The Journal of Systems and Software 105 (2015) 18–39

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Improving multi-objective code-smells correction using development history

Ali Ouni [a,*], Marouane Kessentini [b], Houari Sahraoui [c], Katsuro Inoue [a], Mohamed Salah Hamdi [d]

[a] Graduate School of Information Science and Technology, Osaka University, Osaka, Japan
[b] Computer and Information Science Department, University of Michigan, Michigan, USA
[c] DIRO, Université de Montréal, QC, Canada
[d] IT Department, Ahmed Ben Mohamed Military College, Qatar

ARTICLE INFO

Article history:
Received 2 December 2013
Revised 7 January 2015
Accepted 11 March 2015
Available online 19 March 2015

Keywords:
Search-based software engineering
Refactoring
Code-smells

ABSTRACT

One of the widely used techniques to improve the quality of software systems is refactoring. Software refactoring improves the internal structure of the system while preserving its external behavior. These two concerns drive the existing approaches to refactoring automation. However, recent studies demonstrated that these concerns are not enough to produce correct and consistent refactoring solutions. In addition to quality improvement and behavior preservation, studies consider, among others, construct semantics preservation and minimization of changes. From another perspective, development history was proven as a powerful source of knowledge in many maintenance tasks. Still, development history is not widely explored in the context of automated software refactoring. In this paper, we use the development history collected from existing software projects to propose new refactoring solutions taking into account context similarity with situations seen in the past. We propose a multi-objective optimization-based approach to find good refactoring sequences that (1) minimize the number of code-smells, and (2) maximize the use of development history while (3) preserving the construct semantics. To this end, we use the non-dominated sorting genetic algorithm (NSGA-II) to find the best trade-offs between these three objectives. We evaluate our approach using a benchmark composed of five medium and large-size open-source systems and four types of code-smells (Blob, spaghetti code, functional decomposition, and data class). Our experimental results show the effectiveness of our approach, compared to three different state-of-the-art approaches, with more than 85% of code-smells fixed and 86% of suggested refactorings semantically coherent when the change history is used.

1. Introduction

During software maintenance and evolution, software systems undergo continuous changes, through which new features are added, bugs are fixed, and business processes are adapted constantly (Mens and Demeyer, 2008; Zimmermann et al., 2005). To support these activities, many tools emerged to manage source code such as concurrent versions system (CVS), and subversion (SVN) (Cederqvist and Dec, 2003) where all documentation, configuration, and code-changes are archived and called "software-development history". Hence, these historical data in software engineering provide a lot of solid knowledge that can be used to make sound data driven decisions for solving many software engineering problems (Ratzinger et al., 2007; Soetens et al., 2013; Beyer and Noack, 2005; Ying et al., 2004) and designing software engineering tools (Hassan and Holt, 2004; Gall et al., 1998; Ratiu et al., 2004). Furthermore, reuse is a common practice for developers during software development/maintenance to save time and efforts.

One of the widely used techniques during software development/maintenance to improve code quality is refactoring, the process of improving the internal structure of software systems without affecting its overall behavior (Opdyke, 1992). In general, to apply refactoring, we need to identify (1) where a program should be refactored and (2) which refactorings to apply (Fowler et al., 1999; Mens and Tourwé, 2004). Automating the refactoring suggestion task is essential and useful to efficiently help software developers in improving the quality of their code such as reusability, maintainability, flexibility, and understandability, etc.

Recently,search-based approaches have been applied to automate software refactoring (O'Keeffe and Cinnéide, 2006; Ouni et al., 2012; Harman and Tratt, 2007; Seng et al., 2006). Most of these works formulated refactoring as a single-objective optimization problem, in which the main goal is to improve code quality while preserving

* Corresponding author. Tel.: +15143436111.
E-mail addresses: ouniali@iro.umontreal.ca, ali@ist.osaka-u.ac.jp (A. Ouni), marouane@umich.edu (M. Kessentini), sahraouh@iro.umontreal.ca (H. Sahraoui), inoue@ist.osaka-u.ac.jp (K. Inoue), mshamdi@abmmc.edu.qa (M.S. Hamdi).

http://dx.doi.org/10.1016/j.jss.2015.03.040
0164-1212/

the behavior (see, for example, O'Keeffe and Cinnéide, 2006; Seng et al., 2006; Kessentini et al., 2011; Qayum and Heckel, 2009). In other works, other objectives are also considered such as reducing the effort (number of code changes) (Ouni et al., 2013a), preserving semantic coherence (Ouni et al., 2012), and improving quality metrics (O'Keeffe and Cinnéide, 2006). However, structural and semantic information are not, sometimes, enough to generate powerful refactoring strategies (Ouni et al., 2013b,c).

The use of development history can be helpful to propose efficient refactoring solutions (Ouni et al., 2013b,c). Code fragments that are modified over the past in the same period are semantically connected (i.e. belong to the same feature). Furthermore, fragments that are extensively refactored in the past bear a high probability for refactoring in the future. Moreover, code fragments to refactor can be similar to some patterns that can be found in the development history thus developers can easily reuse and adapt them. However, despite its importance, the history of code changes has not been widely investigated/used in the context of refactoring (Ouni et al., 2013c; Kagdi et al., 2007).

To the best of our knowledge, the use of historical data to automate software refactoring is not explored before our recent work in Ouni et al. (2013c). In this paper, we extend this work, published in GECCO 2013 (Ouni et al., 2013c), which uses the change history of a software system to suggest new refactoring opportunities to fix code-smells (Ouni et al., 2013c). We considered three different measures: (a) similarity with previous refactorings applied to the same code fragments within the same system, (b) number of changes applied in the past to the same code elements to be refactored, and (c) a score that characterizes the co-change of elements that will be refactored. The approach was successfully applied and evaluated on two different software systems using their change history and three types of code smells. However, the proposed approach can be applied only to systems with existing histories of changes.

In this paper, we consider the situation when the change history is not available or when we deal with newly developed software systems. We extend our previous work (Ouni et al., 2013c) by considering the history of past refactorings, applied to similar contexts, borrowed from different software projects. The primary contributions of this paper can be summarized as follows:

(1) We introduce a novel measure that aims at calculating the context similarity score between proposed refactoring operations, and a set of refactorings collected from different software projects.
(2) We define an extended formulation of the mutation genetic operator to better explore the search space.
(3) We extend the evaluation of the approach. We present an empirical study based on a quantitative and qualitative evaluation using (a) three additional subject systems, (b) an additional type of code-smells, and (c) two new quantitative evaluation metrics, hypervolume and spread (Zitzler et al., 2003; Deb, 2009), to better evaluate the performance of our approach. The quantitative evaluation investigates whether our approach is able to improve software quality while fixing code-smells using the development change history. For the qualitative evaluation, we evaluated the efficiency of our approach with three subjects from both academia and industry.

Our experimental results show that most of the detected code-smells were fixed with an average of 85%, and an average of 86% of suggested refactorings were semantically coherent. In addition, statistical analysis of our experiments over 31 runs shows that NSGA-II performed significantly better than state-of-the-art metaheuristic techniques in terms of hypervolume and spread (Zitzler et al., 2003; Deb, 2009).

The rest of this paper is organized as follows: Section 2 describes the relevant background and summarizes the related work in which the current paper is located; Section 3 describes the refactoring challenges through a motivating example. Section 4 describes the used multi-objective search-based algorithm and its design. Experimental results and evaluation of the approach are reported in Section 5. Section 6 is dedicated to the discussion, while Section 7 presents the threats to validity. Finally, concluding remarks and directions for future work are provided in Section 8.

2. Background

2.1. Definitions

2.1.1. Code-smells

Code-smells, also called anomalies (Fowler et al., 1999), anti-patterns (Brown et al., 1998), design flaws (Marinescu, 2004) or bad smells (Fowler et al., 1999), are problems resulting from bad design and programming practices and refer to situations that adversely affect the software maintenance and evolution. According to Fowler et al. (1999), code-smells are unlikely to cause failures directly, but may do it indirectly. In general, they make a system difficult to change, which may in turn introduce bugs. Different types of code-smells, presenting a variety of symptoms, have been studied in the intent of facilitating their detection (Moha et al., 2010) and suggesting improvement solutions. Most of code-smells identify locations in the code that violate object-oriented design heuristics, such as the situations described by Riel (1996) and by Coad and Yourdon (1991). Code-smells are not limited to design flaws since most of them occur in code and are not related to the original design. In fact, most of code-smells can emerge during the evolution of a system. In Fowler et al. (1999), Beck defines 22 sets of symptoms of code-smells and proposes the different possible refactoring solutions to improve the system design. These include large classes, feature envy, long parameter lists, and lazy classes. Each code-smell type is accompanied by refactoring suggestions to remove it. Van Emden and Moonen (2002) developed one of the first automated code-smell detection tools for Java programs. Mantyla studied the manner of how developers detect and analyze code smells (Mäntylä et al., 2003). Previous empirical studies have analyzed the impact of code-smells on different software maintainability factors including defects (Monden et al., 2002; Li and Shatnawi, 2007; Sjoberg et al., 2013) and effort (Deligiannis et al., 2003, 2004). In fact, software metrics (quality indicators) are sometimes difficult to interpret and suggest some actions (refactoring) as noted by Marinescu (2004) and Anda et al. (Mens and Demeyer, 2008). Code-smells are associated with a generic list of possible refactorings to improve the quality of software systems. In addition, Yamashita and Moonen (2013a,b) show that the different types of code-smells can cover most of maintainability factors (Yamashita and Moonen, 2012). Thus, the detection of code-smells can be considered as a good alternative of the traditional use of quality metrics to evaluate the quality of software products. Brown et al. (1998) define another category of code-smells that are documented in the literature, and named anti-patterns. In this paper, we focus on the four following code-smell types to evaluate our approach:

- **Blob**: It is found in designs where one large class monopolizes the behavior of a system (or part of it), and the other classes primarily encapsulate data.
- **Data class**: It contains only data and performs no processing on these data. It is typically composed of highly cohesive fields and accessors.
- **Spaghetti code**: It is a code with a complex and tangled control structure.
- **Functional decomposition**: It occurs when a class is designed with the intent of performing a single function. This is found in code produced by non-experienced object-oriented developers.

We decided to focus our attention on these code-smells because they are among the most related to faults or change proneness (Khomh et al., 2009) and the most common in the literature and frequently targeted for detection and correction in recent studies (see, for example, Kessentini et al., 2011; Ouni et al., 2013a,b,c; Moha et al., 2010). Hence, various approaches and tools supporting code-smells detection have been proposed in the literature such as JDeodorant (Tsantalis et al., 2008), infusion (Infusion hydrogen 2012), Décor (Moha et al., 2010), iPasma (iPlasma, 0000) Kessentini et al. (2011), Ouni et al. (2013a). The vast majority of these tools provide different environments and methods to detect code-smells. However, the correction is not mature yet, and several problems should be addressed.

2.1.2. Refactoring

One of the well-known development activities that can help fix code-smells and reduce the increasing complexity of a software system is *refactoring*. Fowler et al. (1999) define refactoring as a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior. The notion of refactoring was introduced by Opdyke (1992), which provided a catalogue of refactorings that could be applied in specific situations (Opdyke, 1992). The idea is to reorganize variables, classes and methods to facilitate future adaptations and extensions. This reorganization is used to improve different aspects of software-quality such as maintainability, extensibility, reusability, etc. (Fowler et al., 1999; Mens and Tourwé, 2004). For these precious benefits on design quality, some modern integrated development environments (IDEs), such as Eclipse,[1] NetBeans,[2] and Refactoring Browser,[3] provide semi-automatic support for applying the most commonly used refactorings, e.g., move method, rename class, etc. However, automatically suggesting/deciding where and which refactorings to apply is still a real challenge in software engineering. Roughly speaking, we can identify two distinct steps in the refactoring process: (1) detect where a program should be refactored and (2) identify which refactorings should be applied (Fowler et al., 1999).

2.1.3. Construct semantics

Construct semantics represents the semantic coherence in the structure of program entities. In object-oriented (OO) programs, objects reify domain concepts and/or physical objects. They implement their characteristics and behavior. Unlike for other programming paradigms, grouping data and behavior into classes is not guided by development or maintenance considerations. Operations and fields of classes characterize the structure and behavior of the implemented domain elements. Consequently, a program could be syntactically correct, implement the right behavior, but violates the domain and construct semantics if the reification of domain elements is incorrect. During the initial design/implementation, programs capture well the construct semantic when the OO principles are applied. However, when these programs are (semi) automatically modified during maintenance, the adequacy with domain semantics could be compromised. To this end, it is important to preserve the construct semantics during refactoring.

2.2. Related work

In this section, we review and discuss related work on software refactoring, and the use of change history for several purposes in software engineering. A large number of research works have addressed the problem of code-smells correction and software refactoring in recent years. We start by surveying those works that can be classified mainly into two broad categories: manual and semi-automated approaches, and search-based approaches.

2.2.1. Manual and semi-automated approaches

We start by summarizing existing manual and semi-automated approaches for software refactoring. In Fowler's book (Fowler et al., 1999), a non-exhaustive list of low-level design problems in source code has been defined. For each design problem (i.e., code-smell), a particular list of possible refactorings is suggested to be applied by software maintainers manually. Indeed, in the literature, most of the existing approaches are based on quality metrics improvement to deal with refactoring. Sahraoui et al. (2000) have proposed an approach to detect opportunities of code transformations (i.e., refactorings) based on the study of the correlation between some quality metrics and refactoring changes. To this end, different rules are defined as a combination of metrics/thresholds to be used as indicators for detecting code-smells and refactoring opportunities. For each code-smell a pre-defined and standard list of transformations should be applied in order to improve the quality of the code. Another similar work is proposed by Du Bois et al. (2004) who starts from the hypothesis that refactoring opportunities correspond of those which improve cohesion and coupling metrics to perform an optimal distribution of features over classes. Du Bois et al. analyze how refactorings manipulate coupling and cohesion metrics, and how to identify refactoring opportunities that improve these metrics. However, these two approaches are limited to only some possible refactoring operations with a small set of quality metrics. In addition, improving some quality metrics does not mean that existing code-smells are fixed.

Moha et al. (2008) proposed an approach that suggests refactorings using formal concept analysis (FCA) to correct detected code-smells. This work combines the efficiency of cohesion/coupling metrics with FCA to suggest refactoring opportunities. However, the link between code-smells detection and correction is not obvious, which make the inspection difficult for the maintainers. Similarly, Joshi 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 in one go. Further, the approach guides refactoring opportunity identification such as extract class, move method, localize attributes and remove unused attributes. In addition, Tahvildari and Kontogiannis (2003) also proposed a framework of object-oriented metrics used to suggest to the software engineer refactoring opportunities to improve the quality of an object-oriented legacy system.

Other contributions are based on rules that can be expressed as assertions (invariants, pre and post-condition). The use of invariants has been proposed to detect parts of code that require refactoring by Kataoka et al. (2001). In addition, Opdyke (1992) has proposed the definition and the use of pre- and post-condition with invariants to preserve the behavior of the software when applying refactoring. Hence, behavior preservation is based on the verification/satisfaction of a set of pre and post-condition. All these conditions are expressed in terms of rules.

The major limitation of these manual and semi-automated approaches is that they try to apply refactorings separately without considering the whole program to be refactored and its impact on the other artifacts. Indeed, these approaches are limited to only some possible refactoring operations and a few quality metrics to assess quality improvement. In addition, improving some quality metrics does mean necessary that actual code-smells are fixed.

2.2.2. Search-based approaches

Search-based approaches can be classified into two main categories: mono-objective and multi-objective optimization approaches.

