Multi-criteria Code Refactoring Using Search-Based Software Engineering: An Industrial Case Study

ALI OUNI, Osaka University
MAROUANE KESSENTINI, University of Michigan-Dearborn
HOUARI SAHRAOUI, University of Montreal
KATSURO INOUE, Osaka University
KALYANMOY DEB, Michigan State University

One of the most widely used techniques to improve the quality of existing software systems is refactoring – the process of improving the design of existing code by changing its internal structure without altering its external behavior. While it is important to suggest refactorings that improve the quality and structure of the system, many other criteria are also important to consider such as reducing the number of code changes, preserving the semantics of the software design and not only its behavior, and maintaining consistency with the previously applied refactorings. In this paper, we propose a multi-objective search-based approach for automating the recommendation of refactorings. The process aims at finding the optimal sequence of refactorings that (*i*) improves the quality by minimizing the number of design defects, (*ii*) minimizes code changes required to fix those defects, (*ii*) preserves design semantics, and (*iv*) maximizes the consistency with the previously code changes. We evaluated the efficiency of our approach using a benchmark of six open-source systems, 11 different types of refactorings (move method, move field, pull up method, pull up field, push down method, push down field, inline class, move class, extract class, extract method and extract interface) and 6 commonly occurring design defect types (blob, spaghetti code, functional decomposition, data class, shotgun surgery and feature envy) through an empirical study conducted with experts. In addition, we performed an industrial validation of our technique, with 10 software engineers, on a large project provided by our industrial partner. We found that the proposed refactorings succeed in preserving the design coherence of the code, with an acceptable level of code change score while reusing knowledge from recorded refactorings applied in the past to similar contexts.

General Terms: Algorithms, Reliability

Additional Key Words and Phrases: Search-based Software Engineering, Refactoring, Software Maintenance, Multi-Objective Optimization, Software Evolution

ACM Reference Format:
Ali Ouni, Marouane Kessentini, Houari Sahraoui, Katsuro Inoue, Kalynmoy Deb, 2015. Multi-criteria Code Refactoring using Search-Based Software Engineering: An Industrial Case Study. *ACM Trans. Softw. Eng. Methodol.* 9, 4, Article 39 (March 2015), 54 pages.
DOI: 0000001.0000001

1. INTRODUCTION

Large scale software systems exhibit high complexity and become difficult to maintain. In fact, it has been reported that the software cost attributable to maintenance and evolution activities is more than 80% of total software costs [Erlikh 2000]. To facilitate maintenance tasks, one of the most widely used techniques is refactoring which improves design structure while preserving external behavior [Mens and Tourwé 2004; Opdyke 1992].

Author's addresses: Ali Ouni, Osaka University, ali@ist.osaka-u.ac.jp, Marouane Kessentini, University of Michigan-Dearborn, marouane@umich.edu, Houari Sahraoui, University of Montreal, sahraouh@iro.umontreal.ca, Katsuro Inoue, Osaka University, inoue@ist.osaka-u.ac.jp, and Kalyanmoy Deb, Michigan State University, kdeb@egr.msu.edu.

Even though most of the existing refactoring recommendation approaches are powerful enough to suggest refactoring solutions to be applied, several issues are still need to be addressed. One of the most important issues is the semantic coherence of the refactored program, which is not considered by most of the existing approaches [Ouni et al. 2012a; Du Bois et al. 2004; Moha et al. 2008; Mens and Tourwé 2004]. Consequently, the refactored program could be syntactically correct, implement the correct behavior, but be semantically incoherent. For example, a refactoring solution might move a method `calculateSalary()` from class `Employee` to class `Car`. This refactoring could improve the program structure by reducing the complexity and coupling of class `Employee` and satisfy the pre- and post-conditions to preserve program behavior. However, having a method `calculateSalary()` in class `Car` does not make any sense from the domain semantics standpoint, and is likely to lead to comprehension problems in the future. Another issue is related to the number of code changes required to apply refactorings, something that is not considered in existing refactoring approaches whose only aim is to improve code quality independently of the cost of code changes. Consequently, applying a particular refactoring may require a radical change in the system or even its re-implementation from scratch. Thus, it is important to minimize code changes to help developers in understanding the design after applying the proposed refactorings. In addition, the use of development history can be an efficient aid when proposing refactorings. Code fragments that have previously been modified in the same time period are likely to be semantically related (e.g., refer to the same feature). Furthermore, code fragments that have been extensively refactored in the past have a high probability of being refactored again in the future. Moreover, the code to refactor can be similar to some refactoring patterns that are to be found in the development history, thus, developers can easily adapt and reuse them.

One of the limitations of the existing works in software refactoring [Du Bois et al. 2004; Qayum and Heckel 2009; Fokaefs et al. 2011; Harman and Tratt 2007; Moha et al. 2008; Seng et al. 2006] is that the definition of semantic coherence is closely related to behavior preservation. Preserving the behavior does not means that the design semantics of the refactored program is also preserved. Another issue is that the existing techniques are limited to a small number of refactorings and thus it could not be generalized and adapted for an exhaustive list of refactorings. Indeed, semantic coherence is still hard to ensure since existing approaches do not provide a pragmatic technique or an empirical study to prove whether the semantic coherence of the refactored program is preserved.

In this paper, we propose a multi-objective search-based approach to address the above-mentioned limitations. The process aims at finding the sequence of refactorings that: (1) improves design quality; (2) preserves the design coherence and consistency of the refactored program; (3) minimizes code changes; and (4) maximizes the consistency with development change history. We evaluated our approach on six open-source systems using an existing benchmark [Ouni et al. 2012a; Moha et al. 2010; Moha et al. 2008]. We report the results of the efficiency and effectiveness of our approach, compared to existing approaches [Harman and Tratt 2007; Kessentini et al. 2011]. In addition, we provide an industrial validation of our approach on a large-scale project in which the results were manually evaluated by 10 active software engineers. The study also evaluated the relevance and usefulness of our refactoring technique in an industrial setting.

The remainder of this paper is structured as follows. Section 2 provides the necessary background and challenges related to refactoring and code smells. Section 3 defines refactoring recommendation as a multi-objective optimization problem, while Section 4 introduces our search-based approach to this problem using the non-dominated sorting genetic algorithm (NSGA-II) [Deb et al. 2002]. Section 5 describes the method used in our empirical studies and presents the obtained results, while Section 6 provides further discussions. Section 7 presents an industrial case study long with a discussion of the obtained results. Section 8 discusses the threats to validity and the limitations of the proposed approach, while Section 9 describes the related work. Finally, Section 10 concludes and presents directions for future work.

2. CHALLENGES IN AUTOMATED REFACTORING RECOMMENDATION

In this section, we define the issues and challenges related to software refactoring.

2.1. Background and definitions

Refactoring is defined as the process of improving a code after it has been written by changing its internal structure without changing its external behavior [Opdyke 1992]. The idea is to reorganize variables, classes and methods, mainly to facilitate future adaptations and extensions. This reorganization is used to improve various aspects of software quality such as maintainability, extensibility, reusability, etc. [Fowler 1999; Baar and Marković 2007]. The refactoring process consists of 6 distinct steps [Baar and Marković 2007]:

(1) Identify where the software should be refactored.
(2) Determine which refactoring(s) should be applied to the identified places.
(3) Guarantee that the applied refactoring preserves behavior.
(4) Apply the refactoring.
(5) Assess the effect of the refactoring on quality characteristics of the software (e.g., complexity, understandability, maintainability) or the process (e.g., productivity, cost, effort).
(6) Maintain the consistency between the refactored program code and other software artifacts (such as documentation, design documents, requirement specifications, tests, etc.).

We focus in this paper on steps 1, 2 and 5. In order to find out which parts of the source code need to be refactored, most existing work [Dhambri et al. 2008; Moha et al. 2010; Marinescu 2004; Murphy-Hill and Black 2010] relies on the notion of design defects or bad smells. In this paper, we do not focus on the first step related to the detection of refactoring opportunities. We assume that a number of different design defects have already been detected, and need to be corrected. Typically, design defects, also called anomalies [Brown et al. 1998], design flaws [Marinescu 2004], bad smells [Fenton and Pfleeger 1998], or anti-patterns [Fowler 1999], refer to design situations that adversely affect the development of software. As stated by Fenton and Pfleeger [Fenton and Pfleeger 1998], design defects are unlikely to cause failures directly, but may do so indirectly [Yamashita and Moonen 2013]. In general, they make a system difficult to change, which may often introduce bugs. In this paper, we focus on the following six design defect types [Brown et al. 1998; Murphy-Hill and Black 2010; Mäntylä et al. 2003] to evaluate our approach:

— *Blob*: It is found in designs where much of the functionality of a system (or part of it) is centralized in one large class, while the other related classes primarily expose data and provide little functionality.
— *Spaghetti Code*: This involves a code fragment with a complex and tangled control structure. This code smell is characteristic of procedural thinking in object-oriented programming. Spaghetti Code is revealed by classes declaring long methods with no parameters, and utilising global variables. Names of classes and methods may suggest procedural programming. Spaghetti Code does not exploit, and indeed prevents the use of, object-oriented mechanisms such as inheritance and polymorphism.

— *Functional Decomposition*: This design defect consists of a main class in which inheritance and polymorphism are hardly used, that is associated with small classes, which declare many private fields and implement only a few methods. This is frequently found in code produced by inexperienced object-oriented developers.
— *Data Class*: It is a class that contains only data and performs no processing on these data. It is typically composed of highly cohesive fields and accessors. However, depending on the programming context some Data Classes might suit perfectly and, therefore, not design defects.
— *Shotgun Surgery*: This occurs when a method has a large number of external methods calling it, and these methods are spread over a significant number of classes. As a result, the impact of a change in this method will be large and widespread.
— *Feature Envy*: It is found when a method heavily uses attributes and data from one or more external classes, directly or via accessor operations. Furthermore, in accessing external data, the method uses data intensively from at least one external source.

We choose these design defect types in our experiments because they are the most important and common ones in object-oriented industrial projects based on recent empirical studies [Ouni et al. 2012a; Moha et al. 2008; Ouni et al. 2013]. Moreover, it is widely believed that design defects have a negative impact on software quality that often leads to bugs and failures [Li and Shatnawi 2007; D'Ambros et al. 2010; Deligiannis et al. 2003; Mäntylä et al. 2003]. Consequently, design defects should be identified and corrected by the development team as early as possible for maintainability and evolution considerations. 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 method and extract class.

In the next subsection, we discuss the different challenges related to fixing design defects using refactoring.

2.2. Problem statement

Even though most existing refactoring approaches are powerful enough to provide refactoring solutions, some open issues need to be targeted to provide an efficient and fully automated refactoring recommendation.

Quality improvement: Most of the existing approaches [Qayum and Heckel 2009; O'Keeffe and Cinnéide 2008; Moha et al. 2008; Seng et al. 2006] consider refactoring as the process to improve code quality by improving structural metrics. However, these metrics can be conflicting and it is difficult to find a compromise between them. For example, moving methods to reduce the size or complexity of a class may increase the global coupling. Furthermore, improving some quality metrics does not guarantee that the detected design defects are fixed. Moreover, there is no consensus about the metrics that need to be improved in order to fix defects. Indeed, the same type of defect can be fixed by improving completely different metrics.

Semantic coherence: In object-oriented programs, objects reify domain concepts and/or physical objects, implementing their characteristics and behavior. Methods and fields of classes characterize the structure and behavior of the implemented domain elements. Consequently, a program could be syntactically correct, implement the appropriate behavior, but violate the domain semantics if the reification of domain elements is incorrect. During the initial design/implementation, programs usually capture well the domain semantics when object-oriented principles are applied. However, when these programs are (semi-)automatically modified/refactored during maintenance, the adequacy with regards to domain semantics could be compromised. Indeed, semantic coherence is an important issue to consider when applying refactorings.

Most of the existing approaches suggest refactorings mainly with the perspective of only improving some design/quality metrics. As explained, this may not be sufficient. We need to preserve the rationale behind why and how code elements are grouped and connected when applying refactoring operations to improve code quality.

Code changes: When applying refactorings, various code changes are performed. The amount of code changes corresponds to the number of code elements (e.g., classes, methods, fields, relationships, field references, etc.) modified through adding, deleting, or moving operations. Minimizing code changes when suggesting refactorings is important to reduce the effort and help developers understand the modified/improved design. In fact, most developers want to keep as much as possible with the original design structure when fixing design defects [Fowler 1999]. Hence, improving software quality and reducing code changes are conflicting. In some cases, correcting some design defects corresponds to changing radically a large portion of the system or is sometimes equivalent to re-implementing a large part of the system. Indeed, a refactoring solution that fixes all defects is not necessarily the optimal one due to the high code adaptation/modification that may be required.

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

2.3. Motivating example

To illustrate some of these issues, Figure 1 shows a concrete example extracted from *JFreeChart*[1] v1.0.9, a well-known Java open-source charting library. We consider a design fragment containing four classes `XYLineAndShapeRenderer`, `XYDotRenderer`, `SegmentedTimeline`, and `XYSplineRenderer`. Using design defect detection rules proposed in our previous work [Kessentini et al. 2011], the class `XYLineAndShapeRenderer` is detected as a design defect: blob (i.e., a large class that monopolizes the behavior of a large part of the system).

We consider the scenario of a refactoring solution that consists of moving the method `drawItem()` from class `XYLineAndShapeRenderer` to class `SegmentedTimeline`. This refactoring can improve the design quality by reducing the number of functionalities in this blob class. However, from the design semantics standpoint, this refactoring is incoherent since `SegmentedTimeline` functionalities are related to presenting a series of values to be used for a curve axis (mainly for Date related axis) and not for the task of drawing objects/items. Based on textual and structural information, using respectively a semantic lexicon [Amaro et al. 2006], and cohesion/coupling [Ouni et al. 2012b], many other target classes are possible including `XYDotRenderer` and `XYSplineRenderer`. These two classes have approximately the same structure that can be formalized using quality metrics (e.g., number of methods, number of attributes, etc.) and their textual similarity is close to `XYLineAndShapeRenderer` using a vocabulary-based measure. Thus, moving elements between these three classes is likely to be semantically coherent and meaningful. On the other hand, from previous versions of *JFreeChart*, we recorded that there are some methods such as `drawPrimaryLineAsPath()`, `initialise()`, and `equals()` that have been moved from class `XYLineAndShapeRenderer` to class `XYSplineRenderer`. As a conse-

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

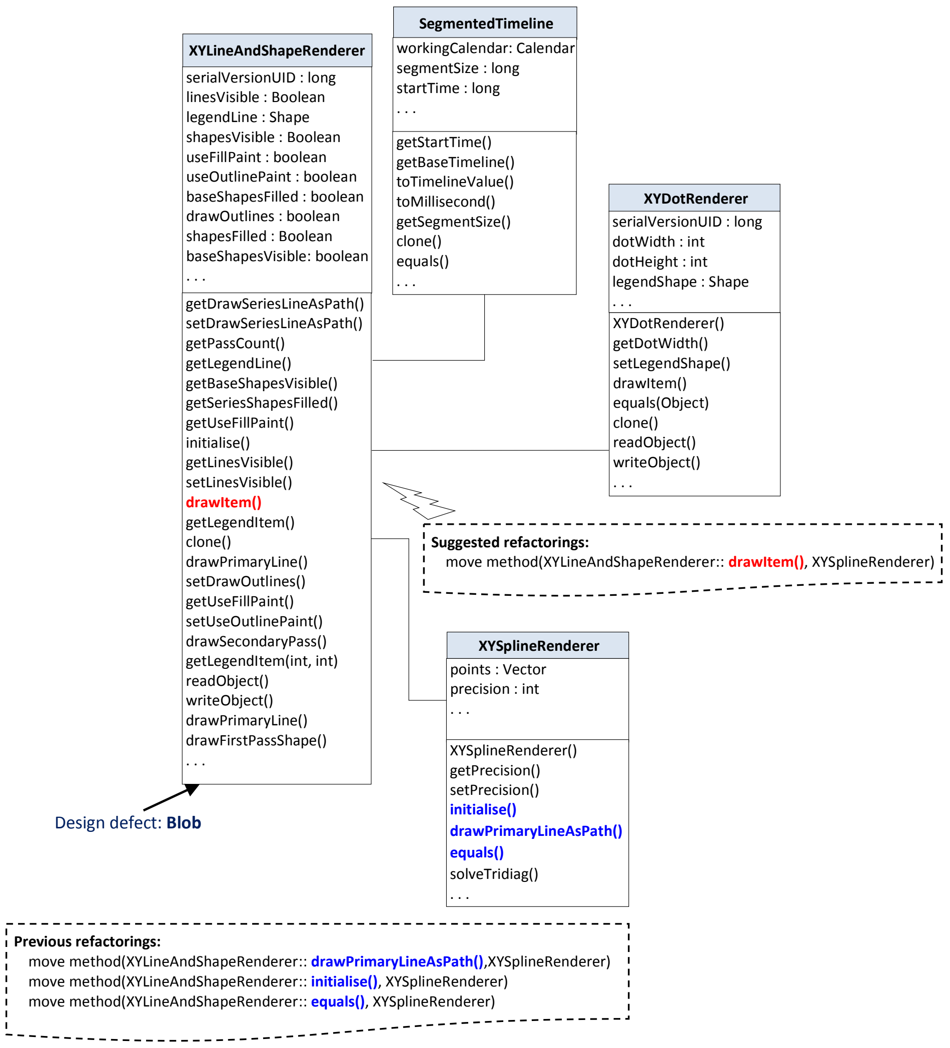


Fig. 1: Design fragment extracted from *JFreeChart* v1.0.9.

quence, moving methods and/or attributes from class `XYLineAndShapeRenderer` to class `XYSplineRenderer` has higher correctness probability than moving methods or attributes to class `XYDotRenderer` or `SegmentedTimeline`.

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

3. REFACTORING: A MULTI-OBJECTIVE PERSPECTIVE

3.1. Overview

Our approach aims at exploring a large search space to find refactoring solutions, i.e., a sequence of refactoring operations, to correct bad smells. The search space is determined not only by the number of possible refactoring combinations, but also by the order in which they are applied. A heuristic-based optimization method is used to generate refactoring solutions. We have four objectives to optimize: 1) maximize quality improvement (bad smells correction); 2) minimize the number of design coherence errors by preserving the way code elements are semantically grouped and connected together; 3) minimize code changes needed to apply the refactorings; and 4) maximize the consistency with development change history. We thus consider the refactoring task as a multi-objective optimization problem using the non-dominated sorting genetic algorithm (NSGA-II) [Deb et al. 2002].

The general structure of our approach is sketched in Figure 2. It takes as input the source code of the program to be refactored, a list of possible refactorings that can be applied (label A), a set of bad smell detection rules (label B) [Ouni et al. 2012a], our technique for approximating code changes needed to apply refactorings (label C), a set of textual and design coherence measures described in Section 3 (label D), and a history of applied refactorings to previous versions of the system (label E). Our approach generates as output a near-optimal sequence of refactorings that improves the software quality by minimizing as much as possible the number of design defects, minimizing code changes required to apply the refactorings, preserving design semantics, and maximizing the consistency with development change history. Our approach currently supports eleven refactoring operations including move method, move field, pull up field, pull up method, push down field, push down method, inline class, extract method, extract class, move class, and extract interface (cf. Table II) [Fowler 1999], but not all refactorings in the literature[2]. We selected these refactorings because they are the most frequently used refactorings and they are implemented in most modern IDEs such as Eclipse and Netbeans. In the following, we describe the formal formulation of the four objectives to optimize.

