Software Qual J
DOI 10.1007/s11219-014-9233-7

Prioritizing code-smells correction tasks using chemical reaction optimization

Ali Ouni · Marouane Kessentini · Slim Bechikh · Houari Sahraoui

Abstract The presence of code-smells increases significantly the cost of maintenance of systems and makes them difficult to change and evolve. To remove code-smells, refactoring operations are used to improve the design of a system by changing its internal structure without altering the external behavior. In large-scale systems, the number of code-smells to fix can be very large and not all of them can be fixed automatically. Thus, the prioritization of the list of code-smells is required based on different criteria such as the risk and importance of classes. However, most of the existing refactoring approaches treat the code-smells to fix with the same importance. In this paper, we propose an approach based on a chemical reaction optimization metaheuristic search to find the suitable refactoring solutions (i.e., sequence of refactoring operations) that maximize the number of fixed riskiest code-smells according to the maintainer's preferences/criteria. We evaluate our approach on five medium- and large-sized open-source systems and seven types of code-smells. Our experimental results show the effectiveness of our approach compared to other existing approaches and three different others metaheuristic searches.

Keywords Search-based software engineering · Refactoring, software quality · Code-smells · Chemical reaction optimization

A. Ouni (✉) · H. Sahraoui
DIRO, GEODES Lab, University of Montreal, Montreal, QC, Canada
e-mail: ouniali@iro.umontreal.ca

H. Sahraoui
e-mail: sahraouh@iro.umontreal.ca

A. Ouni · M. Kessentini · S. Bechikh
CIS, SBSE-Michigan Lab, University of Michigan, Michigan, MI, USA
e-mail: marouane@umich.edu

S. Bechikh
e-mail: slimb@umich.edu

Published online: 29 April 2014

1 Introduction

The maintenance and evolution of large-scale systems requires 90 % of the total software costs (Erlikh 2000). Adding new functionalities, correcting bugs, and modifying the code to improve its quality are major parts of those costs. These continuous changes applied on software systems increase their complexity and take them away from their original architecture and design. This may in turn introduce poor design effects known as *code-smells* (Brown et al. 1998; Fenton and Pfleeger 1997; Fowler et al. 1999). Code-smells are known to have a negative impact on quality attributes such as flexibility or maintainability (Brown et al. 1998). They are often introduced unintentionally by software engineers during the initial design or during software development and maintenance due to bad design choices/decisions.

One of the widely used techniques to fix code-smells is *refactoring* that improves design structure while preserving the overall functionalities and behavior (Fowler et al. 1999). In general, refactoring is performed through two main steps: (1) detection of code fragments that need to be improved (e.g., code-smells) and (2) identification of refactoring solutions to achieve this goal. The first step is well covered in the literature, and there exists a growing number of techniques to identify code-smells (Fowler et al. 1999; Marinescu 2004; Moha et al. 2010; Kessentini et al. 2010, 2011; Ouni et al. 2012a). Once detected, not all code-smells have equal effects and importance (Brown et al. 1998; Olbrich et al. 2009, 2010). In general, developers need to start by fixing the higher risk code-smells. However, in the literature, the majority of existing contributions (Fowler et al. 1999; Marinescu 2004; Alikacem 2006; Du Bois et al. 2004; Harman and Tratt 2007; Moha et al. 2008) proposes manual or semiautomated refactoring solutions that can be applied to fix particular types of code-smells (e.g., blobs and spaghetti code) (Brown et al. 1998) or to improve some quality metrics (e.g., cohesion and coupling) (Seng et al. 2006; Du Bois et al. 2004; Harman and Tratt 2007) without taking into consideration the importance/risk of the code-smells to fix. Based on our previous work on code-smells correction and refactoring (Ouni et al. 2012a, b, 2013), we found that most of the important and riskiest code fragments are not improved and most of the riskiest code-smells, notably the blob code-smell (Riel 1996), are very difficult to fix using such a manual or an automated approach. For instance, in Ouni et al. (2012a, 2013), we found that most of the non-corrected code-smells are related to the blob that requires typically a large number of refactorings. This type of code-smells is detected in general on important classes in the system that change frequently during the development/maintenance process, which make this kind of code-smell more severe than other code-smells.