In the first category, the majority of the existing work combines several metrics in a single fitness function to find the best sequence

[1] http://www.eclipse.org/.
[2] https://netbeans.org/.
[3] http://www.refactory.com/refactoring-browser.

of refactorings. Seng et al. (2006) have proposed a single-objective optimization-based approach using genetic algorithm to suggest a list of refactorings to improve software quality. The search process uses a single fitness function to maximize a weighted sum of several quality metrics. The used metrics are mainly related to various class level properties such as coupling, cohesion, complexity and stability. Indeed, the authors have used some pre-conditions for each refactoring. These conditions serve at preserving the program behavior (refactoring feasibility). However, in this approach, the semantic coherence of the refactored program is not considered. In addition, the approach was limited only on the refactoring operation "*move method*". Furthermore, there is another similar work of O'Keeffe and Cinnéide (2006) that have used different local search-based techniques such as hill climbing and simulated annealing to provide an automated refactoring support. Eleven object-oriented design metrics have been used to evaluate the quality improvement. One of the earliest works on search-based approaches is the work by Qayum and Heckel (2009) who considered the problem of refactoring scheduling as a graph transformation problem. They expressed refactorings as a search for an optimal path, using ant colony optimization, in the graph where nodes and edges represent respectively refactoring candidates and dependencies between them. However the use of graphs is limited only on structural and syntactical information and therefore does not consider the domain semantics of the program neither its runtime behavior. Furthermore, Fatiregun et al. (2004) showed how search-based transformations could be used to reduce code size and construct amorphous program slices. They have used small atomic level transformations in their approach. In addition, their aim was to reduce program size rather than to improve its structure/quality. Recently, Kessentini et al. (2011) have proposed a single-objective combinatorial optimization using genetic algorithm to find the best sequence of refactoring operations that improve the quality of the code by minimizing as much as possible the number of code-smells detected on the source code. Also, Otero et al. (2010) use a new search-based refactoring. The main idea behind this work is to explore the addition of a refactoring step into the genetic programming iteration. There will be an additional loop in which refactoring steps drawn from a catalogue of such steps will be applied to individuals of the population. 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. (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.

In the second category of work, Harman and Tratt (2007) have proposed a search-based approach using Pareto optimality that combines two quality metrics, CBO (coupling between objects) and SDMPC (standard deviation of methods per class), in two separate objective functions. The authors start from the assumption that good design quality results from a good distribution of features (methods) among classes. Their Pareto optimality-based algorithm succeeded in finding good sequence of "move method" refactorings that should provide the best compromise between CBO and SDMPC to improve code quality. However, one of the limitations of this approach is that it is limited to unique refactoring operation (move method) to improve software quality and only two metrics to evaluate the preformed improvements. Recently, 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. One of the earliest works on multi-objective search based refactoring is the work by Ouni et al. (2013a) who proposed a multi-objective optimization approach to find the best sequence of refactorings using NSGA-II. The proposed approach is based on two objective functions, quality (proportion of corrected code-smells) and code modification effort, to recommend a sequence of refactorings that provide the best trade-off between quality and effort.

To conclude, the vast majority of existing search-based software engineering approaches focused only on the program structure improvements. However, the main limitation is that the semantics preservation is not considered in the search process. Moreover, suggesting new refactorings should not be made independently to previous changes and maintenance/development history. This is one of the most relevant limitations on search-based refactoring approaches, which do not consider how the software has been changed and evolved and how software elements are impacted by these changes.

2.2.3. The use of historical data in software engineering

There is some few research work that uses the change history in the context of refactoring. Soetens et al. (2013) proposed an approach to detect (reconstruct) refactorings that are applied between two software versions based on the change history. The scope of this contribution is different than the one proposed in this paper, since our aim is to suggest refactoring solutions to be applied in the future to improve software quality while maintaining the consistency with the change history. Ratzinger et al. (2007) used change history mining to predict the likelihood of a class to be refactored in the next two months using machine learning techniques. Their goal is to identify classes that are refactoring or non-refactoring prone. In their prediction models they do not distinguish different types of refactorings (e.g. create super class, extract method, etc.); they only assess the fact that developers try to improve the design. In contrast, in our approach, we suggest concrete refactoring solution to improve code quality and not only identifying refactoring opportunities.

In addition, data extraction from development history/repository is very well covered. Research has been carried out to detect and interpret groups of software entities that change together. These co-change relationships have been used for different purposes. Zimmermann et al. (2005) have used historical changes to point developers to possible places that need change. In addition historical common code changes are used to cluster software artifacts (Beyer and Noack, 2005; Gîrba et al., 2007), to predict source code changes by mining change history (Zimmermann et al., 2005; Ying et al., 2004), to identify hidden architectural dependencies (Gall et al., 1998) or to use them as change predictors (Hassan and Holt, 2004). In addition, recently, co-change has been used in several empirical studies in software engineering. However, in the best of our knowledge, until now, the development change history is not used for software refactoring.

3. Refactoring challenges

3.1. Limitations and challenges

Various techniques are proposed to automate the refactoring process (Mens and Tourwé, 2004; Ouni et al., 2012, 2013a; Seng et al., 2006; Choinzon and Ueda, 2006; Mens and Tourwé, 2004). Most of these techniques are based on structural information using a set of quality metrics. The structural information is used to ensure that applied refactorings improve some quality metrics such as the number of methods/attributes per class, coupling, and cohesion. However, this could not be enough to confirm that a refactoring makes sense and preserves the design semantic coherence. It is important to preserve the rationale behind why and how code elements are grouped and connected.

To solve this issue, semantic measures are used to evaluate refactoring suggestions such as those based on coupling and cohesion or information retrieval techniques (e.g. cosine similarity of the used vocabulary) (Ouni et al., 2012). However, these semantic measures depend heavily on the meaning of code elements name/identifier (e.g.,

name of methods, name of classes, etc.). Indeed, due to some time-constraints, developers often select meaningless names for classes, methods, or fields (e.g., not clearly related to the functionalities). Thus, it is risky to only use techniques such as cosine similarity to find a semantic approximation between code fragments. In addition, when applying a refactoring like move method between two classes many target classes can have the same values of coupling and cohesion with the source class. To make the situation worse, it is also possible that the different target classes have the same structure. Furthermore, suggesting new refactorings should not be made independently to previous changes and maintenance/development history. This is one of the most relevant limitations existing work, which do not consider how the software has been changed and evolved and how software elements are impacted by these changes.

To circumvent the above mentioned problems, we use, in this paper, knowledge mined from past maintenance and development history to suggest refactoring solutions. Many aspects can help to improve the automation of refactoring (Ouni et al., 2013c): (1) code elements which underwent changes in the past, at approximately the same time, are in general semantically dependent, (2) code elements changed many times in the past have a good probability to be "badly-designed", (3) the development history can be used to propose new refactoring solutions in similar contexts. However, when we deal with newly developed software systems or with systems where the development is not available, other considerations should be taken into account to find similarities between the new candidate refactoring solutions and refactorings applied in other projects in similar contexts.

3.2. Motivating scenario

To illustrate some of the above-mentioned issues, Fig. 1 shows a concrete example extracted from JVacation,[4] an open-source standalone travel-booking-client for travel-agencies written in Java. JVacation was developed in 2007, and contains a large number of code-smells, which has been at the origin of the slowdown of their development. It has not been actively maintained/evolved since its first release. Consequently, JVacation has no development change history since it has not been widely used by the open-source community.

We consider a design fragment that contains four classes *DBClient, Booking, CustomerModel* and *JourneyModel*. Using the detection rules proposed in Ouni et al. (2013a), two code-smells are detected. The class *DBClient* is detected as a blob code-smell, and the class *CustomerModel* is detected as a data class. One possible refactoring solution to improve the design quality is to move some methods and/or fields from the blob class to other classes in the program, mainly data classes. In this way, we will reduce the number of functionalities implemented in the blob class *DBClient* and add additional behavior/functionalities to some other data classes such as *CustomerModel*. A refactoring is proposed to move the method *checkNotation*() from the smelly class *DBClient* to another suitable class in a way that we preserve the semantic coherence of the original program. Based on semantic and structural information (Ouni et al., 2012) many other target classes are possible, including *Booking, CustomerModel* and *JourneyModel* using vocabulary-based measures, cohesion and coupling (Ouni et al., 2012). However, since JVacation has no development change history, there are no refactorings applied in the past to these code fragments, and consequently, the history measures we defined in Ouni et al. (2013c) are not useful for this case.

An alternative solution can be to use the development history of other projects that are refactored in the past in similar contexts. Thus, we found in other software projects such as JFreeChart and JHotDraw similar contexts to our classes to refactor described in Fig. 1. We found in previous versions of JFreeChart some methods (such as *drawPrimaryLinePath(), initialise()*, and *equals()*) that have been moved from the class *XYLineAndShapeRenderer* to the class *XYSplineRenderer*. Moreover, we found in JHotDraw that the method *draw()* has been moved from the class *ArrowTip* to the class *AbstractLineDecoration*. These recorded refactorings in both JFreeChart and JHotDraw are applied to similar context with the structure of our classes being refactored in JVacation (i.e., the classes *DBClient* and *CustomerModel*). The similarity between code fragments can be estimated using not only quality metrics (e.g. coupling, number of methods, number of attributes, etc.) but also using semantic similarity measures (Ouni et al., 2012). As a consequence, moving methods and/or attributes from the class *DBClient* to the class *CustomerModel* has a high probability of correctness. Thus, in this paper, we are based on the hypothesis that the more a recommended refactoring is similar to refactorings applied to different software projects in the past in similar contexts, the more the meaningfulness of the recommended refactoring is high. The idea is to encourage the reuse of collected refactorings that are applied to similar contexts in different software projects.

The refactorings applied in the previous versions of a system can be detected using existing refactoring detection tools such as RefFinder. Several remarkable techniques have been proposed for recording code changes directly in the development framework as they are applied by a developer (Kim et al., 2013; Ekman and Asklund, 2004) or by analyzing two versions of a system and computing the differences between them. However, this is the first attempt to use development history to propose new refactoring solutions after our contribution in Ouni et al. (2013c).

4. Multi-objective search-based refactoring

In this section, we give an overview about our proposal where the use of development history is adapted to overcome the different limitations discussed in Section 2. Then, we describe our formulation of the refactoring suggestion problem and how the non-dominated sorting genetic algorithm (NSGA-II) (Deb, 2009) is used.

4.1. Approach overview

Our approach aims at exploring a huge search space to find refactoring solutions, i.e., a sequence of refactoring operations, to correct code-smells. We have three objectives to optimize: (1) maximize quality improvement (code-smells correction); (2) minimize the number of construct semantic incoherencies by preserving the way how code elements are semantically grouped and connected together; and (3) maintain the consistency with the previous refactorings applied to similar contexts using cross-project context similarity. To this end, we consider the refactoring task as a multi-objective optimization problem instead of a single-objective one using the non-dominated sorting genetic algorithm (NSGA-II) (Deb, 2009).

The general structure of our approach is sketched in Fig. 2. It takes as input the source code of the program to be refactored, a list of possible refactorings that can be applied, a set of code-smells detection rules (Ouni et al., 2013a; Kessentini et al., 2011), a set of semantic measures (Ouni et al., 2012), and a history of applied changes (CVS, change log, collected refactorings). Our approach generates as output the optimal set of refactorings, selected from the list of possible refactorings, which improves the software quality by minimizing as much as possible the number of code-smells, preserving construct semantics, and maximizing the reuse of development change history to similar contexts.

4.2. Problem formulation

In our previous work (Ouni et al., 2013c), we proposed an automated approach, to improve the quality of a system while preserving

[4] https://sourceforge.net/projects/jvacation/.

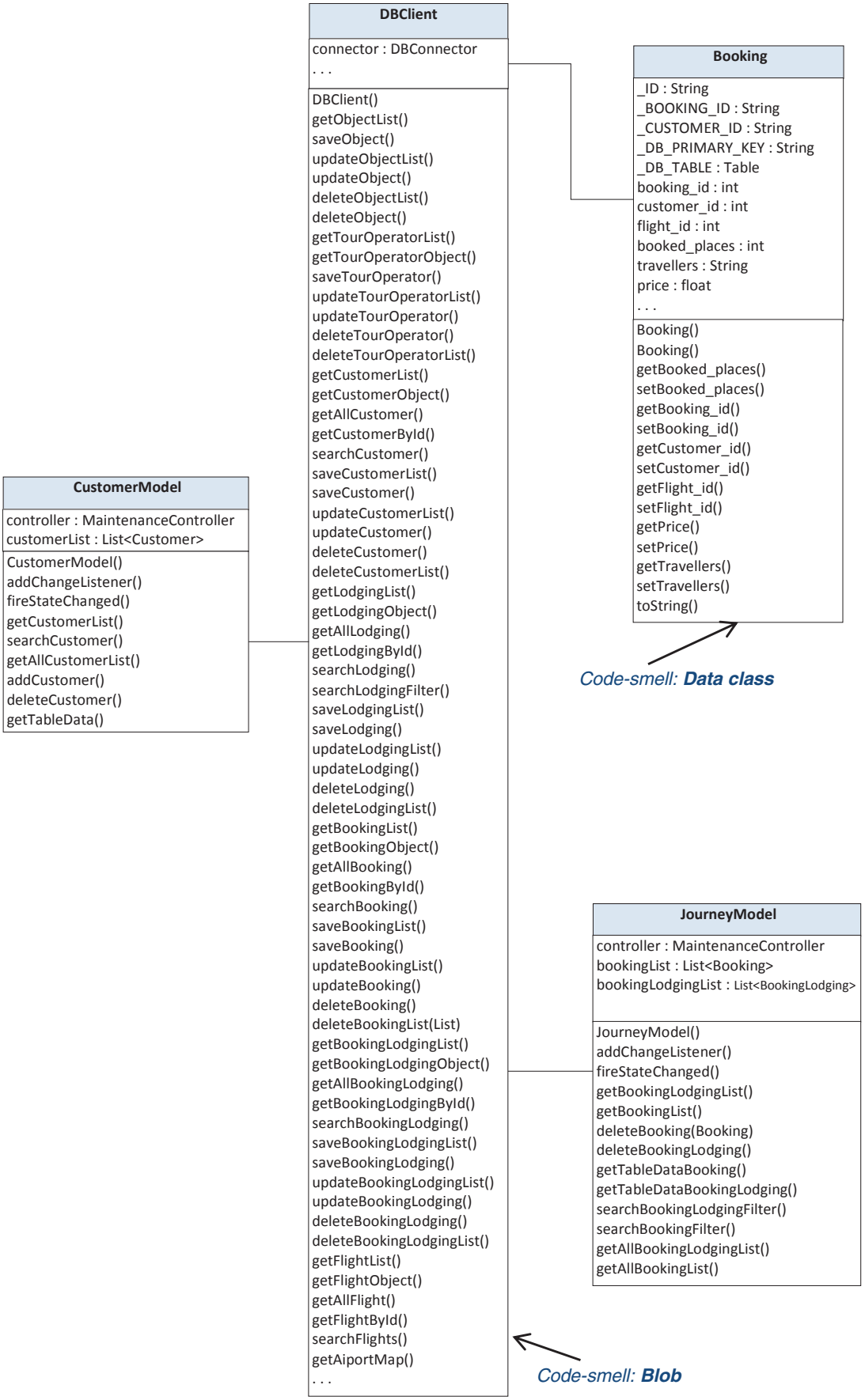


Fig. 1. Design fragment extracted from JVacation v1.0.