3.2. Modeling the refactoring process as a multi-objective problem

3.2.1. Quality. The Quality criterion is evaluated using the fitness function given in Equation 1. The quality value increases when the number of defects in the code is reduced after refactoring. This function returns the complement of the ratio of the number of design defects after refactoring (detected using bad smells detection rules) over the total number of defects that are detected before refactoring. The detection of defects is based on some metrics-based rules according to which a code fragment can be classified as a design defect or not (without a probability/risk score), i.e., 0 or 1, as defined in our previous work [Kessentini et al. 2011; Ouni et al. 2012a]. The accuracy of the genetic programming approach for code smells detection proposed in our previous studies was an average of 91% of precision and 87% of recall on 8 large-scale systems. The defect correction ratio function is defined as follows:

$$DCR = 1 - \frac{\#\ \textit{defects after applying refactorings}}{\#\ \textit{defects before applying refactorings}} \tag{1}$$

3.2.2. Code changes. Refactoring Operations (ROs) are classified into two types: Low-Level ROs (LLR) and High-Level ROs (HLR) [Ouni et al. 2012a]. A HLR is a sequence

[2]http://refactoring.com/catalog/

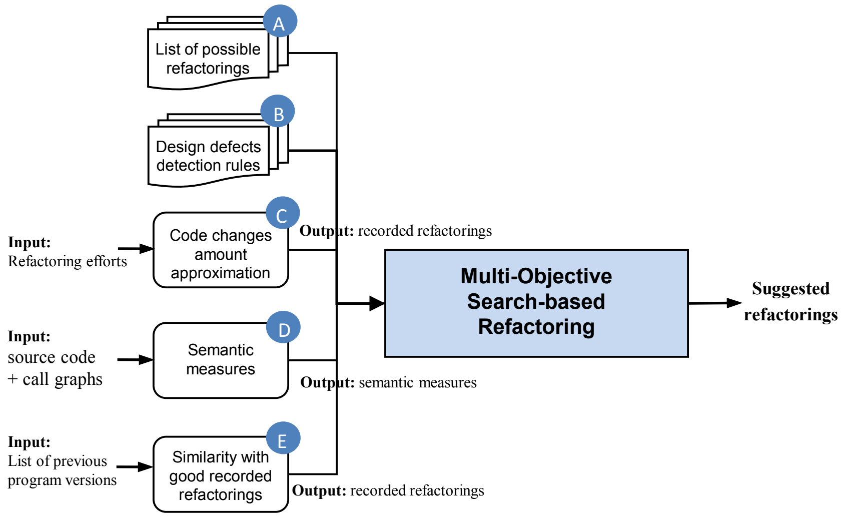

A
List of possible refactorings
B
Design defects detection rules
C
Input: Refactoring efforts
Code changes amount approximation
Output: recorded refactorings
D
Input: source code + call graphs
Semantic measures
Output: semantic measures
E
Input: List of previous program versions
Similarity with good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

Fig. 2: Multi-objective search-based refactoring framework.

of two or more ROs. An LLR is an elementary refactoring consisting of just one basic RO (e.g., *Create Class*, *Delete Method*, *Add Field*). The weight w_i for each RO is an integer number in the range [1, 2, 3] depending on code fragment complexity, and on change impact. For a refactoring solution consisting of p ROs, the code changes score is computed as:

$$Code\_changes = \sum_{i=1}^{p} w_i \tag{2}$$

Table I shows how the code change score is calculated for each refactoring operation. As described in the table, to estimate the number of required code changes for a high level refactoring, our method considers the number of low level refactoring operations (atomic changes) needed to actually implement such a refactoring based on the Soot tool. For instance, to move a method m from a class c_1 to a class c_2, the required number of chance is calculated as follows: 1 add method with a weight $w_i = 1$, 1 delete method with a $w_i = 1$, n redirect method call with a $w_i = 2$, and n redirect field access with a $w_i = 2$ as described in Table I. Using appropriate static code analysis, Soot allows to easily calculate the value n, by capturing the number of field references/accesses from a method, the number of calls that should be redirected based on call graph), the number of return types and parameters of a method, as well as the control flow graph of a method, and so on.

3.2.3. Similarity with recorded code changes. We defined the following function to calculate the similarity score between a proposed refactoring operation and a recorded code change:

$$Sim\_refactoring\_history(RO) = \sum_{j=1}^{n} e_j \tag{3}$$

where n is the number of recorded refactoring operations applied to the system in the past, and ej is a refactoring weight that reflects the similarity between the suggested refactoring operation (RO) and the recorded refactoring operation j. The weight e_j is computed as follows: if the suggested and the recorded refactorings being compared are

Table I: High and low level refactoring operations and their associated change scores.

	Low level refactoring												
Hight level refactoring	Create class	Delete class	Add method	Delete method	Add field	Delete field	Redirect method call	Redirect field access	Add parameter	Remove parameter	Rename method	Rename field	Rename class
Weight w_i	**2**	**3**	**1**	**3**	**1**	**3**	**2**	**2**	**1**	**2**	**1**	**1**	**1**
Move method			1	1			n	n					
Move field					1	1		n					
Pull up field					1	1		n				n	
Pull up method			1	1			n	n			n		
Push down field					1	1		n				n	
Push down method			1	1			n	n			n		
Inline class		1											n
Extract method			1				n	n	n	n	n		
Extract class	1		n	n	n	n	n	n					
Move class	1	1					n	n					n
Extract interface	1	n	n	n	n	n	n	n					n

identical, e.g., *Move Method* between the same source and target classes, then weight $e_j = 2$. If the suggested and the recorded refactorings are similar, then $e_j = 1$. We consider two refactoring operations as similar if one of them is composed of the other or if their implementations are similar, using equivalent controlling parameters, i.e., the same code fragments, as described in Table II. Some complex refactoring operations, such as Extract Class can be composed of other refactoring operations such as *Move Method*, *Move Field*, *Create New Class*, etc., the weight $w_j = 1$. Otherwise, $w_j = 0$. More details about the similarity scores between refactoring operations can be found in [Ouni et al. 2013].

3.2.4. Semantics. To the best of our knowledge, there is no consensual way to investigate whether refactoring can preserve the design semantics of the original program. We formulate semantic coherence using a meta-model in which we describe the concepts from a perspective that helps in automating the refactoring recommendation task. The aim is to provide a terminology that will be used throughout this paper. Figure 3 shows the semantic-based refactoring meta-model. The class Refactoring represents the main entity in the meta-model. As mentioned earlier, we classify refactoring operations into two types: low-level ROs (LLR) and high-level ROs (HLR). A LLR is an elementary/basic program transformation for adding, removing, and renaming program elements (e.g., *Add Method*, *Remove Field*, *Add Relationship*). LLRs can be combined to perform more complex refactoring operations (HLRs) (e.g., *Move Method*, *Extract Class*). A HLR consists of a sequence of two or more LLRs or HLRs; for example, to perform *Extract Class* we need to *Create New Empty Class* and apply a set of *Move Method* and *Move Field* operations.

To apply a refactoring operation we need to specify which actors, i.e., code fragments, are involved in this refactoring and which roles they play when performing the refactoring operation. As illustrated in Figure 3, an actor can be a package, class, field,

Table II: Refactoring operations and their involved actors and roles.

Refactoring operation	Actors	Roles
Move method	class	source class, target class
	method	moved method
Move field	class	source class, target class
	field	moved field
Pull up field	class	source class, target class
	field	moved field
Pull up method	class	source class, target class
	method	moved method
Push down field	class	source class, target class
	field	moved field
Push down method	class	source class, target class
	method	moved method
Inline class	class	source class, target class
Extract method	class	source class, target class
	method	source method, new method
	statement	moved statements
Extract class	class	source class, new class
	field	moved fields
	method	moved methods
Move class	package	source package, target package
	class	moved class
Extract interface	class	source classes, new interface
	field	moved fields
	method	moved methods

method, parameter, statement, or variable. In Table II, we specify for each refactoring operation the involved actors and their roles.

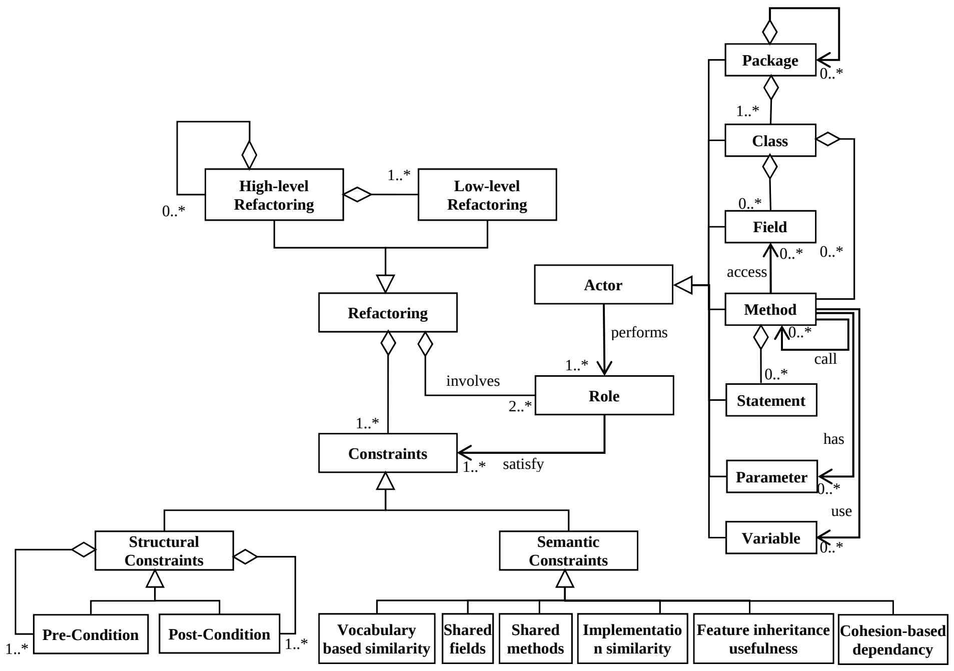


Fig. 3: Semantics-based refactoring meta-model.

3.3. Design coherence measures

3.3.1. Vocabulary-based similarity (VS). This kind of similarity is interesting to consider when moving methods, fields, or classes. For example, when a method has to be moved from one class to another, the refactoring would make sense if both actors (source class and target class) use similar vocabularies [Ouni et al. 2012b]. The vocabulary could be used as an indicator of the semantic/textual similarity between different actors that are involved when performing a refactoring operation. We start from the assumption that the vocabulary of an actor is borrowed from the domain terminology and therefore can be used to determine which part of the domain semantics an actor encodes. Thus, two actors are likely to be semantically similar if they use similar vocabularies.

The vocabulary can be extracted from the names of methods, fields, variables, parameters, types, etc. Tokenisation is performed using the Camel Case Splitter [Corazza et al. 2012], which is one of the most used techniques in Software Maintenance tools for the preprocessing of identifiers. A more pertinent vocabulary can also be extracted from comments, commit information, and documentation. We calculate the semantic similarity between actors using an information retrieval-based technique, namely cosine similarity, as shown in Equation 4. Each actor is represented as an n-dimensional vector, where each dimension corresponds to a vocabulary term. The cosine of the angle between two vectors is considered as an indicator of similarity. Using cosine similarity, the conceptual similarity between two actors c_1 and c_2 is determined as follows:

$$Sim(c_1, c_2) = Cos(\vec{c_1}, \vec{c_2}) = \frac{\vec{c_1} \cdot \vec{c_2}}{\|\vec{c_1}\| \times \|\vec{c_2}\|} = \frac{\sum_{i=1}^{n} w_{i,1} \times w_{i,2}}{\sqrt{\sum_{i=1}^{n} w_{i,1}^2} \times \sqrt{\sum_{i=1}^{n} w_{i,2}^2}} \tag{4}$$

where $\vec{c_1} = (w_{1,1}, ..., w_{n,1})$ is the term vector corresponding to actor c_1 and $\vec{c_2} = (w_{1,2}, ..., w_{n,2})$ is the term vector corresponding to c_2. The weights wi, j can be computed using information retrieval based techniques such as the Term Frequency – Inverse Term Frequency (TF-IDF) method. We used a method similar to that described in [Hamdi 2011] to determine the vocabulary and represent the actors as term vectors.

3.3.2. Dependency-based similarity (DS). We approximate domain semantics closeness between actors starting from their mutual dependencies. The intuition is that actors that are strongly connected (i.e., having dependency links) are semantically related. As a consequence, refactoring operations requiring semantic closeness between involved actors are likely to be successful when these actors are strongly connected. We consider two types of dependency links based on use the Jaccard similarity coefficient as the way you compute the similarity [Jaccard 1901]:

— **Shared Field Access (SFA)** that can be calculated by capturing all field references that occur using static analysis to identify dependencies based on field accesses (read or modify). We assume that two software elements are semantically related if they read or modify the same fields. The rate of shared fields (read or modified) between two actors c_1 and c_2 is calculated according to Equation 5. In this equation, $fieldRW(c_i)$ computes the number of fields that may be read or modified by each method of the actor c_i. Note that only direct field access is considered (indirect field accesses through other methods are not taken into account). By applying a suitable static program analysis to the whole method body, all field references that occur can be easily computed.

$$sharedFieldsRW(c_1, c_2) = \frac{|\ fieldRW(c_1) \cap fieldRW(c_2)\ |}{|\ fieldRW(c_1) \cup fieldRW(c_2)\ |} \tag{5}$$

— **Shared Method Calls (SMC)** that can be captured from call graphs derived from the whole program using CHA (Class Hierarchy Analysis) [Vallée-Rai et al. 2000]. A call graph is a directed graph which represents the different calls (call in and call out) among all methods of the entire program. Nodes represent methods, and edges represent calls between these methods. CHA is a basic call graph that considers class hierarchy information, e.g, for a call $c.m(...)$ assume that any $m(...)$ is reachable that is declared in a subtype or sometimes supertype of the declared type of c. For a pair of actors, shared calls are captured through this graph by identifying shared neighbours of nodes related to each actor. We consider both, shared call-out and shared call-in. Equations 6 and 7 are used to measure respectively the shared call-out and the shared call-in between two actors c_1 and c_2 (two classes, for example).

$$sharedCallOut(c_1, c_2) = \frac{\mid callOut(c_1) \cap callOut(c_2) \mid}{\mid callOut(c_1) \cup callOut(c_2) \mid} \tag{6}$$

$$sharedCallIn(c_1, c_2) = \frac{\mid callIn(c_1) \cap callIn(c_2) \mid}{\mid callIn(c_1) \cup callIn(c_2) \mid} \tag{7}$$

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

3.3.3. Implementation-based similarity (IS). For some refactorings like ***Pull Up Method***, methods having similar implementations in all subclasses of a super class should be moved to the super class [Fowler 1999]. The implementation similarity of the methods in the subclasses is investigated at two levels: signature level and body level. To compare the signatures of methods, a semantic comparison algorithm is applied. It considers the methods names, the parameter lists, and return types. Let $Sig(m_i)$ be the signature of method m_i. The signature similarity for two methods m_1 and m_2 is computed as follows:

$$Sig\_sim(m_1, m_2) = \frac{\mid Sig(m_1) \cap Sig(m_2) \mid}{\mid Sig(m_1) \cup Sig(m_2) \mid} \tag{8}$$

To compare method bodies, we use Soot [Vallée-Rai et al. 2000], a Java optimization framework, which compares the statements in the body, the used local variables, the exceptions handled, the call-outs, and the field references. Let $Body(m)$ (set of statements, local variables, exceptions, call-outs, and field references) be the body of method m. The body similarity for two methods m_1 and m_2 is computed as follows:

$$Body\_sim(m_1, m_2) = \frac{\mid Body(m_1) \cap Body(m_2) \mid}{\mid Body(m_1) \cup Body(m_2) \mid} \tag{9}$$

The implementation similarity between two methods is the average of their Sig_Sim and $Body\_Sim$ values.

3.3.4. Feature inheritance usefulness (FIU) . This factor is useful when applying the ***Push Down Method*** and ***Push Down Field*** operations. In general, when method or field is used by only few subclasses of a super class, it is better to move it, i.e., push it down, from the super class to the subclasses using it [Fowler 1999]. To do this for a method, we need to assess the usefulness of the method in the subclasses in which it appears. We use a call graph and consider polymorphic calls derived using XTA (Separate Type Analysis) [Tip and Palsberg 2000]. XTA is more precise than CHA by giving a more local view of what types are available. We are using Soot [Vallée-Rai et al. 2000] as a standalone tool to implement and test all the program analysis techniques required in our approach. The inheritance usefulness of a method is given by Equation 10:

$$FIU(m,c) = 1 - \frac{\sum_{i=1}^{n} call(m,i)}{n} \tag{10}$$

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

For the refactoring operation *Push Down Field*, a suitable field reference analysis is used. The inheritance usefulness of a field is given by Equation 11:

$$FIU(f,c) = 1 - \frac{\sum_{i=1}^{n} use(f,c_i)}{n} \tag{11}$$

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

3.3.5. Cohesion-based dependency (CD). We use a cohesion-based dependency measure for the *Extract Class* refactoring operation. The cohesion metric is typically one of the important metrics used to identify and fix design defects [Moha et al. 2010; Moha et al. 2008; Marinescu 2004; Bavota et al. 2011; Tsantalis and Chatzigeorgiou 2011]. However, the cohesion-based similarity that we propose for code refactoring, in particular when applying extract class refactoring, is defined to find a cohesive set of methods and attributes to be moved to the newly extracted class. A new class can be extracted from a source class by moving a set of strongly related (cohesive) fields and methods from the original class to the new class. Extracting this set will improve the cohesion of the original class and minimize the coupling with the new class. Applying the *Extract Class* refactoring operation on a specific class will result in this class being split into two classes. We need to calculate the semantic similarity between the elements in the original class to decide how to split the original class into two classes.

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

The cohesion matrix is obtained as follows: for the method-method similarity, we consider both vocabulary and dependency-based similarity. For the method-field similarity, if the method m_i may access (read or write) the field f_j, then the similarity value is 1. Otherwise, the similarity value is 0. The column “Average”" contains the average of similarity values for each line. The suitable set of methods and fields to be moved to a new class is obtained as follows: we consider the line with the highest average value and construct a set that consists of the elements in this line that have a similarity value that is higher than a threshold equals to 0.5. We used a trial and error strategy to find this suitable threshold value after executing our similarity measure more than 30 times.

Our decision to use such a technique is driven by the computation complexity since heavy and complex techniques might affect the whole search process. While cohesion is one of the strongest metrics which is already used in related work [Fokaefs et al. 2011; Bavota et al. 2014a; Bavota et al. 2011; Bavota et al. 2014b; Fokaefs et al. 2012; Bavota et al. 2010] for identifying extract class refactoring opportunities, we are planning to

Table III: Example of a cohesion matrix.

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

combine it with coupling metric, in order to reduce coupling between the extracted class and the original one.

4. NSGA-II FOR SOFTWARE REFACTORING

This section is dedicated to describing how we encoded the problem of finding a good refactoring sequence as an optimization problem using the non-dominated sorting genetic algorithm NSGA-II [Deb et al. 2002].

4.1. NSGA-II overview

One of the most powerful multi-objective search techniques is NSGA-II [Deb et al. 2002] that has shown good performance in solving several software engineering problems [Harman et al. 2012].

A high-level view of NSGA-II is depicted in Algorithm 1. NSGA-II starts by randomly creating an initial population P_0 of individuals encoded using a specific representation (line 1). Then, a child population Q_0 is generated from the population of parents P_0 (line 2) using genetic operators (crossover and mutation). Both populations are merged into an initial population R_0 of size N (line 5). *Fast-non-dominated-sort* [Deb et al. 2002] is the technique used by NSGA-II to classify individual solutions into different dominance levels (line 6). Indeed, the concept of non-dominance consists of comparing each solution x with every other solution in the population until it is dominated (or not) by one of them. According to Pareto optimality: "A solution x_1 is said to dominate another solution x_2, if x_1 is no worse than x_2 in all objectives and x_1 is strictly better than x_2 in at least one objective". Formally, if we consider a set of objectives f_i , $i \in 1..n$, to maximize, a solution x_1 dominates x_2 :

$$\textit{iff } \forall i,\ f_i(x_2) \leqslant f_i(x_1) \text{ and } \exists j \mid f_j(x_2) < f_j(x_1)$$

The whole population that contains N individuals (solutions) is sorted using the dominance principle into several fronts (line 6). Solutions on the first Pareto-front F_0 get assigned dominance level of 0 Then, after taking these solutions out, *fast-non-dominated-sort* calculates the Pareto-front F_1 of the remaining population; solutions on this second front get assigned dominance level of 1, and so on. The dominance level becomes the basis of selection of individual solutions for the next generation. Fronts are added successively until the parent population P_{t+1} is filled with N solutions (line 8). When NSGA-II has to cut off a front F_i and select a subset of individual solutions with the same dominance level, it relies on the crowding distance [Deb et al. 2002] to make the selection (line 9). This parameter is used to promote diversity within the population. The crowding distance of a non-dominated solution serves for getting an estimate of the density of solutions surrounding it in the population. It is calculated by the size of the largest cuboid enclosing each particle without including any other point. Hence, the crowding distance mechanism ensures the selection of diversified solutions having the same dominance level. The front F_i to be split, is sorted in descending order (line 13), and the first (N- $|P_{t+1}|$) elements of F_i are chosen (line 14). Then a new population Q_{t+1} is created using selection, crossover and mutation (line 15). This

Algorithm 1 High level pseudo code for NSGA-II

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

process will be repeated until reaching the last iteration according to stop criteria (line 4).

