Université de Montréal

Multi-Objective Optimization for Software Refactoring and Evolution

par

Ali Ouni

Département d’informatique et de recherche opérationnelle
Faculté des études supérieures et postdoctorales

Thèse présentée à la Faculté des études supérieures et postdoctorales
dans le cadre de l'examen predoc oral partie 3 en vue de l’obtention du grade de
Philosophiæ Doctor (Ph.D.)
en informatique

Décembre, 2012

Université de Montréal

Faculté des études supérieures et postdoctorales

Cette thèse intitulée:

Multi-Objective Optimization for Software Refactoring and Evolution

Présenté par :

Ali Ouni

a été évalué par un jury composé des personnes suivantes :

Jean Meunier, président-rapporteur

Houari Sahraoui, directeur de recherche

Marouane Kessentini, co-directeur

Bruno Dufour, membre du jury

Résumé

Des nombreuses études montrent que la maintenance des logiciels, traditionnellement définie comme des modifications apportées sur un système logiciel après sa livraison, consomme plus de 90% du coût total d'un logiciel. Ajouter de nouvelles fonctionnalités, corriger des bugs, ou modifier le code pour améliorer sa qualité sont les majeures activités de la maintenance. Pour faciliter ces activités, 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.

En général, le refactoring s'applique 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. Cette thèse porte sur l'automatisation de ces deux étapes de refactoring. Pour l'étape d'identification de solutions de refactoring, une approche se basant sur une recherche heuristique multi-objectif est proposée. 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 détectés. De plus, nous explorons d'autres objectifs à optimiser: l'effort requis pour appliquer la solution de refactoring, la préservation de la sémantique, et la similarité avec des refactorings appliqués dans le passé dans des contextes similaires. L'effort correspond au coût d'adaptation/modification du code pour appliquer les refactorings proposés. La préservation de la sémantique sert à assurer que le programme restructuré est sémantiquement équivalent à celui d'origine, et qu'il modélise correctement le domaine de la sémantique. En outre, pour améliorer l'automatisation de refactoring, nous utilisons l'historique de changement du code pour proposer des nouvelles solutions de refactoring dans des contextes similaires. Finalement, nous planifions d'étudier empiriquement la corrélation entre la correction des défauts de conception et l'introduction de nouveaux défauts, ou corriger implicitement d'autres défauts.

Mots-clés : Meta-heuristiques, Maintenance des logiciels, Défauts de Conception, Refactoring.

Abstract

Many studies reported that software maintenance, traditionally defined as any modification made on a software system after its delivery, consumes up to 90% of the total cost of a typical software project. Adding new functionalities, correcting bugs, and modifying the code to improve its quality are major parts of those costs. To ease these maintenance activities, one of the most-used techniques is the *refactoring* which improves design structure while preserving the external behavior.

In general, refactoring is performed through two main steps: 1) detection of code fragments corresponding to design defects that need to be improved/fixed and 2) identification of refactoring solutions to achieve this goal. Our research work addresses the automation of these two refactoring steps. For the refactoring identification step, a multi-objective search-based approach is also used. The process aims at finding the optimal sequence of refactoring operations that improve the software quality by minimizing the number of detected defects. In addition, we explore other objectives to optimize: the effort needed to apply refactorings, semantics preservation, and the similarity with good refactorings applied in the past to similar contexts. Hence, the effort corresponds to the code modification/adaptation score needed to apply the suggested refactoring solutions. On the other hand, the semantics preservation insures that the refactored program is semantically equivalent to the original one, and that it model correctly the domain-semantics. Indeed, we use knowledge from historical code changes to propose new refactoring solutions in similar contexts to improve the automation of refactoring. Indeed, we plan to conduct an empirical study to understand the correlation between correcting design defects and introducing new ones or fixing other defects implicitly.

Keywords : Search-based Software Engineering, Software Maintenance, Design Defects, Refactoring, Multi-Objective Optimisation.

Contents

List of tables

List of figures

Chapter 1: Introduction

1.1 Research context

Software maintenance is defined in IEEE Standard 1219 [19] as: The modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment. A startling 90% of the cost of a typical software system is incurred during the maintenance phase [53]. 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 design defects. Typically, design defects refer to design situations that adversely affect the development of software. As stated by Fenton and Pfleeger [2], design defects are unlikely to cause failures directly, but may do it indirectly. In general, they make a system difficult to change, which may often introduce bugs. The most well-known example of defects is the *blob* (found in designs where one large class monopolizes the behavior of a system, and other classes primarily encapsulate data). Thus, design defects should be identified and corrected by the development team as early as possible for maintainability and evolution considerations.

1.2 Problem statement

Detecting and fixing design defects is still, to some extent, a difficult, time-consuming, error-prone and manual process. As a consequence, automating design defects detection and correction is considered as a very challenging software engineering task.

The defect detection process 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 [32] [36] [41] consist of defining rules manually to identify key symptoms that characterize a defect using combinations of mainly quantitative, structural, and/or lexical information. However, in an exhaustive scenario, the number of possible defects to manually characterize with rules can be very large. In the other hand, other work [8] proposes to generate detection rules using formal definitions of defects. Although this partial automation of rule writing helps developers basing on symptom description, still, translating symptoms into rules is not obvious because there is no consensus of defining design defects based on their symptoms [12]. In the next subsection, we highlight the different design defects detection and correction challenges.

Once design defects are detected, they need to be fixed. To correct detected defects, one of the most-used techniques is the refactoring which improves design structure while preserving the external behavior [12]. William Opdyke [1] defines 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: reusability, maintainability, complexity, etc. [1]. 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 defect, many refactoring operations can be used to reduce the number of functionalities in a specific class, such as move methods and extract class. In the best of our knowledge, most of existing contributions suggest refactorings with the perspective of improving only some design/quality metrics. However, this could not be enough to obtain good refactoring solutions. Hence, to obtain good refactoring strategies other considerations have to be targeted such as the correction effort minimization, semantics preservation and also the similarity with good refactorings applied/recorded in the past in similar contexts.

Another observation is that existing work in refactoring focus on detecting and fixing design defects/bad smells as two separate steps. Thus, they do not try to understand the correlation between correcting design defects and introducing new design defects or fixing other defects implicitly.

In the next subsections we highlight the different problems addressed in this thesis that are mainly related to the design defects detection and correction.

1.2.1 Automating defects detection

Although there is a substantial amount of research work focusing on the detection of design defects [2] [1] [8] [12] [14] [36] [32], there are many open issues that need to be addressed when defining a detection strategy. In the following, we highlight these open issues.

Problem 1.1 Most of 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 defect.

Problem 1.2 There is no consensus on the definition of design defects based on their symptoms. Although when consensus exists, the same symptom could be associated to many defect types, which may compromise the precise identification of defect types.

Problem 1.3 The majority of existing detection methods does not provide an efficient manner to guide a manual inspection of candidate defects. Indeed, the process of manually defining detection rules, understanding the defect candidates, selecting the true positives, and correcting them is very long, fastidious, and not always profitable.

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

1.2.2 Automating defects correction

Detected defects can be fixed by applying the suitable refactoring operations [25]. Although there exists several works on software refactoring [20] [21] [48] [47] [43], until now, they do not provide an efficient and fully automated approach. Therefore, several open issues should be addressed when searching for refactoring solutions to improve the software quality (i.e., defects correction). Hence, several problems need to be addressed.