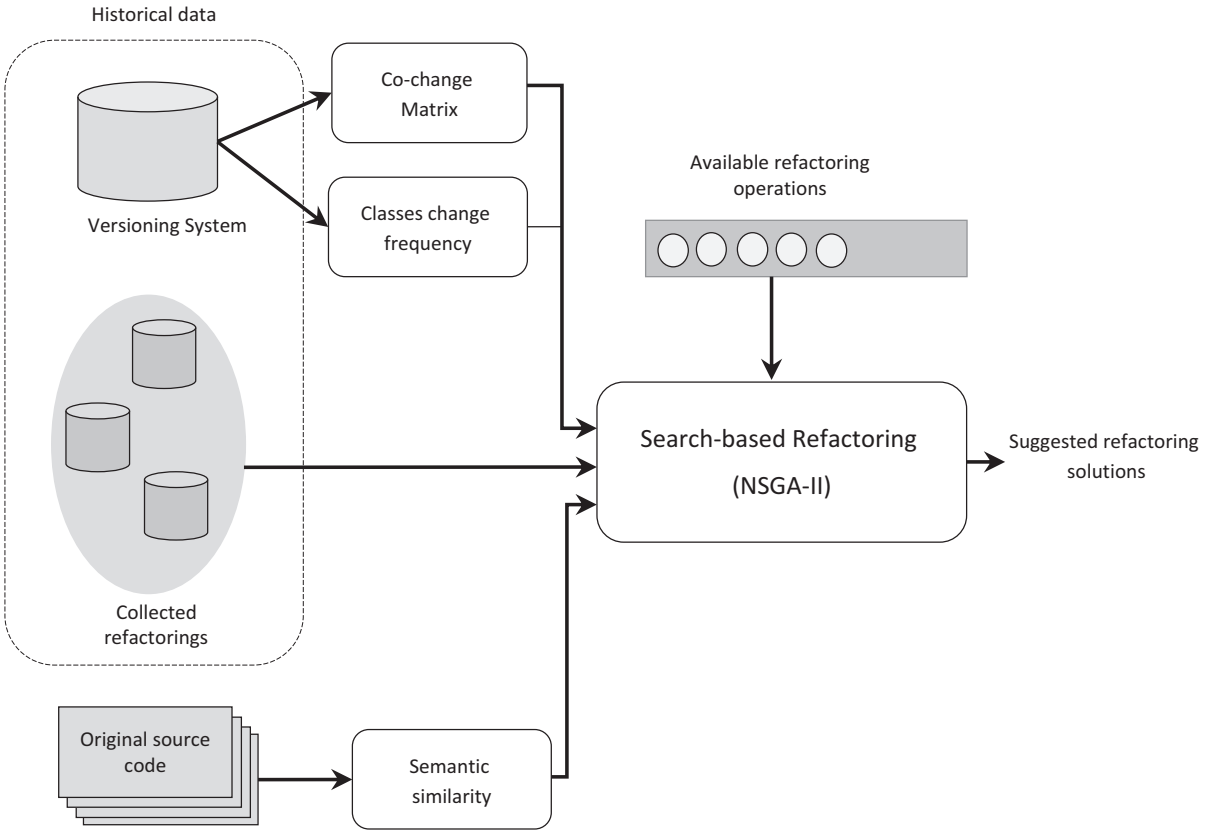


Fig. 2. The use of development history for recommending software refactoring: an overview.

its construct semantics. It uses multi-objective optimization to find the best compromise between code quality improvement and construct semantics preservation. Improving software quality corresponds to correcting code-smells. We used code-smell detection rules, proposed in our previous work (Ouni et al., 2013a), to find the best refactoring solutions, from the list of possible refactorings, which should fix as much as possible the number of detected code-smells. On the other hand, preserving the construct semantics after applying the suggested refactorings is ensured by maximizing different semantic measures: (1) vocabulary-based similarity (cosine similarity between words: name of code elements); and (2) dependency-based similarity (call-graph, coupling and cohesion) (Ouni et al., 2012). Moreover, we seek to maximize the use of development history to improve the construct semantics preservation.

Due to the large number of possible refactoring solutions and the conflicting objectives related to the quality improvements and the construct semantic preservation, we considered the refactoring as a multi-objective optimization problem instead of a single-objective one (O'Keeffe and Cinnéide, 2006; Seng et al., 2006; Kessentini et al., 2011). The search-based process takes as inputs, the source code, code-smells detection rules, a set of refactoring operations, and a call graph for the whole program. As output, our approach recommends a set of refactoring solutions. A solution consists of a sequence of refactoring operations that should be applied to improve the quality of the input code. The used detection rules are expressed in terms of metrics and threshold values. Each rule detects a specific code-smell type (e.g., blob, spaghetti code, functional decomposition, etc.) and is expressed as a logical combination of a set of quality metrics/threshold values (Kessentini et al., 2011; Ouni et al., 2013a). The process of generating a correction solution (refactorings) can be viewed as the mechanism that finds the best list among all available refactoring operations that best reduces the number of detected code-smells and at the same time, preserves the construct semantics of the initial program. In other words, this process aims at finding the best trade-off between these two conflicting criteria.

We extend, in this paper, our previous work (Ouni et al., 2013c) by considering the history of collected refactorings from different software projects applied to similar contexts. Concretely, we consider three different measures: (a) similarity with previous refactorings applied to similar contexts in other software projects, (b) number of changes applied in the past to the same code elements to modify, and (c) a score that characterizes the co-change of elements that will be refactored.

4.2.1. Similarity with recorded refactorings applied to similar contexts

The overall idea is to maximize/encourage the use of new refactorings that are similar to those applied to different software projects in similar contexts. To calculate the cross-project refactoring similarity score between a candidate refactoring operation and a set of collected refactorings, we use the following function:

$$SimRefactoringHistory\,(RO) = \sum_{i=1}^{n} w_{RO_i} * ContextSimilarity(RO,\ RO_i) \quad (1)$$

where:

- n is the number of recorded refactoring operations collected from different software systems;
- w_i is a refactoring weight that reflects the similarity between the suggested refactoring operation RO and the recorded refactoring operation RO_i. Table 1 describes the used weigh values. The weight w_i is computed as follows: if the two refactoring operations being compared have exactly the same type (e.g., "Move Method" and "Move Method"), the weight $w_i = 2$. If the refactoring operations being compared are similar, the weight $w_i = 1$. We consider two refactoring operations as similar if their implementations are

Table 1
Similarity weights between refactoring types.

	Move method	Move field	Pull up field	Pull up method	Push down field	Push down method	Inline class	Move class	Extract class	Extract interface
Move method	2	1	0	1	0	1	0	0	1	0
Move field	1	2	0	0	1	0	0	0	1	0
Pull up field	0	1	2	1	0	0	0	0	0	1
Pull up method	1	0	1	2	0	0	0	0	0	1
Push down field	0	1	0	0	2	1	0	0	0	1
Push down method	1	0	0	0	1	2	0	0	0	1
Inline class	1	1	1	1	1	1	2	0	0	0
Move class	1	1	1	1	1	1	0	2	1	0
Extract class	1	1	0	0	0	0	0	0	2	0
Extract interface	0	0	1	1	1	1	0	0	0	2

similar, or if one of them is composed by the other. For instance, some complex refactoring operations, such as "Extract Class", can be composed by combining other refactoring operations such as "Move Method", "Move Field", "Create New Class", etc. Otherwise, the weight $w_i = 0$.

To calculate the similarity score of the whole proposed refactoring solution with historical code changes, we calculate the sum of the similarity value of each refactoring operation in this solution with the base of collected refactorings. The function $ContextSimilarity(C_{RO},\ C_{RO_i})$ aims at calculating the context similarity between the refactoring operation RO applied to the code fragment C_{RO} and the refactoring operation RO_i applied to the code fragments C_{RO_i}. The aim of this measure is to estimate whether two refactorings are in similar contexts. The similarity score is calculated using (1) structure-based and (2) semantic-based metrics (Ouni et al., 2012), to assess the context similarity between original code to be refactored and code from different software projects that underwent refactorings in the past. Context similarity between two refactorings RO and RO_i is defined as follows:

$$ContextSimilarity(RO,\ RO_i) = \alpha \times \frac{(CBO_{sim}(C_{RO}, C_{RO_i}) + D_{sim}(C_{RO}, C_{RO_i}))}{2} + \beta \times SemanticSim(C_{RO}, C_{RO_i}) \quad (2)$$

where $\alpha + \beta = 1$ are the coefficients for the two components representing the structural and semantic similarities. In our experiments, we give equal weight values to both coefficients ($\alpha = \beta = 0.5$). These two parameters can be adjusted according to the maintainer preferences. The obtained values are then normalized in the range [0, 1] using min–max normalization and defined as follow:

4.2.1.1. Structural similarity. This measure can be formalized using quality metrics (Chidamber and Kemerer, 1994). In our approach, we consider CBO (coupling between objects) that was defined by Chidamber and Kemerer (1994) and formulated later in Briand et al.'s metrics catalogue (Briand et al., 1999). We chose CBO because of its ability to capture the context in which a class is located with respect to the rest of classes in the system. Moreover, CBO is one of the most used metrics to detect and fix code-smells (Kessentini et al., 2011; iPlasma, 0000) as well as finding refactoring opportunities (O'Keeffe and Cinnéide, 2006; Harman and Tratt, 2007; Ó Cinnéide et al., 2012), predict changeability, and impact analysis. By definition, CBO counts the number of other classes to which a given class is coupled and, hence, denotes the dependency of one class on other classes in the design. Formally, CBO is defined as follow:

$$\mathrm{CBO}(c) = |\{d \in C - \{c\} \backslash |\ uses(c, d) \vee\ uses(d, c)\}| \quad (3)$$

where C is the set of all classes in the system and uses(x, y) is a predicate that is true if there is a relationship between the two classes x and y e.g. a method call from x to y, an attribute or local variable of type y in x.

To measure the context similarity between a candidate refactoring RO that involves the code fragments C_{RO}, and a recorded refactoring RO_i applied in past that involves the code fragments C_{RO_i}, we define the function $\mathrm{CBO}_{\mathrm{sim}}$ as follow:

$$\mathrm{CBO}_{\mathrm{sim}}\left(C_{RO},\ C_{RO_i}\right) = \left|\mathrm{CBO}\left(C_{RO}\right) - \mathrm{CBO}\left(C_{RO_i}\right)\right| \quad (4)$$

We also consider the dependency-based similarity (D_{sim}; Ouni et al., 2012) when comparing the context similarity. In fact, some refactorings involve only one class to be applied (e.g., extract method), whereas other refactorings involve a pair of classes (e.g., move method, move field, inline class). For the latter, we approximate context similarity between classes starting from their mutual dependencies. The intuition is that classes that are strongly or slightly connected (i.e., having dependency links) are likely to be involved in specific changes such as move method or move field, etc. To compare context similarity in terms of D_{sim} between a candidate refactoring RO that involves the code fragments $C_{\mathrm{RO}} = \{c_1, c_2\}$, and a recorded refactoring RO_i applied in the past that involves the code fragments $C_{RO_i} = \{c'_1, c'_2\}$, we define the function D_{sim} as follow:

$$D_{\mathrm{sim}}\left(C_{RO},\ C_{RO_i}\right) = \left|\mathrm{coup}\left(c_1,\ c_2\right) - \mathrm{coup}\left(c'_1, c'_2\right)\right| \quad (5)$$

where coup(x, y) returns the number of relationships between the two classes x and y, e.g. method calls, attribute or local variable of type x in y or vice versa.

4.2.1.2. Semantic similarity. We consider another kind of coupling to assess the context similarity. This measure is based on vocabulary-based similarity (Soetens et al., 2013), also called conceptual coupling (Poshyvanyk and Marcus, 2006), and uses semantic information obtained from source, encoded in identifiers and comments.

This kind of coupling is interesting to consider when measuring the context similarity of a class with the rest of classes in the system. The vocabulary could be used as an indicator of the semantic similarity between different code elements (e.g., method, field, class, etc.). We start from the assumption that the vocabulary of an actor is borrowed from the domain terminology and therefore could be used to determine which part of the domain semantics is encoded by an actor (e.g., class, method, package, interface, etc.). Thus, two code elements could be semantically similar if they use similar vocabularies. The vocabulary is extracted from the code element identifiers (i.e., names of methods, fields, variables, parameters, types, etc.) as well as comments. Tokenization is performed using the Camel Case Splitter which is one of the most used techniques in Software Maintenance tools for the preprocessing of identifiers. To extract vocabulary from comments, a stop word list is used to cut-off and filter all common English words.[5] In addition, we excluded all the Java keywords

[5] http://www.textfixer.com/resources/common-english-words.txt.

(e.g., Boolean, abstract, class, string, etc.). Furthermore, our semantic similarity utilizes WordNet[6] to enrich the extracted vocabulary. WordNet is a widely used lexical database that groups nouns, verbs, adjectives, etc. into the sets of synsets, i.e., cognitive synonyms, each representing a distinct concept. Then all the extracted vocabulary is stored as a vector where each dimension represents an extracted word. Furthermore, more vocabulary can also be extracted from documentation and commits information. We then calculate the semantic similarity between extracted vocabulary from these code elements (e.g., classes) using information retrieval-based techniques (e.g., cosine similarity). Eq. (6) calculates the cosine similarity between two classes. Each class is represented as a n dimensional vector, where each dimension corresponds to a vocabulary term. The cosine of the angle between two vectors is considered as an indicator of similarity. Using cosine similarity, the semantic similarity between two classes c_1 and c_2 in the same project is determined as follows:

$$Sim(c_1, c_2) = cos(\overrightarrow{c_1} \cdot \overrightarrow{c_2}) = \frac{\overrightarrow{c_1} \cdot \overrightarrow{c_2}}{\left\|\overrightarrow{c_1}\right\| * \left\|\overrightarrow{c_2}\right\|} = \frac{\sum_{i=1}^{n} (w_{i,1} * w_{i,2})}{\sqrt{\sum_{i=1}^{n} (w_{i,1})^2} \sqrt{\sum_{i=1}^{n} (w_{i,2})^2}} \quad (6)$$

where $\overrightarrow{c_1} = (w_{1,1}, \ldots, w_{n,1})$ is the term vector corresponding to actor c_1 and $\overrightarrow{c_2} = (w_{1,2}, \ldots, w_{n,2})$ is the term vector corresponding to c_2. The weights $w_{i,j}$ is computed using information retrieval based technique, i.e., term frequency - inverse term frequency (TF-IDF) method. The vocabulary is gathered and saved as vector.

To calculate the cross-project similarity in terms of semantic similarity between a candidate refactoring RO that involves the code fragments $C_{RO} = \{c_1, c_2\}$, and a recorded refactoring RO_i applied in the past that involves the code fragments $C_{RO_i} = \{c'_1, c'_2\}$, we define the function *SemanticSim* as follow:

$$Semantic\ Sim\ (C_{RO},\ C_{RO_i}) = \left|Sim\ (c_1,\ c_2) - Sim\ (c'_1, c'_2)\right| \quad (7)$$

where $Sim(x, y)$ returns the cosine similarity between the two classes x and y, as defined in Eq. (6). The less the SemanticSim value is, the more the two refactorings are in similar contexts.

