Université de Montréal

A Mono- and Multi-objective Approach for Recommending Software Refactoring

par

Ali Ouni

Département d'informatique et de recherche opérationnelle

Faculté des arts et des sciences

Thèse présentée à la Faculté des arts et des sciences

en vue de l'obtention du grade de Philosophiæ Doctor (Ph.D.)

en Informatique

Août, 2014

Résumé

Les systèmes logiciels sont devenus de plus en plus répondus et importants dans notre société. Ainsi, il y a un besoin constant de logiciels de haute qualité. Pour améliorer la qualité de logiciels, l'une des techniques les plus utilisées est le *refactoring* qui sert à améliorer la structure d'un programme tout en préservant son comportement externe. Le refactoring promet, s'il est appliqué convenablement, à améliorer la compréhensibilité, la maintenabilité et l'extensibilité du logiciel tout en améliorant la productivité des programmeurs. En général, le refactoring pourra s'appliquer au niveau de spécification, conception ou code. Cette thèse porte sur l'automatisation de processus de recommandation de refactoring, au niveau code, s'appliquant en deux étapes principales: 1) la détection des fragments de code qui devraient être améliorés (*e.g.*, les défauts de conception), et 2) l'identification des solutions de refactoring à appliquer.

Pour la première étape, nous traduisons des régularités qui peuvent être trouvés dans des exemples de défauts de conception. Nous utilisons un algorithme génétique pour générer automatiquement des règles de détection à partir des exemples de défauts.

Pour la deuxième étape, nous introduisons une approche se basant sur une recherche heuristique. Le processus consiste à trouver la séquence optimale d'opérations de refactoring permettant d'améliorer la qualité du logiciel en minimisant le nombre de défauts tout en priorisant les instances les plus critiques. De plus, nous explorons d'autres objectifs à optimiser: le nombre de changements requis pour appliquer la solution de refactoring, la préservation de la sémantique, et la consistance avec l'historique de changements. Ainsi, réduire le nombre de changements permets de garder autant que possible avec la conception initiale. La préservation de la sémantique assure que le programme restructuré est sémantiquement cohérent. De plus, nous utilisons l'historique de changement pour suggérer de nouveaux refactorings dans des contextes similaires.

En outre, nous introduisons une approche multi-objective pour améliorer les attributs de qualité du logiciel (la flexibilité, la maintenabilité, etc.), fixer les « *mauvaises* » pratiques de conception (défauts de conception), tout en introduisant les « *bonnes* » pratiques de conception (patrons de conception).

Mots-clés : Défauts de Conception, Restructuration de logiciel, Maintenance de logiciels.

Abstract

Software systems have become prevalent and important in our society. There is a constant need for high-quality software. Hence, to improve software quality, one of the most-used techniques is the *refactoring* which improves design structure while preserving the external behavior. Refactoring has promised, if applied well, to improve software readability, maintainability and extendibility while increasing the speed at which programmers can write and maintain their code. In general, refactoring can be performed in various levels such as the requirement, design, or code level. In this thesis, we mainly focus on the source code level where automated refactoring recommendation can be performed through two main steps: 1) detection of code fragments that need to be improved/fixed (e.g., code-smells), and 2) identification of refactoring solutions to achieve this goal.

For the code-smells identification step, we translate regularities that can be found in such code-smell examples into detection rules. To this end, we use genetic programming to automatically generate detection rules from examples of code-smells.

For the refactoring identification step, a search-based approach is used. The process aims at finding the optimal sequence of refactoring operations that improve software quality by minimizing the number of detected code-smells while prioritizing the most critical ones. In addition, we explore other objectives to optimize using a multi-objective approach: the code changes needed to apply refactorings, semantics preservation, and the consistency with development change history. Hence, reducing code changes allows us to keep as much as possible the initial design. On the other hand, semantics preservation insures that the refactored program is semantically coherent, and that it models correctly the domain-semantics. Indeed, we use knowledge from historical code change to suggest new refactorings in similar contexts.

Furthermore, we introduce a novel multi-objective approach to improve software quality attributes (i.e., flexibility, maintainability, etc.), fix "bad" design practices (i.e., code-smells) while promoting "good" design practices (i.e., design patterns).

Keywords: Search-based Software Engineering, Software Maintenance, Code-smells, Refactoring.

Contents

List of tables

List of figures

List of Acronyms

CCR	Code-smell Correction Ratio
CD	Cohesion-based dependency
CRO	Chemical Reaction Optimization
DS	Dependency-based Similarity
FIU	Feature inheritance usefulness
GA	Genetic Algorithm
GP	Genetic Programming
ICR	Importance Correction Ratio
IS	Implementation-based similarity
MOGA	Multi-Objective Genetic Algorithm
NSGA-II	Non-Dominated Sorting Genetic Algorithm
PSO	Particle Swarm Optimization
RCR	Risk Correction Ratio
RM	Refactoring Meaningfulness
RO	Refactoring Operation
RP	Refactoring Precision
RS	Random Search
SA	Simulated Annealing
SCR	Severity Correction Ratio
VS	Vocabulary-based Similarity

To my parents

To my sisters

To my friends

Acknowledgments

Feeling gratitude and not expressing it is like wrapping a present and not giving it.

(by William Arthur Ward, 1921–1994).

I would like to thank the Tunisian Ministry of Higher Education and Scientific Research, and the University of Montreal for co-funding this research work through many excellence scholarships.

I would like to thank my PhD supervisor Prof. Houari Sahraoui for giving me the chance to do such a work within the GEODES Software Engineering Laboratory and for his valuable comments and critics which led to results of excellent quality. He taught me how to assess the research value of each proposal and how to be selective. He is a patient and thoughtful mentor; he listens to his students, not only what they are saying but also what they are not saying. I am proud to be one of his students.

I would like to express my infinite gratitude to my co-supervisor Prof. Marouane Kessentini for his unending encouragement and support, for showing me the joy of research, for shaping my path to research, for creating an enjoyable working environment, and for becoming a friend! His guidance was very constructive and rich so that they helped me to improve the quality of my work. Marouane was a great co-supervisor with kind personality attitudes. I consider him as more than a supervisor but even a member of my family.

I would like to thank Prof. Katsuro Inoue for accepting to be my external examiner and for furnishing efforts for reviewing my dissertation. I thank Prof. Olga Baysal for enthusiastically accepting to be on my PhD committee and for reviewing my dissertation.

Many thanks for Prof. Jean Meunier who was already on my pre-doctoral committee and who has enthusiastically accepted to chair my doctoral committee.

Thanks to all my friends in the Université de Montréal, Missouri University of Science and Technology, and University of Michigan. These years and the people I have met are now a part of me, and part of these pages. To them, and to those who were with me from the beginning, I owe this work.

Last but not least, I would like to express my special thanks to my parents Naceur and Kadija and my sisters who never stopped believing in me. I would have never managed any of this without their love and encouragement.

— Ali Ouni

Chapter 1 : Introduction

1.1 Research context

Source code of large systems is iteratively refined, restructured and evolved due to many reasons such as correcting errors in design, modifying a design to accommodate changes in requirements, and modifying a design to enhance existing features. Many studies reported that these software maintenance activities consume more than 70% of the overall cost of a typical software project [3].

This high cost could potentially be greatly reduced by providing automatic or semi-automatic solutions to avoid bad-practices and increase in turn software understandability, adaptability and extensibility. As a result, there has been much research focusing on the study of bad design and programming practices, also called code-smells, design defects, anti-patterns or anomalies [1] [2] [10] [12] in the literature. Although code-smells are sometimes unavoidable, they should be in general prevented by the development teams and removed from their code base as early as possible for maintainability and evolution considerations. In fact, improving the quality of existing software will drastically improve productivity and competitiveness of our software industry.

Improving the quality of software induce the detection and correction of code-smells. Typically, code smells refer to bad programming practices that adversely affect the development of software. As stated by Fenton and Pfleeger [2], code-smells are unlikely to cause failures directly, but may do it indirectly. In general, they make a software system difficult to maintain, which may often introduce bugs. The most well-known example of code-smells is the *Blob* which is found in code fragments where one large class monopolizes the behavior of a system, and other classes primarily contain data. Removing these code-smells help developers to easily understand, maintain and evolve their source code [1].

One of the widely used techniques to fix code-smells is *refactoring* – the process of changing software structure while preserving its external behavior [62] – which has been practiced by programmers for many years. The idea is to reorganize variables, classes and methods in order to facilitate future extensions. This reorganization is used to improve different aspects of software-quality such as reusability, maintainability, complexity, etc. In general, refactoring is performed through two main steps: 1) detection of code fragments that need to be improved (e.g., code-smells) and 2) identification of refactoring solutions to achieve this goal. In this thesis, we explore and address the automation of these two steps.

1.2 Problem statement

Detecting and fixing code-smells is still, to some extent, a difficult, time-consuming, manual and error-prone process [10]. As a consequence, automating code-smells detection and correction is considered as a challenging software engineering task [10] [12] [71]. In the next subsections, we highlight the different problems and challenges addressed in this thesis that are mainly related to the automation of code-smells detection and refactoring tasks.

1.2.1 Automating code-smells detection

The code-smells detection task consists of finding code fragments that violate common object-oriented principles (structure or semantic properties) on code elements such as the ones involving coupling and complexity. In fact, the common idea in existing contributions [33] [38] [43] consist of defining rules manually to identify key symptoms that characterize a code-smell using combinations of mainly quantitative and/or structural information. However, in an exhaustive scenario, the number of possible code-smells to manually characterize according to rules can be very large. On the other hand, Moha et al. [8] proposes to generate detection rules using formal definitions of code-smells. Although this partial automation of rule writing helps developers with symptom description, still, translating symptoms into rules is not obvious because there is no consensus of defining code-smells based on their symptoms [12]. Although there is a substantial amount of research work focusing on the detection of code-smells [1] [2] [8] [12] [14] [33] [38], there are many open issues that need to be addressed when defining a detection strategy. We highlight these open issues, as follows:

Problem #1.1. Most of the existing approaches are based on translating symptoms into rules. However, there is a difference between detecting symptoms and asserting that the detected situation is an actual code-smell.

Problem #1.2. There is no consensus on the definition of code-smells based on their symptoms. Although when consensus exists, the same symptom could be associated to many code-smell types, which may compromise the precise identification of smell types.

Problem #1.3. The majority of existing detection methods does not provide an efficient manner to guide a manual inspection of code-smell candidates. Indeed, the process of manually defining detection rules, understanding the code-smell candidates, selecting the true positives, and correcting them is time-consuming, fastidious, and not always profitable.

Problem #1.4. Existing approaches require an expert to manually write and validate detection rules.

1.2.2 Automating code-smells correction

Once code-smells are detected, they need to be fixed. One of the widely used techniques to fix code-smells and improve the quality of software systems is refactoring. Software refactoring improves design structure of the system while preserving its external behavior [1]. These two concerns drive the existing approaches to refactoring automation. To the best of our knowledge, most of the existing contributions [20] [21] [45] [49] suggest refactorings with the perspective of improving only some design/quality metrics while satisfying a set of pre- and post-conditions [17] to preserve the external behaviour. However, these concerns may not be enough to produce optimal and consistent refactoring solutions. In addition to quality improvement and behavior preservation, other aspects should be taken into consideration. Hence, to obtain good refactoring strategies, other considerations have to be targeted such as preserving the semantic coherence of the refactored program, reducing the amount of code changes required to apply refactoring, maintaining the consistency with prior code changes and reuse good refactorings applied/recorded in the past in similar contexts. In this setting, several open issues should be addressed when searching for refactoring solutions to improve the quality of software systems (i.e., fix code-smells). Hence, we identify the following problems.

Problem #2.1. The majority of existing approaches [1] [40] [41] have manually defined "standard" refactorings for each code-smell type to remove its symptoms as described in Fowler's book [1]. However, it is difficult to define "standard" refactoring solutions for each code-smell type and to generalize them because these solutions may vary depending on the programs and their context.

Problem #2.2. Removing code-smell symptoms does not mean that the actual code-smell is corrected, and, in the majority of cases, these "standard" solutions are unable to remove all symptoms for each code-smell.

Problem #2.3. Different possible refactoring strategies should be defined for the same type of code-smell. The problem is how to find the "best" refactoring solutions from a large list of candidate refactorings and how to combine them in an appropriate order? The list of all possible refactoring strategies, for each code-smell, can be very large [25]. Thus, the process of defining refactoring strategies manually, from an exhaustive list of refactorings, is fastidious, time-consuming, and error-prone.

Problem #2.4. In the majority of existing approaches [20] [21] [22] [49], code quality can be improved without fixing code-smells. In other terms, improving some quality metrics does not guarantee that the detected code-smells are fixed. Therefore, the link between code-smells detection (refactoring opportunities) and correction is not obvious. Thus, we need to ensure whether the refactoring concretely fixes the detected code-smells.

Problem #2.5. Existing approaches consider the refactoring (*i.e.,* the correction process) as a local process by fixing code-smells (or improving quality) separately. That is, a refactoring solution should not be specific to only one code-smell type. Instead, the impact of refactoring should be considered on the whole system. For example, moving methods to reduce the size/complexity of a class may increase the global coupling, or fixing some code-smells may create other code-smells in other code fragments.

Problem #2.6. In practice, not all code-smells have equal effects and importance [63]. Each individual instance has its severity score that allows developers to immediately spot and fix the most critical instances of each code-smell. Concretely, the same code-smell type can occur in different code fragments but with different effect and risk [62] [64] [81]. In general, developers need to focus their effort on fixing code-smells of the higher risk and severity. Thus, the prioritization of the list of detected code-smells is required based on different criteria such as severity, risk, importance, development team preferences, etc. However, most of the existing refactoring approaches deal with code-smells to fix as if they are of same importance.

Problem #2.7. A refactoring solution that fixes all code-smells is not always the optimal one due to the high code adaptation/modification needed. When applying refactoring, different code changes are performed. The amount of code changes corresponds to the number of code elements (e.g., classes, methods, fields, relationships, field references, etc.) modified through adding, deleting, or moving operations. Minimizing code changes when recommending refactoring is very important to help developers understand the modified/improved design. Moreover, most developers want to keep as much as possible the original design structure when fixing code-smells [17]. Hence, improving software quality and reducing code changes are conflicting. In some cases, correcting some code-smells corresponds to performing substantial changes in the system or is, sometimes, equivalent to re-implementing a large part of the system.

Problem #2.8. In general, refactoring restructures a program to improve its structure without altering its external behavior. However, it is challenging to preserve the semantic coherence of a program when refactoring is decided/implemented automatically. Indeed, a program could be syntactically correct, and have the right behavior, but incorrectly model the domain semantics. We need to preserve the rationale behind why and how code elements are grouped and connected when applying refactoring operations to improve code quality.

Problem #2.9. The majority of the existing work did not consider the history of changes applied in the past when performing refactorings. However, the history of code changes can be helpful in increasing the correctness of new refactoring solutions. To better guide the search process, recorded code changes applied in the past can be reused in similar contexts. This knowledge can be combined with structural and semantic information to improve the automation of refactoring. Moreover, it is important to maintain the consistency with prior changes when recommending new changes, i.e., refactorings.

Problem #2.10. Most of the existing studies focus mainly on fixing code-smells and/or improving some software metrics. However, this may not be sufficient to make the source code easier to understand and to modify. Introducing design patterns that represent good design practices can greatly improve the quality of systems. Nevertheless, very few works exploit the richness of refactoring to introduce design patterns.

All of these observations are at the origin of the work conducted in this thesis. In the next section, we give an overview of our research directions to address the above-mentioned problems.

1.3 Research objectives and main contributions

1.3.1 Objectives

The main objectives of this thesis are the following:

1. By applying search-based software engineering (SBSE) techniques, we can automate the refactoring recommending task. SBSE has been shown to be a practical and efficient way in solving several software engineering problems [91]. Software refactoring problem is, by its nature, ideal for the application of SBSE techniques, in its two steps (1) identification code fragments that need to be refactored, and (2) identification of the suitable refactoring operations to apply.
2. Automating the code-smells' detection task to support developers and relieve them from the burden of doing so manually. Developers no longer need to manually define

rules/constraints to automate code-smells' detection task. Instead, they simply provide a set of examples of code-smells that are already detected in different software systems.

3. Automating the refactoring recommending task. The aim of this thesis is to circumvent the problems mentioned in previous section. The majority of existing work deals with refactoring from a single perspective which is improving the quality. In this thesis, we formulate the refactoring recommending problem as a multi-objective optimization problem to find the best compromise between different objectives: improving the quality, preserving semantic coherence, reducing the number of changes, and maintaining the consistency with development/maintenance history.
4. Take advantage of the richness of refactoring through a multi-objective approach to introduce “good” design practices (i.e., design patterns), fix “bad” design practices (i.e., code-smells), while improving software quality attributes (i.e., flexibility, maintainability, etc.).

1.3.2 Contributions

To overcome the previously identified problems, we propose the following contributions, organized into three major parts (cf. Figure 1.1):

Part 1: Code-smells detection

Contribution 1.1: Search-based code-smells detection

To automate the detection of code-smells we propose a search-based approach [28] using genetic algorithm to automatically generate detection rules. Our proposal consists of using knowledge from previously manually inspected projects (i.e., code-smell examples) in order to detect code-smells that will serve to generate new detection rules based on the combinations of quality metrics and threshold values. As illustrated in Figure 1.1, the detection process takes as inputs a base (*i.e.*, a set) of code-smell examples and takes as controlling parameters a set of quality metrics (the usefulness of these metrics

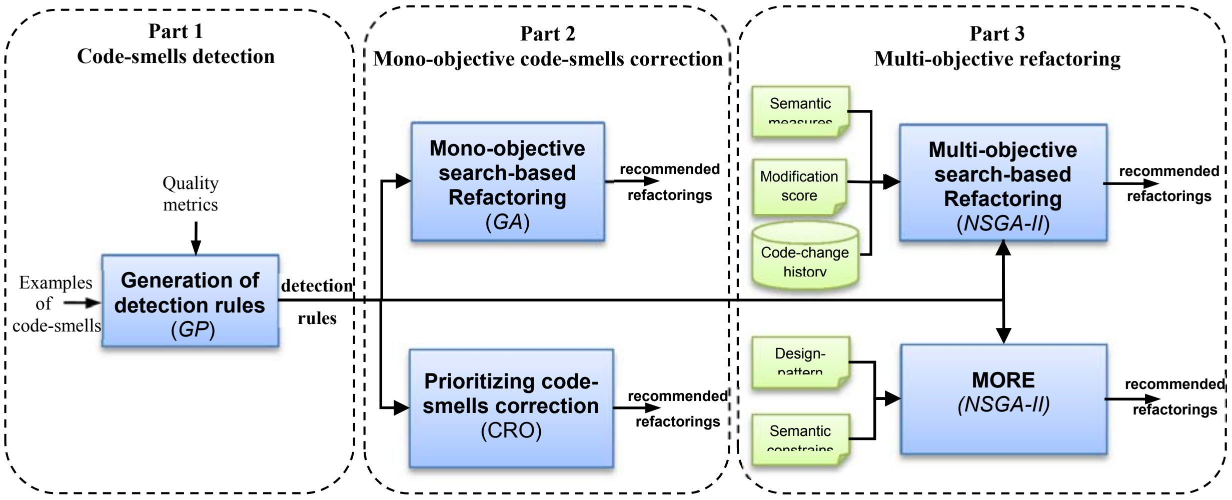


Figure 1.1 - Thesis contributions

was defined and discussed in the literature [30]). This step generates a set of code-smells detection rules. Consequently, a solution to the code-smell detection problem is represented as a set of rules that best detect the code-smells presented on the base of examples with high precision and recall.

Part 2: Mono-objective code-smells correction

Contribution 2.1: Mono-objective search-based code-smells correction

To fix the detected code-smells, we need to find the suitable refactoring solution. As a first contribution, we consider the process of generating correction solutions as a single-objective optimization problem. A correction solution is defined as a combination of refactoring operations that should minimize, as much as possible, the number of detected code-smells using the detection rules. To this end, we use genetic algorithm (GA) [52] to find and recommend the best combination of refactoring operations from a large list of available refactorings (i.e., the suitable metrics and their appropriate threshold values). Indeed, one of the advantages of our approach is that it does not correct code-smells separately since we consider the correction task as a global process instead of local one. In addition, we don't need to define an exhaustive list of code-smells and specify standard refactoring for each code-smell type. We evaluate the efficiency of our approach in finding potential code-smells in five different open-source systems.

Contribution 2.2: Prioritizing code-smells correction

We extend our previous contribution to prioritize the correction of code-smells. We propose an approach that supports automated refactoring recommendation for correcting code-smells where riskiest code-smells are prioritized during the correction process. Hence, we formulated the refactoring recommending problem as an optimization problem to find the near-optimal sequence of refactorings from a huge number of possible refactorings according to a prioritization schema. To this end, we used a novel metaheuristic search by the means of Chemical Reaction Optimization (CRO) [163], a newly established metaheuristics, to find the suitable refactoring solutions (i.e., sequence of refactoring operations) that maximize the number of corrected code-smells while prioritizing the most important, e.g., riskiest, and severest code fragments according to the developer's preferences.

Part 3: Multi-objective refactoring recommending

Contribution 3.1. A Multi-objective approach for recommending software refactoring

In this contribution, we deal with the refactoring recommending task as a multi-objective optimization problem. The process aims at finding the optimal sequence of refactoring operations that improve the software quality by minimizing the number of detected code-smells. In addition, we explore other objectives to optimize: reduce the number of modifications/adaptations needed to apply refactorings, preserve the semantic coherence of the refactored program, and maintaining the consistency with development/maintenance history. The idea is to find the best compromise between all of these objectives. Hence, by reducing the number of modifications, we reduce the complexity of the recommended refactorings and keep as much as possible the original design/code structure. Moreover, it is mandatory to preserve the semantic coherence and prevent arbitrary changes on code elements, especially when refactoring are decided automatically. Furthermore, historical data in software engineering provide a lot of solid knowledge that can be used to make sound data-driven decisions for several software engineering problems [152] [153]. Reuse is a common practice for developers during software development to save time and efforts. We show in this contribution, how recorded/historical code changes could be an effective way to propose new refactoring solutions. We evaluate the efficiency of our approach using a benchmark of six different industrial size open-source systems, and six frequent code-smells types through an empirical study conducted with software engineers.

Contribution 3.2. A multi-objective refactoring approach to introduce design patterns and fix code-smells

From another perspective, we propose a multi-objective formulation of refactoring recommending task where we consider the introduction of design patterns. We propose, in this contribution, a recommending framework for automated multi-objective refactoring to (1) introduce design patterns, (2) fix code-smells, and (3) improve design quality (as defined by software quality metrics). To evaluate our approach, we conducted a quantitative and qualitative evaluation with software engineers using a benchmark composed of four open source systems. The obtained results confirm the efficiency of our proposal compared to the state-of-the-art of refactoring techniques.

1.4 Thesis organisation

This thesis is organized as follows. Chapter 2 provides a review of the literature on previous research that is relevant to the main themes of this dissertation: code-smells detection and correction, software refactoring, existing search-based techniques for software refactoring, and the use of historical data in software engineering. Chapter 3 reports our contribution for the detection of code-smells. We present the use of genetic programming (GP) [99] and its adaptation for generating code-smells detection rules. In Chapter 4, we present our mono-objective search-based approach for fixing code-smells. The proposed approach uses an adaptation of the GA to find the suitable refactoring solutions that should be applied to fix code-smells. Chapter 5 presents an extension of the previous contribution to prioritize the correction code-smells using Chemical Reaction Optimization. In Chapter 6, we introduce our multi-objective approach using the Non-dominated Sorting Genetic Algorithm (NSGA-II) [24] to find the optimal refactoring solution to fix code-smells, preserve the semantic coherence, and maintain the consistency with the change history, while reducing as much as possible the amount of modifications needed to apply refactoring. In Chapter 7, we describe our multi-objective approach for introducing design patterns while fixing code-smells. Finally, Chapter 8 summarizes the contributions of the work presented in this thesis, underlines its main limitations, and describes our future research directions.

Chapter 2 : State of the art

2.1 Introduction

This chapter provides a literature review on research work related to this thesis. We first provide the background material that is required to understand this thesis. Then, we survey the related work that is relevant to the main themes of this research work. In particular, the related work can be divided broadly into five research areas: (1) detection of code-smells, (2) management and prioritization of code-smells, (3) refactoring and correction of code-smells, (4) recommendation systems in software engineering, and (5) the use of historical data in software engineering.

This chapter is structured as follows. Section 2.2 presents the background need for unfamiliar readers with code-smells, software refactoring, and search-based software engineering. We present in Section 2.3 different metaheursitcs. Section 2.4 summarises exiting works in code-smells detection. We classify existing detection strategies into mainly seven classes. Section 2.5 describes existing research work on prioritizing and managing code-smells. Section 2.6 discusses the state of the art of software refactoring and code-smells correction; Section 2.7 is devoted to describe recommendation systems in software engineering including software refactoring and their usefulness. In Section 2.8, we provide a description of research work on mining software repository and their role to improve software design/reuse and support the maintenance of software systems. In Section 2.9, we conclude this chapter with a discussion on the limitations of the presented work with regard to our thesis.

2.2 Background

In this section, we provide the necessary background for code-smells, software refactoring and search-based software engineering. This section is aimed at readers who are unfamiliar with these concepts.

2.2.1 Code smells

Code-smells, also called in the literature anti-patterns [1], anomalies [12], design flaws [10] or bad smells [2], are a metaphor to describe problems resulting from bad design and programming practices. Along of this dissertation, we will use the term code-smell. Originally, code-smells are used to find the places in software that could benefit from refactoring. According to Fowler [1], code-smells are unlikely to cause failures directly, but may do it indirectly. In general, they make a system difficult to change, which may in turn introduce bugs. Different types of code-smells, presenting a variety of symptoms, have been studied in the intent of facilitating their detection [8] and suggesting improvement solutions. Most of code-smells identify locations in the code that violate object-oriented design heuristics, such as the situations described by Riel [70] and by Coad et al. [73]. Code-smells are not limited to design flaws since most of them occur in code and are not related to the original design. Indeed, most of code-smells can emerge during the maintenance and evolution of a system.

In [1], Beck defines 22 sets of symptoms of code-smells and proposes the different possible refactoring solutions to improve the system design. These include God classes, feature envy, long parameter lists, and data classes. Each code-smell type is accompanied by refactoring suggestions to remove it. Van Emden and Moonen [71] developed, as far as we know, the first automated code-smell detection tools for Java programs. Mantyla studied the manner of how developers detect and analyse code smells [72]. Previous empirical studies have analysed the impact of code-smells on different software maintainability factors including defects [74] [75] [76] and effort [77] [78]. In fact, software metrics (quality indicators) are sometimes difficult to interpret and suggest some actions (refactoring) as noted by Anda et al. [79] and Marinescu et al. [10]. Code-smells are associated with a generic list of possible refactorings to improve the quality of software systems. In addition, Yamashita et al. [67] [68] show that the different types of code-smells can cover most of maintainability factors [69]. Thus, the detection of code-smells can be considered as a good alternative of the traditional use of quality metrics to evaluate the quality of software products. Brown et al. define another category of code-smells that are documented in the literature, and named anti-patterns.

2.2.2 Refactoring

To fix code-smells, one of the most-used techniques is refactoring which improves design structure while preserving the external behavior [12]. Refactoring has been practiced by programmers for many years. Refactoring is widely recognized as a crucial technique applied when evolving object-oriented software systems. More recently, tools that (semi-) automate the process of refactoring has emerged in various programming environments such as Eclipse [53] and Netbeans [54]. These tools have promised, if applied well, to increase the speed at which programmers can write and maintain code while decreasing the likelihood that programmers will introduce new bugs [86] [87].

Opdyke and Johnson [1] defined refactoring as the process of improving a code after it has been written by changing its internal structure without changing the external behavior. The idea is to reorganize variables, classes and methods in order to facilitate future extensions. This reorganization is used to improve different aspects of software-quality such as reusability, maintainability, complexity, etc. [1] [18]. Later, it was popularized by Martin Fowler's book [1], but refactoring has been practiced for as long as programmers have been writing programs. Fowler's book is largely a catalog of refactorings [25]; each refactoring captures a specific structural change that has been observed frequently in different programming languages and application domains.

Roughly speaking, we can identify two distinct steps in the refactoring process: (1) detect when a program should be refactored and (2) identify which refactorings should be applied and where [18]. For example, after detecting a Blob code-smell, many refactoring operations can be used to reduce the number of functionalities in a specific class, such as move methods/fields and extract class. An exhaustive list of refactoring operations can be found in [25].

2.2.3 Search-based software engineering

The research topic of this Ph.D. thesis is about using search techniques in software engineering, called search-based software engineering (SBSE). The term SBSE was first used by Harman and Jones in 2001 and defined as the application of search-based approaches to solving optimization problems in software engineering [91]. Essentially, SBSE is based on the idea of reformulating software engineering problems as search problems by defining them in terms of solution representation, fitness function and solution change operators. Once a software engineering task is framed as a search problem, there are many metaheuristic techniques that can be applied to discover near optimal solutions to that problem.

In 2001, Harman and Jones [91] expected to see a dramatic development of the field of search based-software engineering. The authors have concluded that in the near future metaheuristic search will be applied to several areas of software engineering. As expected, SBSE has become a growing research and practice domain which is popularized in both academia and industry [90]. Indeed, in the last decade many SBSE approaches has been applied to a wide variety of software engineering problems, including software testing [94], requirements engineering [93], bug fixing [92], project management [95], refactoring [20], service-oriented software engineering [96], and model-driven software engineering [164]. The most studied and known models are based on classic evolutionary algorithms (EAs) such as simulated annealing (SA) [98], genetic algorithm (GA) [99], particle swarm optimization (PSO) [100], and tabu search (TS) [92].

We will investigate in this thesis the use of SBSE techniques for automating the detection and correction of code-smells as well as automatically recommending refactoring.

2.3 Metaheuristic search techniques

Different mono- and multi-objective metaheuristic techniques are used in this thesis. We provide in this section the necessary background for unfamiliar readers with metaheursitics. More specifically, we used the following metaheuristics: Genetic Algorithm, Genetic Programming, Chemical Reaction Optimization, and Non-dominated Sorting Genetic Algorithm.

2.3.1 Genetic Algorithm

Genetic Algorithm (GA) [99] is a powerful heuristic search optimization method inspired by the Darwinian theory of evolution. The basic idea is to explore the search space by making a population of candidate solutions, also called individuals, evolve toward a "good" solution of a specific problem.

In GA, an individual is usually string/vector of numbers that represents a candidate solution. Every individual of the population is evaluated by a fitness function that determines a quantitative measure of its ability to solve the target problem. The exploration of the search space is achieved by the evolution of candidate solutions using selection and genetic operators such as crossover and mutation. The selection operator insures selection of individuals in the current population proportionally to their fitness values, so that the fitter an individual is, the higher the probability is that it be allowed to transmit its features to new individuals by undergoing crossover and/or mutation operators. The crossover operator insures the generation of new children, or offspring, based on parent individuals. The crossover operator allows transmission of the features of the best fitted parent individuals to new individuals. This is usually achieved by replacing a randomly selected sub tree of one-parent individual with a randomly chosen sub tree from another parent individual to obtain one child. A second child is obtained by inverting parents. Finally, the mutation operator is applied, with a probability which is usually inversely proportional to its fitness value, to modify some randomly selected nodes in a single individual. The mutation operator introduces diversity into the population and allows escaping from local solutions found during the search.

Once the selection, mutation and crossover operators have been applied with given probabilities, the individuals in the newly created generation are evaluated using the fitness function. This process is repeated iteratively, until a stopping criterion is met. The criterion usually corresponds to a fixed number of generations, or when the fitness function reaches the desired fitness value. The result of GA (the best solution found) is the fittest individual produced along all generations.

2.3.2 Genetic Programming

Genetic Programming (GP) [171] is a branch of Genetic Algorithm. The main difference between genetic programming and genetic algorithm is the representation of the solution. Genetic algorithms create a string of numbers that represent the solution. Genetic programming creates (computer) programs as the solution which is usually represented as a tree, where the internal nodes are functions, and the leaf nodes are terminal symbols. Both the function set and the terminal set must contain elements that are appropriate for the target problem. For instance, the function set can contain arithmetic operators, logic operators, mathematical functions, etc., whereas the terminal set can contain the variables (attributes) of the target problem.

2.3.3 Chemical Reaction Optimization

Chemical reaction optimization (CRO) is a new recently proposed metaheuristics [176] inspired from chemical-reaction. It is not difficult to discover the correspondence between optimization and chemical reaction. Both of them aim to seek the global minimum (but with respect to different objectives) and the process evolves in a stepwise fashion. With this discovery, CRO was developed for solving optimization problems by mimicking what happens to molecules in chemical reactions. It is a multidisciplinary design which loosely couples computation with chemistry (see Table 2.1). The manipulated agents are molecules and each has a profile containing some properties. A molecule is composed of several atoms and characterized by the atom type, bond length, angle, and torsion. One molecule is distinct from another when they contain different atoms and/or different number of atoms. The term “molecular structure” is used to summarize all these characteristics and it corresponds to a solution in the matheuristic meaning. The representation of a molecular structure depends on the problem we are solving, provided that it can express a feasible solution of the problem. A molecule possesses two kinds of energies, i.e., potential energy (PE) and kinetic energy (KE). The former quantifies the molecular structure in terms of energy and it is modeled as the objective function value when evaluating the corresponding solution. A change in molecular structure (chemical reaction) is tantamount to switching to another feasible solution. CRO evolves a population of molecules by means of four chemical reactions called: (1) On-wall ineffective collision, (2) Decomposition, (3) Inter-molecular ineffective collision and (4) Synthesis. Consequently, similarly to genetic algorithm (GA), the molecule corresponds to the population individual and chemical reactions correspond to the variation operators. However, CRO is distinguished by the fact that environmental selection is performed by the variation operator. Differently to GA which generates an offspring population then makes a competition between the latter and the parent population, in CRO once an offspring is generated, it competes for survival with its parent(s) within the realization of the corresponding chemical reaction. Algorithm 2.1 illustrates the pseudocode of the CRO where it begins by initializing the different parameters that are:

- *PopSize*: the molecule population size,
- *KELossRate*: the loss rate in terms of Kinetic Energy (*KE*) during the reaction,
- *MoleColl*: a parameter varying between 0 and 1 deciding whether the chemical reaction to be performed is uni-molecular (on wall ineffective collision or decomposition) or mutli-molecular (inter-molecular ineffective collision or synthesis),
- *buffer*: the initial energy in the buffer,
- *InitialKE*: the initial KE energy,
- α, and β: two parameters controlling the intensification and diversification.

Once the initialization set is performed, the molecule population is created and the evolution process begins. The latter is based on the following four variation operators (elementary chemical reactions):

CRO pseudocode

```
1.  Input: Parameter values
2.  Output: Best solution found and its objective function value
3.  /*Initialization*/
4.  Set PopSize, KELossRate, MoleColl, buffer, InitialKE, $\alpha$, and $\beta$
5.  Create PopSize molecules
6.  /*Iterations*/
7.  While the stopping criteria not met do
8.     Generate $b \in [0, 1]$
9.     If ($b > MoleColl$) Then
10.       Randomly select one molecule $M_\omega$
11.       If (Decomposition criterion met) Then
12.          Trigger Decomposition
13.       Else
14.          Trigger OnwallIneffectiveCollision
15.       End If
16.    Else
17.       Randomly select two molecules $M_{\omega 1}$ and $M_{\omega 2}$
18.       If (Synthesis criterion met) Then
19.          Trigger Synthesis
20.       Else
21.          Trigger IntermolecularIneffectiveCollision
22.       End If
23.    End If
24.  Check for any new minimum solution
25.  End While
```

Algorithm 2.1 - Basic CRO pseudocode.

1) *On-wall ineffective collision*: This reaction corresponds to the situation when a molecule collides with a wall of the container and then bounces away remaining in one single unit. In this collision, we only perturb the existing molecule structure (which captures the structure of the solution) ω to ω'. This could be done by any neighborhood operator $N(\cdot)$.
2) *Decomposition*: It corresponds to the situation when a molecule hits a wall and then breaks into several parts (for simplicity, we consider two parts in this work). Any mechanism that can produce ω'_1 and ω'_2 from ω is allowed. The goal of decomposition

is to allow the algorithm to explore other regions of the search space after enough local search by the ineffective collisions.

3) *Inter-molecular ineffective collision*: This reaction takes place when multiple molecules collide with each other and then bounce away. The molecules (assume two) remains unchanged before and after the process. This elementary reaction is very similar to the uni-molecular ineffective counterpart since we generate ω'_1 and ω'_2 from ω_1 and ω_2 such that $\omega'_1 = N(\omega_1)$ and $\omega'_2 = N(\omega_2)$. The goal of this reaction is to explore several neighborhoods simultaneously each corresponding to a molecule.
4) *Synthesis*: This reaction is the opposite of decomposition. A synthesis happens when multiple (assume two) molecules hit against each other and fuse together. We obtain ω' from the fusion of ω_1 and ω_2. Any mechanism allowing the combination of solutions is allowed, where the resultant molecule is in a region farther away from the existing ones in the solution space. The idea behind synthesis is diversification of solutions.

Chemical meaning	**Metaheuristic meaning**
Molecular structure	Solution
Potential energy	Objective function value
Kinetic energy	Measure of tolerance of having worse solutions
Number of Hits	Current total number of moves
Minimum structure	Current optimal solution
Minimum value	Current optimal function value
Minimum hit number	Number of moves when the current optimal solution is found

Table 2.1 - CRO analogy between chemical and metaheuristic meanings. The first column contains the properties of a molecule used in CRO. The second column shows the corresponding meanings in the metaheuristic.

To sum up, on-wall and inter-molecular collisions (ineffective collisions) emphasize more on intensification while decomposition and synthesis (effective collisions) emphasize more on diversification. This allows making a good trade-off between exploitation and exploration as the case of GA. The algorithm undergoes these different reactions until the satisfaction of the stopping criteria. After that, it outputs the best solution found during the overall chemical process.

It is important to note that the molecule in CRO has several attributes, some of which are essential to the basic operations, i.e.: (a) the *molecular structure* ω expressing the solution encoding of the problem at hand; (b) the *Potential Energy* (*PE*) corresponding to the objective function value of the considered molecule and (c) the *Kinetic Energy* (*KE*) corresponding to non-negative number that quantifies the tolerance of the system accepting a worse solution than the existing one (similarly to simulated annealing). The optional attributes are:

i) *Number of hits* (*NumHit*): When a molecule undergoes a collision, one of the elementary reactions will be triggered and it may experience a change in its molecular structure. *NumHit* is a record of the total number of hits (i.e. collisions) a molecule has taken.
ii) *Minimum Structure* (*MinStruct*): It is the ω with the minimum corresponding *PE* which a molecule has attained so far. After a molecule experiences a certain number of collisions, it has undergone many transformations of its structure, with different corresponding *PE*. *MinStruct* is the one with the lowest *PE* in its own reaction history.
iii) *Minimum Potential Energy (MinPE)*: When a molecule attains its *MinStruct*, *MinPE* is its corresponding *PE*.
iv) *Minimum Hit Number* (*MinHit*): It is the number of hits when a molecule realizes *MinStruct*. It is an abstract notation of time when *MinStruct* is achieved.

For more details about the role of each of these attributes in CRO, the reader is invited to refer to [176].

The CRO has been recently applied successfully to different combinatorial and continuous optimization problems [179] [180] [181]. Several nice properties for the CRO have been detected. These properties are as follows:

- The CRO framework allows deploying different operators to suit different problems.
- Its variable population size allows the system to adapt to the problems automatically; thereby minimizing the number of required function evaluations.

- Energy conversion and energy transfer in different entities and in different forms make CRO unique among metaheursitics. CRO has the potential to tackle those problems which have not been successfully solved by other metaheuristics.
- Other attributes can easily be incorporated into the molecule. This gives flexibility to design different operators.
- CRO enjoys the advantages of both SA and GA.
- CRO can be easily programmed in object-oriented programming language, where a class defines a molecule and methods define the elementary reactions.

Based on all these observations, the CRO seems to be an interesting metaheuristic ready to use for tackling SE problems.

2.3.4 Non-dominated Sorting Genetic Algorithm

The basic idea of the Non-dominated Sorting Genetic Algorithm (NSGA-II) [24] is to make a population of candidate solutions evolve toward the near-optimal solution in order to solve a multi-objective optimization problem. NSGA-II is designed to find a set of near-optimal solutions, called non-dominated solutions or the Pareto front. A non-dominated solution is one that provides a suitable compromise between all objectives without degrading any of them. As described in Algorithm 2.2, the first step in NSGA-II is to create randomly a population P_0 of individuals encoded using a specific representation (line 1). Then, a child population Q_0 is generated from the population of parents P_0 using genetic operators such as crossover and mutation (line 2). Both populations are merged into a new population R_0 of size N (line 5).

Fast-non-dominated-sort is the algorithm used by NSGA-II to classify individual solutions into different dominance levels. Indeed, the concept of Pareto dominance consists of comparing each solution x with every other solution in the population until it is dominated by one of them. If no solution dominates it, the solution x will be considered non-dominated and will be selected by the NSGA-II to be a member of the Pareto front. If we consider a set of objectives f_i , $i,j \in 1 \ldots n$, to maximize, a solution x dominates x'

$$iff\ \ \forall i, f_i(x') \le f_i(x) \text{ and } \exists j \mid f_j(x') < f_j(x).$$

The whole population that contains N individuals (solutions) is sorted using the dominance principle into several fronts (line 6). Solutions on the first Pareto-front F_0 get assigned dominance level of 0 Then, after taking these solutions out, fast-non-dominated-sort calculates the Pareto-front F_1 of the remaining population; solutions on this second front get assigned dominance level of 1, and so on. The dominance level becomes the basis of selection of individual solutions for the next generation. Fronts are added successively until the parent population P_{t+1} is filled with N solutions (line 8). When NSGA-II has to cut off a front F_i and select a subset of individual solutions with the same dominance level, it relies on the crowding distance to make the selection (line 9). This parameter is used to promote diversity within the population. This front F_i to be split, is sorted in descending order (line 13), and the first (N-$|P_{t+1}|$) elements of F_i are chosen (line 14). Then a new population Q_{t+1} is created using selection, crossover and mutation (line 15). This process will be repeated until reaching the last iteration according to the stop criteria (line 4).

```
1.  Create an initial population $P_0$
2.  Generate an offspring population $Q_0$
3.  $t$=0;
4.  while stopping criteria not reached do
5.      $R_t = P_t \cup Q_t$;
6.      $F$ = fast-non-dominated-sort ($R_t$);
7.      $P_{t+1} = \emptyset$ and $i$=1;
8.      while $| P_{t+1}| + |F_i| \le N$ do
9.          Apply crowding-distance-assignment($F_i$);
10.         $P_{t+1} = P_{t+1} \cup F_i$ ;
11.         $i = i$+1;
12.     end
13.     Sort($F_i$, $\prec n$);
14.     $P_{t+1} = P_{t+1} \cup F_i[1 : (N\text{-}| P_{t+1} |)]$;
15.     $Q_{t+1}$ = create-new-pop($P_{t+1}$);
16.     $t = t$+1;
17. end
```

Algorithm 2.2 - High-level pseudo-code of NSGA-II.

2.4 Detection of code-smells

There has been much research effort focusing on the study of code-smells. Existing approaches for code-smells detection can be classified into six broad categories: manual approaches, symptom-based approaches, rule-based approaches, probabilistic approaches, machine-learning based approaches, and visualization-based approaches.

2.4.1 Manual approaches

In the literature, the first book that has been specially written for design smells was by Brown et al. [12] which provide broad-spectrum and large views on design smells, and antipatterns that aimed at a wide audience for academic community, as well as in industry. Indeed, in [1], Fowler and Beck have described a list of design smells which may possibly exist on a program. They suggested that software maintainers should manually inspect the program to detect existing design smells. In addition, they specify particular refactorings for each code-smells type. Travassos et al. [31] have also proposed a manual approach for detecting code-smells in object-oriented designs. The idea is to create a set of "reading techniques" which help a reviewer to "read" a design artifact for the purpose of finding relevant information. These reading techniques give specific and practical guidance for identifying code-smells in object-oriented designs. So that, each reading technique helps the maintainer focusing on some aspects of the design, in such way that an inspection team applying the entire family should achieve a high degree of coverage of the code-smells. In addition, in [32], another proposed approach is based on violations of design rules and guidelines. This approach consists of analyzing legacy code, specifying frequent design problems as queries and locating the occurrences of these problems in a model derived from the source code. However, the majority of the detected problems were simple ones, since it is based on simple conditions with particular threshold values. As a consequence, this approach did not address complex code-smells.

The main limitation of exiting manual approaches is that they are ultimately a human-centric process that requires a great human effort and extensive analysis and interpretation effort from software maintainers to find design fragments that corresponds to code-smells. In addition, these techniques are time-consuming, error-prone and depend on programs in their contexts. Another important issue is that locating code-smells manually has been described as more a human intuition than an exact science. To circumvent the above mentioned problems, some semi-automated approaches have emerged using different techniques.

2.4.2 Symptom-based detection

Van Emden and Moonen [71] presented one of the first attempts to automate code-smell detection for Java programs. The authors exanimated a list of code smells and found that each of them is characterized by a number of "smell aspects" that are visible in source code entities such as packages, classes, methods, etc. A given code smell is detected when all its aspects are found in the code. The identified aspects are mainly related to non-conformance to coding standards. The authors distinguish two types of smell aspects: primitive smell aspects that can be observed directly in the code, and derived smell aspects that are inferred from other aspects. An example of a primitive aspect is "method m contains a switch statement", an example of a derived aspect is "class C does not use any methods offered by its superclasses". The developed Java code-smell detection tool allows also visualization of the code and the detected smells. However, conformance to coding standards is not always easy to achieve in practice. Moreover, using such visualization tools, it is still difficult for a programmer to identify potential code-smells, and his decision is most of the time subjective.

Later, Moha et al. [8] proposed a description of anti-pattern symptoms using a domain-specific-language (DSL) for their anti-patterns detection approach called DECOR. They proposed a consistent vocabulary and DSL to specify anti-patterns based on the review of existing work on code-smells found in the literature. To describe anti-pattern symptoms different notions are involved, such as class roles and structures. Symptoms descriptions are later mapped to detection algorithms. However, converting symptoms into rules needs a significant analysis and interpretation effort to find the suitable threshold values. In addition, this approach uses heuristics to approximate some notions which results in an important rate of false positives. The proposed approach has been evaluated on only four well-known code-smells: the Blob, functional decomposition, spaghetti code, and Swiss-army knife because the literature provide obvious symptom descriptions on code-smells. Similarly, Munro [33] has proposed description and symptoms-based approach using a precise definition of bad smells from the informal descriptions given by the originators Fowler and Beck [1]. The characteristics of code-smells have been used to systematically define a set of measurements and interpretation rules for a subset of code-smells as a template form. This template consists of three main parts: a code smell name, a text-based description of its characteristics, and heuristics for its detection.

The most notable limitation of symptoms-based approaches is that there exists no consensus in defining symptoms or smell aspects. A code-smell may have several and different interpretations by a maintainer. Another limitation is that for an exhaustive list of code-smells, the number of possible smells to be manually described, characterized with rules and mapped to detection algorithms can be very large. Indeed, the background and knowledge of maintainers may influence their understanding of code-smells, given a set of symptoms. As a consequence, symptoms-based approaches are also considered as time-consuming, error-prone and subjective. Thus automating the detection of code-smells is still a real challenge.

2.4.3 Metric-based approaches

Most of automated code-smell detection techniques are based on software metrics. The idea to automate the problem of code-smells detection is not new, neither is the idea to use software metrics to evaluate or improve the quality of software systems. Marinescu [10] have proposed a mechanism called "detection strategy" for formulating metrics-based rules that capture deviations from good design principles and heuristics. Detection strategies allow to a maintainer to directly locate classes or methods affected by a particular code-smell. As such, Marinescu have defined detection strategies for capturing around ten important flaws of object-oriented design found in the literature. Later, Raiu and his colleagues [115], refined the original concept of detection strategy, by using historical information of the suspected code-smell structures. Using this approach, the authors showed how the detection of God Classes and Data Classes can become more accurate. The approach refines the characterisation of suspects, which lead to a twofold benefit.

After his suitable symptom-based characterization of code-smells, Munro [33] proposed metric-based heuristics for detecting code-smells, which are similar to Marinescu's detection strategies. Munro has also conducted an empirical study to justify his choice of metrics and thresholds for detecting smells. Salehie et al. [39] proposed a metric-based heuristic framework to detect and locate object-oriented design flaws similarly to those illustrated by Marinescu [10]. The detection technique is based on evaluating design quality of an object-oriented system through quantifying deviations from good design heuristics and principles by mapping these design flaws to class level metrics such as complexity, coupling and cohesion by defining rules. Erni et al. [13] introduce the concept of multi-metrics, as an n-tuple of metrics expressing a quality criterion (*e.g.*, modularity). Unfortunately, multi-metrics neither encapsulate metrics in a more abstract construct, nor do they allow a flexible combination of metrics. In [105], Fourati et al. proposed an approach that identifies and predicts anti-patterns in UML diagrams through the use of existing and newly defined quality metrics. Operating at the design level, the proposed approach examines structural and behavioral information through the class and sequence diagrams.

More recently, Palomba et al. [102] have proposed a new approach called HIST (Historical Information for Smell detection) to detect specific types of code-smell using a set of metrics derived from change history extracted from version control systems. Though revision histories often display changes at a file level granularity, they use a tool called the

Change History Extractor to parse changes at a method- and class-level granularity, and then they identify code-smells from the parsed logs using specific rules. However, the developers of HIST point out that not all code smells are possible to detect using just source code change history because only some are by definition characterized by how the code changes during the project development (e.g., Divergent Change, and Shotgun Surgery). Thus the approach is limited to few types of code smell and cannot be generalized. Moreover, the authors defines Blobs as classes modified (in any way) in more than a given percentage threshold of commits. Therefore, classes containing two methods can be detected as a Blob. The results may not always accurate. In fact, it is very important to look at the type of changes that are applied but actually HIST just count the number of commits without considering the type of change. For example, a class detected as Blob where more than 80% of changes applied are delete methods/attributes, or move method to another class, thus the class becomes a data or lazy class and not a Blob; however HIST detected it as a Blob.

In general, the effectiveness of combining metric/threshold is not obvious. That is, for each code-smell, rules that are expressed in terms of metric combinations need a significant calibration effort to find the fitting threshold values for each metric. Since there exists no consensus in defining code-smells, different threshold values should be tested to find the best ones.