Problem 2.1 The majority of existing approaches [1] [38] [39] have manually defined "standard" refactorings for each defect type to remove its symptoms. However, it is difficult to define "standard" refactorings for each defect type and to generalize it because generally it depends on the programs in their context.

Problem 2.2 Removing defect symptoms does not mean that the defect is corrected, and, in the majority of cases, these “standard” solutions are enable to remove all symptoms for each defect.

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

Problem 2.4 In the majority of existing approaches [22] [47] [20] [21], the code quality can be improved without fixing design defects (i.e., improving some quality metrics does not guarantee that detected defects are fixed). Therefore, the link between defect detection (refactoring opportunities) and correction is not obvious. In other terms, we need to ensure whether the refactoring concretely corrects detected defects.

Problem 2.5 Existing approaches consider the refactoring (*i.e.,* the correction process) as local process by correcting defects (or improving quality) separately. In fact, a refactoring solution should not be specific to only one defect type and the correction process should consider the impact of refactoring. For example, moving methods to reduce the size/complexity of a class may increase the global coupling.

Problem 2.6 Reduce the refactoring effort (code adaptation/modification effort). Hence, improving software quality and reducing the effort are complementary and sometimes conflicting considerations. In some cases, correcting some defects corresponds to re-implementing a large portion of the system. In fact, a refactoring solution that fixes all defects is not necessarily the optimal one due to the high adaptation/modification effort needed.

Problem 2.7 In general, refactoring restructures a program to improve its structure without altering its behavior. However, it is challenging to preserve the domain semantics of a program when refactoring is decided/implemented automatically. Indeed, a program could be syntactically correct, and have the right behavior, but still model incorrectly 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.

1.2.3 Correlation between defects detection and correction

A general conclusion to be drawn from existing refactoring work is that most of the effort has been devoted to the definition of automatic and semi-automatic approaches supporting refactoring operations. However, in this proposal we suggest to analyze the relation between refactorings and design defects:

Problem 3.1. To the best of our knowledge there is no study aimed at thoroughly investigating whether a set of refactorings occurred in a software system induced new design defects; when fixing specific design defect type can correct implicitly other type of defects, and what kind of refactorings might induce more design defects than others.

All of these observations are at the origin of the work conducted in the scope of this thesis. In the next section we give an overview and our research direction to solve the above mentioned problems.

1.3 Proposed solutions

To address the above mentioned problems, we propose the following solutions which are organised into three principal contributions.

Contribution 1: Design defects detection

To automate the detection of design defects we propose a search-based approach [27] using genetic algorithm to automatically generate detection rules. To this end, knowledge from defect examples is used to generate detection rules. The detection process takes as inputs a base (*i.e.*, a set) of defect examples and takes as controlling parameters a set of quality metrics (the usefulness of these metrics were defined and discussed in the literature [29]). This step generates a set of design defects detection rules. Consequently, a solution to the defect detection problem is represented as a set of rules that best detect the defects presented on the base of examples with high score of precision and recall.

Contribution 2: Design defects correction

To correct the detected defects, we need to find the suitable refactoring solution. A refactoring solution is a combination of refactoring operations. To overcome the list of problems stated previously, we divide our contribution into four sub-steps.

Contribution 2.1. As first step, 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 defects using the detection rules. To this end, we use genetic algorithm (GA) [51] to find and suggest the best combination of refactoring operations from a large list of available refactorings [26].

Contribution 2.2. Then, we extend this contribution, to see the refactoring process as a multi-objective optimisation problem. We define a "good" refactoring solution as the combination of refactoring operations that should maximize as much as possible the number of corrected defects with minimal code modification/adaptation effort *(i.e.,* the cost of applying the refactoring sequence) [27]. The idea is to find the best compromise between maximizing quality and minimizing code adaptability effort. Thus, the aim is to reduce the complexity of the suggested refactorings by achieving meaningful reductions in terms of code modifications score. Indeed, this contribution does not correct defects separately since we consider the correction as a global process instead of local one. In addition, we don't need to define an exhaustive list of defects and to specify standard refactoring for each defect type.

Contribution 2.3. As a third step, it is challenging to integrate the semantics preservation especially that refactoring solutions are decided automatically using our search-based approach. In fact, a program could be syntactically correct, have the right behavior, but model incorrectly the domain semantics. One of the key issues we need to solve is to approximate the semantics similarity between code fragments, because the semantics is not clearly defined in the code. In this contribution, we propose a multi-objective optimization approach to find the best sequence of refactorings that maximizes quality improvements (*i.e.*, minimize the number of defects) and at the same time minimizes potential semantic errors. To this end, we use the non-dominated sorting genetic algorithm (NSGA-II) to find the best compromise between these two conflicting objectives [28]. In nutshell, this contribution shows how to automatically suggest refactoring solution in a way that preserves as much as possible the semantics of the original software system.

Contribution 2.4. To improve the automaton of refactoring, we start from the observation that recorded/historical code changes could be used to propose new refactoring solutions in similar contexts. In addition, this knowledge can be combined with structural and semantic information to improve the automation of refactoring. In this contribution, we propose a multi-objective optimization to find the compromise between all of the mentioned objectives. We define "optimal" refactoring solutions as the sequence of refactorings that minimizes the number of defects, minimizes the adaptation/modification effort, minimizes semantic errors and maximizes the use of refactorings applied in the past to similar contexts

To conclude, our contribution is built on the idea that a "good" refactoring solution should minimize, as much as possible, the number of detected defects, with minimal code modification effort, while preserving the domain semantics of the original program.

Contribution 3: Empirical Studies of Refactoring Impact on Design Defects

We are planning to investigate an empirical study, using an existing corpus [7] [27] [28], to understand the correlation between correcting design defects and introducing new design defects or fixing other defects implicitly.

Chapter 2: Related work

In this chapter, we brings the state of the art of existing work related to our research area concerned in this thesis, and we highlights the open research problems for the research community. Hence, the related work can be divided broadly into three research areas: (1) detection of design defects, (2) correction of design defects, and (3) the impact of refactoring. These topics are discussed below.

2.1 Detection of design defects

There has been much research effort focusing on the study of maintainability defects, also called anomalies [12], anti-patterns [1], or smells [2] in the literature. Such defects include, for example, the blob (very large class), spaghetti code (tangled control structure), functional decomposition (a class representing only a single function), etc.

Existing approaches for design defects 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.

2.1.1 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 defect type. Travassos et al. [30] have also proposed a manual approach for detecting defects 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 defects 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 design defects. In addition, in [31], 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 design defects.

The main disadvantage of exiting manual approaches is that they are ultimately a human-centric process that requires a great human effort and strong analysis and interpretation effort from software maintainers to find design fragments that corresponds to defects. In addition, these techniques are time-consuming, error-prone and depend on programs in their contexts. Another important issue is that locating defects 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.

2.1.2 Symptom-based detection

Moha et al. [8] started by describing defect symptoms using a domain-specific-language (DSL) for their approach called DECOR. They proposed a consistent vocabulary and DSL to specify antipatterns based on the review of existing work on design defects found in the literature. To describe defect 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. Indeed, this approach has been evaluated on only four well-known design defects: the Blob, functional decomposition, spaghetti code, and Swiss-army knife because the literature provide obvious symptom descriptions on these defects. Similarly, Munro [32] have 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 design defects have been used to systematically define a set of measurements and interpretation rules for a subset of design defects 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 major limitation of symptoms-based approaches is that there exists no consensus in defining symptoms. A defect may have several and different interpretations by a maintainer. Another limitation is that for an exhaustive list of defects, the number of possible defects to be manually described, characterized with rules and mapped to detection algorithms can be very large. Indeed, the background and knowledge of maintainers affect their understanding of defects, given a set of symptoms. As a consequence, symptoms-based approaches are also considered as time-consuming and error-prone. Thus automating the detection of design defects is still a real challenge.