To illustrate the context similarity for both structural and semantic measures, let us consider the example of our motivating scenario described in Section 3.2. The aim is to recommend refactoring operations to our running software system JVacation using recorded refactorings from other projects (i.e., JHotDraw and JFreeChart). The list of recorded refactorings includes the following refactoring operations:

R1: move_method(drawPrimaryLinePath(), XYLineAndShapeRenderer, XYSplineRenderer)
R2: move_method(initialize(), XYLineAndShapeRenderer, XYSplineRenderer)
R3: move_method(equals(), XYLineAndShapeRenderer, XYSplineRenderer)
R4: move_method(draw(), ArrowTip, AbstractLineDecoration)

One can recommend a candidate refactoring R: move_method(getCustomerObject(), DBClient, CustomerModel) to JVacation. In this settings, the context similarity between the candidate refactoring R and our recorded refactorings R1, R2, R3 and R4 is calculated as follows:

$$\begin{aligned} SimRefactoringHistory(R) &= \sum_{i=1}^{4} w_{R_i} * ContextSimilarity(R, R_i) \\ &= w_{R_1} * ContextSimilarity(R, R_1) \\ &\quad + w_{R_2} * ContextSimilarity(R, R_2) \\ &\quad + w_{R_3} * ContextSimilarity(R, R_3) \\ &\quad + w_{R_4} * ContextSimilarity(R, R_4) \end{aligned}$$

where:

According to Table 1: $W_{R_1} = W_{R_2} = W_{R_3} = W_{R_4} = 2$

And according to Eq. (2):

$$\begin{aligned} &ContextSimilarity\ (R, R_1) \\ &= ContextSimilarity\ (R, R_2) = ContextSimilarity\ (R, R_3) \\ &= ContextSimilarity(\text{move method (DBClient, CustomerModel),} \\ &\quad \text{move method (XYLineAndShapeRenderer, XYSplineRenderer))} \\ &= 0.5 * (\text{CBO}_{\text{sim}}\ (C_R, C_{R_1}) + D_{\text{sim}}\ (C_R, C_{R_1}))\ /2 \\ &\quad + 0.5 * \left|SemanticSim\ (C_R) - SemanticSim\ (C_{R_1})\right| \end{aligned}$$

where:

$$\begin{aligned} &\text{CBO}_{\text{sim}}\ (C_R, C_{R_1}) \\ &= |\text{CBO(DBClient)} - \text{CBO(XYLineAndShapeRenderer)}| \\ &\quad + |\text{CBO(CustomerModel)} - \text{CBO (XYSplineRenderer)}\ | \\ &= |38 - 30| + |7 - 10| = 11 \text{ (Eq. (4))} \end{aligned}$$

and,

$$\begin{aligned} &D_{\text{sim}}(C_R, C_{R_1}) \\ &= |\text{coup(DBClient, CustomerModel)} \\ &\quad - \text{coup(XYLineAndShapeRenderer, XYSplineRenderer)}| \\ &= |7 - 5| = 2 \text{ (Eq. (5))} \end{aligned}$$

According to Eq. (7):

$$\begin{aligned} &SemanticSim(\text{C}_\text{R}, \text{C}_{\text{R1}}) \\ &= |\text{sim(DBClient, CustomerModel)} \\ &\quad - \text{sim(XYLineAndShapeRenderer, XYSplineRenderer)}| \\ &= |0.39 - 0.5| = 0.11 \end{aligned}$$

After normalizing each of the CBO_{sim} and D_{sim} values in the scale [0, 1] using min-max normalization, we obtain:

$$\begin{aligned} ContextSimilarity\ (R, R_1) &= ContextSimilarity\ (R, R_2) \\ &= ContextSimilarity\ (R, R_3) = 0.31 \end{aligned}$$

Similarly:

$$ContextSimilarity\ (R, R_4) = 0.36$$

Finally,

$$\begin{aligned} &SimRefactoringHistory(R) \\ &= W_{R_1} * ContextSimilarity\ (R, R_1) \\ &\quad + W_{R_2} * ContextSimilarity\ (R, R_2) \\ &\quad + W_{R_3} * ContextSimilarity\ (R, R_3) \\ &\quad + W_{R_4} * ContextSimilarity\ (R, R_4) \\ &= 2 * 0.31 + 2 * 0.31 + 2 * 0.31 + 2 * 0.36 = 2.58 \end{aligned}$$

The aim behind these structural and semantic similarities is to find patterns and regularities when applying refactoring. For instance, as shown in the previous example, the case of move method refactoring is often applied between semantically and structurally similar classes. The more CBO_{sim}, D_{sim} and semantic similarity values between the pair of classes being refactored are close to the ones of collected refactorings, the more the suggested refactoring is encouraged.

4.2.2. Change frequency

In a typical software system, important code fragments are those who change frequently during the development/maintenance process to add new functionalities, to accommodate new changes or to improve its structure. Moreover, as reported in the literature (Olbrich et al., 2010; Olbrich et al., 2009; Khomh et al., 2009), classes participating in design problems (e.g., code-smells) are significantly more likely to be subject to changes and to be involved in fault-fixing changes

[6] wordnet.princeton.edu.

(bugs; Khomh et al., 2009). Indeed, if a class undergoes many changes in the past and it is still smelly, so it needs to be fixed as soon as possible. On the other hand, not every code-smell is assumed to have negative effects on the maintenance/evolution of a system. It has been shown that in some cases, a large class might be the best solution (Olbrich et al., 2010). Moreover, it is reported that if a code-smell (e.g., god class) is created intentionally and remains unmodified or did not change frequently, the system may not experience any problems (Olbrich et al., 2010). For these reasons, code-smells related to more frequently changed classes should be prioritized during the correction process.

This score represents the number of changes applied during the past to the same code elements to modify. If this number is high, then we can conclude that this is a good indication that this code fragment is badly designed, thus represents a refactoring opportunity. This second measure is defined as:

$$ChangeFrequency\,(RO) = \sum_{i=1}^{n} t\,(e) \tag{8}$$

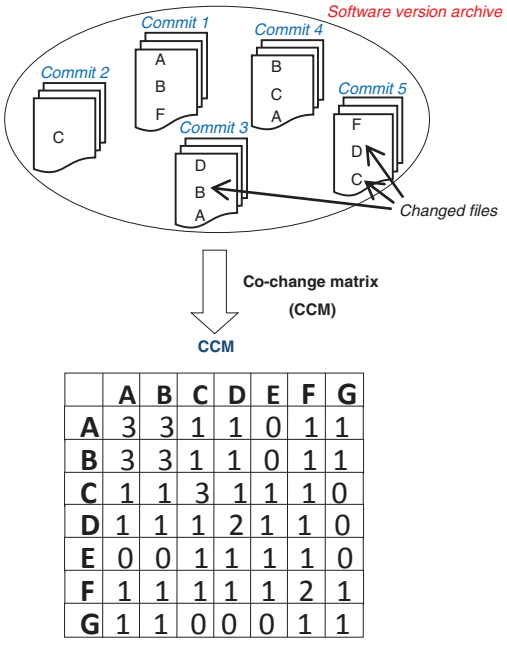
where $t(e)$ is the number of times that the code element(s) e was refactored in the past and n is the size of the list of possible refactoring operations.

4.2.3. Co-change

Recently, research has been carried out to mining software version repositories (Gall et al., 1998; Zimmermann et al., 2005; Beyer and Noack, 2005; Gall et al., 1998). One of the important challenges addressed is to detect and interpret groups of software entities that change together. These co-change relationships have been used for different purposes: to identify hidden architectural dependencies (Gall et al., 1998), to point developers to possible places that need change (Zimmermann et al., 2005), or to use them for software clustering (Beyer and Noack, 2005). Detecting historical co-changes is mostly based on mining versioning systems such as CVS and in identifying pairs of changed entities. Entities are usually files, and the change is determined through observing additions or deletions of lines of code. In the past decades, many CVS archives of open-source and industrial systems are freely available, e.g., via SourceForge.net.

We analyze CVS repositories to identify which classes have been changed together at the same time. To this end, we use only source code files to extract co-change; however, other non-useful files (e.g., documentation, configuration files, etc.) are filtered out. We consider that when a source code file has been changed then each class in this file is changed (almost only one or two classes per file are implemented in actual systems). Each commit/revision contains a lot of information such as the file name, the commit/revision number, the developer who performed the commit, the log message, and the timestamp of the check-in. Co-change can be easily detected by mining version archives, and is less expensive than program analysis (coupling, cohesion, etc.) (Ying et al., 2004). We depict in Fig. 3 an example of how to detect the co-change between classes through different commits from version history archives to generate a co-change matrix CCM. $CCM_{i,j}$ represented how many times the class i have been changed at the same (i.e., in the same commit)time with class j (co-change). In our adaptation, code changes are applied at the same time if they are introduced in the same commit. For example, we can observe that almost when the class A change, the class B change also. This reveals that the implementations and semantics of these two classes are strongly related. Therefore, moving code elements (methods/attributes) between them is likely to be semantically coherent since many changes have been manually performed between them in the past by software developers.

In the next sections we give an overview of NSGA-II, and its adaptation for the problem of refactoring using the change history.



	A	B	C	D	E	F	G
A	3	3	1	1	0	1	1
B	3	3	1	1	0	1	1
C	1	1	3	1	1	1	0
D	1	1	1	2	1	1	0
E	0	0	1	1	1	1	0
F	1	1	1	1	1	2	1
G	1	1	0	0	0	1	1

Fig. 3. Co-changes matrix.

4.3. NSGA-II overview

The basic idea of NSGA-II (Deb et al., 2002) is to make a population of candidate solutions evolve toward the near-optimal solution in order to solve a multi-objective optimization problem. NSGA-II is designed to find a set of optimal solutions, called non-dominated solutions, also Pareto set. A non-dominated solution is the one which provides a suitable compromise between all objectives without degrading any of them. As described in Algorithm 1, the first step in NSGA-II is to create randomly a population P_0 of individuals encoded using a specific representation (line 1). Then, a child population Q_0 is generated from the population of parents P_0 using genetic operators such as crossover and mutation (line 2). Both populations are merged into an initial population R_0 of size N (line 5).

Fast-non-dominated-sort (Deb et al., 2002) is the algorithm used by NSGA-II to classify individual solutions into different dominance levels. Indeed, the concept of Pareto dominance consists of comparing each solution x with every other solution in the population until it is dominated by one of them. If no solution dominates it, the solution x will be considered non-dominated and will be selected by the NSGA-II to be one of the set of Pareto front. If we consider a set of objectives f_i , $i \in 1 \ldots n$, to maximize, a solution x dominates x'

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

The whole population that contains N individuals (solutions) is sorted using the dominance principle into several fronts (line 6). Solutions on the first Pareto-front F_0 get assigned dominance level of 0.Then, after taking these solutions out, fast-non-dominated-sort calculates the Pareto-front F_1 of the remaining population; solutions on this second front get assigned dominance level of 1, and so on. The dominance level becomes the basis of selection of individual solutions for the next generation. Fronts are added successively until the parent population P_{t+1} is filled with N solutions (line 8). When NSGA-II has to cut off a front F_i and select a subset of individual solutions with the same dominance level, it relies on the crowding distance (Deb et al., 2002) to make the selection (line 9). This parameter is used to promote diversity within the population. This front F_i to be split, is sorted in descending order (line 13), and the first (N-$|P_{t+1}|$) elements of F_i are chosen (line 14). Then a new population Q_{t+1} is created using selection, crossover and mutation (line 15). This process will be

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

Algorithm 1. High-level pseudo-code of NSGA-II.

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

4.4. NSGA-II design

This section describes how NSGA-II (Deb, 2009) can be used for the refactoring suggestion problem. In general, to apply NSGA-II to a specific problem, the following elements have to be defined: representation of the individuals (solution coding), evaluation of individuals using a fitness function for each objective to optimize, selection of the individuals to transmit from one generation to another, creation of new individuals using genetic operators (crossover and mutation) to explore the search space, and generation of a new population.

4.4.1. Solution coding

As defined in the previous sections, a solution comprises a sequence of n refactoring operations applied to certain elements in the source code under refactoring. To represent a candidate solution (individual), we use a vector-based representation. Each dimension of the vector represents a refactoring operation where the order of application of the refactoring operations corresponds to their positions in the vector. The standard approach of pre- and post-conditions (Fowler et al., 1999; Opdyke, 1992) is used to ensure that the refactoring operation can be applied while preserving program behavior. For each refactoring operation, a set of controlling parameters (e.g., actors and roles as illustrated in Table 2) is randomly picked from the program to be refactored. Assigning randomly a sequence of refactorings to certain code fragments generates the initial population. An example of a solution is given in Fig. 4 containing eight refactorings. To apply a refactoring operation we need to specify which actors, i.e., code fragments, are involved/impacted by this refactoring and which roles, they play to perform the refactoring operation. An actor can be a package, class, field, method, parameter, statement or variable. Table 2 depicts, for each refactoring, its involved actors and its role.

To generate an initial population, we start by defining the maximum vector length (maximum number of operations per solution).

move field (Person, Employee, salary)
extract class(Person, Adress, streetNo, city, zipCode, getAdress(), updateAdress())
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 (Student, Course, courseName, CourseCode, addCourse(), rejectCourse()

Fig. 4. Solution coding.

Table 2
Refactoring operations and their controlling parameters.

Refactoring operation	Actors	Roles
Move method	Class Method	Source class, target class Moved method
Move field	Class Field	Source class, target class Moved field
Pull up field	Class Field	Sub classes, super class Moved field
Pull up method	Class Method	Sub classes, super class Moved method
Push down field	Class Field	Super class, sub classes Moved field
Push down method	Class Method	Super class, sub classes Method
Inline class	Class	Source class, target class
Extract class	Class Field Method	Source class, new class Moved fields Moved methods
Move class	Package Class	Source package, target package Moved class
Extract interface	Class Field Method	Source classes, new interface Moved fields Moved methods

The vector length is proportional to the number of refactorings that are considered, the size of the program being refactored, and the initial number of detected code-smells. 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 software maintainer or derived randomly from the sizes of the program, the given refactoring list, and the initial number of detected code-smells. During the creation, the solutions have random sizes inside the allowed range. To create the initial population, we normally generate a set of *PopSize* solutions randomly in the solution space.

4.4.2. 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 population, 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.

4.4.3. Genetic operators

Two genetic operators are defined: the crossover and mutation operators. To better explore the search space, in this paper, we extend our previous definition of genetic operators used in Ouni et al. (2013c).

4.4.3.1 Mutation. Mutation operator simply picks at random some positions in the vector and changes them by other refactoring operations. For each of the selected refactorings, we apply randomly one of the following possible changes using equal probabilities:

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

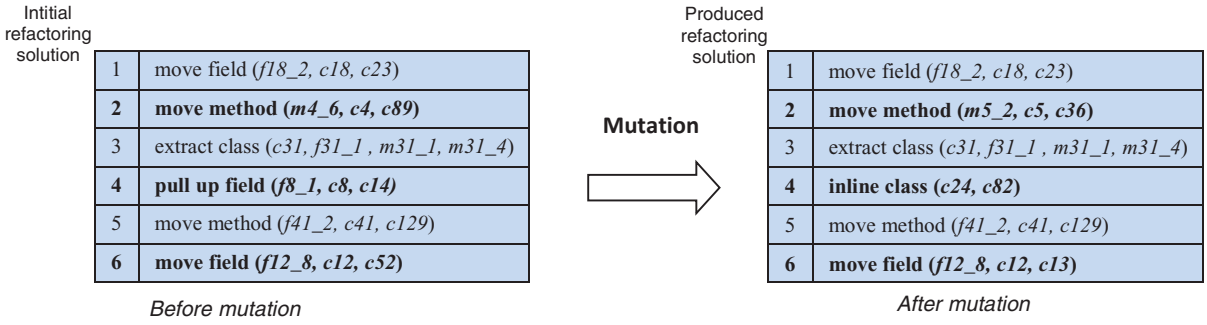


Fig. 5. Example of mutation operator.

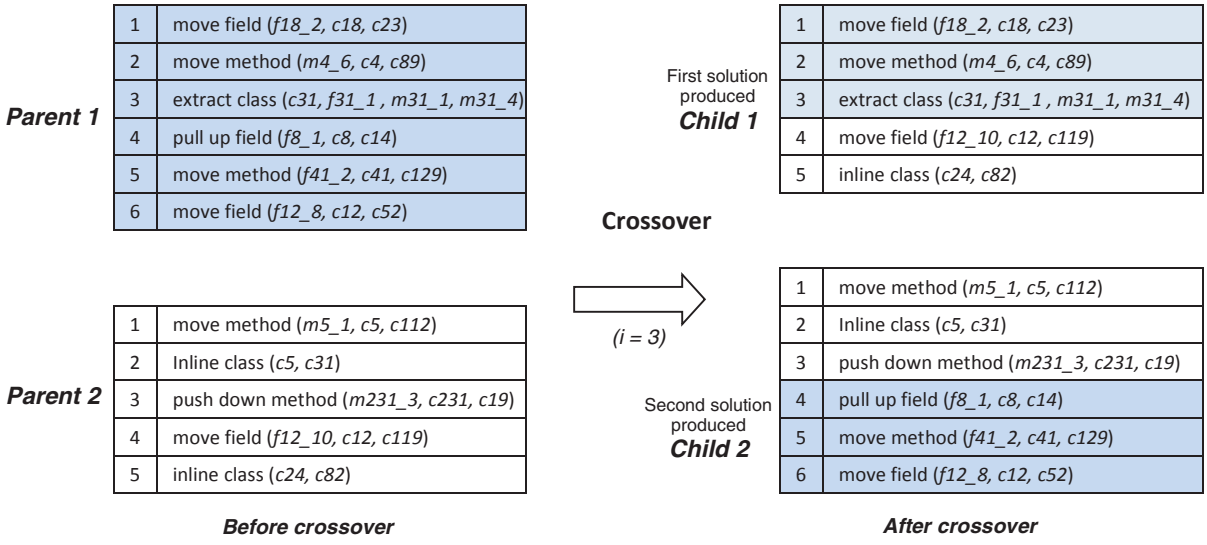


Fig. 6. Example of crossover operator.

source and/or target classes by other classes that can be selected from the whole system.

