Xerces-J	52/82	20.2	32.8	1h10	50/82	19.4	30.2	1h02	46/82	19.1	29.1	N/A
JFreeChart	43/73	18.2	23.6	1h04	38/73	17.9	21.6	1h00	37/73	17.2	20.4	N/A
GanttProject	20/56	13.7	17.6	1h06	21/56	16.9	15.8	0h56	22/56	15.8	14.2	N/A
ApacheAnt	51/91	20.4	31.2	1h14	48/91	22.3	27.1	0h54	41/91	21.2	27.1	N/A
JHotDraw	13/33	05.3	07.1	1h24	12/33	06.2	07.2	0h51	10/33	05.9	06.3	N/A
Rhino	40/74	18.6	19.5	1h16	38/74	17.2	17.4	1h01	34/74	19.1	17.2	N/A

Log4J	32/64	18.4	17.6	1h10	31/64	15.9	17.8	1h04	29/64	16.5	15.1	N/A
Nutch	43/72	21.2	20.1	1h02	40/72	18.4	19.6	0h48	31/72	17.1	18.2	N/A
JDI-Ford	61/88	29.4	20.4	1h01	56/88	27.1	18.2	0h52	43/88	25.9	17.6	N/A

Results for RQ2.2, RQ2.3 and RQ2.4: Comparison with Mono-Objective and non-Search-Based Approaches.

Table 5.5 also shows the results of comparing our robust approach based on NSGA-II with two mono-objective refactoring approaches, a mono-objective genetic algorithm (Mono-EA) that has a single fitness function aggregating the two objectives, and a practical refactoring technique where developers used a refactoring plug-in in Eclipse to suggest solutions to fix code smells. It is apparent from Table 5.5 that our NSGA-II adaptation outperforms mono-objective approaches in terms of smell-fixing ability (FCS) all the cases. In addition, our NSGA-II adaptation outperforms all the mono-objective and manual approaches in 100% of experiments in terms of the two robustness metrics, SCS and ICS. This is can be explained by the fact that NSGA-II aims to find a compromise between the three objectives however the remaining approaches did not consider robustness but only quality. Thus, NSGA-II sacrifices a small amount of quality in order to improve importance and severity. Furthermore, the number of code smells fixed by NSGA-II is very close to the number fixed by the mono-objective and manual approaches, so the sacrifice in solution quality is quite small. When comparing NSGA-II with the remaining approaches we considered the best solution selected from the Pareto-optimal front using the knee point-based strategy described above. Another interesting observation is that our refactoring solutions prioritized fixing code fragments containing severe code smells and also located in important classes. In fact, as described in Table 5.5 our approach fixed more important and severe code smells than all other existing approaches based on the SCS and ICS metrics. It is also well-known that a mono-objective algorithm requires lower execution time for convergence since only one objective is handled.

To answer RQ2.3 and RQ2.4, the results of Table 5.5(b) support the claim that our NSGA-II formulation provides a good trade-off between importance, severity and quality, and outperforms on average the state of the art of refactoring approaches, both search-based and manual, with a low robustness cost.

Results for RQ3: Manual Evaluation of the Results by Developers.

To answer the last question RQ3 a manual evaluation were performed by subjects to estimate the correctness of the suggested refactoring. Figure 5.3 depicts the different Pareto surface obtained on the JDI-Ford system using NSGA-II to optimize the three objectives of quality, severity and importance. Due to space limitations, we show only this example of the Pareto-optimal front approximation. Similar fronts were obtained on the remaining systems. The 3D projection of the Pareto front helps developers to select the best trade-off solution between the three objectives based on their own preferences. Based on the plot of Figure 5.3, the developer could degrade quality in favor of importance and severity while controlling visually the robustness cost, which corresponds to the ratio of the quality loss to the achieved importance and severity gain. In this way, the preferred robust refactoring solution can be realized.

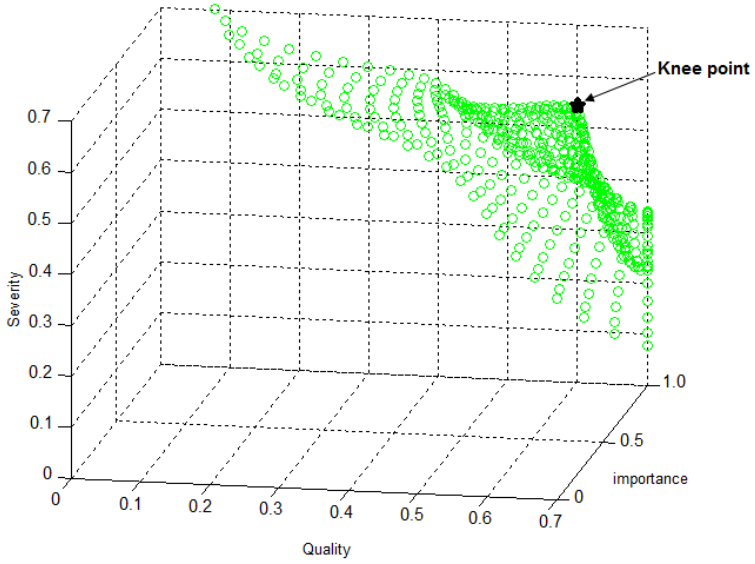


Figure 5.3. 3D projection of the Pareto-Front solutions on the JDI-Ford system.

One striking feature is that starting from the highest quality solution the trade-off between the three objectives is in favor of quality, meaning that the quality degrades slowly with a fast increase in importance and severity up to the knee point, marked in each figure. Thereafter, there is a sharp drop in quality with only a small increase in importance. It is very interesting to note that this property of the Pareto-optimal front is apparent in all the problems considered in this study. Without any robustness consideration in the search process, one would obtain the highest quality solution all the time (which is not robust at all), but Figure 5.3 shows how a better robust solution (importance and severity) can be obtained by sacrificing just a little in quality.

Figure 5.3 shows the impact of different levels of perturbation on the Pareto-optimal front. However, it is difficult to generalize the observation that *the sacrifice in solution quality is quite small*. The developers may select the solution based on their preferences and the current context. In case that the developers do not have enough time to fix all or most of the defects, they may select the refactoring solution fixing the most severe ones or those located in important classes. In other situations where there is enough time before the next release and several developers are available, a solution that minimizes the sacrifice in quality is more adequate. The slight sacrifice on quality was only observed on few systems, thus, it is hard to generalize the results.

Our approach takes as input as the maximum level of perturbation applied in the smell severity and class importance at each iteration during the optimization process. A high level of perturbation generates more robust refactoring solutions than those generated with lower variations, but the solution quality, in this case, will be higher. As described by Figure 5.3, the developer can choose the level of perturbation based on his/her preferences to prioritize quality or robustness. Although the Pareto-optimal front changes depending on the perturbation level, there still exists a knee point, which makes the decision making by a developer easier in such problems.

Figure 5.4 describes the manual qualitative evaluation of some suggested refactoring solutions. It is clear that results are almost similar to our proposal and existing approach in terms of the semantic coherence of suggested refactorings. We consider that a semantic precision more than 70% acceptable since most of the solutions should be executed manually by developers and our tool is a recommendation system. Thus, developers can evaluate if it is interesting or not in applying some refactorings based on their preferences and the semantic coherence.

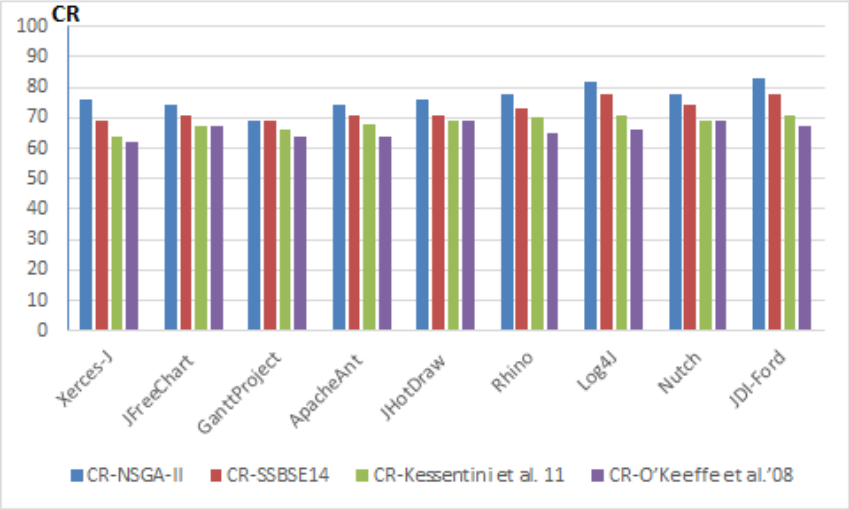


Figure 5.4. The median correctness values (CR) of the recommended refactorings based on 51 runs. The obtained results are statistically analyzed by using the Wilcoxon rank sum test with a 95% confidence level (α = 5%).

To answer RQ3 more adequately, we considered two real-world scenarios to justify the importance of taking into consideration robustness when suggestion refactoring solutions. In the first scenario, we modified the degree of severity of the four types of code smells over time and we evaluated the impact of this variation on the robustness of our refactoring solution in terms of smell severity (SCS). This scenario is motivated by the fact that there is no general consensus on the severity score of detected code smells thus software engineers can have divergent opinions about the severity of detected code smells.

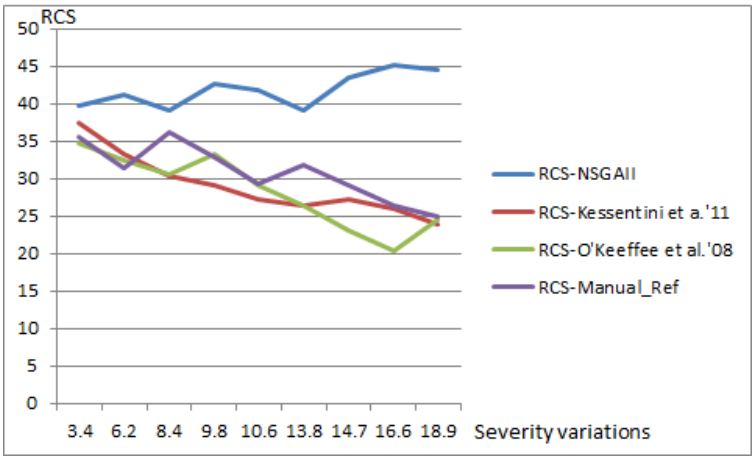


Figure 5.5. The impact of code smells severity variations on the robustness of refactoring solutions for ApacheAnt. The obtained results are statistically analyzed by using the Wilcoxon rank sum test with a 95% confidence level (α = 5%).

Figure 5.5 shows that our NSGA-II approach generates robust refactoring solutions on the Ant Apache system in comparison to existing state of the art refactoring approaches. In fact, the more the variation in severity increases over time the more the refactoring solutions provided by existing approaches become non-robust. Thus, our multi-objective approach enables the most severe code smells to be corrected even with slight modifications in the severity scores. The second scenario involved applying randomly a set of commits, collected from the history of changes of the open source systems [86], and evaluating the impact of these changes on the robustness of suggested refactoring proposed by our NSGA-II algorithm and non-robust approaches.

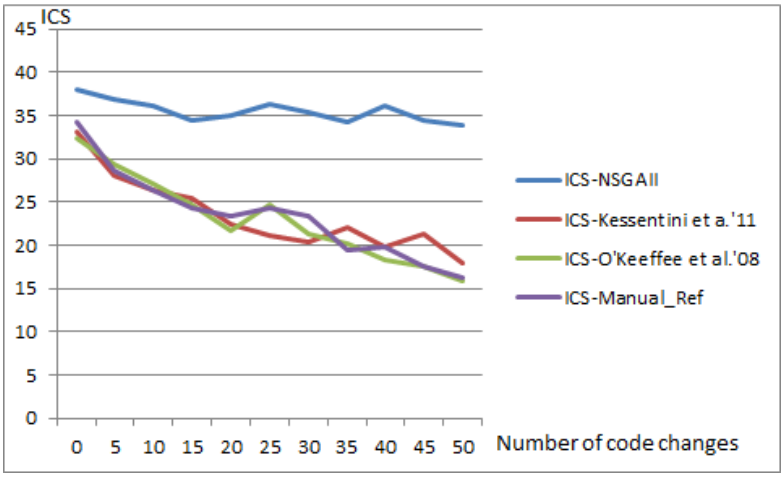


Figure 5.6. The impact of class importance variation on the robustness of refactoring solutions for Apache Ant. The obtained results are statistically analyzed by using the Wilcoxon rank sum test with a 95% confidence level (α = 5%).

As depicted in Figure 5.6, the application of new commits modifies the importance of classes in the system containing code smells and the refactoring solutions proposed by mono-objective and manual approaches become ineffective. However, in all the scenarios it is clear that our refactoring solutions are still robust and fixing code smells in most of the important classes in the system even with a high number of new commits (more than 40 commits). We also compared the results achieved by the different techniques for different values of severity and class importance using the Wilcoxon test. The obtained results are statistically analyzed by using the Wilcoxon rank sum test with a 95% confidence level (α = 5%). We also compared the refactoring solution at the knee-point (robust) for ApacheAnt with the best refactoring solution that maximizes only the quality (non-robust) to understand why the former solution is robust in both scenarios. We

found that the knee-point solution rectified some code smells that were not very risky and not located in important classes but these code-smells become more important after new commits. Thus, we can conclude that the simulation of changes in both importance and severity helps our NSGA-II to predict some future changes and adapt the best solutions according to that. Hence, we conclude that RQ3 is affirmed and that the robust multi-objective approach has value for software engineers in a real-world setting.

5.1.4 Conclusion

We have introduced a novel formulation of the refactoring problem that takes into account the uncertainties related to code smell correction in the dynamic environment of software development where code smell severity and class importance cannot be regarded as fixed. Code smell severity will vary from developer to developer and the importance of the class that contains the smell will vary as the code base itself evolves. We have reported the results of an empirical study of our robust technique compared to different existing approaches, and the results obtained have provided evidence to support the claim that our proposal enables the generation of robust refactoring solutions without a high loss of quality based on a benchmark of six large open source systems.

Our consideration of robustness as a separate objective has revealed an interesting feature of the refactoring problem in general. In our experiments, the trade-off between quality and robustness resulted in a knee solution in every case. From the highest quality solution to the knee point, the trade-off is in favor of quality, while after the knee point quality degrades more quickly than robustness. Based on this observation, we can recommend the knee solution to the software engineer as the most likely quality-robustness trade-off solution to consider.

Future work involves extending our approach to handling additional code smell types in order to test further the general applicability of our methodology. We focused on the use of a structural metric to estimate class importance, but this can be extended to consider also the pattern of repository submits to achieve another perspective on class importance. In a similar vein, our notion of smell severity assumes each smell type has a certain severity, but a more realistic model is to allow each individual smell instance to be assigned its own severity. If

further experiments confirm our observation that the knee point is indeed a trademark of the quality-robustness trade-off frontier for all software refactoring problems, then it would be interesting to apply straightway a knee-finding algorithm [122] to the bi-objective problem and determine if it yields any computational benefit. In an interactive software refactoring tool, the potential speed-up might be critical to success. Overall the use of robustness as a helper objective in the software refactoring task opens up a new direction of research and application with the possibility of finding new and interesting insights about the quality and severity trade-off in the refactoring problem.

5.2 Many-objective Software Refactoring

5.2.1 Introduction

Search-based software engineering (SBSE) studies the application of meta-heuristic optimization techniques to software engineering problems. Once a software engineering task is framed as a search problem, by defining it in terms of solution representation, objective function, and solution change operators, there is a multitude of search algorithms that can be applied to solve that problem. Search-based techniques are widely applied to solve software engineering problems such as in testing, modularization, refactoring, planning, etc. [84].

Based on a recent SBSE survey [5], the majority of existing work treats software engineering (SE) problems from a single objective point of view, where the main goal is to maximize or minimize one objective, e.g., correctness, quality, etc. However, most SE problems are naturally complex in which many conflicting objectives need to be optimized such as model transformation, design quality improvement, test suite generation etc. The number of objectives to consider for most of software engineering problems is, in general, high (more than three objectives); such problems are termed *many-objective*. We claim that the reason that software engineering problems have not been formulated as many-objective problems is because of the challenges in constructing a many-objective solution. In this context, the use of traditional multi-objective techniques, e.g., NSGA-II, widely used in SBSE, is clearly not sufficient [83].

There is a growing need for scalable search-based software engineering approaches that address software engineering problems where a large number of objectives are to be optimized. Recent work in optimization has proposed several solution approaches to tackle many-objective optimization problems [82] [123] using e.g., objective reduction, new preference ordering relations, decomposition, etc. However, these techniques have not yet been widely explored in SBSE [5]. To the best of our knowledge and based on recent SBSE surveys [5], very few studies used many-objective techniques to address software engineering problems such as the work proposed by Abdel-Salam et al. [36] that uses a many-objective approach, IBEA (Indicator-Based Evolutionary Algorithm) [124], to address the problem of software product line creation. However, the number of considered objectives is limited to five. In our recent work [125], we proposed a many-objective approach for software refactoring using a set of 15 quality metrics as separate objectives. Software refactoring is one of those software engineering problems that require several quality objectives to be satisfied. Although, the approach has given promising results, some limitations have been raised, investigated and resulted in this extension that copes with the following concerns: the choice of using 15 metrics as separate objectives, at the expense of higher computational complexity and higher number of solutions, made the benefit of selecting good refactoring solutions unclear due to conflict uncertainty between some objectives and the difficult interpretation of their metric values. Another limitation of our previous work is the exclusive optimization of the system's structure without taking into consideration the semantic coherence of its actors (classes, methods, attributes). Furthermore, One of the main objectives of Pareto-optimality is allowing the user to choose among equivalent solutions the one(s) that satisfies better his/her preferred objectives, in our context, it is harder to ask a developer to express his preference in terms of 15 internal quality attributes, so it is difficult for developers to select a solution from the high number of non-dominated refactoring solutions.

To address the above challenges, we propose to extend our previous work using a different many-objective formulation. Whereas, we assume that it would be more convenient for developers to formulate their quality preferences in terms of external quality attributes such as reusability, flexibility and understandability instead of a large number of quality metrics [70]. Thus, the goal of improving the software overall quality is still maintained while the number of objectives has been reduced. This is being done through the aggregation of metrics, previously

optimized separately, into 8 objectives described as external quality attributes. This representation helps in analyzing the impact of applying refactoring operations on raising the conflicts between these objectives during the solutions evolution. Thus, the purpose of this work is to formulate the refactoring problem using the quality attributes of QMOOD [126] as objectives along with the number of refactorings and the design coherence preservation. To this end, we adapted the many-objective algorithm NSGA-III [20]. NSGA-III is a many-objective algorithm proposed by Deb and Jain. The basic framework remains similar to the original NSGA-II algorithm [21], with significant changes in its selection mechanism. We implemented our approach and evaluated it on seven large open source systems and one industrial project provided by our industrial partner. We compared our findings to several other many-objective techniques, an existing multi-objective approach [86], a mono-objective technique [71] and an existing refactoring technique not based on heuristic search [106]. Statistical analysis of our experiments over 31 runs shows the efficiency of our approach.

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

(1) Many-objective formulation of the refactoring problem through several objectives using NSGA-III.

(2) The reported results provide evidence to support the claim that our proposal is more efficient, on average, than several of existing refactoring techniques based on a benchmark of seven open source systems and one industrial project. As part of the validation, an evaluation of the relevance and usefulness of the suggested refactoring is done for software engineers to improve the quality of their systems.

5.2.2 Approach

The refactoring problem involves searching for the best refactoring solution among the set of candidate ones, which constitutes a huge search space. A refactoring solution is a sequence of refactoring operations where the goal of applying the sequence to a software system *S* is typically to fix maintenance issues in S. Usually in SBSE approaches, we use two or three metrics as objective functions for a particular multi-objective heuristic algorithm to find these design issues and correct them. In reality, we assume that increasing the number of metrics to

optimize may increase the quality of the refactored code. However, the high number of suggested solutions in a 16-objectives Pareto-Front quickly exceeds the developer's ability to manually choose between them. It has been known that developers most likely want a unique optimal solution that better satisfies his/her preferences that can be easily expressed in terms of quality objectives. Motivated by this observation, we propose in this research work to consider the six objectives of the QMOOD model where each represents a separate objective function along with two other objectives to reduce the number of refactorings to apply and maximize the design coherence after refactoring. In this way, we obtain a many-objective (8-objective) formulation of the refactoring problem that could not be solved using standard multi-objective approaches. This formulation is given as follows:

$$Maximize \quad F(x,S) = [\, f_1(x,S),\, f_2(x,S),...,\, f_8(x,S)]$$
$$subject\ \ to \quad x = (x_1,...,x_n) \in X$$

where X is the set of all legal refactoring sequences starting from S, xi is the i^{th} refactoring operation, and $f_k(x,S)$ is the k^{th} objective.

The concern about using these operations is whether each one of them have a positive impact on the refactored code quality. In this context, previous work has studied the impact analysis of refactoring operations on internal and external quality metrics. Du Bois et al. [127] proposed the evaluation of a selected set of refactorings based on their impact on the internal CK quality metrics. They extended their work by including more studied operations while enhancing cohesion and coupling measures [62]. They provided guidelines to distinguish between operations that optimize software quality and ruled out those which their application will increase coupling or decrease cohesion. Similarly, Alshayeb [128] has quantitatively assessed, using internal metrics, the effect of refactorings on different quality attributes to help developers in the estimation of refactoring effort in terms of the cost and time.

In the following, we will describe in details the different objectives considered in our formulation.

5.2.2.1 QMOOD model quality attributes as objectives

Many studies have been utilizing structural metrics as a basis of defining quality indicators for a good system design [129] [130] [131]. As an illustrative example, Bansiya et al. [126] proposed a set of quality measures, using the ISO 9126 specification, called QMOOD. Each of these quality metrics is defined using a combination of high-level metrics detailed in Table 5.6.

Table 5.6. QMOOD metrics description.

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

These above-mentioned metrics were eventually used to define the quality attributes that are enumerated in the following Table 5.7. The adaptation of the QMOOD model in the problem formularization has provided six quality attributes as separate objectives: *reusability*, *flexibility*, *understandability*, *functionality*, *extendibility* and *effectiveness*.

Table 5.7. Quality attributes and their computation equations.

Quality attribute	**Definition** **Computation**
Reusability	A design with low coupling and high cohesion is easily reused by other designs. -0.25*Coupling+0.25*Cohesion+0.5*Messaging+0.5*Design Size
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*Design Size
Functionality	Classes with given functions that are publically stated in interfaces to be used by others. 0.12*Cohesion+0.22*Polymorphism+0.22*Messaging+0.22*DesignSize+0.22*Hierarchies
Extendibility	Measurement of design's allowance to incorporate new functional requirements. 0.5*Abstraction-0.5*Coupling+0.5*Inheritance+0.5*Polymorphism
Effectiveness	Design efficiency in fulfilling the required functionality. 0.2*Abstarction+0.2*Encapsulation+0.2*Composition+0.2*Inheritance+0.2*Polymorphism

5.2.2.2 Number of code changes as an objective

It is known that multiple refactoring sequences may have a completely different set of operations which their execution will give two different resulting designs but they might have the same quality. So, the execution of a specifically suggested refactoring sequence may require an effort that is comparable to the one of re-implementing part of the system from scratch. Taking this observation into account, it is trivial to minimize the number of suggested operations in the refactoring solution since the designer can have some preferences regarding the percentage of deviation with the initial program design. In addition, most developers prefer solutions that minimize the number of changes applied to their design [71]. Thus, we formally defined the fitness function as the number of refactoring operations (size of the solution) to be minimized:

$f_7(x) = Size(x)$ where x is the solution to evaluate.

The different code changes (refactoring types) used in our approach may have different impacts on the maintainability/quality objectives considered in our formulation. We show in the following that the different quality objectives are conflicting since the different refactoring types

considered in our approach may decrease some quality attributes and increase some others. In Shatnawi et al. [132], the impact analysis of some of Fowler's catalog operations on some external quality factors (effectiveness, flexibility, extendibility and reusability) has shown that not all refactorings necessarily improve these quality factors. Rules of thumb have been established using heuristics to dictate which refactorings to use in order to enhance a given quality attribute. The following table has been extracted from [132] to show the impact analysis of the refactorings applied on restructuring EclipseIU 2.1.3 and Struts (1.1 and 1.2.4).

Table 5.8. Refactorings impact analysis on QMOOD internal metrics.