2.1.3 Metric-based approaches

The idea to automate the problem of design defects detection is not new, neither is the idea to use quality metrics to 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 design defect. As such, Marinescu have defined detection strategies for capturing around ten important flaws of object-oriented design found in the literature. After his suitable symptom-based characterization of design defects, Munro [32] proposed metric-based heuristics for detecting defects, which are similar to Marinescu’s detection strategies. Munro has also performed an empirical study to justify his choice of metrics and thresholds for detecting smells. Salehie et al. [37] proposed a metric-based heuristic framework to detect and locate object-oriented design flaws similar to those illustrated by Marinescu [10]. It is accomplished by 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 general, the effectiveness of combining metric/threshold is not obvious. That is, for each defect, 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 design smells, different threshold values should be tested to find the best ones.

2.1.4 Probabilistic approaches

Probabilistic approaches represent another way for detecting defects. Alikacem et al [14] have considered the defects 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 defect 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 defect type, *i.e.*, the degree of uncertainty for a class to be a defect. They also showed that BBNs can be calibrated using historical data from both similar and different context.

Although in probabilistic approaches, the above-mentioned problems in chapter 1 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.

2.1.5 Machine learning based approaches

Machine learning represents another alternative for detecting design defects. Catal et al. [35] 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. [34] have proposed an automated approach for discovering design defects. 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 defect. Similarly, Hassaine et al. [36] have proposed an approach for detecting design 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 defects reported in the literature and freely-available.

The major benefit of machine-learning based approaches is that it does not require great experts' knowledge and interpretation. In addition, they succeeded to some extent, to detect and discover potential defects by reporting classes that are similar (even not identical) to the detected defects. However, these approaches depend on the quality and the efficiency of data, *i.e.*, defect instances, to learn from. Indeed, the high level of false positives represents the main obstacle for these approaches.

2.1.6 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 defects 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 defects 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. Indeed, since visualization approaches and tools such as VERSO [33] 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 design defects, none have presented a complete and fully automated technique. Detecting design defects is still, to some extent, a difficult, time-consuming, and manual process [9]. Indeed, the number of software defects typically exceeds the resources available to address them. In many cases, mature software projects are forced to ship with both known and unknown defects for lack of development resources to deal with every defect. Thus, to correct detected design defects and improve software quality the most useful and efficient software engineering activity is the refactoring.

2.2 Correction of design defects

2.2.1 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., design defect), 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. In [27], Sahraoui et al. 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. [39] 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 correcting design defects.

Moha et al. [40] proposed an approach that suggests refactorings using Formal Concept Analysis (FCA) to correct detected design defects. This work combines the efficiency of cohesion/coupling metrics with FCA to suggest refactoring opportunities. However, the link between defect detection and correction is not obvious, which make the inspection difficult for the maintainers. Similarly, Joshi et al. [42] have presented a 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. [41] 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 [49]. 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.

The major limitation of these manual and semi-automated approaches is that they try 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 design defects correction. Another important issue is that these approaches do not take into consideration the effort needed to apply the suggested refactorings neither the semantics coherence of the refactored program.

Recently, there exists a few works focusing on refactorings that involves semantics preservation. Bavota et al. [43] have proposed an approach of automating 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 [45], 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. In fact, they consider that the semantics preservation "*means that the observable behaviors of original and refactored programs coincide*". However, 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 move attribute and do not consider an exhaustive list of refactorings [25]. Another semantics-based framework has been proposed by Logozzo [46] 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 limitation of the mentioned works is that the definition of semantic preservation is closely related to behaviour preservation. Preserving the behavior does not means that the semantics 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 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 task. Several studies have been focused on automating software refactoring in recent years using different meta-heuristic search-based techniques for automatically searching for the suitable refactorings to be applied.

2.2.1 Meta-heuristic search-based approaches

In this subsection, we summarize existing approaches where search-based techniques have been used to automate refactoring activities.

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 [51] 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 the work of O'Keeffe et al. [22] 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 [47], 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. [48] [58] have proposed another search-based approach for finding program transformations. 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 no real semantic analysis has been performed. Recently, Otero et al. [59] use a new search-based refactoring. The core 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.

Harman et al. [20] have proposed a search-based approach using Pareto optimality that combines two quality metrics, CBO (coupling between objects) and SDMPC (standard deviation of methods per class), in two separate fitness functions. 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 unique refactoring operation (move method) to improve software quality 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.

2.3 Refactoring impact

A general conclusion to be drawn from existing refactoring work is that most of the effort has been devoted to the definition of automatic and semi-automatic approaches supporting refactoring operations. However, in this project we propose to analyze the relation between refactorings and design defects. To the best of our knowledge, only few works have dealt with that problem for bugs and not design defects. Weissgerber and Diehl [54] studied if refactoring operations are less error-prone than any other code changes. They analyzed the correlation between days with a high number of bugs and those with large number of new refactoring applied. Their results described that there is no clear correlation between the number of refactoring operations and the number of bugs created in the following days. Our proposal has several differences with the work by Weissgerber and Diehl [54]. First, Weissgerber and Diehl [54] are considering only 8 types of refactoring operations during their empirical study. In this project, we are planning to use 52 different kinds of refactoring operations based on an existing corpus describing the evolution of many open source systems over 10 years [25]. Another important difference is that we are interested to understand the correlation between refactoring operations and design defects (not bugs). Another study was presented by Ratzinger et al. [55] 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. Differently from our proposal, Ratzinger et al. study focus only on software bugs and they do not analyze the correlation between refactoring operation and specific bug types.

To conclude, each approach has its strengths and weaknesses. It helps for conducting research for automating the detection and the correction (refactoring) of design defects. In the next chapter, we describe our contributions and we show how to circumvent the above mentioned problems in both detection and correction steps. Even though most of existing refactoring approaches are powerful, to some extent, to detect design defects and to provide refactoring solutions, as discussed above, several open issues need to be addressed.

Chapter 3: Proposal

In this chapter, we describe our proposal to circumvent the problems identified in the related work and mentioned in section 1.2. Hence, we start by presenting our research objective addressed by our research. We next give an overview of our methodology to achieve these goals.

3.1 Research objective

The main objective of our research lies mainly in design defects detection and correction as a multi-objective search problem to find the compromise between the different objectives discussed in the previous section: improving the quality, preserving semantics, reducing the effort. The aim is to circumvent the problems mentioned in section by investigate the use of heuristic-search algorithms for code transformation (refactoring) and study the correlation between refactoring operations and fixing design defects by conducting a rigorous empirical evaluation using a large corpus of project histories to understand the correlation between correcting design defects and introducing new design defects or fixing other defects implicitly.