A mutation example is shown in Fig. 5. Three refactorings are randomly selected from the initial vector: one *refactoring type change* (dimension number 4), and two *controlling parameters change* (dimensions number 2 and 6).

4.4.3.2 Crossover. 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 Fig. 5, 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. An example of crossover is given by Fig. 6.

4.4.4. Fitness functions

After creating a solution, it should be evaluated using fitness function to ensure its ability to solve the problem under consideration. The solution proposed in this paper is based on studying how to preserve the way how code elements are semantically grouped and connected together when refactorings are decided automatically, and how to use the development change history to automate refactoring suggestion. Semantic preservation is captured by different heuristics/measures that could be integrated into existing refactoring approaches to help preserving semantic coherence. Since we have three objectives, we define three different fitness functions:

(1) *Quality fitness function* calculates the ratio of the number of corrected code-smells over the initial number of code-smells using detection rules (Ouni et al., 2013a).
(2) *Semantic fitness function* corresponds to the weighted sum of semantic measures (Ouni et al., 2012): vocabulary similarity (cosine similarity), and structural dependency (shared method calls and shared field access) used to approximate the semantic proximity between modified code elements. Hence, the semantic fitness function of a refactoring solution corresponds to the average of the semantic measure for each refactoring operation in the vector.
(3) *History of changes' fitness function* calculates an average of three measures described in Section 3.2: (a) similarity with previous refactorings applied to similar contexts, (b) number of changes applied in the past to the same code elements to modify (change frequency), and (c) a score that characterizes the co-change of code elements (actors) that will be refactored. In the case when the program is newly developed or if the development history is not available, only the first measure is considered.

5. Evaluation

In this section, we describe the evaluation of our approach, including the research questions to address, the studied open-source

systems, the evaluation metrics considered in our experiments, and the results' analysis.

5.1. Research questions and objectives

In our study, we assess the performance of our refactoring approach by finding out whether it could generate meaningful sequences of refactorings that fix code-smells while preserving the construct semantics of the original design, and reusing as much as possible the development history applied to similar contexts. Our study refines and extends the research questions presented in our previous work (Ouni et al., 2013c). We conduct now a quantitative and qualitative evaluation on five subject systems instead of two, an additional type of code smells, and two new evaluation metrics to better evaluate the scalability and the performance of our approach. Consequently, we address the following research questions:

- **RQ1.1 (SBSE validation):** *How does NSGA-II perform compared to random search?* We compared our NSGA-II formulation with random search to make sure that there is a need for an intelligent method to explore the search-space. The next two sub-questions are related to the comparison between our proposal and other multi and mono-objective metaheuristics that address also the refactoring problem within the same formulation. These three questions are a standard 'baseline' questions asked in any attempt at an SBSE formulation.
- **RQ1.2 (SBSE validation):** *How does NSGA-II perform compared to another multi-objective algorithm in terms of code-smells correction and refactoring suggestion meaningfulness?* We compare NSGA-II with another popular multi-objective algorithm, MOGA (multi-objective genetic algorithm) (Zitzler and Thiele, 1998), using the same adaptation (solution coding, fitness functions, etc.) to justify the use of NSGA-II.
- **RQ1.3 (SBSE validation):** *How does our approach using NSGA-II performs compared to mono-objective evolutionary algorithms?* This comparison is required to ensure that the refactoring solutions, found by a multi-objective search (i.e., NSGA-II and MOGA) provide better trade-offs between quality and refactoring meaningfulness, than those found by a mono-objective approach, e.g., genetic algorithm (GA) (Fonseca and Fleming, 1993). Otherwise, there is reason to use multi-objective optimization.
- **RQ2 (Efficiency):** *To what extent can the proposed approach correct code-smells in the situation where the change history is not available?* We compare the efficiency of our approach in fixing code-smells in both situations where the change history is available and where it is not available.
- **RQ3 (Usefulness):** *To what extent can the proposed approach recommend meaningful refactorings in the situation where the change history is not available?* We compare the meaningfulness of the recommended refactorings in both situations where the change history is available and where it is not available.
- **RQ4 (Comparison to state-of-the-art):** *How does our NSGA-II approach perform comparing to existing refactoring approaches?* A comparison with existing refactoring approaches (Kessentini et al., 2011; Ouni et al., 2012) and (Harman and Tratt, 2007) is helpful to evaluate the efficiency of our proposed approach.
- **RQ5 (Insight):** *How our multi-objective refactoring approach can be useful for software engineers in real-world setting?* The expected benefit from refactoring is not limited to fixing code-smells but to improve also the user understandability, reusability, flexibility, as well as the effectiveness of the refactored program. To this end, it is important to also evaluate the benefit of our approach from this perspective.

Table 3
Systems statistics.

Systems	Release	# classes	# code-smells	KLOC	# previous code-changes
Xerces-J	v2.7.0	991	87	240	7493
JFreeChart	v1.0.9	521	84	170	2009
GanttProject	v1.10.2	245	45	41	91
Artofllusion	v2.8.1	459	68	87	594
JHotDraw	V7.0.6	468	16	57	1006

Table 4
Analyzed versions and data collection.

Systems	Release	Collected refactorings		# previous code-changes
		Previous releases	# Refactorings	
Xerces-J	v2.7.0	v1.4.2–v2.6.1	70	7493
JFreeChart	v1.0.9	v1.0.6–v1.0.9	76	2009
GanttProject	v1.10.2	v1.7–v1.10.2	91	91
Artofllusion	v2.8.1	v1.2–v1.8.2	247	594
JHotDraw	v7.0.6	v5.1–v6.1	64	1006

5.2. Experimental setting

5.2.1. Systems studied

We applied our approach to a benchmark composed of five medium and large-size well-known open-source java projects: Xerces-J,[7] JFreeChart,[8] GanttProject,[9] ArtOfIllusion,[10] and JHotDraw.[11] Xerces-J is a family of software packages for parsing XML. JFreeChart is a powerful and flexible Java library for generating charts. GanttProject is a cross-platform tool for project scheduling. JHotDraw is a GUI framework for drawing editors. Finally, Art of Illusion is a 3D-modeller, renderer and raytracer written in Java.

Table 3 provides some descriptive statistics about these five software systems. We selected these systems for our validation because they range from medium to large-sized open-source projects, which have been actively developed over the past 10 years, and their design has not been responsible for a slowdown of their development. In addition, these systems are well studied in the literature, and their code-smells have been detected and analyzed manually (Moha et al., 2010; Kessentini et al., 2010, 2011; Ouni et al., 2013a). In these corpuses, the four following code smell types were considered: Blob (a large class that monopolizes the behavior of a system or part of it, and the other classes primarily encapsulate data); Data Class (a class that contains only data and performs no processing on these data); Spaghetti Code (code with a complex and tangled control structure) and Functional Decomposition (when a class performs a single function, rather than being an encapsulation of data and functionality). We chose these code-smell types in our experiments because they are among the most common ones that are detected and corrected in recent studies and existing corpuses (Kessentini et al., 2010, 2011; Ouni et al., 2013a; Moha et al., 2010).

To collect refactorings applied in previous versions of the studied systems, we use Ref-Finder (Prete et al., 2010). Ref-Finder is a tool implemented as an Eclipse plug-in that identifies refactoring operations between two consecutive releases of a software system. Table 4 shows the analyzed versions and the number of refactoring operations collected using Ref-Finder between each subsequent couple of analyzed versions, after a manual validation. In our study, we consider only the set of refactoring types described in Table 2.

[7] http://xerces.apache.org/xerces-j/.
[8] http://www.jfree.org/jfreechart/.
[9] www.ganttproject.biz.
[10] www.artofillusion.org/.
[11] http://www.jhotdraw.org/.

5.2.2. Analysis method

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

5.2.2.1. Qualitative evaluation. To evaluate the usefulness of the suggested refactorings, we performed a qualitative evaluation with three Ph.D. students in Software Engineering; two of whom are working at General Motors as senior software engineers. The subjects have an average of 6.5 years programming experience in Java and familiar with the five studied systems. We asked the participants to manually evaluate, for each system, a sample of 10 refactoring operations that are selected at random from the suggested refactoring solutions. To this end, we asked them to apply the proposed refactorings using Eclipse[12] and check the semantic coherence of the modified code fragments. When a construct semantic incoherency is found manually, participants consider the refactorings related to this change as a bad recommendation. In this setting, participants assign a meaningfulness score in the range [0, 5] for each refactoring according to its coherence with the program semantics. We consider a refactoring operation as meaningful if its assigned score is greater than or equal to 3. Participants were aware that they are going to evaluate the semantic coherence of refactoring solutions, but do not know the particular experimental research questions (the approaches and algorithms being compared). To this end, we define the metric refactoring meaningfulness (RM) that corresponds to the number of meaningful refactoring operations, in terms of construct semantic coherence, over the total number of evaluated refactorings. RM is given by Eq. (9).

$$\text{RM} = \frac{\#\ meaningful\ refactorings}{\#\ evaluated\ refactorings} \tag{9}$$

5.2.2.2. Quantitative evaluation. To answer **RQ1.∗,** we used mainly two performance indicators to compare the different algorithms used in our experiments. When comparing two mono-objective algorithms, it is usual to compare their best solutions found so far during the optimization process. However, this is not applicable when comparing two multi-objective evolutionary algorithms since each of them gives as output a set of non-dominated (Pareto equivalent) solutions. Different metrics for measuring the performance of multi-objective optimization methods exist. Zitzler et al. (2003) provide a comprehensive review of quality indicators for multi-objective optimization, finding that many commonly used metrics do not reliably reflect the performance of an optimization algorithm. One of the few recommended metrics is the *Hypervolume* and the *Spread* indicators.

- *Hypervolume (HV)*: This metric calculates the proportion of the volume covered by members of a non-dominated solution set returned by the algorithm. A higher Hypervolume value means better performance, as it indicates solutions closer to the optimal Pareto Front. The most interesting features of this indicator are its Pareto dominance compliance and its ability to capture both convergence and diversity.
- *Spread (Δ)*: It measures the distribution of solutions into a given front. The idea behind the spread indicator is to evaluate diversity among non-dominated solutions. An ideal distribution has zero value for this metric when the solutions are uniformly distributed. An algorithm that achieves a smaller value for Spread can get a better diverse set of non-dominated solutions. For further details about the formulation of these indicators, please refer to Deb (2009) and Zitzler et al. (2003).

In addition to these two multi-objective performance evaluation measures, we used other metrics to assess the efficiency and the usefulness of our approach and to compare mono-objective and multi-objective approaches.

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

To answer **RQ2**, we compare the refactoring results in terms of code-smells correction ratio in the situation where the change history is available and where it is not. To this end, we define the metric code-smells correction ratio (CCR). CCR is given by Eq. (10) and calculates the number of corrected code-smells over the total number of code-smells detected before applying the proposed refactoring sequence.

$$\text{CCR} = \frac{\#\ corrected\ code\_smells}{\#\ code\_smells\ before\ applying\ refactorings} \in [0, 1] \tag{10}$$

To answer **RQ3**, we also consider both cases where the refactoring history is available and where it is not available. We used mainly a qualitative validation based on the refactoring meaningfulness (RM) as described in Eq. (9). To this end, we perform a five-fold cross validation. Each fold consists of a set of refactorings collected from four systems as a base of historical refactorings and the remaining system as the test system (we consider that its development history is not available). In other words, we execute our approach on this system while calculating the cross-project refactoring similarity with the other four projects. In this setting, the higher the RM metric is, the more the change history from other projects in similar context is useful.

To answer **RQ4**, we compared our refactoring results to those produced by three state-of-the-art approaches (Harman and Tratt, 2007; Kessentini et al., 2011; Ouni et al., 2012). Harman and Tratt (2007) proposed a multi-objective approach that uses two quality metrics (coupling between objects, and standard deviation of methods per class) to improve after applying the refactorings sequence. In Kessentini et al. (2011), a single-objective genetic algorithm is used to correct code-smells by finding the best refactoring sequence that reduces the number of code-smells. Ouni et al. (2012) have proposed a new multi-objective refactoring to find the best compromise between quality improvement and semantic coherence using two heuristics related to the vocabulary similarity and structural coupling. In all contributions, the development history is not used explicitly. We evaluate the efficiency of our refactoring solutions in producing more coherent changes than those produced by Harman and Tratt (2007), Kessentini et al. (2011), and, Ouni et al. (2013a).

Finally, to answer **RQ5**, we assess the benefit of the suggested refactoring solutions on software quality. Thus, although our primary goal in this work is to demonstrate that code-smells can be automatically refactored using change history information, it is also important to assess the refactoring impact on the design quality. The expected benefit from refactoring is to enhance the overall software design quality as well as fixing code-smells (Fowler et al., 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 (O'Keeffe and Cinnéide, 2006; Jensen and Cheng, 2010; Seng et al., 2006) to assess the effect of refactoring, and (2) it has the advantage that define six high level design quality attributes (reusability, flexibility, understandability, functionality, extendibility, and effectiveness) that are 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.

Table 5
QMOOD quality factors (Bansiya and Davis, 2002).

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

Table 6
QMOOD quality 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

- 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 functionality factor 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 relatively a subjective quality factor; and, still, using a model of merely static measures to evaluate extendibility is inadequate. Tables 5 and 6 summarize the QMOOD formulation of these quality attributes (Bansiya and Davis, 2002). For space reasons, we omit references for the metric definitions.

The quality gain is calculated by comparing each of the quality attributes values before and after refactoring independently to the number of fixed code-smells. Hence, the total gain in quality G for each of the considered QMOOD quality attributes q_i before and after refactoring can be estimated as:

$$G_{q_i} = q_i' - q_i \tag{11}$$

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

5.2.3. Used inferential statistical methodology

Due to the stochastic nature of the algorithms/approaches we are studying, they can provide different results for the same problem instance from one run to another. To cater for this issue and to make inferential statistical claims, our experimental study is performed based on 31 independent simulation runs for each algorithm/technique studied. The obtained results are statistically analyzed using the Wilcoxon rank sum test (Arcuri and Briand, 2011) with a 95% confidence level ($\alpha = 5\%$). Wilcoxon rank sum test (Arcuri and Briand, 2011) is applied between NSGA-II and each of the other algorithms/techniques (Kessentini et al., 2011, Ouni et al., 2012, GA, MOGA, and random search) in terms of CCR. Our tests show that the obtained results are statistically significant with p-value$<$0.05.

In the result reported in this paper, we are considering the median value for each approach through 31 independent runs. The Wilcoxon rank sum test allows verifying whether the results are statistically different or not, however it does not give an idea about the difference magnitude. In order to quantify the latter, we compute the effect size by using the Cohen's d statistic (Cohen, 1988). The effect size is considered: (1) *small* if $0.2 \leq d < 0.5$, (2) *medium* if $0.5 \leq d < 0.8$, or (3) *large* if $d \geq 0.8$.

Furthermore, as for our evaluation we need to select only one solution from the entire Pareto front, with the same manner for all the trial runs, we use a technique similar to the one described in Ouni et al. (2013a). Eq. (12) is used to choose the solution that corresponds of the best compromise between our three objective functions: quality, semantic coherence, and context similarity. In our case the ideal solution has the best quality value (equals to 1), the best semantic coherence value (equals to 1), and the highest context similarity value (equals to 1 after normalization). Hence, we select the nearest solution to the ideal point in terms of Euclidian distance.

$$bestSol = \min_{i=0}^{n-1} \left(\sqrt{(1 - Quality[i])^2 + (1 - Semantic[i])^2 + +(1 - ContextSimilarity[i])^2} \right) \tag{12}$$

where n is the number of solutions in the Pareto front returned by NSGA-II.

5.3. Experimental results

This section describes and discusses the results obtained for the different research questions of Section 5.1.