4.2. NSGA-II adaptation

This section describes how NSGA-II [Deb et al. 2002] can be used to find refactoring solutions with multiple conflicting objectives. To apply NSGA-II to a specific problem, the following elements have to be defined: representation of the individuals, creation of a population of individuals, evaluation of individuals using a fitness function for each objective to be optimized to determine a quantitative measure of 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, generation of a new population.

The next sections explain the adaptation of the design of these elements for the generation of refactoring solutions using NSGA-II.

4.2.1. Solution representation. To represent a candidate solution (individual), we used a vector representation. Each vector's dimension represents a refactoring operation. Thus, a solution is defined as a sequence of refactorings applied to different parts of the system to fix design defects. When created, the order of applying these refactorings corresponds to their positions in the vector. In addition, for each refactoring, a set of controlling parameters (stored in the vector), e.g., actors and roles, as illustrated in Table II, are randomly picked from the program to be refactored and stored in the same vector. An example of a solution is presented in Figure 4a.

Moreover, when creating a sequence of refactorings (an individual), it is important to guarantee that they are feasible and that they can be legally applied. The first work in the literature was proposed by [Opdyke 1992] who introduced a way of formalizing the preconditions that must be met before a refactoring can be applied and ensure that the behavior of the system is preserved. Opdyke created functions which could be used to formalize constraints. These constraints are similar to the Analysis Functions used later by [Cinnéide 2001] and [Roberts and Johnson 1999].

For each refactoring operation we specify a set of pre- and post-conditions to ensure the feasibility of applying them using a static analysis. For example, to apply the refactoring operation move method(`Person`, `Employee`, `getSalary()`), a number of necessary preconditions should be satisfied, e.g., `Person` and `Employee` should exists and should be classes; `getSalary()` should exist and should be a method; classes `Person` and `Employee` should not be in the same inheritance hierarchy; the method `getSalary()` should be implemented in `Person`; the method signature of `getSalary()` should not be present in class `Employee`. As postconditions, `Person`, `Employee`, and `getSalary()` should exist; `getSalary()` declaration should be in class `Employee`; and `getSalary()` declaration should not exist in class `Person`. Figure 4b describes for each refactoring operation its pre and post conditions that should be satisfied. To express these conditions we defined a set of functions. These functions include:

— `isClass(c)`: checks whether `c` is a class (similarly for `areClasses()`).
— `isInterface(c)`: checks whether `c` is an interface (similarly for `areInterfaces()`).
— `isMethod(m)`: checks whether `m` is a method.
— `Sig(m)`: returns the signature of the method `m`.
— `isField(f)`: checks whether `f` is a field.
— `defines(c,e)`: checks whether the code element `e` (method or field) is implemented in the class/interface `c`.
— `exists(e)`: checks whether the code element `e` exists in the current version of the code model (Similarly for `exist()`).
— `inheritanceHierarchy(c1,c2)`: checks whether both classes `c1` and `c2` belong to the same inheritance hierarchy.
— `isSuperClassOf(c1,c2)`: checks whether `c1` is a superclass of `c2`.
— `isSubClassOf(c1,c2)`: checks whether `c1` is a subclass of `c2`.
— `fields(c)`: returns the list of fields defined in the class or interface `c`.
— `methods(c)`: returns the list of methods implemented in class or interface `c`.

For composite refactorings, such as extract class and inline class, the overall pre and post conditions should be checked. For a sequence of refactorings which may be of any length, we simplify the computation of its full precondition by analyzing the precondition of each refactoring in the sequence and the corresponding effects on the code model (postconditions).

4.2.2. Fitness functions. After creating a solution, it should be evaluated using fitness function to ensure its ability to solve the problem under consideration. Since we have four objectives to optimize, we are using four different fitness functions to include in our NSGA-II adaptation. We used the four fitness functions described in the previous section:

(1) **Quality fitness function**. It aims at calculating the number of fixed design defects after applying the suggested refactorings.
(2) **Design coherence fitness function**. It aims at approximating the design preservation after applying the suggested refactorings. In Table IV, we specify, for each refactoring operation, which measures are taken into account to ensure that the refactoring operation preserves design coherence.
(3) **Code changes fitness function**. It calculates the amount of code changes required to apply the suggested refactorings.
(4) **History of changes fitness function**. It calculates the consistency of the suggested refactorings with prior code changes.

4.2.3. Selection. To guide the selection process, NSGA-II uses a binary tournament selection based on dominance and crowding distance [Deb et al. 2002]. NSGA-II sorts the population using the dominance principle which classifies individual solutions into different dominance levels. Then, to construct a new offspring population Q_{t+1}, NSGA-II uses a comparison operator based on a calculation of the crowding distance [Deb et al. 2002] to select potential individuals having the same dominance level.

move field (Person, Employee, salary)
extract method (Person,,printInfo(), printContactInfo())
move method (Person, Employee, getSalary())
push down field (Person, Student, studentId)
inline class (Car, Vehicle)
move method (Person, Employee, setSalary())
move field (Person, Employee, tax)
extract,class(Person, Adress, streetNo, city, zipCode, getAdress(), updateAdress())

(a) Solution representation.

Refactorings	**Pre and post-conditions**	
Move Method(c1,c2,m)	Pre:	exist(c1,c2, m) AND areClasses(c1,c2) AND isMethod(m) AND NOT(inheritanceHierarchy(c1,c2)) AND defines(c1,m) AND NOT(defines(c2,sig(m))
	Post:	exist(c1,c2,m) AND defines(c2,m) AND NOT(defines(c1,m))
Move Field(c1,c2,f)	Pre:	exist(c1, c2,f) AND areClasses(c1,c2) AND isField(f) AND NOT(inheritanceHierarchy(c1,c2)) AND defines(c1,f) AND NOT(defines(c2,f))
	Post:	exist(c1,c2, f) AND defines(c2,f) AND NOT(defines(c1,f))
Pull Up Field(c1,c2,f)	Pre:	exist(c1, c2,f) AND areClasses(c1, c2) AND isField(f) AND isSuperClassOf(c2,c1) AND defines(c1,f) AND NOT(defines(c2, f))
	Post:	exist(c1,c2,f) AND defines(c2,f) AND NOT(defines(c1,f))
Pull Up Method(c1,c2,m)	Pre:	exist(c1,c2,m) AND areClasses(c1,c2) AND isMethod(m) AND isSuperClassOf(c2,c1) AND defines(c1,m) AND NOT(defines(c2,sig(m))
	Post:	exist(c1,c2,m) AND defines(c2,m) AND NOT(defines(c1,m))
Push Down Field(c1,c2,f)	Pre:	exist(c1,c2, f) AND areClasses (c1,c2) AND isField(f) AND isSubClassOf(c2,c1) AND defines(c1,f) AND NOT(defines(c2,f))
	Post:	exist(c1) AND exists(c2) AND exits(m) AND defines(c2,m) AND NOT(defines(c1,m))
Push Down Method(c1,c2,m)	Pre:	exist(c1,c2,m) AND areClasses (c1,c2) AND isMethod(m) AND isSubClassOf(c2,c1) AND defines(c1,m) AND NOT(defines(c2,sig(m))
	Post:	exist(c1,c2,m) AND defines(c2,m) AND NOT(defines(c1,m))
Inline Class(c1,c2)	Pre:	exist(c1,c2) AND areClasses(c1,c2)
	Post:	exists(c1) AND NOT(exists(c2))
Extract Class(c1,c2)	Pre:	exists(c1) AND NOT(exists(c2)) AND isClass(c1) AND |methods(c1)|≥2
	Post:	exist(c1,c2) AND isClass(c2)
Extract Interface(c1,c2)	Pre:	exists(c1) AND NOT(exists(c2)) AND isInterface(c1) AND |methods(c1)|≥2
	Post:	exist(c1,c2) AND isInterface(c2)
Extract Super Class(c1,c2)	Pre:	exists(c1) AND NOT(exists(c2)) AND isClass(c1) AND |methods(c1)|≥2
	Post:	exist(c1,c2) AND isClass (c2) AND isSuperClass(c1,c2)
Extract Sub Class(c1,c2)	Pre:	exists(c1) AND NOT(exists(c2)) AND isClass(c1) AND |methods(c1)|≥2
	Post:	exist(c1,c2) AND isClass(c2) AND isSubClass(c1,c2)

(b) Pre- and post- conditions of refactorings.

Fig. 4: Representation of an NSGA-II individual and used constraints.

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

For *crossover*, we use a single, random, cut-point crossover. It starts by selecting and splitting at random two parent solutions. Then crossover creates two child solutions by putting, for the first child, the first part of the first parent with the second part of the second parent, and, for the second child, the first part of the second parent with the second part of the first parent. This operator must ensure that the length limits are respected by eliminating randomly some refactoring operations. As illustrated in Figure 5, crossover splits the parent solutions in the position $i = 3$ within their representative vectors in order to generate new child solutions. Each child combines some of the refactoring operations of the first parent with some ones of the second parent. In any given generation, each solution will be the parent in at most one crossover operation.

Table IV: Refactoring operations and their semantic measures.

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

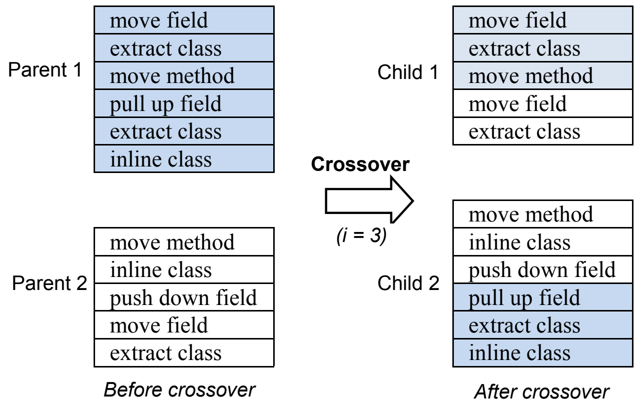


Fig. 5: Crossover operator.

The *mutation* operator picks randomly one or more operations from a sequence and replaces them with other ones from the initial list of possible refactorings. An example is shown in Figure 6 where a mutation operator is applied with two random positions to modify two dimensions of the vector in the third and the fifth dimensions ($j = 3$ and $k = 5$).

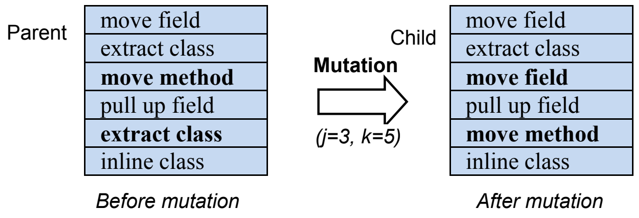


Fig. 6: Mutation operator.

After applying genetic operators (mutation and crossover), we verify the feasibility of the generated sequence of refactoring by checking the pre and post conditions. Each refactoring operation that is not feasible due to unsatisfied preconditions will be removed from the generated refactoring sequence. The new sequence is considered valid in our NSGA-II adaptation if the number of rejected refactorings is less than 5% of the total sequence size. We used trial and error to find this threshold value after several executions of our algorithm. The rejected refactorings will not be considered anymore in the solution.

5. VALIDATION AND EXPERIMENTATION DESIGN

In order to evaluate the feasibility and the efficiency of our approach for generating good refactoring suggestions, we conducted an experiment based on different versions of open-source systems. We start by presenting our research questions. Then, we describe and discuss the obtained results. All experimentation materials are available online[3].

5.1. Research questions

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

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

— **RQ1.1**: To what extent can the proposed approach fix different types of design defects?
— **RQ1.2**: To what extent does the proposed approach preserve design semantics when fixing defects?
— **RQ1.3**: To what extent can the proposed approach minimize code changes when fixing defects?
— **RQ1.4**: To what extent can the use of previously-applied refactorings improve the effectiveness of the proposed refactorings?
— **RQ2**: How does the proposed multi-objective approach based on NSGA-II perform compared to other existing search-based refactoring approaches and other search algorithms?
— **RQ3**: How does the proposed approach perform compared to existing approaches not based on heuristic search?
— **RQ4**: Is our multi-objective refactoring approach useful for software engineers in real-world setting?

To answer **RQ1.1**, we validate the proposed refactoring operations to fix design defects by calculating the defect correction ratio (DCR) on a benchmark composed of six open-source systems. DCR is given by Equation 1 which corresponds to the complement of the ratio of the number of design defects after refactoring (detected using bad smells detection rules) over the total number of defects that are detected before refactoring.

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

[3]http://www-personal.umd.umich.edu/~marouane/tosemref.html

Equation 12.

$$RP = \frac{\textit{\# coherent refactorings}}{\textit{\# suggested refactorings}} \in [0,1] \quad (12)$$

For the automatic validation we compare the proposed refactorings with the expected ones using an existing benchmark [Ouni et al. 2012a; Moha et al. 2010; Moha et al. 2008] in terms of recall (Equation 13) and precision (Equation 14). The expected refactorings are those applied by the software development team to the next software release. To collect these expected refactorings, we use Ref-Finder [Prete et al. 2010], an Eclipse plug-in designed to detect refactorings between two program versions. Ref-Finder allows us to detect the list of refactorings applied to the current version of a system (see Table VI).

$$RE_{recall} = \frac{|\textit{ suggested refactorings} \cap \textit{expected refactorings }|}{|\textit{ expected refactorings }|} \in [0,1] \quad (13)$$

$$RE_{precision} = \frac{|\textit{ suggested refactorings} \cap \textit{expected refactorings }|}{|\textit{ suggested refactorings }|} \in [0,1] \quad (14)$$

The intuition behind this metric is to assess whether the suggested refactorings are similar to the ones that a programmer would expect and perform.

To answer **RQ1.3**, we evaluate, using our benchmark, if the proposed refactorings are useful to fix detected defects with low code changes by calculating the code change score. The code change score is calculated using our model described in Section 3.2.2. We then compare the obtained code change scores with and without integrating the code change minimization objective in our tool.

To answer **RQ1.4**, we use the metric RP to evaluate the usefulness of the recorded refactorings and their impact on the quality of the suggested refactorings in terms of design coherence (RP). Consequently, we compare the obtained code RP scores with and without integrating the reuse of recorded refactorings in our tool. In addition, in order to evaluate the importance of reusing recorded refactorings in similar contexts, we define the metric “reused refactoring" (RR) that calculates the percentage of operations from the base of recorded refactorings used to generate the optimal refactoring solution by our proposal. RR is given by Equation 15.

$$RR = \frac{\textit{\#used refactorings from the base of recorded refactorings}}{\textit{\#refactorings in the base of recorded refactorings}} \in [0,1] \quad (15)$$

To answer **RQ2**, we compared our approach to two other existing search-based refactoring approaches: (*i*) Kessentini et al. [Kessentini et al. 2011], and (*ii*) Harman et al. [Harman and Tratt 2007] that consider the refactoring suggestion task only from the quality improvement perspective. Kessentini et al. formulated refactoring suggestion as a single objective problem to reduce as much as possible the number design defects, while Harman et al. formulated refactoring recommendation as multi-objective to find a trade-off between two quality metrics, CBO (coupling between objects) and SDMPC (standard deviation of methods per class). Both approaches [Kessentini et al. 2011] [Harman and Tratt 2007] did not consider the design coherence, the history of changes and the required effort when suggesting refactorings. Moreover, we assessed the performance of our multi-objective algorithm NSGA-II compared to another multi-objective algorithm (*i*) MOGA, (*ii*) random search, and (*ii*) mono-objective genetic algorithm (GA) where one fitness function is used (an average of the four objective functions).

To answer **RQ3**, we compared our refactoring results with a popular design defects detection and correction tool JDeodorant [Fokaefs et al. 2011; Fokaefs et al. 2012] that does not use heuristic search techniques in terms of DCR, change score and RP. The current version of JDeodorant [Fokaefs et al. 2012] is implemented as an Eclipse plug-in that identifies some types of design defects using quality metrics and then proposes a list of refactoring strategies to fix them.

To answer **RQ4**, we asked 6 software engineers (2 groups of 3 developers each) to refactor manually some of the design defects, and then compare the results with those proposed by our tool. We, thus, define the following precision metric:

$$Precision = \frac{\mid R_t \cap R_m \mid}{R_m} \in [0, 1] \tag{16}$$

where R_t is the set of refactorings suggested by our tool, and R_m is the set of refactorings suggested manually by software engineers. We calculated an exact matching score when comparing between the parameters (i.e., actors as described in Table II) of the refactoring suggested by our approach and the ones identified by developers. However, we do not consider the order of the parameters in the comparison formula.

5.2. Experimental setting and instrumentation

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

5.2.1. Subjects. Our study involved a total number of 24 subjects divided into 8 groups (3 subjects each). All the subjects are volunteers and familiar with Java development. The experience of these subjects on Java programming ranged from 2 to 15 years. The participants who evaluated the open source systems have a good knowledge about these systems and they did similar experiments in the past on the same systems. We selected also the groups based on their familiarity with the studied systems.

The first six groups are drawn from several diverse affiliations: the University of Michigan (USA), University of Montreal (Canada), Missouri University of Science and Technology (USA), University of Sousse (Tunisia) and a software development and web design company. The groups include 4 undergraduate students, 7 master students, 8 PhD students, one faculty member, and 4 junior software developers. The three master students are working also at General Motors as senior software engineers. Subjects were familiar with the practice of refactoring.

5.2.2. Systems studied and data collection. We applied our approach to a set of six well-known and well-commented industrial open source Java projects: Xerces-J[4], JFreeChart[5], GanttProject[6], Apache Ant[7], JHotDraw[8], and Rhino[9]. Xerces-J is a family of software packages for parsing XML. JFreeChart is a powerful and flexible Java library for generating charts. GanttProject is a cross-platform tool for project scheduling. Apache Ant is a build tool and library specifically conceived for Java applications. JHotDraw is a GUI framework for drawing editors. Finally, Rhino is a JavaScript interpreter and compiler written in Java and developed for the Mozilla/Firefox browser. We selected these systems for our validation because they range from medium to large-sized open-source projects, which have been actively developed over the past 10 years,

[4]http://xerces.apache.org/xerces-j/
[5]http://www.jfree.org/jfreechart/
[6]www.ganttproject.biz
[7]http://ant.apache.org/
[8]http://www.jhotdraw.org/
[9]http://www.mozilla.org/rhino/

Table V: Programs statistics

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

Table VI: Analysed versions and refactorings collection

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

and their design has not been responsible for a slowdown of their developments. Table V provides some descriptive statistics about these six programs.

To collect refactorings applied in previous program versions, and the expected refactorings applied to next version of studied systems, we use Ref-Finder [Prete et al. 2010]. Ref-Finder, implemented as an Eclipse plug-in, can identify refactoring operations applied between two releases of a software system. Table VI reports the analyzed versions and the number of refactoring operations, identified by Ref-Finder, between each subsequent couple of analyzed versions, after the manual validation. In our study, we consider only refactoring types described in Table II.