3.2 Methodology

To achieve our research objectives and circumvent the above-mentioned issues in section 1.2, we propose an approach in three steps:

1. Design defects detection: we consider the detection step as a search-based process to find the suitable detection rules for each type of design defect, by means of a genetic algorithm [51]. We use real instance examples of design defects to derive detection rules.
2. Design defects correction: we use the derived detection rules to find the best refactoring solution, which maximizes the quality (by minimizing the number of detected defects). At the same time we optimize other objectives such as

minimizing the effort needed to apply refactorings, minimize semantic errors, and maximize similarity with good refactorings applied in the past to similar contexts.

3. Empirical study: We are planning also to investigate an empirical study, using an existing corpus [27] [28] [56], to understand the correlation between correcting design defects and introducing new design defects or fixing other defects implicitly.

Figure 3.1 gives an overview of our different contributions. The following subsections give more details about our expected contributions.

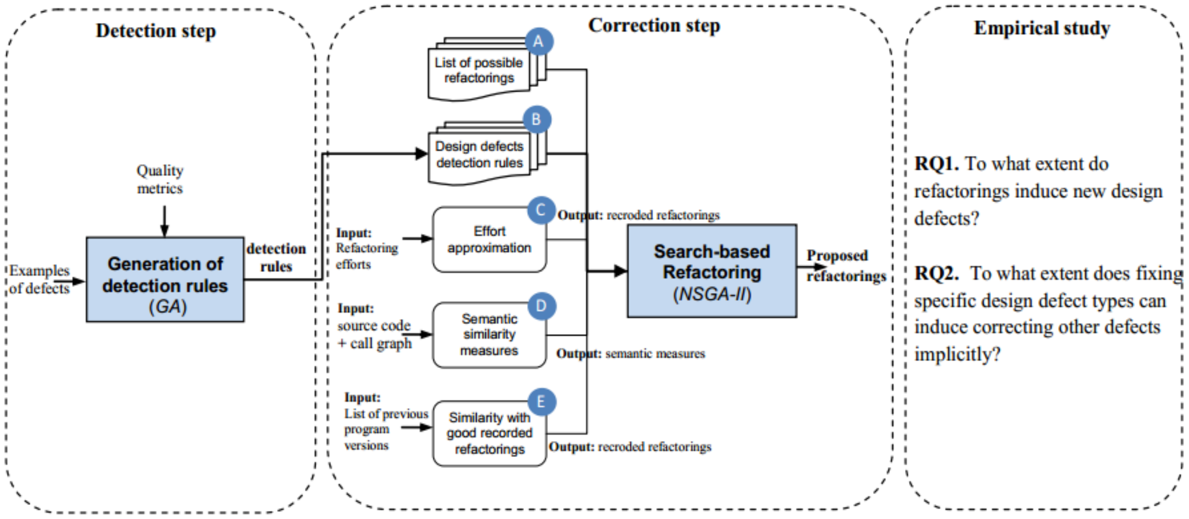
Detection step
Quality metrics
Examples of defects
Generation of detection rules (GA)
detection rules
Correction step
List of possible refactorings
Design defects detection rules
Input: Refactoring efforts
Effort approximation
Output: recroded refactorings
Input: source code + call graph
Semantic similarity measures
Output: semantic measures
Input: List of previous program versions
Similarity with good recorded refactorings
Output: recroded refactorings
Search-based Refactoring (NSGA-II)
Proposed refactorings
A
B
C
D
E
Empirical study
RQ1. To what extent do refactorings induce new design defects?
RQ2. To what extent does fixing specific design defect types can induce correcting other defects implicitly?

Figure 3.1 Our research methodology

3.2.1 Step 1: Defects detection

For the detection step, our proposal consists of using knowledge from previously manually inspected projects (called defects examples as illustrated in figure 3.1) in order to detect design defects that will serve to generate new detection rules based on combinations of quality metrics and threshold values. In short, the suitable detection rules are automatically derived by an optimization process based on genetic algorithm [51] that exploits the available examples.

3.2.2 Step 2: Defects correction

To correct the detected design defects, we use a search-based process to find the suitable refactoring solutions. To this end, use single objective optimization based approach driven by quality improvement. Then, we extend our approach by including different objectives as multi-objective optimization search-based process.

a. Quality-based refactoring suggestion

Our proposed approach aims at finding, from an exhaustive list of possible refactorings [25], the suitable refactoring solutions that fixes the detected defects by the means of a genetic algorithm [51]. Hence, a refactoring solution corresponds to a sequence of refactoring operations that should minimize as much as possible the number of defects. To this end, our search based process is guided by an evaluation function that calculates the number of detected design defects using learned detection rules from the step 1. As illustrated in figure 3.1, our approach takes as input a source code with defects and as output it suggests the suitable refactoring solutions.

b. Multi-objective Refactoring suggestion

We extend this contribution starting from the observation that finding refactoring solutions based only on the quality criteria is not enough to obtain optimal refactoring solutions. In addition to the quality criterion, we explore other objectives to optimize. Thus, to improve our suggested refactoring solutions, a multi-objective search-based approach is also used. The process aims at finding the optimal sequence of refactoring operations that optimize the following objectives:

- Minimize the number of detected defects
- Minimize the effort needed to apply refactorings
- Ensure the semantics preservation
- Maximize the similarity with refactorings applied in the past to similar contexts.

Thus, the effort corresponds to the code modification score and the cost of implementing the refactoring solution. This also includes understanding and addressing the impact of these operations. On the other hand, the semantic preservation insures that the refactored program is semantically equivalent to the original one, and that it model correctly the domain-semantics. Indeed, we use knowledge from historical code changes to propose new refactoring solutions in similar contexts to improve the automation of refactoring.

3.2.3 Step 3: Refactoring impact

As discussed in the previous chapter, design defects are not bugs. As stated by Fenton and Pfleeger [2], design defects 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. To the best of our knowledge there is no study aimed at thoroughly investigating whether a set of refactorings occurred in a software system induced new design defects; when fixing specific design defect type can correct implicitly other type of defects, and what kind of refactorings might induce more design defects than others. In the context of the study, we formulated the following research questions:

RQ1: To what extent do refactorings induce new design defects? This research question aims at investigating whether code fragments subjects to refactoring can include/generate new design defects. For example, fixing a blob defect can introduce a huge number of move method refactoring that can generate another blob.

RQ2: To what extent does fixing specific design defect types can induce correcting other defects implicitly? This research question investigates whether fixing different kinds of design defects can correct implicitly other defect types. The rationale is to investigate whether some kinds of design defects may be more important to correct than other, so that we can reduce the correction effort.

Chapter 4: Research progress

This chapter describes our progress related to the problem of design defects detection and refactoring. The first section is dedicated to the use of genetic programming to generate detection rules from design defect examples. The adaptation of genetic algorithm for the correction of the detected design defects is detailed in the second section. In the third section, we show how we considered refactoring as a multi-objective problem by integrating new objectives such as the effort minimization, semantics preservation, and similarity with recorded code changes.

4.1 Step 1: Design defect detection

In this first section, we describe our proposal for the detection of design defects. To this end, knowledge from real defect examples is used to generate detection rules. As illustrated in figure 4.1, our approach takes as inputs a base (*i.e.*, a set) of defect examples and a set of quality metrics (the definition and the usefulness of these metrics were defined and discussed in the literature [2]). As output, our approach derives a set of detection rules. Using genetic algorithm, our rules derivation process generates randomly, from a given list of quality metrics, a combination of quality metrics/threshold for each defect type. Thus, the generation process can be viewed as a search-based combinatorial optimisation to find the suitable combination of metrics/thresholds that best detect the defects examples. In other words, the best set of rules is the one that detects the maximum number of defects (we consider both precision and recall scores).