Refactoring Operation	DSC	NOH	ANA	DAM	DCC	CAMC	MOA	MFA	NOP	CIS	NOM
Extract class	+	0	0	0	+	+	+	0	0	0	0
Extract interface	+	0	+	0	0	0	0	+	+	0	0
Inline class	-	0	0	0	-	-	-	0	0	0	0
Move field	0	0	0	0	0	+	0	0	0	0	0
Move method	0	0	0	0	-	+	0	0	0	-	0
Push down field	0	0	0	0	0	+	0	0	0	0	0
Push down method	0	0	0	0	+	0	0	+	+	+	0
Pull up field	0	0	0	0	0	-	0	0	0	0	0
Pull up method	0	0	0	0	-	0	0	-	-	-	0

Based on Table 5.8, the operations have various implications on the internal metrics that can reach the degree of conflict. The composition of these metrics values can be an indicator of how the external quality attributes will be affected. Table 5.9 shows the resulting impact analysis on QMOOD quality attributes and the potential conflict between them.

Table 5.9. Refactorings impact analysis on QMOOD quality attributes.

Refactoring Operation	Reusability	Flexibility	Understandability	Functionality	Extendibility	Effectiveness
Extract class	+	+	-	+	-	+
Extract interface	+	+	-	+	+	+
Inline class	-	-	+	-	+	-
Move field	+	0	+	+	0	0
Move method	0	+	+	-	+	0
Push down field	+	0	+	+	0	0
Push down method	+	+	-	+	+	+
Pull up field	-	0	-	-	0	0
Pull up method	-	-	+	-	-	-

Although the statistical significance of the reported results in Table 5.9 was not studied in this work and kept as part of our future investigations, it does not affect our problem formulation, in fact, if the degree of conflict between two objectives is not considerable for a random set of refactorings, this degrades the heuristics' performance by increasing the computational time of the heuristics but it does not affect the quality of the results.

In general, the related work has given the following observations: (1) Not all refactoring operations have a desired impact on internal and external quality attributes. (2) It is difficult to theoretically come up with an optimal set of corrections to increase a chosen quality attribute without decreasing another. (3) A goal-oriented process has been given to include or/and exclude refactorings based on developer's preferred quality attribute.

These limitations motivated our formulation that (1) is not limited to specific types of refactorings and that (2) is not limited to optimizing a preferred quality attribute with disregard to the others.

5.2.2.3 Design coherence preservation as an objective

It is usually the designer's responsibility to manually inspect the feasibility of the suggested refactorings and evaluate their meaningfulness from the design coherence perspective. Sometimes, the new refactored design may be structurally improved but introduce several design incoherences. To preserve the semantics design, we present two main formulated different measures in which we describe the following sections. The fitness function is formulated as an average of the two following measures.

A. Vocabulary-based similarity (VS)

This kind of similarity should be eventually considered when moving methods, or attributes between classes. For instance, when moving a method or an attribute from one source class to another destination class, this operation would make sense if both source and target classes have similar vocabularies. In this case, it is assumed that the vocabulary of naming the code elements in classes is reflecting a specific domain terminology. That's why two code elements could be semantically similar if they use similar vocabularies [52].

A token-based extraction [133] of vocabulary is performed on the naming of classes, methods, attributes, parameters and comments. This Tokenization process is widely used in code clones detection techniques and it is known to be more robust to code changes compared to text-based approaches. The semantic similarity is calculated based on information retrieval-based techniques (e.g., cosine similarity). The following equation calculates the cosine similarity between two classes. 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 classes, c1 and c2, is determined as follows:

$$Sim(c1,c2) = \cos(\vec{c1},\vec{c2}) = \frac{\vec{c1}.\vec{c2}}{\|\vec{c1}\| * \|\vec{c2}\|} = \frac{\sum_{i=1}^{n}(w_{i,1} * w_{i,2})}{\sqrt{\sum_{i=1}^{n}(w_{i,1})^2}\sqrt{\sum_{i=1}^{n}(w_{i,2})^2}} \in [0,1] \quad (5.5)$$

where $\vec{c1} = (w_{1,1},..., w_{n,1})$ and $\vec{c2} = (w_{1,2},..., w_{n,2})$ are respectively two vectors corresponding to c1 and c2 The weights $w_{i,j}$ are automatically generated by information retrieval-based techniques such as the Term Frequency – Inverse Term Frequency (TF-IDF) method. We used a method similar to that described in [133] to determine the vocabulary and represent the classes as term vectors.

B. Dependency-based similarity (DS)

Similarly to vocabulary similarity, the semantic closeness can be also extracted from mutual dependencies. In general, a high coupling (i.e., multiple call in and call out) between two classes is usually not recommended, and if it exists, developers are usually prompted to merge them to reduce the design complexity, this also hints that these two classes are semantically close. Intuitively, the application of refactoring on highly dependent classes is not only beneficial to the design quality, but also has a higher probability to eventually be meaningful. To follow up with dependency, we point out two types of dependency links:

1) *Shared Method Calls (SMC)* that can be easily detected through the application call graphs using the Class Hierarchy Analysis (CHA) [134]. Methods are modeled as graphs while calls

represent the edges between nodes. A call graph can either be a call in or call out. This technique is applied for each couple of classes, shared calls are being detected through the graph by identifying shared neighbors of nodes related to each actor. Shared call-in and shared call-out are distinguished and separately calculated for a given couple c1 and c2 (i.e., two classes) using the following equations.

$$\text{sharedCall Out}(c1, c2) = \frac{|\text{callOut}(c1) \cap \text{callOut}(c2)|}{|\text{callOut}(c1) \cup \text{callOut}(c2)|} \in [0,1] \quad (5.6)$$

$$\text{sharedCall In}(c1, c2) = \frac{|\text{callIn}(c1) \cap \text{callIn}(c2)|}{|\text{callIn}(c1) \cup \text{callIn}(c2)|} \in [0,1] \quad (5.7)$$

2) *Shared field access (SFA)* is also known as data coupling and occurs when a class refers to another as a type or shares a method that references another class as a parameter type. Static analysis is adopted to view occurrences of possible invocation of calls of field accesses through methods or constructors. Two classes have a high shared field access rate if they read or modify the same fields belonging to one or both of them. This violation of the principle of modularity can be, for example, fixed by either merging these two classes. In this context, the rate of shared field access is used as an indicator of a semantic closeness between two classes *c1* and *c2*, and it is calculated according to the following equation.

$$\text{sharedFieldsRW}(c1, c2) = \frac{|\text{fieldRW}(c1) \cap \text{fieldRW}(c2)|}{|\text{fieldRW}(c1) \cup \text{fieldRW}(c2)|} \in [0,1] \quad (5.8)$$

where fieldRW(ci) refers to the number of fields that may be read or write by each method of the module *ci*.

To illustrate the dependency similarity measure, let us take the example of two classes A and B with no direct calls between them, if a third class *C* calls both of them, then the *callIn(A)* inter *callIn(B)* will be incremented, the intersection between *callIns* determines the number of classes that both call these two classes. It is divided by the overall number of *callIns* received by these two classes. Similarly for the *callsOut* which informs about the number of common entities called by two given classes. For the shared field access, the idea is similar, even if sharing attributes is a bad practice indeed, if it exists, then it creates a dependency between the class

sharing the attribute and the other classes accessing it, we use this as an indicator of semantic closeness between them.

5.2.2.4 Solution representation

As defined in the previous section, a solution consists of a sequence of *n* refactoring operations applied to different code elements in the source code to fix. In order to represent a candidate solution (individual/chromosome), we use a vector-based representation. Each vector's dimension represents a refactoring operation where the order of applying these refactoring operations corresponds to their positions in the vector. For each of these refactoring operations, we specify pre- and post-conditions in the style of Fowler [15] to ensure the feasibility of their application. The initial population is generated by assigning randomly a sequence of refactorings to some code fragments.

5.2.2.5 Solution variation

In each search algorithm, the variation operators play the key role of moving within the search space with the aim of driving the search towards optimal solutions. For 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. This operator must ensure the respect of the length limits (size of the solution is limited to up-to 500 refactorings in our experiments) by eliminating randomly some refactoring operations. It is important to note that in many-objective optimization, it is better to create children that are close to their parents in order have a more efficient search process [20]. For this reason, we control the cutting point of the one-point crossover operator by restricting its position to be either belonging to the first tier of the refactoring sequence or belonging to the last tier. For mutation, we use the bit-string mutation operator that picks probabilistically one or more refactoring operations from its or their associated sequence and replaces them by other ones from the initial list of possible refactorings.

5.2.2.6 Solution evaluation

Each generated refactoring solution is executed on the system *S*. Once all required data is computed, the solution is evaluated based on the quality of the resulting design (the six quality attribute objectives to be maximized), along with the aggregation of semantics similarity functions (to be maximized) followed by the complexity of the refactoring operations (to be minimized). These values are the coordinates of the solution in the objectives space and so, it is assigned a non-domination rank as well as a particular reference point which is the closest to the solution.

5.2.2.7 Normalization of population members

Duo to the heterogeneous nature of objective functions (i.e., they have different ranges of values), we used the normalization procedure proposed by Deb et al. [20] to circumvent this problem. At each generation, the minimal and maximal values for each metric are recorded and then used by the normalization procedure to calculate respectively the Nadir and ideal points [135]. Normalization allows the population members and the reference points to have the same range, which is a pre-requisite for diversity preservation.

5.2.2.8 Final solution selection

Once the final Pareto front has been generated, for each fitness function, the reference sets are calculated from the union of all Pareto front approximations which are being normalized with respect to ideal and Nadir point. The choice of one individual among the large set of Pareto-optimal solutions is not trivial. The preference of the developer can be then used to determine which of the solution may better satisfy his/her needs since the objectives have been defined in terms of quality attributes along with minimizing the size and maximizing the semantic coherence, it is easier for the developer to specify a ranking that can be used as input to the hyperplane and consider only reference points that match (or closest to) the rank. This will reduce drastically the number of preferred reference points. Still, if few solutions satisfy such condition, the niche point selection will sort them and send the top of the queue as output. In the case of absence of developer's input, we choose the nearest solution to the Knee point [136] (i.e.,

the vector composed of the best objective values among the population members in all iterations).

5.2.3 Validation

The goal of the study is to evaluate the usefulness our many-objective refactoring tool in practice. We conducted experiments on popular open source systems and one industrial project using the Goal, Question, Metrics (GQM) assessment approach [137]. This Section is organized as follows. Section 5.2.3.1 poses three research questions that drive our experiments whose settings have been detailed in Section 5.2.3.2. Finally, Section 0 is dedicated to subjects and scenarios that constitute the non-subjective evaluation conducted with potential developers who can use our tool.

5.2.3.1 Research questions

In our study, we assess the performance of our refactoring approach by finding out whether it could generate meaningful sequences of operations that improve the quality of the systems while reducing the number of code changes and preserving the semantic coherence of the design. Our study aims at addressing the following research questions outlined below. We also explain how our experiments are designed to address these questions. The main question to answer is to what extent the proposed approach can propose meaningful refactoring solutions that can help developers to improve the quality of their systems. To find an answer, we defined the following three research questions:

RQ1: To what extent can the proposed approach improve the quality of the evaluated systems?

RQ2: How does the proposed many-objective approach based on NSGA-III perform compared to other many/multi-objective algorithms or a mono-objective approach for software refactoring and to an existing approach that is not based on heuristic search?

RQ3: How our many-objective refactoring approach can be useful for software engineers in real-world setting?

One of the challenges in SBSE is to find the most suitable search algorithm for a specific software engineering problem. The only proof is the experimental results, thus, it is important to address this question when designing a new software engineering problem as an optimization problem. In addition, a comparison with a mono-objective algorithm may justify the need to use a many-objective approach to show that the different objectives are really conflicting and cannot be merged into one fitness function. It is maybe also not sufficient to show that the proposed many-objective formulation outperforms others search algorithms, thus it is important to compare with a non-search-based approach to confirm the practical value of the proposed search-based approach.

To answer **RQ1**, we validate the proposed refactoring technique on seven open-source systems and one industrial project to evaluate the quality improvements of systems after applying the suggested refactoring solution. We calculate the overall *Quality Gain* (QG) for the six quality attributes as follows: Let Q={q1, q2,... q6} and Q'={q'1, q'2,... q'6} be respectively the set of quality attribute values before and after applying the suggested refactorings, and {w1, w2,... w6} the weights assigned to each of these quality factors. Then the QG is calculated by:

$$QG = \sum_{i=1}^{6} w_i * (q'_i - q_i) \qquad (5.9)$$

In addition, we validate the proposed refactoring operations to fix code smells by calculating the *Defect Correction Ratio* (DCR) which is given by the following equation and corresponds to the ratio of the number of corrected design defects to the initial number of detected defects before applying the suggested refactoring solution. The code smells were collected using existing detection tools DÉCOR [55] and InCode [138].

$$DCR = \frac{|\text{Corrected Defects Instances}|}{|\text{All Defects Instances}|} \in [0,1] \qquad (5.10)$$

Since it is important to validate the proposed refactoring solutions from both quantitative and qualitative perspectives, we use two different validation methods: manual validation and automatic validation of the efficiency of the proposed solutions. For the manual validation, we asked groups of potential users (software engineers) of our refactoring tool to evaluate,

manually, whether the suggested operations are feasible, make sense semantically. We define the metric *Manual Precision* (MP) which corresponds to the number of semantically coherent operations over the total number of suggested operations. MP is given by the following equation.

$$MP = \frac{|\text{Coherent Operations}|}{|\text{Suggested Operations}|} \in [0,1] \tag{5.11}$$

For the automatic validation, we compared the proposed refactorings with the expected ones using the different systems 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 [89], 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 of a system (see Table 5.12):

$$\text{RE}_{recall} = \frac{|\text{suggested operations}| \cap |\text{expected operations}|}{|\text{expected operations}|} \in [0,1] \tag{5.12}$$

$$\text{PR}_{precesion} = \frac{|\text{suggested operations}| \cap |\text{expected operations}|}{|\text{suggested operations}|} \in [0,1] \tag{5.13}$$

To answer **RQ2**, we compared the performance of NSGA-III based approach with four many-objective techniques, Gr-EA [139], DBEA-Eps [91], IBEA [124] and MOEA/D [123], an existing work based on a multi-objective NSGA-II algorithm [86] and also a mono-objective evolutionary algorithm [71]. The approaches are briefly introduced as follows:

Grid-based Evolutionary Algorithm (Gr-EA) partitions the search space into grids (also called hypercubes). The number of divisions in Gr-EA is a parameter. Then, it recombines them based on the current objective values in the population. Just like in dominance-based algorithms, it also ranks the population by fronts but the crowding distance and the spread of solutions are calculated from grid-based metrics.

Decomposition Based Evolutionary Algorithm with Epsilon Sampling (DBEA-Eps) is another decomposition-based EA with a variation in the decomposition method which generates reference points via systematic sampling and deals with a constraint by an adaptive epsilon scheme to manage the balance between convergence and diversity.

Indicator-Based Evolutionary Algorithm (IBEA) is distinguished among other EAs by its continuous dominance criteria where each solution is assigned a weight that is calculated from quality indicators, usually given by the user.

Multi-objective Evolutionary Algorithm Based on Decomposition (MOEA/D) simultaneously performs an optimization of previously decomposed sub-problems, according to their neighborhood information. MOEA/D assigns a weight vector to every individual in the population, each one being focused on the resolution of the sub-problem represented by its weight vector. The solutions evaluation is done by the Tchebycheff approach and by computing their distance to a reference point.

The comparison between these many-objective algorithms is performed in terms of convergence of the Pareto Front and with respect to the diversity of the obtained solutions. In order to estimate the convergence and diversity, we used the Inverted Generational Distance (IGD), which is the sum of distances from each point of the true Pareto front to the nearest point of the non-dominated set found by the algorithm in all iterations, the lower the IGD value, the better the approximation is. Since both indicators measure the convergence and spread of the obtained set of solutions, we will only use the IGD as a performance indicator and it will be statistically analyzed to assess the significance of results.

As part of our experiments, to demonstrate the importance of taking each quality attribute as separate objective instead of simply aggregating them into a single fitness function, a comparison with a mono-objective approach that aggregates several quality attributes in one objective is required. The comparison between many-objective algorithms with a mono-objective one is not straightforward. The first one returns a set of non-dominated solutions while the second one returns a single optimal solution. In order to cope with this situation, for each many-objective algorithm, the nearest solution to the Knee point is selected as a candidate solution to be compared with the single solution return by the mono-objective algorithm. We compared NSGA-III with an existing mono-objective refactoring approach [71] based on the use of QMOOD quality attributes aggregated in one fitness function.

We compared our proposal to the popular design defects detection and correction tool JDeodorant [106] that does not use heuristic search techniques. The current version of JDeodorant is implemented as an Eclipse plug-in that identifies some types of design defects using quality metrics and then proposes a list of refactoring strategies to fix them.

For **RQ3**, we evaluated the benefits of our refactoring tool by several software engineers. To this end, they classify the suggested refactorings (*IR*) one by one as interesting or not. The difference with the *MP* metric is that the operations are not classified from a semantic coherence perspective but form a usefulness one.

$$IR = \frac{|\text{Useful Operations}|}{|\text{Suggested Operations}|} \in [0,1] \tag{5.14}$$

To answer the above research questions, we selected the solution from the set of non-dominated ones providing the maximum trade-off using the following strategy when comparing the different algorithms (expect the mono-objective algorithm where we select the solution with the highest fitness function). In order to find the maximal trade-off solution of the multi-objective or many-objective algorithm, we use the trade-off worthiness metric proposed by Rachmawati and Srinivasan [93] to evaluate the worthiness of each non-dominated solution in terms of compromise between the objectives. This metric is expressed as follows:

$$\mu(x_i, S) = \underset{x_j \in S, x_i \nprec x_j, x_j \nprec x_i}{Min} T(x_i, x_j) \tag{5.15}$$

where

$$T(x_i, x_j) = \frac{\sum_{m=1}^{M} \max\left[0, \frac{f_m(x_j) - f_m(x_i)}{f_m^{\max} - f_m^{\min}}\right]}{\sum_{m=1}^{M} \max\left[0, \frac{f_m(x_i) - f_m(x_j)}{f_m^{\max} - f_m^{\min}}\right]}$$

We note that x_j denotes members of the set of non-dominated solutions S that are non-dominated with respect to x_i. The quantity $\mu(x_i, S)$ expresses the least amount of improvement per unit deterioration by substituting any alternative x_j from S with x_i. We note also that $f_m(x_i)$ corresponds to the m^{th} objective value of solution x_i and $f_m^{\max}/f_m^{\min}$ corresponds to the maximal/minimal value of the m^{th} objective in the population individuals. In the above equations,

normalization is performed in order to prevent some objectives being predominant over others since objectives are usually incommensurable in real world applications. In the last equation, the numerator expresses the aggregated improvement gained by substituting x_j with x_i. However, the denominator evaluates the deterioration generated by the substitution.

Table 5.10. Summary of the empirical study design.

Research questions	**Metrics and measurements**
Quality improvement of refactored systems?	Total Quality Gain (QG) (Figure 5.7)
	Defect Correction Ratio (DCR) (Figure 5.8)
	Manual Precision (MP), Precision (PR) and Recall (RE) (Figure 5.9)
NSGA-III performance compared to other mono/many/multi-objective algorithms and a non-search-based approach?	The Inverted Generational Distance (IGD) (Figure 5.10)
	Computational Time (CT) (Figure 5.11)
	Total Quality Gain (QG) (Figure 5.12)
	Manual Precision (MP), Precision (PR) and Recall (RE) (Figure 5.13)
	Average Number of Suggested Refactorings (Figure 5.14)
Usefulness of the refactoring approach in a real-world setting?	Useful Refactorings (IR) (Figure 5.15)

5.2.3.2 Experimental Setting

Software Systems. We used a set of well-known open-source java projects that have been investigated in our previous work [125] and one project from our industrial partner Ford Motor Company. We applied our approach to the following open-source java projects: ArgoUML v0.26, Xerces v2.7, ArgoUML v0.3, Ant-Apache v1.5, Ant-Apache v1.7.0, Gantt v1.10.2 and Azureus v2.3.0.6. Xerces-J is a family of software packages for parsing XML. ArgoUML is a Java open source UML tool that provides cognitive support for object-oriented design. Apache Ant is a build tool and library specifically conceived for Java applications. GanttProject is a cross-platform tool for project scheduling. Azureus is a Java BitTorrent client for handling multiple torrents. We also considered in our experiments an industrial project, JDI, provided by our industrial partner the Ford Motor Company. It is a Java-based software system that helps Ford Motor Company analyze useful information from the past sales of dealerships data and suggests which vehicles to order for their dealer inventories in the future. This system is main key software application used by Ford Motor Company to improve their vehicles sales by selecting the right vehicle configuration to the expectations of customers. JDI is a highly

structured and several versions were proposed by software engineers at Ford during the past 10 years. Due to the importance of the application and the high number of updates performed during a period of 10 years, it is critical to ensure good refactoring solutions of JDI to reduce the time required by developers to introduce new features in the future and understand existing implementations.

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 because their defects are known and were the subject of various previous studies [55] [80] [86] [71]. Table 5.11 provides some descriptive statistics about these projects.

Table 5.11. Statistics of the studied systems.

Systems	Release	classes	KLOC	Code smells
Xerces-J	v2.7.0	991	240	82
Azureus	v2.3.0.6	1449	264	108
ArgoUML	v0.26	1358	283	1358
ArgoUML	v0.3	1409	271	1409
Ant-Apache	v1.5.0	1024	266	103
Ant-Apache	v1.7.0	1839	294	124
GanttProject	v1.10.2	245	81	41
JDI-Ford	v5.8	638	247	88

To collect operations applied in previous program versions, we used Ref-Finder. Table 5.12 shows the analyzed versions and the number of operations, identified by Ref-Finder, between each subsequent couple of analyzed versions, after the manual validation.

Table 5.12. Analyzed versions and operations collection.

Systems	Collected operation	
	Previous releases	Refactorings
Xerces-J	v1.4.2 - v2.6.1	52
GanttProject	v1.7 - v1.10.1	113
Azureus	v2.1.0.0- v2.3.0.0	146
ArgoUML	v0.11.4 - v0.17.2	182
Ant-Apache	v1.1.0- v1.4.0	177
JDI-Ford	v2.4 – v5.6	97

Parameter tuning. The algorithms have been configured according to the parameters detailed in Table 5.13. Different values have been used for the population size and the maximum number of evaluations, generating a variety of configurations related the projects sizes and the

number of objectives. For the mono-objective EA, we adopted the same approach using best fitness value criterion since multi-objective metrics cannot be used for single-objective algorithms.

Table 5.13. Parameters configuration.

Global parameters	
Population Size	190
Objectives	8
Max Evaluations	1400
Crossover Weight	0.8
Mutation Weight	0.2
Reference Points	156
NSGA-III/Gr-EA parameters	
Number of Divisions	4
IBEA parameters	
Archive Size	100
MOEA/D parameters	
Neighborhood Size	8
Max Replacements	2
H	99

Statistical Tests. 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 performed based on 31 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 (α = 5%). The latter verifies 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. For example, we compute the p-value obtained by comparing NSGA-II, IBEA, MOEA/D and mono-objective search results with NSGA-III ones. In this way, we determine whether the performance difference between NSGA-III and one of the other approaches is statistically significant or just a random result.

Subjects. Our study involved 11 subjects from the University of Michigan and 5 software engineers from Ford Motor Company. Subjects include 6 master students in Software Engineering, 4 Ph.D. students in Software Engineering, 1 faculty member in Software Engineering, and 5 junior software developers. 3 of them are females and 8 are males. All the subjects are volunteers and familiar with Java development. The experience of these subjects on Java programming ranged from 2 to 14 years. The evaluated solutions by the subjects are those that represent the maximum trade-off between the objectives using the trade-off worthiness metric proposed by Rachmawati as described in the previous section.

Scenarios. We designed our study to answer our research questions. The subjects were invited to fill a questionnaire that aims to evaluate the suggested refactorings. We divided the subjects into 8 groups according to 1) the number of studied systems (Table 5.11), 2) the number of refactoring solutions to evaluate, and 3) the number of techniques to be tested.

As shown in Table 5.14, for each system, several solutions have to be evaluated. In Table 5.14, we summarize how we divided subjects into 8 groups. In addition, as illustrated in Table 5.14, we are using a cross-validation to reduce the impact of subjects on the evaluation. Each subject evaluates different refactoring solutions for different systems.

Subjects were first asked to fill out a pre-study questionnaire containing seven questions. The questionnaire helped to collect background information such as their role within the company, their programming experience, their familiarity with software refactoring. In addition, all participants attended one lecture of 50 minutes about software refactoring and passed 10 tests to evaluate their performance to evaluate and suggest refactoring solutions. Then, the groups are formed based on the pre-study questionnaire and the tests result to make sure that all the groups have almost the same average skills. Group 8 is composed by only software engineers from Ford and evaluated only refactoring suggestions for JDI-Ford.

The participants were asked to justify their evaluation of the solutions and these justifications are reviewed by the organizers of the study (one faculty member, one postdoc and one PhD student). In addition, our experiments are not only limited to the manual validation but also the automatic validation can verify the effectiveness of our approach.

Table 5.14. Survey organization.