5.2.3. Scenarios. We designed the study to answer our research questions. Our experimental study consists of two main scenarios: (1) the first scenario is to evaluate the quality of the suggested refactoring solutions with potential users (RQ1-3), and (2) the second scenario is to fix manually a set of design defects and compare the manual results with those proposed by our tool (RQ4). All the recommended refactorings are executed using the Eclipse platform.

All the software engineers who accepted an invitation to participate in the study, received a questionnaire, a manuscript guide that helps to fill the questionnaire, and the source code of the studied systems, in order to evaluate the relevance of the suggested refactorings to fix. The questionnaire is organized in an excel file with hyperlinks to visualize the source code of the affected code elements easily. The participants were able to edit and navigate the code through Eclipse.

Scenario 1: The groups of subjects were invited to fill a questionnaire that aims to evaluate our suggested refactorings. The questionnaires rely on a four-point Likert scale [Likert 1932] in which we offered a choice of pre-coded responses for every question with no ‘neutral’ option. Thereafter, we assigned to each group a set of refactoring solutions suggested by our tool to evaluate manually. The participants were able to edit and navigate the code through the Eclipse IDE. Table VII describes the set of refactoring solutions to be evaluated for each studied system in order to answer our research questions. We have three multi-objective algorithms to be tested

Table VII: Refactoring solutions for each studied system considering each objective: quality (Q), Semantic coherence (S), Code changes (CC) , Recorded refactorings (RR), CBO (Coupling Between Objects) and SDMPC (Standard Deviation of Methods Per Class).

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

for the refactoring suggestion task: NSGA-II (Non-dominated Sorting Genetic Algorithm) [Deb et al. 2002], MOGA (Multi-Objective Genetic Algorithm) [Fonseca et al. 1993], and RS (Random Search) [Zitzler and Thiele 1998]. Moreover, we compared our results with a mono-objective genetic algorithm (GA) to assess the need for a multi-objective formulation. In addition, two refactoring solutions of both state-of-the art works (Kessentini et al [Kessentini et al. 2011] and Harman et al. [Harman and Tratt 2007]) are empirically evaluated in order to compare them to our approach in terms of design coherence.

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

Subjects (groups A-F) were aware that they are going to evaluate the design coherence of refactoring operations, but do not know the particular experiment research questions (algorithms used, different objectives used and their combinations). Consequently, each group of subjects who accepted to participate to the study, received a questionnaire, a manuscript guide to help them to fill the questionnaire, and the source code of the studied systems, in order to evaluate 6 solutions (10 refactorings per solution). The questionnaire is organized within a spreadsheet with hyperlinks to visualize easily the source code of the affected code elements. Subjects are invited to select for each refactoring operation one of the possibilities: "*Yes*" (coherent change), "*No*" (non-coherent change), or "*May be*" (if not sure). All the study material is available in [Deb et al. 2002]. Since the application of refactorings to fix design defects is a subjective process, it is normal that not all the programmers have the same opinion. In our case, we considered the majority of votes to determine if a suggested refactoring is correct or not.

Scenario 2: The aim of this scenario is to compare our refactoring results for fixing design defects suggested by our tool with manual refactorings identified by developers. Thereafter, we asked two groups of subjects (groups G and H) to fix a set of 72 design defect instances that are randomly selected from each subject system (12 defects per system) covering all the six different defect types considered. Then we compared their sequences of refactorings that are suggested manually with those proposed by our approach. The more our refactorings are similar to the manual ones, the more our tool is assessed to be useful and efficient in practice.

Table VIII: Survey organization.

Scenarios	Subject groups	Systems	Algorithm / Approach	Solutions
Scenario 1	Group A	GanttProject	NSGA-II	Solution 1
			Genetic Algorithm	Solution 4
		Xerces	MOGA	Solution 2
			Harman et al.	Solution 6
		JFreeChart	RS	Solution 3
			Kessentini et al.	Solution 5
	Group B	GanttProject	MOGA	Solution 2
			Harman et al.	Solution 6
		Xerces	RS	Solution 3
			Kessentini et al.	Solution 5
		JFreeChart	NSGA-II	Solution 1
			Genetic Algorithm	Solution 4
	Group C	GanttProject	RS	Solution 3
			Kessentini et al.	Solution 5
		Xerces	NSGA-II	Solution 1
			Genetic Algorithm	Solution 4
		JFreeChart	MOGA	Solution 2
			Harman et al.	Solution 6
	Group D	ApacheAnt	NSGA-II	Solution 1
			Genetic Algorithm	Solution 4
		JHotDraw	MOGA	Solution 2
			Harman et al.	Solution 6
		Rhino	RS	Solution 3
			Kessentini et al.	Solution 5
	Group E	ApacheAnt	MOGA	Solution 2
			Harman et al.	Solution 6
		JHotDraw	RS,	Solution 3
			Kessentini et al.	Solution 5
		Rhino	NSGA-II	Solution 1
			Genetic Algorithm	Solution 4
	Group F	ApacheAnt	RS	Solution 3
			Kessentini et al.	Solution 5
		JHotDraw	NSGA-II	Solution 1
			Genetic Algorithm	Solution 5
		Rhino	MOGA,	Solution 2
			Harman et al.	Solution 6
Scenario 2	Group G	All systems	Manual correction of design defects	N.A.
	Group H	All systems	Manual correction of design defects	N.A.

5.2.4. Algorithms configuration. In our experiments, we use and compare different mono and multi-objective algorithms. For each algorithm, to generate an initial population, we start by defining the maximum vector length (maximum number of operations per solution). The vector length is proportional to the number of refactorings that are considered, the size of the program to be refactored, and the number of detected design defects. A higher number of operations in a solution does not necessarily mean that the results will be better. Ideally, a small number of operations should be sufficient to provide a good trade-off between the fitness functions. This parameter can be specified by the user or derived randomly from the sizes of the program and the employed refactoring list. During the creation, the solutions have random sizes inside the allowed range. For all algorithms NSGA-II, MOGA, Random search (RS), and genetic algorithm (GA), we fixed the maximum vector length to 700 refactorings, and the population size to 200 individuals (refactoring solutions), and the maximum number of iterations to 6,000 iterations. We also designed our NSGA-II adaptation to be flexible in a way that we can configure the number of objectives and which objectives to consider in the execution.

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

Another element that should be considered when comparing the results of the four algorithms is that NSGA-II does not produce a single solution like GA, but a set of optimal solutions (non-dominated solutions). The maintainer can choose a solution from them depending on their preferences in terms of compromise. However, at least for our evaluation, we need to select only one solution. Thereafter, and in order to fully automate our approach, we proposed to extract and suggest only one best solution from the returned set of solutions. In our case, the ideal solution has the best value of quality (equal to 1), of design coherence (equal to 1), and of refactoring reuse (equal to 1), and code changes (normalized value equal to 1). Hence, we select the nearest solution to the ideal one in terms of Euclidian distance, as described in [Ouni et al. 2012b].

5.2.5. Inferential Statistical Test Methods Used. Our approach is stochastic by nature, i.e., two different executions of the same algorithm with the same parameters on the same systems generally leads to different sets of suggested refactorings. For this reason, our experimental study is performed based on 31 independent simulation runs for each problem instance, and the obtained results are statistically analyzed by using the Wilcoxon rank sum test with a 95% confidence level ($\alpha = 0.05$). The Wilcoxon signed-rank test is a non-parametric statistical hypothesis test used when comparing two related samples to verify whether their population mean-ranks differ or not. In this way, we could decide whether the difference in performance between our approach and the other detection algorithms is statistically significant or just a random result.

The Wilcoxon rank sum test allows verifying whether the results are statistically different or not. However, it does not give any idea about the difference magnitude. We, thus, investigate the effect size using the Cliff's Delta statistic [Cliff 1993]. The effect size is considered: (1) negligible if $\mid d \mid < 0.147$, (2) small if $0.147 \leq \mid d \mid < 0.33$, (3) medium if $0.33 \leq \mid d \mid < 0.474$, or (4) large if $\mid d \mid \geq 0.474$.

5.3. Empirical study results

This section reports the results of our empirical study, which are further discussed in the next sections. We first start by answering our research questions. We use two different validations: manual and automatic validations to evaluate the efficiency of the proposed refactorings.

Results for RQ1.1: As described in Table IX, after applying the proposed refactoring operations by our approach (NSGA-II), we found that, on average, 84% of the detected defects were fixed (DCR) for all the six studied systems. This high score is considered significant in terms of improving the quality of the refactored systems by fixing the majority of defects of various types (blob, spaghetti code, functional decomposition, data class, shotgun surgery, and future envy [Fenton and Pfleeger 1998; Kessentini et al. 2011]). For the different systems, the total number of refactorings generated by our approach was between 91 and 119 as described in the refactoring precision (RP) column of Table IX. Furthermore, we assessed the required time to implement the suggested refactorings on three systems. The average time required by 6 of the participants in our experiments to implement all the suggested refactorings was 11.5 hours per developer for each system, including the time required to understand and inspect the code before and after applying the refactorings. We believe that this required time is quite acceptable comparing to the time that the developer may spend to identify these refactoring opportunities manually from hundreds or thousands of classes and millions of lines of code. In addition, while the effect of refactoring is clearly translated by fixing the vast majority of design defects (84%) and significantly improving quality factors (see Section 6.1), other effects on the systems quality (maintainability, extendibility, etc.) cannot be assessed immediately.

Results for RQ1.2: To answer RQ1.2, we evaluated the correctness/meaningfulness of the suggested refactorings from the developers' point of view. We reported the results of our empirical evaluation in Table IX (RP column) related to Scenario 1. On average, for all of our six studied systems, 80% of proposed refactoring operations are considered by potential users to be semantically meaningful and do not generate design incoherence. We also automatically evaluated our approach. Thus, we compared the proposed refactorings with the expected ones. The expected refactorings are those applied by the software development team for the next software release as described in Table VI. We used Ref-Finder [Prete et al. 2010] to identify refactoring operations that are applied between the program version under analysis and the next version. Table IX (RP-automatic column) summarizes our results. We found that a considerable number of proposed refactorings (an average of 36% for all studied systems in terms of recall) were already applied to the next version by a software development team. Of course, this precision score is low because that not all refactorings applied to next version are related to quality improvement, but also to add new functionalities, increase security, fix bugs, etc. Moreover, the obtained results provide evidence that our approach is relatively stable through different executions as the standard deviation is still less than 3.23 in terms of DCR, 3.09 in terms of RP-automatic and 123.3 in terms of code changes[10].

To conclude, we found that our approach produces good refactoring suggestions in terms of defect-correction ratio, design coherence from the point of view of (1) potential users of our refactoring tool and (2) expected refactorings applied to the next program version.

Results for RQ1.3 and RQ1.4: To answer these two research questions, we need to compare different objective combinations (two, three, or four objectives) to ensure the efficiency and the impact of using each of the objectives we defined. We executed the NSGA-II algorithm with different combinations of objectives: maximize quality (Q), minimize design incoherence (S), minimize code changes (CC), and maximize the reuse of recorded refactorings (RR) as presented in Table X and Figure 7.

To answer RQ1.3, we present in Figure 7a and Table X, the code change scores obtained when the CC objective is considered (Q+S+RC+CC). We found that our approach succeeded in suggesting refactoring solutions that do not require high code changes (an average of only 2,937) with a relatively stable standard deviation of while having more than 3,888 as a code change score when the CC objective is not considered in the other combinations. At the same time, we found that the DCR score (Figure 7c) is not significantly affected with and without considering the CC objective.

To answer RQ1.4, we present the obtained results in Figure 7b. The best RP scores are obtained when the recorded code changes (RC) are considered (Q+S+RC), while

[10]Note that only for the RP metric, we did not report the standard deviation as we directly conducted the qualitative evaluation with subjects on the suggested refactoring solution having the median DCR score from 31 independent runs.

Table IX: Empirical study results on 31 runs (Median & STDev). The results was statistically significant on 31 independent runs using the Wilcoxon rank sum test with a 95% confidence level ($p - value < 0.05$) in terms of defect correction ratio (DCR), code changes score, refactoring precision (RP), and RP-automatic.

Systems	**Approach**	**DCR**		**Code changes**		**RP**	**RP-automatic**	
		Median	**STDev**	**Median**	**STDev**		**Median**	**STDev**
Xerces	NSGA-II	83% (76|91)	1.58	3,843	123.3	81 % (74|91)	26% (10|39)	2.09
	Harman et al. '07	N.A	N.A	2,669	78.4	41 %	8 % (3|39)	2.09
	Kessentini et al. '11	89% (81/91)	2.24	4,998	102.8	37 %	13% (5|39)	2.97
JFreeChart	NSGA-II	86% (62|72)	2.44	2,016	89.8	82 % (87|106)	35% (11|31)	2.98
	Harman et al. '07	N.A	N.A	3,269	86.2	36 %	0 % (0|31)	1.51
	Kessentini et al. '11	90% (65|72)	2.86	3,389	85.82	37 %	13% (4|31)	2.65
GanttProject	NSGA-II	85% (42|49)	3.23	2,826	73.82	80 % (63|78)	46% (21|46)	2.27
	Harman et al. '07	N.A	N.A	4,790	83.72	23 %	0% (0|46)	1.01
	Kessentini et al. '11	95% (47|49)	2.96	4,697	86.7	27 %	15% (7|46)	2.45
ApacheAnt	NSGA-II	78% (87|112)	1.18	4,690	112.9	78 % (93|119)	31% (24|78)	2.22
	Harman et al. '07	N.A	N.A	6,987	77.63	40 %	04% (3|78)	0.96
	Kessentini et al. '11	80% (90|112)	1.89	6,797	83.1	30 %	0% (0|78)	1.7
JHotDraw	NSGA-II	84% (21|25)	3.21	2,231	97.65	80 % (79|98)	44% (18|41)	3.09
	Harman et al. '07	N.A	N.A	3,654	77.63	37 %	10% (4|41)	2.69
	Kessentini et al. '11	84% (21|25)	5.32	3,875	90.83	43 %	7% (3|41)	2.73
Rhino	NSGA-II	85% (59|69)	2.69	1,914	89.77	80 % (90|112)	33% (15|46)	2.91
	Harman et al. '07	N.A	N.A	2,698	77.63	37 %	0% (0|46)	1.003
	Kessentini et al. '11	87% (60|69)		3,365	77.61	32 %	9% (4|46)	2.97
Average (all systems)	NSGA-II	84%		2,937		80 %	36%	
	Harman et al. '07	N.A		4,011		36 %	4%	
	Kessentini et al. '11	89%		4,520		34 %	9%	

having good correction ration DCR (Figure 7c). In addition, we need more quantitative evaluation to investigate the effect of the use of recorded refactorings, on the design coherence (RP). To this end, we compare the RP score with and without using recorded refactorings. In most of the systems when recorded refactoring is combined with semantics, the RP value is improved. For example, for Apache Ant RP is 83% when only quality and semantics are considered, however, when recorded refactoring reuse is included the RP is improved to 87% (Figure 7b).

We notice also that when code changes reduction is included with quality, semantics and recorded changes, the RP and DCR scores are not significantly affected. Moreover, we notice in Figure 7c that there is no significant variation in terms of DCR with all

Table X: Median refactoring results and standard deviation (STDev) of different objective combinations with NSGA-II (average of all the systems) on 31 runs in terms of defect correction ratio (DCR), refactoring precision (RP), code changes reduction and recorded refactorings (RR). The results was statistically significant on 31 independent runs using the Wilcoxon rank sum test with a 95% confidence level ($p - value < 0.05$).

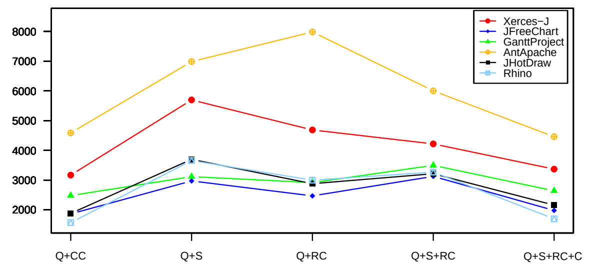
Objectives combinations	**DCR**		**Code changes**		**RR**		**RP (empirical evaluation)**
	Median	**STDev**	**Median**	**STDev**	**Median**	**STDev**	
Q + CC	75%	1.84	2591	87.12	N.A.	N.A.	45%
Q + S	81%	1.93	4355	94.6	N.A.	N.A.	82%
Q + RC	85%	2.16	3989	89.76	41%	2.87	54%
Q + S + RC	81%	1.56	3888	106.24	35%	3.21	84%
Q + S + RC + CC	84%	2.39	2917	97.91	36%	3.82	80%

different objectives combinations. When four objectives are combined the DCR value induces a slight degradation with an average of 82% in all systems which is even considered as promising results. Thus, the slight loss in the defect-correction ratio is largely compensated by the significant improvement of the design coherence and code changes reduction. Moreover, we found that the optimal refactoring solutions found by our approach are obtained with a considerable percentage of reused refactoring history (RR) (more than 35% as shown in Table X). Thus, the obtained results support the claim that recorded refactorings applied in the past are useful to generate coherent and meaningful refactoring solutions and can effectively drive the refactoring suggestion task.

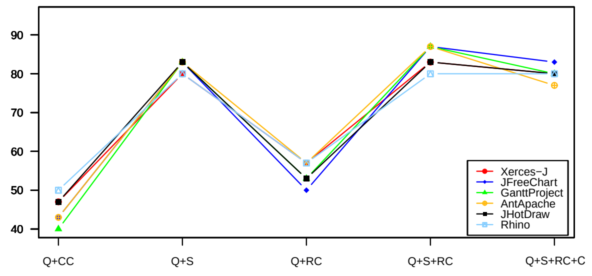
In conclusion, we found that the best compromise is obtained between the four objectives using NSGA-II comparing to the use of only two or three objectives. By default, the tool considers the four objectives to find refactoring solutions. Thus, a software engineer can consider the multi-objective algorithm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

Results for RQ2: To answer RQ2, we evaluate the efficiency of our approach comparing to two other contributions of Harman et al. [Harman and Tratt 2007] and Kessentini et al. [Kessentini et al. 2011]. In [Harman and Tratt 2007], Harman et al. proposed a multi-objective approach that uses two quality metrics to improve CBO (coupling between objects) and SDMPC (standard deviation of methods per class) after applying the refactorings sequence. In [Kessentini et al. 2011], a single-objective genetic algorithm is used to correct defects by finding the best refactoring sequence that reduces the number of defects. The comparison is performed in terms of: (1) defect correction ratio (DCR) that is calculated using defect detection rules, (2) refactoring precision (RP) that represents the results of the subject judgments (Scenario 1), and (3) code changes needed to apply the suggested refactorings. We adapted our technique for calculating code changes scores for both approaches Harman et al. and Kessentini et al. Table 8 summarizes our findings and reports the median values and standard deviation (STDev) of each of our evaluation metrics obtained for 31 simulation runs of all projects.

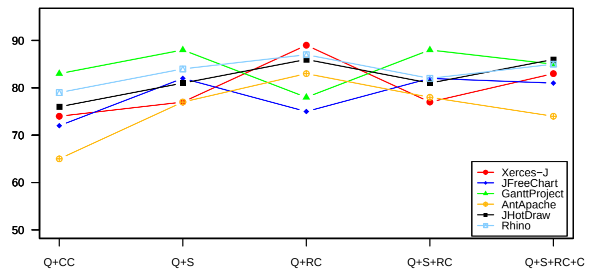
As described in Table IX, after applying the proposed refactoring operations, we found that more than 84% of detected defects were fixed (DCR) as an average for all the six studied systems. This score is comparable to the correction score of Kessentini et al. (89%), an approach that does not consider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



(a) Code Change score (CC).



(b) Refactoring Precision (RP).



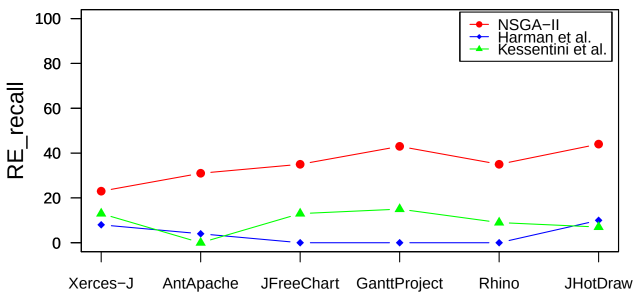
(c) Defect Correction Ratio (DCR).