2.4.4 Probabilistic approaches

Probabilistic approaches represent another way for detecting code-smells. Alikacem et al. [14] have considered the code-smells detection process as fuzzy-logic problem, using rules with fuzzy labels for metrics, e.g., small, medium, large. To this end, they proposed a domain-specific language that allows the specification of fuzzy-logic rules that include quantitative properties and relationships among classes. The thresholds for quantitative properties are replaced by fuzzy labels. Hence, when evaluating the rules, actual metric values are mapped to truth values for the labels by means of membership functions that are obtained by fuzzy clustering. Although, fuzzy inference allows to explicitly handle the uncertainty of the detection process and ranks the candidates, authors did not validate their approach on real programs. Recently, another probabilistic approach has been proposed by Khomh et al. [11] extending the DECOR approach [8], a symptom-based approach, to support uncertainty and to sort the code-smells candidates accordingly. This approach is managed by Bayesian belief network (BBN) that implement the detection rules of DECOR. The detection outputs are probabilities that a class is an occurrence of a code-smell type, *i.e.,* the degree of uncertainty for a class to be a code-smell. They also showed that BBNs can be calibrated using historical data from both similar and different context. More recently, Dimitrios et al. [106] explored the ways in which the anti-pattern ontology can be enhanced using Bayesian networks in order to reinforce the existing ontology-based detection process. Their approach allows software developers to quantify the existence of anti-patterns using Bayesian networks, based on probabilistic knowledge contained in the anti-pattern ontology regarding relationships of anti-patterns through their causes, symptoms and consequences.

Although in probabilistic approaches, the above-mentioned problems in Section 1.2 related to the use of rules and metrics/thresholds do not arise, it still suffers from the problem of selecting the suitable metrics to conduct a detection process.

2.4.5 Machine learning based approaches

Machine learning represents another alternative for detecting code-smells. Catal et al. [36] used different machine learning algorithms to predict defective modules. They investigated the effect of dataset size, metrics set, and feature selection techniques for software fault prediction problem. They employed several algorithms based on artificial immune systems (AIS). Kessentini et al. [35] have proposed an automated approach for discovering code-smells. The detection is based on the idea that the more code deviates from good practices, the more likely it is bad. Taking inspiration from AIS, this approach learns from examples of well designed and implemented software elements, to estimate the risks of classes to deviate from “normality”, *i.e.,* a set of classes representing “good” design that conforms to object-oriented principles. Elements of assessed systems that diverge from normality to detectors are considered as risky. Although this approach succeeded in discovering risky code, it does not provide a mechanism to identify the type of the detected code-smell. Similarly, Hassaine et al. [38] have proposed an approach for detecting code-smells using machine learning technique inspired from the AIS. Their approach is designed to systematically detect classes whose characteristics violate some established design rules. Rules are inferred from sets of manually-validated examples of code-smells reported in the literature and freely-available. Recently, Maiga et al. [103] [104] introduced an approach called SMURF to detect anti-patterns, based on a machine learning technique. SMURF is based on SVM (support vector machines) using polynomial kernel and take into account practitioners' feedback. The proposed approach takes as input a training dataset that contains classes derived from object-oriented systems including instances of code-smells. The approach calculates object-oriented metrics that will be used as the attributes for each class in the dataset during the learning process.

The major benefit of machine-learning based approaches is that they do not require great experts' knowledge and interpretation. In addition, they succeeded to some extent, to detect and discover potential code-smells by reporting classes that are similar (even not identical) to the detected code-smells. However, these approaches depend on the quality and the efficiency of data, *i.e.,* code-smell instances, to learn from. Indeed, the high level of false positives represents the main obstacle for these approaches. Moreover, the selection of the suitable metrics for the learning process is a difficult task and is still a subjective decision.

2.4.6 Visualization-based approaches

The high rate of false positives generated by the above mentioned approaches encouraged other teams to explore semi-automated solutions. These solutions took the form of visualization-based environments. The primary goal is to take advantage of the human capability to integrate complex contextual information in the detection process. Kothari et al. [15] present a pattern-based framework for developing tool support to detect software anomalies by representing potential code-smells with different colors. Dhambri et al. [16] have proposed a visualization-based approach to detect design anomalies by automatically detecting some symptoms and letting others to human analyst. The visualization metaphor was chosen specifically to reduce the complexity of dealing with a large amount of data. Although visualization-based approaches are efficient to examine potential code-smells on their program and in their context, they do not scale to large systems easily. In addition, they require great human expertise and thus they are still time-consuming and error-prone strategies. Moreover, the information visualized is mainly metric-based, meaning that complex relationships can be difficult to detect. More recently, Murphy-Hill et al. [101] have proposed a smell detector tool called Stench Blossom that provides an interactive ambient visualization designed to first give programmers a quick, high-level overview of the smells in their code, and then, if they wish, to help in understanding the sources of those code smells. Indeed, since visualization approaches and tools such as Stench Blossom [101], VERSO [34] are based on manual and human inspection, they still, not only, slow and time-consuming, but also subjective.

Although these approaches have contributed significantly to automate the detection of code-smells, none have presented a complete and fully automated technique. Detecting code-smells is still, to some extent, a difficult, time-consuming, and manual process [9]. Indeed, the number of code-smells typically exceeds the resources available to address them. In many cases, mature software projects are forced to ship with both known and unknown code-smells for lack of development resources to deal with every code-smell.

2.4.7 Code-smell detection tools

Different tools for code-smell detection have been developed as research prototypes or commercial tools using different detection techniques. The detection techniques are usually based on the computation of a particular set of combined metrics [10], standard object-oriented metrics or metrics defined for the smell detection purpose. For instance, JDeodorant [83] is a code-smell detection tool implemented as an Eclipse plugin that automatically identifies four types of code-smells (Feature Envy, God Class, Long Method, and Type Checking) in Java object oriented programs. JDeodorant is based on the evaluation of a set of software metrics to identify possible code-smells. Moreover, JDeodorant provides a list of possible refactorings according to the detected code-smell.

iPlasma [81] [107] is an integrated platform for quality assessment of object-oriented systems that includes support for all the necessary phases of analysis: from model extraction up to high-level metrics based analysis, or detection of code duplication. iPlasma is designed to detect several code smells called disharmonies, such as Brain Class Brain Method, Data Class, Dispersed Coupling, Feature Envy, and God Class.

InFusion [82] supports the analysis, diagnosis quality improvement of a system at the architectural, as well as at the code level and covers all the necessary phases of the analysis process. InFusion allows to detect more than 20 code smells, such as Code Duplication, classes that break encapsulation (Data Class, God Class), methods and classes that are heavily coupled or ill designed class hierarchies and other code smells (Cyclic Dependencies, Brain Method, Shotgun Surgery). InFusion has its roots in iPlasma, and then extended with more functionalities. InCode [110] has been developed by the same team of inFusion and is very similar to Infusion. InCode is an Eclipse plugin that provides continuous detection of design problems (i.e. problems are detected as code is written) complementing thus the code reviews, which can be performed with other tools.

PMD [108], is another tool that scans Java source code and looks for potential problems like: possible bugs, such as dead code, empty try/catch/finally/switch statements, unused local variables, parameters and duplicate code. Moreover, PMD is able to detect three smells (Large Class Long Method Long Parameter List) and allows setting the thresholds values for the metrics.

Stench Blossom is a visualization-based code-smells detection tool developed by Murphy et al. [101] and implemented as an Eclipse plugin. StenchBlossom provides an interactive ambient visualization designed to first give programmers a quick, high-level overview of the smells in their code, and then, to help understand the sources of the code-smells. It does not provide numerical values, but only a visual threshold: the size of a petal is directly proportional to the entity of the code-smells. However, the only possible procedure to find code-smells is to manually inspect the source code, looking for a petal whose size is big enough to assume that there is a code smell. Stench Blossom provides the programmer with three different views, which progressively offer more information about the smells in the code being visualized.

CheckStyle [109] has been developed to help programmers write Java code that adheres to a coding standard. It is able to detect duplicate code and three other smells, Long Method, Long Parameter List and Large Class. DÉCOR [8] is implemented as a BlackBox, allows the specification and automatic detection of code and design smells such as Large Class, Lazy Class, Long Method, Long Parameter List. DÉCOR uses a symptom-based approach as described in Section 2.4.2.

In fact, in all these tools there is no consensus on the detection of code smells. For instance, the code-smell large class detected by Stench Blossom and PMD is different from Large class recognized by DECOR. Indeed, Stench Blossom and PMD simply concern Large Class as a class with many lines of code, whereas DECOR considers both the size in terms of number of methods and attributes and the cohesion of the class. There are also remarkable differences concerning the number of classes and methods reported by each tool [111].

Despite the latest advances on automated code-smell detection approaches and tools, each has its limitations, and still there are no answers to address different detection problems that we underlined in Section 1.2.1. It is also not clear how much additional effort is required to interpret the results from the automated detection of code smells to decide optimally which code-smells should be prioritized over others.

2.5 Management and prioritization of code-smells

Studies that consider the management and prioritization of code smells have emerged recently. That is, in practice, not all code-smells have equal effects/importance. Each individual instance has its severity score that allows designers to immediately spot and fix the most critical instances of each code-smell. Concretely, the same code-smell type can occur in different code fragments but with different impact scores on the system design [81] [82].

The first tool is Code Metrics [122], a .NET based add-in for the Visual Studio development environment that is able to calculate a set of metrics. Once the metrics are calculated, the tool assigns a "maintainability index" score to each of the analyzed code elements. This score is based on the combination of these metrics for each code element. The second tool is inFusion tool [82] that provides a "*severity*" index to help software engineer in classifying and understanding code-smell harmfulness. This severity index is defined by R. Marinescu [114] as: "Severity is computed by measuring how many times the value of a chosen metric exceeds a given threshold". The severity index takes into consideration size, encapsulation, complexity, coupling and cohesion metrics. However, the use of only metrics and thresholds is not always sufficient to understand the harmfulness of code-smells. Other aspects should be considered to better understand the impact and the harmfulness of code-smells such as the change history, the context, and the characteristics of the smell, etc. For instance, if a code-smell (e.g., Blob) is created intentionally and remains unmodified or hardly undergo changes, the system may not experience any problems [63]. Classes participating in code/design problems (e.g., code-smells) are significantly more likely to be subject to changes and to be involved in fault-fixing changes (bugs) [118]. Using history information, Raiu et al. [115] succeeded in eliminating false positives code-smells in their detection approach by filtering out the harmless suspects from those provided by a single-version detection strategy. Their approach allows also the identification of most dangerous suspects by using additional information on the evolution of initial suspects over their analyzed history. However, the proposed approach is limited only on God and Data Class code-smells. In [112], Arcelli et al. proposed an approach called JCodeOdor to filter and prioritize code-smells. To this end they defined an index called Code Smell Harmfulness to approximate how harmful each code smell is. The idea behind the Harmfulness computation is the need to have a way to prioritize the code smells, taking into account the characteristics of the smell, captured by metrics used in the detection strategy. The Harmfulness computation is strictly joined with the threshold computation, and relies on the metrics distribution. However, still using only metrics and thresholds to quantify the harmfulness of code-smells is not enough, other aspects should be considered. More recently, Arcoverde, et al. [121] present and evaluate four different heuristics for helping developers to prioritize code-smells, based on their potential contribution to the software architecture degradation. Those heuristics exploit different characteristics of a software project, such as change-density and error-density, for automatically ranking code elements that should be refactored more promptly according to their potential architectural relevance. The goal is to support software maintainers by the recommended rankings for identifying which code anomalies are harming architecture the most, helping them to invest their refactoring efforts into solving architecturally relevant problems.

Other management and prioritization approaches focus on specific code-smells such as Duplicated code (also called code clones). In [119] [123], Zibran and his colleagues introduced an approach to schedule prioritized code clone refactoring. They capture the risks of refactoring in a priority scheme. To this end, they proposed an effort model for estimating the effort required to refactor code clones in object-oriented (OO) programs. Then, taking into account the effort model and a variety of possible constraints, they formulated the scheduling of code clone refactoring activities as a constraint satisfaction optimization problem (CSOP), and solve it by applying constraint programming (CP) technique that aims to maximize benefits while minimizing refactoring efforts. In [116], Duala-Ekoko et al. proposed a tool called CloneTracker, an Eclipse plug-in that provides support for tracking code clones in evolving software. They start from the assumption that the elimination of code clones through refactoring is not always practical, feasible or cost-effective. With CloneTracker, developers can specify clone groups they wish to track, and the tool will automatically generate a clone model that is robust to changes to the source code, and can be shared with other collaborators of the project. When future modifications intersect with tracked clones, CloneTracker will notify the developer, provide support to consistently apply changes to a corresponding clone region, and provide support for updating the clone model. In another contribution, Zibran et al. [117] developed a language independent matching engine (LIME), a tool for fast localization of all k-difference (edit distance) occurrences of one code fragment inside another. The developed tool is an IDE-based clone management system to flexibly detect, manage, and refactor exact and near-miss code clones using a k-difference hybrid suffix tree algorithm. However, these specific techniques are limited only on code clones and cannot be generalized for other code-smells.

To develop a generalized prioritization schema several aspects, such as the change frequency, the context, the severity and the relative risk, should be considered and combined in a suitable way to approximate the harmfulness of each code-smells. Thus a suitable prioritization strategy could help software maintainers in identifying which code-smells are harming software the most, helping them to invest their refactoring efforts into solving relevant problems.

2.6 Refactoring and code-smells correction

Several techniques and approaches have been proposed in the literature to support software refactoring. We classify existing refactoring approaches into three broad categories 1) manual and semi-automated approaches, 2) search-based approaches, and 3) automated approaches.

2.6.1 Manual and semi-automated approaches

In Fowler's book [1], a non-exhaustive list of low-level design problems in source code have been defined. For each design problem (i.e., smells), a particular list of possible refactorings are suggested to be applied by software maintainers manually. Indeed, in the literature, most of existing approaches are based on quality metrics improvement to deal with refactoring. Fowler's book is largely a catalog of refactorings [25]; each refactoring captures a structural change that has been observed repeatedly in various programming languages and application domains. To apply refactoring, programmers should take the time to examine and then select the suitable refactorings to apply continuously along the development and maintenance process. In this context, Fowler states: "*In almost all cases, I'm opposed to setting aside time for refactoring. In my view refactoring is not an activity you set aside time to do. Refactoring is something you do all the time in little bursts*". [1]

Sahraoui et al. [28] have proposed an approach to detect opportunities of code transformations (*i.e.*, refactorings) based on the study of the correlation between some quality metrics and refactoring changes. To this end, different rules are defined as a combination of metrics/thresholds to be used as indicators for detecting bad smells and refactoring opportunities. For each bad smell a pre-defined and standard list of transformations should be applied in order to improve the quality of the code. Another similar work is proposed by Du Bois et al. [41] who starts form the hypothesis that refactoring opportunities corresponds of those which improves cohesion and coupling metrics to perform an optimal distribution of features over classes. Du Bois et al. analyze how refactorings manipulate coupling and cohesion metrics, and how to identify refactoring opportunities that improve these metrics. However, this two approaches are limited to only some possible refactoring operations with few number of quality metrics. In addition, the proposed refactoring strategies cannot be applied for the problem of code-smells correction.

Moha et al. [42] proposed an approach that suggests refactorings using Formal Concept Analysis (FCA) to correct detected code-smells. This work combines the efficiency of cohesion/coupling metrics with FCA to suggest refactoring opportunities. However, the link between code-smells detection and correction is not obvious, which make the inspection difficult for the maintainers. Similarly, Joshi et al. [44] have presented an approach based on concept analysis aimed at identifying less cohesive classes. It also helps identify less cohesive methods, attributes and classes in one go. Further, the approach guides refactoring opportunities identification such as extract class, move method, localize attributes and remove unused attributes. In addition, Tahvildari et al. [43] 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. Other contributions are based on rules that can be expressed as assertions (invariants, pre and post-condition). The use of invariants has been proposed to detect parts of program that require refactoring by [50]. In addition, Opdyke [17] have proposed the definition and the use of pre- and post-condition with invariants to preserve the behavior of the software when applying refactoring. Hence, behavior preservation is based on the verification/satisfaction of a set of pre and post-condition. All these conditions are expressed in terms of rules.

Furthermore, there are few research works that focus on the automation of design patterns introduction using refactoring. One of the earliest works to introduce design patterns was that of Ó Cinnéide and Nixon [124] [127] who presented a methodology for the development of design pattern transformations in a behavior preserving fashion. The identified a number of "pattern aware" composite refactorings called mini-transformations that, when composed, can create instances of design patterns. They defined a starting point for each pattern transformation, termed a precursor. This is where the basic intent of the pattern is present in the code, but not in its most flexible pattern form. However, the proposed approach is currently adapted only the Visitor design pattern. Later, Jensen and Cheng [126] have proposed the first an approach that supports composition of design changes and makes the introduction of design patterns a primary goal of the refactoring process. They used genetic programming, software metrics, and the set of mini-transformations identified by Ó Cinnéide and Nixon [127] to identify the most suitable set of mini-transformations to maximize the number of design patterns in a software design.

Roberts et al. [128] use sequences of basic refactoring operations to introduce design patterns in existing programs, including the Visitor pattern. Their approach was implemented within the Smalltalk Refactoring Browser. The approach was semi-automated, thus one of the key design criteria was to create a tool that could refactor Smalltalk programs with the same interactive style that Smalltalk developers are used to. In [18], Mens and Tourwé presented an approach to transform class hierarchy into a Visitor pattern. The approach is presented as a pseudo-algorithm that show how the introduction of a Visitor design pattern can be applied starting from a given point. The pseudo-algorithm describes six steps to apply the Visitor. However, the proposed pseudo-algorithm are not described and formulated in an automated way. Recently, Ajouli et al. [125] have described how to use refactoring tools (IntelliJ, and Eclipse) to transform a Java program conforming to the Composite design pattern into a program conforming to the Visitor design pattern with the same external behavior, and vice versa. To this end, the authors have selected four common variations in the implementation of the Composite pattern and have studied how these variations reflect in the Visitor pattern. For each variation, they have extended the previously defined transformation. The resulting transformations are automated and invertible.

The major limitation of these manual and semi-automated approaches is that they seek to apply refactorings separately without considering the whole program to be refactored and its impact on the other artifacts. Indeed, these approaches are limited to only some possible refactoring operations and few number of quality metrics to asses quality improvement. In addition, the proposed refactoring strategies cannot be applied for the problem of code-smells correction. Another important issue is that these approaches do not take into consideration the effort (i.e. the number of modifications/adaptations) needed to apply the suggested refactorings neither the semantics coherence of the refactored program.

2.6.2 Semantics preservation for software refactoring

Recently, there research works focusing on software refactoring have involved semantics preservation. For instance, Bavota et al. [45] have proposed an approach to automate the refactoring extract class based on graph theory that exploits structural and semantic relationships between methods. The proposed approach uses a weighted graph to represent the class to be refactored, where each node represents a method of that class. The weight of an edge that connects two nodes (representing methods) is a measure of the structural and semantic relationship between two methods that contribute to class cohesion. After that, they split the built graph in two sub-graphs, to be used later to build two new classes having higher cohesion than the original class. In [47], Baar et al. have presented a simple criterion and a proof technique for the semantic preservation of refactoring rules that are defined for UML class diagrams and OCL constraints. Their approach is based on formalization of the OCL semantics taking the form of graph transformation rules. However, their approach does not provide a concrete semantics preservation since there is no explicit differentiation between behaviour and semantics preservation. Hence, they consider that the semantics preservation "*means that the observable behaviors of original and refactored programs coincide*". Moreover, they use the semantics preservation in the model level with a high level of abstraction and therefore the code level and the implementation issues are not considered. In addition, this approach uses only the refactoring operation move attribute and do not consider an exhaustive list of refactorings [25]. Another semantics-based framework has been proposed by Logozzo [48] for the definition and manipulation of class hierarchies-based refactorings. The framework is based on the notion of observable of a class, *i.e.,* an abstraction of its semantics when focusing on a behavioral property of interest. They define a semantic subclass relation, capturing the fact that a subclass preserves the behavior of its superclass up to a given observed property.

The most notable limitation of the mentioned works is that the definition of semantic preservation is closely related to behaviour preservation. However, preserving the behavior does not mean that the semantic coherence of the refactored program is also preserved. Another issue is that the proposed techniques are limited to a small number of refactorings and thus it could not be generalized and adapted for an exhaustive list of refactorings. Indeed, the semantics preservation is still hard to define and ensure since the proposed approaches does not provide a pragmatic technique or an empirical study to prove whether the semantic coherence of the refactored program is preserved.

As far as semantics preservation issues, the above mentioned approaches does not provide a fully automated framework for automating the refactoring recommending task. Several studies have been focused on automating software refactoring recommending in recent years using different meta-heuristic search-based techniques for automatically searching for the suitable refactorings to be applied.

2.6.3 Search-based refactoring approaches

To automate refactoring activities, new approaches have emerged where search-based techniques have been used. These approaches cast the refactoring as an optimization problem, where the goal is to improve the design quality of a system based mainly on a set of software metrics. After formulating the refactoring as an optimization problem, several different techniques can be applied for automating refactoring, e.g., genetic algorithm, simulated annealing, and Pareto optimality, etc. Hence, we classify those approaches into two main categories: mono-objective and multi-objective optimization approaches.

In the first category, the majority of existing work combines several metrics in a single fitness function to find the best sequence of refactorings. Seng et al. [21] have proposed a single-objective optimization based-approach using genetic algorithm to suggest a list of refactorings to improve software quality. The search process uses a single fitness function to maximize a weighted sum of several quality metrics. The used metrics are mainly related to various class level properties such as coupling, cohesion, complexity and stability. Indeed, the authors have used some pre-conditions for each refactoring. These conditions serve at preserving the program behavior (refactoring feasibility). However, in this approach the semantic coherence of the refactored program is not considered. In addition, the approach was limited only on the refactoring operation move method. Furthermore, there is another similar work of O'Keeffe et al. [22] [23] that have used different local search-based techniques such as hill climbing and simulated annealing to provide an automated refactoring support. Eleven weighted object-oriented design metrics have been used to evaluate the quality improvements. In [49], Qayum et al. considered the problem of refactoring scheduling as a graph transformation problem. They expressed refactorings as a search for an optimal path, using Ant colony optimization, in the graph where nodes and edges represent respectively refactoring candidates and dependencies between them. However the use of graphs is limited only on structural and syntactical information and therefore does not consider the domain semantics of the program neither its runtime behavior.

Furthermore, Fatiregun et al. [59] [60] have proposed another search-based approach for finding program transformations to reduce code size and construct amorphous program slices. They apply a number of simple atomic transformation rules called axioms. Indeed, the authors presume that if each axiom preserves semantics then a whole sequence of axioms ought to preserve semantics equivalence. However, semantics equivalence depends on the program and the context and therefore it could not be always proved. Indeed, the semantic equivalence is based only on structural rules related to the axioms and there is no real semantic analysis has been performed. Moreover, they have used small atomic level transformations in their approach and their aim was to reduce program size rather than improving its structure/quality through refactoring. Otero et al. [61] use a new search-based refactoring. The main idea in this work is to explore the addition of a refactoring step into the genetic programming iteration. There will be an additional loop in which refactoring steps drawn from a catalogue of such steps will be applied to individuals of the population. By adding in the refactoring step the code evolved is simpler and more idiomatically structured, and therefore more readily understood and analysed by human programmers than that produced by traditional GP methods. Jensen and Cheng [126] have proposed the first search-based refactoring approach that supports composition of design changes and makes the introduction of design patterns a primary goal of the refactoring process. They used genetic programming, software metrics, and the set of mini-transformations identified by Ó Cinnéide and Nixon [127] to identify the most suitable set of mini-transformations to maximize the number of design patterns in a software design. However, maximizing the number of design patterns is not always profitable. That is applying a design pattern where it is not needed is highly undesirable as it introduces an unnecessary complexity to the system for no benefit. In addition one of the important limitations of this work is that the starting point to introduce a design pattern is not considered which may lead to arbitrary changes in the source code. That is the basic intent of the pattern should be present in the code. Furthermore, Kilic et al. [165] explore the use of a variety of population-based approaches to search-based parallel refactoring, finding that local beam search could find the best solutions. Recently, Zibran et al. [117] formulated the problem of scheduling of code clone refactoring activities as a constraint satisfaction optimization problem (CSOP) to fix known duplicate code code-smells. The proposed approach consists of applying constraint programming (CP) technique that aims to maximize benefits while minimizing refactoring efforts. An effort model is used for estimating the effort required to refactor code clones in object-oriented (OO) codebase. However, the proposed approach does not ensure the semantic coherence of the refactored program.

Although these approaches are powerful enough to improve software quality as expressed by software quality metrics, this improvement does not mean that they are successful in removing actual instances of code-smells. Moreover, combining several metrics/objectives into a single function may deteriorate the search process since one objective may dominate during the search.

In the second category of work, Harman et al. [20] have proposed the first search-based approach using Pareto optimality that combines two quality metrics, CBO (coupling between objects) and SDMPC (standard deviation of methods per class), in two separate fitness functions. The authors start from the assumption that good design quality results from good distribution of features (methods) among classes. Their Pareto optimality-based algorithm succeeded in finding good sequence of move method refactorings that should provide the best compromise between CBO and SDMPC to improve code quality. However, one of the limitations of this approach is that it is limited to a unique refactoring operation (which is move method) to improve the code structure and only two metrics to evaluate the preformed improvements. In addition, it is odd that there is no semantic evaluator to prove that the semantic coherence is preserved. Recently, Ó Cinnéide et al. [166] have proposed a multi-objective search-based refactoring to conduct an empirical investigation to assess some structural metrics and to explore relationships between them. To this end, they have used a variety of search techniques (Pareto-optimal search, semi-random search) guided by a set of cohesion metrics.

The main limitation of all of the existing approaches is that the semantics preservation has not been explicitly considered to obtain correct and meaningful refactorings.

2.6.4 Refactoring tools

Refactoring tools automate refactorings that programmers would perform with a programming editor. Most of modern and popular development environments for a variety of languages now include refactoring tools such as Eclipse[1], Microsoft Visual Studio[2], Xcode[3], and Squeak[4]. A more extensive list is available in [141]. These tools are integrated in their development environments, but do not support programmers to decide when, where or how to apply refactorings. For large software, selecting and deciding the suitable refactorings to apply is a labor extensive, and error prone task.

To this end, researchers have proposed various ways to improve automated refactoring. For instance, Murphy-Hill et al. [86] [130] [131] proposed several techniques and empirical studies to support refactoring activities. In [86] [87]t ,he authors proposed new tools to assist software engineers in applying refactoring by hand such as selection assistant, box view, and refactoring annotation based on structural information and program analysis techniques. Recently, Ge and Murphy-Hill [132] have proposed new refactoring tool called GhostFactor that allow the developer to transform code manually, but check the

[1] http://eclipse.org
[2] http://msdn.microsoft.com/vstudio
[3] http://developer.apple.com/tools/xcode
[4] http://squeak.org

correctness of her transformation automatically. However, the correction is based mainly on the structure of the code and do not consider its semantics. Mens et al. formalize refactoring by using graph transformations [133]. Bavota et al. [134] automatically identify method chains and refactor them to cohesive classes using extract class refactoring. The aim of these approaches is to provide specific refactoring strategies; the aim of our research in this thesis is to provide a generic and automated refactoring recommendation framework to help developers to refactor their code.

Although refactoring tools offer many potential benefits, programmers appear not to use them as much as they could [130]. There is a need to better assist programmer in their refactoring task using suitable recommendation systems.

2.7 Recommendation systems in software engineering

Recommendation Systems for Software Engineering (RSSEs) are an emerging research area [136]. For example, CodeBroker [142] analyzes developer comments in the code to detect similarities to class library elements that could help implement the described functionality. CodeBroker uses a combination of textual-similarity analysis and type-signature matching to identify relevant elements. It works in push mode, producing recommendations every time a developer writes a comment. It also manages user-specific lists of “known components,” which it automatically removes from its recommendations. Dhruv [143] recommends people and artifacts relevant to a bug report. It operates chiefly in the open source community, which interacts heavily via the Web. Using a three-layer model of community (developers, users, and contributors), content (code, bug reports, and forum messages), and interactions between these, Dhruv constructs a Semantic Web that describes the objects and their relationships. It recommends objects according to the similarity between a bug report and the terms contained in the object and its metadata. Expertise Browser [144] is a tool that recommends people by detecting past changes to a given code location or document. It assumes that developers who changed a method have expertise in it. Finding the right software experts to consult can be difficult, especially when they are geographically distributed. Strathcona [137] can recommend relevant source code fragments to help developers to use frameworks and APIs. Another recommendation system called eRose [138] recommends and predict software artifacts that must be changed together. SemDiff [139] recommend replacement methods for adapting code to a new library version.

Recently, there is much interest in recommendation systems in the field of software refactoring. For instance, in [146], Terra et al. describe the preliminary design of a recommendation system to provide refactoring guidelines for developers and maintainers during the task of reversing an architectural erosion process. They formally describe first recommendations proposed in their research and results of their application in a web-based application. Tsantalis and Chatzigeorgiou have proposed a methodology to suggest Move Method refactoring opportunities [140]. Their general goal is to tackle coupling and cohesion anomalies. More recently, Silva et al. [135] proposed an approach to identify and rank Extract Method refactoring opportunities that are directly automated by IDE-based refactoring tools. Their approach aims to recommend new methods that hide structural dependencies that are rarely used by the remaining statements in the original method.

Thies et al. [149] presents a tool for recommending rename refactorings to harmonize variable names based on an analysis of assignments and static type information. They focus on assignments to discover possible inconsistency of naming, exploiting that a variable assigned to another likely points to same objects and, if declared with the same type, is likely used for the same purpose. However, the proposed approach does not consider other applications such as method, class or package renames which is very important top support other refactoring recommendation tools.

JDeodorant [83] is a system proposed by Tsantalis et al. that can identify and apply some common refactoring operations on Java systems, including Extract Method, Move Method. Their approach is implemented as an Eclipse plugin and relies on the concept of program slicing to select related statements that can be extracted into a new method.

Specifically, two criteria are used to compute such slices: 1) the full computation of a variable, referred to as complete computation slice; 2) all code that affects the state of an object, referred to as object state slice. More recently, Sales et al. [148] describes an approach for identifying Move Method refactoring opportunities based on the similarity between dependency sets. This technique is implemented by a recommendation system called JMove, which detects methods located in incorrect classes and then suggests moving such methods to more suitable ones. More specifically, the proposed technique initially retrieves the set of static dependencies established by a given method m located in a class C. Then JMove calculates based on different static similarity measures if another candidate class can receive the method m. Moreover, Bavota et al. [147] proposed a technique to recommend Move Method refactoring opportunities and remove the Feature Envy code-smell from source code. Their approach, coined as Methodbook, is based on Relational Topic Models (RTM), a probabilistic technique for representing and modeling topics, documents (methods in Methodbook) and known relationships among these. Methodbook uses RTM to analyze both structural and textual information gleaned from software to better support move method refactoring.

Bavota et al. [150] proposed an approach that support extract class refactoring based on graph theory. The proposed approach represent a class to be refactored as a weighted graph in which each node represents a method of the class and the weight of an edge that connects two nodes (methods) represents the structural and syntactical similarity of the two methods. This approach always splits the class to be refactored in two classes. The approach has been extended aiming at splitting a class in more classes [151] where the transitive closure of the incident matrix is computed to identify sets of methods representing the new classes to be extracted.

Furthermore, most of search-based approaches [20] [21] [22] [60] described in Section 2.6.3 are framed into recommendation systems since their goal is to suggest sequences of refactoring operations that could be applied according to different purposes.

A general conclusion to be drawn from existing refactoring work is that most of the effort has been devoted to the definition of manual and (semi-)automatic approaches supporting refactoring based mainly on structural information. Moreover, still existing refactoring approaches are limited only on one or few possible refactoring operations and their usefulness is limited to specific contexts where particular refactoring are needed, e.g., extract method, move method to improve particular aspects of software system. In addition, most of these approaches and tools are based on only structural information which is not always enough to understand and preserve the semantic coherence of the source code when recommending refactoring. Other aspects could significantly help on developing more efficient and practical refactoring recommendation systems such the semantic program analysis and the use of development change history.

2.8 Mining software repositories and historical data

The field of Mining Software Repositories analyzes the data available in systems repositories to uncover interesting information about software systems. Historical information stored in software repositories contains a wealth of information regarding the evolutionary history of a software system and unique view of the actual evolutionary path taken to realize a software system. Here software repositories refer to artifacts that are produced and archived during software evolution [152]. They include sources such as the information stored in source code version-control systems (e.g., the Concurrent Versions System (CVS)), requirements/bug-tracking systems (e.g., Bugzilla), communication archives (e.g., e-mail) and other information stored/extracted along software evolution (e.g., applied refactorings, added requirements, enhanced features, fixed code-smells, etc.).

Software practitioners and researchers are recognizing the benefits of mining this information to support the maintenance of software systems, improve software design/reuse, and empirically validate novel ideas and techniques. Research is now proceeding to uncover the ways in which mining these repositories can help to understand software development and software evolution, to support predictions about software development, and to exploit this knowledge in planning future development [153].

Recently, research work that uses the change history emerged in the in the context of refactoring. Demeyer et al. [155] proposed an approach to detect (reconstruct) refactorings that are applied between two software versions based on the change history. The scope of this contribution is different than the one proposed in [155], since our aim is to suggest refactoring solutions to be applied in the future to improve software quality while maintaining the consistency with the change history. Ratzinger et al. [154] mined change history to predict the likelihood of a class to be refactored in the next two months using machine learning techniques. Their goal is to identify classes that are refactoring or non-refactoring prone. In their prediction models they do not distinguish different types of refactorings (e.g., create super class, extract method, etc.); they only assess the fact that developers try to improve the design. In contrast, in our approach, we suggest concrete refactoring solution to improve code quality and not only identifying refactoring opportunities. Another study was presented by Ratzinger et al. [57] that use refactoring history information to support bugs prediction. They found that refactorings and bugs have an inverse correlation. Thus, when the number of bugs decreases then the number of refactorings increases. Hayashi et al. [162] proposed a technique to instruct how and where to refactor a program by using a sequence of its modifications. They consider that the histories of program modifications reflect developers' intentions, and focusing on them allows us to provide suitable refactoring guides. Their technique can be automated by storing the correspondence of modification patterns to suitable refactoring operations. For instance, when a programmer repeats copy-and-paste operation of a certain part of the program, as a result, instances of code clone appear in several parts of the modified program, and it shows a sign of the code-smell code duplication. At this time, i.e., immediately after the developer performs a sequence of copy-and-paste operations, the proposed approach suggests the developer to proceed to refactor the occurrences of this code clone. However, the proposed approach is not really using traditional mining software repository techniques, rather, it works online to abstract patterns of program modifications executed by developers and make them correspond to refactoring operations. However, the proposed approach is limited only on two types of characteristic modifications: duplication of codes and change of complexity measures, and few refactoring operations.

As far as software refactoring, there are several works on extracting and mining historical data from software repositories in the literature. Research has been carried out to detect and interpret groups of software entities that change together. These co-change relationships have been used for different purposes. Zimmermann et al. [156] have used historical changes to point developers to possible places that need change. In addition historical common code changes are used to cluster software artifacts [157] [161], to predict source code changes by mining change history [156] [157], to identify hidden architectural dependencies [159] or to use them as change predictors [158]. In addition, recently, co-change has been used in several empirical studies in software engineering. However, in the best of our knowledge, until now, the development change history is not used for recommending software refactoring.

Our approach in this dissertation is largely inspired by contributions in mining software repository research. We will describe in Chapter 6 how the development and maintenance change history can be an effective way to recommend software refactoring.

2.9 Summary

Through this chapter, we have provided a comprehensive review of the existing work in different domains related to our work. Several approaches and tools have been proposed to detect code-smells. The vast majority of these techniques rely on declarative rule specification [8] [9] [10] [11]. In these settings, rules are manually defined to identify the key symptoms that characterize a code-smell. These symptoms are described using quantitative metrics, structural, and/or lexical information. Indeed, we share with all the above authors the idea that good code-smell detection strategies relies on the selection of the suitable metrics to characterise these code-smells. However, for each code-smell, rules that are expressed in terms of metric combinations need substantial calibration efforts to find the right threshold value for each metric, above which a code-smell is said to be detected. Since there is no consensus in defining code-smells, different threshold values should be tested to find the best one. This led us to introduce our search-based approach to relieve software developers from burden of manually defining code-smells detection rules.

After detecting code-smells, the next step is to fix them. Authors, such as in Fowler [1], Liu et al. [9], Mens and Tourwé [18], Sahraoui et al. [40], proposed "standard" refactoring solutions that can be applied by hand for each kind of code-smell. However, it is difficult to prove or ensure the generality of these solutions to all code-smells or code fragments. In fact, the same code-smell type can have different possible refactoring solutions. Automated approaches are used in the majority of existing works (O'Keeffe and Cinnéide [23]; Harman and Tratt [20]; Seng et al. [21]) to formulate the refactoring problem as a single-objective optimization problem to improve software structure while preserving the external behaviour. These two concerns drive the existing approaches to refactoring automation. Each approach has its strengths and weaknesses. It helps for conducting research for automating detection and correction (refactoring) of code-smells. However, several concerns and challenges that we stressed in Section 1.2 should be considered to propose efficient and practical refactoring solutions. To tackle these problems, a mono- and multi-objective search-based approach is proposed. The search process aims at finding the optimal sequence of refactoring operations that minimize as much as possible the number of detected code-smells. In addition, we explore other objectives to optimize: the amount of code changes needed to apply refactorings, the semantics preservation, and maintaining the consistency with the change history.

In the next chapter, we describe our contributions for code-smells detection, and we show how to circumvent the above mentioned problems in both detection and correction steps. Our contribution is based on a search-based process to find the suitable code-smells detection rules learned from a base of real instances of code-smells using genetic algorithm.

Part 1: Code-smells detection

The first part of this thesis presents our solution for the detection of code-smells. In this contribution, we propose a search-based approach using code-smell examples that are generally manually validated and available in software repositories of software development companies. Indeed, we translate regularities that can be found in such code-smell examples into detection rules. Instead of specifying rules manually for detecting each code-smell type, or semi-automatically using code-smell definitions, we extract these rules from instances of code-smells. This is achieved using Genetic Programming. Unlike existing approaches, our proposed approach brings a lot of advantages: 1) it does not require to define the different code-smell types, but only to have some code-smell examples; 2) it does not require an expert to write detection rules manually; 3) it does not require specifying the metrics to use or their related threshold values.

Chapter 3 : Search-based code-smells detection

3.1 Introduction

This chapter introduces our first contribution, which consists of the automatic detection of code-smell. To automate the detection of code-smells, we propose a search-based approach, using genetic programming (GP) [171], to generate detection rules. Our proposal consists of using knowledge from previously inspected projects (i.e., code-smell examples) in order to detect code-smells that will serve to generate new detection rules based on combinations of quality metrics and threshold values. A solution to the code-smell detection problem is represented as a set of rules that best detect the code-smells presented on the base of examples with high scores of precision and recall.

This chapter is structured as follows. Section 3.2 recalls the different problems and challenges related to the detection of code-smells and addressed by our approach. Then, we introduce our approach and explain how GP is used to generate code-smells detection rules in Section 3.3. In this section, details are given on the adaptation of GP to the problem of code-smells detection. In Section 3.4, we present an evaluation of the approach, and we discuss the obtained results in Section 3.5. Section 3.6 is dedicated to the limitations of the approach and the threats to validity of the evaluation. Finally, Section 3.7 concludes the chapter and describes our future research work.

3.2 Code-smells detection challenges

Unlike software bugs, there is no general consensus on how to decide if a particular design violates a quality heuristic. Indeed, the vast majority of existing techniques relies on declarative rule specification [8] [9] [10] [14]. In these settings, rules are manually defined to identify the key symptoms that characterize a code-smell. These symptoms are described using quantitative metrics, structural, and/or lexical information. For example, large classes have different symptoms like the high number of attributes, relations and methods that can be expressed using quantitative metrics. However, in an exhaustive scenario, the number of possible code-smells to be manually characterized with rules can be very large. For example, [12] [1] [2] describe more than sixty code-smell types. In addition, this list is not exhaustive because different code-smells are not documented.

Furthermore, there is a difference between detecting symptoms and asserting that the detected situation is an actual code-smell. For example, consider an object-oriented program with hundred classes from which one class implements all the behavior and the other classes are mainly classes with attributes and accessors. No doubt, we are in presence of a Blob. Unfortunately, in real-life systems, we can find many large classes, each one using some data classes and some regular classes. Deciding which classes are Blob candidates heavily depends on the interpretation of each analyst. In some contexts, an apparent violation of a design principle may be consensually accepted as normal practice. For example, a “Log” class responsible for maintaining a log of events in a program, used by a large number of classes, is a common and acceptable practice. However, from a strict code-smell definition, it can be considered as a class with an abnormally large coupling.

Moreover, even when consensus exists, the same symptom could be associated to many code-smell types, which may compromise the precise identification of code-smell types. In fact, translating code-smell definitions from the natural language to metric combinations is a subjective task. For this reason, different code-smells are characterized by the same metrics. Thus, it is difficult to identify some code-smells types. These difficulties explain a large portion of the high false-positive rates reported in most of the existing contributions.

Another very important issue is related to the definition of thresholds when dealing with quantitative information. Indeed, there is a general agreement on extreme manifestations of code-smells. That is, for each code-smell, rules that are expressed in terms of metric combinations need substantial calibration efforts to find the right threshold value for each metric, above which a code-smell is said to be detected. Since there is no consensus in defining code-smells, different threshold values should be tested to find the best one. For instance, the Blob detection involves information such as class size. Although we can measure the size of a class, an appropriate threshold value is not trivial to define. A class considered large in a given program/community of users could be considered average in another.

Besides the previous approaches, software repositories are available in many companies, where code-smells in projects under development are manually identified, corrected and documented. However, this valuable knowledge is not used to mine regularities about code-smell manifestations, although these regularities could be exploited both to detect and correct code-smells.

In the next section, we introduce our approach to overcome some of the above-mentioned limitations for code-smells detection. The proposed approach brings a lot of advantages: 1) it does not require to define the different code-smell types, but only to have some code-smell examples; 2) it does not require an expert to write detection rules manually; 3) it does not require specifying the metrics to use or their related threshold values.

3.3 Approach

This section describes our contribution for code-smells detection problem. The key idea is to see the detection problem as a search based combinatorial optimization problem to find the appropriate detection rules from an exhaustive list of possible metrics and threshold values.

The rest of this section describes the proposed approach in more detail. Section 3.3.1 introduces the proposed approach while Section 3.3.2 explains the adaptation and the design of GP in terms of solution representation, fitness function, selection and genetic operators.

3.3.1 Approach overview

We propose an approach that uses knowledge from previously manually inspected projects, called code-smell examples, in order to detect code-smells that will serve to generate new detection rules based on combinations of software metrics. In short, the detection rules are automatically derived by an optimization process that exploits the available examples. Figure 3.1 shows the general structure of our approach. It takes as inputs a base (*i.e.*, a set) of code-smell examples and a set of quality metrics (the definition and the usefulness of these metrics were discussed in the literature [2]). As output, our approach derives a set of detection rules. Using GP, our rules' derivation process generates randomly, from a given list of quality metrics, a combination of quality metrics/threshold for each code-smell type. Thus, the generation process can be viewed as a search-based combinatorial optimization to find the suitable combination of metrics/thresholds that best detect the code-smell examples. In other words, the best set of rules is the one that detects the maximum number of code-smells (we consider both precision and recall scores).

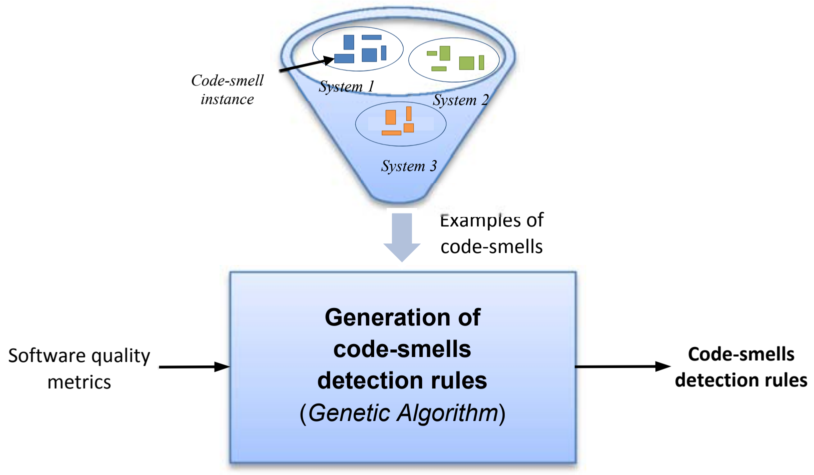


Figure 3.1 - Approach overview.

As showed in Figure 3.2, the base of examples contains projects (systems) that were manually inspected to detect possible code-smells. During a training stage, these systems are iteratively evaluated using rules generated by the algorithm. A fitness function calculates the quality of each solution (rules) by comparing the list of detected code-smells with the expected ones from the base of examples.

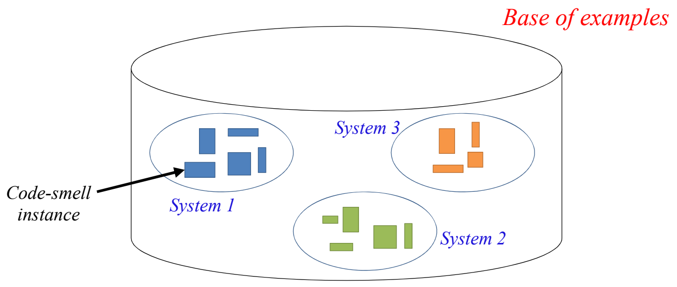


Figure 3.2 - Base of examples

As many metrics combinations are possible, the detection rules generation process is, by nature, a combinatorial optimization problem. The number of possible solutions quickly becomes huge as the number of metrics increases. A deterministic search is not practical in such cases, and the use of heuristic search is warranted. The dimensions of the solution space are set by the metrics and logical operations between them: union (metric1 OR metric2) and intersection (metric1 AND metric2). A solution is determined by assigning a threshold value to each metric. The search is guided by the quality of the solution according to the number of detected code-smells in comparison to the expected ones form the base of examples. To this end, a heuristic search is needed to explore this large number of combination.

3.3.2 GP adaptation

Our SBSE formulation of code-smells detection is based on GP (cf. Section 2.32.3.12.3).A high level view of the GP approach to the code-smells detection problem is summarized in Algorithm 3.1. The algorithm takes as input a set of quality metrics and a set of code-smell examples that were manually detected in some systems, and finds a solution which corresponds to a set of rules that best detect the code-smells in the base of examples.

Algorithm: Code-smells Detection

```
Input:
    Set of quality metrics
    Set of code-smell examples
Process:
    1. I:= rules(R, Code-smell_Type)
    2. P:= set_of(I)
    3. initial_population(P, Max_size)
    4. repeat
    5.     for all I in P do
    6.         detected_ code-smells := execute_rules(R, I)
    7.         fitness(I) := compare(detected_ code-smells, code-smell_examples)
    8.     end for
    9.     best_solution := best_fitness(I);
    10.    P := generate_new_population(P)
    11.    it:=it+1;
    12. until it=max_it
    13. return best_solution
Output:
    best_solution: detection rule
```

Algorithm 3.1 - High-level pseudo-code for GP adaptation to our code-smells detection problem.

Lines 1–3 construct the initial GP population which is a set of individuals that define possible detection rules. The function rules(R, Code-smell_Type) returns an individual I by randomly combining a set of metrics/thresholds that corresponds to a specific code-smell type, e.g., Blob, spaghetti code, or functional decomposition. The function set_of(I) returns a set of individuals, i.e., detection rules, that corresponds to a GP population. Lines 4–13 encode the main GP loop, which explores the search space and constructs new individuals by combining metrics within rules. During each iteration, we evaluate the quality of each individual in the population, and save the individual having the best fitness (line 9). We generate a new population (P+1) of individuals (line 10) by iteratively selecting pairs of parent individuals from population P and applying the crossover operator to them; each pair of parent individuals produces two children (new solutions). We include both the parent and child variants in the new population P. Then, we apply the mutation operator with a probability score for both parent and child to ensure the solution diversity; this produces the population for the next generation. The algorithm terminates when the termination criterion (maximum iteration number) is met, and returns the best set of detection rules (best solution found during all iterations).

To adapt GP for a specific problem, the following elements have to be defined: representation of the individuals; creation of a population of individuals; definition of the fitness function to evaluate individuals for their ability to solve the problem under consideration; selection of the individuals to transmit from one generation to another; creation of new individuals using genetic operators (crossover and mutation) to explore the search space, and finally, the generation of a new population. In the following, we describe more precisely our adaption of GP to the code-smells detection problem.

a) Individual representation

An individual is a set of IF – THEN rules. For instance, let us consider the following detection rule, i.e., individual, and its interpretation (please refer to Appendix A for the definition of the used metrics):

R1: **IF** (LOC(c) ≥ 1500 AND NOM(c) ≥ 29) OR (WMC(c) ≥ 60) **THEN** Blob(c)
R2: **IF** (CBO(c) ≥ 51) **THEN** spaghetti code(c)
R3: **IF** (NPA(c) ≥ 4 AND WMC(c) < 3) **THEN** functional decomposition (c)

Consequently, a detection rule has the following structure:

IF "Combination of metrics with their threshold values" THEN "Code-smell type"

The IF clause describes the conditions or situations under which a code-smell type is detected. These conditions correspond to logical expressions that combine some metrics and their threshold values using logic operators (AND, OR). If some of these conditions are satisfied by a class, then this class is detected as the code-smell figuring in the THEN clause of the rule. Consequently, THEN clauses highlight the code-smell types to be detected. We will have as many rules as types of code-smell to be detected. In our case, mainly for illustrative reasons, and without loss of generality, we focus on the detection of three code-smell types, namely Blob, spaghetti code and functional decomposition.

Consequently, as it is shown in Figure 3.3, we have three rules, R1 to detect Blobs, R2 to detect spaghetti codes, and R3 to detect functional decomposition.

One of the most suitable computer representations of rules is based on the use of trees [171] [172]. 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 different quality metrics with their threshold values. The functions that can be used between these metrics correspond to logical operators, which are Union (OR) and Intersection (AND). Figure 3.3 represents an example detection rule represented as a tree. This tree representation corresponds to an OR composition of three sub-trees, each sub-tree represents a rule: R1 OR R2 OR R3.