Subject groups	Systems	Algorithms / Approaches
Group 1	Xerces-Jv2.7.0	Gr-EA / DBEA-Eps / IBEA / JDeodorant
	ArgoUML v0.26	MOEA/D / NSGA-II / GA
	Ant-Apach v1.5.0	DBEA-Eps / IBEA / NSGA-II / GA
Group 2	Azureus v2.3.0.6	MOEA/D / JDeodorant / Gr-EA
	ArgoUML v0.3	Gr-EA / DBEA-Eps / IBEA / JDeodorant
	Ant-Apache v1.7.0	Gr-EA / DBEA-Eps / IBEA / JDeodorant
Group 3	GanttProject v1.10.2	Gr-EA / DBEA-Eps / IBEA / JDeodorant
	Xerces-Jv2.7.0	MOEA/D / NSGA-II / GA
	ArgoUML v0.26	Gr-EA / DBEA-Eps / IBEA / JDeodorant
Group 4	Ant-Apach v1.5.0	MOEA/D / JDeodorant / Gr-EA
	Azureus v2.3.0.6	DBEA-Eps / IBEA / NSGA-II / GA
	ArgoUML v0.3	MOEA/D / NSGA-II / GA
Group 5	Ant-Apache v1.7.0	MOEA/D / NSGA-II / GA
	GanttProject v1.10.2	MOEA/D / NSGA-II / GA
	JDI-Ford v5.8	Gr-EA / DBEA-Eps / IBEA / MOEA/D
Group 6	ArgoUML v0.3	DBEA-Eps / IBEA / NSGA-II / GA
	Ant-Apache v1.7.0	DBEA-Eps / IBEA / NSGA-II / GA
	JDI-Ford v5.8	NSGA-II / GA / JDeodorant
Group 7	GanttProject v1.10.2	DBEA-Eps / IBEA / NSGA-II /GA
	Xerces-Jv2.7.0	DBEA-Eps / IBEA / NSGA-II / GA
	JDI-Ford v5.8	Gr-EA / DBEA-Eps / IBEA / MOEA/D
Group 8	JDI-Ford v5.8	Gr-EA / DBEA-Eps / IBEA / MOEA/D / NSGA-II / GA / JDeodorant

Subjects were aware that they are going to evaluate the design coherence and the usefulness of the suggested refactorings, but do not know the particular experiment research questions (algorithms used, different objectives used and their combinations). Consequently, each group of subjects who accepted to participate in the study, received an online questionnaire, a manuscript guide to help them to fill the questionnaire, and the source code of the studied systems, in order to evaluate the solutions. The questionnaire is organized in an excel file with hyperlinks to visualize easily the source code of the affected code elements. Subjects are invited to select for each operation one of the possibilities: *"Yes"*, "*No"*, or *"Maybe"* (if not sure) about the design coherence and usefulness. 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.

5.2.3.3 Results and Discussions

Results for RQ1. Figure 5.7 summarizes the results of median values of the quality improvement metrics over 31 independent simulation runs after applying the proposed operations by the refactoring solution selected using the knee-point strategy [136]. In our experiments, we used all the 8 objectives in our many-objective formulation. It is clear from Figure 5.7 that all the six quality objectives are improved using our NSGA-III algorithm compared to the program version before refactoring. The reusability, understandability and extendibility are the most improved metrics and this can be explained by the fact that refactoring is not expected to change a lot the behavior/functionality of a system and this explains that some objectives were not improved significantly such as the functionality improvements metric. The same observation regarding the behavior preservation is valid for the extendibility factor because it is, to some extent, a subjective quality factor and using a model of merely static measures to evaluate extendibility is may be a not very good estimator. Overall, the NSGA-III algorithm was able to find a good trade-off between all the six quality objectives since most of them were significantly increased and no one of these metrics was decreased comparing the initial version of the system before refactoring. The variation in terms of quality improvements between the different systems is not high. ArgoUML and Ant-Apache were the main systems who are significantly improved and this can be explained by the lower quality of these projects comparing to the remaining systems before refactoring.

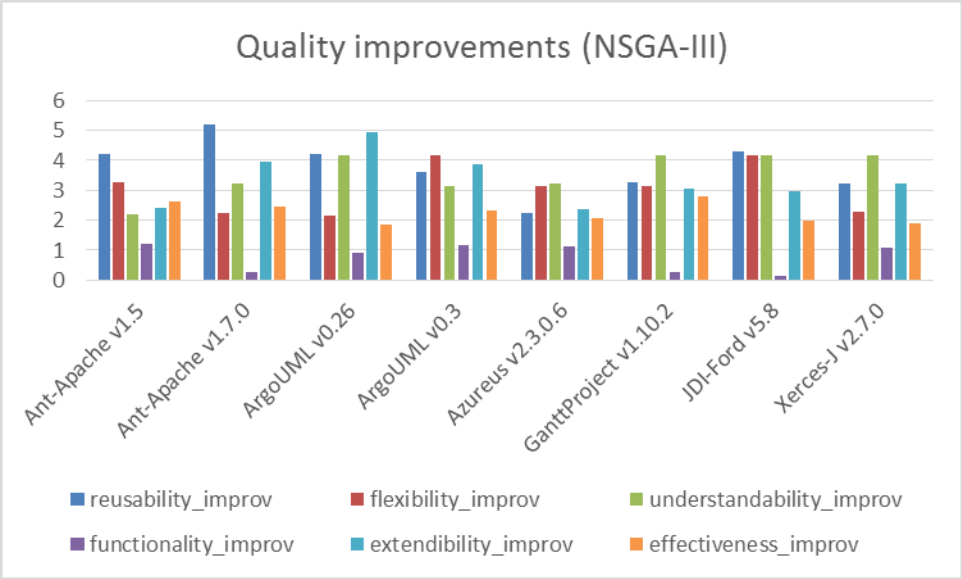


Figure 5.7. Average quality improvements, over 31 runs, on the different systems using NSGA-III.

As described in Figure 5.8, after applying the proposed refactoring operations by our approach (NSGA-III), we found that, on average, 82% of the detected defects were fixed (DCR) for all the eight studied systems. This high score is considered significant in terms of improving the quality of the refactored systems by fixing the majority of defects of various types (blob, spaghetti code, functional decomposition [55], god class, data class, and feature envy [138]).

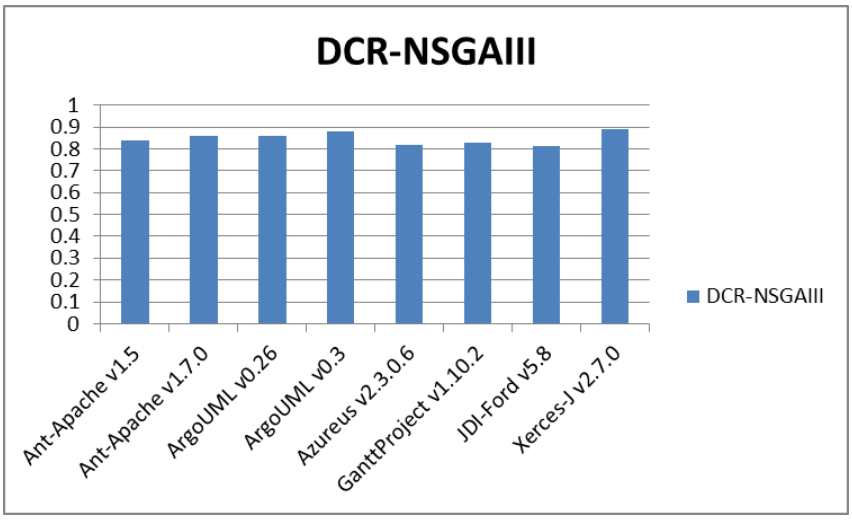


Figure 5.8. Average percentage of fixed defects, over 31 runs, on the different systems using NSGA-III.

A closer look to the fixed defects in Xerces-J v2.7.0, the one with the highest percentage of fixed defects (86%), is detailed in Table 5.15. Fixed code smells distribution in Xerces-J v2.7.0.

Table 5.15. Fixed code smells distribution in Xerces-J v2.7.0.

Code smell	Flawed classes	Number of fixed code smells
Blob	143 (32% overlap)	30 (%89)
Data Class		19 (%83)
God Class		10 (%64)
Feature Envy		25 (%96)
Functional Decomposition		13 (%94)
Spaghetti Code		39 (%92)

On Table 5.15. Fixed code smells distribution in Xerces-J v2.7.0., it is noticeable that some code smells are harder to fix (such as God classes) compared to others (Feature Envy), further analysis needs to be done to better understand this observation. Although the bad smell detection literature suggests a wide variety of code smells to be corrected, we narrowed our selection to the five types that have given significant results compared to the others. This can be explained by the fact that the defects types that are not fixed require the considerations of more refactoring operations rather than those included in this work. In addition, some of these defects are difficult to detect just using structural metrics [80].

We also need to assess the correctness/meaningfulness of the suggested refactorings from the developers' point of view. Figure 5.9 confirms that the majority of the suggested refactorings improve significantly the code quality while preserving design's semantic coherence. On average, for all of our studied systems, an average of around 91% of proposed refactoring operations are considered by potential users to be semantically feasible and do not generate semantic incoherence.

In addition to the manual evaluation, we automatically evaluate our approach without using the feedback of potential users to give a 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 Table 5.12. We use Ref-Finder to identify refactoring operations that are applied to the program version under analysis and the next version. Figure 5.9 summarizes our results. We found that a considerable number of proposed refactorings (an average of 59% 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 suggestions in terms of defect-correction ratio, semantic coherence from the point of view of 1) potential users of our refactoring tool and 2) expected refactorings applied to the next program version.

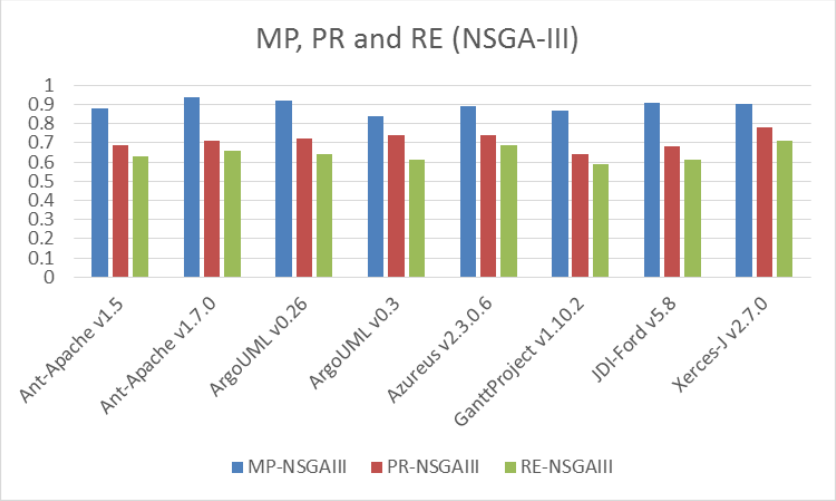


Figure 5.9. Average manual and automatic design coherence measures (MP, PR and RE), over 31 runs, on the different systems using NSGA-III.

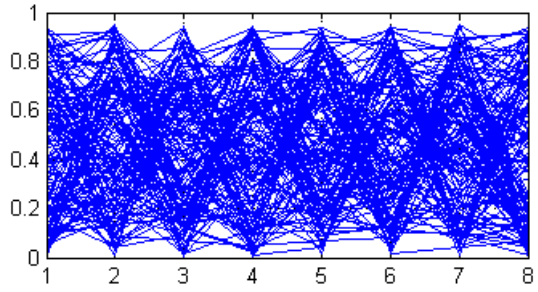
Results for RQ2. In this section, we focus first on the comparison between our NSGA-III adaption and other many-objective algorithms using the same adaptation. Table 5.16 shows the median IGD values over 31 independent runs for all algorithms under comparison. We have used pairwise comparisons, so we do not need to adjust p-values. After applying Cohen's d effect size we noticed that the effect size between the paired comparison of NSGA-III with each of the remaining algorithms is higher than 0.8 except for Gr-EA, which effect values were found to be medium.

Table 5.16. Median IGD values on 31 runs (best values are in bold and underlined, second best values are in bold). ~ means a large value that is not interesting to show. The results were statistically significant on 31 independent runs using the Wilcoxon rank sum test with a 95% confidence level (α = 5%).

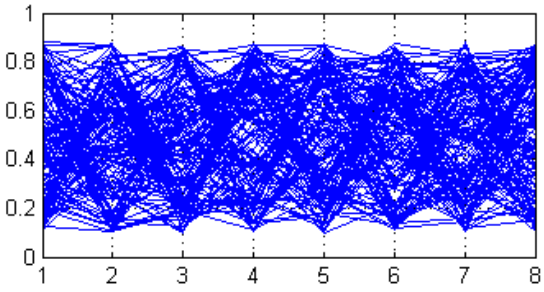
System	NSGA-III	Gr-EA	DBEA-Eps	IBEA	MOEA/D	NSGA-II
ArgoUML v0.26	**4.113 x 10^{-3}**	4.229 x 10^{-3}	**4.206 x 10^{-3}**	4.329 x 10^{-3}	4.342 x 10^{-3}	~
Xerces v2.7	**7.998 x 10^{-3}**	8.308 x 10^{-3}	**8.181 x 10^{-3}**	8.399 x 10^{-3}	8.431 x 10^{-3}	~
ArgoUML v0.3	**5.499 x 10^{-3}**	5.677 x 10^{-3}	**5.603 x 10^{-3}**	5.712 x 10^{-3}	5.733 x 10^{-3}	~
Ant-Apache v1.5	**6.008 x 10^{-4}**	6.256x 10^{-4}	**6.224 x 10^{-4}**	6.325 x 10^{-4}	6.333 x 10^{-4}	~
Ant-Apache v1.7.0	**6.202 x 10^{-3}**	6.412 x 10^{-3}	**6.377 x 10^{-3}**	6.489 x 10^{-3}	6.539 x 10^{-3}	~
Gantt v1.10.2	**7.806 x 10^{-3}**	8.002 x 10^{-3}	**7.968 x 10^{-3}**	8.088 x 10^{-3}	8.101 x 10^{-3}	~
Azureus v2.3.0.6	**6.933 x 10^{-4}**	7.112 x 10^{-4}	**7.075 x 10^{-4}**	7.191 x 10^{-4}	7.208 x 10^{-4}	~
JDI-Ford	**5.748 x 10^{-4}**	6.066 x 10^{-4}	**5.851 x 10^{-4}**	6.294 x 10^{-4}	6.646 x 10^{-4}	~

All the results were statistically significant on the 31 independent simulations using the Wilcoxon rank sum test with a 95% confidence level (α = 5 %). NSGA-III strictly outperforms NSGA-II and gives slightly better results to those of the other many-objective algorithms. It is worth noting that for problems formulations with more 3 objectives, NSGA-II performance is dramatically degraded, which is simply denoted by the ~ symbol. The performance of NSGA-III could be explained by the interaction between (1) Pareto dominance-based selection and (2) reference point-based selection, which is the distinguishing feature of NSGA-III compared to other existing many-objective algorithms.

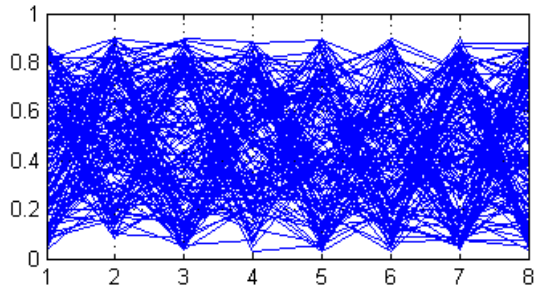
NSGA-III



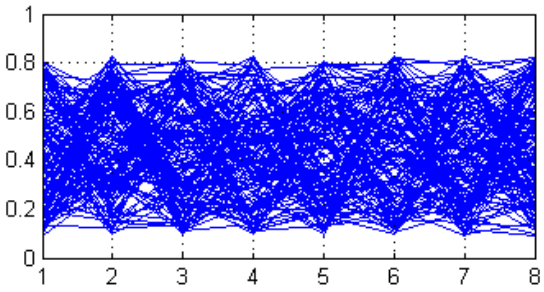
IBEA



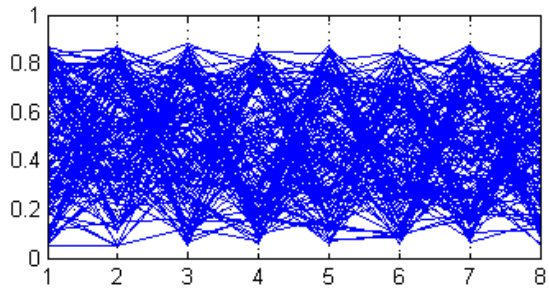
DBEA-Eps



MOEA/D



Gr-EA



NSGA-II

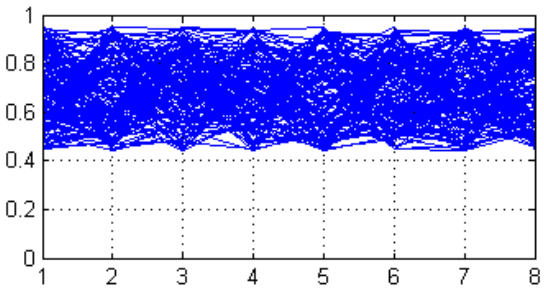


Figure 5.10. Value path plots of non-dominated solutions obtained by NSGA-III, DBEA-Eps, Gr-EA, IBEA, IBEA, and NSGA-II during the median run of the 8-objective refactoring problem on ArgoUML v0.26. The X-axis represents the different objectives while the Y-axis shows the variation of fitness values between [0...1].

Figure 5.10 describes value path plots of all algorithms for the 8-objective refactoring problem on Argo-UML. The horizontal axis shows the objective functions while the vertical axis marks its related values. The objectives values were normalized between 0 and 1 and set to be

minimized. In terms of convergence, the algorithm whose solutions are closest to the ideal vector of height zeros has better convergence ratio. Thus, NSGA-III and DBEA-Eps outperform the remaining algorithms. Also, the spread of NSGA-III solutions vary in between [0, 0.9] presents a slightly better diversity than its follower DBEA-Eps whose solutions vary in between [0, 0.85]. On the other hand, the worst convergence is associated to NSGA-II as its solutions are so far from the ideal vector, and even it diversity is so reduced which may explain the stagnation of its evolutionary process. We conclude that although NSGA-II is the most famous multi-objective algorithm in SBSE, it is not adequate for problems involving over 3 objectives and NSGA-III is a very good candidate solution for tackling many-objective SBSE problems.

When using optimization techniques, the most time-consuming operation is the evaluation step. Thus, we studied the execution time of all many/multi-objective algorithms used in our experiments. Figure 5.11 shows the evolution of the running times of the different algorithms on the ArgoUMLv0.26 system, the largest system in our experiments. It is clear from this figure that for 8 objectives NSGA-III is faster than IBEA. This observation could be explained by the computational effort required to compute the contribution (IGD) of each solution. In comparison to MOEA/D, MOEA/D is slightly faster than NSGA-III since it does not make use of non-dominated sorting. Note that the experiments were conducted on a single machine (i7 – 2.70 GHz, 12.0 GB – DDR3, SSD - 520MB/s) which may not be the optimal setting for some of these heuristics that can perform faster in an appropriate distributed or parallel environment.

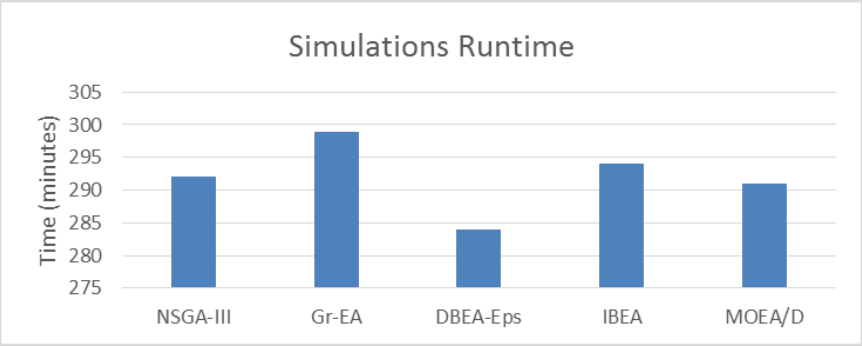


Figure 5.11. Average Computational time values on 31 runs on refactoring ArgoUMLv0.26.

We compared also the different search algorithms using metrics related to quality improvements, number of fixed defects, number of generated refactorings and a manual

inspection of the results to check the correctness of the suggested operations. Figure 5.12 shows that our NSGA-III algorithm presents the best compromise between the different quality attributes among all the other search algorithms. In addition, it is clear that the many-objective algorithms propose a better trade-off in terms of quality improvements than the mono and multi-objective techniques.

Since it is not sufficient to outperform existing search-based refactoring techniques, we compared our proposal to a popular design defects detection and correction tool JDeodorant. We first note that JDeodorant (like mono-objective approaches also) provides only one refactoring solution, while NSGA-III generates a set of non-dominated solutions. It can be seen that NSGA-III provides better results than JDeodorant, in average. The main reason can be related to the fact that JDeodrant provides a template of possible refactorings to apply for each detected defect but it is difficult to generalize such refactoring solutions since a defect can have several different refactoring strategies.

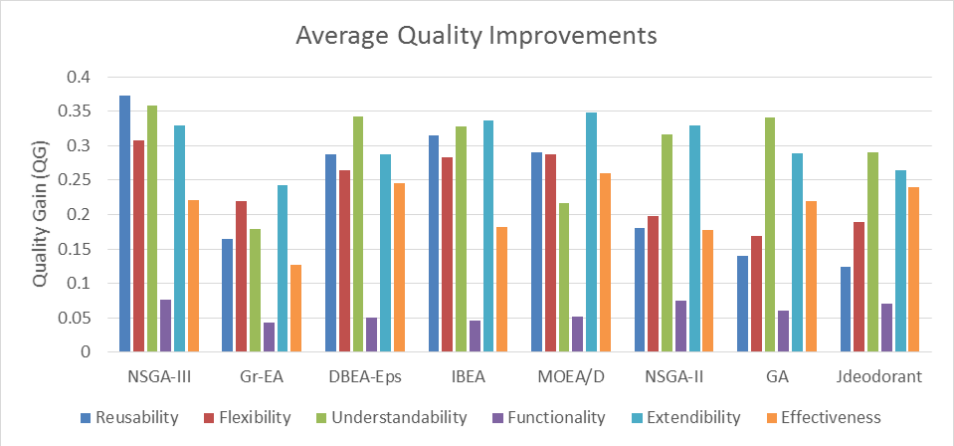


Figure 5.12. Average quality improvements, over 31 runs, on the different systems.

Figure 5.12 confirms that the majority of the suggested refactorings by NSGA-III improve significantly the code quality while preserving design's semantic coherence better than most of the other search algorithms. In addition, we automatically evaluated our approach without using the feedback of potential users to give a more quantitative evaluation. Thus, we compared 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 Table 5.12. Figure 5.13 confirms the outperformance of NSGA-III comparing to the remaining techniques.

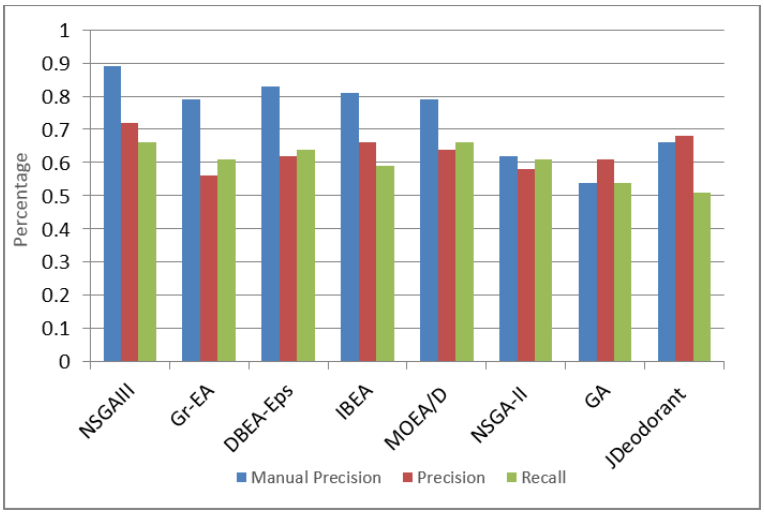


Figure 5.13. Average manual and automatic design coherence measures (MP, PR and RE), over 31 runs, on the different systems.

We evaluated the number of operations (NO) suggested by the best refactoring solutions on the different systems over 31 runs. Figure 5.14 presents the code changes score for each algorithm, calculated by summing the size (number of operations) of each solution assigned to one project, divided by the number of projects. It is clear that our NSGA-III approach succeeded in suggesting solutions that do not require high code changes. However, IBEA generated less number of refactorings than our approach but this can be due to the fact that our technique improved better the quality comparing to IBEA's solutions. Thus, it may require a higher number of refactorings to better improve the quality attributes. Another observation is that the number of refactorings proposed by JDeodorant is lower than the number of refactorings suggested by NSGA-III. However, the number of defects fixed by NSGA-III is higher than JDeodorant thus it is normal in this case that NSGA-III generates a higher number of refactorings.