Fig. 7: Refactoring results of different objectives combination with NSGA-II in terms of (a) code changes reduction (CC), (b) design preservation (RP), (c) defects correction ratio (DCR).

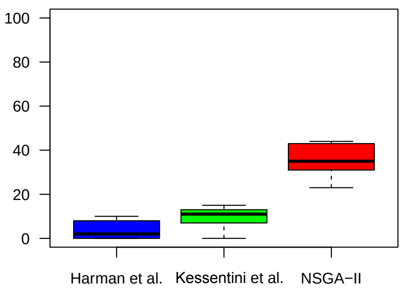
Regarding the semantic coherence, for all of our six studied systems, an average of 80% of proposed refactoring operations are considered as semantically feasible and do not generate design incoherence. This score is significantly higher than the scores of the two other approaches having respectively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

We compared the three approaches in terms of automatic RE_{recall}, as depicted in Figure 8. We found that a considerable number of proposed refactorings, an average of 36% for all studied systems in terms of recall, are already applied to the next version by the software development team. By comparison, the results for Harman et al. and Kessentini et al. are only 4% and 9% respectively, as reported in figure 8b. Moreover, this score shows that our approach is useful in practice unlike both other approaches. In fact, the RE_{recall} of Harman et al. is not significant, since only the move method refactoring is considered when searching for refactoring solutions to improve coupling and standard deviation of methods per class. Moreover, expected refactorings are not related only to quality improvement, but also for adding new functionalities, and other maintenance tasks. This is not considered in our approach when we search for the optimal refactoring solution that satisfies our four objectives. However, we manually inspected expected refactorings and we found that they are mainly related to adding new functionality (related to adding new packages, classes or methods).

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



(a) RE_recall results for each system.



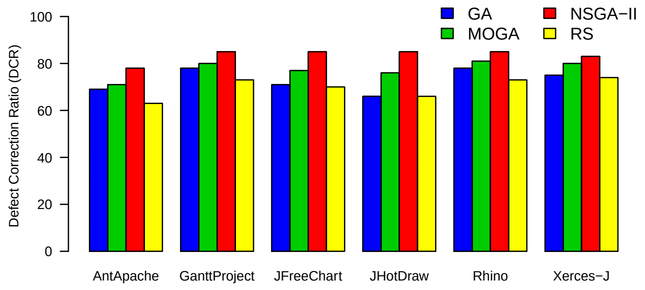
(b) Boxplots for RE_recall.

Fig. 8: Automatic refactoring score (RE_recall) comparison between our approach (NSGA-II), Harman et al. and Kessentini et al.

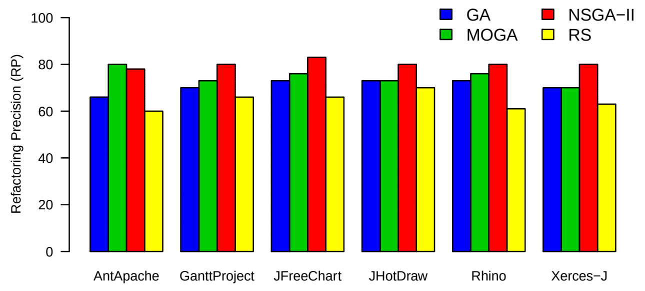
Furthermore, to justify the use of NSGA-II, we compared the performance of our proposal to two other multi-objective algorithms: MOGA, and a random search and a mono-objective algorithm (genetic algorithm). In a random search, the change operators (crossover and mutations) are not used, and populations are generated randomly and evaluated using the four objective functions. In our mono-objective adaptation, we considered a single fitness function, which is the normalized average score of the four objectives using a genetic algorithm. Moreover, since in our NSGA-II adaptation, we select a single solution without giving more importance to some objectives, we give equal weights for each fitness function value. As shown in Figure 9, NSGA-II outperforms significantly MOGA, random-search, and the mono-objective algorithm in terms of defects-correction ratio (DCR), semantic coherence preservation (RP), and code change reduction. For instance, in JFreeChart, NSGA-II performs much better than MOGA, random search, and genetic algorithm in terms of DCR and RP scores (respectively Figure 9a and Figure 9b). In addition, NSGA-II reduces significantly code changes for all studied systems. For example, for Rhino, the number of code changes was reduced to almost the half comparing to random search as shown in Figure 9c.

Furthermore, an interesting finding is that the random search (RS) works as well as the single-objective GA. Indeed, we used RS with a multi-objective version by switching off individual selection based on fitness value, in our original framework. The performance of RS is clearly less than the other multi-objective algorithms being compared (NSGA-II and MOGA). Some of the results of RS can be considered acceptable, this can be explained by the limited number of refactoring types considered in our experiments (limited search space). For GA, after 2,000 generations, we noticed that the search produced entire populations with high DCR and CC values but lower S and RR values that has resulted in a relative increase in the combined fitness function which led to comparable results to the multi-objective RS. The comparable results between RS and GA suggest that our formulation to the refactoring recommendation problem as a multi-objective formulation is adequate.

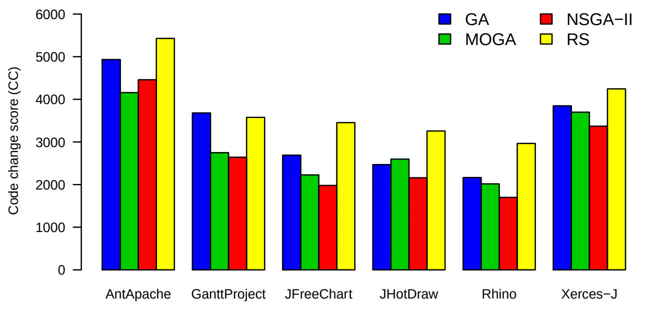
Another interesting observation from the results in figure 9 is that MOGA has less code changes and higher RP value than NSGA-II in Apache Ant. By looking at the produced results, we noticed that none of the blob design defects was fixed in Apache Ant using MOGA. Indeed, the blob design defect is known as one of the most difficult design defects to fix, and typically requires a large number of refactoring operations and code changes (several extract class, move method and move field refactorings). This is also explained by the higher RP score, as it also complicated for developers to approve such refactorings.



(a) Defect Correction Ratio (DCR).



(b) Refactoring Precision (RP).



(c) Code Change Score (CC).

Fig. 9: Refactoring results of different algorithms NSGA-II, MOGA, GA and RS in terms of (a) defects correction ratio, (b) refactoring precision and (c) code changes reduction.

For all experiments, we obtained a large difference between NSGA-II results and the mono-objective approaches (Harman et al., Kessentini et al., GA and random search) using all the evaluation metrics. However, when comparing NSGA-II against MOGA, we have found the following results: a) On small and medium-scale software systems (JFreeChart, Rhino and GanttProject) NSGA-II is better than MOGA on most systems with a small and medium effect size; b) On large-scale software systems (Xerces-J, Apache Ant and JDI-Ford), NSGA-II is better than MOGA on most systems with a high effect size.

Results for RQ3: JDeodorant uses only structural information to detect and fix design defects, but does not handle all the six design defect types that we considered in our experiments. Thus, to make the comparison fair, we performed our comparison using only two design defects that can be fixed by both tools: blob and feature envy. Figure 10 summarizes our findings for the blob (figure 10a) and feature envy (figure 10b). It is clear that our proposal outperforms JDeodorant, on average, on all the systems in terms of the number of fixed defects with a minimum number of changes and semantically coherent refactorings. The average number of fixed code smells is comparable between both tools. However, our approach is clearly better in terms of semantically coherent refactorings. This can be explained by the fact that JDeodorant uses only structural metrics to evaluate the impact of suggested refactorings on the detected code smells. In addition, our proposal supports more types of refactorings than JDeodorant and this is also explains our outperformance. However, one of the advantages of JDeodorant is that the suggested refactorings are easier to apply than those proposed by our technique as it provides an Eclipse plugin to suggest and then automatically apply a total of 4 types of refactorings, while the current version of our tool requires to apply refactorings by the developers using the Eclipse IDE with more complex types of refactorings.

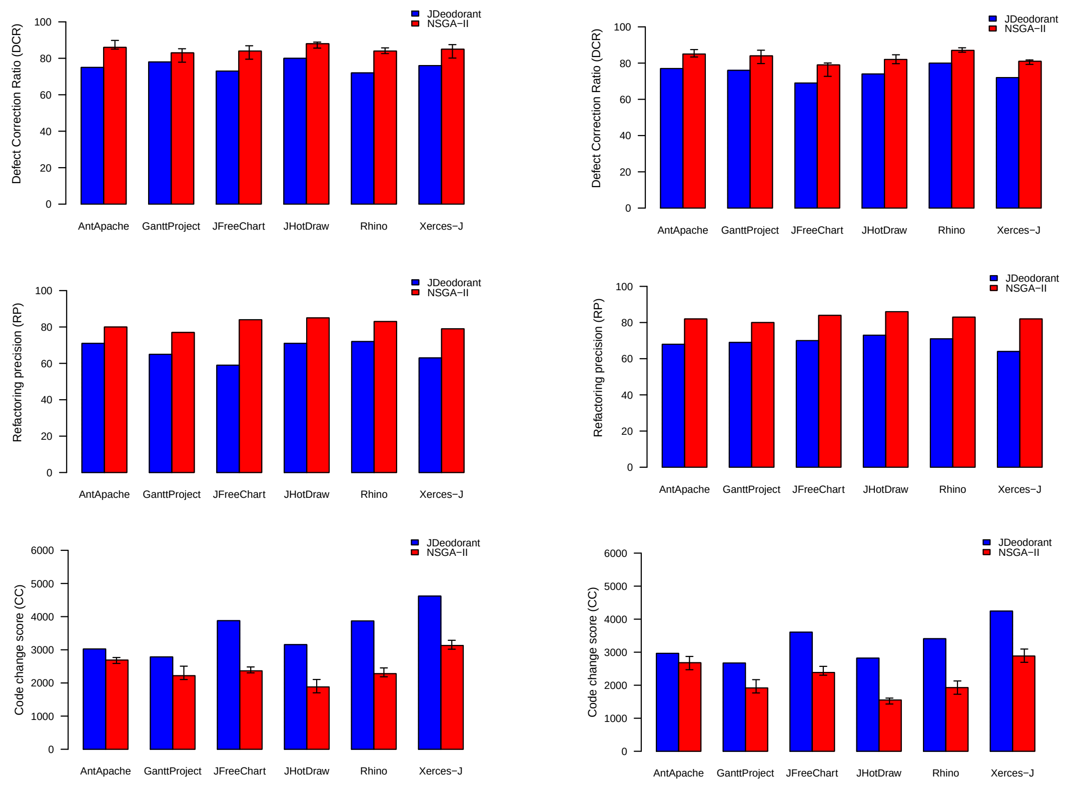
Results for RQ4: To evaluate the relevance of our suggested refactorings with our subjects, we compared the refactoring strategies proposed by our technique and those proposed manually by groups G and H (6 subjects) to fix several defects on the six systems. Figure 11 shows that most of the suggested refactorings by NSGA-II are similar to those applied by developers with an average of more than 73%. Some defects can be fixed by different refactoring strategies, and also the same solution can be expressed in different ways (complex and atomic refactorings). Thus we consider that the average precision of more than 73% confirms the efficiency of our tool for real developers to automate the refactoring recommendation process. We discuss, in the next section, in more detail the relevance of our automated refactoring approach for software engineers.

6. DISCUSSIONS

The obtained results from Section 5.3 suggest that our approach performs better than two existing approaches. We also compared different objective combinations and found that the best compromise is obtained between the four objectives using NSGA-II when compared to the use of only two or three objectives. Therefore, our four objectives are efficient for providing "good" refactoring suggestions. Moreover, we found that the results achieved by NSGA-II outperforms the ones achieved by both multi-objective algorithms, MOGA and random search, and the mono-objective algorithm, GA.

We now provide more quantitative and qualitative analyses of our results and discuss some observations drawn from our empirical evaluation of our refactoring approach. We aim at answering the following research questions:

— **RQ5:** What is the effect of suggested refactorings on the overall quality of systems?
— **RQ6:** What is the effect of multiple executions on the refactoring results?



(a) Blob. (b) Feature envy.

Fig. 10: Comparison results of our approach (NSGA-II) with JDeodorant in terms of defects correction ratio (DCR), design coherence (RP) and code changes score (CC) for each system. For NSGA-II, we report the average DCR and CC scores and standard deviation obtained through 31 independent runs. Note that for RP score, we did not report the standard deviation as we directly conducted the qualitative evaluation with subjects on the suggested refactoring solution that have the median DCR score.

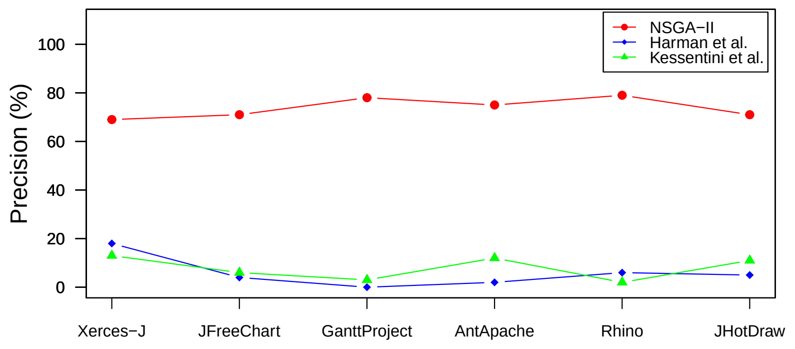


Fig. 11: Comparison of our refactoring results with manual refactorings in terms of Precision.

— **RQ7:** What is the distribution of the suggested refactoring types?

In the following subsections we answer each of these research questions.

6.1. The refactoring impact (RQ5)

Although our primary goal in this work is to demonstrate that design defects can be automatically refactored, it is also important to assess the refactoring impact on design quality. The expected benefit from refactoring is to enhance the overall software design quality as well as fixing design defects [Fowler 1999]. We use the QMOOD (Quality Model for Object-Oriented Design) model [Bansiya and Davis 2002] to estimate the effect of the suggested refactoring solutions on quality attributes. We choose QMOOD, mainly because 1) it is widely used in the literature [Shatnawi and Li 2011; O'Keeffe and Cinnéide 2008; Zibran and Roy 2011] to assess the effect of refactoring, and 2) it has the advantage of defining six high level design quality attributes (reusability, flexibility, understandability, functionality, extendibility and effectiveness) that can be calculated using 11 lower level design metrics [Bansiya and Davis 2002]. In our study we consider the following quality attributes:

— *Reusability:* The degree to which a software module or other work product can be used in more than one computer program or software system.
— *Flexibility:* The ease with which a system or component can be modified for use in applications or environments other than those for which it was specifically designed.
— *Understandability:* The properties of designs that enable it to be easily learned and comprehended. This directly relates to the complexity of design structure.
— *Effectiveness:* The degree to which a design is able to achieve the desired functionality and behavior using OO design concepts and techniques.

We did not assess the issue of functionality because we assume that, by definition, refactoring does not change the behavior/functionality of systems; instead, it changes the internal structure. We have also excluded the extendibility factor because it is, to some extent, a subjective quality factor and using a model of merely static measures to evaluate extendibility is inadequate. Tables XI and XII summarize the QMOOD formulation of these quality attributes [Bansiya and Davis 2002].

The improvement in quality can be assessed by comparing the quality before and after refactoring independently to the number of fixed design defects. Hence, the total gain in quality G for each of the considered QMOOD quality attributes qi before and after refactoring can be easily estimated as:

$$G_{q_i} = q'_i - q_i \tag{17}$$

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

In Figure 12, we show the obtained gain values (in terms of absolute value) that we calculated for each QMOOD quality attribute before and after refactoring for each studied system. We found that the systems quality increase across the four QMOOD quality factors much better than existing approaches. Understandability is the quality factor that has the highest gain value; whereas the Effectiveness quality factor has the lowest one. This mainly due to many reasons 1) the majority of fixed design defects (blob, spaghetti code) are known to increase the coupling (DCC) within classes, which heavily affect the quality index calculation of the Effectiveness factor; 2) the vast majority of suggested refactoring types were move method, move field, and extract class (Figure 12) that are known to have a high impact on coupling (DCC), cohesion (CAM) and the design size in classes (DSC) that serves to calculate the understandability

Table XI: QMOOD metrics for design properties.

Design Property	Metric	Description
Design size	DSC	Design size in classes
Complexity	NOM	Number of methods
Coupling	DCC	Direct class coupling
Polymorphism	NOP	Number of polymorphic methods
Hierarchies	NOH	Number of hierarchies
Cohesion	CAM	Cohesion among methods in class
Abstraction	ANA	Average number of ancestors
Encapsulation	DAM	Data access metric
Composition	MOA	Measure of aggregation
Inheritance	MFA	Measure of functional abstraction
Messaging	CIS	Class interface size

Table XII: QMOOD quality factors.

Quality attribute	Quality Index Calculation
Reusability	= -0.25 * DCC + 0.25 * CAM + 0.5 * CIS + 0.5 * DSC
Flexibility	= 0.25 * DAM - 0.25 * DCC + 0.5 * MOA +0.5 * NOP
Understandability	= -0.33 * ANA + 0.33 * DAM - 0.33 * DCC + 0.33 * = CAM -0.33 * NOP + 0.33 * NOM - 0.33 * DSC
Effectiveness	= 0.2 *ANA + 0.2 *DAM + 0.2*MOA + 0.2 * MFA + 0.2 *NOP

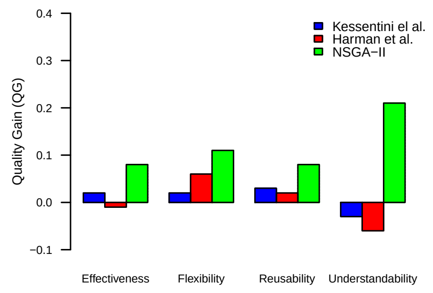
quality factor. Furthermore, we noticed that JHotDraw produced the lowest quality increase for the four quality factors. This is justified by the fact that JHotDraw is known to be of good design and implementation practices [Kessentini et al. 2010] and it contains a small number of design defects compared to the five other studied systems.

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

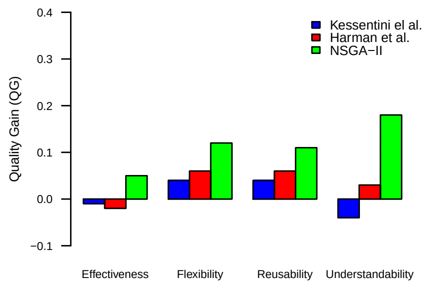
Finally, it is worth to notice that since the application of refactorings to fix design defects is a subjective process, it is normal that not all the programmers have the same opinion. Thus it is important to study the level of agreement between subjects. To address this issue, we evaluated the level of agreement using Cohen's Kappa coefficient κ [Cohen et al. 1960], which measures to what extent the subjects agree when voting for a recommended refactoring operation. The Kappa coefficient assessments was 0.78, which is characterized as "*substantial agreement*" by Landis and Koch [Landis and Koch 1977]. This obtained score makes us more confident that our suggested refactorings are meaningful from software engineer's perspective.