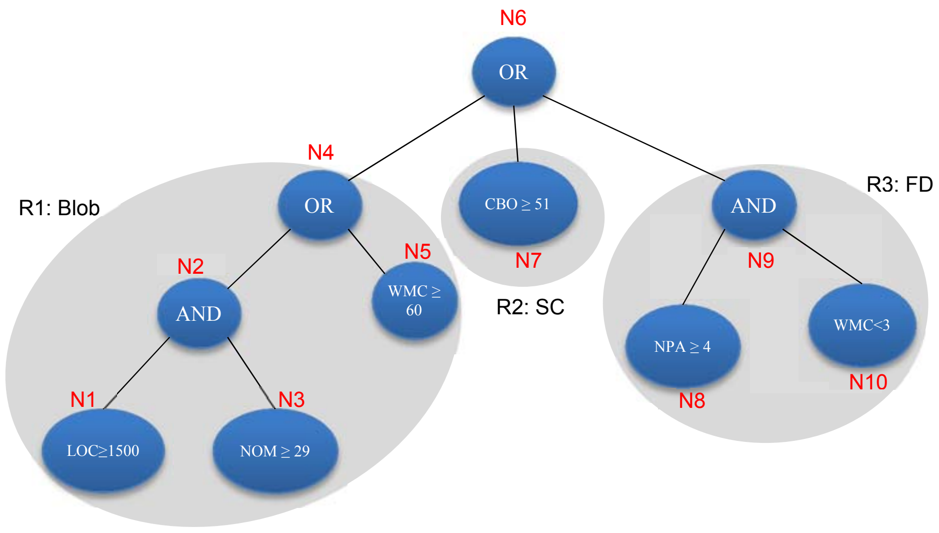


Figure 3.3 - A tree representation of an individual.

For instance, rule R1 is represented as a sub-tree of nodes starting at the branch (N1 - N5) of the individual tree representation of Figure 3.3. Since this rule is dedicated to detect Blob code-smells, we know that the branch (N1 – N5) of the tree will figure out the THEN clause of the rule. Consequently, there is no need to add the code-smell type as a node in the sub-tree dedicated to a rule.

b) 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. The actual tree length will vary with the number of metrics to use for code-smell detection. Notice that a high tree length value does not necessarily mean that the results are more precise since, usually, only a few specific metrics are needed to detect a specific code-smell. These metrics can be specified either by the user or determined randomly. Because the individuals will evolve with different tree lengths (structures), with the root (head) of the trees unchanged, we randomly assign for each one:

- one metric and threshold value to each leaf node
- a logic operator (AND, OR) to each function node

Since any metric combination is possible and correct semantically, we do need to define some conditions to verify when generating an individual.

c) Genetic operators

Selection. To select the individuals that will undergo the crossover and mutation operators, we used stochastic universal sampling (SUS) [171], in which the probability to select an individual is directly proportional to its relative fitness in the population. For each iteration, we used SUS to select *population_size/2* individuals from population *p* to generate population *p+1*. These (*population_size*/2) selected individuals will "give birth" to another (*population_size*/2) new individuals using crossover operator.

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. The crossover operator can be applied only on parents having the same type of code-smell to detect. Each child thus combines information from both parents.

Figure 3.4 shows an example of the crossover process. Indeed, the rule R1 and a rule R2 from another individual (solution) are combined to generate two new rules. The right sub tree of R1 is swapped with the left sub tree of R2.

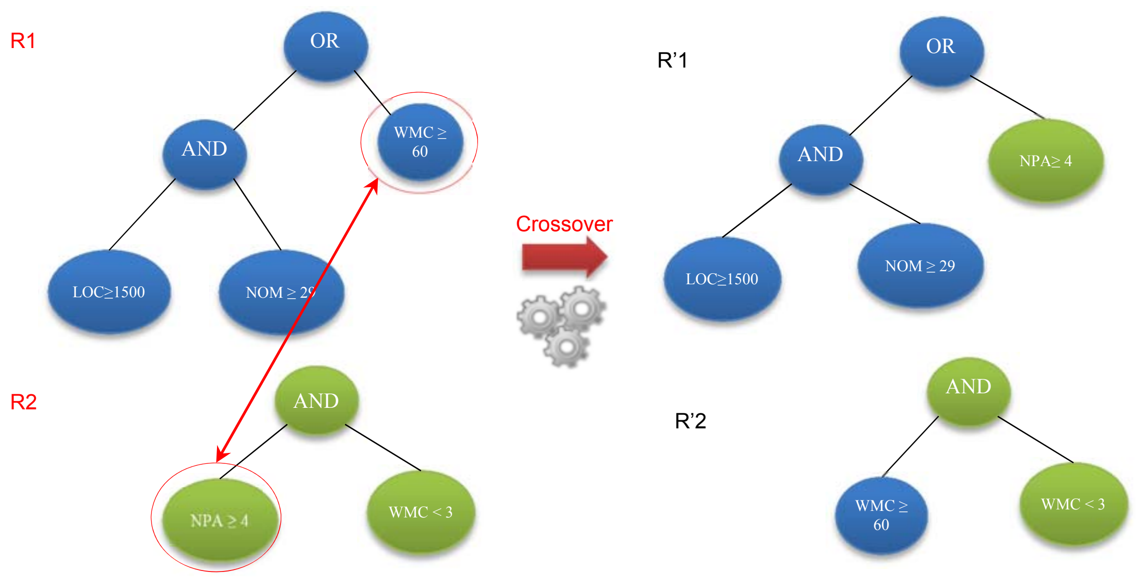


Figure 3.4 - Crossover operator.

As a result, after applying the cross operator the new rule R1 to detect Blob will be:

R1: **IF** (LOC(c) ≥ 1500 AND NOM(c) ≥ 29) OR (NPA(c) ≥ 4) **THEN** Blob(c)

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 (threshold value of a quality metric), it is replaced by another terminal. The new terminal either corresponds to a threshold value of the same metric or the metric is changed, and a threshold value is randomly fixed. 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 sub trees are replaced by a new randomly generated sub tree.

To illustrate the mutation process, consider again the example that corresponds to a candidate rule to detect the Blob code-smell. Figure 3.5 illustrates the effect of a mutation that deletes the node NMD, leading to the automatic deletion of node OR (no left sub tree), and that replaces the node LOCMETHOD by node NPRIVFIELD with a new threshold value. Thus, after applying the mutation operator the new rule R1 to detect Blob will be:

R1: IF (LOCCLASS(c) ≥ 1500 AND NPRIVFIELD(c) ≥ 14)) THEN Blob(c)

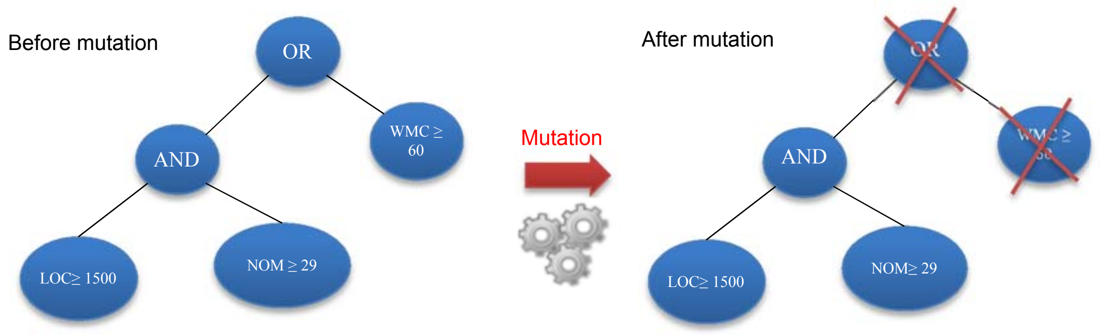


Figure 3.5 - Mutation operator

d) Encoding of an individual

The quality of an individual is proportional to the quality of the different detection rules composing it. In fact, the execution of these rules on the different projects extracted from the base of examples detects various classes as code-smells. Then, the quality of a solution (set of rules) is determined with respect to the number of detected code-smells in comparison to the expected ones in the base of examples. In other words, the best set of rules is the one that detects the maximum number of code-smells.

For instance, let us suppose that we have a base of code-smell examples having three classes X, W, and T that are considered respectively as Blob, functional decomposition and another Blob. A solution contains different rules that detect only X as Blob. In this case, the quality of this solution will have a value of 1/3 = 0.33 (only one detected code-smell over three expected ones).

e) Fitness function

The encoding of an individual should be formalized in a fitness function that quantifies the quality of the generated rules. The goal is to define an efficient and simple (in the sense of not computationally expensive) fitness function in order to reduce computational complexity.

As discussed in Section 3.3.1, the fitness function aims to maximize the number of detected code-smells in comparison to the expected ones in the base of examples. In this context, we define the fitness function of a solution, normalized in the range [0, 1], as:

$$f_{norm} = \frac{\frac{\sum_{i=1}^{p} a_i}{t} + \frac{\sum_{i=1}^{p} a_i}{p}}{2} \in [0,1]$$

where *t* is the number of code-smells in the base of examples, *p* is the number of detected classes with code-smells, and a_i has value 1 if the i^{th} detected class exists in the base of examples (with the same code-smell type), and value 0 otherwise.

To illustrate the fitness function, we consider a base of examples containing one system evaluated manually. In this system, six (6) classes are subject to three (3) types of code-smells as shown in Table 3.1.

Class	Blob	Functional decomposition	Spaghetti code
Student		X	
Person		X	
University		X	
Course	X		
Classroom			X
Administration	X		

Table 3.1 - Code-smells example.

The classes detected after executing the solution generating the rules R1, R2 and R3 of Figure 3.3 are described in Table 3.2.

Class	Blob	Functional decomposition	Spaghetti code
Person		X	
Classroom	X		
Professor		X	

Table 3.2 - Detected classes.

Thus, only one class corresponds to a true code-smell (Person). Classroom is a code-smell but the type is wrong and Professor is not a code-smell. The fitness function has the value:

$$f_{norm} = \frac{\frac{1}{3} + \frac{1}{6}}{2} = 0.25$$

with t=3 (only one code-smell is detected over 3 expected code-smells), and p=6 (only one class with a code-smell is detected over 6 expected classes with code-smells).

3.4 Evaluation

To evaluate our approach, we studied its usefulness to guide quality assurance efforts for six large and medium-size open-source software systems. In this section, we describe our experimental setup and present the results of an exploratory study.

3.4.1 Research questions

We designed our experiments to answer the following research questions:

- **RQ1:** To what extent can the proposed approach detect code-smells?
- **RQ2:** What types of code-smells does it locate correctly?

To answer RQ1, we used an existing corpus of known code-smells [8] to evaluate the precision and recall of our approach. We compared our results to those produced by a rule-based strategy [8]. To answer RQ2, we investigated the type of code-smells that were found.

3.4.2 Systems studied

We used six large-size open-source Java projects to perform our experiments: GanttProject v1.10.2, Xerces-J v2.7.0, ArgoUML v0.19.8, Quick UML v2001, LOG4J v1.2.1, and AZUREUS v2.3.0.6. GanttProject[1] (Gantt for short) is a cross-platform tool for project scheduling. Xerces-J[2] is a family of software packages for parsing XML. ArgoUML[3] is a popular UML modeling tool which includes support for all standard UML 1.4 diagrams. Quick UML[4] is an editor creating and sharing UML diagrams with people on many different platforms and generate Java source code from. LOG4J[5] is a well-known logging library for Java. Finally, AZUREUS[6] is a P2P file sharing client using the bittorrent protocol that search and download torrent files, play, convert and transcode videos and music. Table 3.3 provides some relevant information about the programs. The base of code-smell examples contains more than examples. Table 3.3 provides some descriptive statistics about these six programs.

Systems	# of classes	KLOC	# of code-smells
GanttProject v1.10.2	245	31	41
Xerces-J v2.7.0	991	240	66
ArgoUML v0.19.8	1230	1160	89
Quick UML v2001	142	19	11
LOG4J v1.2.1	189	21	17
AZUREUS v2.3.0.6	1449	42	93

Table 3.3 - Program statistics.

We selected these systems for our validation because they came from six different organisations, involved different kinds of software engineering development and had different sizes, ranging from 21 to 1160 KLOC with a considerable number of code-smell

[1] http://www.ganttproject.biz
[2] http://xerces.apache.org/xerces-j/
[3] http://argouml.tigris.org/
[4] http://sourceforge.net/projects/quj/
[5] http://logging.apache.org/log4j/1.2/
[6] http://sourceforge.net/projects/azureus/

instances. The version of Gantt studied was known to be of poor quality, which led to a major revised version. ArgoUML, Xerces-J, LOG4J, AZUREUS and Quick UML, on the other hand, has been actively evolved over the past 10 years and their design has not been responsible for a slowdown of their developments.

3.4.3 Analysis method

In [8], the authors asked three groups of students to analyse the libraries and tag instances of specific antipatterns in order to validate their detection technique, DECOR. For replication purposes, they provided a corpus of describing instances of different antipatterns that includes Blob classes, spaghetti code, and functional decompositions. As described in Section 2.2.1, Blobs are classes that do or know too much; spaghetti Code (SC) is a code that does not use appropriate structuring mechanisms; finally, functional decomposition (FD) is a code that is structured as a series of function calls (please refer to Appendix C for the definition of these code-smells). These represent different types of design risks. In our study, we verified the capacity of our approach to locate classes that corresponded to instances of these anti-patterns. We used a 6-fold cross validation procedure. For each fold, one open source project is evaluated by using the remaining five systems as the base of examples. For example, Xerces-J is analyzed using detection rules generated from some code-smell examples from ArgoUML, LOG4J, AZUREUS, Quick UML and Gantt.

DECOR [8] reported the number of detected antipatterns, the number of true positives, the recall (number of true positives over the total number of code-smells) and the precision (ratio of true positives over the number of detected code-smells). The obtained results were compared to those of DÉCOR in terms of recall and precision when using our approach for each code-smell in Xerces-J, AZUREUS, LOG4J, Quick UML, ArgoUML and Gantt. Recall and precision are defined as follow:

$$Recall = \frac{true\ positives}{total\ number\ of\ code\ smells}$$

$$Precision = \frac{true\ positives}{number\ of\ detected\ code\ smells}$$

3.4.4 Results

Table 3.4 summarizes our findings. For Gantt, our average code-smell detection precision was 94%. DECOR, on the other hand, had a combined precision of 59% for the same code-smells. The precision for Quick UML was about 86%, over twice the value of 43% obtained with DECOR. In particular, DECOR did not detect any spaghetti code as opposed to our approach. For Xerces-J, our precision average was 90%, while DECOR had a precision of 67% for the same dataset. Finally, the comparison results were mixed for ArgoUML, AZUREUS and LOG4J; still, our precision was consistently higher than 75% in comparison to DECOR.

On the negative side, our recall for the different systems was systematically lower than that of DECOR. In fact, the rules defined in DECOR are too general, which increases the recall at the cost of precision. The main reason that our approach finds better precision results is that the threshold values are well-defined using our GP. Indeed, with DECOR the user should test different threshold values to find the best ones. Thus, it is a fastidious task to find the best threshold combination for all metrics. The Blob code-smell is detected better using DECOR because it is easy to find the thresholds and metrics combination for this kind of code-smells. The hypothesis to have 100% of recall justifies low precision, sometimes, to detect code-smells. In fact, there is a compromise between precision and recall. However, still our approach provides better compromise in terms of F-Measure equals to 88% for all code-smells, whereas DECOR provides only 78.6%. The detection of FDs by only using metrics seems difficult. This difficulty is alleviated in DECOR by including an analysis of naming conventions to perform the detection process. However, using naming conventions lead to results that depend on the coding practices of the development team. We obtained comparable results without having to leverage lexical information. We can also mention that fixed code-smells correspond to the different code-smell types.

In the context of this experiment, we can conclude that our technique is able to identify code-smells, in average, more accurately than DECOR (answer to research question RQ1 above).

System	Precision	Precision-DECOR	Recall	Recall-DECOR
GanttProject	Blob : **100%** SC : **93%** FD : **91%**	90% 71.4% 26.7%	**100%** 97% 94%	**100%**
Xerces-J	Blob : **97%** SC: **90%** FD: **88%**	88.6% 60.5% 51.7%	**100%** 88% 86%	
ArgoUML	Blob : **93%** SC: **88%** FD: **82%**	86.2% 86.4% 38.6%	**100%** 91% 89%	
QuickUML	Blob : 94% SC: **84%** FD: **81%**	**100%** 0% 30%	98% 93% 88%	
AZUREUS	Blob : 82% SC: 71% FD: **68%**	**92.7%** **81.7%** 38.6%	94% 81% 86%	
LOG4J	Blob : 87% SC: **84%** FD: **66%**	**100%** 66.7% 54.5%	90% 84% 74%	

Table 3.4 - Code-smells detection results compared to DÉCOR

As described in Table 3.5, we compared our GP detection results with those obtained by another local search algorithm, simulated annealing (SA) [98]. The detection results for SA are also acceptable. For small systems, the precision when using SA is even better than with GP. In fact, GP is a global search that performs best in a large search space (which corresponds to large systems). In addition, the solution representation used in GP (tree) is suitable for rule generation, while SA uses a vector-based representation that is not. Furthermore, SA takes a lot of time, comparing to GP, to converge to an optimal solution (more than 10 minutes).

System	Precision-GP	Precision-SA
GanttProject	Blob : **100%** SC : 93% FD : **91%**	**100%** **94%** 90%
Xerces-J	Blob : **97%** SC: **90%** FD: **88%**	83% 69% 79%
ArgoUML	Blob : **93%** SC: **88%** FD: **82%**	83% 84% 67%
QuickUML	Blob : 94% SC: 84% FD: 81%	**100%** **88%** **83%**
AZUREUS	Blob : 82% SC: **71%** FD: **68%**	**91%** 63% 54%
LOG4J	Blob : 87% SC: 84% FD: 66%	**100%** **88%** **73%**

Table 3.5 – GP code-smells detection results compared to Simulated Annealing.

3.5 Discussion

We noticed that our technique does not have a bias towards the detection of specific code-smell types. In Xerces-J, we had an almost equal distribution of each code-smell (14 SCs, 13 Blobs, and 14 FDs). On Gantt, the distribution was not as balanced, but this is principally due to the number of actual code-smells in the system. We found all four known FDs and nine Blobs in the system, and eight of the seventeen FDs, four more than DECOR.

An important consideration is the impact of the example base size on detection quality. Drawn for AZUREUS, the results of Figure 3.6 show that our approach also proposes good detection results in situations where only few examples are available. The precision and recall scores seem to grow steadily and linearly with the number of examples and rapidly grow to acceptable values (75%). Thus, our approach does not need a large number of examples to obtain good detection results.

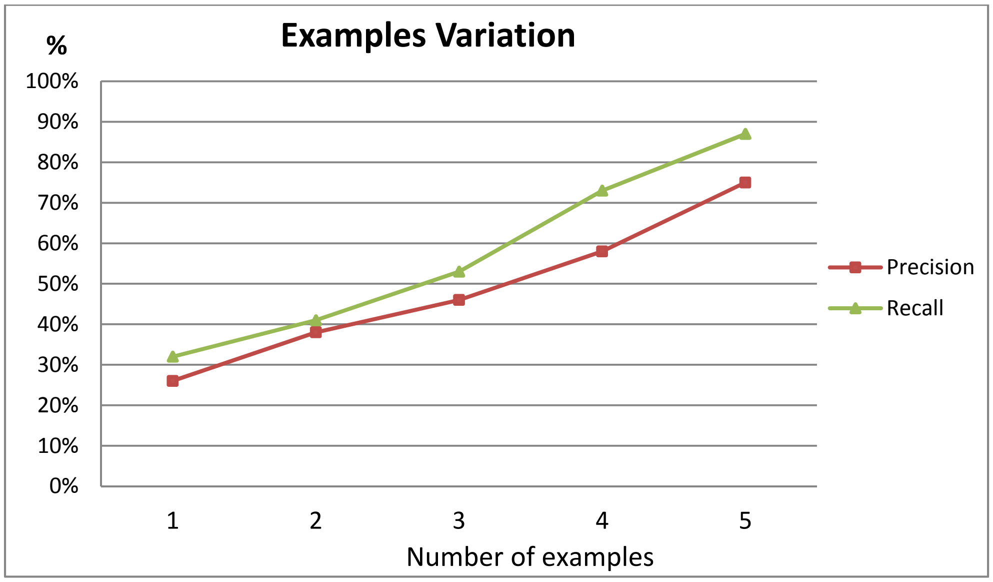


Figure 3.6 - Examples-size variation (example = system).

The reliability of the proposed approach requires an example set of bad code. It can be argued that constituting such a set might require more work than identifying, specifying, and adapting rules. In our study, we showed that by using six open source projects directly, without any adaptation, the technique can be used out of the box and will produce good detection precision and recall results for the detection of code-smells for the studied systems.

In an industrial setting, we could expect that a company starts with some few open-source projects, and gradually evolves its set of bad code examples to include context-specific data. This might be essential if we consider that different languages and software environments have different best/worst practices.

Finally, since we viewed the code-smells detection 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 (Pentium CPU running at 2 GHz with 3GB of RAM). The execution time for rules generation with a population size of 400 individuals and number of iterations (stopping criteria) fixed to 3500 was less than four minutes (3min27s). This indicates that our approach is reasonably scalable from the performance standpoint. However, the execution time depends on the number of used metrics and the size of the base of examples. It should be noted that more important execution times may be obtained than when using DECOR. In any case, our approach is meant to apply mainly in situations where manual rule-based solutions are not easily available.

3.6 Threats to validity

Following the methodology proposed by Wohlin et al. [167], there are four types of threats that can affect the validity of our experiments. We consider each of these in the following paragraphs.

Conclusion validity is concerned with the relation between the treatment and the outcome. We used the Wilcoxon rank sum test [170] with a 95% confidence level to test whether significant differences exist between the measurements for different treatments. This test makes no assumption that the data is normally distributed and is suitable for ordinal data, so we can be confident that the statistical relationships we observed are significant. In our comparison with the technique not based on heuristic search, we considered the parameters provided with the tool. This is can be considered as a threat that can be addressed in the future by evaluating the impact of different parameters on the quality of the results of DECOR.

Internal validity is concerned with the causal relationship between the treatment and the outcome. A possible threat to the internal validity resides in the use of stochastic algorithms. To circumvent this threat 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 [170] with a 95% confidence level ($\alpha = 5\%$). Still, the parameter tuning of the different optimization algorithms used in our experiments creates another internal threat that we need to evaluate in our future work. The parameters' values used in our experiments are found by a trial-and-error method, which is commonly used in the SBSE community [169]. However, it would be an interesting perspective to design an adaptive parameter tuning strategy [168] for our approach so that parameters are updated during the execution in order to provide the best possible performance.

Construct validity is concerned with the relationship between theory and what is observed. Most of what we measure in our experiments are standard metrics such as precision and recall that are widely accepted as good proxies for quality of code-smells detection solutions. Another construct validity threat is related to the absence of similar work that uses search-based algorithms for code-smells detection. For that reason, we compare our proposal with other existing techniques not based on search-based algorithms. Another threat to construct validity arises because, although we considered three types of code-smells, we did not evaluate the detection of other types of code-smells. In future work, we plan to evaluate the performance of our proposal to detect some other types of code-smell. Another construct threat can be related to the corpus of manually detected code-smells since developers do not all agree if a candidate is a code-smell or not. We will ask some new experts to extend the existing corpus and provide additional feedback regarding the detected code-smells.

External validity refers to the generalizability of our findings. In this study, we performed our experiments on six different widely used open-source systems belonging to different domains and with different sizes, as described in Table 3.3. However, we cannot assert that our results can be generalized to industrial Java applications, other programming languages, and to other practitioners. Future replications of this study are necessary to confirm our findings.

3.7 Conclusion

In this chapter, we proposed a new search-based approach for code-smells detection. Typically, researchers and practitioners try to characterize different types of common code-smells and present symptoms to search for in order to locate these code-smells in a system. In this work, we have shown that this knowledge is not necessary to perform the detection. Instead, we use examples of code-smells to automatically generate detection rules. Our study shows that our technique outperforms DECOR [8], a state-of-the-art metric-based approach, where rules are defined manually, on its test corpus. The proposed approach was tested on six medium and large-size open-source systems, and the results are promising.

As part of future work, we plan to extend our base of examples with additional badly-designed code in order to consider more programming contexts. Another direction worth to explore is to improve the detection of potential code-smells through the use of knowledge from software change history. Indeed, as reported in the literature [64] [118], classes participating in design problems (e.g., code-smells) are significantly more likely to be changed [118]. Moreover, if a code-smell (e.g., God Class) is created intentionally and remains unmodified or hardly undergoes changes, the system may not experience any problems [63] [117]. Indeed, it has been shown that, in some cases, a large class might be the best solution [63]. For these reasons, combining software static metrics with software change-based metrics can be an effective way to improve the detection of code-smells.

Once code-smells are detected, they should be fixed as early as possible for maintainability, quality assurance, and evolution considerations. In the next chapter, we introduce our approach to fix code-smells.

Part 2: Mono-objective code-smells correction

In the first part of this thesis we presented our search-based approach for code-smells detection. We used genetic programming to generate code smell-detection rules learned from real code-smell instances.

Once code-smells detected, they need to be fixed as early as possible for maintainability and evolution considerations. Indeed, it is widely believed that refactoring is an effective technique to fix code-smells [1]. In this second part of this thesis, we focus on code-smells correction through refactoring. We deal with refactoring recommending task as a mono-objective optimization problem to improve software quality by fixing code-smells. In this setting, we consider two scenarios for practitioners or software development companies: 1) they have enough time and resources to address all the detected code-smells; 2) there are time and resources limitations.

For the first scenario, we introduce a search-based approach using genetic algorithm to find the optimal sequence of refactoring steps that fixes as much as possible the number of detected code-smells. The approach was successfully applied to six medium and large size software systems by fixing the majority of existing code-smells (90%). Our experimental results provide evidence that refactoring is by nature an optimization problem.

For the second scenario, where there is no enough time and resources to address all the detected. Practitioners need to focus their efforts on fixing only the most critical code-smells. That is, not all code-smells have equal effects and importance. Indeed, it would be important to determine which are the more critical code-smells in order to prioritize their correction. To this end, we introduce a novel approach to prioritize code-smells correction using chemical reaction optimization [176], a newly established metaheuristics.

Chapter 4 : Search-based code-smells correction

4.1 Introduction

We presented in Chapter 3 how code-smell detection rules can be automatically generated from examples of code-smell instances. Due to their harmful impact on the quality, maintenance and evolution of software systems, code-smells should be prevented and removed from the code as early as possible. Hence, it is widely believed that refactoring is an efficient technique to fix code-smells, improve software quality, and above all, increase and developer's productivity by making software systems easier to maintain and understand.

In this chapter, we introduce our approach for recommending refactoring solutions to fix the detected code-smells. At this stage, we consider the refactoring recommending task as a single-objective optimization problem. Our search-based approach aims at finding, from a large list of possible refactorings, the suitable refactoring solutions that fixes the detected code-smells by the means of genetic algorithm (GA) [99]. Indeed, a refactoring solution corresponds to a sequence of refactoring operations that should minimize as much as possible the number of code-smells. To this end, our search based process is guided by an evaluation function that calculates the number of fixed code-smells using detection rules. We evaluate our approach on a benchmark composed of six large and medium size software systems. We found that our approach is able to suggest refactoring solutions to correct the majority (more than 90%) of the detected code-smells.

This chapter is organized as follows. Section 4.2 recalls the different problems and challenges related to the correction of code-smells that are addressed by our approach. Section 4.3 introduces our approach for fixing code-smells using refactoring. In this section, details are given on the adaptation of GA to the refactoring and code-smells correction problem. While Section 4.4 presents an evaluation of the proposed approach, Section 4.5 presents a discussion about the obtained results. Section 4.6 is dedicated to discuss the different limitations and threats to validity. Finally, Section 4.7 concludes the chapter and describes our future research work.

4.2 Code-smells correction and refactoring challenges

Several problems and challenges should be considered when recommending refactoring. Our approach described in this chapter represents a preliminary research direction to show how refactoring strategies can be handled as an optimization problem. At this stage, we mainly address the problems 2.1 - 2.5 identified in Section 1.2.

In fact, the majority of existing approaches [1] [40] [41] have manually defined "standard" refactorings for each code-smell type to remove its symptoms as described In Fowlers book [1]. However, it is difficult to define “standard” refactoring solutions for each code-smell type and to generalize them because it depends mainly on the programs in their context. To make the situation worst, removing code-smell symptoms does not mean that the actual code-smell is corrected, and, in the majority of cases, these “standard” solutions are unable to remove all symptoms for each code-smell.

Furthermore, different possible refactoring strategies should be defined for the same type of code-smell. The problem is how to find the “best” refactoring solutions from a large list of candidate refactorings and how to combine them in an appropriate order? The list of all possible refactoring strategies, for each code-smell, can be very large [25]. Thus, the process of defining refactoring strategies manually, from an exhaustive list of refactorings, is fastidious, time-consuming, and error-prone.

From another perspective, in the majority of existing approaches [20] [21] [22] [49], code quality can be improved without fixing code-smells. In other terms, improving some quality metrics does not guarantee that the detected code-smells are fixed. Therefore, the link between code-smells detection (refactoring opportunities) and correction is not obvious. Thus, we need to ensure whether the refactoring concretely corrects detected code-smells.

More significantly, existing approaches consider the refactoring (*i.e.,* the correction process) as local process by fixing code-smells (or improving quality) separately. That is, a refactoring solution should not be specific to only one code-smell type; instead, the impact of refactoring should be considered on the whole system. For example, moving methods to reduce the size/complexity of a class may increase the global coupling, or fixing some code-smells may create other code-smells in other code fragments.

These observations were at the origin of the work described in this chapter.

4.3 Approach

In this Section, we introduce our approach for recommending refactoring to fix code-smells. We also show the importance of heuristic search to explore the large search space of possible refactoring solutions. We start by presenting an overview of our approach in Section 4.3.1. Then, we describe, in Section 4.3.2, GA adaptation for the refactoring recommending problem in terms of solution representation, fitness function, selection and genetic operators.

4.3.1 Approach overview

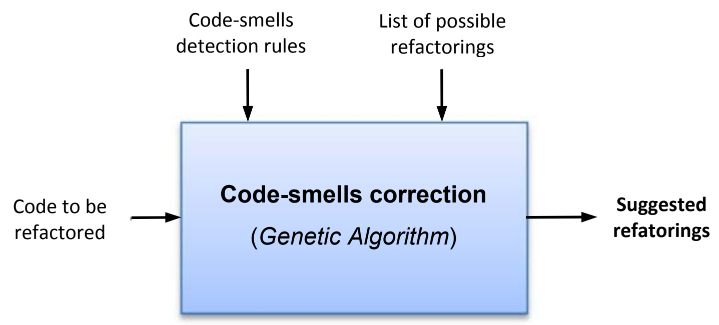


Figure 4.1 - Approach overview.

To correct the detected code-smells, we propose a search-based approach that aims at finding, from a large list of possible refactoring operation, the suitable refactorings that fixes the detected code-smells. To this end, we use GA to find the suitable refactoring solutions. Our main aim is to find refactoring solutions that should minimize as much as possible the number of code-smells. As illustrated in Figure 4.1, our approach takes as input a smelly source code (i.e., contains code-smells), a list of possible refactoring operations that can be applied (please refer to Appendix B for the list of considered refactorings), and code-smells detection rules. As output, our approach suggests the optimal sequence of refactoring operations to fix the detected code-smells.

4.3.2 GA adaptation

Our SBSE formulation of code-smells correction is based on GA (cf. Section 2.32.3.1). A high level view of the GA approach to the code-smells correction problem is summarized in Algorithm 4.1. The algorithm takes as input code fragments to be corrected *Smelly_code*, a set of possible refactoring operations *RO*, and a set of code-smell detection rules *D*. Lines 1–5 construct an initial solution population based on a

```
1.   Algorithm: Code-smells Correction
Input:
   Smelly_code,
   Set of refactoring operations RO,
   Code-smells detection rules D,
Process:
   1. initial_population(P, Max_size)
   2. P0:= set_of(S)
   3. S:= sequence_of(RO)
   4. code:= Smelly_code
   5. t:=0
   6. repeat
   7.     for all Si in P do
   8.             code:= execute_refactoring(Si, Smelly_code);
   9.               fitness(Si) := calculate_Quality(D, code);
   10.   end for
   11.   best_solution := best_fitness(Si);
   12.   P := generate_new_population(P)
   13.   it:=it+1;
   14. until it=max_it
   15. return best_solution
Output:
   best_solution: refactoring solution
```

Algorithm 4.1 - High-level pseudo-code for GA adaptation to our code-smells correction problem.

specific representation, using the list of *RO* given at the input. This initial population stands for a set of possible code-smell correction solutions (i.e., sequence of refatorings) returned by the function *set_of(S)*, each one representing sequences of *RO* selected and combined randomly using the function *sequence_of(RO)*.

Lines 6-14 encode the main GA loop whose goal is to make a population of candidate solutions evolve toward the best sequence of RO, *i.e.*, the one that minimizes as much as possible the number of code-smells. During each iteration *t*, each refactoring sequence in the current population is executed on the smelly code (line 8). Then, each solution should be evaluated using our fitness function *calculate_Quality* (line 9) by calculating the number of fixed code-smells over the initial number of code-smells using the detection rules. After that, the best solution is recorded in a specific variable called *best_solution*. Then, a new population is generated using our defined genetic operators, i.e., crossover and mutation (line 12). The algorithm terminates when it reaches the termination criterion, i.e., maximum iteration number, (line 14). The algorithm then returns the best solution obtained during all iterations (line 15).

One key element when applying a search-based technique is to find a suitable mapping between the problem to solve and the techniques to use, i.e., in our case, fixing code-smells. Applying GA to a specific problem requires specifying the following elements: representation of a solution, the fitness function to evaluate the candidate solutions, the selection of the fittest solutions, and the change operators to derive new solutions from existing ones. In our approach, these elements are defined as follows:

a) Solution Representation

In our GA design, we use a vector-based solution coding. Each vector's dimension represents a refactoring operation. When created, the order of applying these refactorings corresponds to their positions in the vector. In addition, for each refactoring, a set of controlling parameters, e.g., actors and roles, as illustrated in Table 4.1, are randomly picked from the program to be refactored. An example of a solution is given in Figure 4.2. Hence, we construct a refactoring solution incrementally. First, we create an empty vector

Ref.	**Refactorings**	**Controlling parameters**
MM	Move Method	(source class, target class, method)
MF	Move Field	(source class, target class, field)
PUF	Pull Up Field	(source class, target class, field)
PUM	Pull Up Method	(source class, target class, method)
PDF	Push Down Field	(source class, target class, field)
PDM	Push Down Method	(source class, target class, method)
IC	Inline Class	(source class, target class)
EC	Extract Class	(source class, new class)
EI	Extract Interface	(Source class, interface)
ESuC	Extract Super Class	(Source class, super class)
ESC	Extract Sub Class	(Source class, sub class)

Table 4.1 - Refactorings and its controlling parameters.

MoveMethod (*Person, Employee, getSalary()*)
ExtractMethod (*Person, printInfo(), printContactInfo()*)
MoveMethod (*Departement, University, division ()*)
PushDownField (*Person, Student, studentId*)
InlineClass (*Car, Vehicle*)
MoveMethod (*Person, Employee, setSalary()*)
MoveField (*Person, Employee, tax*)
ExtractClass(*Person, Address, streetNo, city, zipCode, getAdress(), updateAdress()*)

Figure 4.2 - Representation of a GA individual.

that represents the current refactoring solution. Then, we randomly select 1) a refactoring operation from the list of possible refactorings and 2) its controlling parameters (i.e., the code elements), after that, 3) we apply this refactoring operation to an intermediate model that represents the original source code. The model will be updated after applying each refactoring operation and the process will be repeated n times until reaching the maximal solution length (n). This means that in each iteration i, we have a different model according to the (i-1) applied refactoring operations. That is, in each iteration, the controlling parameters will be selected from the current version of the model. For this reason, the order of applying the refactoring sequence influences the refactoring results. To ease the manipulation of these operations, we use logic predicates to describe them. For example, the predicates *MoveMethod (Person, Employee, getSalary())* indicates that the method *getSalary()* is moved from class *Person* to class *Employee*.

Moreover, when creating a sequence of refactorings (individuals), it is important to guarantee that they are structurally feasible and that they can be legally applied. The first work in the literature was proposed by Opdyke [17] who introduced a way of formalising the preconditions in order to preserve the behavior of the system. These preconditions must be imposed before a refactoring can be applied. Opdyke created functions which could be used to formalise constraints. These constraints are similar to the Analysis Functions used later by Ó Cinneide [124] and Roberts [175] who developed an automatic refactoring tool to reduce program analysis. In our approach, we used a system to check a set of conditions, taking inspiration from the work proposed by Ó Cinnéide [124]. Although we suggest a recommendation system and we do not apply refactorings automatically, we verify the applicability of the suggested refactorings.

Similarly to [124], our search-based refactoring tool simulates refactorings using pre- and post-conditions that are expressed in terms of conditions on a code model. For instance, to apply the refactoring operation *MoveMethod(Person, Employee, getSalary())*, a number of necessary preconditions should be satisfied, e.g., *Person* and *Employee* should exists and should be classes; *getSalary()* should exist and should be a method; the classes *Person* and *Employee* should not be in the same inheritance hierarchy; the method *getSalary()* should be implemented in *Person*; the method signature of *getSalary()* should not be present in class *Employee*. As post-conditions, *Person*, *Employee*, and *getSalary()* should exists; *getSalary()* declaration should be in the class *Employee*; and *getSalary()* declaration should not exists in the class *Person*.

For composite refactorings, such as extract class and inline class, the overall pre and post conditions should be checked. For a sequence of refactorings which may be of any length, we simplify the computation of its full precondition by analyzing the precondition of each refactoring in the sequence and the corresponding effects on the code model (postconditions). For more details about the pre- and post-conditions the interested reader is invited to confer to [17].

b) Fitness function.

After creating a solution, it should be evaluated using a fitness function to ensure its ability to solve the problem under consideration. The fitness function quantifies the quality of the proposed refactoring solutions. In fact, the fitness function calculates the number of fixed code-smells using the detection rules. In this context, we define the fitness function as the ratio of the number of fixed code-smells after applying refactoring over the initial number of code-smells, as follows:

$$Quality = \frac{\#\ of\ code\ smells\ after\ refactoring}{n}$$

where n is the initial number of code-smells before refactoring.

c) Selection.

To generate new refactoring solutions, roulette-wheel selection is used [52]. This technique assigns to each refactoring solution a probability of being selected that is proportional to its fitness. This selection strategy favors the fittest refactoring solutions while still giving a chance of being selected to the others. Note that some refactoring solutions could be included directly into the new population, i.e., elitist strategy.

d) Genetic operators.

To better explore the search space, the crossover and mutation operators are defined.

For crossover, we use a single, random, cut-point crossover. Crossover operator starts by selecting and splitting at random two parent solutions. Then, the 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, for the second child, the first part of the second parent with the second part of the first parent. This operator must ensure that the length limits are respected by eliminating randomly some refactoring operations. As illustrated in Figure 4.3, each child combines some of the refactoring 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.

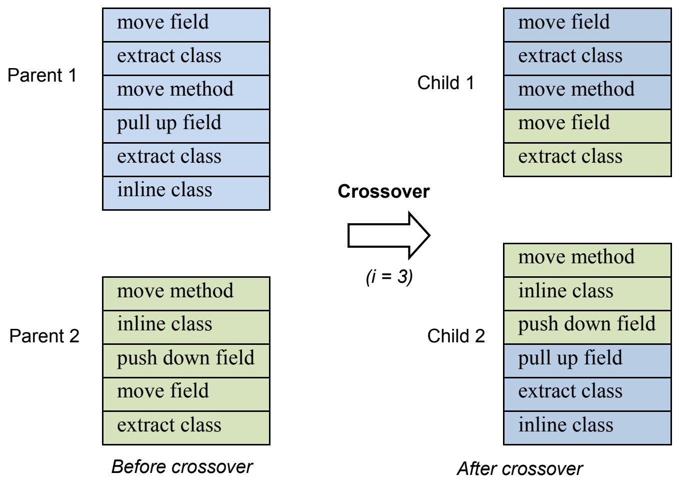


Figure 4.3 - Crossover operator.

The mutation operator picks randomly one or more operations from a sequence of refactoring operations and replaces them by other ones from the initial list of possible refactorings. An example is shown in Figure 4.4.

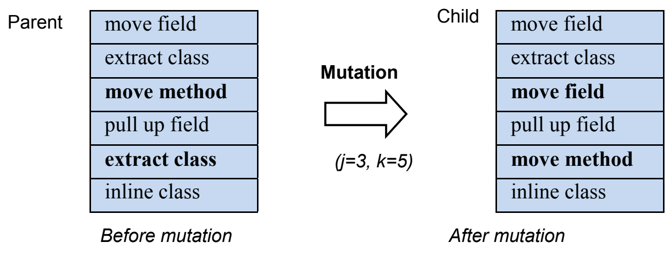


Figure 4.4 - Mutation operator.

4.4 Evaluation

In order to evaluate the feasibility and the efficiency of our approach for generating refactoring solutions to fix code-smells, we conducted an experimental evaluation based on six different software systems. In this section, we describe our experimental setup and present the obtained results.

4.4.1 Research questions

We designed our experiments to answer the following research questions:

The goal of the study is to evaluate the efficiency of our approach for the detection and correction of code-smells from the perspective of a software maintainer conducting a quality audit. We present the results of our experiments that are designed to answer the following research questions:

- **RQ1:** To what extent can the proposed approach correct code-smells?
- **RQ2:** To what extent are the recommended refactorings feasible?

4.4.2 Analysis method

To answer **RQ1**, we check the efficiency of the recommended refactorings in fixing the detected code-smells. To this end, we introduce an evaluation metric called *code-smells correction ratio* (CCR) that calculates the ratio of the number of corrected code-smells over the initial number of detected code-smells before refactoring. CCR is defined as follow:

$$CCR = \frac{\#\, of\ corrected\ code\ smells}{intial\ number\ of\ code\ smells}$$

To answer **RQ2**, we validated manually if the proposed refactoring solutions are useful and feasible. A refactoring operation is considered as feasible if it doesn't affect the semantic coherence of the original program. To this end, we introduce the evaluation metric *precision* that calculates the number of feasible refactorings over the total number of recommended refactorings. The precision metric is defined as follow:

$$Precision = \frac{\#\, of\ feasible\ refactorings}{total\ number\ of\ recommended\ refactorings}$$

Although, the refactored program is correctly working as refactoring preserves the behavior (i.e., operational semantics), it may be semantically incoherent in its internal structure with respect to the domain semantics (i.e., descriptive/modelling semantics). When a semantic error is found manually, we consider the operations related to this change as a bad recommendation. The precision metric is a performance indicator that should be evaluated.

4.4.3 Systems studied

To evaluate the efficiency of our approach, we conducted our experiments on six medium and large size open-source systems: GanttProject, Xerces-J, ArgoUML, Quick UML, LOG4J, and AZUREUS. These systems are described in Section 3.4.2. Moreover, Table 3.3 provides descriptive statistics about these six programs.

We selected these systems for our validation because they are well studied in the related work. Moreover, they came from six different organisations, involved different kinds of software engineering development, and had different sizes ranging from 21 to 1160 KLOC with a high number of code-smell instances.

4.4.4 Results

Table 4.2 summarize our findings to answer **RQ1**. The obtained result shows that our approach succeeded in finding refactoring solutions that are able to correct most of detected code-smells. For instance, for GanttProject 95% (39 over 41) of the detected code-smells (10 over 11 of blob, 17 over 18 of functional decomposition, and 12 over 12 of spaghetti code) was fixed after applying the proposed refactorings. The lowest CCR score is obtained for ArgoUML (85%). In average, as shown in Table 4.2, 90% of code-smells were fixed, for all the studied systems. Thus, the obtained results give evidence that our approach is efficient in fixing code-smells and the obtained correction scores are considered significant. However, we found that the majority of non-fixed code-smells are related to Blob type. Indeed, this type of code-smells requires, in general, a large number of refactoring operations and it is very difficult to fix since it is known to be a very heavy class in terms of behaviour and functionalities that it implements.

To answer **RQ2**, we applied the proposed refactorings by hand using the Eclipse[1] IDE, and verified manually the feasibility of the different proposed refactoring sequences for each system. As shown in Table 4.2, we found that a large number of the proposed refactorings are feasible and can be successfully applied. For all the studied systems, we found that, in average, 55% of the recommended refactorings are semantically coherent and feasible.

System	CCR	Precision
GanttProject	95% (39|41)	52 %
Xerces-J	89% (59|66)	49 %
ArgoUML	85% (76|89)	59 %
QuickUML	90% (26|29)	59 %
LOG4J	88% (15|17)	51 %
AZUREUS	94% (87|93)	57 %

Table 4.2 - Correction results: CCR median values of 31 independent runs of GA.

Thus, we found that some of the proposed refactorings are arbitrary changes, and therefore, they are unfeasible from software programmer's perspective. Consequently, some semantic errors (coherence of the refactored program with the domain semantics) were found. We calculate a correctness precision score (ratio of feasible refactoring operations over the number of proposed refactoring) as performance indicator of our algorithm (the last column in Table 4.2). For each proposed sequence of refactorings, we evaluate the proposed refactorings to eliminate those which are not semantically coherent with the program semantics.

Furthermore, to evaluate the efficiency of our approach in suggesting feasible refactoring solutions, we compared our results with another state-of-the-art search-based approach [20]. Harman et al. [20] proposed a search-based approach to find the optimal sequence of move method refactorings to optimize two quality metrics: CBO (coupling between objects), and SDMPC (standard deviation of methods per class). Figure 4.5 shows the comparison results. For all the six studied systems, we found comparable results for

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

both approaches with an average of 55% for our approach and 56% Harman's approach. Indeed, our obtained precision score is reasonably acceptable, at this stage, since the semantics coherence of the refactored program is not explicitly considered.

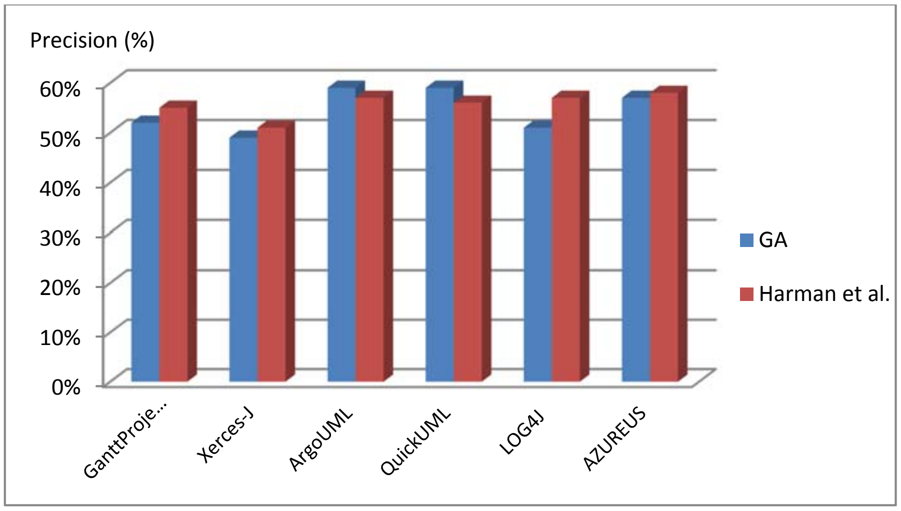


Figure 4.5 - GA precision comparison with Harman et al.

4.5 Discussions

Although our approach produces good refactoring recommendations for fixing code-smells in terms of CCR, it is important to investigate its scalability. Indeed, there is a pressing need for scalable solutions to Software Engineering problems. Scalability is widely considered as one of the key problems for Software Engineering research and development [174]. To evaluate the scalability of the performance of our approach for systems of increasing size, we executed GA on the six studied systems that were from different sizes ranging from 142 for Quick UML to 1449 classes for AZUREUS. As shown in Figure 4.6, when the size of the systems increase, the execution time is not significantly affected in turn.

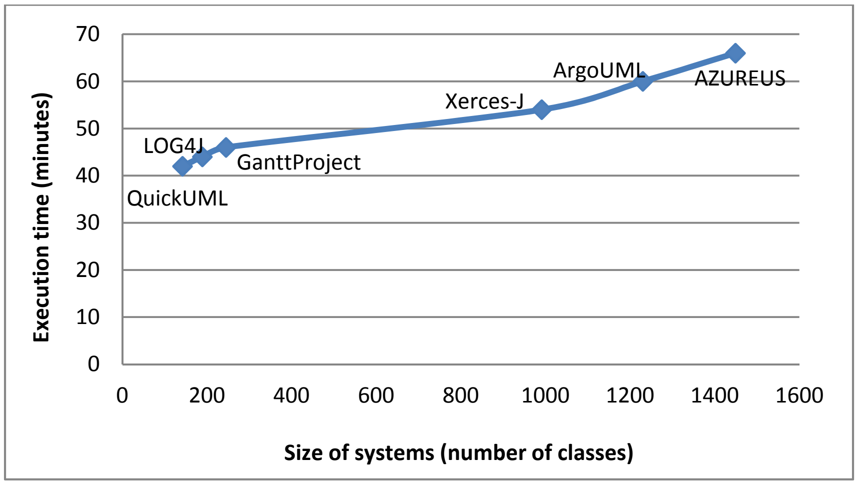


Figure 4.6 - Scalability of GA on different systems sizes.

Furthermore, in SBSE, an important step is the tuning parameters. In general, there is no standard parameters (e.g., population size, crossover/mutation rates, etc.) that should be used in SBSE. This requires a calibration effort to find the best parameters. In our implementation, we assume that the number of refactoring operations per individual cannot exceed a certain threshold value. Because of the nature of crossover or mutation operations, newly created individuals might violate this constraint. After any genetic operation, if this constraint is violated for a new individual, repair operation is performed by eliminating some operations. In order to calibrate crossover and mutation rates, we considered four population sizes (100, 150, 200, and 250) and varied the value of both rates between 5% and 90% with increments of 5%. Due to the GA's randomness effect, we ran the GA 31 times for each configuration and then calculated the average of all the outputs in terms of CCR. To analyze the impact of crossover and mutation, we used two representative systems: GanttProject and Xerces. Our experiments showed that with higher population sizes (200, 250) best performance is achieved, when crossover rate is over 0.6. Regardless of the crossover settings, low mutation rates ended up with lower CCR (worse results) than high mutation rates. Particularly, best performance in terms of CCR is achieved when fixing the mutation rate to 0.1. In addition, when we increased the mutation rate to 0.5, crossover did not show any effect on the CCR. Usually high mutation destructs the discovered good solutions and has negative impact on the performance. We also noticed that crossover operator showed similar behavior with software systems with different number of code-smells and different sizes. Thus, our analysis suggests the use of high crossover rate (0.6) and low mutation rate (0.1), as commonly used in GA experiments.

Furthermore, we noticed that in some cases, applying the proposed refactorings need a considerable effort in terms of code modification/adaptation score. In other words, in some situations many related software artifacts are affected when applying refactoring, and therefore, they need to be changed and adapted by the maintainer. Moreover, another important issue is to study the semantic preservation of the refactored code when searching the refactoring operations.

Thus, to circumvent these problems, the amount of required code changes and semantic preservation should be explicitly considered during the search process.