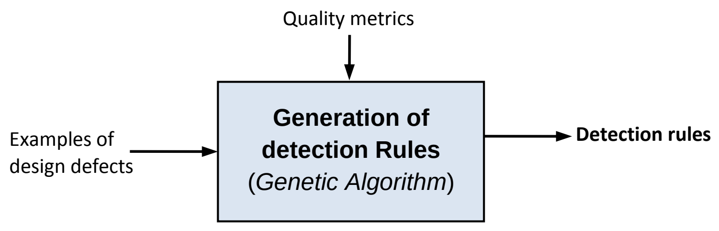
Quality metrics
Examples of design defects
Generation of detection Rules (Genetic Algorithm)
Detection rules

Figure 4.1 Overview of our design defects detection approach

To apply GA to 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.

To adapt genetic algorithm for the problem of design defects detection, we start by randomly generating an initial population which is a set of individuals that define possible detection rules. For instance, an individual is a random combination of a set of metrics/thresholds for each defect type, *e.g.*, blob, spaghetti code or functional decomposition. The GA will be iteratively executed to explore the search space (and constructs new individuals by combining metrics/thresholds within rules. During each iteration, we evaluate the quality of each individual in the population, and save the individual having the best fitness. Then, using genetic operator, we generate a new population of individuals by iteratively selecting pairs of parent individuals from the previous population 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 generated population. 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). More details related to our adaptation can be found in [27].

To test our approach, we studied its efficiency to guide quality assurance efforts. To this end, we used a set of open-source Java projects to perform our experiments. In addition, we conducted a comparative study with existing work to insure its accuracy. The results of our experiments reported in [27] have showed that our approach is able to detect more than 80% of defects which outperforms existing approaches. In addition our approach is able to circumvent the problems mentioned in section 1.2.1. After generating the detection rules, we use them in the correction step. More details about this contribution can be found in [27].

4.2 Step 2: Quality-based Refactoring suggestion

To correct the detected design defects, we propose a search-based approach that aims at finding, from an exhaustive list of possible refactorings [25], the suitable refactorings that fixes the detected defects. To this end we use genetic algorithm to find the suitable refactoring solutions. For instance, our main aim is to find refactoring solutions that should minimize as much as possible the number of defects. As illustrated in figure 4.2, our approach takes as input a source code with defects and as output it suggests the suitable refactoring solutions.

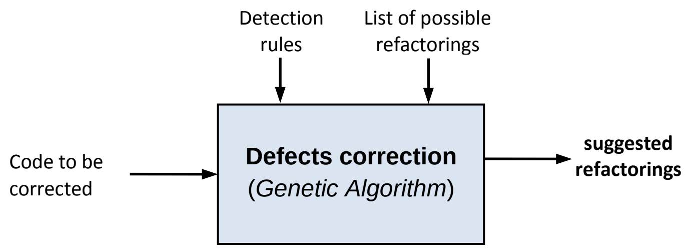

Figure 4.2 Overview of our design defects correction approach

One key issue 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, correcting detected defect classes. We view the set of potential refactoring solutions as points in a n-dimensional space, where each dimension corresponds to one refactoring operation. The sequence of applying the refactorings corresponds to their order in the vector (dimension number). Indeed, the executions of these refactorings are conformed to some pre and post conditions [1] (to avoid conflicts). To evaluate a given refactoring solution, we use an evaluation function that calculates the number of corrected design defects using learned detection rules from the step 1.

To evaluate our approach we use some open source java programs. We found that our approach is able to suggest refactoring solutions to correct the majority of the detected defects (more than 88%). More details about this contribution can be found in [27].

4.3 Step 3: Multi-objective refactoring suggestion

We found two main limitations of our mono-objective correction approach described in the previous section. First, sometimes many solutions have the same defects correction ratio but they need different code modification/adaptation effort. In some cases applying the proposed refactorings need a great. As a consequence, a large part of the program to be refactored is affected and therefore need to be modified and adapted by the maintainer. This is still time-consuming and error prone-activity. The second and the more relevant issue, is that sometimes a large number of suggested refactoring operations are not feasible due to some semantic constraints. For this reason, we contrast that the effort and the semantics preservation have to be taken into account when searching for the suitable refactoring solutions. As a consequence, we see the refactoring as a multi-objective search based problem instead of a single one.

In the literature, the common idea of the refactoring is to redistribute/reorganize software elements, e.g., classes, methods and variables, in order to improve software quality while preserving its behaviour. However, this is not enough to give concrete vision of the refactoring. There is no formal framework for refactoring that involves the semantics issues and the effort needed for the code adaptation. Hence, it is extremely challenging to formalize refactoring involving these two concepts: the semantics validity and the effort. To this end, as a first step we define a metamodel formalizing most of refactoring operations. Then we show how we use the proposed metamodel to automatically suggest refactoring solutions to correct design defects in real systems.

4.3.1 Semantic-based refactoring metamodel

We start by presenting a metamodel to describe the semantic coherence that should be preserved by refactoring tasks. We categorize refactoring from this perspective and we discuss all related issues, in particular how to ensure the semantics preservation. Figure 4.3 represents our metamodel. This metamodel is subject to refinement depending on new requirements.