In this paper, we introduce a novel approach to support automated refactoring suggestion for correcting code-smells where riskiest code-smells are prioritized during the correction process. Hence, we formulated the refactoring suggestion problem as a combinatorial optimization problem to find the near-optimal sequence of refactorings from a huge number of possible refactorings (http://www.refactoring.com/catalog/) (Harman 2007; Harman et al. 2012). To this end, we used a novel metaheuristic search by means of chemical reaction optimization (CRO) (Lam and Li 2010) to find the suitable refactoring solutions (i.e., sequence of refactoring operations) that maximize the number of corrected code-smells while prioritizing the most important and riskiest code fragments. Proposed by Lam and Li (2010), CRO loosely mimics what happens to molecules in a chemical reaction system and tries to capture the energy in the reaction process. The CRO is a population-based intelligent algorithm that outperforms classic metaheuristic algorithms such as genetic algorithm (GA) (Goldberg 1989), simulated annealing (SA) (Kirkpatrick et al. 1983), and particle swarm optimization (Kennedy and Eberhart 1995) in solving many optimization problems (Lam and Li 2010; Yu et al. 2012; Xu et al. 2011; Sun et al. 2012; Lam et al. 2012a).

More specifically, the primary contributions of the paper can be summarized as follows: (1) The paper introduces a novel formulation of the refactoring suggestion problem using CRO, and to the best of our knowledge, this is the first attempt in SBSE (Harman 2007; Harman et al. 2012) to use CRO to solve software engineering problems. (2) The paper reports the results of an empirical study on five different medium- and large-sized projects (www.ganttproject.biz, http://www.jfree.org/jfreechart/, http://www.jhotdraw.org/, www.artofillusion.org/, and http://xerces.apache.org/xerces-j) compared to two other approaches that do not use prioritization while fixing code-smells. (3) The paper reports statistical comparisons between the CRO approach with three popular metaheuristics, GA (Goldberg 1989), SA (Kirkpatrick et al. 1983), and particle swarm optimization (PSO) (Kennedy and Eberhart 1995), which have been shown to have good performance in solving many software engineering problems (Harman 2007; Harman et al. 2012). Our results indicate that the CRO approach has great promise. The statistical analysis of the obtained results provides evidence to support the claim that CRO is more efficient and effective than three other popular metaheuristics. Furthermore, we compare our approach to two other techniques that do not prioritize the correction of code-smells. Over 31 runs for each approach, our CRO-based approach significantly outperforms the two other refactoring approaches in terms of number of corrected code-smells as well as the number of important, severest, riskiest code-smells that can be fixed.

The rest of this paper is organized as follows: Sect. 2 describes the relevant background and related work in which the current paper is located. Sect. 3 describes the used metaheuristic algorithm and its adaptation. Experimental results and evaluation of the approach are reported in Sect. 4. Section 5 is dedicated to the discussion. Finally, concluding remarks and directions for future work are provided in Sect. 6.

2 Background

2.1 Code-smells

Code-smells, also called anti-patterns (Brown et al. 1998), anomalies (Fowler et al. 1999), design flaws (Marinescu 2004), or bad smells (Munro 2005), are problems resulting from bad design practices and refer to design situations that adversely affect the software maintenance. According to Fowler (Fowler et al. 1999), bad 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. In (Fowler et al. 1999), Beck defines 22 sets of symptoms of code-smells. These include large classes, feature envy, long parameter lists, and lazy classes. Each code-smell type is accompanied by refactoring suggestions to remove it. 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 following seven 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 the data. It is a large class that declares many fields and methods with a low cohesion and almost has no parents and no children.