4.6 Threats to validity

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

Regarding the conclusion validity, we used the Wilcoxon rank sum test with a 95% confidence level to test if significant differences existed between the measurements for different treatments. This test makes no assumption that the data is normally distributed and is suitable for ordinal data, so we can be confident that the statistical relationships we observed are significant. The reported GA and Harman's precision values obtained with the median CCR values of 51 independent runs. The p-values of the Wilcoxon rank sum test indicate whether the median of Harman et al.'s approach is statistically different from GA with a 95% confidence level ($\alpha = 0.05$). A statistical difference is accepted at $p<=0.05$.

For the internal validity, the used techniques to detect code-smells can lead to some false positives that may have an impact on the results of our experiments. To mitigate this threat, we implemented our approach in a flexible way to support the adaptation of code-smells detection rules from other state-of-the-art approaches or other existing code-smells detection tools according to the user preferences.

Construct validity is concerned with the relationship between theory and what is observed. The manual evaluation of the feasibility of the suggested refactorings depends on the expertise of the developer and also it is a subjective process to make sure that a detected code-smell is fixed.

External validity refers to the generalizability of our findings. In this study, we performed our experiments on several different widely used open-source systems belonging to different domains and with different sizes. However, we cannot assert that our results can be generalized to other industrial Java applications, other programming languages, and to other practitioners. We plan to conduct more experiments to test our approach on other software systems and compare our results with other approaches.

4.7 Conclusion

In this chapter, we presented a novel approach to the problem of code-smells correction. We start by generating some code-smells correction solutions that represents a combination of refactoring operations to apply. A fitness function calculates, after applying the proposed refactorings, the number of fixed code-smells using the detection rules. The best solution has the maximum fitness value. Due to the large number of refactoring combination, a genetic algorithm is used. The proposed approach was tested on six open-source systems and the results are promising. Our study shows that our technique succeeded in fixing most of the detected code-smells (90%) while having reasonably accepted score of feasible refactorings (55%). Typically, researchers and practitioners try to each detected fix code-smell separately form a software system. In this work, we have shown how the correction process can be a global process instead of a local one to prevent introducing new code-smells implicitly when fixing existing ones.

Despite these encouraging results, there is still plenty of room for improvement. First, in large-scale systems, the number of code-smells to fix can be very large and not all of them can be fixed automatically. Thus, the prioritization of the list of code-smells is required based on different criteria such as the severity, risk, and importance of classes, etc. Moreover, from software engineer’s perspective, we intend explore additional techniques to help preserving the semantic coherence of the refactored program. Another direction worth to explore is to reduce the amount of code changes when recommending refactoring in order to keep as much as possible with the initial design.

Chapter 5 : Prioritizing code-smells correction

5.1 Introduction

In this chapter, we describe our approach based on chemical reaction optimization metaheuristic search to prioritize the correction of severest and riskiest code-smells according to maintainer's preferences and criteria. As far as we know, this is the first contribution in SBSE that adopt CRO metaheuristics. More specifically, the primary contributions of the chapter can be summarized as follows:

1. We introduce a novel formulation of the refactoring suggestion problem using chemical reaction optimization (CRO) and, to the best of our knowledge, this is the first attempt in SBSE to adopt CRO to solve software engineering problems.
2. We present a prioritization schema based on four prioritization heuristics: severity, risk, importance, and the priority according to maintainers' preferences.
3. We report the results of an empirical study on a benchmark composed of five different medium and large size software systems. We compare our approach to two other approaches that do not prioritize the correction of code-smells.
4. We report statistical comparisons between our CRO-based approach with three popular metaheuristics, genetic algorithm (GA) [99], simulated annealing (SA) [98], and particle swarm optimization (PSO) [100], which have been shown to have good performance in solving many software engineering problems.

Our experimental study indicates that the CRO approach has a great promise. The statistical analysis of the obtained results provides evidence to support the claim that CRO is more efficient and effective than three other popular metaheuristics. Over 31 runs for each approach, our CRO based approach significantly outperforms the two other refactoring approaches in terms of number of corrected code-smells, as well as the number of important, severest, riskiest code-smells that can be fixed.

This chapter is organized as follows. Section 5.2 presents the concept of prioritizing code-smells correction tasks. Section 5.3 describes our approach. Section 5.4 reports our experimental evaluation, while Section 5.5 discusses the obtained result. Section 5.6 is dedicated for the threats to validity. Finally, we conclude and suggest our future research direction in Section 5.7.

5.2 Code-smells prioritization

In large-scale systems, the number of code-smells to fix can be very large and not all of them can be fixed automatically. Moreover, once detected, not all code-smells have equal effects and importance [12] [63] [64] [112]. In general, developers need to start by fixing the higher risk code-smells. However, most of existing refactoring approaches [42], [46] treat the code-smells to fix with the same importance. The majority of existing contributions proposes manual or semi-automated refactoring solutions that can be applied to fix particular types of code-smells (e.g., Blobs, spaghetti code, etc.) [12] [21] [42] or to improve some quality metrics (e.g., cohesion, coupling, etc.) [23] [41] without taking into consideration the importance/risk of the code fragments.

Thus, it is important to focus the attention on code-smells that represents severest problems to be removed first through refactoring. Thus, the prioritization of the list of code-smells is required based on different criteria such as the severity, risk, and importance of classes, etc. Indeed, it would be important to determine which are the more critical code-smells in order to prioritize their correction. For instance, based on our prior work on code-smells correction and refactoring using GA described in Chapter 4, we found that most of the important and riskiest code-fragments are not improved. Moreover, most of riskiest code-smells, notably the Blob code-smell [70], are very difficult to fix using such a manual or an automated approach. Typically, the Blob requires a large number of refactorings. This type of code-smells can be detected most of the time within important classes that change frequently during the development/maintenance process, which make this kind of code-smell more severe than other code-smells.

5.3 Approach

This section presents our approach to support automated refactoring suggestion to fix code-smells where riskiest code-smells are prioritized during the correction process. Hence, we formulated the refactoring suggestion problem as a combinatorial optimization problem to find the near-optimal sequence of refactorings from a large number of possible refactorings. To this end, we used a novel metaheuristic search by the means of CRO [176] to find the suitable refactoring solutions that maximize the number of corrected code-smells while prioritizing the most important and riskiest code fragments. We first present an overview of our approach and the problem formulation and, subsequently, present the CRO algorithm and its adaptation for prioritizing code-smells correction problem.

5.3.1 Approach overview

Our approach is designed to support automated code-smells correction according to a prioritization schema where the more critical code-smells are prioritized while taking into consideration the preferences of developers. In practice, a suitable prioritization scheme can significantly improve and maximize the efficiency of allocating maintenance efforts. Our approach aims at finding, from a large list of possible refactorings, the suitable refactoring solutions that should fix as much as possible the number of detected code-smells according to a prioritization schema.

To find the suitable refactoring solution, a large search space of possible refactorings should be explored. Indeed, the search space is determined not only by the number of possible refactorings, their possible combinations, and the order in which they should be applied, but also by the software system's size (number of packages, classes, methods, fields, etc.). To this end, we see the refactoring suggestion problem as a search-based optimization problem to explore this large search space, in order to find the suitable refactoring solutions by the means of CRO [163].

The general structure of our approach is sketched in Figure 5.1. It takes as inputs: the source code of the program to be refactored, a list of possible refactorings that can be applied, a set of code-smells detection rules, risk and severity score for each detected code-smell, software maintainer prioritization/preferences, and a history of code changes applied to the system during its lifecycle. Our approach generates as output the optimal sequence of refactorings, selected from the list of possible refactorings that should improve software quality by minimizing as much as possible the number of more critical code-smells.

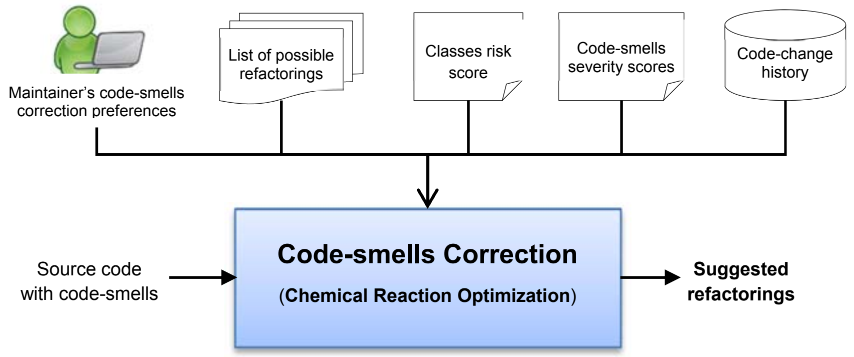

Maintainer's code-smells correction preferences
List of possible refactorings
Classes risk score
Code-smells severity scores
Code-change history
Source code with code-smells
Code-smells Correction
(Chemical Reaction Optimization)
Suggested refactorings

Figure 5.1 - Approach overview.

5.3.2 Problem formulation

We now describe our formulation of the refactoring recommendation task taking into consideration a set of criteria, as well as software engineers' preferences. Our goal is to ensure that the more critical code-smells are fixed first. More concretely, let us consider the example of a software system that contains 5 code-smells: 1 Blob, 2 data classes (DC), and 2 functional decompositions (FD). Many possible refactoring solutions can fix these code-smells with the same score. For example, we can have 2 different solutions S_1 and S_2. After applying S_1, both DC and FD are fixed (the correction score $CCR(S_1)=4/5=0.8$), and after applying S_2, the Blob, 1 DC and 2 FD are fixed (the correction score $CCR(S_2)=4/5=0.8$). The same correction score is obtained by both S_1 and S_2. However, the Blob class is known to be the severest and it may significantly affect the design of the whole system since it tends to centralize the functionalities of the system into one class [70]. From this perspective we consider that the solution S_2 is better than S_1. Prioritizing the correction of critical code-smells is the main idea behind this contribution. Furthermore, sometimes changing extensively a software system by applying refactorings may perturb the initial design. To preserve the initial design, software maintainers my ignore some of the suggested refactorings. To this end it is very interesting to start first by fixing important code-smells. Hence, results can be of interest to software engineers, who perform development or maintenance activities and need to take into account and forecast their effort.

In the following we describe a set of four prioritization heuristics (priority, severity, risk and importance) that we adopt in our formulation of the refactoring task to correct code-smells:

Priority. Developers typically give more importance to some code-smell types that can occur with different impacts on the system's quality. Developers can rank different types of detected code-smell according to their preferences. Using such prioritization scheme, developers can save their time and maximize the efficiency of allocating maintenance effort in their software project. For the experiment reported in this chapter, we assigned a priority score of 7 for the Blob code-smell, 6 for functional decomposition, 5 for shotgun surgery, 4 for spaghetti code, 3 for feature envy, 2 for schizophrenic class, and 1 for data classes, so that fixing Blob code-smells instances will be more prioritized.

Severity. In practice, not all code-smells have equal effects/importance [63] [82]. Each individual instance has its severity score that allows designers to immediately spot and fix the most critical instances of each code-smell. Concretely, the same code-smell type can occur in different code fragments but with different impact scores on the system design [112]. This impact score represents the relative severity of the code-smell, as well as the absolute negative impact on overall quality. For example, two code-smell instances having respectively 27 and 36 methods can be detected both as Blob, but each of them have different impact scores on the system quality (number of methods, coupling, cohesion, etc.). We use the inFusion tool [82], which classifies code-smells based on a set of "design-properties" such as size and complexity, encapsulation, coupling, cohesion, and hierarchy [182]. Moreover, these design properties are the most useful ones in existing code-smells detection approaches [8] [10].

Risk. An important score to consider is the risk score. Thus, we consider that the more code deviates from good practices, the more it is likely to be risky [35]. Consequently, the riskiest code-smells should be prioritized during the correction step. A risk score is associated to each detected code-smell that corresponds to the deviance from well design code [35].

Importance. Generally, developers need to know which code fragments (e.g.,classes, packages) are important in the whole software system in order to focus their effort on improving their quality. In a typical software system, important code fragments are those who change frequently during the development/maintenance process to add new functionalities, to accommodate new changes or to improve its structure. Moreover, as reported in the literature [64] [118], classes participating in design problems (e.g., code-smells) are significantly more likely to be subject to changes and to be involved in fault-fixing changes (bugs) [118]. Indeed, if a class undergoes many changes in the past and it is still smelly, it needs to be fixed as soon as possible. On the other hand, not every code-smell is assumed to have negative effects on the maintenance/evolution of a system. It has been shown that in some cases, a large class might be the best solution [63]. Moreover, it is reported that if a code-smell (e.g., God Class) is created intentionally and remains unmodified or hardly undergo changes, the system may not experience any problems [63] [115]. For these reasons, code-smells related to more frequently changed classes should be prioritized during the correction process.

5.3.3 CRO design

In our formulation, we used the chemical reaction optimization (CRO) [176]. A detailed description of CRO is given in Section 2.3.3. To the best of our knowledge, this work represents the first attempt to exploit CRO within the SBSE community. In general, to adapt a metaheuristic search technique to a specific problem, a number of elements have to be defined, and different decisions have to be made. To apply CRO, the following

elements have to be defined: the way in which solutions (molecules) should be encoded so that they can be manipulated by the search process, creation of a population of solutions (a container of molecules), evaluation function to determine a quantitative measure of the ability of candidate solutions to solve the problem under consideration, selection of solutions for elementary reaction operations, creation/modification of new solutions using elementary reaction operations (on-wall ineffective collision, decomposition, synthesis, and intermolecular ineffective collision) to explore the search space.
In the following, we describe the design of these elements for the code-smells correction problem using CRO.

a) Solution representation

In our CRO design, we use the same vector-based solution representation adopted in our GA adaptation. The description of our solution representation is detailed in Section 3.3.2.

b) Creation of the initial population of solutions

To generate an initial population, we start by defining the maximum vector length (maximum number of operations per solution). The vector length is proportional to the number of refactorings that are considered and the size of the program to be refactored. A higher number of operations in a solution do not necessarily mean that the results will be better. Ideally, a small number of operations can be sufficient to provide good solutions. This parameter can be specified by the user or derived randomly from the sizes of the program and the given refactoring list. During the creation, the solutions have random sizes inside the allowed range. To create the initial population, we normally generate a set of PopSize solutions randomly in the solution space.

c) Elementary Reaction Operators

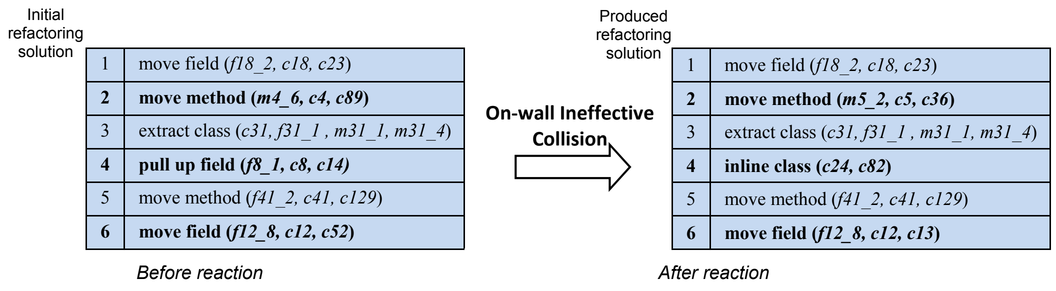
To better explore the search space using CRO, elementary reaction operators are defined. In the following, we describe these operators corresponding to the four elementary reactions of CRO. We denote a refactoring solution in vector form with *w*.

On-wall Ineffective Collision

For on-wall ineffective collision, many possible changes can be applied to a given refactoring solution. To apply this change operator, *n* (one or more) refactorings are first picked at random from the vector representing the refactoring solution (sequence of refactorings). Then, for each of the selected refactorings, we apply one of the following possible changes using a "probabilistic select" [176]:

- **Refactoring type change (RTC):** consists of replacing a given refactoring operation (the refactoring type and his controlling parameters) by another one which is selected randomly from the initial list of possible refactorings. Pre- and post-conditions should be checked before applying this change.
- **Controlling parameters change (CPC):** consists of replacing randomly, for the selected refactoring, only their controlling parameters. For instance, for a "move method", we can replace the source and/or target classes by other classes from the whole system.

An example is shown in Figure 5.2. Three refactorings are randomly selected from the initial vector: one *refactoring type change* (dimension number 4), and two *controlling parameters change* (dimensions number 2 and 6).


Initial refactoring solution
On-wall Ineffective Collision
Produced refactoring solution
Before reaction
After reaction

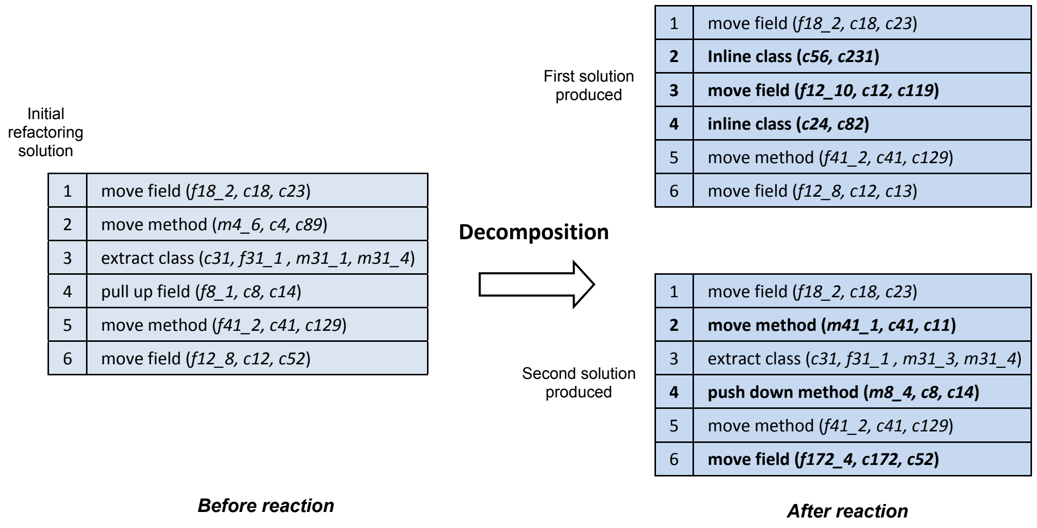
1	move field (*f18_2, c18, c23*)
2	**move method (*m4_6, c4, c89*)**
3	extract class (*c31, f31_1 , m31_1, m31_4*)
4	**pull up field (*f8_1, c8, c14*)**
5	move method (*f41_2, c41, c129*)
6	**move field (*f12_8, c12, c52*)**

1	move field (*f18_2, c18, c23*)
2	**move method (*m5_2, c5, c36*)**
3	extract class (*c31, f31_1 , m31_1, m31_4*)
4	**inline class (*c24, c82*)**
5	move method (*f41_2, c41, c129*)
6	**move field (*f12_8, c12, c13*)**

Figure 5.2 - Example of on-wall ineffective collision operator.

Decomposition

This operator is used to produce two new solutions far away from a given one. We apply "half-total-change" [176] to our implementation. We first duplicate the original solutions. Then, we add perturbations to *n/2* dimensions of the original solution to create new solutions, where *n* is the size of the vector representing the original solution. Each perturbation change could be performed through a *refactoring type change* but also *controlling parameters change*. An example is depicted in Figure 5.3.



1	move field (*f18_2, c18, c23*)
2	move method (*m4_6, c4, c89*)
3	extract class (*c31, f31_1 , m31_1, m31_4*)
4	pull up field (*f8_1, c8, c14*)
5	move method (*f41_2, c41, c129*)
6	move field (*f12_8, c12, c52*)

1	move field (*f18_2, c18, c23*)
2	**Inline class (*c56, c231*)**
3	**move field (*f12_10, c12, c119*)**
4	**inline class (*c24, c82*)**
5	move method (*f41_2, c41, c129*)
6	move field (*f12_8, c12, c13*)

1	move field (*f18_2, c18, c23*)
2	**move method (*m41_1, c41, c11*)**
3	extract class (*c31, f31_1 , m31_3, m31_4*)
4	**push down method (*m8_4, c8, c14*)**
5	move method (*f41_2, c41, c129*)
6	**move field (*f172_4, c172, c52*)**

Figure 5.3 - Example of decomposition operator.

Inter-molecular Ineffective Collision

Inter-molecular ineffective collision is the process of two or more solutions to share information with each other and then produce two or more other different solutions. In our implementation we apply inter-molecular ineffective collision between only two solutions (w_1 and w_2). To this end, two possible change mechanisms could be applied: 1) apply for each of solution on-wall Ineffective Collision, 2) exchange some dimensions between them using an operator similar to a single, random, cut-point crossover, in Genetic Algorithm [99]. First a random value *k* is chosen from [0, 1]. Then inter-molecular ineffective collision creates two new solutions by putting, for the first new solution, the first $k*n_1$ elements from the first parent (with length n_1), followed by the last $(1-k)*n_2$ elements from the second parent (with length n_2). On the other hand, the second new solution, contains the first $k*n_2$ elements from the second parent followed by last $(1-k)*n_1$ element of the first parent. This operator ensures that the generated solutions will never have greater size than the biggest of the parents [183]. As illustrated in Figure 5.4, each child combines some of the refactoring operations of the first parent with some ones of the second parent.

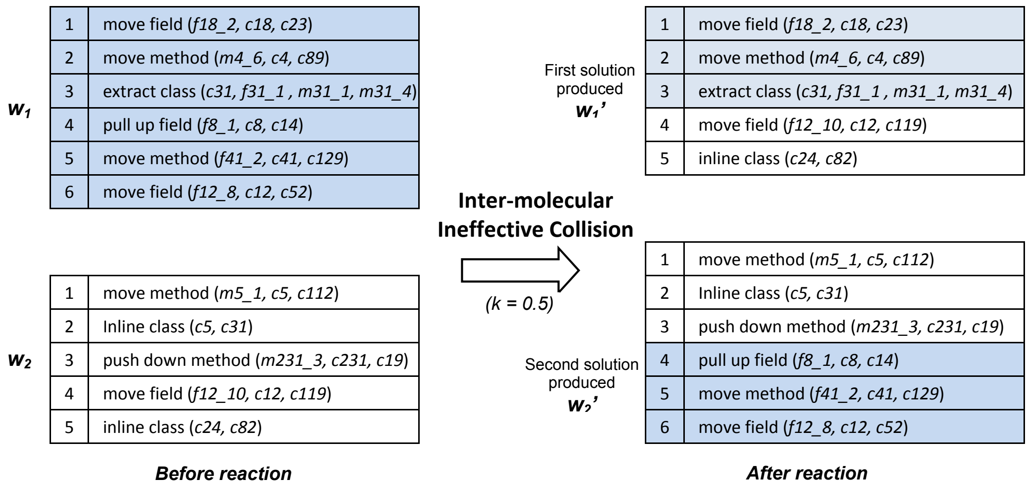

w1
1 move field (f18_2, c18, c23)
2 move method (m4_6, c4, c89)
3 extract class (c31, f31_1 , m31_1, m31_4)
4 pull up field (f8_1, c8, c14)
5 move method (f41_2, c41, c129)
6 move field (f12_8, c12, c52)
w2
1 move method (m5_1, c5, c112)
2 Inline class (c5, c31)
3 push down method (m231_3, c231, c19)
4 move field (f12_10, c12, c119)
5 inline class (c24, c82)
Inter-molecular Ineffective Collision
(k = 0.5)
First solution produced w1'
1 move field (f18_2, c18, c23)
2 move method (m4_6, c4, c89)
3 extract class (c31, f31_1 , m31_1, m31_4)
4 move field (f12_10, c12, c119)
5 inline class (c24, c82)
Second solution produced w2'
1 move method (m5_1, c5, c112)
2 Inline class (c5, c31)
3 push down method (m231_3, c231, c19)
4 pull up field (f8_1, c8, c14)
5 move method (f41_2, c41, c129)
6 move field (f12_8, c12, c52)
Before reaction
After reaction

Figure 5.4 - Example of inter-molecular ineffective collision operator.

Synthesis

This operator is used to combine two refactoring solutions w_1 and w_2 into a new one w. In our approach we are using two different mechanisms for synthesis operator: 1) cross-cut combination, and 2) probabilistic select [163]. To apply synthesis operator, CRO selects randomly one of these two mechanisms.

For the first, an integer value k is randomly generated in the range of [1, n], where n is the shortest vector length of the solutions w_1 and w_2 (n=Min($|w_1|$, $|w_2|$)). Then w is generated by picking the first k values from w_1 and the rest of the (n - k) values from w_2. This operator must ensure that the length limits are respected. If not, some refactoring operations should be eliminated randomly. As shown in Figure 5.5, a new refactoring solution w is formed by combining the first two set of refactorings from w_1 and the last set of refactorings from w_2.

For the second mechanism, using probabilistic select a new solution w is produced from two solution w_1 and w_2. This operator generates w as follows: for each dimension $w(i)$ in w, a random number $t \in [0.1]$ is generated. If t>0.5, we assign that dimension from $w_1(i)$. Otherwise, we assign that dimension from $w_2(i)$.

The idea behind these different synthesis mechanisms is diversification of solutions to better explore the search space.

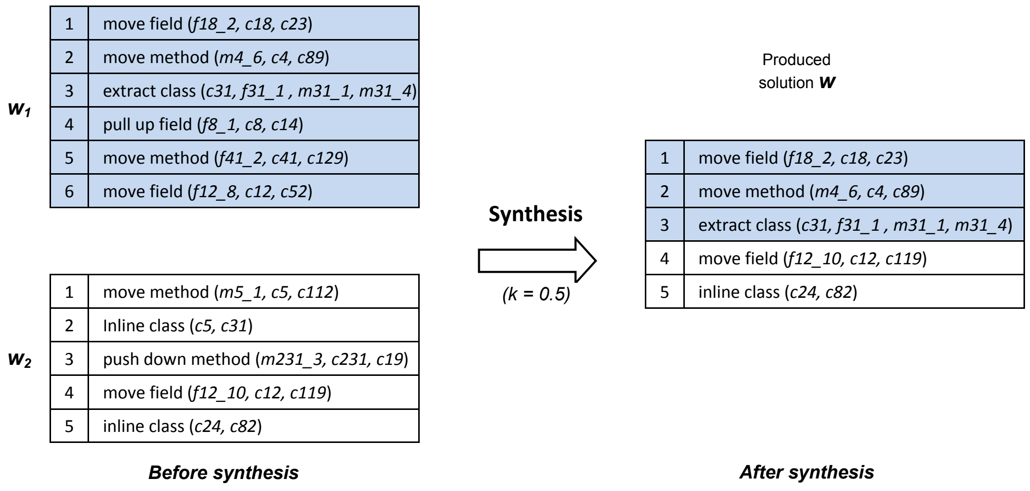

w1
1 move field (f18_2, c18, c23)
2 move method (m4_6, c4, c89)
3 extract class (c31, f31_1 , m31_1, m31_4)
4 pull up field (f8_1, c8, c14)
5 move method (f41_2, c41, c129)
6 move field (f12_8, c12, c52)
w2
1 move method (m5_1, c5, c112)
2 Inline class (c5, c31)
3 push down method (m231_3, c231, c19)
4 move field (f12_10, c12, c119)
5 inline class (c24, c82)
Synthesis
(k = 0.5)
Produced solution W
1 move field (f18_2, c18, c23)
2 move method (m4_6, c4, c89)
3 extract class (c31, f31_1 , m31_1, m31_4)
4 move field (f12_10, c12, c119)
5 inline class (c24, c82)
Before synthesis
After synthesis

Figure 5.5 - Example of synthesis operator.

d) Fitness function

After creating a solution, it should be evaluated using an objective function to ensure its ability to solve the problem under consideration. We used a fitness function that calculates, according to prioritization schema described in the section 3.3, the number of corrected code-smells using detection rules. To calculate the quality of a candidate refactoring solution w, we define the following fitness function:

$$Fitness(w) = \sum_{i=0}^{n-1} \left(x_i * \left(\alpha * Severity(c_i) + \beta * priority(c_i) + \gamma * risk(c_i) + \delta * importance(c_i) \right) \right)$$

where x_i is assigned to 0 if the actual class is detected as a code-smell using our code-smells detection rules, 1 otherwise; and $\alpha+\beta+\gamma+\delta= 1$ and their values express the confidence (i.e., weight) in each measure that can be assigned according to the developers preferences. We have performed comprehensive experiments with different combinations of weights on each prioritization measure. For our experiments, we give equal weights (=0.25) to each of them.

5.3.4 Implementation details

An often overlooked aspect of research on metaheuristic search algorithms relies in the selection and tuning of the algorithms parameters, which is necessary in order to ensure not only fair comparison, but also for potential replication. To this end, we report our algorithmic parameter tuning and selection used to facilitate replication of our findings.

The initial population/solution of CRO, GA, PSO and SA are completely random. The stopping criterion is when the maximum number of function evaluations, set to 8000, is reached. After several trial runs of the simulation, the parameter values of the four algorithms are fixed. There are no general rules to determine these parameters, and thus, we set the combination of parameter values by trial and error. Parameter settings of the four algorithms are shown in Table 5.1. For each algorithm, we repeat the simulation 31 times in each case, and compute the median value.

Algorithms	**Parameters**	**Values**
CRO	Population size	200
	KELossRate	0.05
	MoleColl	0.5
	InitialKE	0.1
	α	40
	β	0.6
GA	Population size	200
	Crossover probability	0.6
	Mutation probability	0.1
	Number of crossing points	1
	Selection	Roulette selection
SA	initial temperature	100
	final temperature	0.157
	cooling coefficient	0.98
	number of iterations	25
PSO	number of particles in a swarm	200
	acceleration coefficient c_1	2
	acceleration coefficient c_2	2

Table 5.1 - Parameter settings used for the different algorithms.

Another issue is that our formulation of code-smells correction problem using prioritization schema is a maximization problem. However, CRO is originally designed to solve minimization problems and the objective function value should not be negative since it is interpreted as energy [176]. Typically, to convert a maximization problem *f* to a minimization one, *–f* is considered as objective function; however this may not be appropriate for CRO [177]. Thus, to keep with CRO principles, we can consider *f'*= 1 - *f*, to make every possible *f'* non-negative. After minimizing *f'*, we can compute the corresponding *f* by *f* = 1 – *f'*. As such, CRO can be adapted to solve maximization problems.

In our experiments reported in this chapter, we considered the following code-smell types: Blob, Data Class, Spaghetti Code, Functional Decomposition, Schizophrenic Class, Shotgun Surgery, and Feature Envy. The description of these smells can be found in Section 2.2.1.

5.4 Evaluation

To evaluate the feasibility and the efficiency of our approach for generating good refactoring suggestions according to prioritization schema, we conducted our experiments based on different versions of medium and large open source systems. In this section, we start by presenting our research questions. Then, we describe the design of our experiments and discuss the obtained results.

5.4.1 Research Questions and Objectives

We assess the performance of our approach by finding out whether it could generate good refactoring strategies that fix code-smells according to a prioritization schema. Our study aims at addressing the four research questions outlined below. We also explain how our experiments are designed to address these questions. The four research questions are:

- **RQ1:** (*Usefulness*) To what extent can the proposed approach correct code-smells?
- **RQ2:** (*Precision*) To what extent can the proposed approach correct severest, riskiest and important code-smells?

- **RQ3:** (*Comparison to state-of-the-art*) To what extent can the proposed approach improves the results of refactoring suggestion compared to existing work that do not use prioritization?
- **RQ4:** (*Comparison with other metaheuristics*) How does the proposed approach using CRO perform compared to other popular search-based algorithms GA [99], SA [92], PSO [100]?

5.4.2 Systems studied

We applied our approach to five large and medium size open-source java projects: Xerces-J[12], JFreeChart[13], GanttProject[14], ArtOfIllusion[15], and JHotDraw[16]. 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, Art of Illusion is a 3D-modeller, renderer and raytracer written in Java. We selected these systems for our experimental study 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 5.2 provides some descriptive statistics about these five programs.

Systems	**Release**	**# classes**	**# code-smells**	**KLOC**	**# previous code-changes**	**Code change method**
Xerces-J	v2.7.0	991	171	240	7493	Change log
JFreeChart	v1.0.9	521	116	170	2009	Change log
GanttProject	v1.10.2	245	53	41	91	Recorded ref.
ArtofIllusion	v2.8.1	459	127	87	594	Recorded ref.
JHotDraw	V7.0.6	468	25	57	1006	Change log

Table 5.2 - Systems statistics.

[12] http://xerces.apache.org/xerces-j
[13] http://www.jfree.org/jfreechart
[14] www.ganttproject.biz
[15] http://www.artofillusion.org
[16] http://www.jhotdraw.org

Previous code-changes applied to previous versions (seventh column in Table 5.2), are used mainly to calculate the importance score. In general, open-source programs and their change history (e.g., change log in Concurrent Versions System named CVS[17], or Apache Subversion System named SVN[18]) are available through SourceForge.net[19]. However, for other software programs especially where code change history is not publicly available, code-changes could be expressed in terms of recorded refactorings that are applied to previous versions (i.e., how many times a class experienced a refactoring). In order to vary our experiments settings, we are using both: change log history (for Xerces and JFreeChart), and recorded refactorings (for GanttProject, AntApache, JHotDraw and Rhino). To collect refactorings applied for each program, we use Ref-Finder [46]. Ref-Finder, implemented as an Eclipse plug-in, can identify refactoring operations between two releases of a software system.

5.4.3 Analysis method

To answer **RQ1**, we used two metrics: code-smells correction ratio (CCR) and refactoring precision (RP).

1) *CCR* calculates the number of corrected code-smells over the total number of code-smells detected before applying the proposed refactoring sequence. CCR is given by the following equation:

$$CCR = \frac{\#\, corrected\ code\_smells}{\#\, code\_smells\ before\ applying\ refactorings} \in [0,1]$$

2) For the refactoring precision (*RP*), we inspect manually the feasibility of the different proposed refactoring sequences for each system. We applied the proposed refactorings using Eclipse IDE and we checked the semantic coherence of the modified code fragments. Some semantic errors (programs behavior) were found. When a semantic error is found manually, we consider the operations related to this

[17] http://cvs.nongnu.org/
[18] http://subversion.apache.org/
[19] http://sourceforge.net/

change as a bad recommendation. Then, we calculate a correctness precision score (ratio of possible refactoring operations over the number of proposed refactoring) as usefulness indicator of our approach. *RP* is defined as follows:

$$RP = \frac{\#\, feasable\ refactorings}{\#\, proposed\ refactorings} \in [0,1]$$

To answer **RQ2**, we define three metrics:

1) The importance correction ratio (*ICR*) that corresponds to the sum of importance score of detected code-smells after applying a given refactoring solution *w* compared to the one before applying refactoring. *ICR* reflects the efficiency of a refactoring solution for correcting important code-smells, so that the higher the *ICR* is, the more a refactoring solution is considered as a good recommendation. *ICR* is defined as follows:

$$ICR(w) = 1 - \frac{\sum_{i=0}^{n-1}(x_i * importance(c_i))}{\sum_{j=0}^{m-1}(x_j * importance(c_j))} \in [0,1]$$

where *n* and *m* are the number of classes in the system, respectively, after and before applying the refactoring solution *w*, the function $importance(c_i)$ returns the importance score of the class c_i , and x_i takes the value 0 if the actual class c_i is detected as code-smell using code-smells detection rules, 1 otherwise.

2) The risk correction ratio (*RCR*) that corresponds to the sum of importance score of detected code-smells after applying a given refactoring solution *w* compared to the one before applying refactoring. *RCR* reflects the efficiency of a refactoring solution for correcting riskiest code-smells, so that the higher the *RCR* is, the more a refactoring solution is considered as a good recommendation. *RCR* is defined as follows:

$$RCR(w) = 1 - \frac{\sum_{i=0}^{n-1}(x_i * risk(c_i))}{\sum_{j=0}^{m-1}(x_j * risk(c_j))} \in [0,1]$$

where *n* and *m* are the number of classes in the system, respectively, after and before applying the refactoring solution *w*, the function $risk(c_i)$ returns the risk score of the class c_i, and x_i takes the value 0 if the actual class c_i is detected as code-smell using code-smells detection rules, 1 otherwise.

3) The severity correction ratio (SCR) that corresponds to the sum of importance score of detected code-smells after applying a given refactoring solution w compared to the one before applying refactoring. SCR reflects the efficiency of a refactoring solution for correcting severest code-smells, so that the higher the SCR is, the more a refactoring solution is considered as a good recommendation. SCR is defined as follows:

$$SCR(w) = 1 - \frac{\sum_{i=0}^{n-1}(x_i * severity(c_i))}{\sum_{j=0}^{m-1}(x_j * severity(c_j))} \in [0,1]$$

where n and m are the number of classes in the system, respectively, after and before applying the refactoring solution w, the function $severity(c_i)$ returns the severity score of the class c_i, and x_i takes the value 0 if the actual class c_i is detected as code-smell using code-smells detection rules, 1 otherwise.

For **RQ3**, we compare our approach to two other different approaches: our GA-based code-smells correction approach, and CRO without the use of prioritization where the refactoring suggestion task consider is considered only from the quality improvement standpoint (i.e., without considering prioritization).

Finally, to answer **RQ4**, we assessed the performance of the CRO algorithm that we use in our approach compared to three other popular meta-heuristic algorithms GA, SA and PSO. We selected these three metaheuristics because they range from global search (GA and PSO) and local search (SA). Moreover, these three metaheuristics are the most frequent ones demonstrating good performance in solving different software engineering problems according to recent surveys [90].

5.4.4 Results

Before delving into details, we provide a high-level view of the experimental approach that we adopted and its rationale. We first compared our approach to two other techniques that do not use prioritization (CRO without prioritization, and our GA-based approach described in Chapter 4), where the fitness function calculates the number of corrected code-smells, to ensure the effectiveness of using such prioritization schema. Then, we compare the performance of CRO to three popular metaheurisitics (GA, SA, and PSO) using the same CRO fitness function to evaluation the performance of CRO. Thus, due to the stochastic nature of the algorithms/approaches we are studying, each time we execute an algorithm we can get slightly different results. To cater for this issue and to make inferential statistical claims, our experimental study is performed based on 31 independent simulation runs for each algorithm/technique studied. Wilcoxon rank sum test [170] is applied between CRO-based approach and each of the other algorithms/techniques (CRO without prioritization, Kessentini et al. 2011) in terms of CCR, ICR, RCR and CSR with a 99% confidence level (α = 1%). Our tests shows that the obtained results are statistically significant with p-value<0.01 and not due to chance.

In the result reported in this experiments, we are considering the median value for each approach through 31 independent run. The Wilcoxon rank sum test allows verifying whether the results are statistically different or not, however it does not give an idea about the difference magnitude. In order to quantify the latter, we compute the effect size by using the Cohen's *d* statistic [184]. 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 $d \geq 0.8$. We have computed the effect size values for the different comparisons and we concluded that our CRO approach with prioritization has mainly: (1) medium effect size values against population-based metaheuristics under comparison, and (2) large effect size values against single solution-based ones.

As described in Table 5.3 and Figure 5.6, the majority of suggested refactorings by our approach improve significantly the code quality with good code-smell correction scores compared to both CRO without prioritization and GA-based approach. For the five studied systems, our approach proves significant performance by fixing, on average, 90% of all existing code-smells, whereas, only 84% and 82% for the other two approaches while focusing on fixing the prioritized code-smells. For instance, for JFreeChart, 92% (24 over 26) of Blobs, 94% (16 over 17) of spaghetti code, 79% of functional decomposition (11

Systems	Approach	Code-smell Correction Ratio (CCR)							*CCR (all code-smells)*
		Blob	**Spaghetti code**	**Functional decomposition**	**Data class**	**Feature Envy**	**Schizophrenic class**	**Shotgun Surgery**	
Xerces-J	CRO (our approach)	94% (29|31)	92% (12|13)	86% (12|14)	90% (26|29)	90% (65|72)	100% (10|10)	100% (2|2)	91% (156|171)
	CRO without prioritization	84% (26|31)	85% (11|13)	93% (13|14)	97% (28|29)	85% (61|72)	80% (8|10)	50% (1|2)	87% (148|171)
	GA-based approach	84% (26|31)	77% (10|13)	86% (12|14)	97% (28|29)	83% (60|72)	70% (7|10)	50% (1|2)	84% (144|171)
JFreeChart	CRO (our approach)	92% (24|26)	94% (16|17)	79% (11|14)	89% (24|27)	85% (17|20)	92% (11|12)	100% (0|0)	89% (103|116)
	CRO without prioritization	81% (21|26)	76% (13|17)	79% (11|14)	100% (27|27)	80% (16|20)	83% (10|12)	100% (0|0)	84% (98|116)
	GA-based approach	73% (19|26)	82% (14|17)	79% (11|14)	100% (27|27)	80% (16|20)	100% (12|12)	100% (0|0)	85% (99|116)
GanttProject	CRO (our approach)	100% (7|7)	83% (5|6)	100% (18|18)	79% (11|14)	86% (6|7)	100% (1|1)	100% (0|0)	91% (48|53)
	CRO without prioritization	71% (5|7)	83% (5|6)	100% (18|18)	93% (13|14)	57% (4|7)	0% (0|1)	100% (0|0)	85% (45|53)
	GA-based approach	57% (4|7)	83% (5|6)	94% (17|18)	93% (13|14)	71% (5|7)	0% (0|1)	100% (0|0)	83% (44|53)
ArtofIllusion	CRO (our approach)	88% (15|17)	92% (11|12)	100% (8|8)	87% (27|31)	95% (42|44)	86% (12|14)	100% (1|1)	91% (116|127)
	CRO without prioritization	76% (13|17)	75% (9|12)	100% (8|8)	97% (30|31)	82% (36|44)	86% (12|14)	100% (1|1)	86% (109|127)
	GA-based approach	71% (12|17)	75% (9|12)	88% (7|8)	94% (29|31)	84% (37|44)	86% (12|14)	0% (0|1)	83% (106|127)
JHotDraw	CRO (our approach)	100% (4|4)	100% (3|3)	100% (5|5)	50% (2|4)	100% (4|4)	80% (4|5)	100% (0|0)	88% (22|25)
	CRO without prioritization	75% (3|4)	100% (3|3)	80% (4|5)	100% (4|4)	75% (3|4)	60% (3|5)	100% (0|0)	80% (20|25)
	GA-based approach	50% (2|4)	67% (2|3)	100% (5|5)	100% (4|4)	50% (2|4)	60% (3|5)	100% (0|0)	72% (18|25)
Average (all code-smells)	CRO (our approach)	95%	92%	93%	79%	91%	91%	100%	90%
	CRO without prioritization	78%	84%	90%	97%	76%	62%	90%	84%
	GA-based approach	67%	77%	89%	97%	74%	63%	70%	82%

Table 5.3 - Refactoring results: code-smells correction score.

over 14) are fixed. This score is higher than the one of the other approaches having respectively only 81%, 73% of Blobs, 76%, 82% of spaghetti code, 79%, 79% of functional decomposition in terms of CCR scores. Moreover, after applying the proposed refactoring operations, for all systems, we found that most of the fixed code-smells (87%) are related to important code fragments; however only 69% and 66% of ICR score are obtained by both other approaches that do not use prioritization (Table 5.4 and Figure 5.7). We also found that most of the fixed code-smells relies with the riskiest ones having a RCR average score of 92%; while both other approaches provide only an average of 85% and 84% of RCR as shown in Table 5.4 and Figure 5.7. Additionally, the obtained results demonstrates that using the proposed prioritization schema, 89% of severe code-smells were fixed; while both other approaches succeeded in fixing less than 81% of severe code-smells.

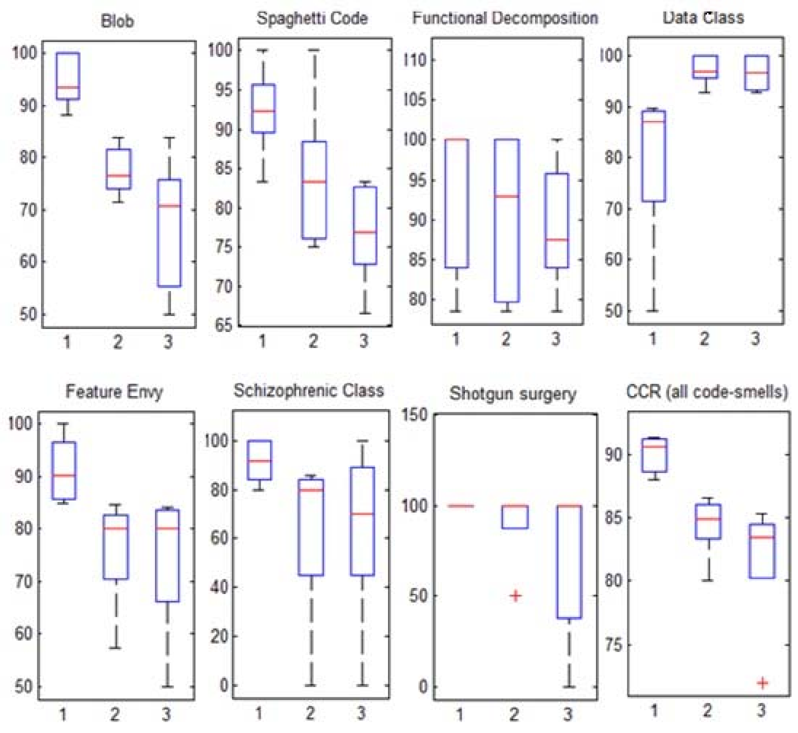


Figure 5.6 - Code-smells correction results per code-smell type for each studied systems for (1) CRO (our approach), (2) CRO without prioritization, and (3) GA-based approach.

Another important observation to highlight is that the majority of non-fixed code-smells obtained with both CRO without prioritization and GA-based approach are related to the Blob type, as shown in Figure 5.7. This type of code-smell usually requires a large number of refactoring operations and is then very difficult to correct without a specific mechanism (e.g., prioritization). On the other hand, the obtained CCR score related to data class is acceptable (an average of 79% in all systems); however we noticed that this score is less than the ones obtained by both other approaches. Thus, the loss in the data class correction ratio is largely compensated by the significant improvement in terms of importance, risk and severity scores as shown in Figure 5.7. In fact, this low score is due to the fact that, data class is not prioritized in our experiments, we assign data classes the lowest priority score (equals to 1) unlike the Blob code-smell, as described in Section 5.3.2. This score is assigned according to developers' preferences. Moreover, in general, data classes do not experience changes frequently during the development and maintenance since it contains mainly data and performs no processing on these data (contains mainly setters and getters). To this end, the importance score related to this code-smell is very low. On the contrary, as shown in Table 6, all the detected shotgun surgery code-smells are fixed (a CCR score of 100%). This is mainly due to the fact that shotgun surgery are extensively connected to a large number of external methods calling it having large and widespread impact of a change. Consequently its importance score is very high, and therefore, it will be more prioritized.

Moreover, to ensure the efficiency and usefulness of our approach, we verified manually the feasibility of the different proposed refactoring sequences for each system. We applied the proposed refactorings using Eclipse IDE. Some semantic errors (programs behavior) were found. When a semantic error is found manually, we consider the operations related to this change as a bad recommendation. We calculate a correctness precision score (ratio of possible refactoring operations over the number of proposed refactoring) as one of the performance indicators of our algorithm. An average of 70% of refactorings is feasible. This score is comparable of the one of both other approaches.

Systems	Approach	ICR (%)	RCR (%)	SCR (%)	RP (%)
Xerces-J	CRO (our approach)	89%	90%	89%	76% (230|302)
	CRO without prioritization	72%	86%	82%	72% (245|341)
	GA-based approach	61%	83%	84%	73% (261|359)
JFreeChart	CRO (our approach)	81%	91%	90%	64% (152|238)
	CRO without prioritization	50%	86%	83%	64% (151|236)
	GA-based approach	53%	85%	81%	64% (155|241)
GanttProject	CRO (our approach)	87%	93%	87%	67% (147|221)
	CRO without prioritization	67%	83%	75%	66% (145|219)
	GA-based approach	60%	81%	76%	69% (166|242)
ArtofIllusion	CRO (our approach)	85%	92%	91%	71% (205|288)
	CRO without prioritization	72%	84%	84%	70% (176|251)
	GA-based approach	70%	85%	84%	72% (180|249)
JHotDraw	CRO (our approach)	95%	94%	86%	72% (146|203)
	CRO without prioritization	82%	84%	73%	73% (160|218)
	GA-based approach	85%	85%	77%	74% (147|198)
Average (all systems)	CRO (our approach)	87%	92%	89%	70%
	CRO without prioritization	69%	85%	80%	69%
	GA-based approach	66%	84%	81%	70%

Table 5.4 - Refactoring results: importance, risk, severity and RP scores.

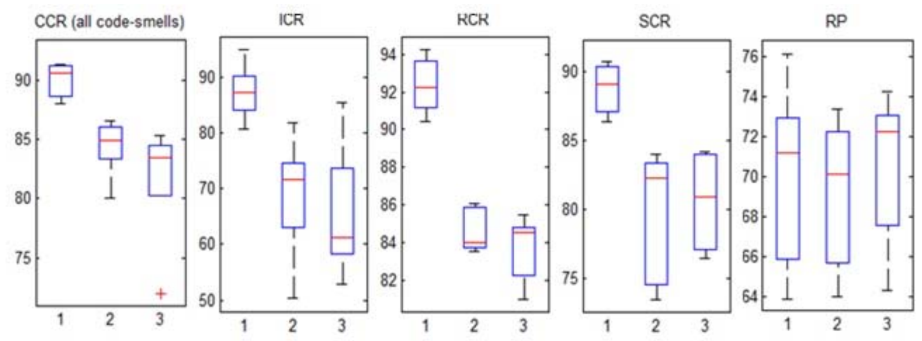


Figure 5.7 - Refactoring comparison results for the five systems for (1) CRO (our approach), (2) CRO without prioritization, and (3) GA-based approach in terms of ICR, RCR, SCR, and RP.

To sum up, we have presented in Figure 5.7 the metric scores for all systems using boxplots. The majority of code-smells (90%), on average, were corrected using our approach which outperforms both CRO without prioritization and GA-based approache in terms of code-smells correction ratio. However, only for data classes the obtained results are slightly less than other approaches. In general, this kind of code-smells is less risky/important than other code-smells and not need an extensive correction effort by software engineers compared to the Blob. Hence, to fix data class, software maintainers can easily apply some refactorings such as inline class, move method/field to add new behavior/functionalities or merge data classes with other existing classes in the system. Although data-classes are not prioritized in our approach, we obtained an acceptable correction score. This is due to the fact that Blob are in general related to data classes; consequently, fixing Blobs can implicitly fix its related data classes. We also had good results in terms of importance, risk and severity correction scores. The majority of important, riskiest and severest code-smells were fixed, and most of the proposed refactoring sequences (70%) are coherent semantically.