6.2. The effect of multiple executions (RQ6)

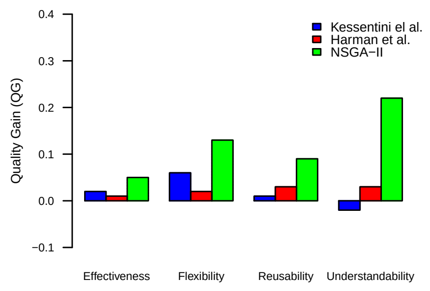
It is important to contrast the results of multiple executions with the execution time to evaluate the performance and the stability of our approach. The execution time for finding the optimal refactoring solution with a number of iterations (stopping criteria) fixed to 6,000 was less than forty-eight minutes as shown in Figure 13. Moreover, we evaluate the impact of the number of suggested refactorings on the DCR, RP, RR, and code change scores in five different executions. Drawn for JFreeChart, the results of figure 13 show that the number of suggested refactorings do not affect the refactoring



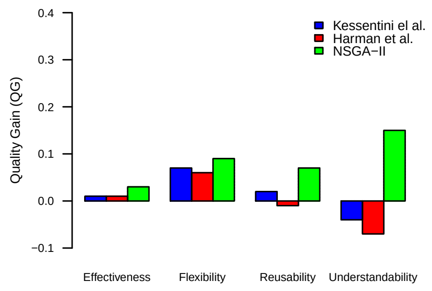
(a) Xerces-J.



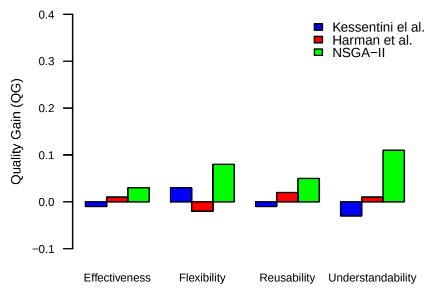
(b) JFreeChart.



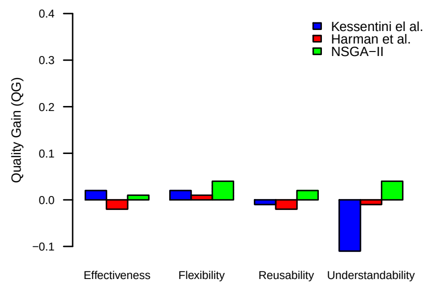
(c) GanttProject.



(d) AntApache.



(e) Rhino.



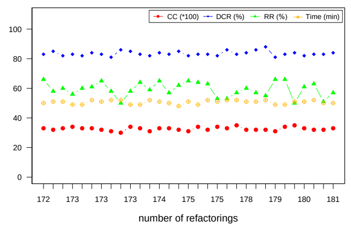
(f) JHotDraw.

Fig. 12: The impact of best refactoring solutions on QMOOD quality attributes.

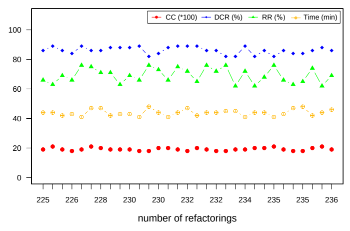
results. Thus, a higher number of operations in a solution does not necessarily mean that the results will be better. Consequently, we could conclude that our approach is scalable from the performance standpoint, especially that our technique is executed, in general, up front (at night) to find suitable refactorings. In addition, the results' accuracy is not affected by the number of suggested refactorings.

Furthermore, it is also important to assess the impact of the number of design defects on the size of the refactoring solution (number of refactorings). Figure 14 reports the correlation between the number of design defects and the number of refactorings for each system. Our findings confirm that the number of design defects does not affect the number of refactorings due to the low value of correlation (0.04).

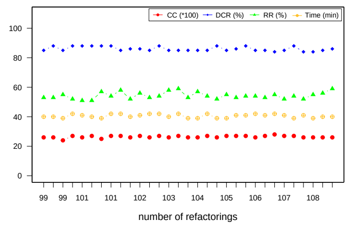
In addition, figure 15 reports the execution time for each of the search algorithms NSGA-II, Harman et al., Kessentini et al., MOGA, GA and RS. As shown in the figure, the execution time of our NSGA-II approach was very similar to MOGA with an average of less than 48 minutes per system. However, the execution time of random search



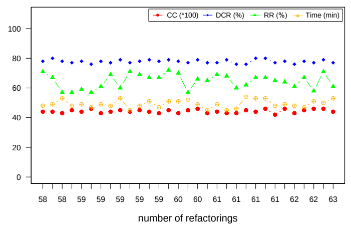
(a) Xerces-J.



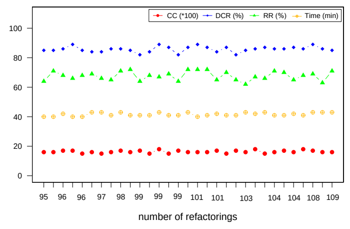
(b) JFreeChart.



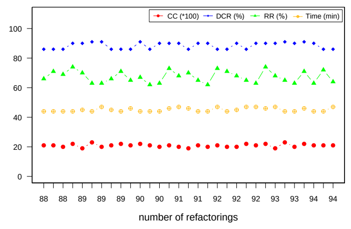
(c) GanttProject.



(d) AntApache.



(e) Rhino.



(f) JHotDraw.

Fig. 13: Results of multiple executions on different project in terms of defect correction ratio (DCR), code changes (CC), reused refactorings (RR), and execution time (Time).

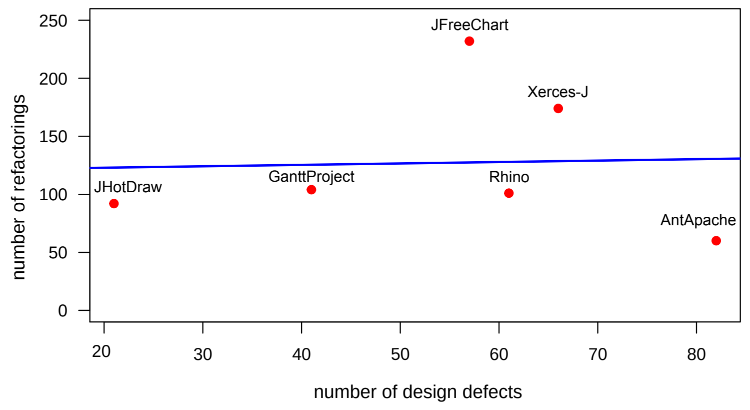


Fig. 14: Impact of the number of design defects on the size of the refactoring solution (number of refactorings).

was half of time spent by NSGA-II and MOGA, but the quality of the random search solutions are much lower. The performance of NSGA-II is slightly better than MOGA based on the different evaluation metrics. However, the adaptation of an NSGA-II algorithm to our refactoring problem is more complex than MOGA.It is expected that the execution time of the remaining mono-objective approach is almost half the NSGA-II one due to the following reasons: (1) they just considered one objective function, (2) the time consuming for semantics and history functions of our approach are not considered by existing mono-objective approaches which require additional time processing, filtering and comparing the identifiers within classes, and (3) existing mono-objective approaches are limited to few types of refactorings. Since our refactoring problem is not a real time one, the execution time of NSGA-II is considered acceptable by all the programmers of our experiments. In fact, they mentioned that it is not required to use the tool daily and they can execute it at the end of the day and check the results the next day.

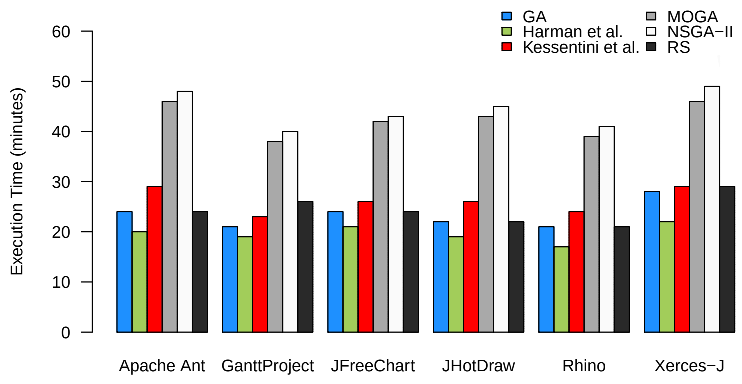


Fig. 15: Comparison of the execution time for each of the search techniques, NSGA-II, Harman et al., Kessentini et al., MOGA, GA and RS.

6.3. The distribution of suggested refactoring types (RQ7)

Another important consideration is the refactoring operations' distribution. We contrast that the most suggested refactorings are move method, move field, and extract class for the majority of studied systems except JHotDraw. For instance, in Xerces-J, we had different distribution of different refactoring types as illustrated in Figure 16. We notice that the most suggested refactorings are related to moving code elements (fields, methods) and extract/inline class. This is mainly due to the type of defects detected in Xerces-J (most of the defects are related to the blob defect) that need particular refactorings to move elements from blob class to other classes in order to reduce the number of functionalities from them. On the other hand, we found for JHotDraw less move method, move field, and extract class refactorings. This is mainly because JHotDraw contains a small number of blobs (only three blobs were detected), and it is known to be of good quality. Thus, our results in Figure 16 reveal an effect we found: refactorings like move field, move method, and extract class are likely to be more useful to correcting the blob defect. As part of future work, we plan to investigate the relationship between defect types and refactoring types.

7. INDUSTRIAL CASE STUDY

The goal of this study is to evaluate the efficiency of our refactoring tool in practice. We conducted an evaluation with potential software engineers, who can use our tool, related to the relevance of our approach for software engineers. One of the advantages of this industrial validation is the participation of the original developers of a system in the evaluation of recommended refactorings.

We performed a small industrial case studybased on one industrial project JDI-Ford v5.8. JDI-Ford is a Java-based software system that implements 638 classes having 247 KLOC. This system is used by our industrial partner, the Ford Motor Company, to analyze useful information from the past sales of dealerships data and to suggest which vehicles to order for their dealer inventories in the future. JDI-Ford is the main key software application used by the Ford Motor Company to improve their vehicle

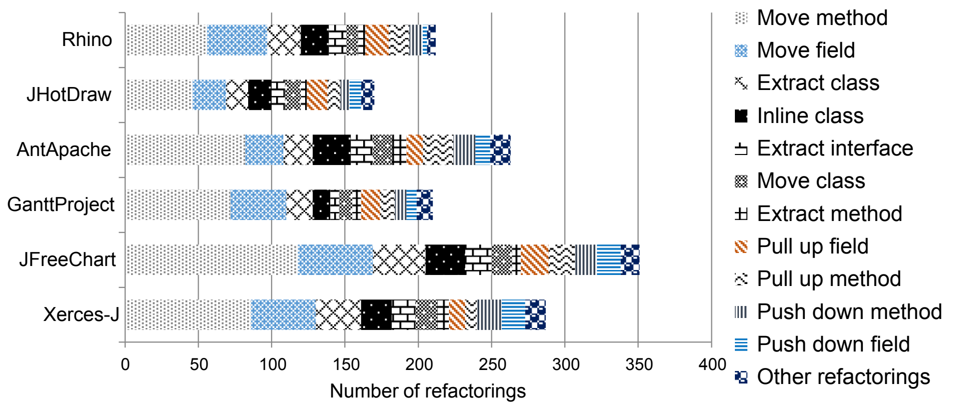


Fig. 16: Suggested refactorings distribution.

sales by selecting the right vehicle configuration to the expectations of customers. Several versions of JDI were proposed by software engineers at Ford during the past 10 years. Due to the importance of the application and the high number of updates performed during a period of 10 years, it is critical to make sure that all the JDI releases are within a good quality to reduce the time required by developers to introduce new features in the future.

The software engineers from Ford manually evaluated all the recommended refactorings for JDI by our tool using the RP metric, described in the previous section, based on their knowledge of the system since they are some of the original developers. We also evaluated the relevance of some of the suggested refactoring for the developers. In addition, we asked 4 out of the 10 software engineers from Ford to manually refactor some code fragments with a poor quality then we compared their suggested refactorings with the recommended ones by our approach. To decide about the quality of a code fragment, we used the domain knowledge of the 10 programmers from Ford (since they are part of the original developers of the systems), the quality metrics and detected design defects (to guide developers to identify a list of refactoring opportunities). Thus, we defined a metric called ER that represents the ratio of the number of good refactoring recommendation over the number of expected refactorings. The four selected software engineers are part of the original developers of the JDI system thus they easily provided different refactoring suggestions.

In this section, we aim at answering to the following two questions:

(1) To what extent can our approach propose correct refactoring recommendations?
(2) To what extent the suggested refactorings are relevant and useful for software engineers?

We describe, first, in this section the subjects participated in our study. Second, we give details about the questionnaire, instructions, and the conducted pilot study. Finally, we describe and discuss the obtained results.

7.1. Subjects

Our study involved 10 software engineers from the Ford Motor Company. All the subjects are familiar with Java development, software maintenance activities including refactoring. The experience of these subjects on Java programming ranged from 4 to 17 years. They were selected, as part of a project funded by Ford, based on having similar development skills, their motivations to participate in the project and their availability. They are part of the original developers' team of the JDI system.

7.2. Pilot Study

Subjects were first asked to fill out a pre-study questionnaire containing six questions. The questionnaire helped to collect background information such as their role within the company, their contribution in the development of JDI, their programming experience, their familiarity with quality assurance and software refactoring.

We divided the subjects into 5 groups (two developers per group) to evaluate the correctness and the relevance of the recommended refactorings according to the number of refactorings to evaluate, and the results of the pre-study questionnaire. All the groups are similar, in average, in terms of programming experience, familiarity with the system and used tools, and have almost the same refactoring and code smells background. The study consists of two parts:

(1) The first part of the questionnaire includes questions to evaluate the correctness of the recommended refactoring using the following options: 1. Not correct; 2. Maybe Correct; and 3. Correct.
(2) The second part of the questionnaire includes questions around the relevance of the recommended refactorings using the following scale: 1. Not at all relevant; 2. Slightly relevant; 3. Moderately relevant; and 4. Extremely relevant.

The questionnaire is completed anonymously thus ensuring confidentiality and this study was approved by the IRB at the University of Michigan: "*Research involving the collection or study of existing data, documents, records, pathological specimens, or diagnostic specimens, if these sources are publicly available or if the information is recorded by the investigator in such a manner that participants cannot be identified, directly or through identifiers linked to the participants*".

The different programmers from the Ford Motor Company were asked not only to evaluate the generated refactoring solutions by our tool but they also used the tool to generate the refactoring solutions for the industrial system to evaluate. Thus, they performed all the required steps from the configuration of the multi-objective algorithm to the generation and analysis of the results. The programmers agreed that the tool was very easy to use due to the friendly graphical interface provided by the tool. All the programmers successfully executed the tool without any help from the supervisors of the experiments. During the entire process, subjects were encouraged to think aloud and to share their opinions, issues, detailed explanations, and ideas with the organizers of the study (one graduate student and one faculty from the University of Michigan) and not only answering the questions.

A brief tutorial session was organized for every participant around refactoring to make sure that all of them have a minimum background to participate in the study. All the developers performed the experiments in a similar environment: similar configuration of the computers, tools (Eclipse, Excel, etc.) and facilitators of the study. Because some support was needed for the installation of our Eclipse plug-in and the other detection techniques considered in our experiments, we added a short description of this instruction for the participants. These sessions were also recorded as audio and the average time required to finish all the questions was 3.5 hours. Thus, the maximum time spent by the developers was 8.5 hours (including the refactoring execution and inspection) however the total average time was 4 hours.

Prior to the actual experiment, we did a pilot run of the entire experiment with one software engineer from Ford. We performed this pilot study to verify whether the assignments were clear and if our estimation of the required time to finalize the experiments evaluation were realistic thus all the assignments could be completed in two sessions (one day) by the subjects. The pilot study pointed out that the assignments and the questions in the questionnaire form were clear and relevant, and that they could be executed as offered by the subjects of the pilot study within a maximum of 5 hours. The pilot study also pointed out that the description of refactorings and the examples were clear and sufficient to understand the different types of refactorings considered in our experiments. Note that the engineer who participated in the pilot study was not involved for the rest of experiment reported in the paper, and was instructed not to share information about the experience prior to the study.

7.3. Results of the Industrial Case Study

In this section, we evaluate the performance of our multi-objective refactoring technique in an industrial setting.

Our first experiment was to assess to correctness of the suggested refactorings. From the set of suggested refactoring, 87 out of 104 refactoring was accepted by Ford developers suggesting that our approach was correct with a precision higher of 84%. For more details, figure 17 reports the different types of refactorings that was correctly suggested by our approach and approved by the majority of software engineers.

Similar facts were found when analyzing the similarity between the refactorings recommended by our approach and those manually proposed by developers for several code fragments. Most of the fixed code fragments by the software engineers were related to the most severe and urgent ones based on their knowledge of the system. A number of 34 out of the 42 refactorings suggested by the developers were also proposed by our technique resulting to a precision of more than 80%. Only 5% of recommended refactorings were considered as not correct and 11% as maybe correct.

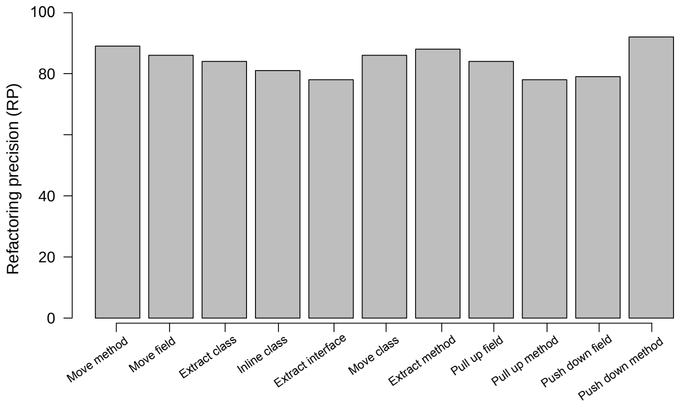


Fig. 17: Correctness of the different types of suggested refactorings.

In fact, the incorrect refactorings are due to some generated conflicts related to the fact that we are combining both complex and atomic refactorings in our solution. Although our repair operator eliminates the detected identical redundant refactorings within one solution, it is challenging to detect such issue when dealing with complex and atomic refactorings. For example, an extract class is composed by several atomic refactorings such create new class, move methods, move attributes, redirect method calls, etc. Thus, it is challenging to eliminate some conflicts between atomic and complex refactorings when it is a redundancy issue. A possible solution is to convert all complex refactorings to atomic ones then we can perform the comparison to detect the

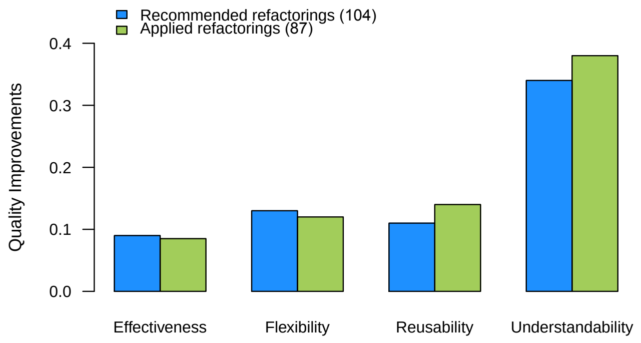


Fig. 18: Quality improvements on JDI-Ford after applying the recommended refactorings.

redundancy. However, the conversion process is not straightforward since one complex refactoring can be translated in different ways in terms of atomic refactorings.

To better investigate the relevance of the recommended refactorings, we evaluated their impact on the quality of JDI-Ford based on QMOOD. Figure 18 depicts the quality attributes improvements of the JDI system after applying (i) all recommended refactorings (104 refactorings), and (ii) only the selected refactorings by the developers (87 refactorings). The obtained results shows that our approach succeeded in improving different aspect of software quality including reusability (0.11 of improvement), flexibility (0.13), understandability (0.34), and effectiveness (0.09). An interesting point here is that the results achieved by the selected refactorings (87 out of 104) outperform the ones achieved by all the recommended refactorings (104) in terms of understandability and reusability. This finding provides evidence that although developers seek to improve the overall quality of their code, they are prioritizing the understandability and reusability than other quality aspects. Indeed, we expected that developers will mainly apply refactorings that improve the readability and understandability of their code.

Moreover, we asked the developers to evaluate the relevance of the recommended refactorings for the JDI-Ford system. Only less than 5% of recommended refactorings are considered not at all relevant by the software engineers, 7% are considered as slightly relevant, 19% are moderately relevant, while 69% are considered as extremely relevant. Moreover, the assessment of the Cohen's Kappa coefficient κ [Cohen et al. 1960], which measures to what extent the developers agree when voting for a recommended refactoring, indicates a score of $\kappa = 0.79$. This significant score indicates "*substantial agreement*" as characterized by Landis and Koch [Landis and Koch 1977]. This confirms the importance of the recommended refactorings for developers that they need to apply them for a better quality of their systems.