- **Data class**: It is a class that contains only data and performs no processing on these data. It is typically composed of highly cohesive fields and accessors.
- **Spaghetti code**: It is a code with a complex and tangled control structure. This code-smell is the characteristic of procedural thinking in object-oriented programming. Spaghetti code is revealed by classes with no structure, declaring long methods with no parameters, and utilizing global variables. Names of classes and methods may suggest procedural programming. Spaghetti code does not exploit and prevents the use of object-orientation mechanisms, polymorphism, and inheritance.
- **Functional decomposition**: It occurs when a class is designed with the intent of performing a single function. This is found in code produced by non-experienced object-oriented developers.
- **Schizophrenic class**: It occurs when a public interface of a class is large and used non-cohesively by client methods, i.e., disjoint groups of client classes use disjoint fragments of the class interface in an exclusive fashion.
- **Shotgun surgery**: It occurs when a method has a large number of external operations calling it, and these operations are spread over a significant number of different classes. As a result, the impact of a change in this method will be large and widespread.
- **Feature envy**: It is found when a method heavily uses attributes and data from one or more external classes, directly or via accessor operations. Furthermore, in accessing external data, the method is intensively using data from at least one external capsule.

We choose these code-smell types in our experiments because they are the most frequent ones to detect and fix based on the recent studies (Moha et al. 2010; Ouni et al. 2012a, b, 2013; Kessentini et al. 2011). Moreover, various approaches (e.g., kessentini et al. 2011; Ouni et al. 2012a) and tools, e.g., JDeodorant (http://loose.upt.ro/iplasma/index.html) (Tsantalis et al. 2008), inFusion (http://www.intooitus.com/products/infusion), Decor (Moha et al. 2010), and iPasma (http://loose.upt.ro/iplasma/index.html) supporting code-smells detection, have been proposed recently in the literature. The vast majority of these tools provide different environments and methods to detect code-smells. For instance, Kessentini et al. (2010) have proposed a detection tool based on the risk evaluation score. The tool uses *well-designed code examples* to find code-smells based on the notion that the more code fragments to evaluate are different from these examples, the more they are considered risky (Kessentini 2010). Other tools such as inFusion (http://www.intooitus.com/products/infusion) allow detecting and classifying code-smells according to a severity score based on some *design properties* (such as size and complexity, encapsulation, coupling, cohesion, and hierarchy). On the other hand, Ouni et al. (2012a) allows detecting code-smells using metric-based detection rules independently to their severity, risk, or importance. Detection rules are expressed in terms of metrics and threshold values. Each rule detects a specific defect type (e.g., blob, spaghetti code, and functional decomposition) and is expressed as a logical combination of a set of quality metrics/threshold values. These detection rules are generated/learned from real instances of code-smells using GA. We show along of this paper how different heuristics and detection strategies can be used to support the code-smells correction step according to a prioritization schema.

2.2 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 (Fowler et al. 1999) defines refactoring as a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior. Refactoring was first introduced by Opdyke (1992), who provided a catalog 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, and reusability. (Fowler et al. 1999; Mens and Tourwé 2004). For these precious benefits on design quality, some modern integrated development environments (IDEs), such as Eclipse (http://www.eclipse.org/), NetBeans (https://netbeans.org/), and Refactoring Browser (http://www.refactory.com/refactoring-browser), provide semi-automatic support for applying the most commonly used refactorings, e.g., move method and rename class. 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).