To better evaluate our approach and to answer RQ4, we compare the results of the CRO-based approach with three different population and single-solution based evolutionary algorithms (GA, SA, and PSO) which have been shown to have good performance in solving different software engineering problems [90] [91]. For all algorithms, we use the same formulation given in Section 5.3.3 (solution representation, objective function, change operators, etc.) with the algorithms configuration described in section 3.4.4. Table 5.5 shows the comparison results among the median of solution's quality for each pair of algorithms using Wilcoxon rank sum test [170]. As shown in Table 5.5, at 99% of confidence level, the median values of CRO and GA; CRO and SA; as well as those of CRO and PSO are statistically different in terms of CCR, ICR, and RCR. However, in terms of RP, CRO and GA; and CRO and PSO are not. The comparison results, sketched in Table 5.5 and Figure 5.8 shows that CRO outperforms the other three algorithms in terms of CCR, ICR, and RCR while having similar performance in terms of RP (70%). For instance, using CRO, an average of 90% of code-smells are fixed, whereas, only 84%, 83% and 84% are obtained with GA, SA, and PSO. Moreover, in terms of ICR, CRO succeeded on fixing, 87% of important code-smells, while obtained ones for other algorithms are less than fixes less than 83%. Based on these results we can conjecture that CRO performs much better in comparison to GA, SA and PSO. Moreover, we notice that SA turns out to be the worst algorithm.

Systems	Algorithms	CCR (%)		ICR (%)		RCR (%)		SCR (%)		RP (%)	
		Score	*p-value*	*Score*	*p-value*	*Score*	*p-value*	*Score*	*p-value*	*Score*	*p-value*
Xerces-J	CRO	91%		89%		90%		89%		76%	
	GA	84%	< 0.01	86%	< 0.01	87%	< 0.01	88%	< 0.01	76%	0.7854
	SA	82%	< 0.01	86%	< 0.01	88%	< 0.01	87%	< 0.01	75%	0.68
	PSO	84%	< 0.01	85%	< 0.01	88%	< 0.01	87%	< 0.01	76%	0.5569
JFreeChart	CRO	89%		81%		91%		90%		64%	
	GA	81%	< 0.01	70%	< 0.01	87%	< 0.01	89%	< 0.01	64%	0.769
	SA	80%	< 0.01	69%	< 0.01	87%	< 0.01	88%	< 0.01	64%	0.661
	PSO	84%	< 0.01	72%	< 0.01	86%	< 0.01	88%	< 0.01	64%	0.487
GanttProject	CRO	91%		87%		93%		87%		67%	
	GA	87%	< 0.01	84%	< 0.01	90%	< 0.01	84%	< 0.01	67%	0.387
	SA	81%	< 0.01	83%	< 0.01	90%	< 0.01	85%	< 0.01	68%	0.489
	PSO	87%	< 0.01	84%	< 0.01	91%	< 0.01	86%	< 0.01	67%	0.23
ArtofIllusion	CRO	91%		85%		92%		91%		71%	
	GA	89%	< 0.01	83%	< 0.01	90%	< 0.01	90%	< 0.01	70%	0.369
	SA	88%	< 0.01	82%	< 0.01	90%	< 0.01	89%	< 0.01	71%	0.217
	PSO	88%	< 0.01	83%	< 0.01	89%	< 0.01	90%	< 0.01	70%	0.062
JHotDraw	CRO	88%		95%		94%		86%		72%	
	GA	84%	< 0.01	92%	< 0.01	90%	< 0.01	81%	< 0.01	71%	0.161
	SA	72%	< 0.01	91%	< 0.01	91%	< 0.01	83%	< 0.01	71%	0.169
	PSO	84%	< 0.01	92%	< 0.01	91%	< 0.01	87%	< 0.01	72%	0.178
Average (all systems)	CRO	90%		87%		92%		89%		70%	
	GA	85%		83%		89%		87%		70%	
	SA	81%		82%		89%		87%		70%	
	PSO	85%		83%		89%		88%		70%	

Table 5.5 - CCR, ICR, RCR, SCR and RP median values of CRO, GP, SA and PSO over 31 independent simulation runs.

The p-values of the Wilcoxon rank sum test indicate whether the median of the algorithm of the corresponding column (GA/SA/PSO) is statistically different from the CRO one with a 99% confidence level ($\alpha = 0.01$). A statistical difference, in terms of the obtained recall values, is detected when the p-value is less than or equal to 0.01.

Another observation is that GA and PSO can produce good refactoring solutions as CRO (but not better than CRO) for medium size systems (e.g.,GanttProject and JHotDraw). However, for large systems (e.g., Xerces, JFreeChart), the performance of CRO is significantly better than GA and PSO.

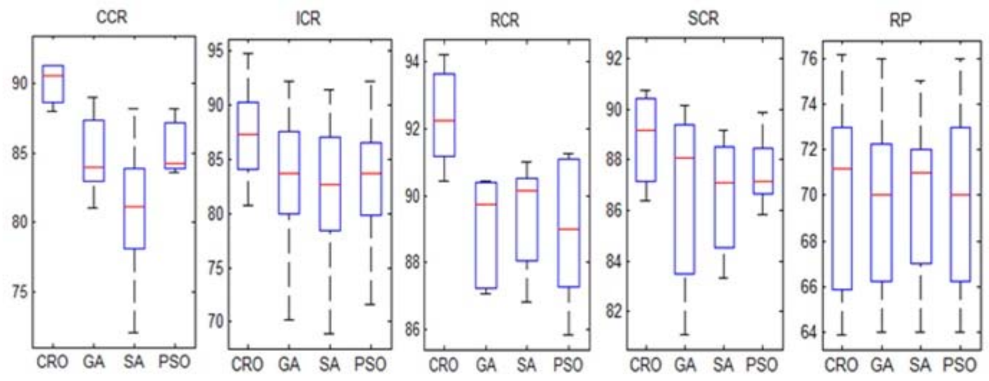


Figure 5.8 - CRO performance comparison with GA, SA and PSO.

Although time constraints is not a real challenge in our proposal, it is relevant to analyse the convergence speed when we compare metaheuristics. To this end, we performed 31 independent simulation runs on the same PC with Intel Core i5-2450M Processor and 4GB of RAM for each of the algorithms CRO, GA, SA and PSO. Hence, SA manipulates a single solution in each iteration, while GA and PSO control a population of solutions at a time. CRO is also a population-based metaheuristics; however, the number of manipulated solutions varies during a simulation run. Through 31 independent simulation run, we found that PSO converges faster than the other algorithms (an average of 48m12s). CRO is the second one in terms of convergence speed (an average of 48m18s over 31 run) while we record an average of 1h32m47s and 1h23m13s for respectively GA and SA.

To sum up, we can conclude that CRO outperforms other popular metaheuristic algorithms [98] [99] [100]. In fact, there are two reasons for the high convergence speed of CRO. The first is the ability for CRO to jump out of a local minimum, by the mean of the four elementary reaction operators, and quickly search other possible better results. The second is due to the efficient encoding scheme and the variety of change operators, which greatly explores the search space.

5.5 Discussions

Our experimental results provide evidence that our approach significantly outperforms two other approaches that do not use the prioritization for correcting code-smells. We also found that CRO performs much better than three other popular metaheuristic algorithms: GA, SA, and PSO. We also contrast the results of multiple executions with the execution time to evaluate the performance and the stability of our approach. Moreover, we evaluate the impact of the number of suggested refactorings on the CCR, ICR, RCR and SCR scores and the execution time through 31 independent simulation run. Over 31 independent simulation runs on JFreeChart, the average value of CCR, ICR, RCR, SCR and execution time for finding the optimal refactoring solution with the suitable prioritization schema was respectively 90%, 80%, 92%, 90% and 57min36s as shown in Figure 5.9. The standard deviation values was lower than 1. Moreover, the results of Figure 5.9, drawn for JFreeChart, show that the number of suggested refactorings does not affect the refactoring results. Thus, a higher number of operations in a solution do not necessarily mean that the results will be better. Thus, we could conclude that our approach is scalable from the performance standpoint, especially that quality improvements are not related in general to real-time applications where time-constraints are very important. In addition, the results accuracy is not affected by the number of suggested refactorings.

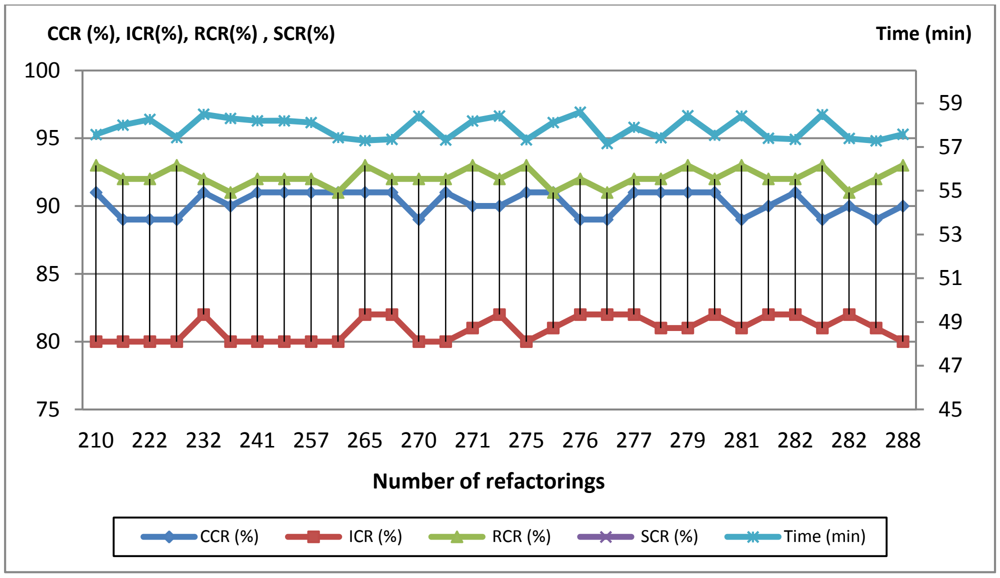


Figure 5.9 - Impact of the number of refactorings on multiple runs on JFreeChart.

Another important consideration is the refactoring operations distribution. We contrast that most of the suggested refactorings are related to move method, move field, and extract class for almost all of the studied systems. For instance, in JFreeChart, we had different distribution of different refactoring types as illustrated in Figure 5.10. We notice that the most suggested refactorings are related to moving code elements (fields, methods) and extract/inline class. This is mainly due to the code-smells types detected in JFreeChart and prioritized during the correction step. Most of code-smells are related to the Blob, functional decomposition and spaghetti code that need particular refactorings. For instance, to fix a Blob code-smell, the idea is to move elements from the Blob class to other classes (e.g., data classes) in order to reduce the number of functionalities from the Blob and add behavior to other classes or to improve some quality metrics such as coupling and cohesion. As such, refactorings like move field, move method, and extract class are likely to be more useful to correcting the bloc code-smell. Furthermore, before starting our experiments and analyzing our refactoring results, we expected that code-smell correction score for data classes will be very low; however we found that most of them are corrected with a good score (an average of 79%). This is mainly due to two reasons 1) data classes are to some extent easy to fix and they don't need lot of refactorings to be fixed; it is sufficient to add some functionalities/methods to them from other related classes, and 2) in general there is a structural relationship between data classes and Blobs [12] [70]; so that fixing Blobs can implicitly fix data classes related to them. Enforcing the correction of Blobs can implicitly increase the correction of data classes. As part of future work we plan to conduct a large empirical study to investigate the relationship between different types of code-smells, and between code-smell types and refactoring types.

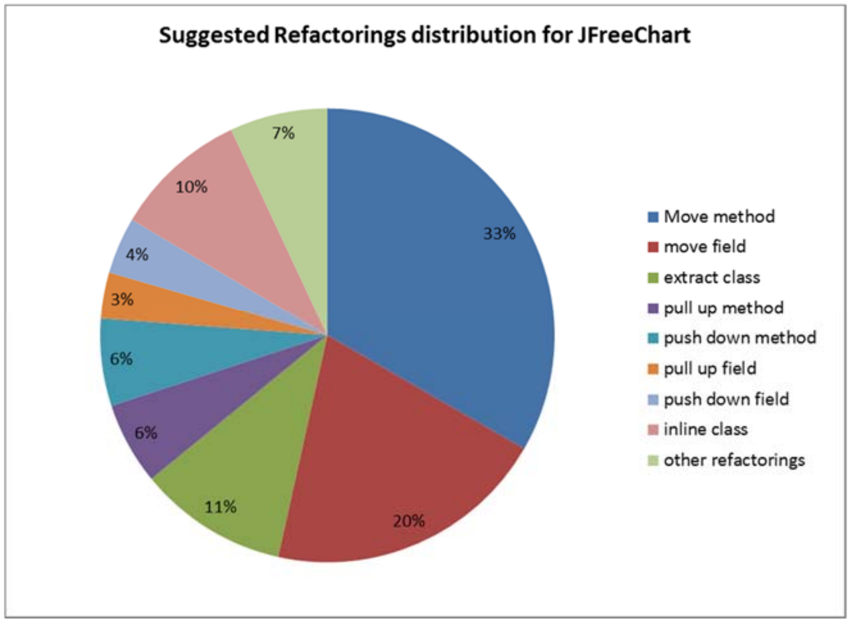


Figure 5.10 - Suggested refactorings distribution for JFreeChart.

5.6 Threats to validity

Some threats can affect the validity of our experiments. We explore in this section the factors that can bias our experimental 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.

Construct validity is concerned with the relationship between theory and what is observed. The manual evaluation of the feasibility of the suggested refactorings depends on the expertise of the developer and also it is a subjective process to make sure that a detected code-smell is fixed. Another construct validity can be related to the used code-smells detection rules we use to measure CCR. To mitigate this threat, we manually inspect and validate each detected code-smell. Moreover, our refactoring tool configuration is flexible and can support other state-of-the-art detection rules.

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 [170] with a 99% confidence level ($\alpha = 1\%$). However, despite we used the same stopping criteria, 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.

External validity refers to the generalizability of our findings. In this study, we performed our experiments on seven different code-smells types and five different widely-used open-source systems belonging to different domains and with different sizes, as described in Table 3. However, we cannot assert that our results can be generalized to industrial applications, other programming languages, and to other practitioners. Future replications of this study are necessary to confirm the generalizability of our findings.

5.7 Conclusion

This chapter presented a novel chemical reaction optimization-based approach to recommend refactoring solutions according to a prioritization schema. The aim is to fix code-smells while prioritizing the severest, riskiest and important code-smells taking into consideration software maintainers' preferences. The suggested refactorings succeed in fixing the majority of critical code-smells. Our experimental study on five medium and large scale software systems and seven code-smell types shows that the proposed approach is superior to two other approaches that do not use prioritization and maintainers preferences to automate the refactoring task. Moreover, the experimental results provide evidence that the proposed CRO-based approach was performs better than GA, SA, and PSO, the most popular metaheuristics in SBSE.

Despite the great advances in software refactoring in the last years, one of the most notable limitations of the majority of existing work, including our GA- and CRO-based approaches, is that they deal with the refactoring problem from a single perspective which is improving software quality while preserving the behavior. These two concerns drive the existing approaches to refactoring automation. However, these concerns are not enough to produce correct and consistent refactoring solutions. Many other criteria are also important to consider such as reducing the number of code changes, preserving the semantics of the design and not only the behavior, and maintaining the consistency with the change history. These observations lead us to deal with the refactoring recommending task as a multi-objective optimization problem. In Chapter 6, we present our multi-objective formulation for the refactoring task.

Part 3: Multi-objective software refactoring

In part 2 of this thesis, we described our contributions to single-objective refactoring recommendation. In this part, we formulate refactoring recommending task as a multi-objective optimization problem. To the best of our knowledge, this is the first attempt to deal with refactoring recommending task as a multi-objective optimization problem. In our multi-objective formulation we consider two scenarios for software practitioners: 1) the main goal is to preserve the design semantics while fixing code-smells; and 2) the main goal is to improve software quality from different perspectives.

For the first scenario, we introduce a novel multi-objective search-based approach that aims at finding the optimal sequence of refactorings to fix code-code-smells while preserving the design semantics from different perspectives, 1) minimizes code changes required to apply refactoring, 2) preserves semantic coherence, and 3) maintain the conformance with prior refactorings applied in previous versions.

For the second scenario, we introduce a new multi-objective formulation to refactoring recommending task. The aim of our approach, called MORE (Mutli-Objective REfactoring recommending) is to improve software quality from different perspectives: 1) improve quality indicators (i.e., flexibility, maintainability, etc.), 2) fix “bad” design practices (i.e., code-smells), and 3) promote “good” design practices (i.e., design patterns).

Chapter 6: A Multi-objective approach for recommending software refactoring

6.1 Introduction

In Chapter 4 and Chapter 5 we formulated the refactoring recommending task as a single-objective optimization problem to fix code-smells. However, while it is important to suggest refactorings that improve software quality, many other criteria are also important to consider to obtain efficient refactoring strategies. These criteria include reducing the number of code changes, preserving the semantic coherence of the software design, and maintaining the consistency with the previous changes. In this chapter, we deal with refactoring recommending task as a multi-objective optimization problem. We introduce a multi-objective search-based approach that aims at finding the optimal sequence of refactorings that 1) improves the quality by minimizing as much as possible the number of code-smells, 2) minimizes code changes required to fix those smells, 3) preserves semantic coherence, and 4) maximizes the consistency with previous changes. To the best of our knowledge, this is the first attempt to deal with refactoring recommending task as a multi-objective optimization problem. The primary contributions of this chapter can be summarized as follows:

1. We introduce a novel multi-objective formulation of the refactoring recommendation task using four different criteria. To this end, we use the non-dominated sorting genetic algorithm (NSGA-II).
2. We reports the results of an empirical study of our multi-objective approach using a benchmark of six medium and large size open-source systems, and six commonly occurring code-smell types through an empirical study conducted with experts. We found that, in addition to fixing code-smells, the proposed refactorings succeed in preserving the semantic coherence of the code, with an acceptable level of code change score while reusing knowledge from recorded refactorings applied in the past to similar contexts.

This chapter is structured as follows: Section 6.2 is dedicated to present the different challenges that we address through a motivating example. Section 6.3 presents our approach and explains how we adapted NSGA-II. Section 6.4 presents quantitative and qualitative evaluation results, while Section 6.5 discusses the obtained results. Section 6.6 discusses the threats to validity. Finally, Section 6.7 concludes and presents plans for future work.

6.2 Challenges in automated refactoring recommending

Even though most of the existing refactoring recommendation approaches are powerful enough to suggest refactoring solutions to be applied, several issues are still need to be addressed.

6.2.1 Problem statement

Our approach is designed to address mainly the following problems:

Quality improvement: Most of the existing approaches [20] [21] [22] [41] [45] consider refactoring as the process to improve code quality by improving structural metrics. However, these metrics can be conflicting and it is difficult to find a compromise between them. For example, moving methods to reduce the size or complexity of a class may increase the global coupling. Furthermore, improving some quality metrics does not guarantee that detected code-smells are fixed. Moreover, there is no consensus about the set of metrics that need to be improved in order to fix code-smells. Indeed, the same type of code-smells can be fixed by improving completely different metrics.

Semantics preservation: In object-oriented programs, objects reify domain concepts and/or physical objects, implementing their characteristics and behavior. Unlike other programming paradigms, grouping data and behavior into classes is not guided by development or maintenance considerations. Methods and fields of classes characterize the structure and behavior of the implemented domain elements. Consequently, a program could be syntactically correct, implement the appropriate behavior, but violate the domain semantics if the reification of domain elements is incorrect. During the initial design/implementation, programs usually capture well the domain semantics when object-oriented principles are applied. However, when these programs are (semi-)automatically modified/refactored during maintenance, the adequacy with regards to domain semantics could be compromised.

Existing approaches suggests refactorings mainly with the perspective of improving only some design/quality metrics. However, this may not be sufficient. We need to preserve the rationale behind why and how code elements are grouped and connected when applying refactoring. Indeed, the refactored program could be syntactically correct, implement the correct behavior, but be semantically incoherent. For example, a refactoring solution might move a method *calculateSalary()* from the class *Employee* to the class *Car*. This refactoring could improve the program structure by reducing the complexity and coupling of the class *Employee* and satisfy the pre- and post-conditions to preserve program behavior. However, having a method *calculateSalary()* in the class *Car* does not make any sense from the domain semantics standpoint, and is likely to lead to comprehension problems in the future. Thus, semantics preservation is an important issue to consider when applying refactoring.

Code changes: When applying refactorings, various code changes are performed. The amount of code changes corresponds to the number of code elements (e.g., classes, methods, fields, relationships, field references, etc.) modified through adding, deleting, or moving operations. Minimizing code changes when suggesting refactorings is important to reduce the effort and help developers in understanding the modified/improved program. In fact, most developers want to keep as much as possible the original design structure when fixing code-smells [1]. However, improving software quality and reducing code changes are conflicting. In some cases, fixing some code-smells corresponds to changing radically a large portion of the system or is sometimes equivalent to re-implementing a large part of the system. Indeed, a refactoring solution that fixes all code-smells is not necessarily the optimal one due to the high code adaptation/modification that may be required.

Consistency with development/maintenance history: The majority of existing work does not consider the history of changes applied in the past when proposing new refactoring solutions. However, the history of code changes can be helpful in increasing the confidence of new refactoring recommendations. To better guide the search process, recorded code changes applied in the past can be considered when proposing new refactorings in similar contexts. This knowledge can be combined with structural and semantic information to improve the automation of refactoring suggestions.

In addition, code fragments that have previously been modified at the same time period are likely to be semantically connected (e.g., refer to the same feature). Furthermore, fragments that have been extensively refactored in the past have a high probability of being refactored again in the future. Moreover, the code to refactor can be similar to some refactoring patterns that are to be found in the development history, thus, developers can easily adapt them.

6.2.2 Motivating example

To illustrate some of the above mentioned issues, Figure 6.1 shows a concrete example extracted from JFreeChart[20] v1.0.9, a well-known Java open-source charting library. We consider a design fragment containing four classes *XYLineAndShapeRenderer*, *XYDotRenderer, SegmentedTimeline,* and *XYSplineRenderer*. Using code-smells detection rules [7], the class *XYLineAndShapeRenderer* is detected as a code-smell: Blob (i.e., a large class that monopolizes the behavior of a large part of the system).

We consider the scenario of a refactoring solution that consists of moving the method *drawItem()* from the class *XYLineAndShapeRenderer* to the class *SegmentedTimeline*. This refactoring can improve the design quality by reducing the number of functionalities in this Blob class. However, from the design semantics standpoint, this refactoring is incoherent since *SegmentedTimeline* functionalities are related to presenting a series of values to be used for a curve axis (mainly for *Date* related axis) and not for the task of drawing

[20] http://www.jfree.org/jfreechart/

objects/items. Based on semantic and structural information, using respectively a semantic lexicon [185], and cohesion/coupling [29], many other target classes are possible including *XYDotRenderer* and *XYSplineRenderer*. These two classes have approximately the same structure that can be formalized using quality metrics (e.g., number of methods, number of attributes, etc.) and their semantic similarity is close to *XYLineAndShapeRenderer* using a vocabulary-based measure. Thus, moving elements between these three classes is likely to be semantically coherent and meaningful. On the other hand, from previous versions of JFreeChart, we recorded that there are some methods such as *drawPrimaryLinePath()*, *initialise()*, and *equals()* that have been moved from the class *XYLineAndShapeRenderer* to

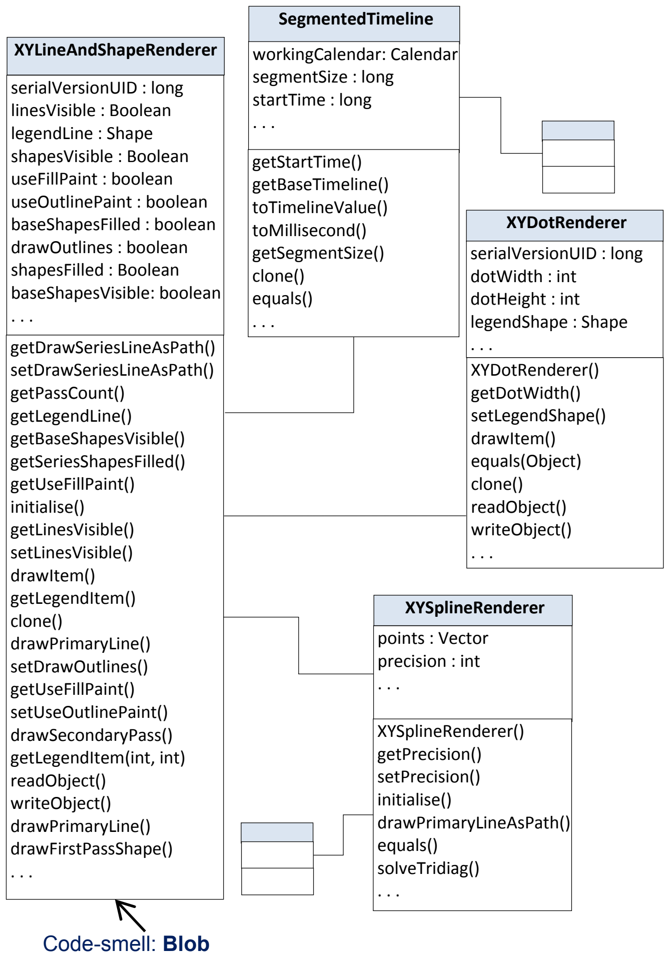


Figure 6.1 - Design fragment extracted from JFreeChart v1.0.9.

the class *XYSplineRenderer*. As a consequence, moving methods and/or attributes from the class *XYLineAndShapeRenderer* to the class *XYSplineRenderer* has higher correctness probability than moving methods or attributes to the class *XYDotRenderer* or *SegmentedTimeline.*

Based on these observations, we believe that it is important to consider additional objectives instead of using only structural metrics to ensure quality improvement. However, in most of the existing work, design semantics, amount of code changes, and development history are not considered. Improving code structure, minimizing semantic incoherencies, reducing code changes, and maintaining the consistency with development change history are conflicting goals. In some cases, improving the program structure could provide a design that does not make sense semantically or could change radically the initial design. For this reasons, an effective refactoring strategy needs to find a compromise between all of these objectives. These observations are the motivation of the work described in this chapter.

6.3 Approach

This section presents our approach. In Section 6.3.1, we present an overview of our approach. Section 6.3.2 describes how we formulated the refactoring recommending task as a multi-objective optimization problem. Section 6.3.3 presents our semantic measures, while Section 6.3.4 describes how we adapted NSGA-II.

6.3.1 Approach overview

Our approach aims at exploring a large search space to find refactoring solutions, i.e., a sequence of refactoring operations, to fix code-smells. The search space is determined not only by the number of possible refactoring combinations, but also by the order in which they are applied. A heuristic-based optimization method is used to generate refactoring solutions. We have four objectives to optimize: 1) maximize quality improvement (code-smells correction); 2) minimize the number of semantic incoherencies by preserving the way how code elements are semantically grouped and connected together; 3) minimize code changes needed to apply the refactorings; and 4) maximize the consistency with development change history. We thus consider the refactoring task as a multi-objective optimization problem using the non-dominated sorting genetic algorithm (NSGA-II) [24].

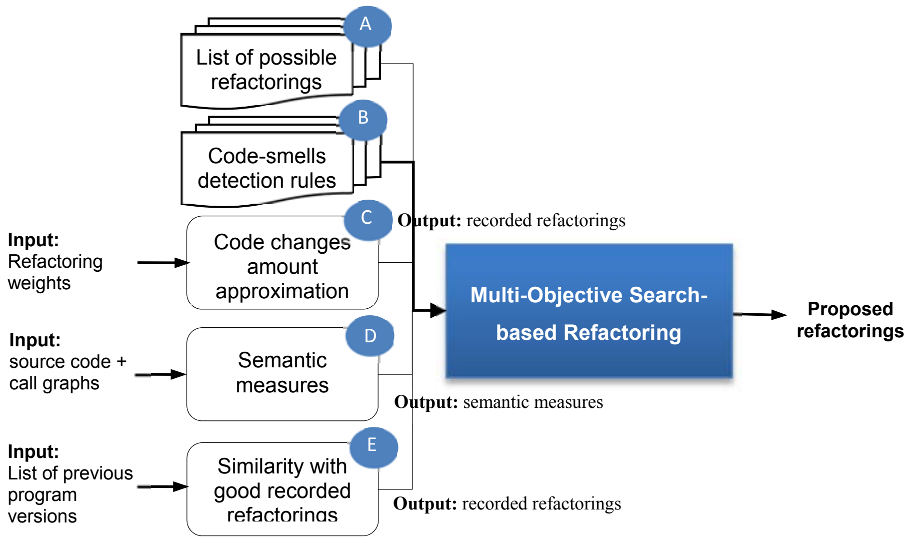

A
List of possible refactorings
B
Code-smells detection rules
C
Code changes amount approximation
D
Semantic measures
E
Similarity with good recorded refactorings
Input: Refactoring weights
Input: source code + call graphs
Input: List of previous program versions
Output: recorded refactorings
Output: semantic measures
Output: recorded refactorings
Multi-Objective Search-based Refactoring
Proposed refactorings

Figure 6.2 - Multi-Objective Search-based Refactoring.

The general structure of our approach is sketched in Figure 6.2. It takes as input the source code of the program to be refactored, a list of possible refactorings that can be applied (label A), a set of code-smells detection rules (label B) [7], our technique for approximating code changes needed to apply refactorings (label C), a set of semantic measures described in Section 6.3.3 (label D), and a history of applied refactorings to previous versions of the system (label E). Our approach generates as output the near-optimal sequence of refactorings that improves software quality by minimizing as much as possible the number of code-smells, minimizing code changes required to apply the refactorings, preserving design semantics, and maximizing the consistency with development change history. In the following, we describe the formulation of the four objectives to optimize.

6.3.2 Modeling refactoring recommending as a multi-objective problem

We consider the following criteria:

6.3.2.1 Quality

The Quality criterion is evaluated using the *Quality* formula given below. The quality value increases when the number of code-smells in the code is reduced after refactoring. This formula returns a real value in the range [0,1] that represents the ratio of the number of fixed code-smells (detected using code-smells detection rules) over the initial number of detected code-smells before refactoring. The detection of code-smells is based on metric-based rules according to which a code fragment can be classified as a code-smell or not (without a probability/risk score), i.e., 0 or 1, as formulated in Chapter 3 [28].

$$Quality = \frac{\#\ corrected\ code\_smells}{intitial\ number\ of\ code\_smells\ before\ refactoring}$$

6.3.2.2 Code changes

Refactoring Operations (ROs) are classified into two types: Low-Level ROs (LLR) and High-Level ROs (HLR) [28]. A *HLR* is a sequence of two or more *ROs*. An *LLR* is an elementary refactoring consisting of just one basic *RO* (e.g., “Create Class”, “Delete Method”, “Add Field”). The weight w_i for each *RO* is an integer number that can be 1, 2, or 3 depending on code fragment complexity and change impact [7]. For a refactoring solution that contains *n* ROs, the code changes score is computed as:

$$Code\_changes = \sum_{i=1}^{n} w_i$$

6.3.2.3 Similarity with recorded code changes

The idea is to encourage the use of refactorings that are similar to those applied to the same code fragments in the past. To calculate the similarity score between a proposed refactoring operation and a recorded code change, we use the following function:

$$Sim\_refactoring\_history(RO) = \sum_{j=1}^{n} w_j$$

where n is the number of recorded refactoring operations applied to the system in the past, and w_j is a refactoring weight that reflects the similarity between the suggested refactoring operation (RO) and the recorded refactoring operation RO_i. The weight w_i is computed as follows: if the refactoring operations being compared are exactly the same type and applied to the same locations (e.g., “Move Method” between the source and target same classes), the weight w_i = 2. If the refactoring operations being compared are similar (we consider two refactoring operations as similar if one of them is composed of the other or if their implementations are similar). Some complex refactoring operations, such as “Extract Class”, can be composed of other refactoring operations such as “Move Method”, “Move Field”, “Create New Class”, etc., the weight w_i = 1. Otherwise, w_i = 0.

6.3.2.4 Semantics

In this section, we do not address the issue of operational semantics that is formulated via pre/post-conditions. Instead, our goal is to recommend refactorings to be applied by developers and not automatic application of refactorings. In fact, Tokuda and Batory [186] found that the pre-conditions originally proposed by Opdyke [17] were not sufficient to guarantee behavior preservation for C++ programs. There are several testing approaches that have found hundreds of bugs in refactoring tools from state-of-the-art and industry [187] [188] [189]. We focus in this section on measures to ensure preservation for design semantics coherence during refactoring.

As far as we know, until now there is no consensual way to investigate whether refactoring can preserve the semantic coherence of the original design. Hence, we formulate semantics preservation through a meta-model in which we describe the necessary concepts from a perspective to help in automating refactoring recommendation. We also provide a terminology that will be used throughout this chapter. Figure 6.3 shows the semantic-based refactoring meta-model. The class *Refactoring* represents the main entity in the meta-model. As mentioned earlier, we classify refactoring operations into two types:

low-level ROs (LLR) and high-level ROs (HLR). A *LLR* is an elementary/basic program transformation for adding, removing, and renaming program elements (e.g., “Add Method”, “Remove Field”, “Add Relationship”). LLRs can be combined to perform more complex refactoring operations (HLR*s*) (e.g., “Move Method”, “Extract Class”). A *HLR* consists of a sequence of two or more LLRs or HLRs; for example, to perform “Extract Class” we need to “Create New Empty Class” and apply a set of “Move Method” and “Move Field” operations.

To apply a refactoring operation we need to specify which *actors*, i.e., code fragments, are involved in this refactoring and which *roles* they play when performing the refactoring operation. As illustrated in Figure 6.3, an *actor* can be a *package*, *class*, *field*, *method*, *parameter*, *statement*, or *variable*. In Table 6.1, we specify for each refactoring operation the involved actors and their roles. In addition to this list of complex refactorings, we considered also in our experiments the Rename refactoring to satisfy the pre/post-conditions of some complex refactorings. Another complex refactoring considered in our experiments is “Extract Method”. We used Soot [190], a java optimization framework, to parse the extracted code for references to any variables (local variables and parameters to the method) that are local in the source method. Different constraints are related to the application of an extract method refactoring as described in [86] [87] [88] (like most of other types of refactoring):

- Local variables used only within this extracted code should be declared in the target method as local variables.
- Parameters used within this extracted code should be considered as parameters in the target method.
- Local variables used (read) within the extracted code should be considered as parameters to the target method.
- Local variable or parameters used (write) within the extracted code should be used as a return type from the target method.
- If the target method do not use (write) any of the local variables or parameters of the

source method, then the return type of the target method should be "void".

In the case when the extracted code uses (write) more than one parameter and/or local variable then it is not possible to apply extract method to the selected fragment. Therefore, another appropriate code fragment should be selected.

Refactoring operation	**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	method
Inline class	class	source class, target class
Extract method	class	source class, target class
	method	source method, new method
	statement	moved statements
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

Table 6.1 - Refactoring examples and their involved actors and roles.

6.3.3 Semantic measures

To approximate semantics preservation, we define the following measures:

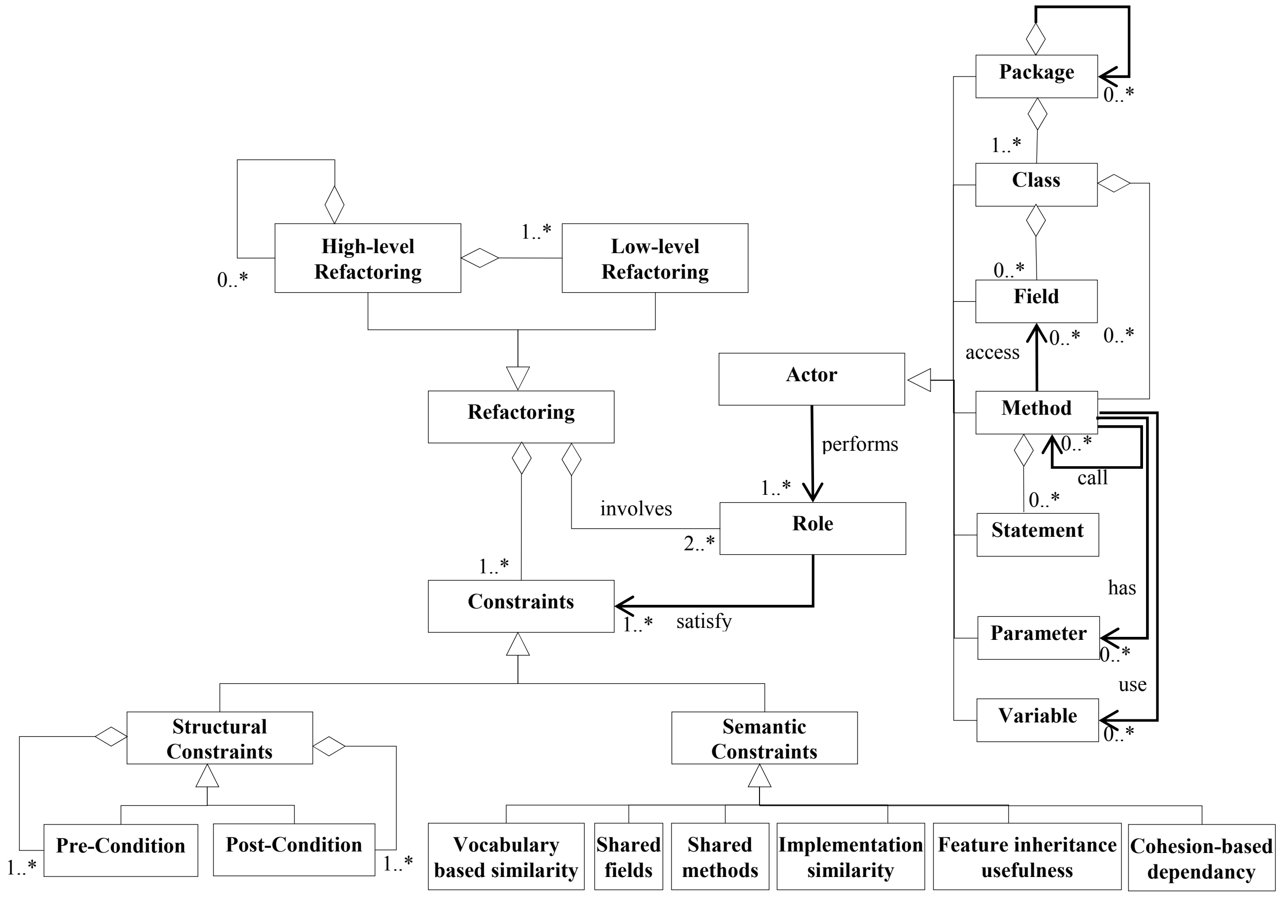


Figure 6.3 - Semantics-based refactoring meta-model.

Vocabulary-based similarity (VS)

This kind of similarity is interesting to consider when moving methods, fields, or classes. For example, when a method has to be moved from one class to another, the refactoring would make sense if both actors (source class and target class) use similar vocabularies [29]. The vocabulary could be used as an indicator of the semantic similarity between different actors that are involved when performing a refactoring operation. We start from the assumption that the vocabulary of an actor is borrowed from the domain terminology and therefore can be used to determine which part of the domain semantics an actor encodes. Thus, two actors are likely to be semantically similar if they use similar vocabularies.

The vocabulary can be extracted from the names of methods, fields, variables, parameters, types, etc. Tokenisation is performed using the Camel Case Splitter, which is one of the most used techniques in Software Maintenance tools for the preprocessing of identifiers. A more pertinent vocabulary can also be extracted from comments, commit information, and documentation. We calculate the semantic similarity between actors using an information retrieval-based technique, namely cosine similarity, as shown in the formula below. 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 semantic similarity between two actors c_1 and c_2 is determined as follows:

$$Sim(c_1, c_2) = \cos(\vec{c_1}, \vec{c_2}) = \frac{\vec{c_1} \cdot \vec{c_2}}{\|\vec{c_1}\| * \|\vec{c_2}\|} = \frac{\sum_{i=1}^{n}(w_{i,1} * w_{i,2})}{\sqrt{\sum_{i=1}^{n}(w_{i,1})^2}\sqrt{\sum_{i=1}^{n}(w_{i,2})^2}} \in [0,1]$$

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

6.3.3.1 Dependency-based similarity (DS)

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

Shared Method Calls (SMC) that can be captured from call graphs derived from the whole program using CHA (Class Hierarchy Analysis) [190]. 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 subtype or sometimes supertype of the declared type of c. For a pair of actors, shared calls are captured through this graph by identifying shared neighbours of nodes related to each actor. We consider both, shared call-out and shared call-in. To measure shared call-out and shared call-in between two actors c_1 and c_2 (e.g., two classes), we define the following formula respectively:

$$sharedCallOut(c_1, c_2) = \frac{|callOut(c_1) \cap callOut(c_2)|}{|callOut(c_1) \cup callOut(c_2)|}$$

$$sharedCallIn(c_1, c_2) = \frac{|callIn(c_1) \cap callIn(c_2)|}{|callIn(c_1) \cup callIn(c_2)|}$$

Shared method call is defined as the average of shared call-in and shared call-out.

Shared Field Access (SFA) that 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 code elements are semantically related if they read or modify the same fields. The rate of shared fields (read or modified) between two actors c_1 and c_2 is calculated as follows:

$$sharedFieldsRW(c_1, c_2) = \frac{|fieldRW(c_1) \cap fieldRW(c_2)|}{|fieldRW(c_1) \cup fieldRW(c_2)|}$$

where *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 can be easily computed.

6.3.3.2 Implementation-based similarity (IS)

For some refactorings like "Pull Up Method", methods having similar implementations in all subclasses of a super class should be moved to the super class [1].

The implementation similarity of the methods in the subclasses is investigated at two levels: signature level and body level. To compare the signatures of methods, a semantic comparison algorithm is applied. It takes into account the methods names, the parameter lists, and return types. Let *Sig(m_i)* be the signature of method m_i. The signature similarity for two methods m_1 and m_2 is computed as follows:

$$Sig\_sim(m_1, m_2) = \frac{|Sig(m_1) \cap Sig(m_2)|}{|Sig(m_1) \cup Sig(m_2)|}$$

To compare methods bodies, we use Soot [190], a Java optimization framework, which compares the statements in the body, the used local variables, the exceptions handled, the call-outs, and the field references. Let *Body(m)* (set of statements, local variables, exceptions, call-outs, and field references) be the body of method *m*. The body similarity for two methods m_1 and m_2 is computed as follows:

$$Body\_sim(m_1, m_2) = \frac{|Body(m_1) \cap Body(m_2)|}{|Body(m_1) \cup Body(m_2)|}$$

The implementation similarity between two methods is the average of their *Sig_Sim* and *Body_Sim* values.

6.3.3.3 Feature inheritance usefulness (FIU)

This factor is useful when applying the "Push Down Method" and "Push Down Field" operations. In general, when method or field is used by only few subclasses of a super class, it is better to move it, i.e., push it down, from the super class to the subclasses using it [1]. To do this for a method, we need to assess the usefulness of the method in the subclasses in which it appears. We use a call graph and consider polymorphic calls derived using XTA (Separate Type Analysis) [205]. XTA is more precise than CHA by giving a more local view of what types are available. We are using Soot [190] as a standalone tool to implement and test all the program analysis techniques required in our approach. The inheritance usefulness of a method is defined as follows:

$$FIU(m, c) = 1 - \frac{\sum_{i=1}^{n} call(m, i)}{n}$$

where *n* is the number of subclasses of the superclass *c*, *m* is the method to be pushed down, and *call* is a function that return 1 if *m* is used (called) in the subclass *i* and 0 otherwise.

For the refactoring operation "Push Down Field", a suitable field reference analysis is used. The inheritance usefulness of a field is defined as follows:

$$FIU(f, c) = 1 - \frac{\sum_{i=1}^{n} use(f, c_i)}{n}$$

where *n* is the number of subclasses of the superclass *c*, *f* is the field to be pushed down, and *use* is a function that return 1 if *f* is used (read or modified) in the subclass c_i and 0 otherwise.

6.3.3.4 Cohesion-based dependency (CD)

We use a cohesion-based dependency measure for the "Extract Class" refactoring operation. The cohesion metric is typically one of the important metrics used to identify code-smells. However, the cohesion-based similarity that we propose for code refactoring, in particular when applying extract class refactoring, is defined to find a cohesive set of methods and attributes to be moved to the newly extracted class. A new class can be extracted from a source class by moving a set of strongly related (cohesive) fields and methods from the original class to the new class. Extracting this set will improve the cohesion of the original class and minimize the coupling with the new class. Applying the "Extract Class" refactoring operation on a specific class will result in this class being split into two classes. We need to calculate the semantic similarity between the elements in the original class to decide how to split the original class into two classes.

We use vocabulary-based similarity and dependency-based similarity to find the cohesive set of actors (methods and fields). Consider a source class that contains *n* methods $\{m_1, \ldots m_n\}$ and *m* fields $\{f_1, \ldots f_m\}$. We calculate the similarity between each pair of elements (method-field and method-method) in a cohesion matrix as shown in Table 6.2.

The cohesion matrix is obtained as follows: 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. The column "Average" contains the average of similarity values for each line. The suitable set of methods and fields 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 threshold equals to 0.5.

	f_1	**f_2**	**…**	**f_m**	**m_1**	**m_2**	**…**	**m_n**	**Average**
m_1	1	0		1	1	0.15		0.1	0.42
m_2	**0**	**1**		**1**	**1**	**1**		**0**	**0.6**
. . .									
m_n	1	0		0	0.6	0.2		1	0.32

Table 6.2 - Example of a cohesion matrix.

The most notable limitation of the existing works in software refactoring is that the definition of semantic preservation is closely related to behaviour preservation. Preserving the behavior does not means that the design semantics of the refactored program is also preserved. Another issue is that the existing techniques are limited to a small number of refactorings and thus it could not be generalized and adapted for an exhaustive list of refactorings. Indeed, semantics preservation is still hard to ensure, and to the best of our knowledge, until now, there is no pragmatic technique or empirical study to prove whether the semantics of the refactored program is preserved.

6.3.4 NSGA-II for refactoring recommending

This section is dedicated to describe how we formulated the refactoring recommending problem as a multi-objective optimization problem using NSGA-II (cf. Section 2.3.4).

One key element when applying a search-based technique is to find a suitable mapping between the problem to solve and the techniques to use. Applying NSGA-II to a specific problem requires specifying the following elements: representation of a solution, generation of the initial population, the fitness function to evaluate the candidate solutions, the selection of the fittest solutions, and the change operators to derive new solutions from existing ones. In our approach, these elements are defined as follows:

a) Solution representation

In our NSGA-II design, we use the same vector-based solution representation adopted in our GA adaptation. The description of our solution representation is detailed in Section 3.3.2.

b) Creation of the initial population of solutions

To generate an initial population, we start by defining the maximum vector length (maximum number of operations per solution). The vector length is proportional to the number of refactorings that are considered and the size of the program to be refactored. A higher number of operations in a solution do not necessarily mean that the results will be better. Ideally, a small number of operations can be sufficient to provide good solutions. This parameter can be specified by the user or derived randomly from the sizes of the program and the given refactoring list. During the creation, the solutions have random sizes inside the allowed range. To create the initial population, we normally generate a set of PopSize solutions randomly in the solution space.

c) Objective functions

After creating a solution, it should be evaluated to quantify its ability to solve the problem under consideration. Since we have four objectives to optimize, we are using four different objective functions in NSGA-II adaptation. We used the four objective functions described in Section 6.3.2 :

1. ***Quality objective function*** that calculates the ratio of the number of corrected code-smells over the initial number of code-smells using detection rules [24].

2. ***Semantic objective function*** that corresponds to the weighted sum of different semantic measures described in Section 6.3.3. The semantic objective function of a refactoring solution corresponds to the average of the semantic values of the refactoring operations in the vector. In Table 6.3, we specify, for each refactoring operation, which measures are taken into account to ensure that the refactoring operation preserves design coherence.

Refactorings	VS	DS	IS	FIU	CD
move method	x	x			
move field	x	x			
pull up field	x	x		x	
pull up method	x	x	x		
push down field	x	x		x	
push down method	x	x		x	
inline class	x	x			
extract class	x	x			x
move class	x	x			
extract interface	x	x			x

Table 6.3 - Refactoring operations and their semantic measures.

3. ***Code changes objective function*** that approximates the amount of code changes needed to apply the suggested refactorings operations. We use the model described in Section 6.3.2.2.
4. ***History of changes objective function*** that maximizes the use of refactorings that are similar to those applied to the same code fragments in the past. To calculate the similarity score between a proposed refactoring operation and a recorded refactoring operation, we use the objective function described in Section 6.3.2.3.

d) Selection

To guide the selection process, NSGA-II uses a binary tournament selection based on dominance and crowding distance [24]. NSGA-II sorts the population using the dominance principle which classifies individual solutions into different dominance levels. Then, to construct a new offspring population Q_{t+1}, NSGA-II uses a comparison operator based on a calculation of the crowding distance [24] to select potential individuals having the same dominance level.

e) Genetic operators

In our NSGA-II design, we use the same genetic operators formulation adopted in our GA adaptation. The description of our genetic operators (crossover and mutation) is detailed in Section 4.3.2 d).

6.4 Evaluation

In order to evaluate the feasibility and the efficiency of our approach for generating good refactoring suggestions, we conducted an experiment based on different versions of open-source systems. We start by presenting our research questions. Then, we describe and discuss the obtained results. All experimentation materials are available online[21].

6.4.1 Research questions

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

- **RQ1.1:** To what extent can the proposed approach fix different types of code-smells?
- **RQ1.2:** To what extent does the proposed approach preserve design semantics when fixing code-smells?
- **RQ1.3:** To what extent can the proposed approach minimize code changes when fixing code-smells?
- **RQ1.4:** To what extent can the use of previously-applied refactorings improve the effectiveness of the proposed refactorings?

[21] http://www-etud.iro.umontreal.ca/~ouniali/TSE2014/

- **RQ2:** How does the proposed multi-objective approach based on NSGA-II perform compared to other existing search-based refactoring approaches and other search algorithms?
- **RQ3:** How does the proposed approach perform compared to existing approaches not based on heuristic search?
- **RQ4:** Is our multi-objective refactoring approach useful for software engineers in real-world setting?

To answer **RQ1.1**, we validate the proposed refactoring operations to fix code-smells by calculating the code-smell correction ratio (CCR) on a benchmark composed of six open-source systems. CCR corresponds to the ratio of the number of corrected code-smells over the initial number of detected code-smells before applying the suggested refactoring solution, and defined as follows:

$$CCR = \frac{\#\, corrected\ code\_smells}{\#\, code\_smells\ before\ applying\ refactorings} \in [0,1]$$

To answer **RQ1.2**, we use two different validation methods: manual validation and automatic validation to evaluate the efficiency of the proposed refactorings. For the manual validation, we asked six groups of potential users of our refactoring tool to evaluate manually whether the suggested refactorings are feasible and make sense semantically. We define the metric "refactoring precision" (RP), which corresponds to the number of meaningful refactoring operations (low-level and high-level), in terms of semantics, over the total number of suggested refactoring operations. RP is defined as follows:

$$RP = \frac{\#coherent\ refactorings}{\#proposed\ refactorings} \in [0,1]$$

For the automatic validation we compare the proposed refactorings with the expected ones using an existing benchmark [8] [7] [42] 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 [83], 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 6.5). Automatic recall and precision are defined, respectively, as follows:

$$RP_{recall} = \frac{|suggested\ refactorings| \cap |expected\ refactorings|}{|expected\ refactorings|} \in [0,1]$$

$$RP_{precision} = \frac{|suggested\ refactorings| \cap |expected\ refactorings|}{|suggested\ refactorings|} \in [0,1]$$

To answer **RQ1.3**, we evaluate, using our benchmark, if the proposed refactorings are useful to fix code-smells with low code changes by calculating the code change score. The code change score is calculated using our model described in Section 6.3.2.2. To this end, we compare the obtained code change scores with and without integrating the code change minimization objective in our tool.