Results for RQ1 (SBSE validation): For the first research question RQ1.1, we describe the obtained results in terms of the two used quality indicators: Hypervolume (HV) and Spread (Δ). For the HV, the higher the value is, the better is the quality of the obtained results. Fig. 7 describes our finding through 31 runs of NSGA-II, MOGA, and random search. According to the obtained results in Fig. 7(a), random search results are generally poor, whereas NSGA-II and MOGA obtain good results for all the five systems. Moreover, as illustrated in Fig. 7(a), NSGA-II significantly outperforms MOGA when applied to Xerces, GanttProject, and JHotDraw while presenting comparable performance for JFreeChart and ArtOfIllusion. This provides evidence the effectiveness of NSGA-II over MOGA in finding a well-converged and well-diversified set of Pareto-optimal refactoring solutions. For the Δ, it is also desired that a multi-objective evolutionary algorithm maintains a good spread of solutions in the obtained set of solutions. Fig. 7(b) shows that NSGA-II has the better spread among all the other search-based algorithms in all systems.

To summarize, the Wilcoxon rank sum test showed that in 31 runs, both NSGA-II and MOGA results were significantly better than random search. We conclude that there is an empirical evidence that our multi-objective formulation surpasses the performance of random search thus our formulation is adequate (this answers RQ1.1).

Another element that should be considered when comparing the results of the three algorithms is that multi-objective evolutionary algorithms do 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 his preferences in terms of compromise. However, at least for our evaluation, we need to select only one solution. To make the comparison fair, we use Eq. (12) as described in Section 5.2.3 to automatically select a candidate solution from the Pareto front.

To answer the next research question, RQ1.2, we first compared NSGA-II to another widely used multi-objective algorithm, MOGA, using the same adapted fitness functions, solution representation, and genetic operators. Then, to answer RQ1.3 we compared NSGA-II to a widely used single-objective evolutionary algorithm, genetic algorithm (GA), using a single fitness function that calculates the average of the three measures we use (quality, construct semantics preservation, and development history reuse). Fig. 8 reports the comparison results between NSGA-II, MOGA and GA in terms of CCR and RM

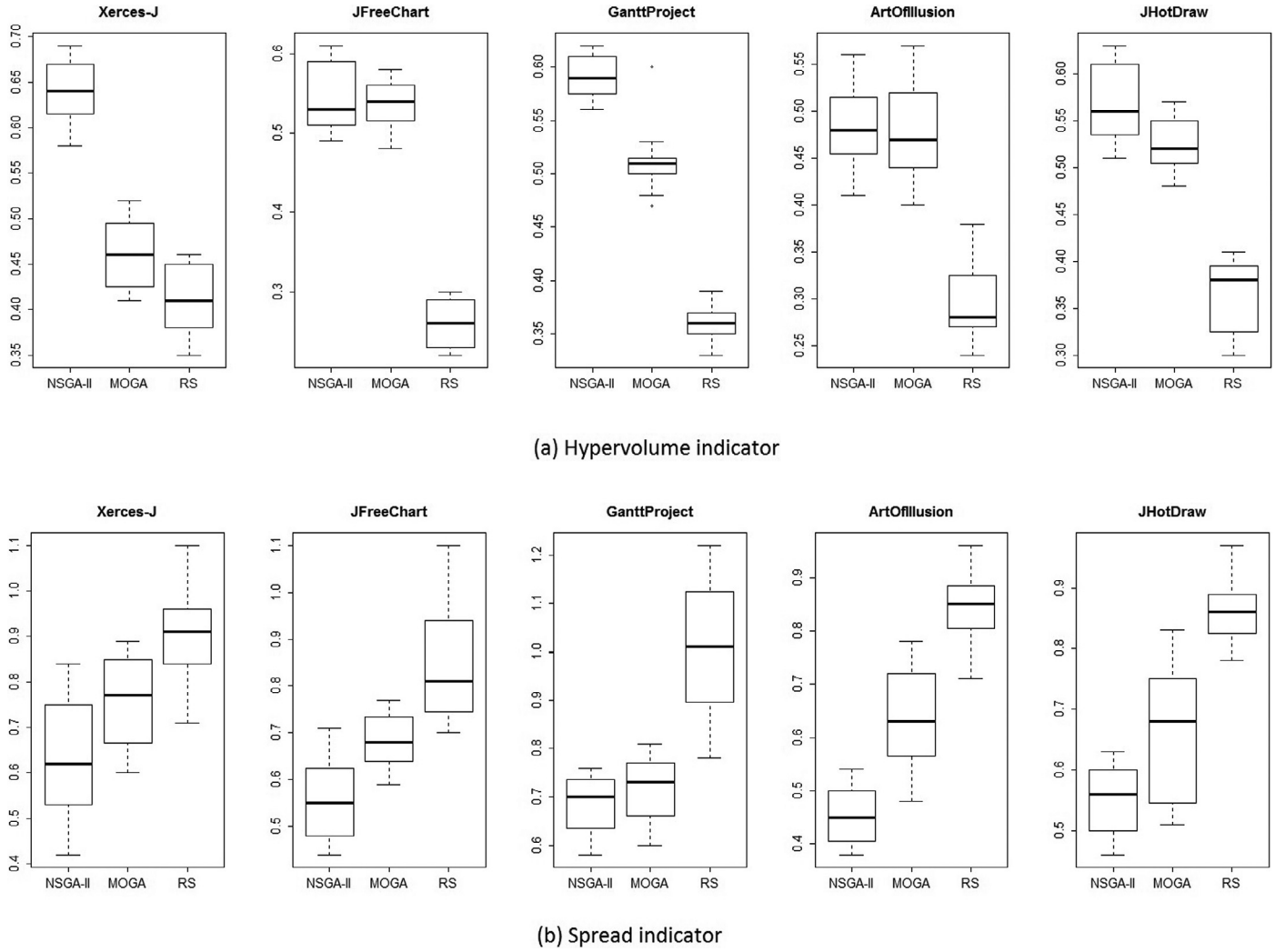


Fig. 7. Boxplots using the quality measures (a) HV, and (b) spread applied to NSGA-II, MOGA, and random search through 31 independent runs.

median values representing obtained over 31 independent simulation runs of the three algorithms. The Wilcoxon test provides evidence that all the CCR and RM values obtained employing NSGA-II were significantly better than those obtained with each of MOGA. Fig. 8 shows that both NSGA-II and MOGA provide the best results regarding the CCR and RM metrics. Moreover, NSGA-II outperforms MOGA in most of the cases for both CCR and RM. MOGA outperforms the NSGA-II approach only in GanttProject in terms of CCR, which is the smallest open source system considered in our experiments, having the lowest number of fixed code-smells (89% of CCR for MOGA, only 87% for NSGA-II), so it appears that MOGA's search operators make a better task of working with a smaller search space. In terms of refactoring meaningfulness (RM), our experiments reported in Fig. 8b provides evidence that NSGA-II gives better results for the five systems with an average of 86%, whereas only 76% and 70% are obtained for MOGA and GA respectively. In addition, NSGA-II outperforms MOGA in terms of both performance indicators HV and Δ as illustrated in Fig. 7. Regarding HV, NSGA-II outperforms significantly MOGA in three out of five systems. For the Δ indicator, NSGA-II outperforms MOGA in all studied systems.

Results for RQ2 (Efficiency): Fig. 9 presents the obtained results. We observe that the majority of suggested refactorings by our approach (where the change history is not available) succeeded in improving significantly the code quality with good correction scores (an average of 85% of detected code-smells was fixed for all the five systems). These results are comparable to the ones obtained by the original approach where the change history is available (an average of 84% for all systems). We observe that our approach achieved slightly superior CCR values with respect to the original approach in two out five projects, and identical results in one out of five projects. For instance, for GanttProject, both approaches provides identical results with 90% of CCR score; while for JHotDraw, our approach provides slightly better CCR results (88%) comparing to the original approach having a CCR score of 81%.

As interesting observation from the results of Fig. 9 is that the CCR medians are close, the results are statistically different but the effect size (Cohen, 1988) which quantifies the difference is small for most of the systems considered in our experiments. The Wilcoxon rank sum test allows verifying whether the results are statistically different or not. However, it does not give any idea about the difference magnitude. The statistical analysis of the obtained results provides evidence that the cross-project similarity did not have a significant effect on CCR on almost all systems, especially for medium and large systems.

Results for RQ3 (Usefulness): Fig. 10 reports the obtained results of the RM values for our approach (using cross-project similarity) and the original one (when the change history is available). For this evaluation measurement, our approach achieved similar results comparing to the original one. In three systems, there were identical distributions between both approaches (i.e., on the systems ArtOfIllusion,

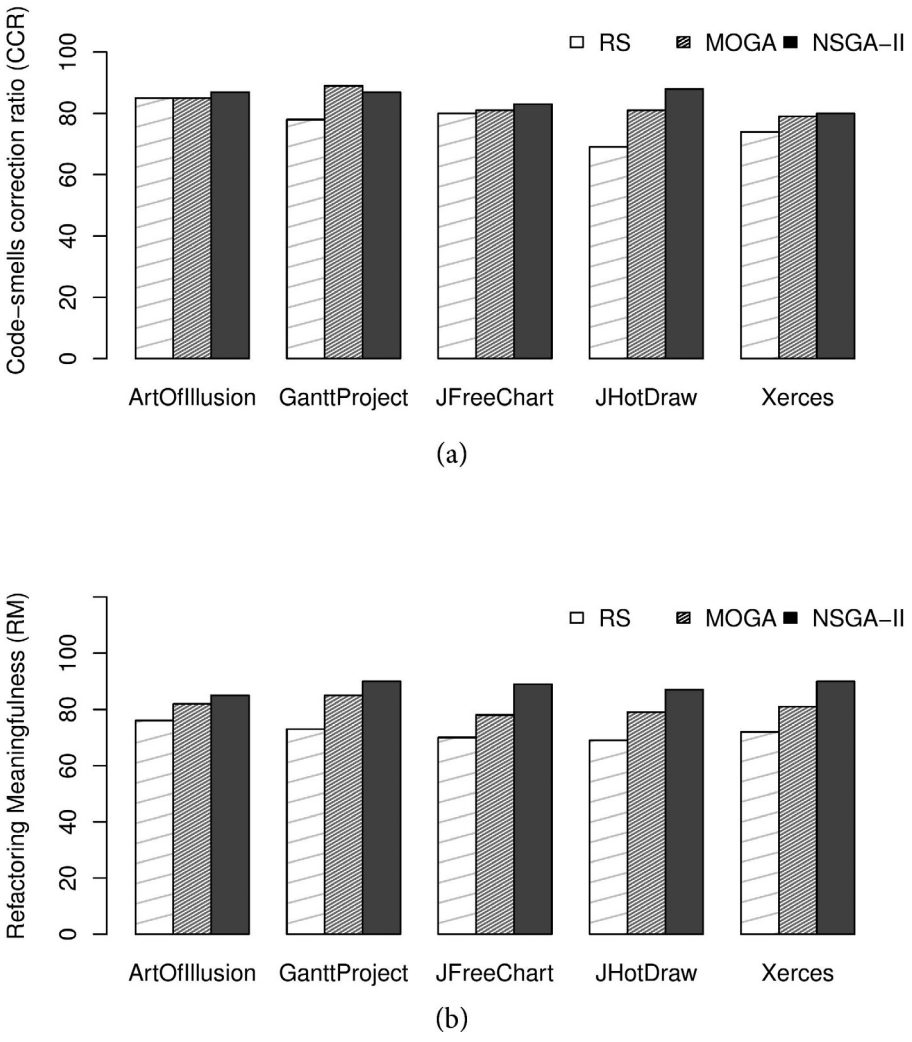


Fig. 8. (a) CCR and (b) RM median values of 31 independent runs of NSGA-II, MOGA, and GA.

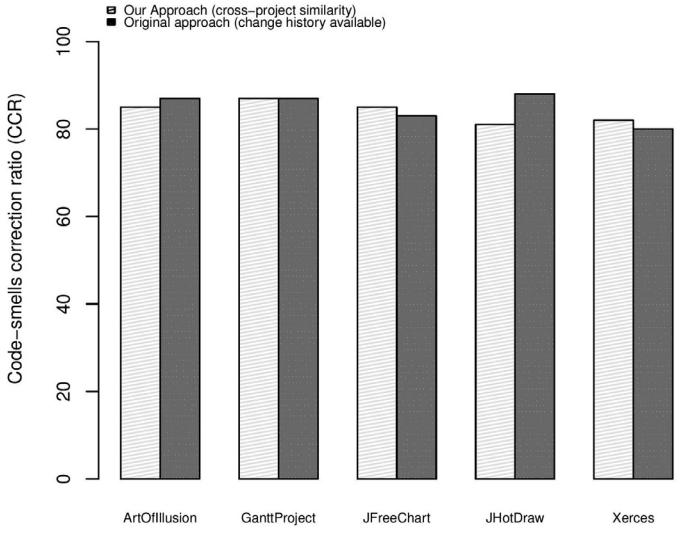


Fig. 9. CCR median values of 31 independent runs of our approach where the change history is not available and the original one where the change history is available.

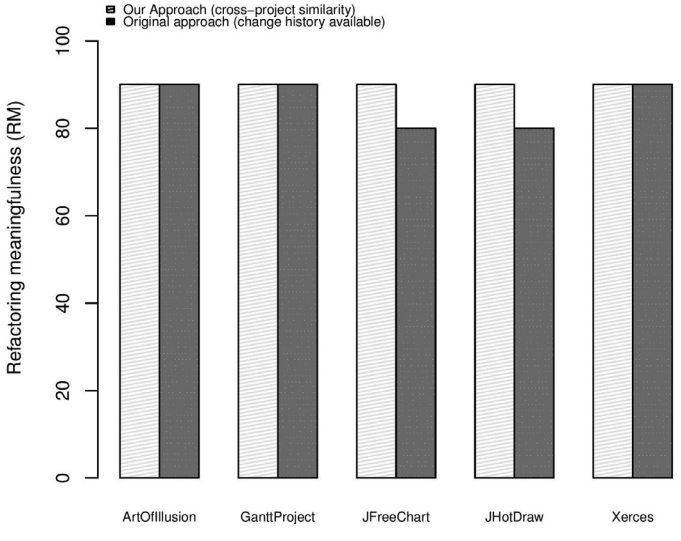


Fig. 10. RM values of our approach where the change history is not available comparing to the original approach where the change history is available.

GanttProject, and Xerces), while only in two systems (i.e., JHotDraw and JFreeChart) there was a significant differences found but small effect size.

In average, over 31 independent runs, our approach utilizing cross-project similarity achieved 86% of RM, while our original approach achieved 90% of RM for all the projects.

Table 7
CCR median values of 31 independent runs, and RM values.

Systems	Approach	CCR	RM
Xerces-J	Our approach	80% (70|87)	90%
	Ouni et al. (ICSM 2012)	78% (68|87)	80%
	Harman et al. (2007)	N.A.	60%
	Kessentini et al. (2011)	90% (78|87)	60%
JFreeChart	Our approach	83% (70|84)	80%
	Ouni et al. (ICSM 2012)	85% (71|84)	80%
	Harman et al. (2007)	N.A.	60%
	Our approach	89% (75|84)	50%
GanttProject	NSGA-II (our approach using context)	87% (39|45)	90%
	Ouni et al. (ICSM 2012)	89% (40|45)	80%
	Harman et al. (2007)	N.A.	60%
	Kessentini et al. (2011)	93% (42|45)	60%
Artoflllusion	Our approach	87% (59|68)	80%
	Ouni et al. (ICSM 2012)	87% (59|68)	70%
	Harman et al. (2007)	N.A.	50%
	Kessentini et al. (2011)	90% (61|68)	50%
JHotDraw	Our approach	88% (14|16)	90%
	Ouni et al. (ICSM 2012)	81% (13|16)	80%
	Harman et al. (2007)	N.A.	50%
	Kessentini et al. (2011)	81% (13|16)	50%
Average values for all systems	Our approach	85%	86%
	Ouni et al. (ICSM 2012)	84%	78%
	Harman et al. (2007)	N.A.	58%
	Kessentini et al. (2011)	88%	54%

Based on the results of Fig. 10, we noticed that RM medians are close, the results are statistically different but the effect size (Cohen, 1988) which quantifies the difference is small for most of the systems considered in our experiments.