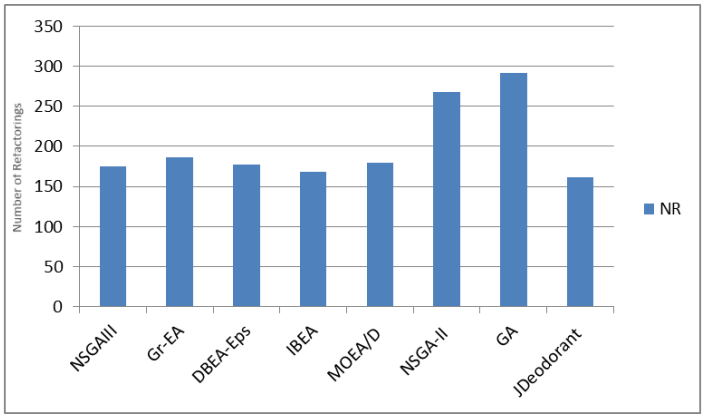


Figure 5.14. Average suggested number of refactoring operations, over 31 runs, on the different systems.

Results for RQ3. We asked the software engineers involved in our experiments to evaluate the usefulness of the suggested refactorings to apply one by one. In fact, sometimes these operations can improve the quality and preserve the semantics but developers will consider them as not useful due to many reasons such as some code fragments are not used/updated anymore or includes some features that are not important. Figure 5.15 shows that NSGA-III clearly outperforms existing work by suggesting useful refactoring operations for developers.

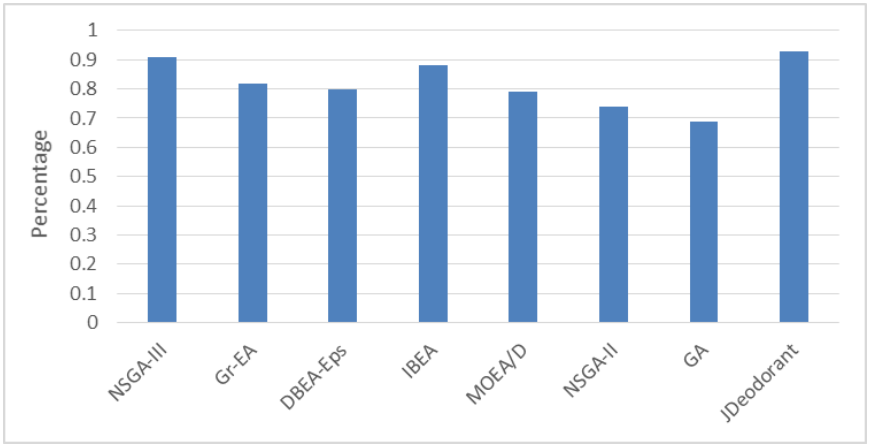


Figure 5.15. Average of percentages of useful operations (IR), on the different systems using NSGA-III.

During the survey, the software engineers confirm that the main limitation related to the use of NSGA-III for software refactoring is the high number of equivalent solutions. However, found the idea of the use of the Knee point as described previously useful to select a good solution. We will investigate in our future work different other techniques to select the region of interest based on the preferences of developers.

5.2.4 Conclusion

We propose a novel formulation of the refactoring problem as a many-objective problem, based on NSGA-III, using the quality attributes of QMOOD as objectives along with the number of refactorings and the design coherence preservation. This initial empirical investigation has shown that a possible conflict among QMOOD objectives may occur depending on the kind of employed refactorings. In this context, a statistical analysis needs to be conducted to prove the validity of this insight. Furthermore, we did not yet prioritize any objective(s) although the definition of QMOOD and the capability of NSGA-III allows it. It would be interesting to compare between multiple refactored systems while each one of them is the result of high prioritization of one quality objective.

We evaluated our approach on seven large open source systems. We implemented our approach and evaluated it on seven large open source systems and one industrial project provided by our industrial partner. We compared our findings to several other many-objective techniques (IBEA, MOEA/D, GrEA, and DBEA-Eps), a mono-objective technique and an existing refactoring technique not based on heuristic search. Statistical analysis of our experiments over 31 runs shows the efficiency of NSGA-III as powerful algorithm to tackle many objective formulations.

We also studied the impact of refactoring on fixing code smells. Although we were able to fix most of the detected defects, our defect selection types were limited due to limited types of used refactorings. Another limitation to take into account is the possibility of introducing code smells while performing the refactoring operations, for example, we noticed that, some fixed feature envied entities have had an intrusion of shotgun surgery defect. Since the later type is not covered in this work, it did not affect our DCR, but it gives a strong indication for further investigation about how to perform refactoring interactively with the detection to avoid such drawback.

For the qualitative evaluation, we were able to show promising results, along with JDeodorant, which, as a tool, was highly appreciated by the participants because of its simplicity,

ease of use and the possibility to preview changes and visualize entities. Unfortunately, limited to four types of smells.

In future work, multiple research directions are to be taken from some of the previously mentioned limitations and they are mainly linked to (1) the problem formulation and (2) NSGA-III tuning. Firstly, investigating the statistical significance of the refactorings impact analysis on internal and external attributes can be an interesting research direction that helps in better understanding to what extent each refactoring type can affect the existing quality models. Moreover, code smells can be also described in terms of structural metrics, such statistical investigation will help in the validation of attributing a subset of refactorings to a known type of code smell. As for NSGA-III tuning, we will investigate the impact of different parameter settings on the quality of our results, in particular, the size of the reference set $|Z^r|$ can either be predefined and calculated based on the number of objectives and the number of desired divisions in the hyperplane or preferentially introduced by the user. Augmenting the density of the hyperplane i.e. increasing the number of used reference points will refine the niche count and thus will provide solutions with better diversity. Since, in our experiments, we only considered the predefined size of reference set, we plan in the future to investigate the impact of varying this parameter on the quality of the generated solutions. Another interesting research direction regards the prioritization of the code smells to be removed, for two candidate solutions extracted from the Pareto-front, they have equivalent fitness functions' values but their impact on reducing code smells may vary in terms of number and types of fixed code smells. So, this can be used as an additional developer preference to compare between given solutions. Moreover, the solution's robustness can be also studied as to take into account the uncertainties related to analyzed classes importance and code smells severities while suggesting refactoring operations. Furthermore, we plan to work on adapting NSGA-III to additional software engineering problems and we will perform more comparative studies on larger open source systems. Nevertheless, this extensive study has shown a direction using NSGA-III to handle as many as 8 objectives in the context of solving software engineering problems and would remain as one of the first studies in which such a large number of objectives have been considered.

5.3 Dynamic Interactive Software Refactoring

5.3.1 Introduction

Successful software products evolve through a process of continuous change. However, this may lead to poor design quality and make systems complex. This complexity leads to significantly reduced productivity, decreased system performance, increased fault-proneness, makes software costly and has even led to projects being canceled. Many studies report that software maintenance activities to improve the quality of systems consume up to 90% of the total cost of a typical software project. It has also been shown that software engineers spend around 60% of their time in understanding the code [140].

Clearly, software engineers need better ways to reduce and manage the growing complexity of software systems and improve their productivity. Refactoring, which improves design structure while preserving functionality [15], is an extremely important solution to address this challenge. There has been much work on various techniques and tools for software refactoring [141] [68] [142] [71], and these studies can be classified into two main categories: manual and fully-automated approaches.

Murphy-Hill et al. [142] found in their empirical study with developers that in almost 90% of cases they performed refactorings manually and did not use automated refactoring tools. Kim et al. [143] confirmed this observation, finding that the interviewed developers from Microsoft preferred to perform refactoring manually in 86% of cases. Several studies [144] have shown that manual refactoring [15] is error-prone, time-consuming, not scalable and not useful for radical refactoring that requires an extensive application of refactorings to correct unhealthy code.

Fully-automated refactoring has several drawbacks as well. It lacks flexibility since developers have to either accept or reject the entire refactoring solution. It fails also to consider the developer perspective and feedback because suggested refactoring solutions cannot be updated dynamically. It is limited to structural improvements, which leads to infeasible refactoring solutions. Finally, it proposes a long static list of refactorings to be applied, but

developers may not have enough time to apply all of them. Thus, fully-automated refactoring methods are not useful for floss refactoring where the goal is to maintain good design quality while modifying existing functionality. The developers have to accept the entire refactoring solution even though they prefer, in general, step-wise approaches where the process is interactive and they have a total control of the refactorings being applied.

Very few studies have considered an interactive approach with the developers to improve software quality [68] [145] [146]. In [145], an approach was proposed for the interactive remodularization of packages using an interactive genetic algorithms. Another similar approach [146] was proposed based on supervised learning to fix existing modularizations of a software design. However, both approaches were limited to just moving classes between packages. In addition, the interaction with the designer is restricted to evaluating most of the solutions which is a time-consuming task. The feedback received from the designer is just an evaluation of the remodularization solutions. In our previous studies [26] [125], we proposed a fully-automated technique, using evolutionary algorithms, to find the best refactoring solutions that improve quality metrics and reduce the number of recommended refactorings. Our previous work did not consider any interactions with the programmers and cannot update/repair refactoring solutions based on new code changes introduced by programmers.

We propose in this work an interactive recommendation approach for software refactoring that dynamically adapts and suggests refactorings to developers based on introduced new code changes (e.g. to update existing features) and the developers' feedback such as accepting, rejecting or modifying a refactoring. The suggested refactorings are presented to the developers one by one as a sequence of transformations. The proposed approach extends our previous work where we proposed a *fully-automated* technique, using the multi-objective evolutionary algorithm NSGA-II , to find the best refactoring solutions that improve quality metrics and reduce the number of recommended refactorings. Our previous work did not consider any interactions with the developers and cannot update/repair refactoring solutions based on new code changes introduced by developers.

Our approach starts by finding upfront a set of refactoring solutions that optimize the two above objectives. One of the challenges when adapting a multi-objective technique to a software

engineering problem, and any real-world problem in general, is how to select the best solution from the set of non-dominated ones, called the Pareto front. To this end, we propose, for the first time, the use of innovization (innovation through optimization) [22] to analyze and explore the Pareto front interactively and implicitly with the developers. Our innovization algorithm starts by finding the most frequent refactorings among the set of non-dominated refactoring solutions. Based on this analysis, the suggested refactorings are ranked and suggested to the developer one by one. The developer can approve, modify or reject each suggested refactoring. This feedback is later used to update the ranking of the suggested refactorings.

After a number of introduced code changes and interactions with the developer (e.g. a number of rejected refactorings), NSGA-II will continue to execute on the new modified system to repair the set of good refactoring solutions based on the feedback received from the developer. The feedback received from the developers will be also used as a set of new constraints to consider for the next iterations of NSGA-II. The algorithm will avoid, for example, including rejected refactorings by the developers when generating new solutions or repairing existing ones.

We implemented our proposed approach and evaluated it on a set of 9 open source systems and 2 industrial systems provided by our industrial partner, i.e., the Ford Motor Company. Statistical analysis of our experiments over 50 runs showed that our proposal performed significantly better than four existing search-based refactoring approaches [86] [6] [71] [147] and an existing refactoring tool not based on heuristic search, JDeodorant [106]. The software developers who participated in our experiments confirmed the relevance of the suggested refactoring and the flexibility of the tool in modifying and adapting the suggested refactorings.

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

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

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

(3) This work reports the results of an empirical study on an implementation of our approach. The results obtained provide evidence to support the claim that our proposal is more efficient, on average, than existing refactoring techniques based on a benchmark of 9 open source systems and 2 industrial projects. This work also evaluates the relevance and usefulness of the suggested refactorings for software engineers in improving the quality of their systems.

5.3.2 Approach

5.3.2.1 Interactive and Dynamic Evolutionary Multi-objective Optimization: Background

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

Interacting with the human user means allowing the latter to inject his/her preferences in the computational search algorithm and then handling these preferences so that the search process is guided based on them. There exists a lot of work about preference incorporation in EMO. To express his/her preferences, the user needs some preference modeling tools. The most used ones are:

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

– *Solution ranking*: The user is provided with a sample of solutions (a subset of the current population) and is invited to perform pairwise comparisons between pairs of solutions in order to rank the sample's solutions where indifference may exist between the solutions to rank.

– *Objective ranking*: Pairwise comparisons between pairs of objectives are performed in order to rank the problem's objectives where indifference may exist between some objectives.

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

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

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

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

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

Based on these preference modeling tools, we observe that the goal of a preference-based EMO algorithm is to assign different importance levels to the problem's objectives with the aim to guide the search towards the ROI (Region of Interest) that is the portion of the Pareto Front that best matches the user preferences. In fact, usually, the user is not interested with the whole Pareto front and thus he/she is searching only for his/her ROI from which the problem's final solution will be selected. Several preference-based EMO algorithms have been proposed and used to solve real problems such as PI-EMOA [148], iTDEA [149], NOSGA [150], DF-SMS-EMOA [151], just to cite a few. There are several algorithmic challenges that should be overcome such the preservation of the Pareto dominance, the preservation of population diversity, the scalability with the number of objectives, etc. [149].

It is very important to note that till today, *the user's preferences are expressed and handled in the objective space*. Hence, one of the original aspects of our work, as detailed later, is

allowing the user (software engineer) to express his/her preferences in the decision space and then handling these preferences to help the user finding his/her most desired refactoring solution. Moreover, our approach helps the user in eliciting his/her preferences, which is very important for any preference-based EMO algorithm. These preferences are introduced implicitly by moving between the Pareto front of non-dominated solutions after getting a feedback from the user about just a few parts of the solution to better understand his preferences. This implicit exploration of the Pareto front will be detailed in the next section about the formulation of our refactoring problem.

The incorporation of the preferences may require handling dynamicity issues related to the introduce changes to the solution or the input (e.g. software system). Handling dynamicity in EMO means solving dynamic problems where the objective functions and or the constraints may change over time such due for example to the dynamic nature of most of software evolution problems including software refactoring. Applying evolutionary algorithms (EAs) to solve Dynamic Multi-Objective Problems (DMOPs) has obtained great attention among researchers thanks to the adaptive behavior of evolutionary computation methods. A DMOP consists in minimizing or maximizing an objective function vector under some constraints over time. Its general form is the following [152]:

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

where M is the number of objective functions, t is the time instant, P is the number of inequality constraints, Q is the number of equality constraints, x_i^L and x_i^U correspond respectively to the lower and upper bounds of the variable x_i. A solution x_i satisfying the (P+Q) constraints is said feasible and the set of all feasible solutions defines the feasible search space denoted by Ω. In this formulation, we consider a minimization MOP since maximization can be easily turned into minimization based on the duality principle by multiplying each objective function by -1 and transforming the constraints based on the duality rules.

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

Definition 1: Pareto optimality

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

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

Definition 2: Pareto dominance

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

Definition 3: Pareto optimal set

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

Definition 4: Pareto optimal front

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

Based on these definitions, we can conclude that a dynamic problem could be seen as a sequence of static problems. In this approach, we consider integrated dynamicity in our interactive multi-objective algorithm by considering a new input (new version of the system after introduced changes and refactorings) for every continuation of the execution of the algorithm

after several interactions with the user. Thus, we considered our dynamic problem as a sequence of static ones. In the next section, we describe an overview of our dynamic interactive refactoring approach then a detailed formulation of our solution.

5.3.2.2 Approach Overview

The goal of our approach is to propose a new dynamic interactive way for software developers to refactor their systems. The general structure of our approach is sketched in Figure 5.16.

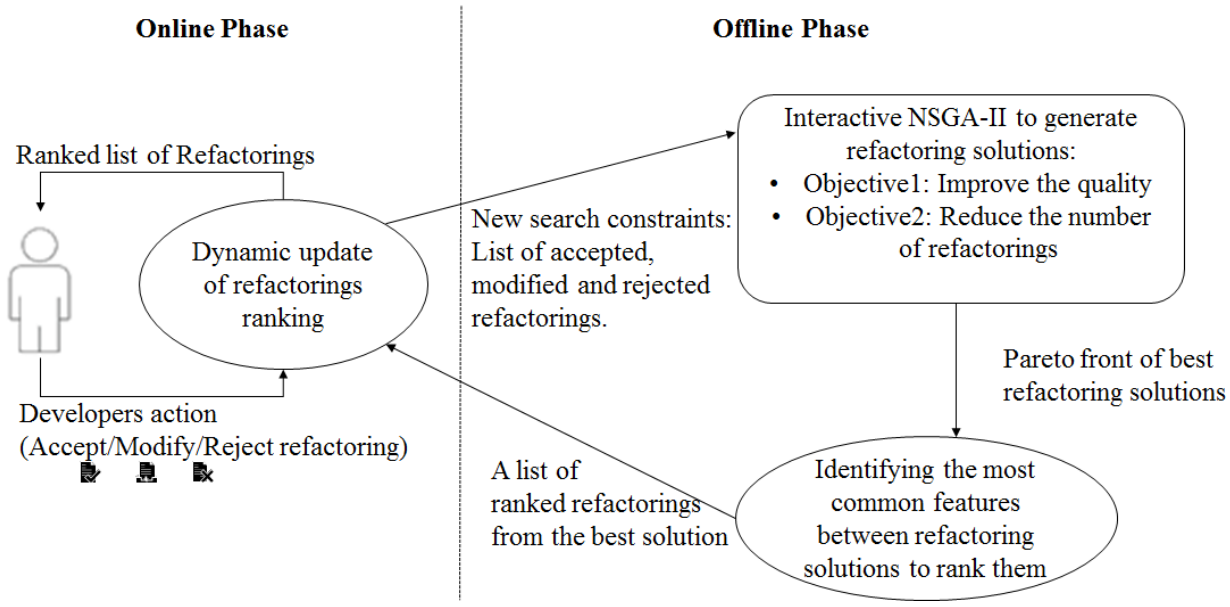


Figure 5.16. Approach overview.

Our technique comprises two main components. The first component is an *offline phase*, executed in the background, when developers are modifying the source code of the system. During this phase, the multi-objective algorithm, NSGA-II [21], is executed for a number of iterations to find the non-dominated solutions balancing the two objectives of improving the quality which corresponds to minimizing the number of code smells, maximizing/preserving the semantic coherence of the design and improving the QMOOD (Quality Model for Object-Oriented Design) quality metrics [126], and the second objective of minimizing the size of refactoring solutions (number of operations).

The output of this first step of the offline phase is a set of Pareto-equivalent refactoring solutions, that optimizes the above two objectives. As explained in Algorithms 1 and 2, the second step of the offline phase explores this Pareto front in an intelligent manner using the innovization algorithm to rank recommended refactorings based on the common features between the non-dominated solutions. In our adaptation, we considered the hypothesis that the most redundant/frequent refactorings between the non-dominated solutions are the most important ones. The best refactoring solution will contain most of the frequently used refactorings between non-dominated solutions. Thus, the output of this second step of the offline phase is a set of ranked solutions based on this redundancy score.

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

After a number of interactions, developers may have modified or rejected a high number of suggested refactorings or have introduced several new code changes (new functionalities, fix bugs, etc.). In this case, the first component of our approach is executed again to update the last set of non-dominated refactoring solutions by continuing the execution of NSGA-II based on the two objectives defined in the first component as described in Algorithm 3 and also the new constraints summarizing the feedback of the developer. In fact, we consider the rejected refactorings by the developer as constraints to avoid generating solutions containing several already rejected refactorings. This may lead to reducing the search space and thus a fast convergence to better solutions. Of course, the continuation of the execution of interactive NSGA-II is taking as input the updated version of the system after the interactions with developers.

The whole process will continue to be executed until the developers decide to stop refactoring the process.

Algorithm 5.1. Pseudo-code of the Dynamic Interactive NSGA-II for software refactoring at generation *t*

Input

Sys: system to evaluate, P_t: parent population

Output

P_{t+1}

00: **Begin**

01: /* Test if any user interaction occurred in the previous

02: iteration */

03: **If** UserFeedback = *TRUE* **then**

04: /* Rejected refactoring operations as constraints */

05: $C_t \leftarrow$ Get-Constraints ();

06: /* Updated source code after applying changes */

07: ...*Sys* ← Get-Refactored-System ();

08: UserFeedback ← *FALSE;*

09: **End If**

10: $S_t \leftarrow \emptyset$, $i \leftarrow 1$;

11: $Q_t \leftarrow$ Variation (P_t);

12: $R_t \leftarrow P_t \cup Q_t$;

$P_t \leftarrow$ evaluate (P_t, C_t, *Sys*);

13: (F_1, F_2, ...) ← Non-dominationed-Sort (R_t);

Repeat

14: $S_t \leftarrow S_t \cup F_i$; $i \leftarrow i+1$;

15: **Until** $| S_t | \geq N$;

$F_l \leftarrow F_i$; //Last front to be included*/

15: **If** $| S_t | = N$ **then**

$P_{t+1} \leftarrow S_t$;

Else

15: $P_{t+1} \leftarrow \bigcup_{j=1}^{l-1} F_j$;

/*Number of points to be chosen from F_l*/

16: $K \leftarrow N - |P_{t+1}|$;

/*Crowding distance of points in F_l */

17: Crowding-Distance-Assignment(F_l);

18: …Quick-Sort(F_l);

19: /*Choose K solutions with largest distance*/

$P_{t+1} \leftarrow P_{t+1} \cup$ Select(F_l, k);

End If

If *t+1 = Threshold* **then**

UserFeedback ← *TRUE;*

/* Select and rank the best front */

...Rank-Solution (F_1);

Threshold ← *Threshold + t+1;*

End If

End

Algorithm 5.2. Pseudo-code of the Innoviazation procedure to rank the non-dominated refactoring solutions.

Rank Solution procedure

Input

NS: Non-dominated SolutionSet of the first front

Output:

RNS: Ranked Non-dominated SolutionSet

Begin

/*Compute the score of each Refactoring Operation*/

HM ← Rank-Refactoring-Operation (NS);

/*Compute the score of each solution*/

For *i=1* **to** |*NS*| **do**

$SolutionRank_i \leftarrow 0$;

For each *j=1* **to** $|NS_i|$ **do**

$SolutionRank_i \leftarrow SolutionRank_i + HM$[Hash($NS_{i,j}$)];

End for

End for

RNS ← Quick-Sort(*NS*);

Get-User-Feedback(*RNS*);

End

Rank Refactoring Operation procedure

Input

NS: Non-dominated SolutionSet of the first front

Output

HM: HashMap of refactorings along with their occurrences.

Begin

HM ← Ø;

/* Compute the number of occurrence of each refactoring operation*/

For *i=1* **to** |*NS*| **do**

For each *j=1* **to** |NS_i| **do**

/* If a refactoring operation does not exist in the list, add its hash and set its occurrence number to 1*/

If ($NS_{i,j} \notin HM$) **then**

HM ← *HM* ∪ Hash($NS_{i,j}$);

HM[Hash($NS_{i,j}$)] ← 1;

/* If a refactoring operation exists in the list, increment its occurrence number */

else

HM[Hash($NS_{i,j}$)] ← *HM*[Hash($NS_{i,j}$)] + 1;

End for

End for

End

Algorithm 5.3. Pseudo-code of the procedure to manage the interactions with the developer (online phase).

GUF (Get User Feedback) procedure

Input

RNS: Ranked Non-dominated SolutionSet

Output

HM: HashMap of refactorings along with their occurrences.

Begin

Applied-Refactorings ← Ø;

Rejected-Refactorings ← Ø;

For *i=1* **to** |*RNS*| **do**

ref[i] ← 0;

End for

/* Main loop to suggest refactorings one by one to the user*/

While |*Rejected-Refactorings*|< α **do**

/* Select index of the the solution with highest rank*/

index ← Max-Rank(*RNS*);
d ← User-Decision($RNS_{index,ref[index]}$);
/* If the user has applied or modified the operation*/
If (*d* = True) **then**
Applied-Refactorings ← *Applied-Refactorings* ∪ $RNS_{index,ref[index]}$;
/* If the user has rejected the operation*/
else
Rejected-Refactorings ← *Rejected-Refactorings* ∪ $RNS_{index,ref[index]}$;
End if
ref[index] ← *ref[index]* + 1;
/* Update solutions indexes */
For *i=1* **to** |*RNS*| **do**
Update-Rank($RNS_{i;}$ *Applied-Refactorings*, *Rejected-Refactorings*)
End While
End

5.3.2.3 Multi-objective formulation

In our previous work [86] [104], we proposed a fully automated approach, to improve the quality of a system while preserving its domain semantics. It uses multi-objective optimization based on NSGA-II to find the best compromise between code quality improvements and reducing the number of code changes. In this work, we are introducing the interactive component to our NSGA-II algorithm which radically change the process of finding good refactoring solutions comparing to [86] [104]. We will compare later in the experiments the performance of both algorithms. We present in the following the different adaptation steps of our approach.