To answer **RQ1.4**, we use the metric RP to evaluate the usefulness of the recorded refactorings and their impact on the quality of the suggested refactorings in terms of semantic coherence (RP). To this end, we compare the obtained code RP scores with and without integrating the reuse of recorded refactorings in our tool. In addition, in order to evaluate the importance of reusing recorded refactorings in similar contexts, we define the metric "reused refactoring" (RR) that calculates the percentage of operations from the base of recorded refactorings used to generate the optimal refactoring solution by our proposal. RR is defined as follows:

$$RR = \frac{\#\ used\ refactorings\ from\ the\ base\ of\ recorded\ refactorings}{\#\ refactorings\ in\ the\ base\ of\ recorded\ refactorings} \in [0,1]$$

To answer **RQ2**, we compare our approach to two other existing search-based refactoring approaches: our GA-based approach (described in Chapter 4), and Harman et al. [20] that consider the refactoring suggestion task only from the quality improvement perspective (single objective). We also assessed the performance of our multi-objective algorithm NSGA-II compared to another multi-objective algorithm MOGA (Multi-Objective Genetic Algorithm) [191], a random search (RS) [192], and a mono-objective genetic algorithm (GA) [99] where one fitness function is used (an average of the four objective scores).

To answer **RQ3**, we compared our refactoring results with a popular code-smells detection and correction tool JDeodorant [83] that does not use heuristic search techniques in terms of DCR, change score and RP. The current version of JDeodorant [83] is implemented as an Eclipse plug-in that identifies some types of code-smells using quality metrics and then proposes a list of refactoring strategies to fix them.

To answer **RQ4**, we asked a group of three software engineers to refactor manually some of the code-smells, and then compare the results with those proposed by our tool. To this end we define the following precision metric:

$$Precision = \frac{|R| \cap |R_m|}{R_m} \in [0,1]$$

where R is the set of refactorings suggested by our tool, and R_m is the set of refactorings suggested manually by software engineers.

6.4.2 Experimental setting and instrumentation

The goal of the study is to evaluate the usefulness and the effectiveness of our refactoring tool in practice. We conducted a non-subjective evaluation with potential users of our tool. Thus, refactoring operations should not only remove code-smells, but should also be meaningful from a developer's point of view.

6.4.2.1 Subjects

Our study involved a total number of 21 subjects divided into 7 groups (3 subjects each). All the subjects are volunteers and familiar with Java development. The experience of these subjects on Java programming ranged from 2 to 15 years.

The first six groups are drawn from several diverse affiliations: the University of Michigan (USA), University of Montreal (Canada), Missouri University of Science and Technology (USA), University of Sousse (Tunisia) and a software development and web design company. The groups include four undergraduate students, six master students, six PhD students, one faculty member, and four junior software developers. The three master students are working also at General Motors as senior software engineers. Subjects were familiar with the practice of refactoring.

6.4.2.2 Systems studied and data collection

We applied our approach to a set of six well-known and well-commented industrial open source Java projects: Xerces-J[22], JFreeChart[23], GanttProject[24], Apache Ant[25], JHotDraw[26], and Rhino[27]. Xerces-J is a family of software packages for parsing XML. JFreeChart is a powerful and flexible Java library for generating charts. GanttProject is a cross-platform tool for project scheduling. Apache Ant is a build tool and library specifically conceived for Java applications. JHotDraw is a GUI framework for drawing editors. Finally, Rhino is a JavaScript interpreter and compiler written in Java and developed for the Mozilla/Firefox browser.

We selected these systems for our validation because they are well studied in the related work. Moreover, they came from six different organisations, involved different kinds of software engineering development, and had different sizes ranging from 25 to 255 KLOC with a high number of code-smell instances. Table 6.4 provides some descriptive statistics about these six programs.

Systems	Release	# classes	# code-smells	KLOC
Xerces-J	v2.7.0	991	91	240
JFreeChart	v1.0.9	521	72	170
GanttProject	v1.10.2	245	49	41
Apache Ant	v1.8.2	1191	112	255
JHotDraw	v6.1	585	25	21
Rhino	v1.7R1	305	69	42

Table 6.4 - Systems statistics.

To collect refactorings applied in previous program versions, and the expected refactorings applied to next version of studied systems, we use Ref-Finder [83]. Ref-Finder,

[22] http://xerces.apache.org/xerces-j/
[23] http://www.jfree.org/jfreechart/
[24] www.ganttproject.biz
[25] http://ant.apache.org/
[26] http://www.jhotdraw.org/
[27] http://www.mozilla.org/rhino/

implemented as an Eclipse plug-in, can identify refactoring operations applied between two releases of a software system. Table 6.5 shows the analyzed versions and the number of refactoring operations, identified by Ref-Finder, between each subsequent couple of analyzed versions, after the manual validation. In our study, we consider only refactoring types described in Table 6.1.

Systems	**Expected refactorings**		**Collected refactorings**	
	Next release	**# Refactorings**	**Previous releases**	**# Refactorings**
Xerces-J	v2.8.1	39	v1.4.2 - v2.7.0	70
JFreeChart	v1.0.11	31	v1.0.6 - v1.0.9	76
GanttProject	v1.11.2	46	v1.7 - v1.10.2	91
Apache Ant	v1.8.4	78	v1.2 - v1.8.2	247
JHotDraw	v6.2	27	v5.1 - v6.1	64
Rhino	1.7R4	46	v1.4R3 - 1.7R1	124

Table 6.5 - Analysed versions and refactorings collection.

6.4.2.3 Scenarios

We designed our study to answer our research questions. To this end, we conducted our experiments through two different scenarios: 1) the first scenario is to evaluate the quality of the suggested refactoring solutions with potential users, and 2) the second scenario is to fix manually a set of code-smells and compare the manual results with those proposed by our tool. All the recommended refactorings are executed using the Eclipse platform.

Scenario 1: The first six groups of subjects were invited to fill a questionnaire that aims to evaluate our suggested refactorings. To this end, we assigned to each group a set of refactoring solutions suggested by our tool to evaluate manually. Table 6.6 describes the set of refactoring solutions to be evaluated for each studied system in order to answer our research questions. We have three multi-objective algorithms to be tested for the refactoring suggestion task: NSGA-II (Non-dominated Sorting Genetic Algorithm) [99], MOGA (Multi-Objective Genetic Algorithm) [191], and RS (Random Search) [192]. Moreover, we compared our results with a mono-objective genetic algorithm (GA) to assess the need for a multi-objective formulation. In addition, two refactoring solutions of both state-of-the-art approaches (GA-based approach [27], and Harman et al. [20]) are empirically evaluated in order to compare them to our approach in terms of semantic coherence.

Ref. Solution	**Algorithm/ Approach**	**# objective Functions**	**Objectives considered**
Solution 1	NSGA-II	4	Q, S, CC, RR
Solution 2	MOGA	4	Q, S, CC, RR
Solution 3	Random Search (RS)	4	Q, S, CC, RR
Solution 4	Genetic Algorithm	1	Q + S + CC + RR
Solution 5	GA-based approach [27]	1	Q
Solution 6	Harman et al. [20]	2	CBO, SDMPC

Table 6.6 - Refactoring solutions for each studied system.

As shown in Table 6.7, for each system, 6 refactoring solutions have to be evaluated. Due to the large number of refactoring operations to be evaluated (36 solutions in total, each solution consists of a large set of suggested refactoring operations), we pick at random a sample of 10 refactorings per solution to be evaluated in our study. In Table 6.7, we summarize how we divided subjects into groups in order to cover the evaluation of all refactoring solutions. In addition, as illustrated in Table 6.7, we are using a cross-validation for the first scenario to reduce the impact of subjects (groups A-F) on the evaluation. Each subject evaluates different refactoring solutions for three different systems.

Subjects (groups A-F) were aware that they are going to evaluate the semantic coherence of refactoring 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 to 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 6 solutions (10 refactorings per solution). The questionnaire is organized within a spreadsheet with hyperlinks to visualize easily the source code of the affected code elements. Subjects are invited to select for each refactoring operation one of the possibilities: *"Yes"* (coherent change), "*No"* (non-coherent change), or *"May be"* (if not sure). All the study material is available online[28]. Since the application of refactorings to fix

[28] http://www-etud.iro.umontreal.ca/~ouniali/TSE2014/

code-smells is a subjective process, it is normal that not all the programmers have the same opinion. In our case, we considered the majority of votes to determine if a suggested refactoring is accepted or not.

Scenarios	Subject groups	Systems	Algorithm / Approach	Solutions
Scenario 1	Group A	GanttProject	NSGA-II Genetic Algorithm	Solution 1 Solution 4
		Xerces	MOGA, Harman et al.	Solution 2 Solution 6
		JFreeChart	RS, GA-based approach	Solution 3 Solution 5
	Group B	GanttProject	MOGA, Harman et al.	Solution 2 Solution 6
		Xerces	RS, GA-based approach	Solution 3 Solution 5
		JFreeChart	NSGA-II Genetic Algorithm	Solution 1 Solution 4
	Group C	GanttProject	RS, GA-based approach	Solution 3 Solution 5
		Xerces	NSGA-II Genetic Algorithm	Solution 1 Solution 4
		JFreeChart	MOGA, Harman et al.	Solution 2 Solution 6
	Group D	Apache Ant	NSGA-II Genetic Algorithm	Solution 1 Solution 4
		JHotDraw	MOGA, Harman et al.	Solution 2 Solution 6
		Rhino	RS, GA-based approach	Solution 3 Solution 5
	Group E	Apache Ant	MOGA, Harman et al.	Solution 2 Solution 6
		JHotDraw	RS, GA-based approach	Solution 3 Solution 5
		Rhino	NSGA-II Genetic Algorithm	Solution 1 Solution 4
	Group F	Apache Ant	RS, GA-based approach.	Solution 3 Solution 5
		JHotDraw	NSGA-II Genetic Algorithm	Solution 1 Solution 5
		Rhino	MOGA, Harman et al.	Solution 2 Solution 6
Scenario 2	Group G	All systems	Manual correction of code-smells	N.A.

Table 6.7 - Survey organization.

Scenario 2: The aim of this scenario is to compare our refactoring results for fixing code-smells suggested by our tool with manual refactorings suggested by software engineers. To this end, we asked Group G to fix a set of 72 code-smell instances that are picked at random from each subject system (12 code-smells per system) that comes from the six different code-smell types considered. Then we compared their sequences of refactorings that are suggested manually with those proposed by our approach. The more our refactorings are similar to the manual ones, the more our tool is assessed to be useful and efficient in practice.

6.4.2.4 Algorithms configuration

In our experiments, we use and compare different mono and multi-objective algorithms. For each algorithm, to generate an initial population, we start by defining the maximum vector length (maximum number of operations per solution). The vector length is proportional to the number of refactorings that are considered, the size of the program to be refactored, and the number of detected code-smells. 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 employed refactoring list. During the creation, the solutions have random sizes inside the allowed range. For all algorithms NSGA-II, MOGA, Random search (RS), and genetic algorithm (GA), we fixed the maximum vector length to 700 refactorings, and the population size to 200 individuals (refactoring solutions), and the maximum number of iterations to 6000 iterations. We also designed our NSGA-II adaptation to be flexible in a way that we can configure the number objectives and which objectives to consider in the execution.

We consider a list of 11 possible refactorings to restructure the design of the original program by moving code elements (methods, attributes) from classes in the same or different packages or inheritance hierarchies or splitting/merging classes/interfaces. Although we believe that our list of refactorings is sufficient at least to fix these specific types of code smells, our refactoring tool is developed in a flexible way so that new refactorings and code smell types can be considered in the future. Moreover, our list of possible refactoring is significantly larger than those of existing code-smells correction techniques.

Another element that should be considered when comparing the results of the four algorithms is that NSGA-II does not produce a single solution like GA, but a set of optimal solutions (non-dominated solutions). The maintainer can choose a solution from them depending on their preferences in terms of compromise. However, at least for our evaluation, we need to select only one solution. 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. In our case, the ideal solution has the best value of quality (equal to 1), of semantic coherence (equal to 1), and of refactoring reuse (equal to 1), and code changes (normalized value equal to 1). Hence, we select the nearest solution to the ideal one in terms of Euclidian distance.

6.4.2.5 Inferential statistical test method used

Our approach, like the two others (GA-based approach and Harman et al.), is stochastic by nature, i.e., two different executions of the same algorithm with the same parameters on the same systems generally leads to different sets of suggested refactorings. 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 Wilcoxon signed-rank test is a non-parametric statistical hypothesis test used when comparing two related samples to verify whether their population mean-ranks differ or not. In this way, we could decide whether the difference in performance between our approach and the other detection algorithms is statistically significant or just a random result.

6.4.3 Empirical study results

This section reports the results of our empirical study, which are further discussed in next section. We first start by answering our research questions. To this end, we use two different validations: manual and automatic validations.

Results for RQ1.1: As described in Table 6.8, after applying the proposed refactoring operations by our approach (NSGA-II), we found that, on average, 84% of the detected code-smells were fixed (CCR) for all the six studied systems. This high score is considered significant in terms of improving the quality of the refactored systems by fixing the majority of code-smells coming from various types (Blob, spaghetti code, functional decomposition, data class, shotgun surgery, and feature envy).

Systems	**Approach**	**CCR**	**Changes score**	**RP**	**RP-automatic**
Xerces	NSGA-II	83% (76|91)	3843	81 %	26% (10|39)
	Harman et al. '07	N.A	2669	41 %	8 % (3|39)
	GA-based approach	89% (81/91)	4998	37 %	13% (5|39)
JFreeChart	NSGA-II	86% (62|72)	2016	82 %	35% (11|31)
	Harman et al. '07	N.A	3269	36 %	0 % (0|31)
	GA-based approach	90% (65\72)	3389	37 %	13% (4|31)
GanttProject	NSGA-II	85% (42|49)	2826	80 %	46%(21|46)
	Harman et al. '07	N.A	4790	23 %	0% (0|46)
	GA-based approach	95% (47|49)	4697	27 %	15% (7|46)
Apache Ant	NSGA-II	78% (87|112)	4690	78 %	31% (24|78)
	Harman et al. '07	N.A	6987	40 %	04% (3|78)
	GA-based approach	80% (90|112)	6797	30 %	0% (0|78)
JHotDraw	NSGA-II	84% (21|25)	2231	80 %	44% (18|41)
	Harman et al. '07	N.A	3654	37 %	10% (4|41)
	GA-based approach	84% (21|25)	3875	43 %	7% (3|41)
Rhino	NSGA-II	85% (59|69)	1914	80 %	33% (15|46)
	Harman et al. '07	N.A	2698	37 %	0% (0|46)
	GA-based approach	87% (60|69)	3365	32 %	9% (4|46)
Average (*all systems*)	**NSGA-II**	**84%**	**2937**	**80 %**	**36%**
	Harman et al. '07	**N.A**	**4011**	**36 %**	**4%**
	GA-based approach	**89%**	**4520**	**34 %**	**9%**

Table 6.8 - Empirical study results on 31 runs. The results were statistically significant on 31 independent runs using the Wilcoxon rank sum test with a 95% confidence level ($\alpha < 5\%$).

Results for RQ1.2: To answer RQ1.2, we need to assess the correctness/meaningfulness of the suggested refactorings from the developers' point of view. We reported the results of our empirical evaluation in Table 6.8 (RP column) related to Scenario 1. We found that the majority of the suggested refactorings improve significantly the code quality while preserving semantic coherence. On average, for all of our six studied systems, 80% of proposed refactoring operations are considered by potential users to be semantically feasible and do not generate semantic incoherence.

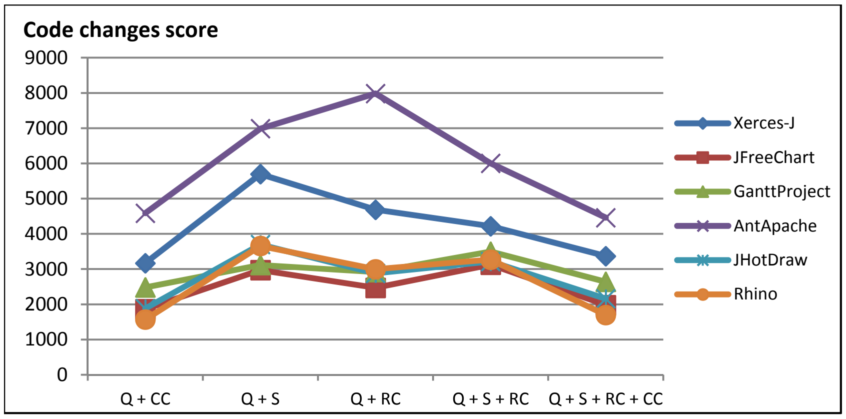
In addition to the empirical evaluation, we automatically evaluate our approach without using the feedback of potential users to give more quantitative evaluation to answer RQ3. Thus, we compare the proposed refactorings with the expected ones. The expected refactorings are those applied by the software development team to the next software release as described in Table 6.5. We use Ref-Finder [46] to identify refactoring operations that are applied between the program version under analysis and the next version. Table 6.8 (RP automatic column) summarizes our results. We found that a considerable number of proposed refactorings (an average of 36% 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 code-smells 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.

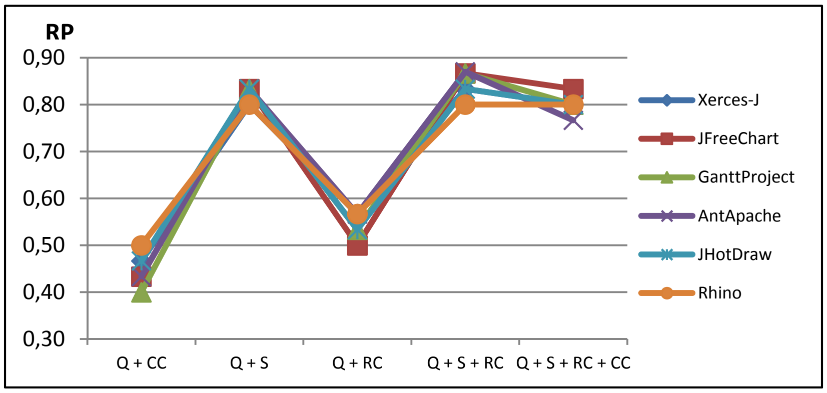
Results for RQ1.3 and RQ1.4: To answer these two research questions, we need to compare different objective combinations (two, three, or four objectives) to ensure the efficiency and the impact of using each of the objectives we defined. To this end, we executed the NSGA-II algorithm with different combinations of objectives: maximize quality (*Q*), minimize semantic incoherence (*S*), minimize code changes (*CC*), and maximize the reuse of recorded refactorings (*RR*) as presented in Table 6.9 and Figure 6.4.

To answer RQ1.3, we present in Figure 6.4.a and Table 6.9, the code change scores obtained when the CC objective is considered (Q+S+RC+CC). We found that our approach succeeded in suggesting refactoring solutions that do not require high code changes (an average of only 2937) while having more than 3888 as code change score when the CC objective is not considered in the other combinations. At the same time we found that the CCR score (Figure 6.4.c) is not significantly affected with and without considering the CC objective.

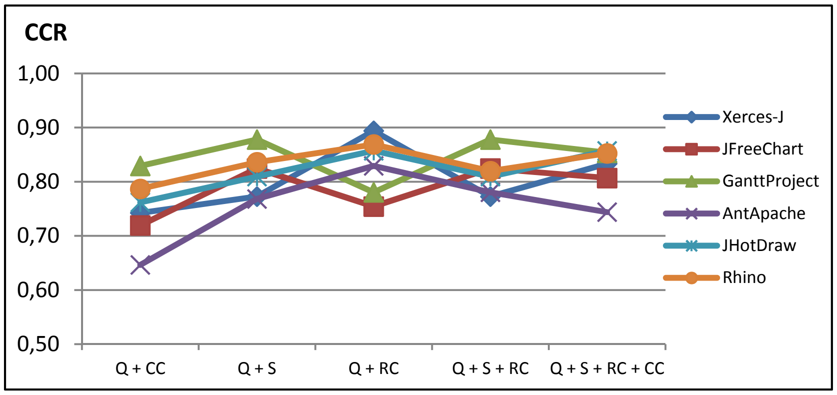
To answer RQ1.4, we present the obtained results in Figure 6.4.b. The best RP scores are obtained when the recorded code changes (RC) are considered (Q+S+RC), while having good correction score CCR (Figure 6.4.c). In addition, we need more quantitative evaluation to investigate effect of the use of recorded refactorings, on the semantic coherence (RP). To this end, we compare the RP score with and without using recorded refactorings. In most of the systems when recorded refactoring is combined with semantics, the RP value is improved. For example, for Apache Ant RP is 83% when only quality and semantics are considered, however when recorded refactoring reuse is included the RP is improved to 87% (Figure 6.4.b). We notice also that when code changes reduction is included with quality, semantics and recorded changes, the RP and CCR scores are not significantly affected. Moreover we notice in Figure 6.4.c that there is no significant variation in terms of CCR with all different objectives combinations. When four objectives are combined the CCR value induces a slight degradation with an average of 82% in all systems which is even considered as promising results. Thus, the slight loss in the correction ratio is largely compensated by the significant improvement of the semantic coherence and code changes reduction. Moreover, we found that the optimal refactoring solutions found by our approach are obtained with a considerable percentage of reused refactoring history (RR) (more than 35% as shown in Table 6.9). Thus, the obtained results support the claim that recorded refactorings applied in the past are useful to generate coherent and meaningful refactoring solutions and can effectively drive the refactoring suggestion task.



(a)



(b)



(c)

Figure 6.4 - Refactoring results of different objectives combination with NSGA-II in terms of (a) code changes reduction, (b) semantics preservation, (c) Code-smells correction ratio.

In conclusion, we found that the best compromise is obtained between the four objectives using NSGA-II comparing to the use of only two or three objectives. By default, the tool considers the four objectives to find refactoring solutions. Thus, a software engineer can consider the multi-objective algorithm as a black-box and he do not need to configure anything related to the objectives to consider. The four objectives should be considered and there is no need to select the objectives by the user based on our experimentation results.

Objectives combinations	**DCR**	**RP (empirical evaluation)**	**Code changes**	**RR**
Q + CC	75%	45%	2591	N.A.
Q + S	81%	82%	4355	N.A.
Q + RC	85%	54%	3989	41%
Q + S + RC	81%	84%	3888	35%
Q + S + RC + CC	84%	80%	2917	36%

Table 6.9 - Average refactoring results of different objective combinations with NSGA-II (average of all systems) on 31 runs. The results were statistically significant on 31 independent runs using the Wilcoxon rank sum test with a 95% confidence level ($\alpha < 5\%$).

Results for RQ2: To answer RQ2, we evaluate the efficiency of our approach comparing to two existing approaches: Harman et al. [20] and GA-based approach. Harman et al. proposed a multi-objective approach that uses two quality metrics to improve (coupling between objects CBO, and standard deviation of methods per class SDMPC) after applying the refactorings sequence. GA-based approach, a single-objective genetic algorithm is used to correct code-smells (see Chapter 4 for more details). The comparison is performed through three levels: 1) code-smell correction ratio (CCR) that is calculated using code-smells detection rules (see Chapter 3) [7], 2) refactoring precision (RP) that represents the results of the subject judgments (Scenario 1), and 3) code changes needed to apply the suggested refactorings. We adapted our technique for calculating code changes scores for both approaches Harman et al. and GA-based approach. Table 6.8 summarizes our findings and reports the median values of each of our evaluation metrics obtained for 31 simulation runs of all projects.

As described in Table 6.8, after applying the proposed refactoring operations, we found that more than 84% of detected code-smells were fixed (CCR) as an average for all the six studied systems. This score is comparable to the correction score of GA-based approach (89%), an approach that does not consider semantic preservation, nor code changes reduction nor recorded refactorings reuse (CCR is not considered in Harman et al. since their aim is to improve only some quality metrics).

We also found that our approach succeeded fixing code-smells with lower code change scores (an average of only 2917) comparing to other approaches having respectively an average of 4011 and 4520 for all studied systems. Consequently, our approach succeeded in reducing significantly the number of code changes to preserve the initial design while having good correction scores (84%).

Regarding the semantic coherence, for all of our six studied systems, an average of 80% of proposed refactoring operations are considered as semantically feasible and do not generate semantic incoherence. This score is significantly higher than the scores of the two other approaches having respectively only 36% and 34% as RP scores. Thus, our approach performs clearly better for RP and code changes score with the cost of a slight degradation in CCR compared to GA-based approach. This slight loss in the CCR is largely compensated by the significant improvement in terms of semantic coherence and code change reduction.

We compared the three approaches in terms of automatic RP_{recall}. We found that a considerable number of proposed refactorings, an average of 36% for all studied systems in terms of recall, are already applied to the next version by the software development team. By comparison, the figures for Harman et al. and GA-based approach are only 4% and 9% respectively (see Figure 6.5). Moreover, this score shows that our approach is useful in practice unlike both other approaches. In fact, the RP_{recall} of Harman et al. is not significant, due to the fact that only the move method refactoring is considered when searching for refactoring solutions to improve coupling and standard deviation of methods per class. Moreover, expected refactorings are not related only to quality improvement, but also for adding new functionalities, and other maintenance tasks. This is not considered in our approach when we search for the optimal refactoring solution that satisfies our four objectives. However, we manually inspected expected refactorings and we found that they are mainly related to adding new functionality (related to adding new packages, classes or methods).

In conclusion, our approach produces good refactoring suggestions in terms of code-smell correction ratio, semantic coherence, and code changes reduction from the point of view of 1) potential users of our refactoring tool and 2) expected refactorings applied to the next program version.

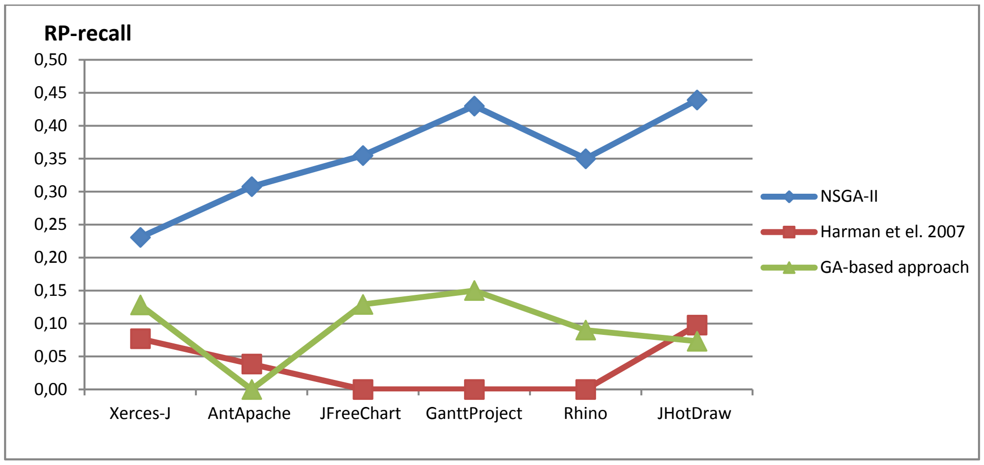
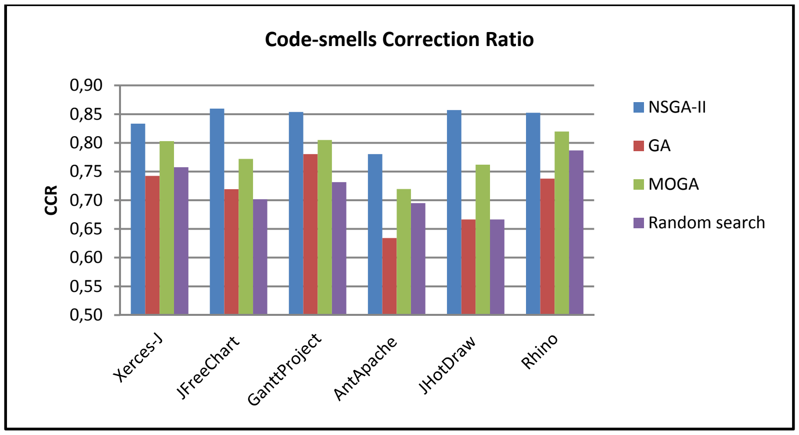
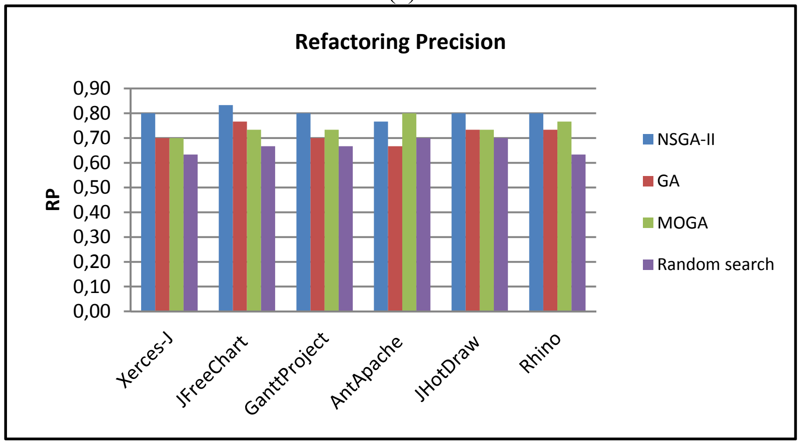


Figure 6.5 - Automatic refactoring precision comparison.

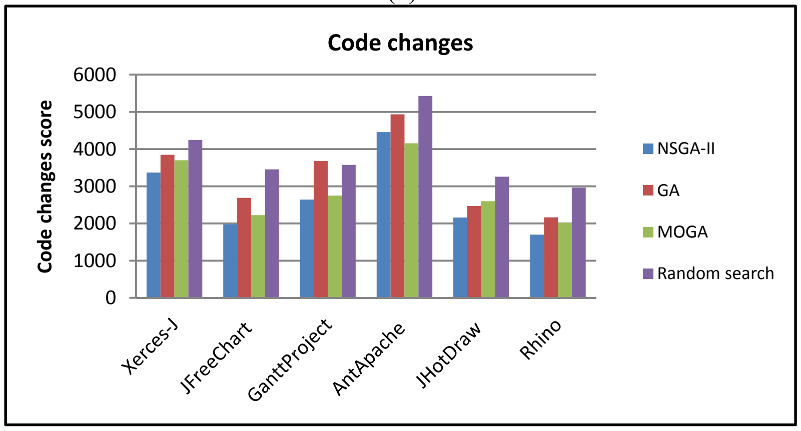
To this end, we compared the performance of our proposal to two other multi-objective algorithms: MOGA, and a random search and a mono-objective algorithm (genetic algorithm). In a random search, the change operators (crossover and mutations) are not used, and populations are generated randomly and evaluated using the four objective functions. In our mono-objective adaptation, we considered a single fitness function, which is the normalized average score of the four objectives using a genetic algorithm. Moreover, since in our NSGA-II adaptation we select a single solution without giving more importance to some objectives, we give equal weights for each fitness function value.



(a)



(b)



(c)

Figure 6.6 - Refactoring results of different algorithms in terms of (a) semantics preservation, (b) code-smells correction ratio, (c) code changes reduction.

As shown in Figure 6.6, NSGA-II outperforms significantly MOGA, random-search, and mono-objective algorithm in terms of code-smells correction ratio (CCR), semantics preservation (RP), and code changes reduction. For instance, in JFreeChart, NSGA-II performs much better than MOGA, random search and genetic algorithm in terms of DCR and RP scores (respectively Figure 6.6.a and Figure 6.6.b). In addition, NSGA-II reduces significantly code changes for all studied systems, approximately by half for Rhino (Figure 6.6.c).

Results for RQ3: JDeodorant uses only structural information to detect and fix code-smells, but does not handle all the six code-smell types that we considered in our experiments. Thus, to make the comparison fair, we performed our comparison using only two code-smells that can be fixed by both tools: Blob and feature envy. Figure 6.7 summarizes our findings. It is clear that our proposal outperforms JDeodorant, on average, on all the systems in terms of number of fixed code-smells with a minimum number of changes and semantically coherent refactorings. The average number of fixed code-smells is comparable between both tools however our proposal is clearly better in terms of semantically coherent refactorings. This can be explained by the fact that JDeodorant uses only structural metrics to evaluate the impact of suggested refactorings on the detected code-smells. In addition, our proposal supports more types of refactorings than JDeodorant and this is also explains our outperformance.

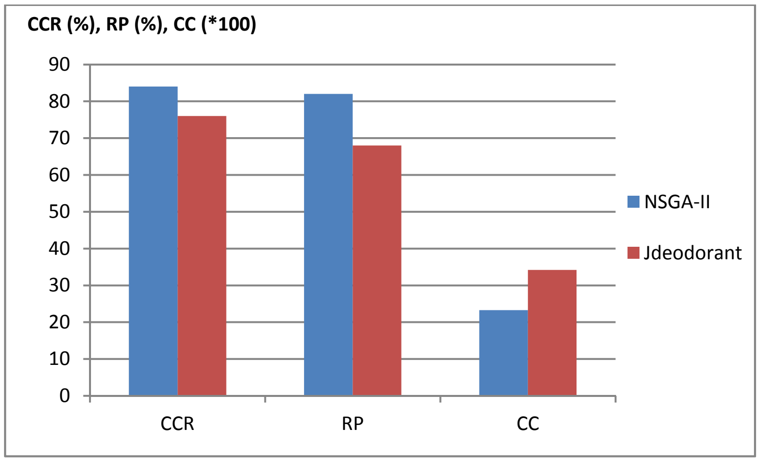


Figure 6.7 - Comparison results of our approach with JDeodorant: average code-smells correction ratio (CCR), semantic coherence (RP) and code changes score (CC) on all the systems.

Results for RQ4: To evaluate the relevance of our suggested refactorings for real software engineers, we compared the refactoring strategies proposed by our technique and those proposed manually by group G (three software engineers) to fix several code-smells on the six systems. Figure 6.8 shows that most of the suggested refactorings by NSGA-II are similar to those applied by developers with an average of more than 75%. Some code-smells can be fixed by different refactoring strategies and also the same solution can be expressed in different ways (complex and atomic refactorings). Thus we consider that the average precision of more than 75% confirms the efficiency of our tool for real developers to automate the refactoring recommendation process. We discuss, in the next section, in more detail the relevance of our automated refactoring approach for software engineers.

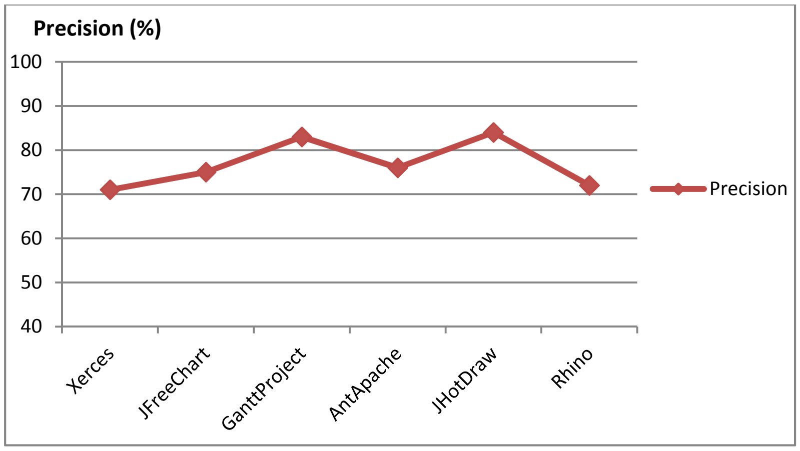


Figure 6.8 - Comparison of our refactoring results with manual refactorings in terms of precision.

6.5 Discussions

We now provide more quantitative and qualitative analyses of our results and discuss some observations drawn from our empirical evaluation of our refactoring approach.

6.5.1 Refactoring impact

We observed that our technique performs better than two existing approaches. We also compared different objective combinations and found that the best compromise is obtained between the four objectives using NSGA-II when compared to the use of only two or three objectives. Therefore, our four objectives are efficient for providing "good" refactoring suggestions. Moreover, we found that NSGA-II performs much better than two other multi-objective algorithms: MOGA and random search, and a mono-objective algorithm (GA).

Thus, although our primary goal in this work is to demonstrate that code-smells can be automatically refactored, it is also important to assess the refactoring impact on design quality. The expected benefit from refactoring is to enhance the overall software design quality, as well as fixing code-smells [62]. We use the QMOOD (Quality Model for Object-Oriented Design) model [193] to estimate the effect of the suggested refactoring solutions on quality attributes. We choose QMOOD, mainly because 1) it is widely used in the literature [22] [126] to assess the effect of refactoring, and 2) it has the advantage that define six high level design quality attributes (reusability, flexibility, understandability, functionality, extendibility and effectiveness) that can be calculated using 11 lower level design metrics [193]. In our study we consider the following quality attributes: reusability, flexibility, understandability, effectiveness. These quality attributes and metrics are defined in Appendix A.

We did not assess the issue of functionality because we assume that, by definition, refactoring does not change the behavior/functionality of systems; instead it changes the internal structure. We have also excluded 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 inadequate.

The improvement in quality can be assessed by comparing the quality before and after refactoring independently to the number of fixed code-smells. 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:

$$G_{q_i} = q_i' - q_i$$

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

In Figure 6.9, we show the obtained gain values (in terms of absolute value) that we calculated for each QMOOD quality attribute before and after refactoring for each studied system. We found that systems' quality increase across the four QMOOD quality factors. Understandability is the quality factor that has the highest gain value; whereas the Effectiveness quality factor has the lowest one. This mainly due to many reasons 1) the majority of fixed code-smells (Blob, spaghetti code) are known to increase the coupling (DCC) within classes which heavily affect the quality index calculation of the Effectiveness factor; 2) the vast majority of suggested refactoring types were move method, move field, and extract class (Figure 6.11) that are known to have a high impact on coupling (DCC), cohesion (CAM) and the design size in classes (DSC) that serves to calculate the understandability quality factor. Furthermore, we noticed that JHotDraw produced the lowest quality increase for the four quality factors. This is justified by the fact that JHotDraw is known to be of good design and implementation practices and contains fiew code-smell instances comparing to the five other studied systems.

To sum up, we can conclude that our approach succeeded in improving the code quality not only by fixing the majority of detected code-smells but also by improving the user understandability, reusability, flexibility, as well as the effectiveness of the refactored program.

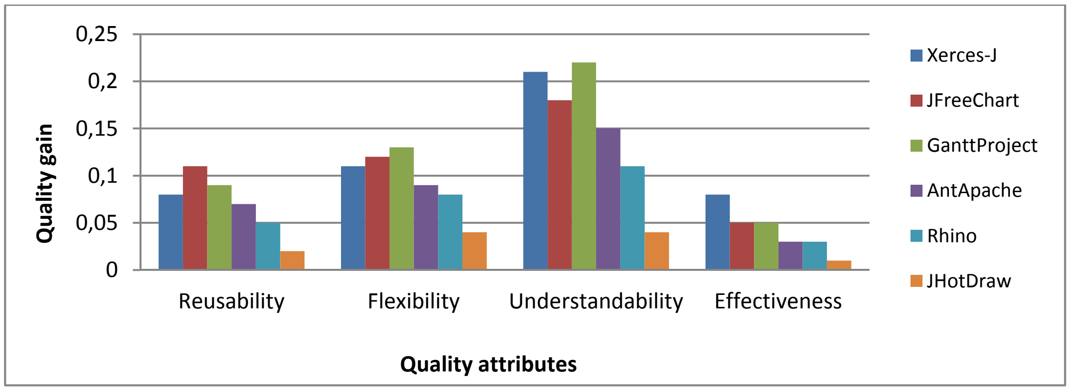


Figure 6.9 - Impact of the suggested refactoring solution on QMOOD quality attributes.

6.5.2 Other observations

It is important to contrast the results of multiple executions with the execution time to evaluate the performance and the stability of our approach. The execution time for finding the optimal refactoring solution with a number of iterations (stopping criteria) fixed to 6000 was less than forty-eight minutes as shown in Figure 6.10. Moreover, we evaluate the impact of the number of suggested refactorings on the CCR, RP, RR, and code changes scores in five different executions. Drawn for JFreeChart, the results of Figure 6.10 show that the number of suggested refactorings does not affect the refactoring results. Thus, a higher number of operations in a solution do not necessarily mean that the results will be better. Thus, we could conclude that our approach is scalable from the performance standpoint, especially that our technique is executed, in general, up front (at night) to find suitable refactorings. In addition, the results accuracy is not affected by the number of suggested refactorings.

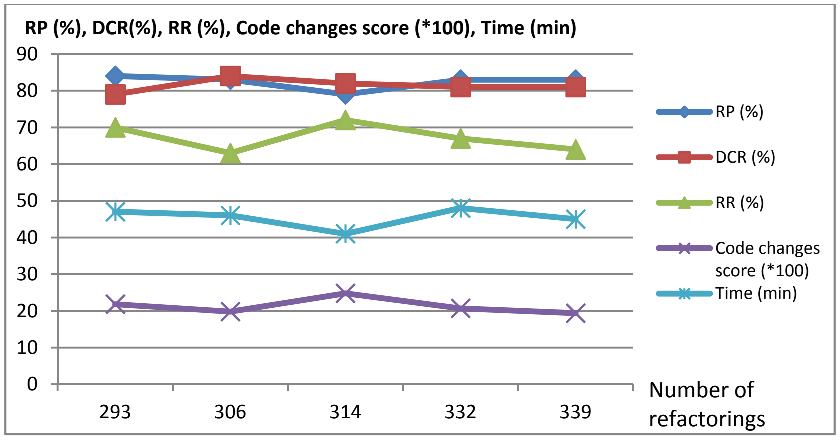


Figure 6.10 - Impact of the number of refactorings on multiple executions on JFreeChart.

Another important consideration is the refactoring operations distribution. We contrast that the most suggested refactorings are move method, move field, and extract class for the majority of studied systems except JHotDraw. For instance, in Xerces-J, we had different distribution of different refactoring types as illustrated in Figure 6.11. We notice that the most suggested refactorings are related to moving code elements (fields, methods) and extract/inline class. This is mainly due to the type of code-smells detected in Xerces-J (most of code-smells are related to the Blob code-smell) that need particular refactorings to move elements from Blob class to other classes in order to reduce the number of functionalities from them. On the other hand, we found for JHotDraw less move method, move field and extract class refactorings. This mainly due to the fact that JHotDraw contains few number of Blob instances (only three Blobs are detected) and it is known to be of good quality. Thus, our results in Figure 6.11 reveal an effect that we found: refactorings like move field, move method, and extract class are likely to be more useful to correcting the Blob code-smell. As part of future work we plan to investigate the relationship between code-smells types and refactoring types.

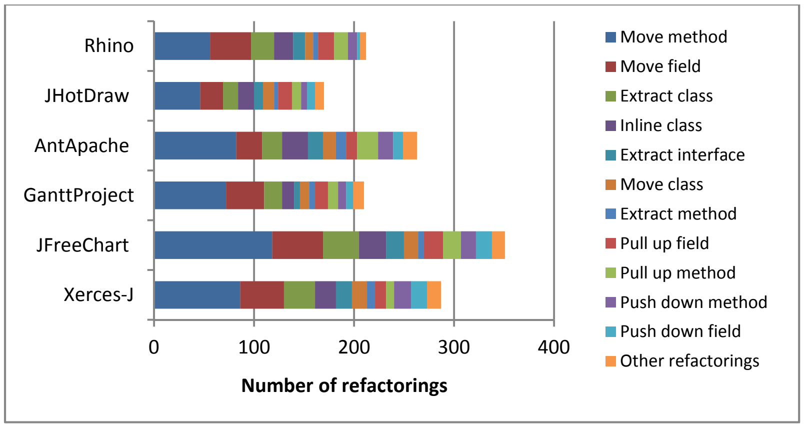


Figure 6.11 - Suggested refactorings distribution.

To illustrate some of these refactorings, let us consider the solution fragment sketched in Figure 6.12. This solution recommends to apply extract class refactoring twice to the class GanttOptions which is detected as a Blob using our code-smells detection rules as it contains 31 attributes and 69 methods. The first extract class creates a new empty class and moves the following set of attributes {x, y, width, weight, myRoleManager} and methods {setWindowPosition, setWindowSize, emptyElement, startElement, endElement, addAttribute, loadRoleSets, saveRoleSets, saveRoles} that are strongly cohesive with low coupling with the original class GanttOptions. A second refactoring is recommended to fix this Blob code-smell to split it again by applying and extract class where another cohesive set of attributes and methods are moved. Both extracted classes are of high cohesion and low coupling with the original class. In addition, based on semantic similarity, move field refactorings are recommended to move the two fields iconSize and openTips to the classes GanttLookAndFeelInfo and GanttProject respectively with a vocabulary-based similarity of 0.38. By applying these refactoring operations, the Blob GanttOptions is successfully fixed without affecting other parts of the system or producing semantic incoherencies.

```
…
--------------------------------------------------------------------
Extract Class:
            source class: net.sourceforge.ganttproject.GanttOptions
            fields to move: x y width weight myRoleManager
            methods to move: setWindowPosition setWindowSize emptyElement startElement endElement addAttribute loadRoleSets saveRoleSets
saveRoles
--------------------------------------------------------------------
Extract Class:
            source class: net.sourceforge.ganttproject.GanttOptions
            fields to move: isLoaded workingDir redline myChartMainFont myMenuFont
            methods to move: save load getUIConfiguration setButtonShow intiByDefault setDefaultTaskColor setResourceColor
setResourceOverloadColor getDefaultColor getResourceColor getResourceOverloadColor
--------------------------------------------------------------------
moveField:
             sourceClass : net.sourceforge.ganttproject.GanttOptions
             targetClass : net.sourceforge.ganttproject.gui.GanttLookAndFeelInfo
             field : iconSize
--------------------------------------------------------------------
moveField:
             sourceClass : net.sourceforge.ganttproject.GanttOptions
             targetClass : net.sourceforge.ganttproject.GanttProject
             field : openTips
--------------------------------------------------------------------
moveMethod:
             sourceClass : net.sourceforge.ganttproject.GanttGraphicArea$Arrow
             targetClass : net.sourceforge.ganttproject.parser.DependencyTagHandler$GanttDependStructure
             method : setDraw
. . .
```

Figure 6.12 - Refactoring solution fragrment executed to GanttProject.

6.6 Threats to validity

Some threats limit the validity of our experimental results.

Construct validity concern the relation between the theory and the observation. In our experiments, code-smell detection rules [7] we use to measure CCR could be questionable since there is no consensus on detecting code-smells. To mitigate this threat, we manually inspect and validate each code-smell. Moreover, our refactoring tool configuration is flexible and can support other state-of-the-art detection rules. Another threat concerns the modification score needed to apply refactoring. In fact, attributing weights manually might not be enough. To mitigate this issue, we are planning to consider other software metrics to calculate change score such as coupling and complexity. Moreover, another threat concerns the data about the actual refactorings of the studied systems. In addition to the documented refactorings, we are using Ref-Finder, which is known to be efficient [46]. Indeed, Ref-Finder was able to detect refactoring operations with an average recall of 95% and an average precision of 79% [46]. To ensure the precision, we manually inspect the refactorings found by Ref-Finder.

Internal validity: we identify three threats to internal validity: selection, learning and fatigue, and diffusion.

For the selection threat, the subject diversity in terms of profile and experience could affect our study. First, all subjects were volunteers. We also mitigated the selection threat by giving written guidelines and examples of refactorings already evaluated with arguments and justification. Additionally, each group of subjects evaluated different refactorings from different systems for different techniques/algorithms. We also took care to randomize the selection refactorings to be evaluated for each refactoring solution.

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 this task. Finally, only ten refactorings per system were randomly picked for the evaluation.

Diffusion threat is limited in our study because most of the subjects are geographically located in three different universities and a company, and the majority do not know each other. For the few ones who are in the same location, they were instructed not to share information about the experience prior to the completion of the study.

Conclusion validity deals with the relation between the treatment and the outcome. Thus, 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 refactorings 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 six different widely-used systems belonging to different domains and with different sizes, as described in Table 6.4. However, we cannot assert that our results can be generalized to industrial Java applications, other programming languages, and to other practitioners. Future replications of this study are necessary to confirm our findings. Another limitation of our results is the selection of the best solution from the Pareto front. We used a technique to select the closest solution to the ideal point in terms of Euclidian distance. We plan in our future work to integrate developers in the loop to select the best solution from the set of non-dominated solutions. In fact, developers can select, sometimes, refactoring solutions that change the behavior or violate the semantics and maximize quality improvements because they consider that fixing these semantic issues is relatively easy for them if they are familiar with the software design.

Finally, our approach takes as input a base of recorded/collected code changes from previous versions. We believe that this data is not always available, especially in the beginning of projects. However, we believe that refactorings recorded/collected for other systems can be used in similar contexts. As a part of our future work, we plan to consider the similarity with not only the refactoring type but also with the contexts (code fragments).

6.7 Conclusion

We have introduced in this chapter new multi-objective search-based approach taking into consideration multiple criteria to find refactoring solutions. The suggested refactorings aims at 1) fixing the detected code-smells, 2) preserving the design semantics of the refactored program, 3) reducing the amount of code change/adaptation, and 4) reusing knowledge from recorded refactorings applied in the past to similar contexts. Our search-based approach succeeded to find a near-optimal trade-off between these multiple conflicting criteria. Thus, our proposal produces more meaningful refactorings with lower code change scores. Moreover, the proposed approach was empirically evaluated on six meduim and large size open-source systems, and compared successfully to two existing approaches and three different search-based algorithms. Furthermore, our approach succeeds in suggesting a significant number of expected refactorings that were applied to the next release of the software system being studied, unlike other approaches, which provides evidence that our approach is efficient and useful in practice.

As future reseach directions we intend to adapt our multi-objective approach to fix other types of code-smells that can occur in new emergent service-based applications such as multi-service and tiny-service. We plan also to conduct an empirical study to understand the correlation between the number of applied refactrings and the number of code-smells, the correlation between code-smells and QMOOD quality attribues, and the relationship between fixing particular code-smells and introducing/fixing other code-smells implicitely in other parts of the system. Furthermore, while it is important to fix “bad” design practices (i.e., code-smells), it is also important to introduce “good” design practices (i.e., design patterns). In the next chapter, we will introduce our approach that aims at fixing code-smells while introducing design patterns.