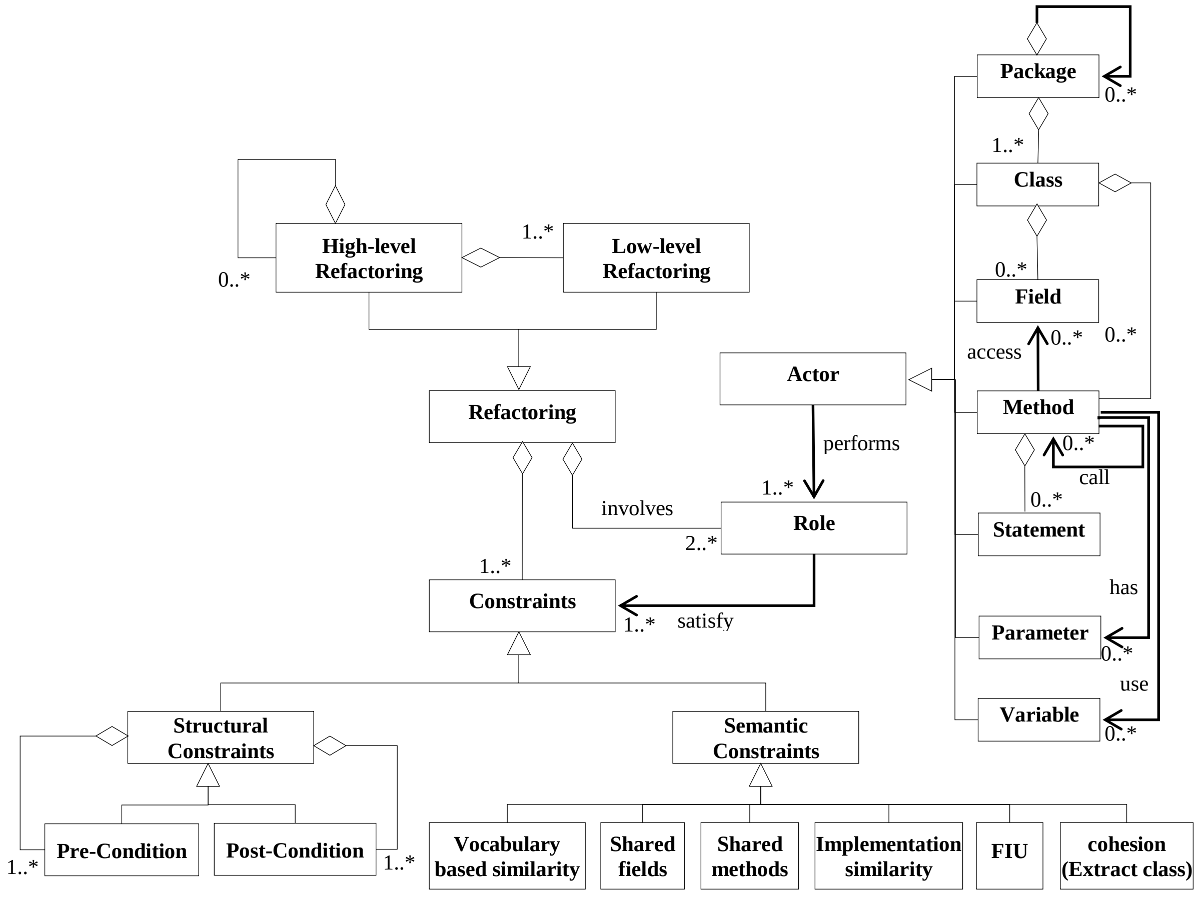

Figure 4.3 Refactoring metamodel

In the following we discuss the classes presented in this metamodel. First, the class *Refactoring* represents the main entity in our metamodel. Although, there exists several definitions for refactoring in the literature [18] [1] [17], there is no consensual definition that includes the semantics and the effort dimension. In our setting, refactoring operation can be classified as *low-level* or *high-level* refactoring. A low-level refactoring is an elementary/basic program transformation operation for adding, removing and renaming program elements (e.g., add method, remove field, add relationship, etc.). These *low-level* refactorings can be combined to perform more complex refactorings called *high-level* refactorings (e.g., move method, extract class, etc.). Furthermore, a *high-level* refactoring can be composed by one or many *low* and/or *high-level* refactorings; for example, to perform extract class we need to perform a set of move methods and move fields.

Indeed, to apply a refactoring operation we need to specify which code fragments, i.e., software artifacts, are involved/impacted by this refactoring and which roles they play to perform and assess its semantics/meaningfulness of the refactoring operation. As illustrated in figure 4.3, an actor can be a package, class, field, method, parameter, statement, or variable. In table 4.1, we summarizes for each refactoring its involved actors and performed roles.

Table 4-1 Refactoring operation and its involved actors and roles.

Refactorings	**Actors**	**Roles**
move method	class	sourceClass, targetClass
	method	movedMethod
move field	class	sourceClass, targetClass
	field	movedField
pull up field	class	subClasses, superClass
	field	movedField
pull up method	class	subClasses, superClass
	method	movedMethod
Push down field	class	superClass, subClasses
	field	movedField
push down method	class	superClass, subClasses
	method	method
inline class	class	sourceClass, targetClass
extract method	class	sourceClass, targetClass
	method	sourceMethod, newMethod
	statement	movedStatements
extract class	class	sourceClass, newClass
	field	moved Fields
	method	moved Methods
move class	package	sourcePackage, targetPackage
	class	movedClass

The class *Constraints* consists of verifying a set of constraints to assess the correctness of applied refactoring. Hence, we distinguish between two kinds of constrains: 1) *structural constraints*, and 2) *semantic constraints*. The structural constraints are extensively investigated in the literature. In [17], Opdyke have defined a set of pre and post-conditions for a large list of refactorings to ensure source-consistency. In addition to check the structural constraints (i.e., pre- and post-conditions), developers should exanimate manually all actors related to the refactoring to apply to inspect the semantic relationship between involved actors. However, to the best of our knowledge, there is no refactoring framework that has been designed to automatically check the semantic validity. In this proposal, we specify a list of heuristics to approximate the semantics between different actors that are involved to apply refactoring.

Vocabulary-based similarity: Until now semantic relationship between source code elements (refactoring actors) have not been investigated to assess refactoring validity. We use the vocabulary as an indicator of the semantic similarity between different actors that are involved by refactoring. This similarity could be interesting to consider when moving methods, fields, methods or classes, etc. For example, when a method has to be moved from one class to another, the refactoring would probably make sense if both actors (source class and target class) share the same vocabulary [28].

Dependency-based similarity: 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. In our measures, we considered two types of dependency links: 1) Shared methods (method call), and 2) Shared fields (field access read or modify). More details about our vocabulary and dependency-based similarity can be found in [28].

Feature inheritance usefulness (FIU) If a feature (method or field) is used by only few subclasses, then it is relevant to move it (i.e., push down it) to its subclasses [1]. The idea is that to insure the meaningfulness of applying push down field or push down method, we need to assess the usefulness of the method or the field in their subclasses. More a given feature is used from its subclass, more applying push down refactoring is likely to be meaningful.

Intra-element similarity We use intra-element similarity measure especially for extract class refactoring that need a particular exploration. Therefore, a new class can be extracted from a large class by moving a set of strongly related fields and methods from the original class into the new one to 1) improve the cohesion of the original class and 2) obtain low coupling with the new class. An illustrative example can be found in [28].

Thus, to improve our genetic algorithm-based approach described in the previous section, we use concepts described in our metamodel to deal with minimizing code adaptation/modification score (effort) and to maximizing semantic correctness our suggested refactorings.

4.3.2 Refactoring as a multi-objective problem

To get optimal refactoring solutions multiple objectives have to be optimized: 1) defects correction, 2) effort minimisation and 3) semantics errors minimisation. We also consider another objective to maximize the similarity with good refactorings applied in the past to similar contexts. To this end, we propose a multi-objective approach to correct detected defects instead of single objective one. We use the Non-dominated Sorting Genetic Algorithm (NSGA-II) to find the best compromise between all of these objectives. In this subsection we describe our proposal. We start by giving an overview of NSGA-II.

4.3.2.1 NSGA-II overview

NSGA-II [24] is a powerful search method inspired from natural selection. The basic idea is to make a population of candidate solutions evolve toward the best solution in order to solve a multi-objective optimization problem. NSGA-II was designed to be applied to an exhaustive list of candidate solutions, which creates a huge search space. Thus, the main idea of NSGA-II is to calculate the Pareto front that corresponds to a set of optimal solutions called non-dominated solutions or Pareto set. A non-dominated solution is one which provides a suitable compromise between all objectives without degrading any of them. The first step in NSGA-II [24] is to randomly create the initial population P_0 of individuals encoded using a specific representation. Then, a child population Q_0 is generated from the population of parents P0 using genetic operators such as crossover and mutation. Both populations are merged and a subset of individuals is selected, based on the dominance principle to create the next generation. This process will be repeated until reaching the last iteration according to stop criteria.

4.3.2.2 NSGA-II adaptation

To be applied, NSGA-II need to specify some elements for its implementation: 1) the representation of individuals used to create a population; 2) a fitness function to evaluate the candidate solutions according to each objective; 3) the crossover and mutation operators that have to be designed according to the individual's representation. In addition, a method to select the best individuals has to be implemented to create the next generation of individuals. As output, NSGA-II returns the best individuals (with highest fitness scores) produced during all iterations. More details on our NSGA-II adaptation are reported in [27] and [28]. In the following, we show how we integrated each objective (quality, effort, semantics, similarity with recorded changes) as described by our methodology in chapter 4.