To get more insights about the 5% of refactorings that are voted as "not at all relevant", we asked the developers to comment on some particular cases. We noticed that most of these rejected refactoring were related to utility classes in JDI, where move method refactorings are suggested to move some utility methods to the classes that are calling them. Developers mentioned that this kind of refactorings tends to be meaningless.

To better evaluate the relevance of the recommended refactorings, we investigated the types of refactorings that developers may consider them more or less important than others. Figure 19 shows that move method is considered as one of the most extremely relevant refactorings. In addition, extract method is also considered as another very important and useful refactoring. This can be explained by the fact that the developers are more interested to fix quality issues that are related to the size of classes or methods. Overall, the different types of refactorings are considered relevant. One reason can be that our approach provides a sequence of refactorings.

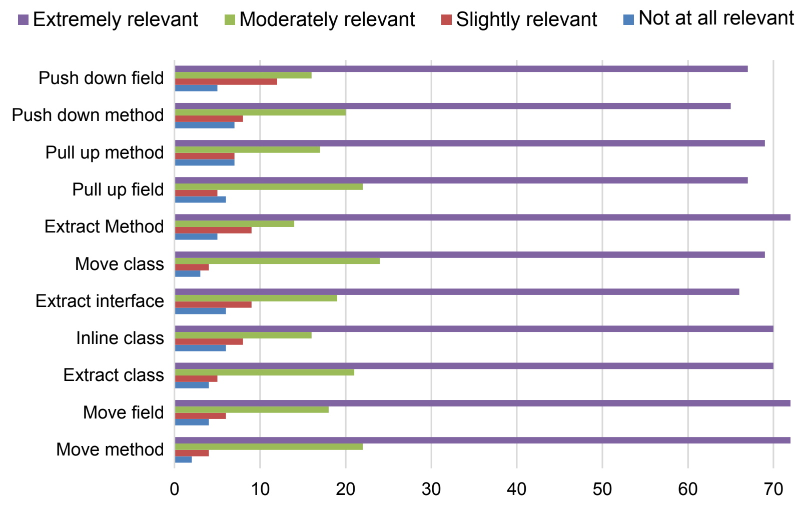


Fig. 19: The relevance of different types of recommended refactorings on JDI-Ford.

It was clear for our participants that our tool can provide faster and similar results that they can manually suggest. The refactoring of large scale system can be time consuming to fix several quality issues. The participants provided some suggestions to make our refactoring better and more efficient. First, the tool does not provide any ranking to prioritize the suggested refactorings. In fact, the developers do not have enough time to apply all the suggested refactorings but they prefer to fix the most severe quality issues. Second, our technique does not provide a support to fix refactoring solutions when the developers did not approve part of the suggested refactorings. Finally, the software engineers prefer that our tool provides a feature to automatically apply some regression testing techniques to generate test cases for the modified code fragments after refactoring. Such a feature is very interesting to include in our tool to automatically test the Java refactoring engine similarly to *SafeRefactor* [Soares et al. 2013].

8. THREATS TO VALIDITY

Some potential threats can affect the validity of our experiments. We now discuss these potential threats and how we deal with them.

Construct validity concerns the relation between the theory and the observation. In our experiments, the design defect detection rules [Ouni et al. 2012a] we use to measure DCR could be questionable. To mitigate this threat, we manually inspect and validate each detected defect. Moreover, our refactoring tool configuration is flexible and can support other state-of-the-art detection rules. In addition, different threshold values were used in our experiments based on trial-and-error, however these values can be configured once then used independently from the system to evaluate. Another threat concerns the data about the actual refactorings of the studied systems. In addition to the documented refactorings, we are using Ref-Finder, which is known to be efficient [Prete et al. 2010]. Indeed, Ref-Finder was able to detect refactoring operations with an average recall of 95% and an average precision of 79% [Prete et al. 2010]. To ensure the precision, we manually inspect the refactorings found by Ref-Finder. We identify three threats to internal validity: selection, learning and fatigue, and diffusion.

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

Randomization also helps to prevent the learning and fatigue threats. For the *fatigue* threat, specifically, we did not limit the time to fill the questionnaire for the open source systems. Consequently, we sent the questionnaires to the subjects by email and gave them enough time to complete the tasks. Finally, only ten refactorings per system was randomly picked for the evaluation. However, all refactoring solutions were evaluated for the industrial system.

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

Conclusion validity deals with the relation between the treatment and the outcome. Thus, to ensure the heterogeneity of subjects and their differences, we took special care to diversify them in terms of professional status, university/company affiliations, gender, and years of experience. In addition, we organized subjects into balanced groups. Having said that, we plan to test our tool with Java development companies, to draw better conclusions. Moreover, the automatic evaluation is also a way to limit the threats related to subjects as it helps to ensure that our approach is efficient and useful in practice. Indeed, we compare our suggested refactorings with the expected ones that are already applied to the next releases and detected using Ref-Finder.

Another potential threat can be related to parameters selection. We selected different parameters of our NSGA-II algorithm, such as the population size, the maximum number of iterations, mutation and crossover probabilities, and the solution length, based on the trial-and-error method and depending on the size of the evaluated systems, the initial number of design defect instances detected, and the number of refactoring types implemented in our tool (11 types, table II). However, as these parameters are independent each other, they can be easily configured according to the preferences of the developers, for example if they want to reduce the execution time (e.g., reduce the number of iterations) and maybe sacrifice a bit on the quality of the solutions.

Also when comparing the different approaches, some of them are using less types of refactorings. We believe that this is one of the limitations of these approaches thus it is interesting to show that considering the 11 types of refactorings of our approach may improve the results (even if programmers may apply them less frequently). Furthermore, when comparing the different approaches from the effort perspective, the code changes score is relative to DCR level. Not all design defects require the same amount of code changes. The process prioritizes the correction of design defects that require less changes to have higher DCR score. In addition, our results were consistent on all the different DCR levels for all the systems.

External validity refers to the generalizability of our findings. In this study, we performed our experiments on different open-source and industrial Java systems belonging to different application domains and with different sizes. However, we cannot assert that our results can be generalized to other programming languages, and to other practitioners.

The industrial validation section was checked by the Ford Motor Company. Our industrial partner accepted to only include the results mentioned in the current validation section for several reasons. Similar to most collaborations with industry, we are not allowed to mention the name of code elements or providing any example from the source code. One of our motivations to use open source systems in our validation is the hard constraint to not share the industrial data. Thus, the readers can at least check different examples of suggested refactorings on the open source system in the website provided with this paper.

9. RELATED WORK

Several studies have been focused on software refactoring in recent years. In this section, we survey those works that can be classified into three broad categories: (*i*) manual and semi-automated approaches, (*ii*) search-based approaches, and (*iii*) semantics-based approaches.

9.1. Manual and semi-automated approaches

The first book in the literature was written by Fowler [Fowler 1999] and provides 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. After Fowler's book several approaches have merged with the goal of taking advantage from refactoring to improve quality metrics of software systems. In [Sahraoui et al. 2000], Sahraoui et al. proposed an approach to detect opportunities of code transformations (i.e., refactorings) based on the study of the correlation between certain quality metrics and refactoring changes. Consequently, different rules are manually defined as a combination of metrics/thresholds to be used as indicators for detecting refactoring opportunities. For each code smell a pre-defined and standard list of transformations should be applied. In contrast to our approach, we do not have a pre-defined list of refactorings to apply, instead, our approach automatically recommends refactorings depending on the context.

Another similar work is proposed by Du Bois et al. [Du Bois et al. 2004] who starts from the hypothesis that refactoring opportunities correspond of those which improves cohesion and coupling metrics to perform an optimal distribution of features over classes. Anquetil et al. analyze how refactorings manipulate coupling and cohesion metrics, and how to identify refactoring opportunities that improve these metrics [Anquetil and Laval 2011]. The authors reported that refactorings manually performed by developers do not necessarily improve the modularity in terms of cohesion/coupling. This suggests that goal-oriented refactoring recommendation is useful to improve specific aspects of the system, which is one of the motivations of our approach. However, these 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. In our approach, we are taking as input the set of code smells that could be detected using the above studies but we did not address the problem of using quality metrics to identify design defects.

Moha et al. [Moha et al. 2008] proposed an approach that suggests refactorings using Formal Concept Analysis (FCA) to fix god class design defect. 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. [Joshi and Joshi 2009] have presented an approach based on concept analysis aimed at identifying less cohesive classes. It also helps identify less cohesive methods, attributes and classes at the same time. Furthermore, the approach guides refactoring opportunities identification such as extract class, move method, localize attributes and remove unused attributes. In addition, Tahvildari et al. [Tahvildari and Kontogiannis 2003] proposed a framework of object-oriented metrics used to suggest refactoring opportunities to improve the quality of object-oriented legacy systems. In contrast, our approach is not based explicitly on quality metrics as indicator for quality improvements, instead, we are based on the number of fixed design defects. Indeed, improving quality does not necessarily mean that actual design defects are fixed.

Another generation of semi-automated refactoring techniques have emerged. Murphy-Hill et al. [Murphy-Hill and Black 2008; Murphy-Hill et al. 2012; Murphy-Hill and Black 2012] propose several techniques and empirical studies to support refactoring activities. In [Murphy-Hill and Black 2008] and [Murphy-Hill and Black 2012], the authors propose new tools to assist software engineers in applying refactoring such as selection assistant, box view, and refactoring annotation based on structural information and program analysis techniques. Recently, Ge and Murphy-Hill have proposed new refactoring tool called GhostFactor [Ge and Murphy-Hill 2014] that allows the developer to transform code manually, but check the correctness of the transformations automatically. However, the correction is based mainly on the structure of the code and does not consider the issue of design coherence as our proposal does. 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 [Kataoka et al. 2001]. In addition, Opdyke [Opdyke 1992] 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. However, unlike our approach, these approaches focus only on behavior preservation and do not consider the design coherence of the program.

Tsantalis et al. [Tsantalis and Chatzigeorgiou 2009] and Sales et al. [Sales et al. 2013] proposed techniques to identify move methods opportunities by studying the existing dependencies between classes. A similar technique was suggested by Fokaefs et al. [Fokaefs et al. 2012] to detect extract class possibilities by analyzing dependencies between methods and classes. However, such approaches are local, i.e., they focus on a specific code fragment. In contrast to our approach, we are providing a generic refactoring approach that consider the effect on the whole system being refactored.

Furthermore, several empirical studies [Kim et al. 2014; Negara et al. 2013; Franklin et al. 2013; Alves et al. 2014] was performed recently to understand the benefits and risk of refactoring. Thee studies show that the main risk that refactorings could introduce is the creation of bugs after refactoring but several benefits could be obtained such as reducing the time that programmers spent to understand existing implemented features.

More details about current literature related to manual or semi-automated software refactoring can be found in the following two recent surveys [Bavota et al. 2014b; Al Dallal 2015].

9.2. Search-based approaches

To automate refactoring activities, new approaches have emerged where search-based techniques have been used. These approaches cast the refactoring problem as an optimization problem, where the goal is to improve the design quality of a system based mainly on a set of software metrics. After formulating the refactoring as an optimization problem, several techniques can be applied for automating refactoring, e.g., genetic algorithms, simulated annealing, and Pareto optimality, etc. Hence, we classify those approaches into two main categories: (1) mono-objective and (2) multi-objective optimization approaches.

In the first category, the majority of existing work combines several metrics in a single fitness function to find the best sequence of refactorings. Seng et al. [Seng et al. 2006] propose a single-objective search-based approach using genetic algorithm to suggest a list of refactorings to improve software quality. The search process uses a single fitness function to maximize a weighted sum of several quality metrics. The employed metrics are mainly related to various class level properties such as coupling, cohesion, complexity and stability while satisfying a set of pre-conditions for each refactoring. These conditions serve at preserving the program behavior (refactoring feasibility). However, in contrast to our approach, this approach does not consider the design coherence of the refactored program and limited only to move method refactoring. Another similar work of O'Keeffe et al. [O'Keeffe and Cinnéide 2008] that uses different local search-based techniques such as hill climbing and simulated annealing to provide an automated refactoring support based on the QMOOD metrics suite. Interestingly, they also found that the understandability function yielded the greatest quality gain, in keeping with our observation in Section 6.2.

Qayum et al. [Qayum and Heckel 2009] 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 does not consider the design semantics neither its runtime behavior. Fatiregun et al. [Fatiregun et al. 2004] show how search-based transformations could be used to reduce code size and construct amorphous program slices. However, they have used small atomic level transformations in their approach. However, their aim was to reduce program size rather than improving its structure/quality.

Otero et al. [Otero et al. 2010] introduced an approach to explore the addition of a refactoring step into the genetic programming iteration. It consists of an additional loop in which refactoring steps, drawn from a catalog, will be applied to individuals of the population. Jensen et al. [Jensen and Cheng 2010] have proposed an approach that supports composition of design changes and makes the introduction of design patterns a primary goal of the refactoring process. They used genetic programming and software metrics to identify the most suitable set of refactorings to apply to a software design. Furthermore, Kilic et al. [Kilic et al. 2011] explore the use of a variety of population-based approaches to search-based parallel refactoring, finding that local beam search could find the best solutions. However, still these approach focusing on specific refctoring types and not not consider the design semantics.

Zibran et al. [Zibran and Roy 2011] formulated the problem of scheduling of code clone refactoring activities as a constraint satisfaction optimization problem (CSOP) to fix known duplicate code code-smells. The proposed approach consists of applying constraint programming (CP) technique that aims to maximize benefits while minimizing refactoring efforts. An effort model is used for estimating the effort required to refactor code clones in object-oriented codebase. Although there is a slight similarity between the proposed effort model and our code changes score model [Ouni et al. 2012a], the proposed approach does not ensure the design coherence of the refactored program.

In the second category, the first multi-objective approach was introduce by Harman et al. [Harman and Tratt 2007] as described earlier. Recently, O Cinneide et al. [Ó Cinnéide et al. 2012] have proposed a multi-objective search-based refactoring to conduct an empirical investigation to assess some structural metrics and to explore relationships between them. To this end, they have used a variety of search techniques (Pareto-optimal search, semi-random search) guided by a set of cohesion metrics. The main weakness in all of these approaches is that the design preservation have not been addressed to obtain correct and meaningful refactorings, neither the effort required to apply refactoring which addressed in our approach.

9.3. Semantic coherence for refactoring

There exists few works focusing on refactorings that involves semantic coherence. In their approach, Fatiregun et al. [Fatiregun et al. 2004; Fatiregun et al. 2005] have applied a number of simple atomic transformation rules called axioms. 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, their proposed semantic equivalence is based only on structural rules related to the axioms, rather than a semantic analysis of the code.

Later, Bavota et al. [Bavota et al. 2012] have proposed an approach for 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 a 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. Similarly, Bavota et al. [Bavota et al. 2014a; Bavota et al. 2014b] used cohesion metrics in order to identify opportunities of extract class.

By exploiting semantic information, Bavota et al. [Bavota et al. 2014a] proposed a technique for the recommendation of move method using semantic information and relational topic models. Other studies tried to formulate semantic information based on cohesion for software clustering and remodularization [Corazza et al. 2011; Bavota et al. 2014; Scanniello et al. 2010]. Mkaouer et al. [Mkaouer et al. 2015] formulated software remodularization as many-objective problem however they used very basic semantic measures and it was limited to few refactorings applied at the package level. Moreover, Bavota et al. [Bavota et al. 2014b] suggested an approach to recommend appropriate refactoring operations to adjust the design according to the teams' activity patterns.

In [Baar and Marković 2007], Baar et al. have presented a simple criterion and a proof technique for the preservation of the design coherence 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 design preservation since there is no explicit differentiation between behaviour and design preservation. In fact, they consider that the semantic coherence "means that the observable behaviors of original and refactored programs coincide". In addition, in contrast to our approach, they partially address the design preservation in the model level with a high level of abstraction without considering the code/implementation level. In addition, this approach uses only the refactoring move attribute and do not consider popular refactorings[11].

Another semantics-based framework was introduced by Logozzo [Logozzo and Cortesi 2006] for the definition and manipulation of class hierarchies-based refactorings. The framework is based on the notion of the observable part of a class, i.e., an

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

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.

Furthermore, it is worth to note that most of the existing techniques are limited to a small number of refactorings (single refactoring based approaches). For instance, Harman et al. [Harman and Tratt 2007], Bavota et al. [Bavota et al. 2014a], Tsantalis and Chatzigeorgiou [Tsantalis and Chatzigeorgiou 2009], and Sales et al. [Sales et al. 2013] recommend only move method refactoring. Bavota et al. [Bavota et al. 2014a; Bavota et al. 2011; Bavota et al. 2014b; Bavota et al. 2010] and Fokaefs et al. [Fokaefs et al. 2011; Fokaefs et al. 2012] address the recommendation of the extract class refactoring. On the other hand, Silva et al., [Silva et al. 2014], Tsantalis and Chatzigeorgiou [Tsantalis and Chatzigeorgiou 2011] introduce an approach for recommending extract method refactorings, while Krishnan and Tsantalis focus on code clone refactorings [Krishnan and Tsantalis 2014]. To help developers with efficient refactoring recommendations, JDeodorant [Fokaefs et al. 2011; Tsantalis and Chatzigeorgiou 2011; Fokaefs et al. 2012; Tsantalis and Chatzigeorgiou 2009] unifies different techniques in one tool that support five refactorings (move method, extract class, extract method, replace conditional with polymorphism, and replace type code with state/strategy) to fix four types of code smells (god class, feature envy, type checking, and long method). Moreover, Seng et al. [Seng et al. 2006] implemented five refactorings (move method, pull up attribute, push down attribute, pull up method, and push down method) but they only focus on move method refactoring in their paper [Seng et al. 2006]. In contrast, one of the strengths of our approach is that it addresses several refactoring types (11 refactorings) at the same time as listed in Table II.

10. CONCLUSIONS AND FUTURE WORK

This paper presented a novel search-based approach taking into consideration multiple criteria to suggest "good" refactoring solutions to improve software quality. The process aims at finding the sequence of refactorings that (*i*) improves design quality, (*ii*) preserves the design coherence of the refactored program, (*iii*) minimizes code changes, and (*iv*) maximizes the consistency with development change history. We, thus, formulated our problem as a multi-objective search problem to find a trade-off between all these objectives using NSGA-II. Moreover, we defined different measures to estimate the design coherence of a code after refactoring and we also used the similarity with previous code changes as an indicator of the design consistency.

To evaluate our approach, we conducted an empirical study from both quantitative and qualitative perspectives on open-source and industrial projects. The open-source evaluation involved six medium and large size open-source systems with a comparison against three existing approaches [Harman and Tratt 2007; Kessentini et al. 2011; Fokaefs et al. 2011]. Our empirical study shows the efficiency of our approach in improving the quality of the studied systems while successfully fixing an average of 84% of design defects with low code change score (an average of 2,937 of low level changes). The qualitative evaluation shows that most of the suggested refactorings (an average of 80%) are considered as relevant and meaningful from developer's point of view. Moreover, unlike existing approaches, the obtained results show that our approach is efficient in suggesting a significant number of expected refactorings that was performed in the next release of the systems being studied which provides evidence that our approach is more efficient and useful in practice.

In addition, we conducted an industrial validation of our approach on a large-scale project and the results was manually evaluated by 10 active software engineers to assess the relevance and usefulness of our refactoring suggestions. The obtained results provide evidence that our approach succeeded in improving different aspect of software quality including reusability (0.11 of improvement), flexibility (0.13), understandability (0.34), and effectiveness (0.09). Moreover, 84% of the recommended (87 out of 104) was meaningful and useful from developer's point of view.