Chapter 7: A Multi-objective refactoring recommendation approach to introduce design patterns and fix anti-patterns

7.1 Introduction

Refactoring is an efficient technique to improve the quality attributes of software systems such as maintainability, readability, and extendibility. To improve these quality attributes, most existing studies focus on the correction of Code-smells. However, this may not be sufficient to make the source code easier to understand and modify. The introduction of design patterns that represent good design practices while fixing code-smells can significantly improve the quality of systems. In this chapter, we introduce an automated multi-objective refactoring recommending approach to (1) improve design quality (as defined by software quality metrics), (2) introduce design patterns, and (3) fix code-smells. To the best of our knowledge, this is the first attempt to promote design patterns for fixing code-smells. To evaluate our approach, we conducted a quantitative and qualitative evaluation with software engineers using a benchmark composed of four open source systems. The statistical analysis of the results provides evidence that our approach is efficient compared to the state-of-the-art of refactoring techniques.

This chapter is organized as follows. Section 7.2 explains the motivations behind this approach. Section 7.3 describes our apprach. Section 7.4 presents our experimental study to evaluate the proposed approach, while Section 7.5 presents and discusses the obtained results. Section 6.7 concludes and presents plans for future work.

7.2 Motivations

To improve the quality of software systems, one of the widely used techniques is refactoring. It can help software developers to reduce the time required for adding new requirements, correcting bugs, understanding the existing implementation, modifying the code to improve its quality, and so on. Consequently, various refactoring tools have been proposed [22] [49] [126] [132].

Despite its significant benefits, recent studies show that automated refactoring tools are underused most of the time [85] [131] [194]. One of the possible reasons is that most existing refactoring tools [22] [45] [147] [148] focus mainly only on improving some quality metrics (e.g., coupling, cohesion, complexity, etc.). For instance, improving software quality factors does not mean that code-smells or bad-design practices that may exist are fixed. Thus, quality metric values can be significantly improved but the original program may still contain a considerable number of code-smells, which may lead, in turn, to maintenance and evolution difficulties. On the other hand, *design patterns* are "good" solutions to recurring design problems, conceived to increase reuse, code quality, code readability and, above all, maintainability and resilience to change [195]. Design patterns can be automatically introduced using refactoring [126] [127], however, most existing refactoring tools do not consider the use of design patterns to fix code-smells and improve the quality of software systems. In addition, applying a design pattern where it is not needed is highly undesirable as it introduces an unnecessary complexity to the system for no benefit [127].

Furthermore, some code-smells can be automatically fixed when applying design patterns. For instance, a Blob code-smell can be fixed by introducing a Visitor design pattern. Additional functionalities and behaviour can be easily added to the Blob class through the visitor pattern. That is, Visitor pattern relates in general to complex hierarchies that have a large number of inherited methods or with Blob classes that can be detected [124].

To address the above-mentioned challenges, we introduce a novel approach to guide the introduction of design patterns by fixing code-smells and improving the overall quality of the system while avoiding semantic incoherencies to the design. As far as we know, this is the first work that suggests refactoring strategies that deals with both design patterns and anti-patterns to improve software quality. To this end, we have developed a multi-objective optimization approach supported by a tool called MORE (Multi-Objective REfactoring) to find the best compromise between 1) improving software quality, 2) fixing anti-patterns and 3) introducing design patterns while satisfying a set of constraints to ensure the semantic coherence of the refactored program. More specifically, the primary contributions of this chapter are as follows:

1. We introduce a multi-objective search-based refactoring approach to improve software quality attributes (i.e., flexibility, maintainability, etc.), introduce "good" design practices (i.e., design patterns) and fix "bad" design practices (i.e., code-smells). We implemented our approach in a tool called MORE. We present a set of constraints, for each refactoring operation, in order to ensure the semantic coherence of the refactored program, e.g., that a method is not moved to a class where it makes no sense.
2. We present an empirical study based on a quantitative and qualitative evaluation using a benchmark composed of four real-world software projects of various sizes. The quantitative evaluation investigates the efficiency of our approach in fixing four types of code-smells (Blob, feature envy, data class, and spaghetti code), introducing three types of design patterns (Factory Method, Visitor, and Singleton) (cf. Appendix C), and improving six quality attributes according to the popular software quality model QMOOD [193] (cf. Appendix A). For the qualitative evaluation, we conducted a non-subjective evaluation with potential users to evaluate the usefulness of our refactoring tool.

7.3 Approach: MORE

This section describes the principles that underlie the proposed approach, called MORE (Multi-Objective REfactoring recommendation) for improving software quality, fixing code-smells, and introducing design patterns while maintaining the coherence of the refactored code. We first describe our approach, its components and the semantic constraints employed and then provide a detailed description of the adaptation of NSGA-II.

7.3.1 Approach overview

The general structure of MORE is described in Figure 7.1. It takes as input the source code of the program to be refactored, and as output it produces a sequence of refactorings that find the optimal trade-off between: 1) improving quality, 2) fixing code-smells, and 3) introducing design patterns. MORE comprises seven components that will be described in the following paragraphs.

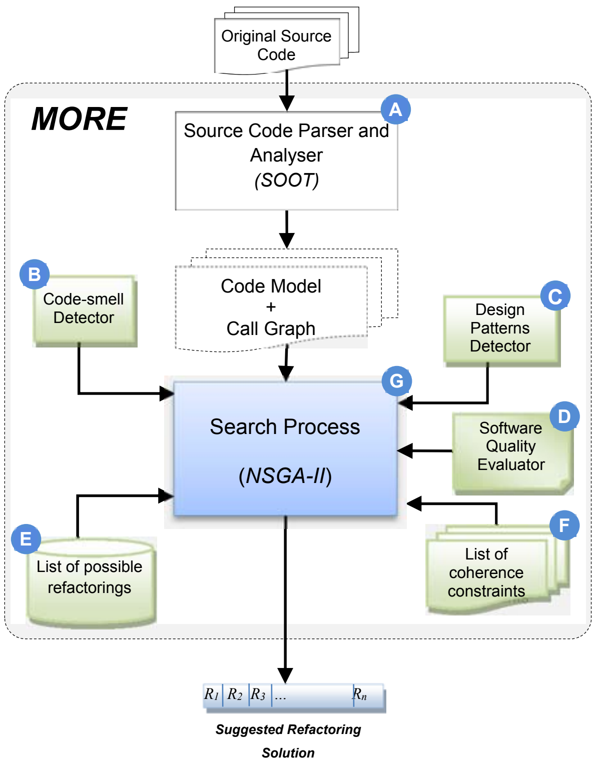

Original Source Code
MORE
A
Source Code Parser and Analyser (SOOT)
B
Code-smell Detector
Code Model + Call Graph
C
Design Patterns Detector
G
Search Process (NSGA-II)
D
Software Quality Evaluator
E
List of possible refactorings
F
List of coherence constraints
R1 R2 R3 ... Rn
Suggested Refactoring Solution

Figure 7.1 - Architecture of MORE.

Source code parser and analyzer (label A). This component aims at parsing and analyzing the source code of the program being refactored. We are using Soot [190], a Java optimization framework. The original source code is analyzed in order to extract from it the relevant code elements (i.e., classes, methods, attributes, etc.) and the existing relationships between them. The outputs are 1) the parsed code in a specific representation that is simple to manipulate during the search process, and 2) a call graph for the entire program that will be used for calculating semantic constraints and software metrics (e.g., coupling, cohesion, etc.).

Code-smell detector (label B). This component scans the entire software program in order to find existing anti-pattern instances using a set of code-smells detection rules [7]. Detection rules are expressed in terms of metrics and threshold values. Each rule detects a specific code-smell type (e.g., *Blob*, *feature envy*, etc.) and is expressed as a logical combination of a set of quality metrics/threshold values. These detection rules are generated/learned from real instances of code-smells using genetic algorithm [7]. When executed, the code-smells detector returns a list of existing code-smell instances in the current version of the program.

Design pattern detector (label C). This component is responsible for detecting existing design pattern instances in the code being refactored. Extensive research has been devoted to develop techniques to automatically detect instances of design patterns in the code and design levels. In our approach, we are using a detection mechanism that is inspired by the work of Heuzeroth et al. [196]. A design pattern *P* is defined by a tuple of program elements such as classes, methods conforming to the restrictions or rules of a certain design pattern. The detection strategy [196] is based on static and dynamic specifications of the pattern. In MORE, we use only the static specifications with a post-processing step to eliminate redundancies. Static specifications are based on predicates to identify the types of code elements like classes, methods, calls, etc. and relate them to the roles in the pattern. Each design pattern $P=(S_c, S_r)$ is then identified as a tuple of code elements S_c that are components of *P*, and a set of binary relations S_r between them. For instance, according to its specifications, the Factory method pattern is defined as follows: $P_{FactoryMethod}=(S_c, S_r)$ where

- S_c={*AbstractCreator, ConcreteCreator, ProductInterface, ConcreteProduct, FactoryMethod, ConcreteFactoryMethod*} represents the code elements involved in the design pattern.

- S_r represents the minimum set of binary relations between the elements of S_c that should be satisfied for the current design pattern, i.e.,
  {ConcreteCreator inherits from AbstractCreator,
  ConcreteProduct implements ProductInterface,
  AbstractCreator defines FactoryMethod,
  FactoryMethod returns ProductInterface,
  ConcreteCreator defines ConcreteFactoryMethod,
  ConcreteFactoryMethod returns ConcreteProduct,
  ConcreteFactoryMethod overrides FactoryMethod}

Software quality evaluator (label D). This component consists of a set of software metrics that serves to evaluate the software design improvements after refactoring. Hence, the expected benefit from refactoring is to enhance the overall software design quality, as well as fixing code-smells [18]. We use, in our approach the QMOOD (Quality Model for Object-Oriented Design) model [193] to estimate the effect of the suggested refactoring solutions on quality attributes.

List of refactorings (label E). The MORE tool currently supports the following refactoring types: Move method, Move field, Pull up field, Pull up method, Push down field, Push down method, Inline class, Extract method, Extract class, Move class, Extract superclass, Extract subclass, and Extract interface [1]. We selected these refactoring because they are the most frequently used and they are implemented in modern IDEs such as Eclipse and Netbeans.

We also considered specific blocks of refactorings to automatically introduce different types of design pattern instances. We are referring to some guidelines given in the literature for introducing instances of design patterns [124] [197]. MORE currently supports the following three design pattern types: Visitor, Factory Method, and Singleton.

Introduce Visitor pattern. To introduce a visitor pattern, a sequence of refactoring operations should be applied in the right order. Algorithm 7.1 illustrates the necessary refactorings to be applied to introduce a Visitor pattern. The starting point is a class hierarchy *H* that has a superclass/interface *SC* and a set of subclasses *CC*. The first step is to create for each functional method a corresponding visitor class (lines 5-10). Then, functional code fragments should be moved from the class hierarchy *H* to the visitor classes. To this end, we apply the Extract Method refactoring to extract the functional code from the functional methods (line 14). The original method will now simply delegate the new extracted one (at a later stage, these methods can be deleted and their call sites updated to use the appropriate visitor). The extracted method will be moved from the class hierarchy to the appropriate newly created visitor class (line 15). The new methods in visitor classes are named "*visit**" using a Rename Method refactoring (line 16). An abstract Visitor class is introduced as a superclass for all the created visitors using an Extract Superclass refactoring (line 19). Now, an "*accept"* method is introduced in all the subclasses *CC* in *H* by extracting it from the initial methods, using an Extract Method refactoring (line 22). All functional methods now call the *accept* method with an instance of the appropriate Visitor subclass. Therefore, their definition can be pulled up to the *SC* class by using a Pull Up Method refactoring.

```
1.  input: hierarchy H
2.  SC = getSuperClass(H)
3.  CC = getSubClasses(H)
4.  visitors = ∅
5.  for each method m in SC do
6.    if(m ∉ SC.constructors())
7.      v = CreateEmptyClass(m.name)
8.      v = renameClass(c.name+"visitor")
9.      visitors = visitors ∪ {v}
10. end
11. for each class c in CC do
12.     for each method m in c do
13.        visClass = V(m)//find visitor class that maps to the name of method
14.        extractMethod(c, m, m1)
15.        moveMethod(c, m1, visClass)
16.        renameMethod(visClass, m1, "visit"+c.name)
17.      end
18. end
19. Visitor=extractSuperClass(Visitors,"Visitor"+SC.name)
20. for each class c in CC do
21.     for each method m in c do
22.         extractMethod(c, m, "accept")
23.         pullUpMethod(m, c, SC)
24.      end
25. end
```

Algorithm 7.1 - Pseudo-code to introduce the Visitor design pattern.

Introduce Factory Method pattern. As described in Algorithm 7.2, which uses the approach developed by Ó Cinnéide and Nixon [127], a Factory Method pattern can be introduced starting from a *Creator* class that creates instances of *Product* class(es). The first step is to apply an extract interface refactoring (line 2) to abstract the public methods of the Product classes into an interface. All references to the Product classes in the Creator class are then updated to refer to this interface (lines 3-6). Then, for each constructor in each of the Product classes, a similar method is added in the Creator class that returns an instance of the correspondent Product class (lines 7-14). Finally all creations of Product objects in the Creator class are updated to use these new methods (line 15-18).

```
1.   input: Class Creator, Class [] Products
2.   extractInterface(Products[], “abstract”+ Products.getName())
3.   for each Object o in Creator do
4.      if o.getType ∈ Products[] then
5.         o.renameType(o.getType()+“abstract”+ o.getType())
6.   end
7.   for each p ∈ Products[] do
8.     for each constructor c in p do
9.       m = addMethod(Creator, “create”+p.name());
10.      m.setReturnType(“abstract”+p.name());
11.      m.setParamList=c.paramList;
12.      m.setBody=(“return new P(”+c.paramList+“);”);
13.    end
14.  end
15.  for each Object o in Creator do
16.    if o.getType ∈ Products[] then
17.      Creator.replaceObjectCreations(o.getType(), “create”+ o.getType());
18.  end
```

Algorithm 7.2 - Pseudo-code to introduce the Factory Method design pattern.

Introduce Singleton pattern. Our formulation for the Singleton pattern is derived from [198] and [199]. Algorithm 7.3 describes the basic steps to introduce the Singleton Pattern. A Singleton class can be introduced starting from a candidate class `Singleton`. The first step (line 2) is to apply the classic refactoring operation, defined in Fowler's catalog [25], Replace Constructor with Factory Method. The aim of this step is make the constructor private. Then access to this class will be performed via the newly generated static method `getSingleton()`, which will be the global access point to the Singleton instance. The second step is to create a static field `singleton` of type `Singleton` with access level private (line 3) that will be initialized to "`new Singleton()`" in the body of the new method `getSingleton()` (line 5). The selection statement ensures that the field `singleton` is instantiated only once, i.e., when it is null.

```
1.  input: Class Singleton
2.  Replace_Constructor_with_Factory_Method(Singleton.constructor,
    “get”+ Singleton.name);
3.  addField(singleton, Singleton, private, static);
4.  if(singleton == null)
5.      initialize(singleton, “new Singleton()”);
6.  end
```

Algorithm 7.3 - Pseudo-code to introduce the Singleton design pattern.

We selected these three design patterns because they are frequently used in practice, and it is widely believed that they embody good design practice [195]. The algorithms here apply a typical implementation of the pattern, and leave to the developer the task of tailoring the implementation to fit the context, if necessary. Note that if an atomic refactoring fails due to a non-satisfied precondition, the whole refactoring sequence that applies the design pattern will be rejected.

Coherence constraints checker (label F). The aim of this component is to prevent incoherent changes to code elements. Most refactorings are relatively simple to implement and it is straightforward to show that they preserve behaviour assuming their pre-conditions are true [17]. However, until now there is no consensual way to investigate whether a refactoring operation is semantically feasible and meaningful [29]. Preserving behavior does not mean that the coherence of the refactored program is also preserved. For instance, a refactoring solution might move a method `calculateSalary()` from the class `Employee` to the class `Car`. This refactoring could improve program structure by reducing the complexity and coupling of the class `Employee` while preserving program behavior. However, having a method `calculateSalary()` in the class `Car` does not make sense from the domain semantics standpoint. To avoid this kind of problem, we defined a set of semantic coherence constraints that must be satisfied before applying a refactoring in order to prevent incoherent changes to code elements.

Search process (label G). Our approach is based on a multi-objective optimization using the Non-dominated Sorting Genetic Algorithm (NSGA-II) [24] to formulate the refactoring suggestion problem. We selected NSGA-II because it is widely-used in the field of multi-objective optimization, and demonstrates good performance compared to other existing metaheuristics in solving many software engineering problems [91]. Thus our approach can be classified as Search Based Software Engineering (SBSE) [91] for which it is established best practice to define a representation, fitness functions and computational search algorithm. Referring to Figure 7.1, the search process takes as input the source code that is then parsed into a more manipulable representation (label A), a set of code-smell detectors (label B), a set of design patterns detectors (label C), a software quality evaluator (label D) that evaluates post- refactoring software quality, a set possible refactoring operations to be applied (label E), and set of constraints (label F) to ensure semantic coherence of the code after refactoring. As output, our approach suggests a list of refactoring operations that should be applied in the right order to find the best compromise between fixing anti-patterns, introducing design patterns, and improving design quality.

7.3.2 Semantic constraints

Unlike existing automated refactoring approaches, MORE defines and uses a set of semantic constraints to prevent arbitrary changes that may affect the semantic coherence of the refactored program. Hence, applying a refactoring where it is not needed is highly undesirable as it may introduce semantic incoherence and unnecessary complexity to the original design. To this end, we considered several semantic constraints that we defined in Section 6.3.3 including Vocabulary-based similarity constraint (VS), Dependency-based similarity constraint (DS), Implementation-based similarity constraint (IS), Feature inheritance usefulness constraint (FIU), and Cohesion-based dependency constraint (CD).

Furthermore, we introduced some semantic constraints related to the introduction of design patterns. Before introducing a design pattern to a particular design fragment, the basic intent of the pattern should exist in that design fragment already. This starting point is termed a “precursor” in the nomenclature of Ó Cinnéide and Nixon [127], and is not taken into account in much of the existing work in automated refactoring. MORE formulates the notion of precursor as a set of semantic constraints that should be satisfied when introducing design patterns.

The semantic constraint we use for the Factory Method pattern is that the Creator class must create a concrete instance of a Product class [127]. This situation could require the application of the Factory Method pattern, if the developer decides that the Creator class should be able to handle several different types of Product. MORE analyzes, using Soot [190], all the method bodies of a candidate Creator class to retrieve statements containing the operator “new” that occur within its functional methods’ body. If the candidate Creator class does not create instances of the Product class, then there is no need to introduce a Factory Method pattern.

The semantic constraints for the Visitor pattern involve the situation when it is required to accumulate new information from an object structure, but the classes of objects in the structure do not support the required behavior [199]. This relates in general to complex hierarchies that have a large number of inherited methods or with God classes that can be detected [127].

The semantic constraints we use for the Singleton pattern is that the class under refactoring (the candidate Singleton): 1) has only one instance, and 2) provide a global point of access to it, i.e., a method called from other classes in the system. These two constraints can be checked using static program analysis technique.

7.3.3 Multi-objective formulation of MORE

a) Search technique

MORE uses NSGA-II, one of the most popular algorithms that have shown good performance in solving SE problems based on recent surveys [90]. A detailed description of NSGA-II is given in Section 2.3.4.

b) Solution representation

In our NSGA-II design, we use the same vector-based solution representation adopted in our GA adaptation. The description of our solution representation is detailed in Section 3.3.2.

c) Solution evaluation

To evaluate the fitness of each refactoring solution, we used three objective functions according to each objective.

- **Code-smells objective function:** It calculates the ratio of the number of corrected code-smells to the initial number of anti-patterns using the anti-patterns detector component. The anti-patterns correction ratio (CCR) is defined as follows:

$$CCR = \frac{number\ of\ corrected\ antipatterns}{initial\ number\ of\ antipatterns}$$

- **Design patterns objective function:** It calculates the number of produced design pattern instances (NP) using the design patterns detector component. The NP values are then normalized in the range [0, 1] using min-max normalization. NP is defined as follows:

$$NP = \sum produced\ design\ patterns$$

- **Quality objective function:** It calculates the quality improvement. MORE use the QMOOD (Quality Model for Object-Oriented Design) model [193] to estimate the effect of the suggested refactoring solutions on quality attributes. We calculate the overall quality gain (QG) for the six QMOOD quality factors (reusability, flexibility, understandability, effectiveness, functionality, and extendibility) that are formulated using 11 low-level design metrics. Full details about QMOOD are available in Bansiya and Davis original work [193] and Appendix A. Let $Q=\{q_1, q_2,... q_6\}$ and $Q'=\{q'_1, q'_2,... q'_6\}$ be respectively the set of quality attribute values before and after applying the suggested refactorings, and {w_1, w_2,... w_6} the weights assigned to each of these quality factors. Then the total quality gain (QG) is estimated as follows:

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

d) Selection and Change operators

To guide the selection process, NSGA-II uses a binary tournament selection based on dominance and crowding distance [24]. NSGA-II sorts the population using the dominance principle, which classifies individual solutions into different dominance levels. Then, to construct a new population, NSGA-II uses a comparison operator based on a calculation of the crowding distance [24] to select potential individuals having the same dominance level.

We use the same genetic operators formulation adopted in our GA adaptation. The description of our genetic operators (crossover and mutation) is detailed in Section 4.3.2.d).

7.4 Design of the experimental study

To evaluate the efficiency of our approach in fixing code-smells, introducing design patterns and improving design quality, we conducted a quantitative and qualitative evaluation with participants from both academia and industry.

7.4.1 Research questions

With this study, we intend to answer the following five research questions:

- **RQ1.** To what extent can the proposed approach improve the quality of software systems?
- **RQ2.** How does our approach perform compared to existing search-based refactoring approaches?
- **RQ3.** How does our approach perform compared to existing non-search-based refactoring approaches?

- **RQ4.** How does NSGA-II perform compared to random search and other multi-objective algorithms?
- **RQ5.** Is our approach useful for software engineers in a real-world setting?

7.4.2 Systems studied

We applied our approach to a benchmark composed of four medium and large-size open-source Java projects: Xerces-J[29], GanttProject[30], AntApache[31], and JHotDraw[32]. Xerces-J is a family of software packages for parsing XML. GanttProject is a cross-platform tool for project scheduling. AntApache is a build tool and library specifically conceived for Java applications Finally, JHotDraw is a GUI framework for drawing editors.

Table 7.1 provides some descriptive statistics about these four programs. We selected these systems for our validation because they came from four different organisations, involved different kinds of software engineering development and had different sizes, ranging from 21 to 240 KLOC with a large number of both design pattern and anti-pattern instances. As we previously note, in these corpora, we considered four different code-smell types (god class, feature envy, data class, and spaghetti code) and three different design patterns (Abstract Method Factory, Visitor and Singleton). Please refer to Appendix C for the definition of these code-smells and design patterns.

Systems	**Release**	**# classes**	**KLOC**	**# code-smells**	**# design patterns**
Xerces-J	v2.7.0	991	240	81	36
GanttProject	v1.10.2	245	41	49	15
AntApache	v1.8.2	1191	255	92	38
JHotDraw	v 6.1	585	21	24	18

Table 7.1 – Systems statistics.

[29] http://xerces.apache.org/xerces-j/
[30] www.ganttproject.biz/
[31] http://ant.apache.org/
[32] http://www.jhotdraw.org/

7.4.3 Analysis method and evaluation metrics

We designed our experiments to answer our research questions. To answer **RQ1**, we conduct a quantitative and qualitative evaluation to evaluate the efficiency of our approach:

Quantitative evaluation. We evaluate the efficiency of our approach for 1) fixing code-smells, 2) introducing design patterns, and 3) improving software quality.

- To evaluate the efficiency of our approach in fixing code-smells, we calculate the code-smells correction ratio (CCR) on our benchmark.
- To evaluate the efficiency of our approach in introducing design patterns, we calculate the number of new design pattern instances (NP) that are introduced.
- To evaluate the efficiency of our approach in improving software quality, we calculate the overall quality gain (QG) using the QMOOD (Quality Model for Object-Oriented Design) model [193].

Qualitative evaluation. To evaluate the usefulness of the suggested refactorings, we performed a qualitative evaluation with six PhD students in Software Engineering; two of whom are working at General Motors as senior software engineers. The participants have an average of 6.5 years programming experience in Java and familiar with the evaluated open-source systems. We asked the participants to manually evaluate, for each system, 10 refactoring operations that are selected at random from the suggested refactoring solutions. Participants assign a correctness score of 0 or 1 for each refactoring according to its coherence with the program semantics. Participants were aware that they are going to evaluate the semantic coherence of refactoring operations, but do not know the particular experimental research questions (the approaches and algorithms being compared). To this end, we define the metric refactoring meaningfulness (RM) that corresponds to the number of meaningful refactoring operations, in terms of semantic coherence, over the total number given to the participants to evaluate. RM is defined as follows:

$$RM = \frac{\#\, meaningful\ refactorings}{\#\, evaluated\ refactorings}$$

To answer **RQ2**, we compared our approach to state-of-the-art approaches that use SBSE in terms of ACR, NP, and the QG. Hence, for a proposed approach to be adopted it must also outperform the state of the art for the problem in hand. To this end, we compared our approach to Seng et al. [21], Jensen et al. [126], and our GA-based approach (describes in Chapter 4). These approaches are designed each for a specific purpose, i.e., improve quality metrics or fix design patterns. Thus, to make the comparison fair, we apply the suggested refactorings of each approach, and we calculate our evaluation metrics (CCR, NP, QG, and RM)

To answer **RQ3**, we compared our refactoring results with a popular anti-patterns detection and correction tool JDeodorant [83] that do not use heuristic search techniques in terms of ACR and RM. The current version of JDeodorant [83] is implemented as an Eclipse plug-in that identifies certain types of code-smells using quality metrics and then proposes a list of refactoring operations to fix them. For instance, to fix God class, JDeodorant suggests standard refactoring solution based on a move method refactorings.

To answer **RQ4,** we used mainly two performance indicators to compare the different algorithms used in our experiments. 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. Different metrics for measuring the performance of multi-objective optimization methods exist. Zitzler et al. [200] provide a comprehensive review of quality indicators for multi-objective optimization, finding that many commonly used metrics do not reliably reflect the performance of an optimization algorithm. One of the few recommended metrics is the *Hypervolume* and the *Spread* indicators.

- *Hypervolume (HV)*: this metric calculates the proportion of the volume covered by members of a non-dominated solution set returned by the algorithm. A higher Hypervolume value means better performance, as it indicates solutions closer to the optimal Pareto Front. The most interesting features of this indicator are its Pareto dominance compliance and its ability to capture both convergence and diversity.

- *Spread (Δ)*: It measures the distribution of solutions into a given front. The idea behind the spread indicator is to evaluate diversity among non-dominated solutions. An ideal distribution has zero value for this metric when the solutions are uniformly distributed. An algorithm that achieves a smaller value for Spread can get a better diverse set of non-dominated solutions. For further details about the formulation of these indicators, please refer to [200] and [201].

To answer **RQ5,** we asked our participants to manually evaluate the usefulness of the introduced design patterns in the current software design by assigning a usefulness score in the range [0,5]. We consider a design pattern as useful if its assigned score is ≥3. We define the metric design patterns usefulness (PU) as follows.

$$PU = \frac{\#\, useful\ design\ patterns}{\#\, introduced\ design\ patterns}$$

Due to the stochastic nature of the algorithms/approaches we are studying, they can provide different results for the same problem instance from one run to another. To cater for this issue and to make inferential statistical claims, our experimental study is performed based on 31 independent simulation runs for each algorithm/technique studied. The obtained results are statistically analyzed using the Wilcoxon rank sum test [202] with a 95% confidence level ($\alpha = 5\%$). The Wilcoxon rank sum test is applied between NSGA-II and each of the other techniques: Seng et al. [21], Jensen et al. [126], and GA-based approach. Our tests show that the obtained results are statistically significant with p-value<0.05.

7.4.4 Algorithms parameter tuning

An important aspect of research on metaheuristic search algorithms is the selection and tuning of the algorithms' parameters, which is necessary in order to ensure not only fair comparisons, but also for potential replication. To this end, we report our algorithmic parameter tuning and selection used to facilitate replication of our findings. The initial population/solutions of NSGA-II, are completely random. The stopping criterion is when the maximum number of function evaluations, set to 120,000, is reached. The crossover operator performs crossover with a probability 0.6. The mutation probability used is 0.3. The semantic constraint thresholds were fixed as follows: VS≥0.55, DS≥0.41, IS≥0.6, FIU≥0.76, and CD≥0.5. After several trial runs of the simulation, the parameter values were fixed. There are no general rules to determine these parameters [203], and therefore we set the combination of parameter values by trial and error. For each algorithm, we repeat the simulation 31 times in each case, and compute the median value.

7.5 Results

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

Results for RQ1. The results relating in RQ1 are described in Table 7.2. After applying the proposed refactoring operations by our approach (MORE), we found that, on average, 86% of the detected code-smells were fixed (CCR) for all the four studied systems. This high score is considered significant to improve the quality of the refactored systems by fixing the majority of code-smells that were from different types (God class, Feature Envy, Data Class, and Spaghetti Code). We found that the majority of non-fixed code-smells are related to the *God class* type. This type of code-smell usually requires a large number of refactoring operations and is known to be very difficult to fix.

Moreover, we found that MORE succeeded in producing design pattern instances. Table 7.2 shows the number of new design pattern instances for each system. MORE successfully introduced an average of 7 design patterns (NP) that were from different types (Factory Method, Visitor and Singleton) for all the four studied systems. This can be very helpful for software engineers who might be interested to the introduction of design patterns to make their software systems more understandable, flexible, and maintainable. In addition, when applying the suggested refactorings we noticed that some God classes are fixed when involved in introducing a visitor pattern. For instance, we observe that the God class `GanttTree` in GanttProject, was fixed automatically when introducing a Visitor pattern. In addition, the new structure of this class become more flexible with the Visitor pattern where new functionalities and behavior can be easily added without affecting the original class.

Systems	**Algorithms**	**ACR**		**NP**		**QG**	
		Score	*p-value*	*Score*	*p-value*	*Score*	*p-value*
Xerces-J	MORE	89%		12		0.47	
	Seng et al.	23%	< 0.05	0	< 0.01	0.54	< 0.02
	Jensen et al.	14%	< 0.04	31	< 0.01	0.41	< 0.01
	GA-based approach	88%	< 0.04	0	< 0.01	0.32	< 0.01
GanttProject	MORE	88%		7		0.34	
	Seng et al.	24%	< 0.02	1	< 0.01	0.33	< 0.01
	Jensen et al.	33%	< 0.05	14	< 0.01	0.35	< 0.01
	GA-based approach	84%	< 0.05	0	< 0.01	0.21	< 0.01
AntApache	MORE	86%		4		0.5	
	Seng et al.	7%	< 0.04	0	< 0.01	0.52	< 0.01
	Jensen et al.	12%	< 0.01	28	< 0.02	0.51	< 0.01
	GA-based approach	87%	< 0.01	0	< 0.01	0.39	< 0.01
JHotDraw	MORE	83%		4		0.17	
	Seng et al.	38%	< 0.01	0	< 0.01	0.19	< 0.01
	Jensen et al.	25%	< 0.01	9	< 0.01	0.14	< 0.01
	GA-based approach	88%	< 0.01	0	< 0.01	0.1	< 0.01
Average (all systems)	MORE	86%		7		0.37	
	Seng et al.	23%		0.25		0.39	
	Jensen et al.	21%		20.5		0.35	
	GA-based approach	86%		0		0.25	

Table 7.2 - ACR, NP, and QG median values of 31 independent runs of MORE, Seng et al., Jensen et al., and GA-based approach.

In terms of quality improvement (QG), as can be seen in Table 7.2, MORE succeeded in improving the quality of the four studied systems, with an average QG score of 0.37 in terms of QMOOD quality attributes. In Figure 7.2, we show the obtained QG values that we calculated for each QMOOD quality attribute before and after refactoring for each studied system. We found that the systems quality increase across the four QMOOD quality factors. Understandability is the quality factor that has the highest QG value; whereas the effectiveness quality factor has the lowest one. This due to two possible reasons 1) the majority of non-fixed code-smells (God class, spaghetti code) are known to increase the coupling (DCC) within classes which heavily affect the quality index calculation of the Effectiveness factor; 2) the vast majority of suggested refactoring types were move method, move field, and extract class that are known to have a high impact on coupling (DCC), cohesion (CAM) and the design size in classes (DSC) that serves to calculate the understandability quality factor. Furthermore, we noticed that JHotDraw produced the lowest quality increase for the four quality factors. This is justified by the fact that JHotDraw is known to be of good design and implementation practices [28] and it contains few code-smell instances comparing to the three other studied systems.

The p-values of the Wilcoxon rank sum test indicate whether the median of the approach (Seng/Jensen/GA-based approach) is statistically different from MORE with a 95% confidence level ($\alpha = 0.05$). A statistical difference is accepted at $p<=0.05$.

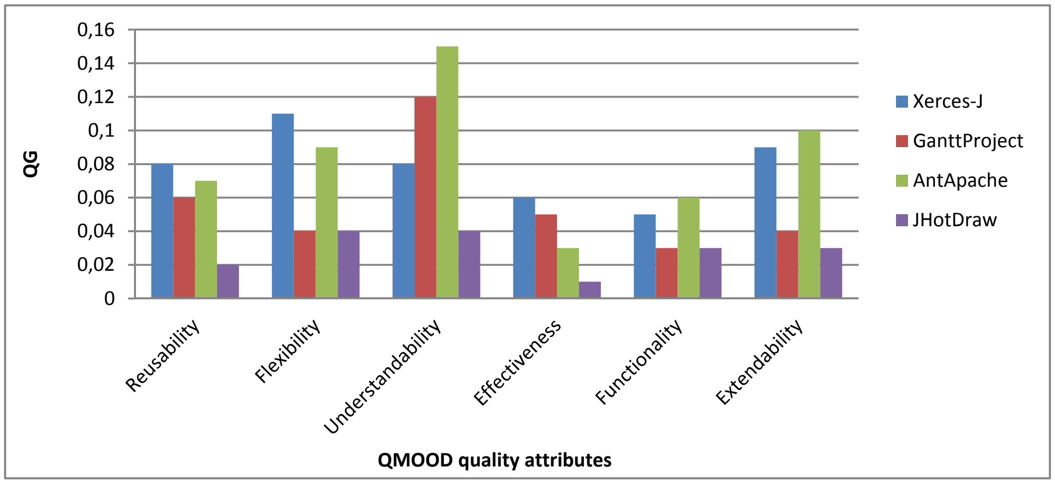


Figure 7.2 - QMOOD quality factors gain obtained by MORE.

The obtained results are promising, however, improving the design structure is not always enough to determine whether our approach produce a coherent program and fit with software engineers expectations. Figure 7.3, describes the results of our qualitative evaluation. We found that the majority of the suggested refactorings (an average of 86% over the four studied systems) could be successfully applied to the program and only a small number of the suggested refactorings were rejected by the participants due to semantic incoherencies in the source code.

To sum up, we can conclude that our approach succeeded in improving the code quality not only by fixing the majority of detected anti-patterns and introducing a considerable number of design patterns but also by a significant improvement on the overall design quality of the refactored program such as the user understandability, the reusability, and the flexibility. At the same time, the majority of the proposed refactoring operations are considered as semantically feasible and do not affect the semantic coherence of the refactored program from the point of view of potential users.

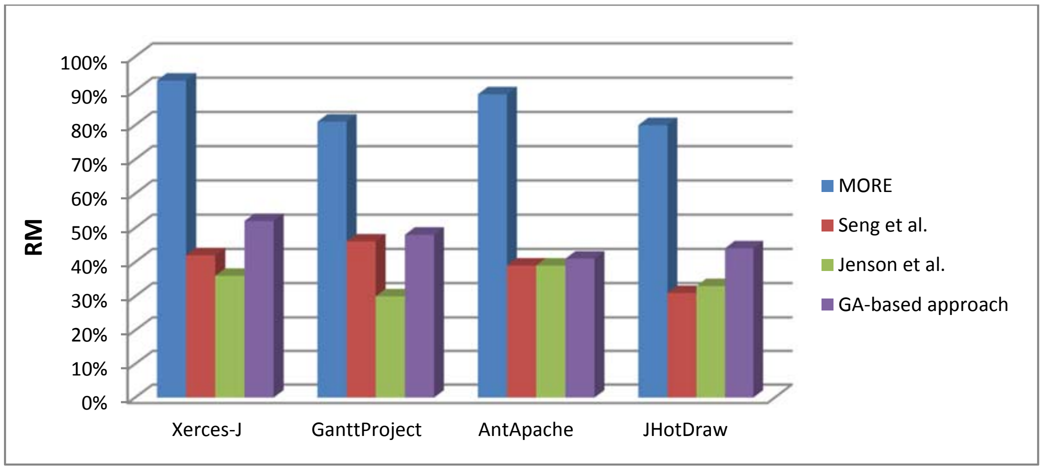


Figure 7.3 - Refactoring meaningfulness (RM) evaluation.

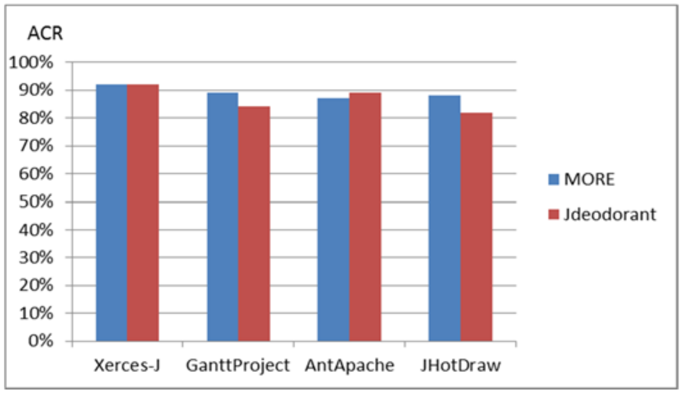
Results for RQ2. The results relating to RQ2 are summarized in Table 7.2. As described in Table 7.2, after applying the proposed refactoring operations, we found that more than 86% of detected code-smells were fixed (CCR) as an average for all the four studied systems. For instance, for GanttProject, 75% (9 over 12) of God classes, 86% (6 over 7) of feature envy, 94% (15 over 16) of spaghetti code, 93% (13 over 14) of Data classes are fixed. This score is comparable to the correction score of GA-based approach having an average of 86%. However, the obtained results are much better than those of Seng et al., and Jensen et al. having respectively only 23% and 21%, on average for all the studied systems.

In terms of patterns introduction, Jensen et al. produces the higher score by introducing, on average for the four systems, 20.5 design patterns. This score is higher than the one obtained by MORE (an average of 7 patterns per system). This can be explained by the fact that Jensen et al. apply design patterns without considering if the design pattern is needed or not in that code fragment, i.e., the sole aim is to produce more design patterns. This is unlikely to be useful and efficient in practice because introducing a design pattern where it is not needed may increase the complexity of the system. For Seng et al. and GA-based approach, we found that they are not able to produce design patterns. This is because the lists of refactorings they use are not geared for the introduction of design patterns.

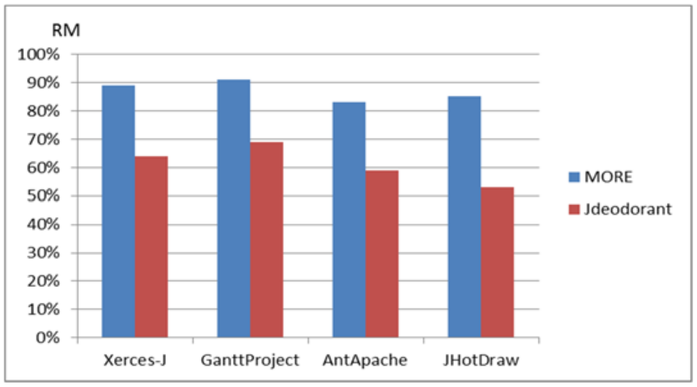
Furthermore, MORE produces comparable QG values to Seng et al. and Jensen et al. having respectively 0.37, 0.39 and 0.35, since the quality metrics improvement is a common component in the objective function of each approach. However GA-based approach produces a lower QG score since their approach is driven only by code-smell correction and not by improving quality metrics. On the other hand, despite the significant improvement in terms of QG for Seng et al. (the highest score), it is not effective at fixing code-smells (only 23% of anti-patterns are fixed). Thus these results provide evidence to support the claim that improving quality metrics does not necessarily mean that existing anti-patterns are fixed.

More notably, we compared MORE to the three other approaches in terms of semantic coherence. Figure 7.3 summarizes our findings. Regarding the refactoring meaningfulness, for all of our four studied systems, an average of 86% of proposed refactoring operations are considered as semantically feasible and do not generate semantic incoherence. This score is significantly higher than the scores of the three other approaches having respectively only 40%, 35% and 46%, as RM scores for respectively, Seng et al., Jensen et al., and GA-based approach. Thus, our approach performs clearly better for RM. Moreover, we noticed that for the larger programs, the performance in terms of refactoring meaningfulness (RM) achieved by MORE is more notable than it is for the smaller programs.

Results for RQ3. JDeodorant uses only structural information to detect and fix code-smells at the code level but not all the four code-smell types that we considered in our experiments. Thus, to make the comparison fair, we performed our comparison using only the two common code-smells, God class and Feature envy, which can be fixed by both tools. Figure 7.4 summarizes our findings. The average number of fixed code-smells is comparable between both tools; however MORE is clearly far better in terms of semantically coherent refactorings. This can be explained by the fact that JDeodorant uses only structural metrics to evaluate the impact of suggested refactorings on the detected code-smells. In addition, it is also worth noting that MORE supports more refactoring types and addresses more code-smell types than does JDeodorant.



(a)

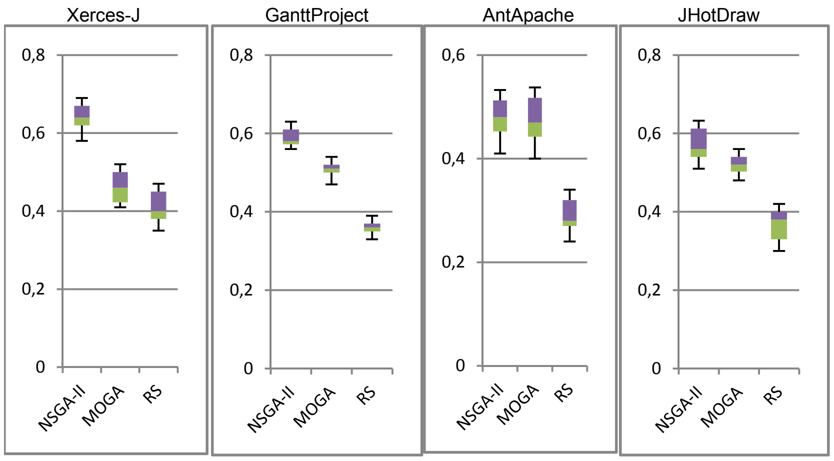


(b)

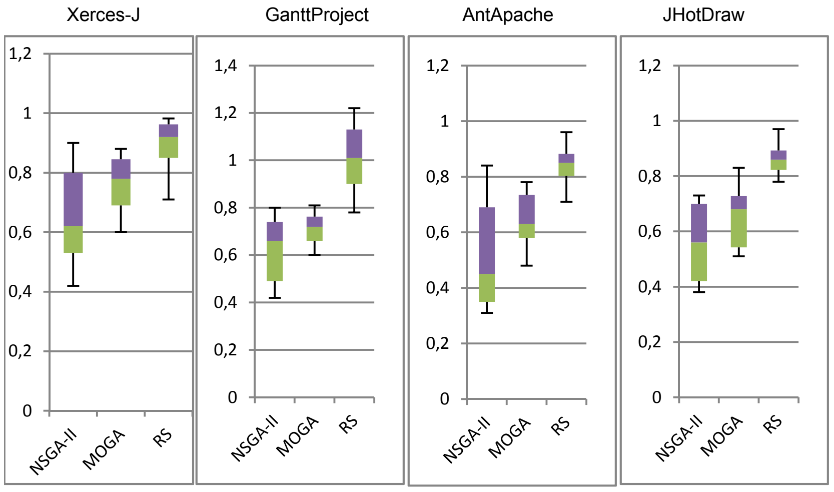
Figure 7.4 - Comparison of our approach with JDeodorant in terms of (a) CCR and (b) RM.

Results for RQ4. Figure 7.5 presents the results for RQ4 using the two quality indicators Hypervolume (HV) and Spread (Δ) through 31 runs of NSGA-II, MOGA, and Random

Search. For the HV, the higher the value, the better the algorithm performance, whereas, for the Δ, the lower value, the better the algorithm performance.



(a) Hypervolume indicator



(b) Spread indicator

Figure 7.5 - Boxplots using the quality measures (a) HV, and (b) Spread applied to NSGA-II, MOGA, and Random Search through 31 independent run.

According to the obtained results in Figure 7.5 (a), Random Search results are generally poor, whereas NSGA-II and MOGA obtain good results for the five systems. Moreover, as illustrated in Figure 7.5 (a), NSGA-II significantly outperforms MOGA when applied to

Xerces, GanttProject, and JHotDraw while presenting a similar performance for AntApache. This fact confirms the effectiveness of NSGA-II over MOGA in finding a well-converged and well-diversified set of Pareto-optimal refactoring solutions. For the Δ, is also desired that a multi-objective evolutionary algorithm maintains a good spread of solutions in the obtained set of solutions. Figure 7.5 (b) shows that NSGA-II has the better spread among all the other search-based algorithms in all systems.

To summarize, the Wilcoxon rank sum test showed that in 31 runs, both NSGA-II and MOGA 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 RQ4).

Results for RQ5. The results relating to RQ4 are summarized in Table 7.3. We observe that the majority (more than 83%) of the design patterns produced by MORE are considered as useful in the four studied systems since their introduction is guided by a set of semantic constraints and not arbitrary. However, we found that a relatively small number of patterns produced by Jensen et al. (less than 36%) are considered as feasible by the group of software engineers. The main reason is that these design patterns are applied in an arbitrary way, without considering if they are needed in that code fragment or not.

Thus MORE produces higher increases in RM than the other three approaches, which is probably the cause of the significant score in terms of patterns usefulness.

Systems	**MORE**	**Jensen et al.**
Xerces-J	83% (10|12)	35% (11|31)
GanttProject	86% (6|7)	36% (5|14)
AntApache	100% (4|4)	14% (4|28)
JHotDraw	100% (4|4)	22% (2|9)

Table 7.3 - Comparison of MORE with Jensen et al. in terms of Patterns usefulness (PU).

7.6 Conclusion

We presented, in this chapter, an automated multi-objective refactoring recommendation approach to improve design quality (as defined by software quality metrics), fix code-smells, and introduce design patterns. To evaluate our approach, we conducted a quantitative and qualitative evaluation with software engineers using a benchmark composed of four open source systems. The statistical analysis of the results provides evidence that our approach is more efficient comparing to the state-of-the-art of refactoring techniques.

As part of our future work, we are planning to extend the validation of our approach with additional types of design patterns and code-smells. In addition, we intend to conduct an empirical study to investigate the correlation between introducing patterns and their impact on several types of code-smells.

Chapter 8: Conclusion

In this chapter, we summarise the results and conclusions of this thesis. We also discuss the limitations and future research directions.

8.1 Summary of contributions

The main objective of this thesis was to develop an automated approach to recommend refactoring to help software engineers charged with the task of maintaining and evolving existing software systems. To this end, we applied different SBSE techniques which have been shown to be a practical and efficient way in solving several software engineering problems. Software refactoring is ideal for the application of SBSE techniques, in its two steps (1) identification of code fragments to be refactored, and (2) identification of the suitable refactoring operations to apply.

The first contribution of our thesis, described in Chapter 3, is about generating code-smells detection rules to support developers and relieve them from the burden of doing so manually. We see the code-smells detection problem as a combinatorial optimization problem to find the suitable detection rules using examples of code-smells. Typically, researchers and practitioners try to characterize different types of common code-smells and present symptoms to search for in order to locate possible code-smells in a system. In our approach, we have shown that this knowledge is not necessary to perform the detection. Instead, we use examples of code-smells to generate detection rules. Our approach succeeded in detecting the majority of code-smells with 86% of precision and 91% of recall.

The second contribution is about automatically recommending refactoring solutions to fix the detected code-smells. We proposed four principal solutions.

In the first solution, we considered the refactoring recommending task as a single-objective optimization problem as described in Chapter 4. A refactoring solution is defined as a combination of refactoring operations that should minimize, as much as possible, the number of detected code-smells. To this end, we use GA to find the best combination of refactoring operations from a large list of available refactorings. Our approach was tested on six medium and large size software systems and succeeded in fixing more than 90% of the detected code-smells. Indeed, one of the advantages of our approach is that it does not correct code-smells separately since we consider the correction task as a global process instead of a local one. In addition, we don't need to define an exhaustive list of code-smells and specify standard refactoring for each code-smell type.

In the second solution, we extended our GA-based approach to prioritize the correction of code-smells in Chapter 5. We propose an approach that supports automated refactoring recommendation to fix code-smells where more critical code-smells are prioritized during the refactoring process. Hence, we formulated the refactoring recommending problem as an optimization problem to find the near-optimal sequence of refactorings according to a prioritization schema. To this end, we used a novel metaheuristic search by the means of Chemical Reaction Optimization, a newly established metaheuristics, to find the suitable refactoring solutions that maximize the number of corrected code-smells while prioritizing the most important, riskiest, and severest code fragments according to the developer's preferences.

The third solution is described in Chapter 6. We deal with the refactoring recommending task as a multi-objective optimization problem. We explore four objectives to optimize: 1) fix code-smells, 2) reduce the number of modifications/adaptations needed to apply refactorings, 3) preserve the semantic coherence of the refactored program, and 4) maintain the consistency with development/maintenance history. The idea is to find the best compromise between all of these objectives. Hence, by reducing the number of modifications, we reduce the complexity of the recommended refactorings and keep as much as possible the original design/code structure. Moreover, it is mandatory to preserve the semantic coherence and prevent arbitrary changes on code elements. Furthermore, we exploit knowledge from previously applied changes to recommend new refactorings. We evaluate the efficiency of our approach using a benchmark of six different industrial size open-source systems, and six frequent code-smells types through an empirical study conducted with software engineers.

From another perspective, as a fourth solution, we proposed a multi-objective formulation of refactoring recommending task where we consider the introduction of design patterns. We described, in Chapter 7, a recommending framework for automated multi-objective refactoring to find the best compromise between (1) introducing design patterns, (2) fixing code-smells, and (3) improving design quality. To evaluate our approach, we conducted a quantitative and qualitative evaluation with software engineers using a benchmark composed of four open source systems. The obtained results confirm the efficiency of our proposal compared to the state-of-the-art of refactoring techniques.