Solution presentation. A solution consists of a sequence of *n* refactoring operations involving one or multiple source code elements of the system to refactor. The vector-based representation is used to define the refactoring sequence. Each vector's dimension has a refactoring operation and its index in the vector indicates the order in which it will be applied. For every refactoring, pre- and post-conditions are specified to ensure the feasibility of the operation as detailed in [87].

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

The size of a solution, i.e. the vector's length is randomly chosen between an upper and lower bound values. The determination of these two bounds is similar to the problem of bloat control in genetic programming where the goal is to identify the tree size limits. Figure 5.19 shows an example of a refactoring solution including 3 operations applied to a simplified version of a solution applied to JVacation v1.0, a Java open-source trip management and scheduling software.

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

For the crossover, we use the one-point crossover operator. It starts by selecting and splitting at random two parent solutions. Then, this operator creates two child solutions by putting, for the first child, the first part of the first parent with the second part of the second parent, and vice versa for the second child. This operator must ensure the respect of the length limits by eliminating randomly some refactoring operations. It is important to note that in multi-objective optimization, it is better to create children that are close to their parents in order to have a more efficient search process. For this reason, we control the cutting point of the one-point crossover operator by restricting its position to be either belonging to the first tier of the refactoring sequence or belonging to the last tier.

For mutation, we use the bit-string mutation operator that picks probabilistically one or more refactoring 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, the different pre- and post-conditions are checked to ensure the applicability of the newly generated solutions [87]. We also apply a repair operator to randomly select new refactorings to replace those creating conflicts.

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

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

$f(x) = sum\_of\_parameters\_in\_operations(x)$ where x is the solution is to evaluate. Any solution with operations being executed in the same code elements will have better (lower) fitness value for this objective. Such definition of the objective is in favor of code locality since it encourages refactoring a same code fragment, as developers eventually prefer to refactor specific elements that they are mostly familiar with instead of applying scattered changes throughout the whole system. The proposed fitness function is different than our previous work [86] where only the number of applied refactorings are counted. In fact, every refactoring type may have a different impact on the system in terms of number of changes which can be identified using our new formulation.

Maximize *software quality as an objective:* Many studies has been utilizing structural metrics as a basis of defining quality indicators for a good system design [70] [125]. As an illustrative example, Bansiya et al. proposed a set of quality measures, using the ISO 9126 specification, called QMOOD. The adaptation of the QMOOD model in the problem of software refactoring has been studied in [25] [92] and provided considerable improvements to the object-oriented system's design. Thus, we used the QMOOD model to estimate the effect of the suggested refactoring solutions on quality attributes. QMOOD has the advantage that it defines six high-level design quality attributes (reusability, flexibility, understandability, functionality,

extendibility, and effectiveness) that can be calculated using 11 lower level design metrics. Its objective function is defined as:

$$Quality = \frac{\sum_{i=1}^{6} QA_i(S)}{6} \quad (5.16)$$

Where QA_i is the quality attribute number i being calculated based on the returned structural metrics from the system S.

Since it may not be sufficient to consider structural metrics, we used the design coherence measures of our previous work [25] to ensure that every refactoring solution preserves the semantics of the design.

5.3.2.4 Interactive Recommendation of Refactorings

The first step of the interactive step will be executed as described in Algorithm 5.1, to investigate if there are some common principles among many of the generated non-dominated refactoring solutions. The algorithm check if the optimal refactoring solutions have some common features such as identical refactoring operations among most or all of the solutions and a specific common order/sequence in which to apply the refactorings. Such information will be used to rank the suggested refactorings for developers using the following formula:

$$Rank(R_{x,y}) = \frac{\sum_{j=0}^{n} \sum_{i=0}^{size(S_j)} [R_{i,j} = R_{x,y}]}{MAX(\sum_{j=0}^{n} \sum_{i=0}^{size(S_j)} [R_{i,j} = R_{x,y}])} \in [0..1] \quad (5.17)$$

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

Once the Pareto front solutions are ranked, the second step of the interactive step will be executed as described in Algorithm 5.2. The refactorings of the best solution, in terms of ranking, are recommended to the developer based on their order in the vector. Then, the ranking score of the solutions will be updated automatically after every feedback (interaction) with the

developer. Our interactive algorithm proposes three levels of interactions as described in Figure 5.17. The developer can check the ranked list of refactorings and then *apply*, *modify* or *reject* the refactoring. If the developers prefer to modify the refactoring, then our algorithm can help them during the modification process as described in Figure 5.18. In fact, our tool proposes to the developer a set of recommendations to modify the refactoring based on the history of changes applied in the past and the semantic similarity between code elements (classes, methods, etc.). For example, if the developer wants to modify a move method refactoring them, having specified, the source method to move, our interactive algorithm automatically suggests a list of possible target classes ranked based on the history of changes and semantic similarity. This is an interesting feature since developers often know which method to move, but find it hard to determine a suitable target class. The same observation is valid for the remaining refactoring types. Another action that the developers can select is to reject/delete a refactoring from the list. After every action selected by the developer, the ranking is updated based on the feedback using the following formula:

$$Rank(S_i) = \sum_{k=1}^{size(S_i)} Rank(R_{k,i}) + (RO \cap Applied\ \text{Re}\ factoringsList) - (RO \cap \text{Re}\ jected\ \text{Re}\ factoringsList) + 0.5 * (RO \cap Modified\ \text{Re}\ factoringsList) \quad (5.18)$$

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

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

refactorings on the version after manual edits. Thus, the ranking of the solutions will change after every interaction. In case that many refactorings are rejected, the interactive NSGA-II algorithm will continue to execute while taking into consideration all the feedback from developers as constraints to satisfy during the search. The rejected refactorings should not be considered as part of the newly generated solutions and the new system after refactoring will be considered in the input of the next iterations of the interactive NSGA-II.

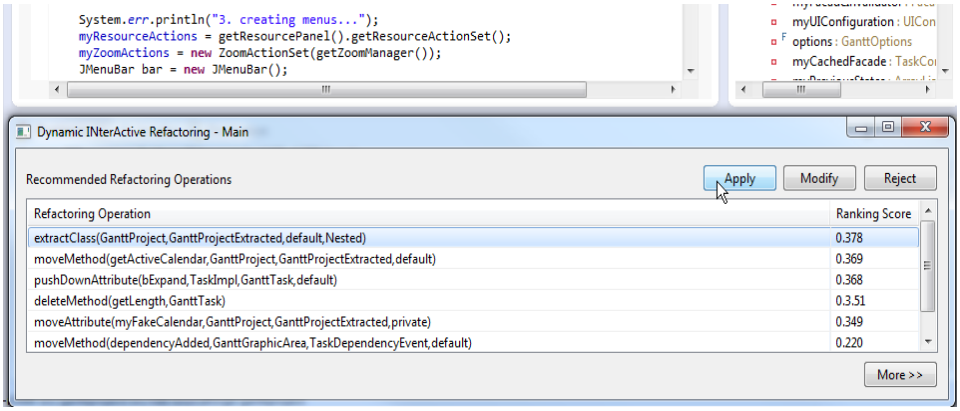


Figure 5.17. Recommended refactorings by DINAR

Fig. 3. Refactorings recommended by our technique.

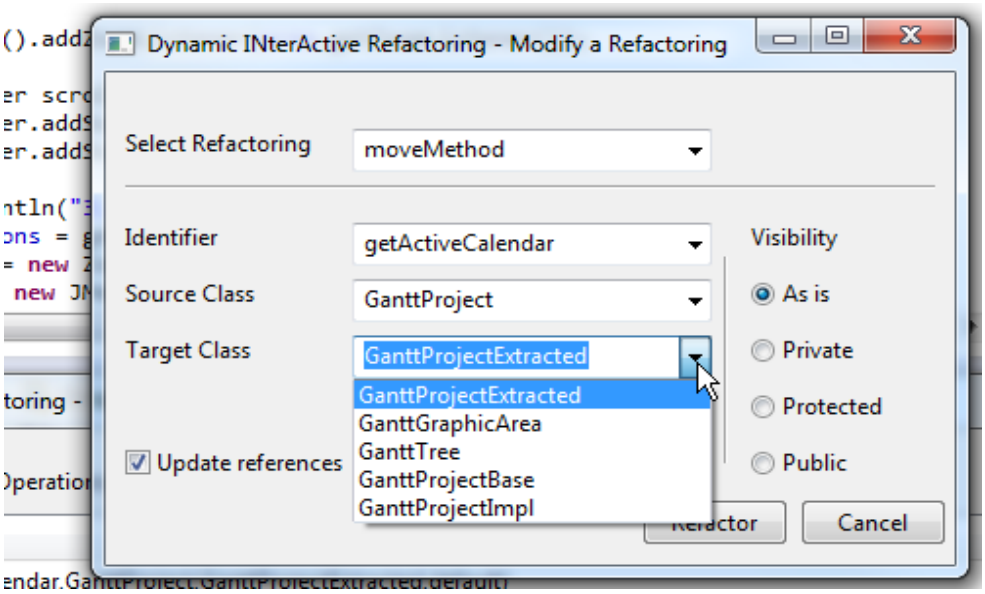


Figure 5.18. Recommended target classes by our technique for a move method refactoring to modify.

In the non-interactive refactoring systems, the set of refactorings, suggested by the best-chosen solution, needs to be fully executed in order to reach the solution's promised results. Thus, any changes applied to the set of refactorings such as changing or skipping some of them could deteriorate the resulting system's quality. In this context, the goal of this work is to cope with the above-mentioned limitation by granting to the developer's the possibility to customize the set of suggested refactorings either by accepting, modifying or rejecting them. The novelty of this work is the approach's ability to take into account the developer's interaction, in terms of introduced customization to the existing solution, by conducting a local search to locate a new solution in the Pareto Front that is nearest to the newly introduced changes. We believe that our approach may narrow the gap that exists between automated refactoring techniques and human intensive development. It allows the developer to select the refactorings that best matches his/her coding preferences while modifying the source code to update existing features.

To illustrate our interactive algorithm, we consider the refactoring of JVacation v1.0, a Java open-source trip management and scheduling software [153]. After generating the upfront list of best refactoring solutions that represents a good trade-off between improving quality while minimizing the number of refactorings, we extracted three solutions from the Pareto front, for the purpose of simplicity, and we considered a fragment of each solution, Figure 5 describes each solution refactorings along with its rank after the execution of the first step of the interactive algorithm. The solutions are ranked based on Equation 5.18 to identify the most common refactorings between the non-dominated solutions. This is achieved by counting the number of occurrences of operation within the Pareto front solution set, this number will be averaged by the maximum number of occurrences found.

Operation	Source/entity	Target entity	Operation rank
Solution1 fitness scores before normalization (0.198, 3)			
Move Method	ctrl.booking.BookingController::handleLodgingViewEvent(java.awt.event.ActionEvent):void	ctrl.booking.LodgingModel	0.4165
Extract Class	ctrl.booking.SelectionModel:: - flightList + addFlight():void +clearFlight():void	ctrl.booking.FlightList	0.2915
Move Method	gui.panels.maintenance.mLodgingsPanel::getStart():java.util.Date	gui.components.DateEdit	0.0525
Solution1 Rank			**0.7605**
Solution1 fitness scores before normalization (0.202, 4)			
Move Method	ctrl.booking.BookingController::handleLodgingViewEvent(java.awt.event.ActionEvent):void	ctrl.booking.lodgingList	0.4025
Move Method	gui.panels.maintenance.mLodgingsPanel::getStart():java.util.Date	gui.components.LabelCombo	0.4025
Inline Class	ctrl.ModelChangeEvent	ctrl.CoreModel	0.0555
Extract Class	ctrl.booking.SelectionModel:: - travelerList + addTraveler():void +clearTraveler():void	ctrl.booking.TravelerList	0.4165
Solution2 Rank			**1.2770**
Solution3 fitness scores before normalization (0.209, 5)			
Extract Class	ctrl.booking.SelectionModel:: - flightList + addFlight():void +clearFlight():void	ctrl.booking.FlightList	0.2915
Move Method	ctrl.booking.BookingController::handleLodgingViewEvent(java.awt.event.ActionEvent):void	ctrl.booking.lodgingList	0.4025
Extract Class	ctrl.booking.SelectionModel:: - travelerList + addTraveler():void +clearTraveler():void	ctrl.booking.TravelerList	0.4165
Inline Class	ctrl.ModelChangeEvent	ctrl.CoreView	0.0555
Move Method	ctrl.booking.BookingController::createBookings():void	ctrl.CoreModel	0.0025
Inline Class	ctrl.ModelChangeEvent	ctrl.Core	0.0025
Solution3 Rank			**1.1710**

Figure 5.19. Three simplified refactoring solutions recommended for JVacation v1.0.

The solution with the highest rank will be selected for execution, its related operations will be shown to the user based on their order in the vector. Figure 5.20 summarizes the various interactions between the developer and the suggested refactorings. The first recommended refactoring of the top ranked solution (Solution 2) was applied by the developer thus the ranking score was increased by 1 for both solutions 2 and 3 since they include this refactoring in their solutions. In the second interaction, the recommended refactoring was partially modified by the programmer thus the ranking score of the second solution was increased by 0.5 for solution 2 but by 1 for solution 1 since the applied operation exists in that solution. In the third interaction, the recommended refactoring was rejected thus the score of the top solution number 2 was decreased by 1.

Operation	R1:MoveMethod(ctrl.booking.BookingController::handleLodgingViewEvent:void, ctrl.booking.LodgingModel)		
Decision	Applied		
Changes	AppliedRefactoringsList = {R1}	RejectedRefactoringsList = {}	
SolutionSet	Solution1	Solution2 *	Solution3
Initial rank	0.7605	1.2770	1.1710
Interation1	0.7605	**2.2770 (+1)**	**2.1710 (+1)**

Operation	R2:MoveMethod(gui.panels.maintenance.mLodgingsPanel::getStart():java.util.Date, gui.components.LabelCombo)		
Decision	Modified to: R2: MoveMethod(gui.panels.maintenance.mLodgingsPanel::getStart():java.util.Date, gui.components.DateEdit)		
Changes	AppliedRefactoringsList = {R1, R2}	RejectedRefactoringsList = {}	
	Solution1	Solution2 *	Solution3
Initial rank	0.7605	1.2770	1.1710
Interation1	0.7605	2.2770 (+1)	2.1710 (+1)
Interation2	**1.7605 (+1)**	**2.7770 (+0.5)**	2.1710

Operation	R3: InlineClass (ctrl.ModelChangeEvent, ctrl.CoreModel)		
Decision	Rejected		
Changes	AppliedRefactoringsList = {R1, R2}	RejectedRefactoringsList = {R3}	
	Solution1	Solution2 *	Solution3
Initial rank	0.7605	1.2770	1.1710
Interation1	0.7605	2.2770 (+1)	2.1710 (+1)
Interation2	1.7605 (+1)	2.7770 (+0.5)	2.1710
Interation3	1.7605	**1.7770 (-1)**	2.1710

Operation	R4: ExtractClass (ctrl.booking.SelectionModel:: - flightList + addFlight():void +clearFlight():void, ctrl.booking.FlightList)		
Decision	Applied		
Changes	AppliedRefactoringsList = {R1, R2, R4}	RejectedRefactoringsList = {R3}	
	Solution1	Solution2	Solution3 *
Initial rank	0.7605	1.2770	1.1710
Interation1	0.7605	2.2770 (+1)	2.1710 (+1)
Interation2	1.7605 (+1)	2.7770 (+0.5)	2.1710
Interation3	1.7605	1.7770 (-1)	2.1710
Interation4	**2.7605 (+1)**	1.7770	**3.1710 (+1)**

Figure 5.20. Four different interaction examples with the developer applied on the refactoring solutions recommended for JVacation v1.0.

The algorithm will stop recommending new refactorings based on the request of the developer or when the system achieve acceptable quality improvements in terms of number of design defects and quality metrics. These parameters can be specified by the developer or the team manager.

5.3.3 Validation

In order to evaluate the ability of our refactoring framework to generate good refactoring recommendations, we conducted a set of experiments based on nine open source systems and

two industrial projects provided by the IT department at the Ford Motor Company. Each experiment is repeated 30 times, and the obtained results are subsequently statistically analyzed with the aim to compare our proposal with a variety of existing approaches [6] [70] [71] [86] [106]. In this section, we first present our research questions and then describe and discuss the obtained results.

5.3.3.1 Research Questions and Evaluation Metrics

We defined four research questions that address the applicability, performance in comparison to existing refactoring approaches, and the usefulness of our interactive multi-objective refactoring approach. The four research questions are as follows:

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

RQ2: To what extent can our approach efficiently rank the recommended refactorings?

RQ3: How does our interactive formulation perform compared to fully-automated refactoring techniques?

RQ4: Can our approach be useful for developers during the development of software systems?

To answer all these research questions, we considered the refactoring solutions recommended by our approach after interactions with the developers as described in the previous section. To answer RQ1, it is important to validate the proposed refactoring solutions from both quantitative and qualitative perspectives. For the quantitative validation, we asked a group of developers to analyze and apply manually several refactoring types using Eclipse on several code fragments extracted from different systems where most of them correspond to code smells identified in previous studies as worth removing by refactoring [71] [86]. Then, we calculated precision and recall scores to compare between refactorings recommended by our approach after interaction with developers and those suggested manually:

$$\mathrm{RC}_{recall} = \frac{\text{suggested operations} \cap \text{expected operations}}{\text{expected operations}} \in [0,1] \tag{5.19}$$

$$\mathrm{PR}_{precision} = \frac{\text{suggested operations} \cap \text{expected operations}}{\text{suggested operations}} \in [0,1] \tag{5.20}$$

Another metric that we considered for the quantitative evaluation is the percentage of fixed code smells (*NF*) by the refactoring solution. The detection of code smells after applying a refactoring solution is performed using the detection rules of [71]. Formally, *NF* is defined as:

$$NF = \frac{\#\text{fixed code smells}}{\#\text{code smells}} \in [0,1] \qquad (5.20)$$

The detection of code smells is very subjective and some developers prefer not to fix some smells because the code is stable or some of them are not important to fix. To this end, we considered another metric *G* based on QMOOD that estimates the quality improvement of the system by comparing the quality before and after refactoring independently from the number of fixed design defects. Four main different quality factors are considered by QMOOD: reusability, flexibility, understandability and effectiveness. All of them are formalized using a set of quality metrics. Hence, the total gain in quality *G* for each of the considered QMOOD quality attributes q_i before and after refactoring can be easily estimated as:

$$G = \frac{\sum_{i=1}^{4} G_{q_i}}{4} \text{ and } G_{q_i} = q'_i - q_i \qquad (5.21)$$

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

For the qualitative validation, we asked groups of potential users of our tool to evaluate, manually, whether the suggested refactorings are feasible and efficient at improving the software quality and achieving their maintainability objectives. We define the metric Manual Correctness (*MC*) to mean the number of meaningful refactorings divided by the total number of suggested refactorings. The *MC* metric is computed after the user interaction is completed. In fact, the number of correct refactorings includes the number of refactorings applied by developers when using our tool since they can either apply, modify or reject a refactoring recommendation. *MC* is given by the following equation:

$$MC = \frac{\#\,\text{coherent applied refactorings}}{\#\,\text{proposed refactorings}} \tag{5.22}$$

To avoid that the computation of the *MC* metric will be biased by the developers' feedback, we asked the developers who did not participate in the experiments to use our tool to manually evaluate the correctness of the recommended refactorings.

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

$$NAR = \frac{\#\,\text{accepted refactorings}}{\#\,\text{recommended refactorings}} \in [0,1] \tag{5.23}$$

$$NRR = \frac{\#\,\text{rejected refactorings}}{\#\,\text{recommended refactorings}} \in [0,1] \tag{5.24}$$

$$NMR = \frac{\#\,\text{modified refactorings}}{\#\,\text{recommended refactorings}} \in [0,1] \tag{5.25}$$

To answer RQ2, we evaluated the relevance of the recommended refactorings in the top *k* where *k* =1, 5, 10 and 15 using the following metrics *PR@k, MC@k, NAR@k, NRR@k and NMR@k*.

To answer RQ3, we compared our approach to four other existing fully-automated search-based refactoring techniques: Kessentini et al. [71], O'Keeffe and Ó Cinnéide [70], Ouni et al. [86] and Harman et al. [6] that consider the refactoring suggestion task only from the quality improvement perspective. Kessentini et al. [71] formulate software refactoring as a mono-objective search problem where the main goal is to fix design defects and improve quality metrics. O'Keeffe and Ó Cinnéide [70] also proposed a mono-objective formulation to automate the refactoring process by optimizing a set of quality metrics. Ouni et al. [86] and Harman et al. [6] proposed a multi-objective refactoring formulation that generates solutions to fix code smells.

Both techniques are non-interactive and fully-automatic. We considered in our experiments another popular design defects detection and correction tool JDeodorant [106] that does not use heuristic search techniques. The current version of JDeodorant is implemented as an Eclipse plug-in that identifies some types of design defects using quality metrics and then proposes a list of refactoring strategies to fix them. Since JDeodorant just recommends a few types of refactoring with respect to the ones considered by our tool. We restricted, in this case, the comparison to the same refactoring types supported by JDeodorant. All these existing techniques are fully-automated and do not provide any interaction with the developers to update their solutions. We used the metrics *MC*, *RC*, *PR*, *NF* and *G* to perform the comparisons.

To answer RQ4, we used a post-study questionnaire that collects the opinions of developers on our tool. We also wished to assess how the refactoring actually increases the software quality and productivity in that the effort to add new features or fixing bugs should reduce after performing the refactorings. To this end, we asked 22 software engineers, including 7 developers from our industrial partner, to add five new features and fix a set of ten bugs. We divided them into two groups. To avoid that the achieved results might be due to the different levels of ability of the two groups, we adapted a counter-balanced design where each subject performed two tasks, one on the original system and one on the refactored system. The details of these scenarios will be described later. To estimate the impact of the suggested refactorings on the productivity of developers, we defined the following metric *TP* to measure the time required to perform the same activities on the system with and without refactoring:

$$TP_i = \frac{\#\text{ minutes required to perform task i on the system after refactoring}}{\#\text{ minutes required to perform task i on the system before refactoring}} \qquad (5.26)$$

5.3.3.2 Software Projects Studied

We used a set of well-known open-source Java projects and 2 systems from our industrial partner, the Ford Motor Company. We selected these 10 systems for our validation because they range from medium to large-sized open-source projects, which have been actively developed over the past 10 years, and their design has not been responsible for a slowdown of their developments. Table 1 provides some descriptive statistics about these programs.

Table 5.17. Statistics of the studied systems.

Systems	Release	#classes	KLOC	#Code smells	#Refactorings
Xerces-J	v2.7.0	991	240	91	83
JHotDraw	v6.1	585	21	25	49
JFreeChart	v1.0.9	521	170	72	88
GanttProject	v1.10.2	245	41	49	56
Apache Ant	v1.8.2	1191	255	112	103
Rhino	v1.7R1	305	42	69	59
Log4J	v1.2.1	189	31	64	71
Nutch	v1.1	207	39	72	84
JDI-Ford	v5.8	638	247	83	94
MROI-Ford	V6.4	786	264	97	119

5.3.3.3 Scenarios

The first group of two developers added these features and fixed the bugs in the system before refactoring. The second group includes the remaining two developers and performed the same activities on the system after refactoring.

Our study involved 14 subjects from the University of Michigan and 8 software engineers from the Ford Motor Company. Subjects include 6 master students in Software Engineering, 8 Ph.D. students in Software Engineering and 8 software developers. All the subjects are volunteers and familiar with Java development and refactoring. The experience of these subjects on Java programming ranged from 2 to 19 years.

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

As described in Table 5.18, we formed 3 groups. Each of the first two groups (A and B) is composed of 3 master students and 4 Ph.D. students. The third group is composed of the 8 software developers from the Ford Motor company since they accept to participate only in the

evaluation of their two software systems. Table 5.18 summarizes the survey organization including the list of systems and algorithms evaluated by the group. 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. We divided the subjects into groups according to the studied systems, the techniques to be tested and developers' experience. Consequently, each group of subjects 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 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 subject evaluates different refactoring solutions for the different techniques and systems.

We executed five different scenarios. In the first scenario, we selected a total of 90 classes from all the systems that include design defects (9 classes to fix per system). Then, we asked every participant to manually apply refactorings to improve the quality of the systems by fixing an average of two of these defects. As an outcome of the 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 different developers to manually evaluate the last recommended solution by our algorithm after the interaction with the user. We did a cross-validation between the groups to avoid that the computation of the *MC* metric will be biased by the developers' feedback thus we asked the developers who did not participate in the experiments to use our tool to manually evaluate the correctness of the recommended refactorings. In the third scenario, we asked the participants to use our tool during a period of two hours on the different systems and then we collected their opinions based on a post-study questionnaire that will be detailed later. In the fourth scenario, we collected a set of 6 bugs per system from the bug reports of the studied release for every project and asked the groups to fix them based on the refactored and non-refactored version. To avoid that the achieved results might be due to the different levels of ability of the two groups, we adapted a counter-balanced design where we asked every group to fix 2 bugs on the version before refactoring and then 2 other bugs on the version after refactoring. We selected the bugs that require almost the same effort to be fixed in terms of number of changes with an average of 15 changes. In the last

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

The participants were asked to justify their evaluation of the solutions and these justifications are reviewed by the organizers of the study (one faculty member, one postdoc, one Ph.D. student and one master student). In addition, our experiments are not only limited to the manual validation but also the automatic validation can verify the effectiveness of our approach. Subjects were aware that they are going to goals of the experiments, but do not know the particular experiment research questions and the used algorithms.