In future work, we are planning to conduct an empirical study to understand the correlation between correcting design defects and introducing new ones or fixing other design defects implicitly. We also plan to adapt our multi-objective approach to fix other types of defects that can occur in new emergent service-based applications [Rotem-Gal-Oz et al. 2012]. Future replications of our study with additional systems and design defect types are necessary to confirm our findings. Another limitation of our current approach is the selection of the best solution from the Pareto front. We used the technique of selecting the closest solution to the ideal point. However, we plan in our future work to integrate the developers preferences to select the best solution from the set of non-dominated solutions. Moreover, one limitation of our approach is that one input is a base of recorded/collected code changes on previous versions. We believe that this data is not always available, especially in the beginning of the projects. As a future work, we plan to reuse refactorings recorded/collected for other similar contexts can be used instead. This can be done by calculating the similarity with not only the refactoring type but also between the contexts (code fragments). Furthermore, we are planning to include more criteria and constraints to improve the meaningfulness of the suggested refactorings, an interesting one is to identify refactorings related to utility classes and prevent moving methods/fields between utility and functional classes, as these refactoring are unlikely to be meaningful. Finally, we are planning to include other fine-grained refactoring operations such as `Decompose Conditional`, `Replace Conditional with Polymorphism`, and `Replace Type Code with State/Strategy` to improve the quality of the code.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their relevant feedback and useful comments that helped them to improve this work. Also, the authors are grateful to the subject students from the University of Michigan-Dearborn and developers who participated in their empirical study. This work was supported by the Ford-University of Michigan alliance Program, Japan Society for the Promotion of Science, Grant-in-Aid for Scientific Research (S) (No.25220003) and by Osaka University Program for Promoting International Joint Research.

REFERENCES

Jehad Al Dallal. 2015. Identifying refactoring opportunities in object-oriented code: A systematic literature review. *Information and Software Technology* 58 (2015), 231–249.

Everton LG Alves, Myoungkyu Song, and Miryung Kim. 2014. RefDistiller: a refactoring aware code review tool for inspecting manual refactoring edits. In *22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*. 751–754.

Raquel Amaro, Rui Pedro Chaves, Palmira Marrafa, and Sara Mendes. 2006. Enriching wordnets with new relations and with event and argument structures. In *Computational Linguistics and Intelligent Text Processing*. Springer, 28–40.

Nicolas Anquetil and Jannik Laval. 2011. Legacy software restructuring: Analyzing a concrete case. In *15th European Conference on Software Maintenance and Reengineering (CSMR)*. 279–286.

Thomas Baar and Slaviša Marković. 2007. A graphical approach to prove the semantic preservation of UML/OCL refactoring rules. In *Perspectives of systems informatics*. Springer, 70–83.

Jagdish Bansiya and Carl G Davis. 2002. A hierarchical model for object-oriented design quality assessment. *IEEE Transactions on Software Engineering* 28, 1 (2002), 4–17.

Gabriele Bavota, Andrea De Lucia, Andrian Marcus, and Rocco Oliveto. 2014a. Automating extract class refactoring: an improved method and its evaluation. *Empirical Software Engineering* 19, 6 (2014), 1617–1664.

Gabriele Bavota, Andrea De Lucia, Andrian Marcus, and Rocco Oliveto. 2014b. Recommending Refactoring Operations in Large Software Systems. In *Recommendation Systems in Software Engineering*. Springer, 387–419.

Gabriele Bavota, Andrea De Lucia, Andrian Marcus, Rocco Oliveto, and Fabio Palomba. 2012. Supporting extract class refactoring in Eclipse: the ARIES project. In *34th International Conference on Software Engineering (ICSE)*. IEEE Press, 1419–1422.

Gabriele Bavota, Andrea De Lucia, and Rocco Oliveto. 2011. Identifying extract class refactoring opportunities using structural and semantic cohesion measures. *Journal of Systems and Software* 84, 3 (2011), 397–414.

Gabriele Bavota, Malcom Gethers, Rocco Oliveto, Denys Poshyvanyk, and Andrea de Lucia. 2014. Improving software modularization via automated analysis of latent topics and dependencies. *ACM Transactions on Software Engineering and Methodology* 23, 1 (2014), 4.

Gabriele Bavota, Rocco Oliveto, Andrea De Lucia, Giuliano Antoniol, and Yann-Gaël Guéhéneuc. 2010. Playing with refactoring: Identifying extract class opportunities through game theory. In *26th International Conference on Software Maintenance (ICSM)*. 1–5.

Gabriele Bavota, Rocco Oliveto, Malcom Gethers, Denys Poshyvanyk, and Andrea De Lucia. 2014a. Methodbook: Recommending move method refactorings via relational topic models. *IEEE Transactions on Software Engineering* 40, 7 (2014), 671–694.

Gabriele Bavota, Sebastiano Panichella, Nikolaos Tsantalis, Massimiliano Di Penta, Rocco Oliveto, and Gerardo Canfora. 2014b. Recommending refactorings based on team co-maintenance patterns. In *29th International Conference on Automated software engineering (ASE)*. 337–342.

William H Brown, Raphael C Malveau, and Thomas J Mowbray. 1998. AntiPatterns: refactoring software, architectures, and projects in crisis. (1998).

Mel O Cinnéide. 2001. *Automated application of design patterns: a refactoring approach*. Ph.D. Dissertation. Trinity College Dublin.

Norman Cliff. 1993. Dominance statistics: Ordinal analyses to answer ordinal questions. *Psychological Bulletin* 114, 3 (1993), 494.

Jacob Cohen and others. 1960. A coefficient of agreement for nominal scales. *Educational and psychological measurement* 20, 1 (1960), 37–46.

Anna Corazza, Sergio Di Martino, and Valerio Maggio. 2012. LINSEN: An efficient approach to split identifiers and expand abbreviations. In *28th International Conference on Software Maintenance (ICSM)*. 233–242.

Anna Corazza, Sergio Di Martino, Valerio Maggio, and Giuseppe Scanniello. 2011. Investigating the use of lexical information for software system clustering. In *15th European Conference on Software Maintenance and Reengineering (CSMR)*. 35–44.

Marco D'Ambros, Alberto Bacchelli, and Michele Lanza. 2010. On the impact of design flaws on software defects. In *10th International Conference on Quality Software (QSIC)*. 23–31.

Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. 2002. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation* 6, 2 (2002), 182–197.

Ignatios Deligiannis, Martin Shepperd, Manos Roumeliotis, and Ioannis Stamelos. 2003. An empirical investigation of an object-oriented design heuristic for maintainability. *Journal of Systems and Software* 65, 2 (2003), 127–139.

K. Dhambri, H. Sahraoui, and P. Poulin. 2008. Visual Detection of Design Anomalies. In *12th European Conference on Software Maintenance and Reengineering (CSMR)*. 279–283.

Bart Du Bois, Serge Demeyer, and Jan Verelst. 2004. Refactoring-improving coupling and cohesion of existing code. In *11th Working Conference on Reverse Engineering (WCRE)*. 144–151.

Len Erlikh. 2000. Leveraging legacy system dollars for e-business. *IT professional* 2, 3 (2000), 17–23.

Deji Fatiregun, Mark Harman, and Robert M Hierons. 2004. Evolving transformation sequences using genetic algorithms. In *4th International Workshop on Source Code Analysis and Manipulation (SCAM)*. 65–74.

Deji Fatiregun, Mark Harman, and Robert M Hierons. 2005. Search-based amorphous slicing. In *12th Working Conference on Reverse Engineering (WCRE)*. IEEE, 10–pp.

Norman E. Fenton and Shari Lawrence Pfleeger. 1998. *Software Metrics: A Rigorous and Practical Approach* (2nd ed.). PWS Publishing Co., Boston, MA, USA.

Marios Fokaefs, Nikolaos Tsantalis, Eleni Stroulia, and Alexander Chatzigeorgiou. 2011. JDeodorant: identification and application of extract class refactorings. In *33rd International Conference on Software Engineering (ICSE)*. 1037–1039.

Marios Fokaefs, Nikolaos Tsantalis, Eleni Stroulia, and Alexander Chatzigeorgiou. 2012. Identification and application of extract class refactorings in object-oriented systems. *Journal of Systems and Software* 85, 10 (2012), 2241–2260.

Carlos M Fonseca, Peter J Fleming, and others. 1993. Genetic Algorithms for Multiobjective Optimization: Formulation, Discussion and Generalization.. In *5th International Conference on Genetic Algorithms*, Vol. 93. 416–423.

Martin Fowler. 1999. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Lyle Franklin, Alex Gyori, Jan Lahoda, and Danny Dig. 2013. LAMBDAFICATOR: from imperative to functional programming through automated refactoring. In *35th International Conference on Software Engineering (ICSE)*. 1287–1290.

Xi Ge and Emerson Murphy-Hill. 2014. Manual refactoring changes with automated refactoring validation. *36th International Conference on Software Engineering (ICSE)* 36 (2014), 1095–1105.

Mohamed Salah Hamdi. 2011. SOMSE: A semantic map based meta-search engine for the purpose of web information customization. *Applied Soft Computing* 11, 1 (2011), 1310–1321.

Mark Harman, S Afshin Mansouri, and Yuanyuan Zhang. 2012. Search-based software engineering: Trends, techniques and applications. *ACM Computing Surveys (CSUR)* 45, 1 (2012), 11.

Mark Harman and Laurence Tratt. 2007. Pareto optimal search based refactoring at the design level. In *9th annual conference on Genetic and evolutionary computation (GECCO)*. 1106–1113.

Paul Jaccard. 1901. Étude comparative de la distribution florale dans une portion des Alpes et des Jura. *Bulletin del la Société Vaudoise des Sciences Naturelles* 37 (1901), 547–579.

Adam C. Jensen and Betty H.C. Cheng. 2010. On the Use of Genetic Programming for Automated Refactoring and the Introduction of Design Patterns. In *12th Annual Conference on Genetic and Evolutionary Computation (GECCO)*. 1341–1348.

Padmaja Joshi and Rushikesh K. Joshi. 2009. Concept Analysis for Class Cohesion. In *13th European Conference on Software Maintenance and Reengineering (CSMR)*. Washington, DC, USA, 237–240.

Yoshio Kataoka, David Notkin, Michael D Ernst, and William G Griswold. 2001. Automated support for program refactoring using invariants. In *International Conference on Software Maintenance (ICSM)*. IEEE Computer Society, 736.

Marouane Kessentini, Wael Kessentini, Houari Sahraoui, Mounir Boukadoum, and Ali Ouni. 2011. Design defects detection and correction by example. In *19th International Conference on Program Comprehension (ICPC)*. 81–90.

Marouane Kessentini, Stéphane Vaucher, and Houari Sahraoui. 2010. Deviance from perfection is a better criterion than closeness to evil when identifying risky code. In *25th International Conference on Automated Software engineering (ASE)*. 113–122.

Hurevren Kilic, Ekin Koc, and Ibrahim Cereci. 2011. Search-based parallel refactoring using population-based direct approaches. In *3rd International Symposium on Search Based Software Engineering (SSBSE)*. Springer, 271–272.

Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan. 2014. An Empirical Study of RefactoringChallenges and Benefits at Microsoft. *IEEE Transactions on Software Engineering* 40, 7 (2014), 633–649.

Giri Panamoottil Krishnan and Nikolaos Tsantalis. 2014. Unification and refactoring of clones. In *Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE)*. 104–113.

J Richard Landis and Gary G Koch. 1977. The measurement of observer agreement for categorical data. *biometrics* (1977), 159–174.

Wei Li and Raed Shatnawi. 2007. An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution. *Journal of systems and software* 80, 7 (2007), 1120–1128.

Rensis Likert. 1932. A technique for the measurement of attitudes. *Archives of psychology* (1932).

Francesco Logozzo and Agostino Cortesi. 2006. Semantic hierarchy refactoring by abstract interpretation. In *Verification, Model Checking, and Abstract Interpretation*. Springer, 313–331.

Mika Mäntylä, Jari Vanhanen, and Casper Lassenius. 2003. A taxonomy and an initial empirical study of bad smells in code. In *International Conference on Software Maintenance (ICSM)*. IEEE, 381–384.

Radu Marinescu. 2004. Detection strategies: metrics-based rules for detecting design flaws. In *20th International Conference on Software Maintenance (ICSM)*. 350–359.

Tom Mens and Tom Tourwé. 2004. A survey of software refactoring. *IEEE Transactions on Software Engineering* 30, 2 (2004), 126–139.

Wiem Mkaouer, Marouane Kessentini, Adnan Shaout, Patrice Koligheu, Slim Bechikh, Kalyanmoy Deb, and Ali Ouni. 2015. Many-Objective Software Remodularization Using NSGA-III. *ACM Transactions on Software Engineering and Methodology* 24, 3 (2015), 17.

Naouel Moha, Yann-Gaël Guéhéneuc, Laurence Duchien, and Anne-Francoise Le Meur. 2010. DECOR: A Method for the Specification and Detection of Code and Design Smells. *IEEE Transactions on Software Engineering* 36, 1 (Jan 2010), 20–36.

Naouel Moha, Amine Mohamed Rouane Hacene, Petko Valtchev, and Yann-Gaël Guéhéneuc. 2008. Refactorings of Design Defects Using Relational Concept Analysis. In *Formal Concept Analysis*, Raoul Medina and Sergei Obiedkov (Eds.). Lecture Notes in Computer Science, Vol. 4933. Springer Berlin Heidelberg, 289–304.

Emerson Murphy-Hill and Andrew P Black. 2008. Breaking the barriers to successful refactoring. In *30th International Conference on Software Engineering (ICSE)*. 421–430.

Emerson Murphy-Hill and Andrew P Black. 2010. An interactive ambient visualization for code smells. In *5th international symposium on Software visualization (VISSOFT)*. ACM, 5–14.

Emerson Murphy-Hill and Andrew P Black. 2012. Programmer-friendly refactoring errors. *IEEE Transactions on Software Engineering* 38, 6 (2012), 1417–1431.

Emerson Murphy-Hill, Chris Parnin, and Andrew P Black. 2012. How we refactor, and how we know it. *IEEE Transactions on Software Engineering* 38, 1 (2012), 5–18.

Stas Negara, Nicholas Chen, Mohsen Vakilian, Ralph E. Johnson, and Danny Dig. 2013. A Comparative Study of Manual and Automated Refactorings. In *27th European Conference on Object-Oriented Programming (ECOOP)*. 552–576.

Mel Ó Cinnéide, Laurence Tratt, Mark Harman, Steve Counsell, and Iman Hemati Moghadam. 2012. Experimental assessment of software metrics using automated refactoring. In *International Symposium on Empirical Software Engineering and Measurement (ESEM)*. 49–58.

Mark O'Keeffe and Mel O Cinnéide. 2008. Search-based refactoring for software maintenance. *Journal of Systems and Software* 81, 4 (2008), 502–516.

William F Opdyke. 1992. *Refactoring: A program restructuring aid in designing object-oriented application frameworks*. Ph.D. Dissertation. PhD thesis, University of Illinois at Urbana-Champaign.

Fernando EB Otero, Colin G Johnson, Alex A Freitas, and Simon J Thompson. 2010. Refactoring in automatically generated programs. In *2nd International Symposium on Search Based Software Engineering (SSBSE)*, Massimiliano Di Penta, Simon Poulding, Lionel Briand, and John Clark (Eds.). Benevento, Italy.

Ali Ouni, Marouane Kessentini, and Houari Sahraoui. 2013. Search-Based Refactoring Using Recorded Code Changes. In *17th European Conference on Software Maintenance and Reengineering (CSMR)*. 221–230.

Ali Ouni, Marouane Kessentini, Houari Sahraoui, and Mounir Boukadoum. 2012a. Maintainability defects detection and correction: a multi-objective approach. *Automated Software Engineering* 20, 1 (2012), 47–79.

Ali Ouni, Marouane Kessentini, Houari Sahraoui, and Mohamed Salah Hamdi. 2012b. Search-based refactoring: Towards semantics preservation. In *28th International Conference on Software Maintenance (ICSM)*. 347–356.

Ali Ouni, Marouane Kessentini, Houari Sahraoui, and Mohamed Salah Hamdi. 2013. The use of development history in software refactoring using a multi-objective evolutionary algorithm. In *15th annual conference on Genetic and Evolutionary Computation (GECCO)*. 1461–1468.

Kyle Prete, Napol Rachatasumrit, Nikita Sudan, and Miryung Kim. 2010. Template-based reconstruction of complex refactorings. In *26th International Conference on Software Maintenance (ICSM)*. 1–10.

Fawad Qayum and Reiko Heckel. 2009. Local Search-Based Refactoring as Graph Transformation. In *1st International Symposium on Search Based Software Engineering (SSBSE)*. 43–46.

Donald Bradley Roberts and Ralph Johnson. 1999. *Practical analysis for refactoring*. Ph.D. Dissertation.

Arnon Rotem-Gal-Oz, Eric Bruno, and Udi Dahan. 2012. *SOA patterns*. Manning.

Houari Sahraoui, Robert Godin, Thieny Miceli, and others. 2000. Can metrics help to bridge the gap between the improvement of oo design quality and its automation?. In *International Conference on Software Maintenance (ICSM)*. 154–162.

Vicent Sales, Ricardo Terra, Luis Fernando Miranda, and Marco Tulio Valente. 2013. Recommending move method refactorings using dependency sets. In *20th Working Conference on Reverse Engineering (WCRE)*. 232–241.

Giuseppe Scanniello, Anna D'Amico, Carmela D'Amico, and Teodora D'Amico. 2010. Using the kleinberg algorithm and vector space model for software system clustering. In *18th International Conference on Program Comprehension (ICPC)*. 180–189.

Olaf Seng, Johannes Stammel, and David Burkhart. 2006. Search-based determination of refactorings for improving the class structure of object-oriented systems. In *8th annual Conference on Genetic and Evolutionary Computation (GECCO)*. ACM, 1909–1916.

Raed Shatnawi and Wei Li. 2011. An empirical assessment of refactoring impact on software quality using a hierarchical quality model. *International Journal of Software Engineering and Its Applications* 5, 4 (2011), 127–149.

Danilo Silva, Ricardo Terra, and Marco Tulio Valente. 2014. Recommending automated extract method refactorings. In *22nd International Conference on Program Comprehension (ICPC)*. 146–156.

Gustavo Soares, Rohit Gheyi, and Tiago Massoni. 2013. Automated behavioral testing of refactoring engines. *IEEE Transactions on Software Engineering* 39, 2 (2013), 147–162.

Ladan Tahvildari and Kostas Kontogiannis. 2003. A metric-based approach to enhance design quality through meta-pattern transformations. In *7th European Conference on Software Maintenance and Reengineering (CSMR)*. 183–192.

Frank Tip and Jens Palsberg. 2000. Scalable Propagation-based Call Graph Construction Algorithms. In *15th Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 281–293.

Nikolaos Tsantalis and Alexander Chatzigeorgiou. 2009. Identification of move method refactoring opportunities. *IEEE Transactions on Software Engineering* 35, 3 (2009), 347–367.

Nikolaos Tsantalis and Alexander Chatzigeorgiou. 2011. Identification of extract method refactoring opportunities for the decomposition of methods. *Journal of Systems and Software* 84, 10 (2011), 1757–1782.

Raja Vallée-Rai, Etienne Gagnon, Laurie Hendren, Patrick Lam, Patrice Pominville, and Vijay Sundaresan. 2000. Optimizing Java Bytecode Using the Soot Framework: Is It Feasible? In *Compiler Construction*, David A. Watt (Ed.). Lecture Notes in Computer Science, Vol. 1781. Springer Berlin Heidelberg, 18–34.

Atsushi Yamashita and Leon Moonen. 2013. Do developers care about code smells? An exploratory survey. In *20th Working Conference on Reverse Engineering (WCRE)*. IEEE, 242–251.

Minhaz F Zibran and Chanchal K Roy. 2011. A constraint programming approach to conflict-aware optimal scheduling of prioritized code clone refactoring. In *11th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 105–114.

Eckart Zitzler and Lothar Thiele. 1998. Multiobjective optimization using evolutionary algorithms–a comparative case study. In *Parallel problem solving from nature–PPSN V*. Springer, 292–301.