This slight degradation might due to the nature of the studied projects coming from different sizes, different code-smell distributions, and different development teams. That is the same set of code-smell types that we are considering in our experiments (blob, spaghetti code, functional decomposition, and data classes) are detected in all the studied projects but with different distributions; consequently such regularities cannot be easily found. Furthermore, in general, from a software engineer stand point, there are no common practices on how refactoring is applied or decided; this depends mainly on the opinion/decision of the developer. For instance to fix a blob class, a programmer may refer to many possible alternatives refactorings such as move method, move field or extract class, etc. (Fowler et al., 1999). Hence, lower RM score is obtained for JHotDraw, this might be due to the fact that JHotDraw is a relatively mature and well-designed and implemented system after more than 10 years of active development, and which have much less code-smell instances comparing to the other projects.

To sum up, despite this slight degradation in terms of RM the obtained results with cross-project similarity are considered as good (86% for all five systems). We can conclude that, despite the non-availability of change history such collected data from other projects can be a valuable knowledge for recommending software refactoring.

Results for RQ4 (Comparison to state-of-the-art): We compare our approach with three existing approaches: Ouni et al. (2012), Kessentini et al. (2011), and Harman et al. (2007). Table 7 presents the obtained results. We observe that our approach achieved much better results comparing to the three other approaches in terms of RM. Overall, of all the five studied projects, our approach provides 86% RM, while Ouni et al., Kessentini et al., and Harman et al., provide only 78%, 58% and 54% respectively. For instance, for Xerces-J, 90% (9 out of 10) of the evaluated refactoring operations by our participants do not generate construct semantic incoherencies, while the other approaches achieved respectively 80%, 60% and 60% of RM.

Using historical software changes, our approach succeeded in proposing more meaningful refactorings and reducing the number of construct semantic incoherencies when applying refactoring. At the same time, after applying the proposed refactoring operations, we found that most of the detected code-smells are fixed with an average of 85% of CCR. The corrected code-smells were from different types (blob, spaghetti code, functional decomposition, and data classes (Ouni et al., 2013a)). The majority of non-fixed code-smells are related to the blob type. This type of code-smell usually requires a large number of refactoring operations and is then very difficult to correct. This obtained correction score is comparable to the one of other existing approaches that do not use development history (89%). Thus, the slight loss in the CCR is largely compensated by the significant improvement of the construct semantic coherence. In addition, this slight loss in terms of CCR can be justified by the nature of the problem under consideration; there is a kind of trade-off between CCR and RM: when RM increase, then CCR decrease. This provides evidence that the quality of the refactoring solutions (number of fixed code-smells) is inherently in conflict with construct semantic. For this reason, we used a multi-objective optimization approach instead of a single objective one.

In conclusion, our approach provides good refactoring suggestions from both perspectives: CCR and construct semantics coherence using the development change history.

To sum up, the development history collected from different software projects represents a valuable knowledge that can be very useful in improving automated software refactoring. This provides evidence that, in the context of refactoring, efficient and effective reuse is extremely important to the success of refactoring strategies. This aligns well with software reuse principles as a common practice for software developers to save time and efforts.

Results for RQ5 (Insights): To answer **RQ5**, we show in Fig. 11 the obtained quality 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. Understandability is the quality factor that has the highest gain value; whereas the effectiveness quality factor has the lowest one. This mainly due to many reasons (1) the majority of fixed code-smells (blob, spaghetti code) are known to increase the coupling (DCC) within classes which heavily affect the quality index calculation of the effectiveness factor; (2) the vast majority of suggested refactoring types were move method, move field, and extract class (Fig. 11) that are known to have a high impact on coupling (DCC), cohesion (CAM) and the design size in classes (DSC) that serves to calculate the understandability quality factor. Furthermore, we noticed that JHotDraw produced the lowest quality increase for the four quality factors. This is justified by the fact that JHotDraw is known to be of good design and implementation practices (Kessentini et al., 2010) and contains few number of code-smells comparing to the four other studied systems.

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

6. Discussions

We noticed that our approach outperforms the other approaches that do not use development history for fixing code-smells and construct semantic coherence. Moreover, we had promising refactoring results even when the development history is not available and only similar collected refactorings applied to similar contexts is used. We also found that our NSGA-II based approach performs much better than two other popular multi and mono-objective evolutionary

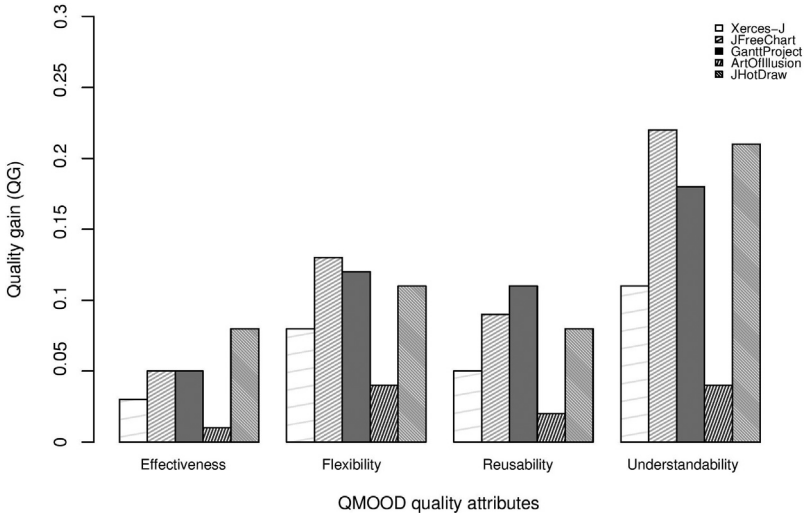


Fig. 11. The impact of best refactoring solutions on QMOOD quality attributes.

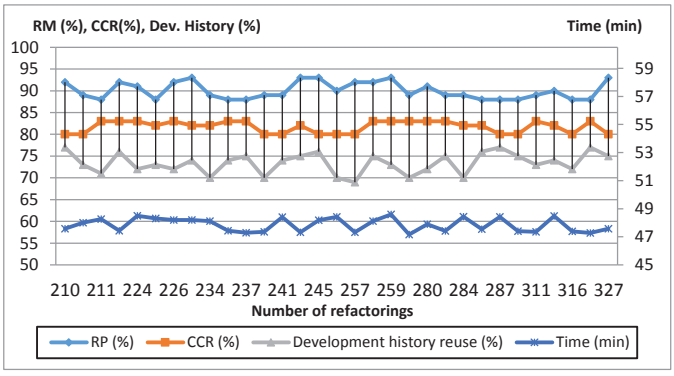


Fig. 12. Impact of the number of refactorings on multiple simulation runs on Xerces-J.

algorithms: MOGA, and GA in terms of Hypervolume, Spread, CCR, and RM. We also studied the results of multiple executions with the execution time to evaluate the performance and the stability of our approach. Over 31 independent simulation runs on Xerces-J, the average value of RM, CCR, development history reuse and execution time for finding the optimal refactoring solution with a number of iterations (stopping criteria) fixed to 8000 was respectively 90%, 82%, 73% and 47min85s as shown in Fig. 12. The standard deviation values were lower than 2, except for development history reuse (2.36). This indicates that our approach is reasonably scalable from the performance standpoint. Moreover, we evaluate the impact of the number of suggested refactorings on the RM, development history reuse, and CCR scores in 31 different executions. The best RMand CCR scores are also obtained with a higher number of suggested refactorings. Thus, we could conclude that our approach is scalable from the performance standpoint, especially that quality improvement is not related in general to real-time applications where time-constraints are very important. In addition, the results' accuracy is not affected by the number of suggested refactorings.

Another important consideration is the refactoring operations' distribution. We contrast that most of the suggested refactorings are related to move method, move field, and extract class for almost all the studied systems. For instance, in GanttProject, we had different distribution of different refactoring types as illustrated in Fig. 13. 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 specific code-smells types detected in GanttProject. Most of the code-smells are related to the blob, data classes, and spaghetti code that need particular refactorings. For instance, to fix a blob code-smell, the idea is to move elements from the blob class to other classes (e.g., data classes) in order to reduce the number of functionalities from the blob and add behavior to other classes (mainly data classes) or to improve some quality metrics such as coupling and cohesion (related to spaghetti code). As such, refactorings like move field, move method, and extract class are likely to be more useful to fix the blob, and data class code-smells.

7. Threats to validity

Some threats need to be considered when interpreting our study results.

The first threat concerns the conclusion validity. We used the Wilcoxon rank sum test with a 95% confidence level to test if significant differences existed between the measurements for different treatments. This test makes no assumption that the data is normally distributed and is suitable for ordinal data, so we can be confident that the statistical relationships we observed are significant. For the internal validity, the used a technique for detecting code-smells may lead to a few number of false positives; this may have an impact on the results of our experiments.

External validity refers to the generalizability of our findings. In this study, we performed our experiments on several different widely used open-source systems belonging to different domains and with different sizes. However, we cannot assert that our results can be generalized to industrial Java applications, other programming languages, and to other practitioners. Another threat can be the limited number of subjects and studied systems, which externally threatens the generalizability of our results. Future replications of this study are necessary to confirm our findings.

Construct validity is concerned with the relationship between theory and what is observed. The manual evaluation of suggested refactorings depends on the expertise of the subjects involved in our experiments. Another threat concerns the data about the collected refactorings of the studied systems where we use Ref-Finder, which is known to be efficient. 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 and filter the set of refactorings found by Ref-Finder.

8. Conclusions and future work

In this paper, we presented a novel search-based approach to improve the automation of refactoring. We used software-development history, combined with structural and semantic information, to

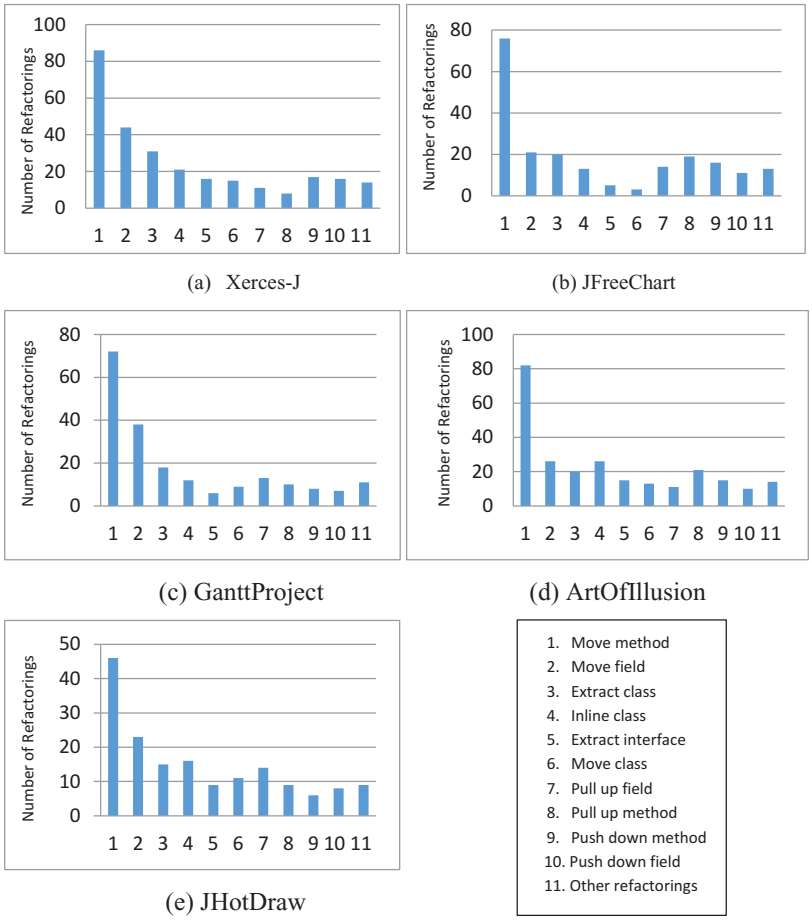


Fig. 13. Suggested refactorings distribution.

improve the precision and efficiency of new refactoring suggestions. We start by generating some solutions that represent a combination of refactoring operations to apply. A fitness function combines three objectives: maximizing design quality, construct semantic preservation and the re-use of the history of changes applied to similar contexts.

To find the best trade-off between these objectives, NSGA-II algorithm is used. We found promising refactoring results, on a set of 5 open source systems, even when the development history is not available, and only similarity collected refactorings applied to similar contexts is used. Moreover, to evaluate the feasibility and efficiency of our approach, we applied three state-of-the-art multi and mono-objective metaheuristics (MOGA, Random Search, and GA) for the refactoring problem using the same adaptation (solution coding, genetic operators, fitness functions). We found that our NSGA-II based approach has been the one computing the best results regarding to two performance indicators (Hypervolume and Spread) over a benchmark composed of five open-source software systems. Moreover, our approach outperforms significantly three state-of-the-art refactoring approaches (Kessentini et al., 2011; Harman and Tratt, 2007; Ouni et al., 2012) in terms of code-smells correction ratio, and construct semantic coherence.

As part of our future work, we will extend our validation to include additional systems and code-smell types to fix and compare our NSGA-II approach with other existing refactoring approaches (O'Keeffe and Cinnéide, 2006; Seng et al., 2006). An interesting future direction to our work would be the investigation of a new strategy where both good and bad changes recorded in the past will be considered. The aim is to maximize the dissimilarity with bad changes applied in the past and maximize the similarity with good ones. Furthermore, another research direction worth to explore is to investigate the effect of code-smells correction on the number of faults on the system. Indeed, recent empirical studies (Hall et al., 2014; D'Ambros et al., 2010) have found that code-smells do correlate with software defects, and an increase in the number of code-smells is likely to generate bugs (D'Ambros et al., 2010). Other studies have found that some code-smells do indicate fault-prone code in some circumstances but the effect that these smells have on faults is small (Hall et al., 2014). While based on some empirical studies fixing code-smells is likely to reduce implicitly faults, it will be interesting to investigate, as future work, whether fixing code-smells through refactoring can reduce the number of faults.

Acknowledgements

This work is partially supported by Japan Society for the Promotion of Science, Grant-in-Aid for Scientific Research (S) (No. 25220003), and also by Osaka University Program for Promoting International Joint Research.

References

Arcuri, A., Briand, L.C., 2011. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In: Proceedings of the 33rd International Conference on Software Engineering. ACM, New York, NY, USA, pp. 1–10.
Bansiya, J., Davis, C.G., 2002. A hierarchical model for object-oriented design quality assessment. IEEE Trans. Softw. Eng. 28 (1), 4–17.
Beyer, D., Noack, A., 2005. Clustering software artifacts based on frequent common changes. In: Proceedings of the 13th International Workshop on Program Comprehension. St. Louis, MO, USA, pp. 259–268.
Briand, L.C., Daly, J.W., Wust, J.K., 1999. A unified framework for coupling measurement in object-oriented systems. IEEE Trans. Softw. Eng. 25 (1), 91–121.