4.3.2.2.1 Quality improvement

Since our search based process is guided by the goal of correcting design defects, we evaluate the quality by the number of corrected design defects. The quality value increases when the number of defects in the code is reduced after the correction. This evaluation returns a real value between 0 and 1 that represents the proportion of defected classes compared to the total number of possible defects that can be detected.

4.3.2.2.2 Effort minimization

We started by integrating the effort minimization as a second objective to optimize with the quality improvement (defects correction). The aim is to find refactoring solutions that correct the detected defects while minimizing the correction effort. Thus, a correction solution is defined as the combination of refactoring operations that should maximize as much as possible the number of corrected defects with minimal effort.

In practice, there exists no consensual way on how to calculate the effort needed to concretely apply refactorings. In our approach, we propose an alternative method to estimate the effort according to code-complexity and code-changes when applying a refactoring operation. To this end, we use an effort calculation model that considers two categories of operations: low-level and high-level refactorings as described in our refactoring metamodel (figure 4.3). For each low-level operation, we manually attribute an effort value (equal to 1, 2, or 3) based on our expertise on performing such an operation. This value reflects the amount of code that needs to be created, modified or inspected. We consider, in particular, the type of the involved fragment (parameter, field, method, class, etc.) and the possible change impact. Then, for each high-level refactoring, we derive the effort by cumulating the effort values of its contained low-level operations [27]. Roughly speaking, our approach takes as input defective code fragments to be corrected, a set of possible refactoring operations with its associated efforts, and a set of defect detection rules as illustrated in figure 4.4. As output, our approach recommends a set of optimal refactoring solutions.

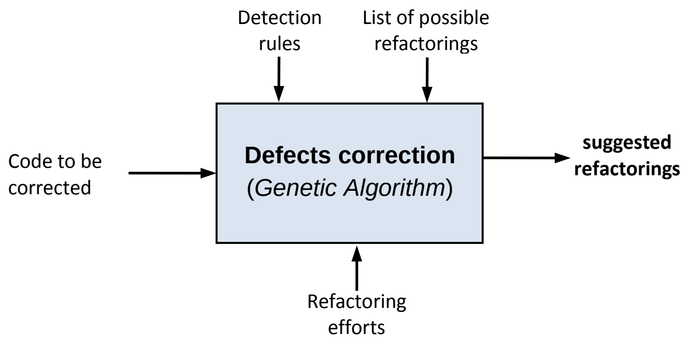

Figure 4.4 Overview of our approach for effort minimization

To test our approach, we studied its usefulness to guide quality assurance efforts for a set of open-source programs. The obtained results are promising. Thus, our experiments show that our approach is able to correct the majority of detected defects with an acceptable correction score (75%). However, our approach provides a significant reduction in terms of code adaptation score comparing with use of single objective genetic algorithm. More details of our finding can be found in [27].

We are convinced that suggesting refactoring strategies using only structural information is one of the important limitations of the existing approaches including the one we proposed for instance. Although our approach have provided promising results, preserving the semantics domain is fundamental when applying refactoring. In the following we describe our novel approach that allows to take into consideration the semantics of the program to be refactored.

4.3.2.2.3 Semantics preservation

To preserve the semantics of the original program during the refactoring process, we combine two techniques to approximate the semantic proximity between classes when moving elements between them. The first technique is related to the vocabulary-based similarity (using names of methods, fields, types, super and sub classes, etc.). The second technique considers the dependency-based similarity between classes using call graphs and field references. These two approximations are described in section 4.3.1 and detailed in [28].

The general structure of our approach is sketched in Figure 4.5. As our aim is to maximize the quality and minimize semantic errors, we consider each one of these criteria as a separate objective for NSGA-II. The NSGA-II algorithm takes as input the whole source code to be corrected, a set of possible refactoring operations, defect detection rules, and a call graph. As output our approach suggests a set of correction solutions (*i.e.*, sequence of refactoring operations).

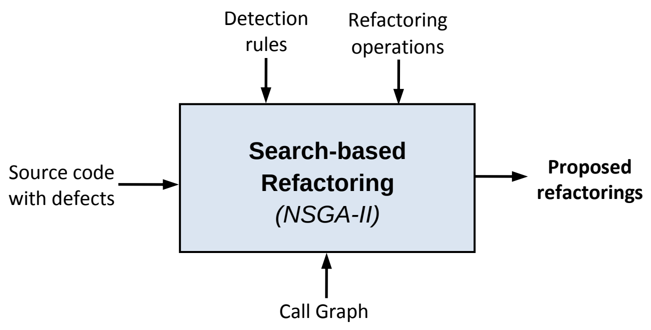

Figure 4.5 Approach overview for semantics preservation

To evaluate our approach, we conducted experiments on two open-source systems. We also conducted a comparative study with some existing approaches to insure the efficiency of including the semantics preservation. Our experiments shows that the proposed approach is able to suggest refactorings preserve the semantics of the program to restructure while correcting existing design defects. Our search-based approach succeeded to produce more semantic and meaningful refactorings in comparison of those of the state of the art. More details of our experiments can be found in [28].

In this contribution, only vocabulary and dependency-based similarity are considered. However, we are convinced that other semantics approximations should be considered to extend this contribution. We plan to improve our proposal by including others measures that are described in our refactoring metamodel (figure 4.3).

4.3.2.2.4 Similarity with recorded code changes

Over the past decades, many techniques and tools have been developed to record the sequence of applied refactorings to improve design quality. We start from the observation that these recorded code changes can be used to propose new refactoring solutions in similar contexts. In addition, this knowledge can be combined with structural and semantic information to improve the automation of refactoring. In this contribution, we propose a multi-objective optimization approach to find the best sequence of refactorings that maximizes the use of refactoring applied in the past to similar contexts, minimizes semantic errors and minimizes the number of defects (improve code quality). To this end, we use the non-dominated sorting genetic algorithm (NSGA-II) to find the best trade-off between these three objectives.

The general structure of our approach is introduced in Figure 4.6. It takes as inputs a history of applied code changes/refactoring to previous versions of the system (label B), a list of possible refactorings that can be applied (label A), our design defects detection rules (label C), and our semantic measures (vocabulary and dependency- based measures) (label D). Our approach generates as output the best sequence of refactoring, selected from the exhaustive list, that maximizes the use of refactoring applied in the past to similar contexts, minimizes semantic incoherence and minimizes the number of defects (improve code quality).