Table 5.18. Survey organization.

Subject groups	Systems	Algorithms / Approaches
Group A	Xerces-J	Interactive NSGA-II, Kessentini et al. [71], O'Keeffe and Ó Cinnéide [70], Ouni et al [86], Harman et al. [6], JDeodorant [106]
	JHotDraw	
	JFreeChart	
	GanttProject	
Group B	Apache Ant	Interactive NSGA-II, Kessentini et al. [71], O'Keeffe and Ó Cinnéide [70], Ouni et al. [86], Harman et al. [6], JDeodorant [106]
	Rhino	
	Log4J	
	Nutch	
Group C	JDI-Ford	Interactive NSGA-II, O'Keeffe and Ó Cinnéide [70], Ouni et al. [86] JDeodorant [106]
	MROI-Ford	

5.3.3.4 Experimental setting

The parameter setting influences significantly the performance of a search algorithm on a particular problem [95]. For this reason, for each algorithm and for each system, we perform a set of experiments using several population sizes: 50, 100, 200, 300 and 500. The stopping criterion was set to 100,000 evaluations for all algorithms in order to ensure fairness of comparison. The other parameters' values were fixed by trial and error and are as follows: (1) crossover probability = 0.8; mutation probability = 0.5 where the probability of gene

modification is 0.3; stopping criterion = 100,000 evaluations. Each algorithm is executed 30 times with each configuration and then the comparison between the configurations is done using the Wilcoxon test. In order to have significant results, for each couple (algorithm, system), we use the trial and error method in order to obtain a good parameter configuration. Regarding the evaluation of fixed code smells, we focus on the eight following code smell types [15]: Blob, Spaghetti Code (SC), Functional Decomposition (FD), Feature Envy (FE), Data Class (DC), Lazy Class (LC), Long Parameter List (LPL), and Shotgun Surgery (SS). We choose these code smell types in our experiments because they are the most frequent and hard to fix based on several studies [15] [55].

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

5.3.3.5 Statistical test methods

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 31 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 (α = 5%). The latter tests the null hypothesis, H_0, that the obtained results of two algorithms are samples from continuous distributions with equal medians, against the alternative that they are not, H_1. The *p*-value of the Wilcoxon test corresponds to the probability of rejecting the null hypothesis H_0 while it is true (type I error). A *p*-value that is less than or equal to α (≤ 0.05) means that we accept H_1 and we reject H_0. 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 Kessentini et al. [71], Ouni et al. [86], Harman et al. [6] and JDeodorant [106] results with DINAR ones. In this way, we determine whether the performance difference between DINAR

and one of the other approaches is statistically significant or just a random result. The results presented were found to be statistically significant on 31 independent runs using the Wilcoxon rank sum test with a 99% confidence level ($\alpha < 1\%$).

The Wilcoxon rank sum test allows verifying whether the results are statistically different or not. However, it does not give any idea about the difference in magnitude. To this end, we used the Vargha and Delaney's A statistics [121] which is a non-parametric effect size measure. In our context, given the different performance metrics (such as *PR, RC, MC*, etc.), the *A* statistics measures the probability that running an algorithm *B1* (interactive NSGA-II) yields better performance than running another algorithm *B2* (such as Kessentini et al. [71], Ouni et al. [86], etc.). If the two algorithms are equivalent, then $A = 0.5$. In our experiments, we have found the following results: a) On small and medium scale software projects (GanttProject, Rhino, Log4J and Nutch) interactive NSGA-II is better than all the other algorithms based on all the performance metrics with an A effect size higher than 0.94; and b) On large scale software projects (JDI-Ford, MROI-Ford, Apache Ant, Xerces-J, JHotDraw and JFreeChart), interactive NSGA-II is better than all the other algorithms with an *A* effect size higher than 0.87.

5.3.3.6 Results and Discussions

Results for RQ1: We reported the results of our empirical qualitative evaluation in Figure 5.21 (*MC*). As reported in Figure 5.21, the majority of the refactoring solutions recommended by our interactive approach were correct and approved by developers. On average, for all of our ten studied projects, 87% of the proposed refactoring operations are considered as semantically feasible, improve the quality and are found to be useful by the software engineers of our experiments. The highest MC score is 93% for the Gantt project and the lowest score is 86% for JFreeChart. Thus, it is clear that the results are independent of the size of the systems and the number of recommended refactorings. Most of the refactorings that were not manually approved by the developers are either violating few pre- or post-conditions or introducing a design incoherence.

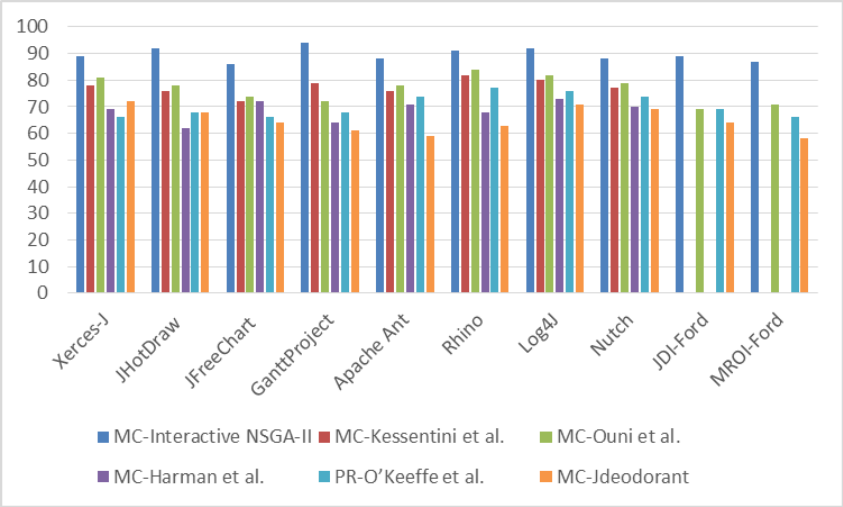


Figure 5.21. Median manual correctness (MC) value over 31 runs on all the five systems using the different refactoring techniques with a 99% confidence level (α < 1%).

Since the *MC* metric just evaluates 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 ten systems. Most of these classes represent some severe code smells detected using our previous work [71]. Figure 5.22 and Figure 5.23 summarize our findings. We found that a considerable number of proposed refactorings, with an average of more than 82% in terms of precision and recall, are 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 is was confirmed by the qualitative evaluation (*MC*). In addition, we found that the slight deviation with the expected refactorings are not related to incorrect operations but to the fact that the developers were interested mainly to fix the severest code smells or improving the quality of the code fragments that they frequently modify.

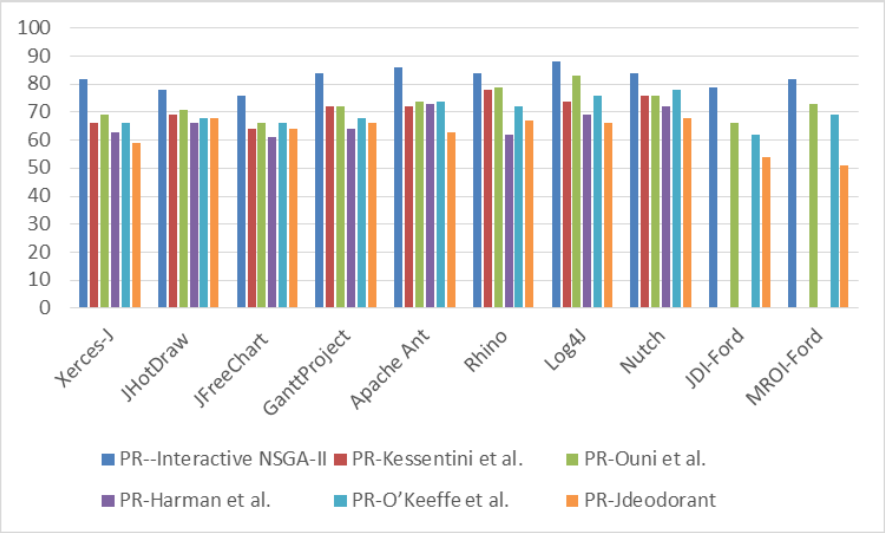


Figure 5.22. Median precision (PR) value over 31 runs on all the five systems using the different refactoring techniques with a 99% confidence level (α < 1%).

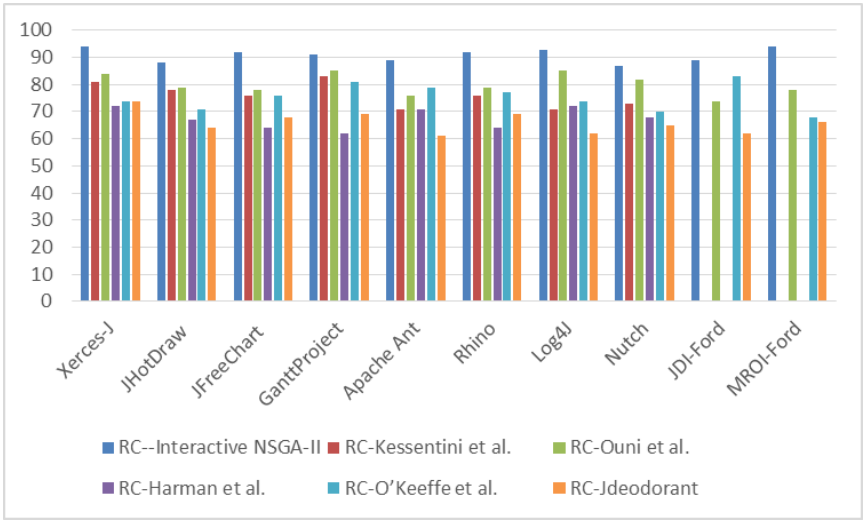


Figure 5.23. Median recall (RC) value over 31 runs on all the five systems using the different refactoring techniques with a 99% confidence level (α < 1%).

We evaluated also the ability of our approach to fixing several types of code smell and to improve the quality as described in Figure 5.24 and Figure 5.25. Figure 5.24 depicts the percentage of fixed code smells (*NF*). It is higher than 82% on all the ten systems, which is an acceptable score since developers may reject or modify some refactorings that fix some code smells because they do not consider them very important (their goal is not to fix all code smells in the system) or the current version of the code becomes stable. Some systems, such as Rhino and Gantt, have a higher percentage of fixed code smells with an average of more than 88%.

This can be explained by the fact that these systems include a higher number of code smells than others. Figure 5.25 shows that the refactorings recommended by the approach and applied by developers improved the quality metrics value (*G*) of the ten systems. For example, the average quality gain for the two industrial systems was the highest among the ten systems with more than 0.3. The improvements in the quality gain confirm that the recommended refactorings helped to optimize different quality metrics.

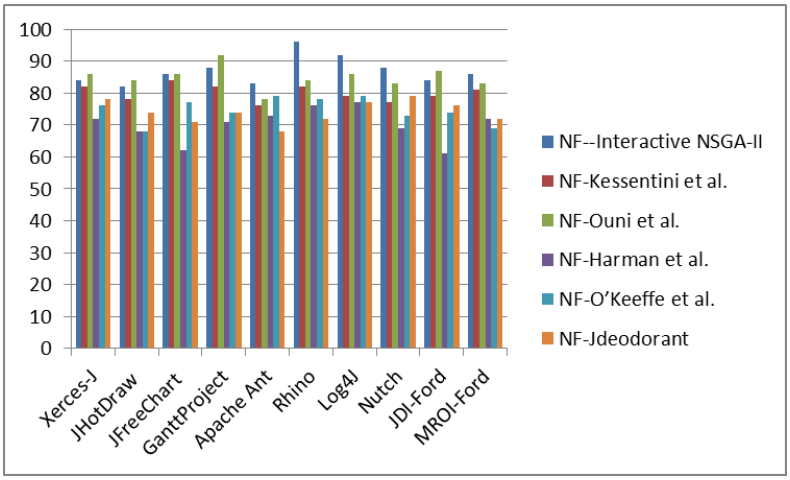


Figure 5.24. Median percentage of fixed code smells (NF) value over 31 runs on all the five systems using the different refactoring techniques with a 99% confidence level (α < 1%).

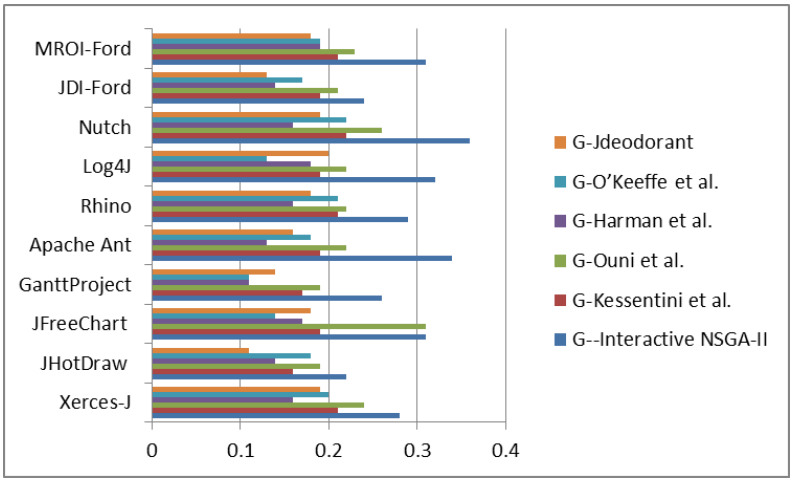


Figure 5.25. Median quality gain (G) value over 31 runs on all the five systems using the different refactoring techniques with a 99% confidence level (α < 1%).

To summarize and answer RQ1, the experimentation results confirm that our interactive approach helps the subjects to refactor their systems efficiently by finding the relevant refactorings and improve the quality of all the ten systems.

Results *for RQ2:* We evaluated the ability of our approach to help software engineers in finding quickly good refactorings based on an efficient ranking of the proposed operations. We evaluated the manual correctness and the precision of the recommended refactoring at the top *k* of the list at the different interactions where *k* was varied between 1, 5, 10 and 15 as described in Figure 5.26 and Figure 5.27. Figure 5.26 shows that the lowest *MC@1* is 93% and the highest is 100% on the different ten systems confirming that the top1 refactoring was almost always correct and relevant for the developers. The *MC@15* presents the lowest results which are expected since we evaluated the manual correctness of the top15 recommended refactorings at several interactions which increase the probability to contain few irrelevant refactorings. However, the average *MC@15* still could be considered acceptable with an average of more than 81%. The same observations are also valid for the *PR@k* however the results are a bit lower than *MC@k*. The average *PR@k* results were respectively 94%, 89%, 84% and 80% for *k* = 1, 5, 10 and 15. Thus, it is clear that the ranking function using by our interactive approach to exploring the Pareto front is efficient.

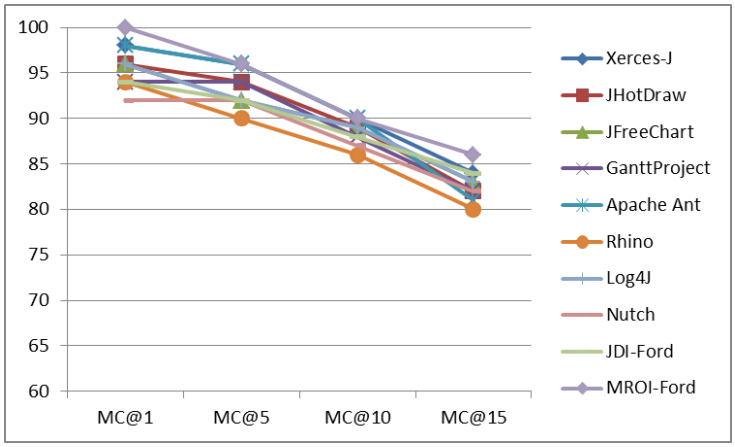


Figure 5.26. Median of manual correctness (MC) of the recommended refactoring at the top k = 1, 5, 10 and 15 with a 99% confidence level (α < 1%).

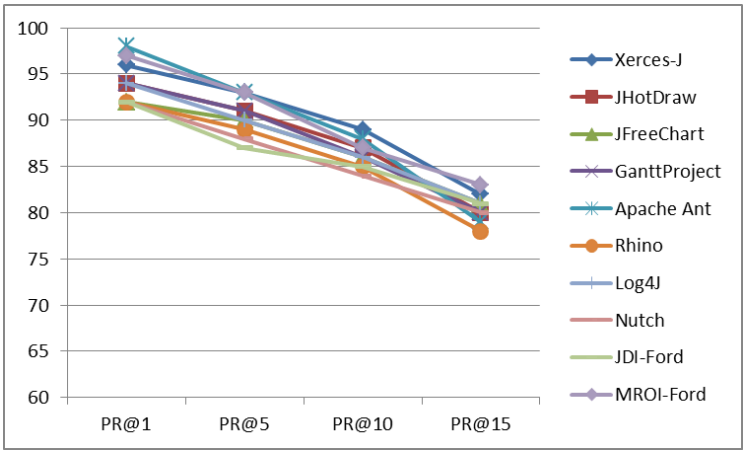


Figure 5.27. Median of precision (PR) of the recommended refactoring at the top k = 1, 5, 10 and 15 with a 99% confidence level (α < 1%).

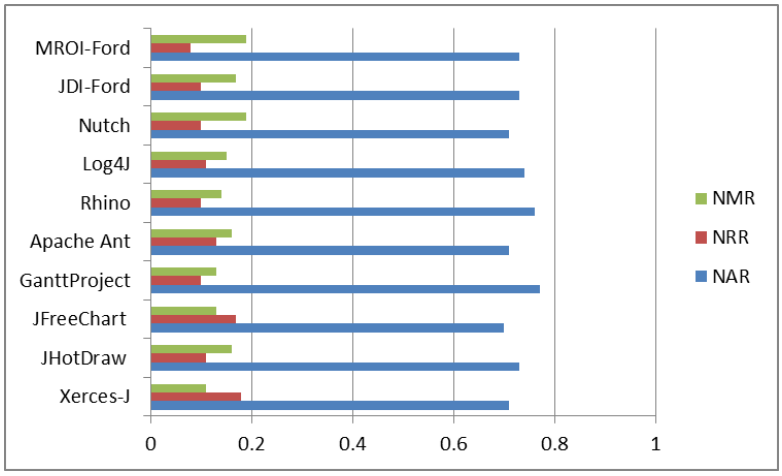


Figure 5.28. Median percentage of accepted refactorings (NAR), percentage of modified refactorings (NMR) and percentage of rejected refactorings (NRR) values over 31 runs on all the five systems with a 99% confidence level (α < 1%).

We have also considered three other metrics *NAR (percentage of accepted refactorings)*, *NMR (percentage of modified refactorings)* and *NRR* (*percentage of rejected refactorings*) to evaluate the efficiency of our interactive approach to rank the refactorings. We collected this data using a feature that we implemented in our tool to record all the actions performed by the developers during the refactoring sessions. Figure 5.28 shows that, in average, more than 71% of the recommended refactorings were applied by the developers. In addition, an average of 17% of the recommended refactorings were modified by the developers. The users reject lower than 12% of suggested refactorings. Thus, it is clear that our recommendation tool successfully suggested a good set of refactorings to apply.

To conclude, our way to rank the recommended refactorings helped software engineers to quickly find good refactorings recommendations (answer to RQ2).

Results for RQ3: Figures from Figure 5.21 to Figure 5.25 confirm the superior performance of DINAR compared to both fully automated and manual refactoring techniques. Figure 5 shows that DINAR provides significantly higher manual correctness results (*MC*) than all other approaches having *MC* scores respectively between 50% and 75%, on average as *MC* scores on the different systems. The same observation is valid for the precision, recall and quality gain as described in, respectively, Figure 5.22, Figure 5.23 and Figure 5.25. However, the percentage of fixed code smells (*NF*) is slightly lower than Kessentini et al. [71] and Ouni et al. [86] as showed in Figure 5.24. This is can be explained by the fact that the main goal of developers is not to fix the maximum number the code smells detected in the system (which was the goal of Kessentini et al. [71] and Ouni et al. [86]) thus they rejected or modified some refactorings suggested by DINAR. In addition, DINAR is based on a multi-objective algorithm to find a trade-off between fixing code smells, improving the quality and preserving the semantics. Thus, the slight loss of *NF* is justified by a better improvement of the quality as described in Figure 5.25. Figure 5.29 shows that DINAR can help developers to find suitable refactorings quicker than existing search-based refactoring approaches and manual refactorings but it was not the case for JDeodorant. This can be explained by the fact that JDeodorant is not using heuristic search but just proposing a template to fix certain types of code smell and does not suggest high-quality refactorings as described previously (lowest quality scores). However, the time required to use DINAR is still comparable to JDeodorant but provides more effective refactoring solutions.

Overall the superior performance of DINAR can be explained by several factors. First, Kessentini et al. [71] and Harman et al. [6] use only structural indications (quality metrics) to evaluate the refactoring solutions thus a high number of refactorings are not feasible semantically. Thus, our approach reduces the number of semantic incoherencies when suggesting operations. Second, the innovization component improved the efficiency of suggested refactoring solutions by DINAR compared to NSGA-II where the developers need to select one solution from the Pareto front that cannot be updated dynamically. Third, JDeodorant proposes some pre-

defined patterns to fix some types of code smells that cannot be sometimes generalized. Finally, manual refactoring is an error-prone process that is also time-consuming.

In our qualitative evaluation, we considered how the refactoring actually increases the software quality and productivity in that the effort to add new features or to fix bugs should reduce after performing the refactorings on the JDI-Ford system. To this end, we created six scenarios as described in Figure 5.29 and we asked four software engineers from our industrial partner, with almost the same experience, to add four new features and fix a set of eight bugs (three scenarios to add and modify existing features, and the remaining ones to fix a set of bugs). Then, we compared the time required by the two groups of developers to finalize the same tasks on the JDI-Ford system before and after refactoring using the metric *TP*. Figure 5.29 shows that the time is reduced on average by 25% to finalize the tasks when the refactored system is used. In some scenarios, the time is reduced by more than 45% since the required tasks are time-consuming (involving the implementation of new features).

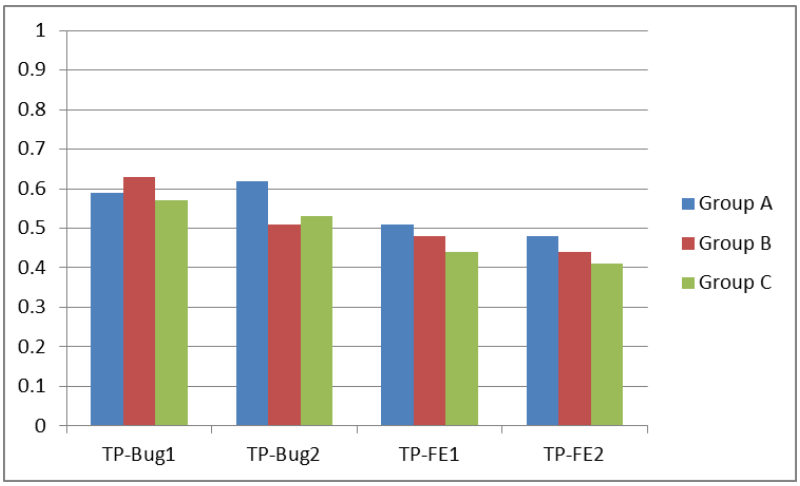


Figure 5.29. Productivity difference (T) value on six different tasks performed on JDI-Ford system.

Results for RQ4: To answer RQ4, we asked the subjects to take a post-study questionnaire after completing the refactoring tasks using DINAR and all the techniques considered in our experiments. The post-study questionnaires collected the opinions of the participants about their experience in using DINAR compared to manual and fully-automated refactoring 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:

- The interactive dynamic refactoring recommendations are a desirable feature in Integrated Development Environments (IDEs).
- The interactive manner of recommending refactorings by DINAR is a useful and flexible way to refactor systems compared to fully-automated or manual refactorings.

The agreement of the participants was 4.7 and 4.4 for the first and second statements respectively. This confirms the usefulness of DINAR for the software developers considered in our experiments.