8.2 Limitations and future research directions

In this section, we discuss some limitations and open research directions related to our proposal. First, for code-smells detection, the performance of our approach depends on the availability of code-smell examples, which could be difficult to collect. We plan to extend our base of examples with additional badly-designed code in order to consider more programming contexts. Another direction worth to explore is to improve the detection of potential code-smells through the use of knowledge from software change history. Indeed, as reported in the literature [64] [118], classes participating in design problems (e.g., code-smells) are significantly more likely to be subject to changes [118]. In other terms, if a code-smell (e.g., God Class) is created intentionally and remains unmodified or hardly undergo changes, the system may not experience any problems [63] [117]. Indeed, it has been shown that, in some cases, a large class might be the best solution [63]. For these reasons, combining software static metrics with software historical metrics can be an effective way to improve the detection of code-smells. Furthermore, we are working on the adaptation of our OO code-smells detection to detect anti-patterns in service-oriented software systems.

For the refactoring step, some limitations can be addressed and different future work directions can be explored. First, our multi-objective approach uses the development change history to recommend new refactorings. Nevertheless, the development change history is not always available especially for newly developed projects. To address this issue, we are working to extend our initial approach to support change history collected from other software systems in similar contexts.

One of the notable limitations of our approach is that recommending refactoring offline a large list of refactorings may be a fastidious task for a software engineer. An important future direction consists of adapting our approach to work dynamically, i.e., online. Code-smells can be detected dynamically when the programmer is writing his code, and a list of possible refactorings can be recommended to help him in fixing the produced code-smells. Such approaches can be very helpful not only for improving software quality, but also for helping programmers to learn from their mistakes. Furthermore, regarding the search process itself, it is very important to put the programmer in the loop. An interactive multi-objective search can be very beneficial to recommend refactoring solutions that take into consideration the programmer’s preferences and needs.

From another perspective, to apply some refactoring operations such as extract class, extract method, our approach assign arbitrary names to the modified code elements. However it is important to recommend consistent names for classes and method involved in refactoring. As part of our future work, we intend to automatically recommend consistent names for the refactored code elements derived from the used vocabulary.

Furthermore, the work conducted in this theses lead as to think about several emprical studies to invertigate some beliefs about code-smells and refactroring. As part of our future research directions we intend to conduct several empirical studies. For instance, it is interesting to investigate 1) the correlation between the number of applied refactorings and the number of code-smells, 2) the correlation between code-smells and QMOOD quality attributes, and 3) the correlation between correcting code-smells and introducing new code-smell instances or fixing other ones implicitly. Moreover, it is interesting to investigate the definition of some code-smells. For instance, a God class is known to be an abnormally large class that monopolize the behaviour of a system. This definition can be empirically investigated through dynamic analysis according to a set of execution scenarios to make sure whether it monopolizes the behaviour.

Related publications

The following is a list of our publications related to this dissertation.

Articles in Journal

1. **Ali Ouni**, Marouane Kessentini, Slim Bechick and Houari Sahraoui, Prioritizing Code-smells Correction Tasks Using Chemical Reaction Optimization, Journal of Software Quality, 2014.
2. **Ali Ouni**, Marouane Kessentini, Mel Ó Cinnéide, Houari Sahraoui, Multi-criteria Software Refactoring: Quality, Code Changes, and Semantics Preservation, IEEE Transactions on Software Engineering, 2014. (under review).
3. Marouane Kessentini, **Ali Ouni**, Philip Langer, Manuel Wimmer and Slim Bechikh, Search-based Metamodel Matching with Structural and Syntactic Measures, Journal of Systems and Software (JSS), pp. 1-14, 2014.
4. Wael Kessentini, Marouane Kessentini, Houari Sahraoui, Slim Bechikh, and **Ali Ouni**, A Cooperative Parallel Search-Based Software Engineering Approach for Code-Smells Detection, IEEE Transactions on Software Engineering, pp. 841-861, 2014.
5. **Ali Ouni**, Marouane Kessentini , Houari Sahraoui and Mohamed Salah Hamdi, Improving Multi-Objective Code-Smells Correction Using Development History. Journal of Systems and Software (JSS), 2014. (under revision).
6. **Ali Ouni**, Marouane Kessentini, Houari Sahraoui, Mel Ó Cinnéide, Kalyanmoy Deb, Automated Multi-Objective Refactoring to Introduce Design Patterns and Fix Anti-Patterns. Journal of Automated Software Engineering, 2014. (submitted).
7. **Ali Ouni**, Marouane Kessentini, Houari Sahraoui and Mounir Boukadoum, Maintainability Defects Detection and Correction: A Multi-Objective Approach, in

Journal of Automated Software Engineering (JASE), 20(1), pp. 47-79, Springer, 2012.

Book Chapters

1. **Ali Ouni**, Marouane Kessentini, Houari Sahraoui, Multi-Objective Optimization for Software Refactoring and Evolution**,** Elsevier, Advances in Computers, volume 94, pp. 103-167, 2014.

Articles in Refereed Conference

1. **Ali Ouni**, Marouane Kessentini, Houari Sahraoui and M. S. Hamdi, The Use of Development History in Software Refactoring Using a Multi-Objective Evolutionary Algorithm, in the Genetic and Evolutionay Computation Conference (GECCO), pp. 1461-1468, July 2013, Amesterdam, The Netherlands. *Invited to a special issue of the Journal of Systems and Software (JSS).*
2. **Ali Ouni**, Marouane Kessentini and Houari Sahraoui, Search-based Refactoring Using Recorded Code Changes, in the 17th European Conference on Software Maintenance and Reengineering (CSMR), pp. 221-230, March 2013, Genova, Italy.
3. **Ali Ouni**, Marouane Kessentini, Houari Sahraoui and M. S. Hamdi, Search-based Refactoring: Towards Semantics Preservation. 28th IEEE International Conference on Software Maintenance (ICSM), pp. 347-356, September 2012, Riva del Garda-Italy.
4. Marouane Kessentini, Wael Kessentini, Houari Sahraoui, Mounir Boukadoum, and **Ali Ouni**, Design Defects Detection and Correction by Example. 19th IEEE International Conference on Program Comprehension (ICPC), pp. 81-90, 22-24 June 2011, Kingston- Canada.

Bibliography

[1] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts: Refactoring – Improving the Design of Existing Code, 1st ed. Addison-Wesley, June 1999.

[2] N. Fenton and S. L. Pfleeger: Software Metrics: A Rigorous and Practical Approach, 2nd ed. London, UK: International Thomson Computer Press, 1997.

[3] R. S. Pressman, Software Engineering – A Practitioner's Approach, 5th ed. McGraw-Hill Higher Education, 2001.

[4] G. Travassos, F. Shull, M. Fredericks, and V. R. Basili, "Detecting defects in object-oriented designs: using reading techniques to increase software quality" in Proceedings of the 14th Conference on Object-Oriented Programming, Systems, Languages, and Applications. ACM Press, pp. 47-56, 1999.

[5] H. A. Simon, Why should machines learn? R.S. Michalski, J.G. Carbonell, T.M. Mitchell (Eds.), Machine Learning, Tioga, Palo Alto, CA 1983 (Chapter 2)

[6] M. Kessentini, W. Kessentini, H. Sahraoui, M. Boukadoum, A. Ouni, Design Defects Detection and Correction by Example, 19th Int. Conf. on Program Comprehension (ICPC), pp. 81-90, 2011.

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

[8] N. Moha, Y.-G. Guéhéneuc, L. Duchien, A.-F. L Meur, DECOR: A method for the specification and detection of code and design smells. IEEE Trans. Softw. Eng.36, pp. 20-36, 2009.

[9] H. Liu, L. Yang, Z. Niu, Z. Ma, W. Shao, Facilitating software refactoring with appropriate resolution order of bad smells. In: Proc. of the ESEC/FSE '09, pp. 265-268, 2009.

[10] R. Marinescu, Detection strategies: metrics-based rules for detecting design flaws, Proceedings of the 20th International Conference on Software Maintenance, IEEE Computer Society Press, pp. 350-359, 2004.

[11] F. Khomh, S. Vaucher, Y.-G. Guéhéneuc, H. Sahraoui, A Bayesian approach for the detection of code and design smells. In Proc. of the ICQS'09, 2009.

[12] W. J. Brown, R. C. Malveau, W. H. Brown, H. W. M. III, and T. J. Mowbray. Anti Patterns: Refactoring Software,Architectures, and Projects in Crisis. John Wiley and Sons, 1st edition, 1998.

[13] K. Erni and C. Lewerentz: Applying design metrics to object-oriented frameworks, in Proc. IEEE Symp. Software Metrics, IEEE Computer Society Press, 1996.

[14] H. Alikacem and H. Sahraoui, Détection d'anomalies utilisant un langage de description de règle de qualité, in actes du 12e colloque LMO, 2006.

[15] S.C. Kothari, L. Bishop, J. Sauceda, G. Daugherty, A pattern-based framework for software anomaly detection. Softw. Qual. J.12(2), pp. 99–120, 2004.

[16] K. Dhambri, H. Sahraoui, P. Poulin, Visual detection of design anomalies. In: CSMR. IEEE, pp. 279-283, 2008.

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

[18] T. Mens, A survey of software refactoring, IEEE Transactions on Software Engineering, vol. 30, no. 2, pp. 126-139, 2004.

[19] IEEE Std. 1219-1998, "Standard for Software Maintenance", IEEE Computer Society Press, Los Alamitos, CA, 1998.

[20] M. Harman, and L. Tratt, Pareto optimal search based refactoring at the design level, In: Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'07), pp. 1106-1113, 2007.

[21] O. Seng, J. Stammel, and D. Burkhart, Search-based determination of refactorings for improving the class structure of object-oriented systems, In: Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'06), pp. 1909-1916, 2006

[22] M. O'Keeffe, and M. O. Cinnéide, Search-based Refactoring for Software Maintenance. J. of Systems and Software, 81(4), pp. 502–516, 2008.

[23] M. O’Keeffe, and M. O. Cinnéide, Search-based software maintenance. In: Proceedings of the 10th European Conference on Software Maintenance and Reengineering (CSMR), pp. 249– 260, 2006.

[24] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, A fast and elitist multiobjective genetic algorithm: NSGA-II, IEEE Trans. Evol. Comput., vol. 6, pp. 182-197, 2002.

[25] M. Fowler. Refactoring Catalog, http://www.refactoring.com/catalog/, (last access: 18 June 2014)

[26] A. Abran and H. Hguyenkim, “Measurement of the maintenance process from a demand-based perspective,” Journal of Software Maintenance: Research and Practice, vol. 5, no. 2, pp. 63-90, 1993.

[27] M. Kessentini, W. Kessentini, H. Sahraoui, M. Boukadoum, A. Ouni, Design Defects Detection and Correction by Example, 19th International Conference on Program Comprehension (ICPC), pp. 81-90, Kingston, Canada, 2011.

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

[29] A. Ouni, M. Kessentini, H. Sahraoui and M. S. Hamdi, Search-based Refactoring : Towards Semantics Preservation, in 28th IEEE International Conference on Software Maintenance (ICSM), September 23-30, Riva del Garda, Italy, 2012.

[30] J. E. Gaffney, Metrics in software quality assurance. In: Proc. of the ACM ’81 Conference, pp. 126–130. ACM, New York, 1981.

[31] G. Travassos, F. Shull, M. Fredericks, V.R. Basili, Detecting defects in object-oriented designs: using reading techniques to increase software quality, Proceedings of the 14th Conference on Object-Oriented Programming, Systems, Languages, and Applications, ACM Press, pp. 47-56, 1999.

[32] O. Ciupke, Automatic detection of design problems in object-oriented reengineering, D. Firesmith (Ed.), Proceeding of 30th Conference on Technology of Object-Oriented Languages and Systems, IEEE Computer Society Press, pp. 18-32, 1999.

[33] M.J. Munro, Product Metrics for Automatic Identification of “Bad Smell” Design Problems in Java Source-Code,”Proc. 11th Internatinal Software Metrics Symp., F. Lanubile and C. Seaman, eds., 2005.

[34] G. Langelier, H.A. Sahraoui, P. Poulin, visualization-based analysis of quality for large-scale software systems, T. Ellman, A. Zisma (Eds.), Proceedings of the 20th International Conference on Automated Software Engineering, ACM Press, 2005.

[35] M. Kessentini, S. Vaucher, H. Sahraoui, Deviance from Perfection is a Better Criterion than Closeness to Evil when Identifying Risky Code, 25th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2010.

[36] C. Catal and B. Diri, Investigating the effect of dataset size, metrics sets, and feature selection techniques on software fault prediction problem, Information Sciences, Elsevier, vol. 179, no. 8, pp. 1040-1058, 2009.

[37] L. Erlikh, “Leveraging legacy system dollars for e-business,”IT Professional, vol. 02, no. 3, pp. 17-23, 2000.

[38] S. Hassaine, F. Khomh, Y. G. Guéhéneuc, S. Hamel, IDS: An Immune-Inspired Approach for the Detection of Software Design Smells. 7th International Conference on the Quality of Information and Communications Technology (QUATIC), pp 343-348, 2010.

[39] M. Salehie, S. Li, L. Tahvildari, A Metric-Based Heuristic Framework to Detect Object-Oriented Design Flaws, in Pro-ceedings of the 14th IEEE International Conference on Pro-gram Comprehension (ICPC’06), 2006.

[40] H. Sahraoui, R. Godin, T. Miceli, Can Metrics Help to Bridge the Gap Between the Improvement of OO Design Quality and Its Automation?, In Proc. of the International Conference on Software Maintenance (ICSM’00), 2000.

[41] B. Du Bois, S. Demeyer, and J. Verelst, “Refactoring—Improving Coupling and Cohesion of Existing Code,” Proc. 11th Working Conf. Reverse Eng. pp. 144-151, 2004.

[42] N. Moha, A. Hacene, P. Valtchev, and Y-G. Guéhéneuc. Refactorings of Design Defects using Relational Concept Analysis. In Raoul Medina and Sergei Obiedkov, editors. Proceedings of the 4th International Conference on Formal Concept Analysis (ICFCA 2008), February 2008.

[43] L. Tahvildari, K. Kontogiannis, A metric-based approach to enhance design quality through meta-pattern transformation. In: Proceedings of the 7st European Conference on Software Maintenance and Reengineering , Benevento, Italy, pp. 183-192, 2003.

[44] P. Joshi, R.K. Joshi, Concept analysis for class cohesion. In: Proceedings of the 13th European Conference on Software Maintenance and Reengineering, Kaiserslautern, Germany, pp. 237-240, 2009.

[45] G. Bavota, A. De Lucia, R. Oliveto, Identifying Extract Class refactoring opportunities using structural and semantic cohesion measures, The Journal of Systems and Software 84 (2011) pp. 397-414, 2011.

[46] M. Kim, M. Gee, A. Loh, and N. Rachatasumrit. Ref-finder: a refactoring reconstruction tool based on logic query templates. In Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engi-neering, FSE ’10, pages 371-372, New York, NY, USA, 2010.

[47] T. Baar, S. Markovic, A Graphical Approach to Prove the Semantic Preservation of UML/OCL Refactoring Rules,

[48] F. Logozzo, A. Cortesi, Semantic Hierarchy Refactoring by Abstract Interpretation, E.A. Emerson and K.S. Namjoshi (Eds.), VMCAI 2006, LNCS 3855, pp. 313-331, 2006.

[49] F. Qayum, R. Heckel, Local search-based refactoring as graph transformation. Proceedings of 1st International Symposium on Search Based Software Engineering; pp. 43-46, 2009.

[50] Y. Kataoka, M. D. Ernst, W. G. Griswold, and D. Notkin, Automated support for program refactoring using invariants, in Int. Conf. on Software Maintenance (ICSM), pp. 736–743, 2001.

[51] L. Tahvildari and K. Kontogiannis. Improving design quality using meta-pattern transformations: a metric-based approach. Journal of Software Maintenance, 16 (4-5), pp. 331-361, 2004.

[52] D. E. Goldberg, Genetic Algorithms in Search, Optimization and Machine Learning. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1989.

[53] http://www.eclipse.org/

[54] https://netbeans.org/

[55] A. Abran and H. Hguyenkim, Measurement of the Maintenance Process from a Demand-Based Perspective, Journal of Software Maintenance: Research and Practice, Vol 5, no 2, 1993.

[56] P. Weißgerber, and S. Diehl, Are refactorings less error-prone than other changes? Proceedings of the 2006 international workshop on Mining software repositories, 2006.

[57] J. Ratzinger, T. Sigmund, H. Gall, On the relation of refactorings and software defect prediction, Proceedings of the 2008 international workshop on Mining software repositories, 2008.

[58] A. Ouni, M. Kessentini and H. Sahraoui, Search-based Refactoring Using Recorded Code Changes, in the 17th European Conference on Software Maintenance and Reengineering (CSMR), march 5-8, Genova, Italy, 2013.

[59] D. Fatiregun, M. Harman, and R. Hierons. Evolving transformation sequences using genetic algorithms. In SCAM 04, pages 65–74, Los Alamitos, California, USA, IEEE Computer Society Press, 2004.

[60] D. Fatiregun, M. Harman, and R. Hierons. Search-based amorphous slicing. In WCRE 05, pp. 3-12, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA, 2005.

[61] F. E. B. Otero, C. G. Johnson, A. A. Freitas, , and S. J. Thompson. Refactoring in automatically generated programs. International Symposium on Search Based Software Engineering, 2010.

[62] W. F. Opdyke, R. E. Johnson, Refactoring: An Aid in Designing Application Frameworks and Evolving Object-Oriented Systems. In Proceedings of the Symposium on Object Oriented Programming Emphasizing Practical Applications (SOOPPA), September 1990.

[63] S. M. Olbrich, D. S. Cruzes, D. I. K. Sjoberg, Are all code smells harmful? A study of God Classes and Brain Classes in the evolution of three open source systems. Software Maintenance, ICSM 2010, pp. 1-10, Timisoara, 2010.

[64] S. Olbrich, D. Cruzes, V. R. Basili, N. Zazworka, The evolution and impact of code smells: A case study of two open source systems, pp. 390-400, ESEM 2009,

[65] J. Ratzinger, T. Sigmund, P. Vorburger, and H. Gall. Mining software evolution to predict refactoring. In Proceedings of the First International Symposium on Empirical Software Engineering and Measurement, pp. 354-363, 2007.

[66] Q. D. Soetens, J. Perez, S. Demeyer, An Initial Investigation into Change-Based Reconstruction of Floss-Refactorings, 2013 29th IEEE International Conference on Software Maintenance (ICSM 2013), pp.384-387, 2013.

[67] A. F. Yamashita, L. Moonen, To what extent can maintenance problems be predicted by code smell detection? - An empirical study. Information & Software Technology 55(12), pp. 2223-2242, 2013.

[68] A. F. Yamashita, L. Moonen, Do developers care about code smells? An exploratory survey. WCRE: 242-25, 2013.

[69] A. F. Yamashita, L. Moonen, Do code smells reflect important maintainability aspects? International Conference on Software Maintenance, pp. 306-315, 2012.

[70] A. J. Riel, Object-Oriented Design Heuristics, 1st ed. Boston, MA, USA: Addison-Wesley, 1996.

[71] E. Van Emden and L. Moonen. 2002. Java Quality Assurance by Detecting Code Smells. In Proceedings of the Ninth Working Conference on Reverse Engineering (WCRE'02), IEEE Computer Society, Washington, DC, USA, 2002.

[72] M.V. Mäntylä, J. Vanhanen, C. Lassenius, A taxonomy and an initial empirical study of bad smells in code, IEEE International Conference on Software Maintenance (ICSM03), pp. 381–384. 2003.

[73] P. Coad and E. Yourdon, Object-Oriented Design. Prentice Hall, 1991.

[74] A. Monden, D. Nakae, T. Kamiya, S. Sato, K. Matsumoto, Software quality analysis by code clones in industrial legacy software, in IEEE Symposium on Software Metrics, pp. 87-94, 2002.

[75] W. Li, R. Shatnawi, An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution, Journal of Systems and Software 80, pp. 1120–1128, 2007.

[76] D. I. K. Sjøberg, A. F. Yamashita, B. C. D. Anda, A. Mockus, T. Dybå, Quantifying the Effect of Code Smells on Maintenance Effort. IEEE Trans. Software Eng. 39(8), pp. 1144-1156, 2013.

[77] I. Deligiannis, M. Shepperd, M. Roumeliotis, I. Stamelos, An empirical investigation of an object-oriented design heuristic for maintainability, Journal of Systems and Software 65, pp. 127-139, 2003.

[78] I. Deligiannis, I. Stamelos, L. Angelis, M. Roumeliotis, M. Shepperd, A controlled experiment investigation of an object-oriented design heuristic for maintainability, Journal of Systems and Software 72, pp. 129-143, 2004.

[79] B. Anda, Assessing software system maintainability using structural measures and expert assessments. In : Software Maintenance, 2007. ICSM 2007. IEEE International Conference on. IEEE, pp. 204-213, 2007.

[80] T. Mens and S. Demeyer, eds., Software Evolution, Springer, 2008.

[81] iPlasma : http://loose.upt.ro/iplasma/index.html

[82] Infusion hydrogen: Design flaw detection tool. Available at: http://www.intooitus.com/products/infusion, 2012.

[83] N. Tsantalis, T. Chaikalis, A. Chatzigeorgiou, JDeodorant: Identification and removal of type-checking bad smells. In Proceedings of CSMR2008, pp 329–331, 2008.

[84] R. Wirfs-Brock and A. McKean, Object Design: Roles, Responsibilities and Collaborations. Addison-Wesley Professional, 2002.

[85] E. R. Murphy-Hill, C. Parnin, A. P. Black, How We Refactor, and How We Know It. IEEE Trans. Software Eng. 38(1), pp. 5-18, 2012.

[86] E. R. Murphy-Hill, A. P. Black, Programmer-Friendly Refactoring Errors. IEEE Trans. Software Eng. 38(6), pp. 1417-1431, 2012.

[87] E. R. Murphy-Hill, A. P. Black, Breaking the barriers to successful refactoring: observations and tools for extract method. ICSE, pp. 421-430, 2008.

[88] X. Ge, E. Murphy-Hill, Manual Refactoring Changes with Automated Refactoring Validation. In Proceedings of the International Conference on Software Engineering (ICSE), Hyderabad, India, 2014.

[89] M. Harman, The current state and future of search based software engineering, in Future of Software Engineering 2007, L. Briand and A. Wolf, Eds. Los Alamitos, California, USA: IEEE Computer Society Press, pp. 342-357, 2007.

[90] M. Harman, S. A. Mansouri, Y. Zhang. Search-based software engineering: Trends, techniques and applications. ACM Comput. Surv. pp. 45-61, 2012.

[91] M. Harman, B. F. Jones, Search-based software engineering. Information and Software Technology, 43(14), pp. 833-839, 2001.

[92] C. Le Goues, T. Nguyen, S. Forrest, W. Weimer, Genprog: A generic method for automatic software repair. IEEE Trans. Software Eng. 38(1), pp. 54-72, 2012.

[93] Y. Zhang, A. Finkelstein, M. Andharman, Search based requirements optimisation: Existing work and challenges. InProceedings of the 14th International Working Conference, Requirements Engineering: Foundation for Software Quality (RefsQ'08).Lecture Notes in Computer Science, vol. 5025. Springer, pp. 88-94. 2008.

[94] P. McMinn, Search-Based software test data generation: A survey. Softw. Test. Verif. Reliab, 2004.

[95] E. Alba, F. Andchicano, Management of software projects with GAs. In Proceedings of the 6th Metaheuristics International Conference (MIC'05), Elsevier, pp. 13-18, 2005.

[96] G. Canfora, M. Di Penta, R. Esposito, M. L. Andvillani, An approach for QoS-aware service composition based on genetic algorithms. In Proceedings of the Conference on Genetic and Evolutionary Computation (GECCO'05), ACM, New York, pp. 1069-1075. 2005.

[97] F. Glover and M. Laguna, Tabu Search. Boston, MA: Kluwer Academic Publishers, 1997.

[98] S. Kirkpatrick, C.D. Gelatt Jr., and M.P. Vecchi, "Optimization by Simulated Annealing,"Science,vol. 220, no. 4598, pp. 671-680, 1983.

[99] DE. Goldberg, Genetic algorithms in search, optimization, and machine learning. Addison-Wesley, Reading, MA, USA,1989.

[100] J. Kennedy and R. C. Eberhart, Particle Swarm Optimization, in Proc. IEEE Int. Conf. Neural Networks, Perth, Australia, pp. 1942-1948, 1995.

[101] E. Murphy-Hill, A. P. Black, An interactive ambient visualization for code smells. In Proceedings of the 5th international symposium on Software visualization, pp. 5-14, ACM, 2010.

[102] F. Palomba, G. Bavota, M. Di Penta, Detecting bad smells in source code using change history information. In : IEEE/ACM 28th International Conference on Automated Software Engineering (ASE), 2013, pp. 268-278, 2013.

[103] A. Maiga, N. Ali, N. Bhattacharya, A. Sabane, Y.-G. Guéhéneuc, E. Aïmeur, a SVM-based incremental anti-pattern detection approach. In : Reverse Engineering (WCRE), 2012 19th Working Conference on. IEEE, pp. 466-475, 2012.

[104] A. Maiga, N. Ali, N. Bhattacharya, A. Sabane, Y.-G. Guéhéneuc, G. Antoniol, E. Aïmeur. Support vector machines for anti-pattern detection. In Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2012. IEEE, pp. 278-281, 2012.

[105] F. Rahma, N. Bouassida, and H. Ben Abdallah. "A metric-based approach for anti-pattern detection in uml designs." Computer and Information Science 2011. Springer Berlin Heidelberg, pp. 17-33, 2011.

[106] S. Dimitrios, A. Cerone, and S. Fenz, Enhancing ontology-based antipattern detection using Bayesian networks. Expert Systems with Applications 39.10, pp. 9041-9053, 2012.

[107] C. Marinescu, R. Marinescu, P. Mihancea, D. Ratiu, and R. Wettel. iplasma: An integrated platform for quality assessment of objectoriented design. In Proceedings of 21st International Conference on Software Maintenance (ICSM 2005), Tools Section, 2005.

[108] PMD http://pmd.sourceforge.net/

[109] CS-CheckStyle: http://checkstyle.sourceforge.net/index.html.

[110] InCode : http://www.intooitus.com/inCode.html

[111] FA. Fontana, E. Mariani, A. Morniroli, R. Sormani, A. Tonello, An Experience Report on Using Code Smells Detection technologies, Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference, pp.450-457, 2011.

[112] F. Arcelli, V. Ferme, M. Zanoni, and A. Yamashita. Filtering and Prioritising Code Smells Detection. In Submitted to conference, 2014.

[113] A. Yamashita, Assessing the capability of code smells to explain maintenance problems: an empirical study combining quantitative and qualitative data. Empirical Software Engineering, 1-33, 2013.

[114] R. Marinescu, Assessing technical debt by identifying design flaws in software systems, IBM Journal of Research and Development, vol. 56, no. 5, pp. 9:1–9:13, 2012.

[115] D. Raiu, S. Ducasse, T. Gîrba, and R. Marinescu, Using history information to improve design flaws detection, in Proc. Conf. Softw. Maintenance Reeng., vol. 8, pp. 223–232, 2004.

[116] D.-E. Ekwa, and M. P. Robillard,. Clonetracker: tool support for code clone management. Proceedings of the 30th international conference on Software engineering. ACM, 2008.

[117] M. F. Zibran, K. R. Chanchal, Towards flexible code clone detection, management, and refactoring in IDE. Proceedings of the 5th International Workshop on Software Clones. ACM, 2011.

[118] F. Khomh, M. D. Penta, and Y.-G. Gueheneuc, "An exploratory study of the impact of code smells on software change-proneness," Proceedings of WCRE 2009 (16th IEEE Working Conference on Reverse Engineering), pp. 75–84, 2009.

[119] M. F. Zibran, K. R. Chanchal, A constraint programming approach to conflict-aware optimal scheduling of prioritized code clone refactoring. Source Code Analysis and Manipulation (SCAM), 2011 11th IEEE International Working Conference on. IEEE, 2011.

[120] D. Hou, P. Jablonski, and F. Jacob. CnP: Towards an environment for the proactive management of copy-and-paste programming. In ICPC, pp. 238–242, 2009.

[121] R. Arcoverde, E. Guimaraes, L. Macia, A. Garcia, Y. Cai, Prioritization of Code Anomalies Based on Architecture Sensitiveness. Software Engineering (SBES), 2013 27th Brazilian Symposium on. IEEE, 2013.

[122] Code Metrics for Microsoft Visual Studio, http://msdn.microsoft.com/en-us/library/bb385911.aspx, (last access: 18 June 2014)

[123] M. F. Zibran, K. R. Chanchal, Conflict-aware optimal scheduling of prioritised code clone refactoring. IET software 7.3, pp. 167-186, 2013.

[124] M. Ó Cinnéide, Automated Application of Design Patterns: A Refactoring Approach. PhD thesis, University of Dublin, Trinity College, 2001.

[125] A. Ajouli, J. Cohen, J. Royer, Transformations between Composite and Visitor Implementations in Java. EUROMICRO-SEAA, pp. 25-32, 2013.

[126] A. Jensen and B. Cheng. On the use of genetic programming for automated refactoring and the introduction of design patterns. In Proceedings of GECCO. ACM, 2010.

[127] M. Ó Cinnéide, P. Nixon: A Methodology for the Automated Introduction of Design Patterns. International Confrence on Software Maintenance, 1999.

[128] D. Roberts, J. Brant, and R. Johnson. A refactoring tool for smalltalk. Theory and Practice of Object Systems, 3(4), pp. 253–263, 1997.

[129] R. C. Martin. Agile Software Development, Principles, Patterns and Practice. Prentice Hall, 2002.

[130] E. R. Murphy-Hill and A. P. Black, Refactoring tools: Fitness for purpose, IEEE Software, vol. 25, no. 5, pp. 38-44, 2008.

[131] X. Ge, E. R. Murphy-Hill, BeneFactor: a flexible refactoring tool for eclipse. OOPSLA Companion, pp. 19-20, 2011.

[132] X. Ge, E. Murphy-Hill, Manual Refactoring Changes with Automated Refactoring Validation. In Proceedings of the Int. Conf. on Soft. Eng. (ICSE), 2014.

[133] T. Mens, N. Van Eetvelde, S. Demeyer, and D. Janssens. Journal of software maintenance and evolution: Research and practice. J. Softw. Maint. Evol., 17(4), pp. 247–276, 2005.

[134] G. Bavota, A. De Lucia, A. Marcus, and R. Oliveto. A two-step technique for extract class refactoring. Int. conference on Automated software engineering, pages 151–154, 2010.

[135] D. Silva, R. Terra, M. T. Valente, Recommending Automated Extract Method Refactorings, Internation Conference on Program Comprehension ICPC, 2014.

[136] M. Robillard, R. Walker, and T. Zimmermann, Recommendation systems for software engineering, IEEE Software, vol. 27, no. 4, pp. 80–86, 2010.

[137] R. Holmes, R. Walker, and G. Murphy, Approximate structural context matching: An approach to recommend relevant examples, IEEE Transactions on Software Engineering, vol. 32, no. 12, pp. 952–970, 2006.

[138] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl, Mining version histories to guide software changes," IEEE Transactions on Software Engineering, vol. 31, no. 6, pp. 429–445, 2005.

[139] B. Dagenais and M. P. Robillard, Recommending adaptive changes for framework evolution, in 30th International Conference on Software Engineering (ICSE), pp. 481–490, 2008.

[140] N. Tsantalis and A. Chatzigeorgiou, Identification of move method refactoring opportunities, IEEE Transactions on Software Engineering, vol. 99, pp. 347–367, 2009.

[141] http://refactoring.com/tools.html

[142] Y. Ye and G. Fischer, Reuse-Conducive Development Environments, Automated Software Eng., vol. 12, no. 2, pp. 199–235, 2005.

[143] A. Ankolekar et al., Supporting Online Problem-Solving Communities with the Semantic Web, Proc. Int'l Conf. World Wide Web, ACM Press, pp. 575–584, 2006.

[144] A. Mockus and J.D. Herbsleb, "Expertise Browser: A Quantitative Approach to Identifying Expertise," Proc. Int'l Conf. Software Eng. (ICSE 02), IEEE CS Press, 2002, pp. 503–512.

[145] S. Thummalapenta and T. Xie, PARSEWeb: A Programming Assistant for Reusing Open Source Code on the Web, Proc. IEEE/ACM Int'l Conf. Automated Software Eng. (ASE 07), ACM Press, pp. 204–213, 2007.

[146] R. Terra, M. T. Valente, K. Czarnecki, R. S. Bigonha, Recommending refactorings to reverse software architecture erosion. In 16th European Conference on Software Maintenance and Reengineering (CSMR), pp. 335-340, IEEE, 2012.

[147] G. Bavota, R. Oliveto, M. Gethers, D. Poshyvanyk, and A. D. Lucia, Methodbook: Recommending move method refactorings via relational topic models, IEEE Transactions on Software Engineering, pp. 1–26, 2014.

[148] V. Sales, R. Terra, L. F. Miranda, and M. T. Valente, Recommending move method refactorings using dependency sets, in 20th Working Conference on Reverse Engineering (WCRE), pp. 232–241, 2013.

[149] A. Thies, R. Christian, Recommending rename refactorings, Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering, 2010.

[150] G. Bavota, A. De Lucia, R. Oliveto, Identifying extract class refactoring opportunities using structural and semantic cohesion measures, Journal of Systems and Software, vol. 84, pp. 397–414, 2011.

[151] G. Bavota, A. De Lucia, A. Marcus, R. Oliveto, Automating extract class refactoring: an improved method and its evaluation. Empirical Software Engineering, pp. 1-48, 2013.

[152] H. Kagdi, M.L. Collard, J.I. Maletic, A Survey and Taxonomy of Approaches for Mining Software Repositories in the Context of Software Evolution, in the Journal of Software Maintenance and Evolution: Research and Practice (JSME), Vol. 19, No. 2, pp. 77-131, 2007.

[153] A. E. Hassan, The road ahead for mining software repositories. Frontiers of Software Maintenance, 2008. FoSM 2008.. IEEE, 2008.

[154] J. Ratzinger, T. Sigmund, P. Vorburger, and H. Gall. Mining software evolution to predict refactoring. In Proceedings of the First International Symposium on Empirical Software Engineering and Measurement, pp. 354-363, 2007.

[155] Q.D. Soetens, J. Perez, S. Demeyer, An Initial Investigation into Change-Based Reconstruction of Floss-Refactorings, 2013 29th IEEE International Conference on Software Maintenance (ICSM 2013), pp.384-387, 2013.

[156] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. In Proc. 26th International Conference on Software Engineering (ICSE), Edinburgh, Scotland, 2004.

[157] T. Gîrba, S. Ducasse, A. Kuhn, R. Marinescu, and R. Daniel. Using Concept Analysis to Detect Co-Change Patterns. In Proceedings of International Workshop on Principles of Software Evolution (IWPSE), 2007.

[158] A. Hassan and R. Holt. Predicting change propagation in software systems. In Proceedings 20th Int. Conference on Software Maintenance (ICSM'04), pp. 284–293, 2004.

[159] H. Gall, K. Hajek, and M. Jazayeri. Detection of logical coupling based on product release history. Int. Proc. of Conf. on Soft. Maintenance (ICSM), pages 190–198, Los Alamitos CA, 1998.

[160] T.T. Ying, G. C. Murphy, R. Ng, and M. C. Chu-Carroll. Predicting source code changes by mining change history. IEEE Transactions on Soft. Eng. (TSE), Vol 30, No. 9, 2004.

[161] D. Beyer, and A. Noack. Clustering Software Artifacts Based on Frequent Common Changes. Proceedings of the 13th International Workshop on Program Comprehension, 2005.

[162] S. Hayashi, M. Saeki, M. Kurihara. Supporting refactoring activities using histories of program modification. IEICE transactions on information and systems 89.4, pp. 1403-1412, 2006.

[163] A. Y. S. Lam, V. O. K. Li, Chemical-Reaction-Inspired Metaheuristic for Optimization. IEEE Trans. Evolutionary Computation 14(3), pp. 381-399, 2010.

[164] M. Kessentini, A. Ouni, P. Langer, M. Wimmer and S. Bechikh, Search-based Metamodel Matching with Structural and Syntactic Measures, Journal of Systems and Software, 2014.

[165] H. Kilic, E. Koc, and I. Cereci. Search-based parallel refactoring using population-based direct approaches. In Proceedings of the Third international Conference on Search Based Software Engineering, SSBSE'11, pages 271–272, 2011.

[166] M. Ó Cinnéide, L. Tratt, M. Harman, S. Counsell, and I. H. Moghadam, Experimental Assessment of Software Metrics Using Automated Refactoring, Proc. Empirical Software Engineering and Management (ESEM), pp. 49-58, 2012.

[167] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, "Experimentation in Software engineering -- An introduction," Kluwer Academic Publishers, ISBN 0-7923-8682-5, 2000.

[168] G. Karafotias, M. Hoogendoorn, and A. E. Eiben, "Why parameter control mechanisms should be benchmarked against random variation," IEEE Congress on Evolutionary Computation, pp. 349–355, 2013.

[169] A. E. Eiben and S. K. Smit, "Parameter tuning for configuring and analyzing evolutionary algorithms," Swarm and Evolutionary Computation, vol. 1, no. 1, pp. 19–31, 2011.

[170] W. H. Kruskal and W. A. Wallis, "Use of ranks in one-criterion variance analysis," Journal of the American Statistical Association, vol. 47, no. 260, pp. 583–621, 1952.

[171] J. R. Koza. Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press, Cambridge, MA, USA, 1992.

[172] R. Davis, B. Buchanan, and E.H. Shortcliffe, Production Rules as a Representation for a Knowledge-base Consultation Program, Artificial Intelligence 8, 15-45, 1977.

[173] M. Zhang, T. Hall, N. Baddoo, Code Bad Smells: a review of current knowledge, In Journal of Software Maintenance and Evolution: Research and Practice 23.3, pp. 179–202, 2011.

[174] I. Sommerville, Software Engineering, 6th ed. Addison-Wesley, 2001.

[175] D. B. Roberts, Practical Analysis for Refactoring. PhD thesis, Department of Computer Science, University of Illinois, 1999.

[176] A. Y. S. Lam, V. O. K. Li, Chemical-Reaction-Inspired Metaheuristic for Optimization. IEEE Trans. Evolutionary Computation 14(3), pp. 381-399, 2010.

[177] A. Y. S. Lam, V. O. K. Li, J. J. Q. Yu, Real-Coded Chemical Reaction Optimization. IEEE Trans. Evolutionary Computation 16(3), pp. 339-353, 2012.

[178] J. J. Q. Yu, A. Y. S. Lam, V. O. K. Li, Real-coded chemical reaction optimization with different perturbation functions. IEEE Congress on Evolutionary Computation, pp. 1-8., 2012.

[179] J. Xu, A. Y. S. Lam, V. O. K. Li, Chemical Reaction Optimization for Task Scheduling in Grid Computing. IEEE Trans. Parallel Distrib. Syst. 22(10), pp. 1624-1631, 2011.

[180] Y. Sun, A. Y. S. Lam, V. O. K. Li, J. Xu, J. J. Q. Yu, Chemical Reaction Optimization for the optimal power flow problem. IEEE Congress on Evolutionary Computation, pp. 1-8, 2012.

[181] A. Y. S. Lam, V. O. K. Li, Z. Wei, Chemical Reaction Optimization for the Fuzzy Rule learning problem. IEEE Congress on Evolutionary Computation, pp. 1-8, 2012.

[182] S. R. Chidamber, C. F. Kemerer, A metrics suite for object oriented design, IEEE Transactions on Software Engineering, Vol. 20, No. 6, pp. 476-493, 1994.

[183] G. Fraser, A. Arcuri, Handling test length bloat. Software Testing., Verification and Reliability. 23 (7). pp. 553-582, 2013.

[184] J. Cohen, Statistical power analysis for the behavioral sciences, 2nd ed. Lawrence Earlbaum Associates, 1988.

[185] R. Amaro, R. P. Chaves, P. Marrafa, S. Mendes, Enriching Wordnets with new Relations and with Event and Argument Structures. In 7th International Conference on Intelligent Text Processing and Computational Linguistics , pp. 28 - 40, Mexico City, Lecture Notes in Computer Science, Springer-Verlag, 2006.

[186] L. Tokuda, D. Batory, Evolving Object-Oriented Designs with Refactorings. Automated Software Engineering. 8, 1, 89-120, 2001.

[187] M. Schäfer, T. Ekman, O. de Moor, Sound and extensible renaming for Java. In Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications (OOPSLA '08). ACM, NY, USA, 277-294, 2008.

[188] G. Soares, R. Gheyi, T. Massoni, Automated Behavioral Testing of Refactoring Engines, Software Engineering, IEEE Transactions on , vol.39, no.2, pp.147-162, 2013.

[189] M. Schäfer, T. Ekman, O. de Moor, Challenge proposal: verification of refactorings. In Proceedings of the 3rd workshop on Programming languages meets program verification, 2009.

[190] R. Vallée-Rai, E. Gagnon, L. J. Hendren, P. Lam, P. Pominville, V. Sundaresan, Optimizing Java bytecode using the Soot framework: Is it feasible? in Int. Conf. on Compiler Construction, pp. 18–34, 2000.

[191] C. M. Fonseca and P. J. Fleming, Genetic algorithms for multiobjective optimization: Formulation, discussion and generalization, in Proceedings of the Fifth International Conference on Genetic Algorithms,S. Forrest, Ed. San Mateo, CA: Morgan Kauffman, pp. 416–423, 1993.

[192] E. Zitzler, and L. Thiele, Multiobjective optimization using evolutionary algorithms-A comparative case study. In Parallel Problem Solving from Nature, pp.292–301, Springer, Germany, 1998.

[193] J. Bansiya and C. G. Davis. A hierarchical model for object-oriented design quality assessment. IEEE Trans. Softw. Engg., 28(1): 4–17, 2002.

[194] M. Kim, T. Zimmermann, and N. Nagappan, A field study of refactoring challenges and benefits, in 20th International Symposium on the Foundations of Software Engineering (FSE), pp. 50:1–50:11, 2012.

[195] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design patterns: elements of reusable object-oriented software. Addison-Wesley Reading, MA, 1995.

[196] D. Heuzeroth, T. Holl, G. Högström, W. Löwe, Automatic Design Pattern Detection, International Workshop on Program Comprehension (IWPC) 2003.

[197] A. Ajouli, J. Cohen, J. Royer, Transformations between Composite and Visitor Implementations in Java. EUROMICRO-SEAA, pp. 25-32, 2013.

[198] A. Ajouli, Vues et Transformations de Programmes pour la Modularité des Évolutions, Ph.D. dissertation, University of Nantes Angers Le Mans, 2013.

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

[200] E. Zitzler, L. Thiele, M. Laumanns, C. M. Fonseca, and da V. G. Fonseca. Performance assessment of multiobjective optimizers: an analysis and review. IEEE Tran. on Evolutionary Comp., 7:117–132, 2003.

[201] K. Deb, Multi-Objective Optimization Using Evolutionary Algorithms. Wiley. 2009.

[202] F. G. Freitas and J. T. Souza, Ten years of search based software engineering: A bibliometric analysis, in 3rd International Symposium on Search based Software Engineering (SSBSE 2011), pp. 18–32, 2011.

[203] A. Arcuri and G. Fraser, Parameter tuning or default values? An empirical investigation in search-based software engineering, Empirical Software Engineering 18(3), Springer, 2013.

[204] R. Marinescu, Measurement and Quality in Object Oriented Design. Doctoral Thesis. Politehnica University of Timisoara, 2002.

[205] F. Tip, J. Palsberg, Scalable Propagation-based Call Graph Construction Algorithms. In Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications, pages 281–293, 2000.

Appendix A: Definitions of the used quality attributes and metrics

In this Appendix, we present the definitions of the quality attributes and metrics used in this thesis.

A.1 Quality attributes

We consider the following quality attributes according to Bansiya and Davis' QMOOD quality model [193]:

- **Reusability:** The degree to which a software module or other work product can be used in more than one computer program or software system.
- **Flexibility:** The ease with which a system or component can be modified for use in applications or environments other than those for which it was specifically designed.
- **Understandability:** The properties of designs that enable it to be easily learned and comprehended. This directly relates to the complexity of design structure.
- **Functionality:** The responsibilities assigned to the classes of a design, which are made available by the incorporation of a new requirements in the design.
- **Extendibility:** Refers to the presence and usage of properties in an existing design that allow for the incorporation of new requirements in the design.
- **Effectiveness:** The degree to which a design is able to achieve the desired functionality and behavior using OO design concepts and techniques.

A.2 Metrics

We consider the following metrics [193] [204] [182]:

- **Design Size in Classes (DSC):** Counts the total number of classes in the design excluding imported library classes.
- **Number Of Hierarchies (NOH)**: Counts the number of class hierarchies in the design.

- **Average Number of Ancestors (ANA):** Signifies the average number of classes from which each class inherits information.
- **Data Access Metric (DAM):** Counts the ratio of the number of private (protected) attributes to the total number of attributes declared in the class.
- **Direct Class Coupling (DCC)**: Counts of the number of different classes that a class is directly related to. The metric includes classes that are directly related by attribute declarations and message passing (parameters) in methods.
- **Cohesion Among Methods of Class (CAM):** Computes the relatedness among methods of a class, computed using the summation of the intersection of parameters of a method with the maximum independent set of all parameter types in the class.
- **Measure Of Aggregation (MOA)**: Counts of the number of data declarations whose types are user-defined classes.
- **Measure of Functional Abstraction (MFA):** Counts the ratio of the number of methods inherited by a class to the number of methods accessible by member methods of the class.
- **Number of Polymorphic Methods (NOP):** Counts the number of the methods that can exhibit polymorphic behaviour. Interpreted as the sum over all classes, where a method can exhibit polymorphic behaviour if it is overridden by one or more descendent classes.
- **Class Interface Size (CIS):** Counts the number of public methods in a class. Interpreted as the average over all classes in a design.
- **Number Of Methods (NOM):** Counts of all the methods defined in a class.
- **Number of Fields (NOF):** Measures the number of fields of the classes.
- **Coupling Between Objects (CBO):** Counts the number of other classes to which a class is coupled.
- **Number Of Attributes (NOA):** Counts the number of attributes in a class.
- **Number Of Public Attributes (NOPA):** Counts the number of public attributes in a class.
- **Number Of Private Attributes (NPA):** Counts the number of private attributes in a

class.

- **Number Of Accessor Methods (NOAM):** Counts the number of getter and setter methods in a class.
- **Access Of Foreign Data (AOFD):** Counts the number of attributes from unrelated classes that are accessed directly or by invoking accessor methods.
- **Tight Class Cohesion (TCC):** Counts the relative number of method pairs of a class that access in common at least one attribute of the measured class.
- **Weight Of Class (WOC):** Counts the number of non-accessor methods in a class divided by the total number of members of the interface.
- **Weighted Method Count (WMC):** Represents the sum of the statical complexity of all methods of a class.
- **Lines Of Code (LOC):** Counts the number of lines of code in a class or method.
- **Changing Methods (CM):** Counts the number of distinct methods that call the measured method.

Appendix B: Definitions of the used refactoring operations

This Appendix presents the definitions of the refactoring operations used in this thesis.

B.1 Refactoring operations

- **Move Method:** Moves a method from a source class to a target class in another hierarchy. This refactoring can be applied when classes have too much behavior or when classes are collaborating too much and are too highly coupled.
- **Move Field:** Moves a field from a source class to a target class. This refactoring can be applied when a field is used by another class more than the class on which it is defined.
- **Extract Class:** Split a class into two classes by moving some methods and fields to a new class. This refactoring can be applied when one class doing work that should be done by two.
- **Incline Class:** Merges two classes into one class by moving all features to the second class and delete it. This refactoring can be applied when a class isn't doing very much.
- **Extract Interface:** Several clients use the same subset of a class's interface, or two classes have part of their interfaces in common.
- **Extract Superclass:** Creates a superclass and move the common features to the superclass. This refactoring can be applied when two classes or more share similar features.
- **Extract Subclass:** Creates a subclass and move the some features to the subclass. This refactoring can be applied when a class has features that are used only in some instances.
- **Push Down Field:** Moves a field from some class to those subclasses that require it.

This refactoring can be aplied to simplify the design by reducing the number of classes that have access to the field.

- **Pull Up Field:** Moves a field from some class(es) to the immediate superclass. This refactoring can be applied to eliminate duplicate field declarations in sibling classes.
- **Push Down Method:** Moves a method from some class to those subclasses that require it. This refactoring can be applied to simplify the design by reducing the size of class interfaces.
- **Pull Up Method:** Moves a method from some class(es) to their immediate superclass. This refactoring can be applied to help eliminate duplicate methods among sibling classes, and hence reduce code duplication in general.

Appendix C: Definitions of the used code-smells and design patterns

This Appendix presents the definitions of the code-smells and design patterns used in this thesis.

C.1 Code-smells

In this thesis, we primarily focus on the detection/correction the following code-smell types [1] [129] [84] [70] :

Code-smells	Description
Blob (also called God Class)	It is found in design fragments where one large class monopolizes the behavior of a system (or part of it), and the other classes primarily contain data. It is a large class that declares many fields and methods with a low cohesion and almost has no parents and no children.
Data Class	It contains only data and performs no processing on these data. It is typically composed of highly cohesive fields and accessors.
Spaghetti Code	It is a code with a complex and tangled control structure. This code-smell is characteristic of procedural thinking in object-oriented programming. Spaghetti Code is revealed by classes with no structure, declaring long methods with no parameters, and utilising global variables. Names of classes and methods may suggest procedural programming. Spaghetti Code does not exploit and prevents the use of object-orientation mechanisms, polymorphism and inheritance.
Functional Decomposition	It occurs when a class is designed with the intent of performing a single function. This is found in code produced by non-experienced object-oriented developers.
Schizophrenic Class	It occurs when a public interface of a class is large and used non-cohesively by client methods i.e., disjoint groups of client classes use disjoint fragments of the class interface in an exclusive fashion.
Shotgun Surgery	It is found when a method heavily uses attributes and data from one or more

	external classes, directly or via accessor operations. Furthermore, in accessing external data, the method is intensively using data from at least one external capsule.

We decided to focus our attention on these code-smells because they are among the most related to faults and/or change proneness and the most common in the literature.

C.2 Design patterns

In this thesis, we primarily focus on the following design patterns [195]:

Design Pattern	**Description**
Visitor	Represent an operation to be performed on the elements of an object structure. Visitor allows defining a new operation without changing the classes of the elements on which it operates. In essence, the visitor allows adding new virtual functions to a family of classes without modifying the classes themselves; instead, one creates a visitor class that implements all of the appropriate specializations of the virtual function.
Factory Method	The Factory Method is a creational pattern which uses factory methods to deal with the problem of creating objects without specifying the exact class of object that will be created. It Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.
Singleton	Restrict the instantiation of a class to one object. This is useful when exactly one object is needed to coordinate actions across the system. The concept is sometimes generalized to systems that operate more efficiently when only one object exists, or that restrict the instantiation to a certain number of objects.