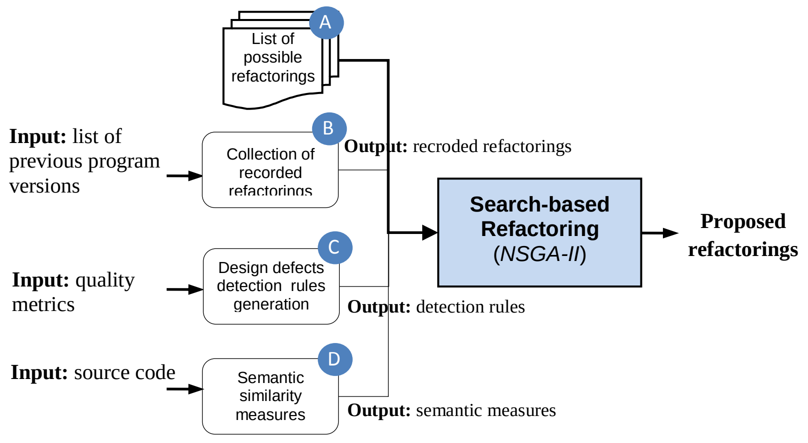
A
List of possible refactorings
B
Input: list of previous program versions
Collection of recorded refactorings
Output: recroded refactorings
Search-based Refactoring (NSGA-II)
Proposed refactorings
C
Input: quality metrics
Design defects detection rules generation
Output: detection rules
D
Input: source code
Semantic similarity measures
Output: semantic measures

Figure 4.6 Multi-objective refactoring using recorded code changes

In order to evaluate the feasibility of our approach for generating good refactoring suggestions, we conducted an experiment based on different versions of large open source systems. In our study, we assess the performance of our approach by finding out whether it could generate meaningful sequences of refactorings to correct design defects, preserve the semantics of the design, and reuse as much as possible a base of recorded code changes to similar contexts. Our experiments reported in [56] shows that the majority of suggested refactorings improve significantly the code quality while preserving the semantics. The results show also that half of recorded/collected refactorings are used by the optimal solution.

Chapter 5: Research plan

5.1 Research Schedule

Step	**Period**	**Status**	**Publication**
Course IFT2125, IFT2015	H11	Completed	
Course IFT6251: Sujets en Génie Logiciel	A11	Completed	
Course IFT6315: Compréhension et analyse de programmes	H12	Completed	
Design defects detection and correction			
Single-objective detection and correction (prove of concepts)	H11	Completed	ICPC 2011: accepted
Predoc I	A11	Completed	
Multi-objective search-based refactoring			
Multi-objective search-based refactoring: effort minimisation	A11	Completed	Journal of Automated Software Engineering 2012: accepted
Search-based Refactoring: Towards Semantics Preservation	H12	Completed	ICSM 2012: accepted
Search-based Refactoring Using Code Changes	E12 and A12	Completed	CSMR 2012: accepted
Automating Software Refactoring Using Multiple Criteria: Quality, Semantic Preservation, Effort and Recorded Code Changes	A12 and H13	80%	IEEE Transactions on Software Engineering
Predoc II	H12	Completed	
Empirical study: Refactoring Impact on Design Defects			
Empirical study to investigate the relationship between refactoring and correcting/introducing new design defects	H13	0%	FSE 2013
Empirical study: Refactoring impact on design defects	E13 and A13	0%	ICSE2014; Transactions on Software Engineering and Methodology
Thesis writing	A13 and H14		

Table 5-1 Research schedule

5.2 Publication list

Refereed Articles in International Journals:

Ali Ouni, Marouane Kessentini, Houari Sahraoui and Mounir Boukadoum, Maintainability Defects Detection and Correction: A Multi-Objective Approach, Journal of Automated Software Engineering (JASE), Springer, 2012. (accepted)

Ali Ouni, Marouane Kessentini, Houari Sahraoui, Automating Software Refactoring Using Multiple Criteria: Quality, Semantic Preservation, and Effort. IEEE Transactions on Software Engineering, 2013. (in progress)

Refereed Articles in International Conferences:

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), march 5-8, 2013, Genova, Italy. (accepted)

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

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), 22-24 June 2011, pp 81-90, Kingston-Canada. (accepted)

Chapter 6: Conclusion

In summary, the main contributions our thesis is to propose a novel approach for automating the detection and correction of design defects and to study the impact of refactoring on design defects during software evolution.

For the design defects detection step, the problem is seen as a search-based combinatorial optimization problem to find the suitable detection rules using real examples of defects. Typically, researchers and practitioners try to characterize different types of common design defects and present symptoms to search for in order to locate possible design defects in a system. In our approach, we have shown that this knowledge is not necessary to perform the detection. Instead, we use examples of design defects to generate detection rules.

After generating the detection rules, we use them in the correction step. To this end, we propose a search-based approach to find the combination of refactoring operations that best correct the detected defects. Thus, a good refactoring solution is a sequence of refactoring operations that minimize as much as possible the number of detected defects to improve software quality. This approach was tested in five open source systems and the results are promising. In addition, we explore other objectives to optimize: 1) the effort needed to apply refactorings, 2) semantics preservation, and 3) the similarity with good refactorings applied in the past to similar contexts.

To minimize the refactoring effort we see the refactoring process as a multi-objective optimisation problem. We define a "good" refactoring solution as the combination of refactoring operations that should maximize as much as possible the number of corrected defects with minimal code modification/adaptation effort (*i.e.*, the cost of applying the refactoring sequence). The idea is to find the best compromise between maximizing quality and minimizing code adaptability effort. In addition, we integrated the semantics preservation especially that, in our search-based approach, refactoring solutions are decided automatically. Hence, a program could be syntactically correct, have the right behavior, but model incorrectly the domain semantics. In this contribution, we propose a multi-objective optimization approach to find the best sequence of refactorings that maximizes quality improvements (*i.e.,* minimize the number of defects) and at the same time minimizes potential semantic errors. Finally, to improve the automaton of refactoring, we start from the observation that recorded/historical code changes could be used to propose new refactoring solutions in similar contexts. In addition, this knowledge can be combined with structural and semantic information to improve the automation of refactoring. In this contribution, we propose a multi-objective optimization to find the compromise between all of the mentioned objectives. Our approaches was tested on a set of open source systems and the results was promising.

As part of our future work, we plan to combine our multiple criteria (quality improvement, effort, semantics preservation, and similarity with recorded refactorings to similar contexts) in order to improve the automaton of the refactoring activity. In addition, we will perform an empirical study to understand the correlation between correcting design defects and introducing new design defects or fixing other defects implicitly.

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, November 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, 1999, pp. 47–56.

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

[9] Liu, H., Yang, L., Niu, Z., Ma, Z., Shao, W.: 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] Khomh, F., Vaucher, S., Guéhéneuc, Y.-G., Sahraoui, H.: 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, March 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] Kothari, S.C., Bishop, L., Sauceda, J., Daugherty, G.: A pattern-based framework for software anomaly detection. Softw. Qual. J.12(2), 99–120, 2004.

[16] Dhambri, K., Sahraoui, H.A., Poulin, P.: 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, February 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), 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, Apr. 2002.

[25] http://www.refactoring.com/catalog/

[26] 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.

[27] 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.

[28] 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, 2012, Riva del Garda, Italy.

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

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] Salehie, M., Li, S., Tahvildari L., 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.

[38] 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.

[39] 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.

[40] 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.

[41] Tahvildari, L., Kontogiannis, K, 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.

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

[43] 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.

[44] 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.

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

[46] 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.

[47] 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.

[48] D. Fatiregun, M. Harman, and R. M. Hierons. Evolving transformation sequences using genetic algorithms. In Proc. of 4th SCAM, pages 66-75, 2004.

[49] 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.

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

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

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

[53] 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.

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

[55] 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.

[56] 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, 2013, Genova, Italy.

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

[58] D. Fatiregun, M. Harman, and R. Hierons. Search-based amorphous slicing. InWCRE 05,pages 3–12, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA, Nov. 2005.

[59] F. E. B. Otero, C. G. Johnson, A. A. Freitas, , and S. J. Thompson. Refactoring in automatically generated programs.Search Based Software Engineering, International Symposium on, 0, 2010.