The remaining questions of the post-study questionnaire were about the benefits and also limitations (possible improvements) of DINAR. We summarize in the following the feedback of the developers. Most of the participants mention that DINAR is faster than manual refactoring since they spent a long time with manual refactoring to find the locations where refactorings should be applied. For example, developers spend time when they decide to extract class to find the methods to move to the new created class or when they want to move a method then it takes time to find the best target class by manual exploration of the source code. Thus, the developers liked the functionality of DINAR that helps them to modify a refactoring and finding quickly the right parameters based on the recommendations. Furthermore, refactorings may affect several locations in the source code, which is a time-consuming task to perform manually, but they can perform it instantly using DINAR.

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

functionalities. c) DINAR allows developers to access all the functionality of the IDE (e.g., Eclipse). d) the suggested refactorings by DINAR can fix code smells (root canal refactoring) or improve some quality metrics (floss canal refactoring) due to the use of the multi-objective approach. Another important feature that the participants mention is that DINAR allows them to take the advantages of using multi-objective optimization for software refactoring without the need to learn anything about optimization and exploring explicitly the Pareto front to select one "ideal" solution. The implicit exploration of the Pareto front in an interactive fashion represents an important advantage of DINAR along with the dynamic update of the recommended list of refactoring using innovization. In fact, the developers found a lot of difficulties with the multi-objective tool of Ouni et al. [86] to explore the Pareto front to find a good refactoring solution. In addition, they did not appreciate the long list of refactoring suggested by [86] since they want to take control of modifying and rejecting some refactorings. In addition, the validation of this long list of refactorings is time-consuming. Thus, they appreciate that DINAR suggests refactoring one by one and update the list based on the feedback of developers.

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

5.3.4 Conclusion

We proposed an interactive recommendation tool, called DINAR, for software refactoring that dynamically adapts and suggests refactorings to developers based on their feedback and introduced code changes. DINAR allows developers to benefit from search-based refactoring tools without explicitly invoking any knowledge about optimization and multi-objective optimization algorithms. To evaluate the effectiveness of DINAR, we conducted a human study with 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 DINAR improves the applicability of software refactoring and proposes a novel way for software developers to refactor their systems.

As part of the future work, we should validate DINAR with additional refactoring types, systems and code smell types in order to conclude about the general applicability of our methodology. We will also compare DINAR with other refactoring techniques. Furthermore, in this work, we only focused on the recommendation of refactorings. We are planning to extend the approach by automating the test and verification of applied refactorings. In addition, we will consider the importance of code smells during the correction step using previous code-changes, classes-complexity, etc. Another future research direction related to our work is to adapt our interactive refactoring recommendation approach to several other software engineering problems such as software re-modularization, change detection and the next release problem.

Chapter 6: Conclusion and Future Work

6.1 Threats to Validity

We explore in this section the factors that can bias our empirical study. These factors can be classified into three categories: construct, internal and external validity. Construct validity concerns the relation between the theory and the observation. Internal validity concerns possible bias with the results obtained by our proposal. Finally, external validity is related to the generalization of observed results outside the sample instances used in the experiment.

In our experiments, construct validity threats are related to the several quantitative measures used in our experiments. To mitigate this threat, we manually inspect and validate the remodularization solutions by a set of experts. Another threat concerns the data about the expected operations of the studied systems. In addition to the documented operations, we are using Ref-Finder which is known to be efficient. Indeed, Ref-Finder was able to detect operations with an average recall of 99% and an average precision of 79%. To ensure the precision, we manually inspect the operations found by Ref-Finder and select only those types considered in our experiments.

We take into consideration the internal threats to validity in the use of stochastic algorithms since our experimental study is performed based on 31 independent simulation runs for each problem instance and the obtained results are statistically analyzed by 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. In fact, parameter tuning of search algorithms is still an open research challenge till today. We have used the trial-and-error method which one of the most used ones [154]. However, the use of ANOVA-based technique [155] could be another

interesting direction from the viewpoint of the sensitivity to the parameter values. Another threat is related to the order of applying the objectives that my influences the outcome of the search. We are planning to investigate the impact of the order of objectives on the results by evaluating several combinations. In addition, the weights used for the semantic functions are selected manually and further experiments are required to study the impact of the variation of these weights on the quality of the results. Another threat is related to our experiments to show how the execution time is related to the size of the system in terms of number of classes. The number of classes of a system can be not enough to have a strong conclusion especially when other factors such as the density of the interdependency graph probably will influence execution time more than the number of classes in the system. In addition, it is may not be enough to show with 3 version of Eclipse if the execution time is a linear or non-linear.

We identify other three threats to internal validity: selection, learning and fatigue, and diffusion. Another internal threat is related to the problem of isomorphic solutions since inferring goodness based on an objective function can sometimes be misleading. To this end, we manually validated the solution as described in the experiments and we found correlation between the fitness function values and the success metrics (different from the fitness functions and correspond to the developers opinion after manually validating some solutions) used in the experiments. However, we cannot generalize our correlation results since it was limited to few solutions and the problem of isomorphic solutions is out of the scope of this work. In addition, it is challenging to address this problem especially for the case of many-objective optimization where a high number of solutions are generated. Thus, we plan in our future work to study the correlation through extensive empirical studies between the improvements of the fitness function values and the quality of solutions validated manually by experts on several software engineering problems. An additional internal threat is that our approach is limited to the use of static metrics analysis. However, our approach is generic thus additional objectives/metrics and inputs can be easily added to extend our algorithm. To this end, we are planning to use dynamic analysis techniques to evaluate the system after remodularization and evaluate the impact of suggested operations on the dynamic/runtime relations.

For the selection threat, the subject diversity in terms of profile and experience could affect our studies. First, all subjects were volunteers. We also mitigated the selection threat by giving written guidelines and examples of operations already evaluated with arguments and justification. Additionally, each group of subjects evaluated different operations from different systems using different techniques/algorithms.

Randomization also helps to prevent the learning and fatigue threats. For the fatigue threat, specifically, we did not limit the time to fill the questionnaire. Consequently, we sent the questionnaires to the subjects by email and gave them enough time to complete the tasks. Finally, only ten operations per system were randomly picked for the evaluation.

Diffusion threat is limited in our study because most of the subjects are geographically located in a university and a company, and the majority does not know each other. For the ones who are in the same location, they were instructed not to share information about the experience before a certain date.

To ensure the heterogeneity of subjects and their differences, we took special care to diversify them in terms of professional status, university/company affiliations, gender, and years of experience. In addition, we organized subjects into balanced groups. This has been said, we plan to test our tool with Java development companies, to draw better conclusions. Moreover, the automatic evaluation is also a way to limit the threats related to subjects as it helps to ensure that our approach is efficient and useful in practice. Indeed, we compare our suggested operations with the expected ones that are already applied to the next releases and detected using Ref-Finder.

External validity refers to the generalizability of our findings. In this study, we performed our experiments on five different systems belonging to different domains and with different sizes. However, we cannot assert that our results can be generalized to other applications, other programming languages, and to other practitioners. Future replications of this study are necessary to confirm the generalizability of our findings. The participants considered in our experiments are not the original developers of the open source systems thus some of their evaluations of the remodularization solutions could be not very accurate since there are, sometimes, good reasons for the design and implementation choices made and this can be mainly

determined by the original developers. However, this not the case for the Ford project since some of the original developers of the system participated in our experiments. In addition, the number of participants in our experiments is limited. We are planning to integrate additional original developers from these open source projects to evaluate the detected code smells as part of our future work.

6.2 Conclusion

It has been, and it is still challenging to automatically refactor software systems while taking into account several developers preferences to assess the maintenance process in a realistic software development context. The related work has emphasized the formalization of refactoring as single and multi-objective optimization problem and automated the generation of refactored systems based on several metrics that vary from the structure to the semantics etc. In this context, our first goal was to investigate how the automation can process incorporate all the developers' preferences and if so, will it generate satisfactory results. We have shown through several contributions the feasibility of considering as many preferences as possible by breaking the boundaries of the few number of objectives that the existing techniques were limited to. Although our approach has proven scalability, its industrial application has revealed practical limitations related to the dynamics of the development environment that requires more flexibility that was not handled by our approach.

The above-mentioned drawbacks were the main motivation for a further investigation of incorporating the developer in the loop of the optimization process. Our goal was to enable a more flexible and robust refactoring process that allows developers to have better control over the suggested code changes. We found that we can also benefit from the developer's feedback to prune the search space and converge towards a region of his / her interest. Therefore, our approach was able to *profile* the developer instead of optimizing several metrics that mathematically define his / her preferences.

To conclude, this thesis investigated the consideration of SE problems as many-objective and interactive problems to overcome the limitation of the existing work which is mostly approaching them from fully single or bi-objective perspective. Eventually, we have introduced a

new scalable search-based software engineering methodologies for various software engineering problems which are linked to model and code transformation. Some of the introduced methodologies presented the first real-world application of a many-objective and innovization paradigms on a domain specific software engineering problem. We formulated several challenges to test the applicability of our proposals and we compared them against related work approaches. The generated results were validated from both quantitative and qualitative perspectives.

6.3 Future Work

Although software refactoring, as a practice, has been issuing considerable improvements in software overall quality, automated refactoring techniques, provided by the literature, have not yet been able to fully evolve while estimating the amount of testing effort needed for the refactored code. Despite the aim of refactoring is to improve the software's quality, the amount of effort required to refactor large systems and test them afterward remains unpredictable. Also, some of the code changes may affect the system's intended functionality or can also introduce some faults. Also, developers interleave functional-oriented code updates with some refactoring operations, as the mean of updating the system to the new specifications. The impact of these changes has not yet been taken into account by the refactoring tools and this represents one of the main limitations of the existing refactoring techniques. Another important drawback is that all techniques assume that refactoring operations are always correctly executed, this transaction-based perspective is not always true especially if refactoring pre- and post-conditions are not well implemented, and with the lack of testing support, it may increase the vulnerability of the refactored code. This side effect of refactoring has not yet been analyzed in the literature and this is another limitation of existing techniques.

Although software testing is a research intensive area as it prevents the propagation of faults, updating or prioritizing the existing test suits is in itself challenging. As the number of possible test cases increases, their execution time after every code change tends to be very expensive and so the testing effort becomes hard to predict. This problem has been well studied in the literature and resulted in many approaches related to test cases minimization, prioritization and selection while maximizing testing coverage has been proposed. However, Minimizing the cost of testing

is not the only challenge that developers face, as the testing effort does depend on the quality of tests cases that are usually designed without foreseeing the upcoming code changes, and without coupling testing to a refactoring plan, developers will be spending much more time in following up with the refactored code to re-update test suites.

To address the previous shortcomings, we will investigate, as a future research direction, the combination of refactorings generation with test suites update as an optimization problem. Our approach aims will be to find the best sequence of refactoring that optimizes the software quality through the reduction of code smells while providing the user with an adequate test cases selection to execute based on the locations in which refactoring operations occurred. Although formulating each of refactoring and testing as optimization problems have been widely studied in Search-Based Software Engineering, no practical work has been done to provide a technique which supports such heterogeneous solution representation due to the impracticality of its evaluation under one uniform search space. We intend to address this challenge by formulating the refactoring / testing practices as a multitasking problem. The multitasking paradigm [156] evolves multiple optimization tasks at once, with each task, i.e. refactoring and testing, contributing a unique factor influencing the evolution of individuals in a composite solution space. Such formulation is referred to as multifactorial optimization.

APPENDIX A. LIST OF ALGORITHMS

APPENDIX B. LIST OF PUBLICATIONS

The full list of publications, based on this dissertation, is the following:

Journals:

1. **Mkaouer, Mohamed Wiem** and Kessentini, Marouane. Model Transformation Using Multiobjective Optimization. *Advances in Computers Journal* 92: 161-202. 2014.
2. **Mkaouer, Mohamed Wiem**, Kessentini, Marouane, Shaout, Adnan, Koligheu, Patrice, Bechikh, Slim, Deb, Kalyanmoy, and Ouni, Ali. Many-objective software remodularization using NSGA-III. *ACM Transactions on Software Engineering and Methodology Journal (TOSEM)* 24, 3, Article 17 (May 2015), 45 pages, ACM.
3. **Mkaouer, Mohamed Wiem**, Kessentini, Marouane, Bechikh, Slim, Ó Cinnéide, Mel, and Deb, Kalyanmoy. On the use of many quality attributes for software refactoring: a many-objective search-based software engineering approach. *Empirical Software Engineering Journal*: 1-43. (First online: 23 December 2015: 10.1007/s10664-015-9414-4), Springer.
4. **Mkaouer, Mohamed Wiem**, Kessentini, Marouane, Ó Cinnéide, Mel , Hayashi, Shinpei, and Deb, Kalyanmoy. A robust multi-objective approach to balance severity and importance of refactoring opportunities. *Empirical Software Engineering Journal*: 1-34. (First online: 04 March 2016: 10.1007/s10664-015-9414-4), Springer.
5. **Mkaouer, Mohamed Wiem**, Kessentini, Marouane, Ali Ouni, Ó Cinnéide, Mel and Deb, Kalyanmoy. Interactive Dynamic Search-Based Software Engineering: A Case Study on Software Refactoring. *ACM Transactions on Software Engineering and Methodology Journal (TOSEM),* Under revisions.

International Conferences:

1. **Mohamed Wiem Mkaouer**, Marouane Kessentini, Slim Bechikh, Mel Ó Cinnéide, Kalyanmoy Deb: Software refactoring under uncertainty: a robust multi-objective approach. GECCO (Companion) 2014: 187-188, ACM0
2. **Mohamed Wiem Mkaouer**, Marouane Kessentini, Slim Bechikh, Kalyanmoy Deb, Mel Ó Cinnéide: High dimensional search-based software engineering: finding tradeoffs among 15 objectives for automating software refactoring using NSGA-III. GECCO 2014: 1263-1270, ACM.
3. **Mohamed Wiem Mkaouer**, Marouane Kessentini, Slim Bechikh, Kalyanmoy Deb, Mel Ó Cinnéide: Recommendation system for software refactoring using innovization and interactive dynamic optimization. In Proceedings of the 29th ACM/IEEE international conference on Automated software engineering (ASE '14). ACM, 330-337, New York, NY, USA.
4. **Mohamed Wiem Mkaouer**, Marouane Kessentini, Slim Bechikh, Mel Ó Cinnéide: A Robust Multi-objective Approach for Software Refactoring under Uncertainty. SSBSE 2014: 168-183
5. **Mohamed Wiem Mkaouer**, Marouane Kessentini, Slim Bechikh, Daniel R. Tauritz: Preference-based multi-objective software modelling. CMSBSE@ICSE 2013: 61-66, IEEE.

Please note that only the following publications were included in this dissertation.

1. **Mkaouer, Mohamed Wiem** and Kessentini, Marouane. Model Transformation Using Multiobjective Optimization. *Advances in Computers Journal* 92: 161-202. 2014.
2. **Mkaouer, Mohamed Wiem**, Kessentini, Marouane, Shaout, Adnan, Koligheu, Patrice, Bechikh, Slim, Deb, Kalyanmoy, and Ouni, Ali. Many-objective software remodularization using NSGA-III. *ACM Transactions on Software Engineering and Methodology Journal (TOSEM)* 24, 3, Article 17 (May 2015), 45 pages, ACM.
3. **Mkaouer, Mohamed Wiem**, Kessentini, Marouane, Bechikh, Slim, Ó Cinnéide, Mel, and Deb, Kalyanmoy. On the use of many quality attributes for software refactoring: a many-objective search-based software engineering approach. *Empirical Software Engineering Journal*: 1-43. (First online: 23 December 2015: 10.1007/s10664-015-9414-4), Springer.
4. **Mkaouer, Mohamed Wiem**, Kessentini, Marouane, Ó Cinnéide, Mel , Hayashi, Shinpei, and Deb, Kalyanmoy. A robust multi-objective approach to balance severity and importance of refactoring opportunities. *Empirical Software Engineering Journal*: 1-34. (First online: 04 March 2016: 10.1007/s10664-015-9414-4), 2016.
5. **Mohamed Wiem Mkaouer**, Marouane Kessentini, Slim Bechikh, Kalyanmoy Deb, Mel Ó Cinnéide: Recommendation system for software refactoring using innovization and interactive dynamic optimization. In Proceedings of the 29th ACM/IEEE international conference on Automated software engineering (ASE '14). ACM, 330-337, New York, NY, USA.
6. **Mkaouer, Mohamed Wiem**, Kessentini, Marouane, Ali Ouni, Ó Cinnéide, Mel and Deb, Kalyanmoy. Interactive Dynamic Search-Based Software Engineering: A Case Study on Software Refactoring. *ACM Transactions on Software Engineering and Methodology Journal (TOSEM),* Under revisions.

BIBLIOGRAPHY

1 IEEE Std. 1219-1998. *Standard for Software Maintenance. IEEE Computer Society Press, Los Alamitos, CA*. 1998.

2 Abran, Alain and Nguyenkim, Hong. *"Measurement of the maintenance process from a demand-based perspective. Journal of Software Maintenance: Research and Practice 5(2): 63-90.* 1993.

3 Miller, Webb and Spooner, David L. *Automatic generation of floating-point test data. IEEE Transactions on Software Engineering 2(3): 223.* 1976.

4 Harman, Mark and Jones, Bryan F. *Search-based software engineering. Information and software Technology 43(14): 833-839.* 2001.

5 Harman, Mark, Mansouri, S. Afshin, and Zhang, Yuanyuan. *Search-based software engineering: Trends, techniques and applications. ACM Computing Surveys 45(1): 1-61*. 2012.

6 Harman, Mark and Tratt, Laurence. *Pareto optimal search based refactoring at the design level. Proceedings of the 9th annual conference on Genetic and evolutionary computation. London, England, ACM: 1106-1113.* 2007.

7 France, Robert and Rumpe, Bernard. *Model-driven development of complex software: A research roadmap. Future of Software Engineering, IEEE Computer Society*. 2007.

8 Taentzer, Gabriele. *AGG: a graph transformation environment for system modeling and validation. Proc. Tool Exihibition at Formal Methods.* 2003.

9 Varró, Dániel and Pataricza, András. *Generic and meta-transformations for model transformation engineering. «UML» 2004—The Unified Modeling Language. Modeling Languages and Applications, Springer: 290-304.* 2004.

10 ATLAS Group. *The ATLAS Transformation Language. http:// www.eclipse.org/gmt*. 2000.

11 Compuware, SUN. *MOF 2.0 Query/Views/Transformations RFP, Revised Submission.OMGDocument ad/2003-08-07. http://www.omg.org/cgi-bin/doc?ad/2003-08-07*. 2003.

12 Clark, T. and Warmer J. *Object Modeling with the OCL: The Rationale behind the Object Constraint Language, volume 2263 of Lecture Notes in Computer Science, Heidelberg, Springer-Verlag. ISBN*. 2002.

13 Ribeiro, Oscar R. and Fernandes, João M. *Some rules to transform sequence diagrams into coloured Petri nets. 7th Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools (CPN 2006), Citeseer*. 2006.

14 Ouardani, Adel, Esteban, Philippe, Paludetto, Mario, and Pascal, Jean-Claude. *A Meta-modeling Approach for Sequence Diagrams to Petri Nets Transformation within the*

requirements validation process. Proceedings of the European Simulation and Modeling Conference (ESM2006), Citeseer. 2006.

15 Fowler, Martin, Beck, Kent, Brant, John, Opdyke, William, and Roberts, Don. *Refactoring – Improving the design of existing code. Addison Wesley, ISBN 978-0201485677*. 1999.

16 Ermel, Claudia, Ehrig, Hartmut, and Ehrig, Karsten. *Refactoring of model transformations. Electronic Communications of the EASST 18.* 2009.

17 Anquetil, Nicolas and Lethbridge, Timothy C. *Experiments with clustering as a software remodularization method. IEEE Sixth Working Conference on roceedings.* 1999.

18 Abdeen, Hani, Ducasse, Stéphane, Sahraoui, Houari, and Alloui, Ilham. *Automatic package coupling and cycle minimization. IEEE 16th Working Conference on Reverse Engineering*. 2009.

19 Bavota, Gabriele and Carnevale, Filomena. *Putting the developer in a loop:an interactive GA for Software Re-modularization. Search based software engineering, Lecture notes in computer science volume 7515. pp 75-89*. 2012.

20 Deb, Kalyanmoy and Jain, Himanshu. *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. 18(4): 577-601*. 2014.

21 Deb, Kalyanmoy, Pratap, Amrit, Agarwal, Sameer, and Meyarivan, T. *A fast and elitist multiobjective genetic algorithm: NSGA-II. IEEE Transactions on Evolutionary Computation 6(2): 182-197*. 2002.

22 Deb, Kalyanmoy and Srinivasan, Aravind. *Innovization: Innovating design principles through optimization. Proceedings of the 8th annual conference on Genetic and evolutionary computation, ACM.* 2006.

23 Mkaouer, Mohamed Wiem and Kessentini, Marouane. *Model Transformation Using Multiobjective Optimization. Advances in Computers 92: 161-202*. 2014.

24 Mkaouer, Mohamed Wiem, Kessentini, Marouane, Shaout, Adnan, Koligheu, Patrice, Bechikh, Slim, Deb, Kalyanmoy, and Ouni, Ali. *Many-objective software remodularization using NSGA-III. ACM Transactions on Software Engineering and Methodology (TOSEM) 24(3): 17*. 2015.

25 Mkaouer, Mohamed Wiem, Kessentini, Marouane, Bechikh, Slim, Ó Cinnéide, Mel, and Deb, Kalyanmoy. *On the use of many quality attributes for software refactoring: a many-objective search-based software engineering approach. Empirical Software Engineering: 1-43.* 2015.

26 Mkaouer, Mohamed Wiem, Kessentini, Marouane, Ó Cinnéide, Mel , Hayashi, Shinpei, and Deb, Kalyanmoy. *A robust multi-objective approach to balance severity and importance of refactoring opportunities. Empirical Software Engineering: 1-34.* 2016.

27 Zhuang, Liu, HeQing, Guo, Dong, Li, Li, Tao, and Juan, Zhang Juan. *Solving Multi-objective and fuzzy multi-attributive integrated technique for QoS-Aware Web Service Selection. IEEE International Conference on Wireless Communications, Networking and Mobile Computing.* 2007.

28 Wada, Hiroshi, Champrasert, Paskorn, Suzuki, Junichi, and Oba, Katsuya. *Multiobjective optimization of sla-aware service composition. IEEE Congress on Services-Part I.* 2008.

29 Bowman, Michael, Briand, Lionel C, and Labiche, Yvan. *Solving the class responsibility assignment problem in object-oriented analysis with multi-objective genetic algorithms. IEEE Transactions onSoftware Engineering. 36(6): 817-837.* 2010.

30 Kremmel, Thomas, Kubalík, Jiří, and Biffl, Stefan. *Software project portfolio optimization with advanced multiobjective evolutionary algorithms. Applied Soft Computing 11(1): 1416-1426.* 2011.

31 Rodríguez, Daniel, Ruiz, Mercedes, Riquelme, José C, and Harrison, Rachel. *Multiobjective simulation optimisation in software project management. Proceedings of the 13th annual conference on Genetic and evolutionary computation, ACM.* 2011.

32 Praditwong, Kata, Harman, Mark, and Yao, Xin. *Software module clustering as a multi-objective search problem. IEEE Transactions on Software Engineering 37(2): 264-282.* 2011.

33 Barros, Marcio de Oliveira. *An analysis of the effects of composite objectives in multiobjective software module clustering. Proceedings of the 14th annual conference on Genetic and evolutionary computation, ACM.* 2012.

34 Colanzi, Thelma Elita and Vergilio, Silvia Regina. *Applying search based optimization to software product line architectures: Lessons learned. Search Based Software Engineering, Springer: 259-266.* 2012.

35 Sarro, Federica, Ferrucci, Filomena, and Gravino, Carmine. *Single and multi objective genetic programming for software development effort estimation. Proceedings of the 27th annual ACM symposium on applied computing, ACM.* 2012.

36 Sayyad, Abdel Salam, Menzies, Tim, and Ammar, Hany. *On the value of user preferences in search-based software engineering: a case study in software product lines. IEEE 35th International conference onSoftware engineering*. 2013.

37 Sayyad, Abdel Salam, Ingram, Joe, Menzies, Tim, and Ammar, Hany. *Scalable product line configuration: A straw to break the camel's back. IEEE/ACM 28th International Conference on Automated Software Engineering*. 2013.

38 Kalboussi, Sabrine , Bechikh, Slim, Kessentini, Marouane, and Ben Said, Lamjed. *Preference-based Many-objective Evolutionary Testing Generates Harder Test Cases for Autonomous Agents. 5th international Symposium on Search-Based Software Engineering*. 2013.

39 Ramírez, Aurora, Romero, José Raúl, and Ventura, Sebastián. *On the performance of multiple objective evolutionary algorithms for software architecture discovery. Proceedings of the 2014 conference on Genetic and evolutionary computation, ACM.* 2014.

40 Olaechea, Rafael, Rayside, Derek, Guo, Jianmei, and Czarnecki, Krzysztof. *Comparison of exact and approximate multi-objective optimization for software product lines. Proceedings of the 18th International Software Product Line Conference-Volume 1, ACM.* 2014.