Brown, W.J., Malveau, R.C., Brown, W.H., III, H.W.M., Mowbray, T.J., 1998. Anti Patterns: Refactoring Software, Architectures, and Projects in Crisis, 1st edition John Wiley and Sons.
Cederqvist, P., Dec. 2003. Version Management with CVS, www.cvshome.org/docs/manual/.
Chidamber, S.R., Kemerer, C.F., 1994. A metrics suite for object-oriented design. IEEE Trans. Softw. Eng. 20 (6), 293–318.
Choinzon, M., Ueda, Y., 2006. Detecting defects in object oriented designs using design metrics. In: Proceedings of the 7th conference on Knowledge-Based Software Engineering. Amsterdam, The Netherlands, pp. 61–72.
Coad, P., Yourdon, E., 1991. Object-Oriented Design. Prentice Hall.
Cohen, J., 1988. Statistical Power Analysis for the Behavioral Sciences Evanston. Routledge, IL, USA.
D'Ambros, M., Bacchelli, A., Lanza, M., 2010. On the impact of design flaws on software defects. In: International Conference on Quality Software.
Deb, K., 2009. Multi-Objective Optimization Using Evolutionary Algorithms. Wiley.
Deb, K., Pratap, A., Agarwal, S., Meyarivan, T., 2002. A fast and elitist multiobjective genetic algorithm: NSGA-II, IEEE Trans. Evol. Comput. 6, 182–197.
Deligiannis, I., Shepperd, M., Roumeliotis, M., Stamelos, I., 2003. An empirical investigation of an object-oriented design heuristic for maintainability. J. Syst. Softw. 65, 127–139.
Deligiannis, I., Stamelos, I., Angelis, L., Roumeliotis, M., Shepperd, M., 2004. A controlled experiment investigation of an object-oriented design heuristic for maintainability. J. Syst. Softw. 72, 129–143.
Du Bois, B., Demeyer, S., Verelst, J., 2004. Refactoring—improving coupling and cohesion of existing code. In: Proceedings of the 11th Working Conference on Reverse Engineering. Tampere, Finland, pp. 144–151.
Ekman, T., Asklund, U., 2004. Refactoring-aware versioning in eclipse. Electron. Notes Theor. Comput. Sci. 207, 57–69.
Fatiregun, D., Harman, M., Hierons, R., 2004. Evolving transformation sequences using genetic algorithms. In: 4th IEEE International Workshop on Source Code Analysis and Manipulation. Los Alamitos, California, USA, pp. 65–74.
Fonseca, C.M., Fleming, P.J., 1993. Genetic algorithms for multiobjective optimization: formulation, discussion and generalization. In: Proceedings of the 5th International Conference on Genetic Algorithms. San Mateo, CA, USA, pp. 416–423.
Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D., 1999. Refactoring – Improving the Design of Existing Code, 1st ed. Addison-Wesley.
Gall, H., Hajek, K., Jazayeri, M., 1998. Detection of logical coupling based on product release history. In: Proceedings of the International Conference on Software Maintenance. Los Alamitos, CA, pp. 190–198.
Gîrba, T., Ducasse, S., Kuhn, A., Marinescu, R., Daniel, R., 2007. Using concept analysis to detect co-change patterns. In: Proceedings of International Workshop on Principles of Software Evolution. Dubrovnik, Croatia, pp. 83–89.
Hall, T., Zhang, M., Bowes, D., Sun, Y., 2014. Some code smells have a significant but small effect on faults. ACM Trans. Softw. Eng. Methodol. 23 (4(33)).
Harman, M., Tratt, L., 2007. Pareto optimal search based refactoring at the design level. In: Proceedings of the Genetic and Evolutionary Computation Conference. London, UK, pp. 1106–1113.
Hassan, A., Holt, R., 2004. Predicting change propagation in software systems. In: Proceedings 20th International Conference on Software Maintenance. Chicago, Illinois, USA, pp. 284–293.
Infusion hydrogen, (2012): Design flaw detection tool. Available at: http://www.intooitus.com/products/infusion.
iPlasma (0000): http://loose.upt.ro/iplasma/index.html.
Jensen, A., Cheng, B., 2010. On the use of genetic programming for automated refactoring and the introduction of design patterns. In: Proceedings of the 12th annual conference on Genetic and evolutionary computation. Portland, Oregon, pp. 1341–1348.
Joshi, P., Joshi, R.K., 2009. Concept analysis for class cohesion. In: Proceedings of the 13th European Conference on Software Maintenance and Reengineering. Kaiserslautern, Germany, pp. 237–240.
Kagdi, H.H., Collard, M.L., Maletic, J.I., 2007. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. J. Softw. Mainten. 19 (2), 77–131.
Kataoka, Y., Ernst, M.D., Griswold, W.G., Notkin, D., 2001. Automated support for program refactoring using invariants. In: International Conference on Software Maintenance. Florence, Italy, pp. 736–743.
Kessentini, M., Kessentini, W., Sahraoui, H., Boukadoum, M., Ouni, A., 2011. Design defects detection and correction by example. In: Proc. of the 19th IEEE Int. Conf. on Program Comprehension. Kingston, ON, Canada, pp. 81–90.
Kessentini, M., Vaucher, S., Sahraoui, H., 2010. Deviance from perfection is a better criterion than closeness to evil when identifying risky code. In: Proceedings of the International Conference on Automated Software Engineering. Lawrence, Kansas, USA, pp. 113–122.
Khomh, F., Penta, M.D., Gueheneuc, Y.-G., 2009. An exploratory study of the impact of code smells on software change-proneness. In: Proceedings of the 16th IEEE Working Conference on Reverse Engineering. Kaiserslautern, Germany, pp. 75–84.
Kilic, H., Koc, E., Cereci, I., 2011. Search-based parallel refactoring using population-based direct approaches. In: Proceedings of the 3rd international Symposium on Search Based Software Engineering. Szeged, Hungary, pp. 271–272.
Kim, M., Notkin, Grossman, D., Wilson, D., Jr, G., 2013. Identifying and summarizing systematic code changes via rule inference. IEEE Trans. Softw. Eng. 39 (1), 45–62.
Li, W., Shatnawi, R., 2007. An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution. J. Syst. Softw. 80, 1120–1128.
Mäntylä, M.V., Vanhanen, J., Lassenius, C., 2003. A taxonomy and an initial empirical study of bad smells in code. In: IEEE International Conference on Software Maintenance. Amsterdam, The Netherlands, pp. 381–384.
Marinescu, R., 2004. Detection strategies: metrics-based rules for detecting design flaws. In: Proceedings of the 20th International Conference on Software Maintenance. IEEE Computer Society Press, Chicago, Illinois, USA, pp. 350–359.
Mens, T., Demeyer, S. (Eds.), 2008. Software Evolution. Springer.
Mens, T., Tourwé, T., 2004. A survey of software refactoring. IEEE Trans. Softw. Eng. 30 (2), 126–139.
Moha, N., Guéhéneuc, Y.-G., Duchien, L., Meur, A.-F.L., 2010. DECOR: A method for the specification and detection of code and design smells. IEEE Trans. Softw. Eng. 36 (1), 20–36.
Moha, N., Hacene, A., Valtchev, P., Guéhéneuc, Y-G., 2008. Refactorings of design defects using relational concept analysis. In: Proceedings of the 4th International Conference on Formal Concept Analysis. Montréal, QC, Canada, pp. 289–304.
Monden, A., Nakae, D., Kamiya, T., Sato, S., Matsumoto, K., 2002. Software quality analysis by code clones in industrial legacy software. In: IEEE Symposium on Software Metrics. Ottawa, Canada, pp. 87–94.
Ó Cinnéide, M., Tratt, L., Harman, M., Counsell, S., Moghadam, I.H., 2012. Experimental assessment of software metrics using automated refactoring. In: Proceedings of the ACM-IEEE International Symposium on Empirical Software Engineering and Management, pp. 49–58.
O'Keeffe, M., Cinnéide, M.O., 2006. Search-based refactoring for software maintenance. J. Syst. Softw. 81 (4), 502–516.
Olbrich, S., Cruzes, D., Basili, V.R., Zazworka, N., 2009. The evolution and impact of code smells: A case study of two open source systems. In: 3rd International Symposium on Empirical Software Engineering and Measurement. Lake Buena Vista, Florida, USA, pp. 390–400.
Olbrich, S.M., Cruzes, D.S., Sjoberg, D.I.K., 2010. Are all code smells harmful? A study of god classes and brain classes in the evolution of three open source systems. In: International Conference Software Maintenance. Timişoara, Romania, pp. 1–10.
Opdyke, W.F., 1992. Refactoring: A Program Restructuring Aid in Designing Object-Oriented Application Frameworks (Ph.D. thesis). University of Illinois at Urbana-Champaign.
Otero, F.E.B., Johnson, C.G., Freitas, A.A., Thompson, S.J., 2010. Refactoring in automatically generated programs. In: International Symposium on Search Based Software Engineering. Benevento, Italy.
Ouni, A., Kessentini, M., Sahraoui, H., 2013b. Search-based refactoring using recorded code changes. In: Proceedings of the 17th European Conference on Software Maintenance and Reengineering. Genova, Italy, pp. 221–230.
Ouni, A., Kessentini, M., Sahraoui, H., Boukadoum, M., 2013a. Maintainability defects detection and correction: a multi-objective approach. J. Autom. Softw. Eng. 20 (1), 47–79 Springer.
Ouni, A., Kessentini, M., Sahraoui, H., Hamdi, M.S., 2012. Search-based refactoring: towards semantics preservation. In: Proceedings of the 28th IEEE International Conference on Software Maintenance (ICSM). Riva Del Garda, Italy, pp. 347–356.
Ouni, A., Kessentini, M., Sahraoui, H., Hamdi, M.S., 2013c. The use of development history in software refactoring using a multi-objective evolutionary algorithm. In: The Genetic and Evolutionary Computation Conference. Amsterdam, The Netherlands, pp. 1461–1468.
Poshyvanyk, D., Marcus, A., 2006. The conceptual coupling metrics for object-oriented systems. In: In the 22nd IEEE International Conference on Software Maintenance. Philadelphia, Pennsylvania, USA, pp. 469–478.
Prete, K., Rachatasumrit, Sudan, N., Kim, M., 2010. Template-based reconstruction of complex refactorings. In: Proceedings of the International Conference on Software Maintenance. Timişoara, Romania, pp. 1–10.
Qayum, F., Heckel, R., 2009. Local search-based refactoring as graph transformation. In: Proceedings of 1st International Symposium on Search Based Software Engineering. Cumberland Lodge, Windsor, UK, pp. 43–46.
Ratiu, D., Ducasse, S., Gîrba, T., Marinescu, R., 2004. Using history information to improve design flaws detection. In: Proceedings of the Conference on Software Maintenance and Reengineering. Tampere, Finland, pp. 223–232.
Ratzinger, J., Sigmund, T., Vorburger, P., Gall, H., 2007. Mining software evolution to predict refactoring. In: Proceedings of the 1st International Symposium on Empirical Software Engineering and Measurement. Madrid, Spain, pp. 354–363.
Riel, A.J., 1996. Object-Oriented Design Heuristics, 1st ed. Addison-Wesley, Boston, MA, USA.
Sahraoui, H., Godin, R., Miceli, T., 2000. Can metrics help to bridge the gap between the improvement of OO design quality and its automation? In: International Conference on Software Maintenance, 2000. San Jose, California, USA, pp. 154–162.
Seng, O., Stammel, J., Burkhart, D., 2006. Search-based determination of refactorings for improving the class structure of object-oriented systems. In: Proceedings of the Genetic and Evolutionary Computation Conference. Seattle, WA, USA, pp. 1909–1916.
Sjoberg, D.I., Yamashita, A., Anda, B.C.D., Mockus, A., Dyba, T., 2013. Quantifying the effect of code smells on maintenance effort. IEEE Trans. Softw. Eng. 39 (8), 1144–1156.
Soetens, Q.D., Perez, J., Demeyer, S., 2013. An initial investigation into change-based reconstruction of floss-refactorings. In: Proceedings of the 29th IEEE International Conference on Software Maintenance. Eindhoven, The Netherlands, pp. 384–387.
Tahvildari, L., Kontogiannis, K., 2003. A metric-based approach to enhance design quality through meta-pattern transformation. In: Proceedings of the 7st European Conference on Software Maintenance and Reengineering. Benevento, Italy, pp. 183–192.

Tsantalis, N., Chaikalis, T., Chatzigeorgiou, A., 2008. JDeodorant: identification and removal of type-checking bad smells. In: 12th European Conference on Software Maintenance and Reengineering. Athens, Greece, pp. 329–331.
Van Emden, E., Moonen, L., 2002. Java quality assurance by detecting code smells. In: Proceedings of the 9th Working Conference on Reverse Engineering. Budapest, Hungary, pp. 97–106.
Yamashita, A.F., Moonen, L., 2012. Do code smells reflect important maintainability aspects? In: Proceedings of the 28th IEEE International Conference on Software Maintenance. Riva Del Garda, Italy, pp. 306–315.
Yamashita, A.F., Moonen, L., 2013a. To what extent can maintenance problems be predicted by code smell detection? – An empirical study. Inf. Softw. Technol. 55 (12), 2223–2242.
Yamashita, A.F., Moonen, L., 2013b. Do developers care about code smells? An exploratory survey. In: the Proceedings of the 20th Working Conference on Reverse Engineering. Genova, Italy, pp. 242–251.
Ying, T.T., Murphy, G.C., Ng, R., Chu-Carroll, M.C., 2004. Predicting source code changes by mining change history. IEEE Trans. Softw. Eng. 30 (9), 574–586.
Zimmermann, T., Zeller, A., Weissgerber, P., Diehl, S., 2005. Mining version histories to guide software changes. IEEE Trans. Softw. Eng. 31 (6), 429–445.
Zitzler, E., Thiele, L., 1998. Multiobjective optimization using evolutionary algorithms – A comparative case study. In: Parallel Problem Solving from Nature. Springer, Germany, pp. 292–301.
Zitzler, E., Thiele, L., Laumanns, M., Fonseca, C.M., Fonseca, da V.G., 2003. Performance assessment of multiobjective optimizers: an analysis and review. IEEE Trans. Evol. Comput. 7, 117–132.

Ali Ouni is a research assistant professor at Osaka University (Japan). He holds a Ph.D. in computer science from the University of Montreal (Canada) in 2014. He is a member of the Software Engineering Laboratory, in the graduate school of information science and technology at Osaka University. He received his Master Degree Diploma (MSc.) and his bachelor degree (BSc.) in computer science from the Higher Institute of Applied Sciences and Technology (ISSAT), University of Sousse, Tunisia, respectively in 2010, and 2008. His research interests include the application of artificial intelligence techniques to software engineering (search-based software engineering). Since 2011, he published several papers in well-ranked journals and conferences. He serves as program committee member and reviewer for several conferences such as GECCO 2015, NASBASE 2015, CMSEBA 2014, GECCO 2014.

Marouane Kessentini is a tenure-track assistant professor at University of Michigan. He is the founder of the research group: Search-based Software Engineering@Michigan. He holds a Ph.D. in Computer Science, University of Montreal (Canada), 2011. His research interests include the application of artificial intelligence techniques to software engineering (search-based software engineering), software testing, model-driven engineering, software quality, and re-engineering. He has published around 50 papers in conferences, workshops, books, and journals including three best paper awards. He has served as program-committee/organization member in several conferences and journals.

Houari A. Sahraoui is full professor at the department of computer science and operations research (GEODES, software engineering group) of University of Montreal. Before joining the university, he held the position of lead researcher of the software engineering group at CRIM (Research center on computer science, Montreal). He holds an Engineering Diploma from the National Institute of computer science (1990), Algiers, and a Ph.D. in Computer Science, Pierre & Marie Curie University LIP6, Paris, 1995. His research interests include the application of artificial intelligence techniques to software engineering, object-oriented metrics and quality, software visualization, and re-engineering. He has published around 100 papers in conferences, workshops, books, and journals, edited three books, and gives regularly invited talks. He has served as program committee member in several major conferences (IEEE ASE, ECOOP, METRICS, etc.), as member of the editorial boards of two journals, and as organization member of many conferences and workshops (ICSM, ASE, QAOOSE, etc). He was the general chair of IEEE Automated Software Engineering Conference in 2003.

Katsuro Inoue received the BE, ME, and DE degrees in information and computer sciences from Osaka University, Japan, in 1979, 1981, and 1984, respectively. He was an assistant professor at the University of Hawaii at Manoa from 1984–1986. He was a research associate at Osaka University from 1984–1989, an assistant professor from 1989–1995, and a professor beginning in 1995. His interests are in various topics of software engineering such as program analysis, repository mining, software development environment, and software process modeling. He is a member of the IEEE, the IEEE Computer Society, and the ACM.

Mohamed Salah Hamdi is currently an Associate Professor at Ahmed Bin Mohammed Military College (Qatar). He earned a "Diplom-Informatiker" Degree in Computer Science from the Technical University of Munich (Germany) in 1993 and a Ph.D. Degree in Computer Science from the University of Hamburg (Germany) in 1999. His research interests are focused on intelligent autonomous agents, e-learning, information customization, effects of ICT on the society, software engineering, neural networks, machine learning, and on artificial intelligence in general. He has published a large number of papers in conferences, books, and journals. He has also served as program-committee member in several conferences and journals.