41 Panichella, Annibale, Kifetew, Fitsum Meshesha, and Tonella, Paolo. *Reformulating branch coverage as a many-objective optimization problem. IEEE 8th International Conference on Software Testing, Verification and Validation.* 2015.

42 Kleppe, Anneke G, Warmer, Jos, and Bast, Wim. *The model driven architecture: practice and promise, Addison-Wesley Longman Publishing Co., Inc., Boston, MA.* 2003.

43 Wimmer, Manuel, Strommer, Michael, Kargl, Horst, and Kramler, Gerhard. *Towards model transformation generation by-example. IEEE 40th Annual Hawaii International Conference on System Sciences*. 2007.

44 Sun, Yu, White, Jules, and Gray, Jeff. *Model transformation by demonstration. Model Driven Engineering Languages and Systems, Springer: 712-726.* 2009.

45 Dolques, Xavier, Huchard, Marianne, Nebut, Clementine, and Reitz, Philippe. *Learning transformation rules from transformation examples: An approach based on relational concept analysis. 14th IEEE International Enterprise Distributed Object Computing Conference Workshops*. 2010.

46 Langer, Philip, Wimmer, Manuel, and Kappel, Gerti. *Model-to-model transformations by demonstration. Theory and Practice of Model Transformations, Springer: 153-167.* 2010.

47 Mancoridis, Spiros, Mitchell, Brian S, Rorres, Chris, Chen, Yih-Farn, and Gansner, Emden R. *Using Automatic Clustering to Produce High-Level System Organizations of Source Code. IWPC, Citeseer.* 1998.

48 Mitchell, Brian and Mancoridis, Spiros. *On the automatic modularization of software systems using the Bunch tool. IEEE Transactions on Software Engineering 32(3): 193-208.* 2006.

49 Harman, Mark, Hierons, Robert M., and Mark, Proctor. *A New Representation And Crossover Operator For Search-based Optimization Of Software Modularization. Proceedings of the Genetic and Evolutionary Computation Conference, Morgan Kaufmann Publishers Inc.: 1351-1358.* 2002.

50 Seng, Olaf, Stammel, Johannes, and Burkhart, David. *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, ACM.* 2006.

51 Bavota, Gabriele, De Lucia, Andrea, Marcus, Andrian, and Oliveto, Rocco. *Software re-modularization based on structural and semantic metrics. 17th Working Conference on Reverse Engineering*. 2010.

52 Ouni, Ali, Kessentini, Marouane, Sahraoui, Houari, and Hamdi, Mohamed Salah. *Search-based refactoring: Towards semantics preservation. 28th IEEE International Conference on Software Maintenance (ICSM)*. 2012.

53 Mäntylä, Mika, Vanhanen, Jari, and Lassenius, Casper. *A taxonomy and an initial empirical study of bad smells in code. International Conference on Software Maintenance*. 2003.

54 Marinescu, Radu. *Detection strategies: Metrics-based rules for detecting design flaws. IEEE International Conference on Software Maintenance*. 2004.

55 Moha, Naouel, Gueheneuc, Yann-Gael, Duchien, Laurence, and Le Meur, Anne-Francoise. *DECOR: A method for the specification and detection of code and design smells. IEEE Transactions on Software Engineering 36(1): 20-36.* 2010.

56 Van Emden, Eva and Moonen, Leon. *Java quality assurance by detecting code smells. IEEE 9th Working Conference on Reverse Engineering.* 2002.

57 Martin, Robert C. *Design principles and design patterns. Object Mentor 1: 34.* 2000.

58 Counsell, Steve, Hierons, Robert M, Najjar, Rajaa, Loizou, George, and Hassoun, Youssef. *The effectiveness of refactoring, based on a compatibility testing taxonomy and a dependency graph. IEEE Academic and Industrial Conference-Practice And Research Techniques*. 2006.

59 Piveta, Eduardo Kessler, Hecht, Marcelo, Pimenta, Marcelo Soares, and Price, Roberto Tom. *Detecting Bad Smells in AspectJ. Journal of Universal Computer Science 12(7): 811-827*. 2006.

60 Meananeatra, Panita. *Identifying refactoring sequences for improving software maintainability. Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. 2012.

61 Tsantalis, Nikolaos, Chaikalis, Theodoros, and Chatzigeorgiou, Alexander. *JDeodorant: Identification and removal of type-checking bad smells. IEEE 12th European Conference on Software Maintenance and Reengineering*. 2008.

62 Du Bois, Bart, Demeyer, Serge, and Verelst, Jan. *Refactoring - improving coupling and cohesion of existing code. 11th Working Conference on Reverse Engineering.* 2004.

63 Murphy-Hill, Emerson. *Improving usability of refactoring tools. Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*. 2006.

64 Xi, Ge and Murphy-Hill , Emerson. *Manual refactoring changes with automated refactoring validation. Proceedings of the 36th International Conference on Software Engineering. Hyderabad, India, ACM: 1095-1105.* 2014.

65 Xi, Ge and Murphy-Hill, Emerson. *BeneFactor: a flexible refactoring tool for eclipse. Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion. Portland, Oregon, USA, ACM: 19-20*. 2011.

66 Foster, Stephen R., Griswold, William G., and Lerner, Sorin. *WitchDoctor: IDE support for real-time auto-completion of refactorings. 34th International Conference onSoftware Engineering (ICSE).* 2012.

67 Tahvildari, L.adan and Kontogiannis , Kostas. *A metric-based approach to enhance design quality through meta-pattern transformations. Seventh European Conference on Software Maintenance and Reengineering*. 2003.

68 Dig, Danny. *A Refactoring Approach to Parallelism. IEEE Software 28(1). pp. 17-22*. 2011.

69 Kataoka, Yoshio, Ernst, Michael D., Griswold, William G. , and Notkin, David. *Automated Support for Program Refactoring Using Invariants. International Conference in Software Maintenance. pp. 736-743.* 2001.

70 O'Keeffe, Mark and Ó Cinnéide, Mel. *Search-based refactoring for software maintenance. Journal of Systems and Software 81(4): 502-516.* 2008.

71 Kessentini, Marouane , Kessentini, Wael, Sahraoui, Houari , Boukadoum, Mounir, and Ouni, Ali. *Design Defects Detection and Correction by Example. IEEE 19th International Conference on Program Comprehension (ICPC)*. 2011.

72 Kilic, Hurevren, Koc, Ekin, and Cereci, Ibrahim. *Search-based parallel refactoring using*

population-based direct approaches. Search Based Software Engineering, Springer: 271-272. 2011.

73 Jouault, Frédéric and Kurtev, Ivan. *Transforming models with ATL. satellite events at the MoDELS Conference, Springer.* 2005.

74 Wiggerts, Theo A. *Using clustering algorithms in legacy systems remodularization. IEEE 4th Working Conference on Reverse Engineering*. 1997.

75 Maqbool, Onaiza and Babri, Haroon A. *Hierarchical clustering for software architecture recovery. IEEE Transactions on Software Engineering. 33(11): 759-780.* 2007.

76 Mitchell, Brian S and Mancoridis, Spiros. *On the evaluation of the Bunch search-based software modularization algorithm. Software Computing 12(1): 77-93.* 2008.

77 Shtern, Mark and Tzerpos, Vassilios. *Methods for selecting and improving software clustering algorithms. IEEE 17th International Conference on Program Comprehension*. 2009.

78 Bavota, Gabriele, De Lucia, Andrea, Marcus, Andrian, and Oliveto, Rocco. *Using structural and semantic measures to improve software modularization. Empirical Software Engineering 18(5): 901-932.* 2013.

79 Abdeen, Hani, Sahraoui, Houari, Shata, Osama, Anquetil, Nicolas, and Ducasse, Stéphane. *Towards automatically improving package structure while respecting original design decisions. 20th Working Conference on Reverse Engineering*. 2013.

80 Palomba, Fabio, Bavota, Gabriele, Di Penta, Massimiliano, Oliveto, Rocco, De Lucia, Andrea, and Poshyvanyk, Denys. *Detecting bad smells in source code using change history information. IEEE/ACM 28th International Conference on Automated Software Engineering*. 2013.

81 Ferrucci, Filomena, Harman, Mark, Ren, Jian, and Sarro, Federica. *Not going to take this anymore: multi-objective overtime planning for software engineering projects. IEEE International Conference on Software Engineering*. 2013.

82 Jaimes, Antonio López, Montano, Alfredo Arias, and Coello Coello , Carlos A. *Preference incorporation to solve many-objective airfoil design problems. IEEE Congress on Evolutionary Computation.* 2011.

83 Deb, Kalyanmoy and Jain, Himanshu. *Handling many-objective problems using an improved NSGA-II procedure. IEEE congress on Evolutionary computation*. 2012.

84 Harman, Mark. *Software engineering: An ideal set of challenges for evolutionary computation. Proceedings of the 15th annual conference companion on Genetic and evolutionary computation.* 2013.

85 Di Pierro, Francesco, Khu, Soon-Thiam, and Savic, Dragan A. *An investigation on preference order ranking scheme for multiobjective evolutionary optimization. IEEE Transactions on Evolutionary Computation11(1): 17-45.* 2007.

86 Ouni, Ali, Kessentini, Marouane, Sahraoui, Houari, and Boukadoum, Mounir. *Maintainability defects detection and correction: a multi-objective approach. Automated Software Engineering 20(1): 47-79.* 2012.

87 Opdyke, William and Johnson, Ralph. *Refactoring: An Aid in Designing Application*

Frameworks and Evolving Object-Oriented Systems. In Proceedings of the Symposium on Object Oriented Programming Emphasizing Practical Applications. ACM. 1990.

88 Hamdi, Mohamed Salah. *SOMSE: A semantic map based meta-search engine for the purpose of web information customization. Applied Soft Computing 11(1): 1310-1321*. 2011.

89 Kim, Miryung, Notkin, David, Grossman, Dan, and Wilson Jr, Gary. *Identifying and summarizing systematic code changes via rule inference. IEEE Transactions on Software Engineering 39(1): 45-62*. 2013.

90 Das, Indraneel and Dennis, John E. *Normal-boundary intersection: A new method for generating the Pareto surface in nonlinear multicriteria optimization problems. SIAM Journal on Optimization 8(3): 631-657*. 1998.

91 Asafuddoula, Md, Ray, Tapabrata, and Sarker, Ruhul. *A decomposition based evolutionary algorithm for many objective optimization with systematic sampling and adaptive epsilon control. Evolutionary Multi-Criterion Optimization, Springer.* 2013.

92 Ó Cinnéide, Mel, Tratt, Laurence, Harman, Mark, Counsell, Steve, and Hemati Moghadam, Iman. *Experimental assessment of software metrics using automated refactoring. Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement, ACM.* 2012.

93 Rachmawati, Lily and Srinivasan, Dipti. *Multiobjective evolutionary algorithm with controllable focus on the knees of the Pareto front. IEEE Transactions on Evolutionary Computation 13(4): 810-824.* 2009.

94 Prete, Kyle, Rachatasumrit, Napol, Sudan, Nikita, and Kim, Miryung. *Template-based reconstruction of complex refactorings. IEEE International Conference on Software Maintenance*. 2010.

95 Arcuri, Andrea and Fraser, Gordon. *Parameter tuning or default values? An empirical investigation in search-based software engineering. Empirical Software Engineering 18(3): 594-623.* 2013.

96 Arcuri, Andrea and Briand, Lionel. *A practical guide for using statistical tests to assess randomized algorithms in software engineering. 33rd International Conference on Software Engineering (ICSE).* 2011.

97 Brockhoff, Dimo and Zitzler, Eckart. *Are all objectives necessary? On dimensionality reduction in evolutionary multiobjective optimization. Parallel Problem Solving from Nature-PPSN IX, Springer: 533-542.* 2006.

98 Cohen, Jacob, Cohen, Patricia, West, Stephen G, and Aiken, Leona S. *Applied multiple regression/correlation analysis for the behavioral sciences, Routledge.* 2013.

99 Glass, Robert L. *Frequently forgotten fundamental facts about software engineering. IEEE software(3): 112,110-111.* 2001.

100 Hall, Tracy, Zhang, Min, Bowes, David, and Sun, Yi. *Some code smells have a significant but small effect on faults. ACM Transactions on Software Engineering and Methodology 23(4): 33.* 2014.

101 Yamashita, Aiko. *Yamashita, A. (2012). Assessing the capability of code smells to support software maintainability assessments: Empirical inquiry and methodological approach. Empirical Inquiry and Methodological Approach. Doctoral Thesis, University of Oslo*. 2012.

102 Jin, Yaochu and Branke, Jürgen. *Jin, Y. and J. Branke (2005). Evolutionary optimization in uncertain environments-a survey. IEEE Transactions on Evolutionary Computation 9(3): 303-317.* 2005.

103 Beyer, Hans-Georg and Sendhoff, Bernhard. *Robust optimization–a comprehensive survey. Computer methods in applied mechanics and engineering 196(33): 3190-3218.* 2007.

104 Mkaouer, Mohamed Wiem, Kessentini, Marouane, Bechikh, Slim, and Ó Cinnéide, Mel. *A robust multi-objective approach for software refactoring under uncertainty. Search-Based Software Engineering, Springer: 168-183.* 2014.

105 Li, Xiaodong. *A non-dominated sorting particle swarm optimizer for multiobjective optimization. Genetic and Evolutionary Computation. Springer*. 2003.

106 Fokaefs, Marios, Tsantalis, Nikolas, Stroulia, Eleni, and Chatzigeorgiou, Alexander. *JDeodorant: identification and application of extract class refactorings. 33rd International Conference on Software Engineering*. 2011.

107 Beyer, Hans-Georg. *Actuator noise in recombinant evolution strategies on general quadratic fitness models. Genetic and Evolutionary Computation–GECCO 2004, Springer.* 2004.

108 Das, Indraneel. *ROBUSTNESS OPTIMIZATION FOR CONSTRAINED NONLINEAR PROGRAMMING PROBLEMS. Engineering Optimization+ A35 32(5): 585-618.* 2000.

109 Jin, Yaochu and Sendhoff, Bernhard. *Trade-off between performance and robustness: an evolutionary multiobjective approach. Evolutionary Multi-Criterion Optimization, Springer.* 2003.

110 Chatzigeorgiou, Alexander and Manakos, Anastasios. *Investigating the evolution of code smells in object-oriented systems. Innovations in Systems and Software Engineering 10(1): 3-18.* 2013.

111 Olbrich, Steffen M, Cruze, Daniela S, and Sjøberg, Dag IK. *Are all code smells harmful? A study of God Classes and Brain Classes in the evolution of three open source systems. IEEE International Conference on Software Maintenance*. 2010.

112 Fontana, Francesca Arcelli, Mäntylä, Mika V, Zanoni, Marco, and Marino, Alessandro. *Comparing and experimenting machine learning techniques for code smell detection. Empirical Software Engineering: 1-49.* 2015.

113 *(Xerces-J). [Online]. Available: http://xerces.apache.org/xerces-j.*

114 *(JFreeChart). [Online]. Available: http://www.jfree.org/jfreechart/.*

115 *(GanttProject). [Online]. Available: http://www.ganttproject.biz.*

116 *(ApacheAnt). [Online]. Available: http://ant.apache.org.*

117 *(JHotDraw). [Online]. Available: http://www.jhotdraw.org.*

118 *(Rhino). [Online]. Available: https://developer.mozilla.org/en-US/docs/Rhino/.*

119 *(Log4J). [Online]. Available: http://logging.apache.org/.*

120 *(Nutch). [Online]. Available: http://nutch.sourceforge.net/.*

121 Vargha, András and Delaney, Harold D. *A critique and improvement of the CL common language effect size statistics of McGraw and Wong. Journal of Educational and Behavioral*

Statistics 25(2): 101-132. 2000.

122 Deb, Kalyanmoy and Gupta, Shivam. *Understanding knee points in bicriteria problems and their implications as preferred solution principles. Engineering optimization 43(11): 1175-1204.* 2011.

123 Zhang, Qingfu and Li, Hui. *MOEA/D: A multiobjective evolutionary algorithm based on decomposition. IEEE Transactions on Evolutionary Computation 11(6): 712-731*. 2007.

124 Zitzler, Eckart and Künzli, Simon. *Indicator-based selection in multiobjective search. Parallel Problem Solving from Nature-PPSN VIII, Springer*. 2004.

125 Mkaouer, Mohamed Wiem, Kessentini, Marouane, Bechikh, Slim, Deb, Kalyanmoy, and Ó Cinnéide, Mel. *High dimensional search-based software engineering: finding tradeoffs among 15 objectives for automating software refactoring using NSGA-III. Proceedings of the 2014 conference on Genetic and evolutionary computation, ACM.* 2014.

126 Bansiya, Jagdish and Davis, Carl G. *A hierarchical model for object-oriented design quality assessment. IEEE Transactions on Software Engineering. 28(1): 4-17.* 2002.

127 Du Bois, Bart and Mens, Tom. *Describing the impact of refactoring on internal program quality. International Workshop on Evolution of Large-scale Industrial Software Applications*. 2003.

128 Alshayeb, Mohammad. *Empirical investigation of refactoring effect on software quality. Information and software Technology 51(9): 1319-1326.* 2009.

129 Chidamber, Shyam R and Kemerer, Chris F. *A metrics suite for object oriented design. IEEE Transactions on Software Engineering 20(6): 476-493.* 1994.

130 Lorenz, Mark and Kidd, Jeff. *Object-oriented software metrics: a practical guide, Prentice-Hall, Inc.* 1994.

131 Brito e Abreu, Fernando. *The MOOD metrics set. Proceedings of the European Conference on Object-Oriented Programming (ECOOP). Workshop on Metrics, Vol. 95, p. 267*. 1995.

132 Shatnawi, Raed and Li, Wei. *An empirical assessment of refactoring impact on software quality using a hierarchical quality model. International Journal of Software Engineering and Its Applications 5(4): 127-149.* 2011.

133 Corazza, Anna, Di Martino, Sergio, and Maggio, Valerio. *LINSEN: An efficient approach to split identifiers and expand abbreviations. 28th IEEE International Conference on Software Maintenance (ICSM).* 2012.

134 Dean, Jeffrey, Grove, David, and Chambers, Craig. *Optimization of object-oriented programs using static class hierarchy analysis. ECOOP'95—Object-Oriented Programming, 9th European Conference, Åarhus, Denmark, August 7–11, 1995, Springer.* 1995.

135 Bechikh, Slim, Ben Said, Lamjed, and Ghedira, Khaled. *Estimating nadir point in multi-objective optimization using mobile reference points. IEEE congress on Evolutionary computation (CEC).* 2010.

136 Bechikh, Slim, Ben Said, Lamjed, and Ghédira, Khaled. *Searching for knee regions of the Pareto front using mobile reference points. Software Computing 15(9): 1807-1823.* 2011.

137 Basili, Victor R. *Software modeling and measurement: the Goal/Question/Metric paradigm*. 1992.

138 Marinescu, Radu, Ganea, George, and Verebi, Ioana. *inCode: Continuous quality assessment and improvement. 14th European Conference on Software Maintenance and Reengineering*. 2010.

139 Yang, Shengxiang, Li, Miqing, Liu, Xiaohui, and Zheng, Jinhua. *A grid-based evolutionary algorithm for many-objective optimization. IEEE Transactions on Evolutionary Computation 17(5): 721-736.* 2013.

140 Brown, William, Malveau, Raphael, and McCormick, Skip. *Anti patterns: refactoring software, architectures, and projects in crisis. John Wiley & Sons, ISBN 978-0471197133*. 1998.

141 Negara, Stas , Chen, Nicholas , Vakilian, Mohsen , Johnson, Ralph, and Dig, Danny. *A Comparative Study of Manual and Automated Refactorings. In the European Conference on Object-Oriented Programming. 552-576*. 2013.

142 Murphy-Hill, Emerson, Parnin, Chris , and Black, Andrew. *E. R. Murphy-Hill, C. Parnin, and A. P. Black, How we refactor, and how we know it, IEEE Transactions on Software Engineering, vol. 38, no. 1, pp. 5–18.* 2013.

143 Kim, Miryung, Zimmermann, Thomas, and Nagappan, Nachiappan. *A field study of refactoring challenges and benefits. Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, ACM.* 2012.

144 Yang, Linchao, Kamiya, Tomoyuki, Sakamoto, Kazunori, Washizaki, Hironori, and Fukazawa, Yoshiaki. *RefactoringScript: A Script and Its Processor for Composite Refactoring. International Conference on Software Engineering and Knowledge Engineering*. 2014.

145 Bavota, Gabriele, Carnevale, Filomena, De Lucia, Andrea, Di Penta, Massimiliano, and Oliveto, Rocco. *Putting the developer in-the-loop: an interactive GA for software re-modularization. Search Based Software Engineering, Springer: 75-89*. 2012.

146 Hall, Mathew , Walkinshaw, Neil , and McMinn, Phil. *Supervised Software Modularization. 28th IEEE International Conference on Software Maintenance. pp. 23-30.* 2012.

147 Ouni, Ali, Kessentini, Marouane , Sahraoui, Houari, and Hamdi, Mohamed Salah. *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. Amsterdam, The Netherlands, ACM: 1461-1468.* 2013.

148 Deb, Kalyanmoy, Sinha, Ankur, Korhonen, Pekka J., and Wallenius, Jyrki. *An interactive evolutionary multiobjective optimization method based on progressively approximated value functions. IEEE Transactions on Evolutionary Computation. 14(5): 723-739.* 2010.

149 Karahan, Ibrahim and Köksalan, Murat. *A territory defining multiobjective evolutionary algorithms and preference incorporation. IEEE Transactions on Evolutionary Computation. 14(4): 636-664.* 2010.

150 Fernandez, Eduardo, Lopez, Edy, Bernal, Sergio, Coello Coello, Carlos A., and Navarro, Jorge. *Evolutionary multiobjective optimization using an outranking-based dominance generalization. Computers & Operations Research 37(2): 390-395.* 2010.

151 Wagner, Tobias and Trautmann, Heike. *Integration of preferences in hypervolume-based multiobjective evolutionary algorithms by means of desirability functions. IEEE*

Transactions on Evolutionary Computation. 14(5): 688-701. 2010.

152 Farina, Marco, Deb, Kalyanmoy, and Amato, Paolo. *Dynamic multiobjective optimization problems: test cases, approximations, and applications. IEEE Transactions on Evolutionary Computation 8(5): 425-442.* 2004.

153 *(JVacation). [Online]. Available: https://sourceforge.net/projects/jvacation/.*

154 Eiben, Agoston E and Smit, Selmar K. *Parameter tuning for configuring and analyzing evolutionary algorithms. Swarm and Evolutionary Computation 1(1): 19-31*. 2011.

155 Miller, Rupert G Jr. *Beyond ANOVA: basics of applied statistics, CRC Press.* 1997.

156 Gupta, Abhishek, Ong, Yew-Soon, and Feng, Liang. *Multifactorial evolution: towards evolutionary multitasking. Accepted IEEE Transactions on Evolutionary Computation 10: 1109.* 2015.

157 Murphy-Hill, Emerson and Black, Andrew. *Breaking the barriers to successful refactoring: observations and tools for extract method. Proceedings of the 30th international conference on Software engineering. Leipzig, Germany, ACM: 421-430.* 2008.

158 Murphy-Hill, Emerson and Black, Andrew. *Programmer-Friendly Refactoring Errors. IEEE Transactions on Software Engineering 38(6): 1417-1431*. 2012.

159 Murphy-Hill, Emerson and Black, Andrew. *Refactoring Tools: Fitness for Purpose. IEEE Software 25(5): 38-44.* 2008.

160 Taentzer, Gabriele. *AGG: a graph transformation environment for system modeling and validation. Proc. Tool Exihibition at Formal Methods.* 2003.

161 Bavota, Gabriel, Carluccio , BD, De Lucia, Andrea, Di Penta, Massimiliano, Oliveto, Rocco, and Strollo, O. *When Does a Refactoring Induce Bugs." An Empirical Study. SCAM.* 2012.

162 Marinescu, Radu. *Marinescu, R. (2004). Detection strategies: Metrics-based rules for detecting design flaws. IEEE 20th International Conference onSoftware Maintenance*. 2004.

163 Jaimes, Antonio López, Coello Coello, Carlos A, and Barrientos, Jesús E Urías. *Online objective reduction to deal with many-objective problems. Evolutionary multi-criterion optimization, Springer*. 2009.

164 Bechikh, Slim, Kessentini, Marouane, Ben Said, Lamjed, and Ghédira, Khaled. *Preference Incorporation in Evolutionary Multiobjective Optimization: A Survey of the State-of-the-Art. Advances in Computers. R. H. Ali, Elsevier. Volume 98: 141-207.* 2015.

165 Yang, Linchao , Kamiya, Tomoyuki , and Sakamoto, Kazunori. *RefactoringScript: A Script and Its Processor for Composite Refactoring: 711-716.* 2014.