2.3 Search-based software engineering

Our approach is largely inspired by contributions in search-based software engineering (SBSE). The term SBSE was first used by Harman and Jones (2001) and defined as the application of search-based approaches to solving optimization problems in software engineering (Harman and Jones 2001) (Freitas and Souza 2011). SBSE seeks to reformulate software engineering problems as search problems by defining them in terms of solution representation, fitness function, and solution change operators. Once a software engineering task is framed as a search problem, there are many search algorithms that can be applied to solve that problem. In the last decade, many SBSE contributions have been proposed for various software engineering problems including software testing (McMinn 2004), requirements engineering (Zhang et al. 2008), bug fixing (Le Goues et al. 2012), project management (Alba and Chicano 2005), refactoring (Harman and Tratt 2007), and service-oriented software engineering (Canfora et al. 2005). The most studied and known models are based on classic evolutionary algorithms (EAs) such as SA (Kirkpatrick et al. 1983), GA (Goldberg 1989), PSO (Kennedy and Eberhart 1995), and tabu search (TS) (Glover and Laguna 1997). CRO (Lam and Li 2010) is a recently proposed EA that mimics the interactions of molecules in chemical reactions. The CRO has been applied for solving many real-world optimization problems (Lam and Li 2010; Lam et al. 2012b; Yu et al. 2012), such as the network optimization problem, the grid scheduling problems, quadratic assignment problem (QAP), and resource-constrained project scheduling problem (RCPSP). Experimental results have shown that the CRO is efficient for solving the optimization problems with multi-optima. Moreover, CRO has a nice ability to jump out of the local optima, which makes it more suitable for solving discrete combinatorial optimization problems.

The scope of this paper is to illustrate one of the first attempts of using CRO to solve software engineering problems (SBSE) and particularly the code-smells correction one. Indeed, to the best of our knowledge and based on the recent surveys in SBSE (Harman et al. 2012), CRO is not yet explored in software engineering, and the idea of treating code-smells correction as an optimization problem to be solved by a SBSE approach was not studied before our proposal in (Kessentini et al. 2011). In this paper, we propose a new search-based approach using CRO with new formulation of refactoring task to prioritize the correction of code-smells.

3 Software refactoring using chemical reaction optimization

In this section describes the principles that underlie the proposed method for fixing code-smells while prioritizing riskiest, severest, and important code-smells during the correction process. Therefore, we first present an overview of CRO algorithm and, subsequently, provide the details of our approach and our CRO adaptation for code-smells correction problem.

3.1 Chemical reaction optimization

Chemical reaction optimization (CRO) is a new recently proposed metaheuristic (Lam and Li 2010) inspired from chemical reaction. It is not difficult to discover the correspondence between optimization and chemical reaction. Both of them aim to seek the global minimum (but with respect to different objectives) and the process evolves in a stepwise fashion. With this discovery, CRO was developed for solving optimization problems by mimicking what happens to molecules in chemical reactions. It is a multidisciplinary design that loosely couples computation with chemistry (see Table 1). The manipulated agents are molecules, and each has a profile containing some properties. A molecule is composed of several atoms and characterized by the atom type, bond length, angle, and torsion. One molecule is distinct from another when they contain different atoms and/or different number of atoms. The term “molecular structure” is used to summarize all these characteristics and it corresponds to a solution in the metaheuristic meaning. The representation of a molecular structure depends on the problem we are solving, provided that it can express a feasible solution of the problem. A molecule possesses two kinds of energies, i.e., potential energy (PE) and kinetic energy (KE). The former quantifies the molecular structure in terms of energy, and it is modeled as the objective function value when evaluating the corresponding solution. A change in molecular structure (chemical reaction) is tantamount to switching to another feasible solution. CRO evolves a population of molecules by means of four chemical reactions called: (1) on-wall ineffective collision, (2) decomposition, (3) inter-molecular ineffective collision, and (4) synthesis. Consequently, similarly to GA, the molecule corresponds to the population individual and chemical reactions correspond to the variation operators. However, CRO is distinguished by the fact that environmental selection is performed by the variation operator. Differently to GA which generates an offspring population then makes a competition between the latter and the parent population, in CRO once an offspring is generated, it competes for survival with

Table 1 CRO analogy between chemical and metaheuristic meanings

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

The first column contains the properties of a molecule used in CRO. The second column shows the corresponding meanings in the metaheuristic

CRO pseudocode

```
Input: Parameter values
Output: Best solution found and its objective function value
/*Initialization*/
Set $PopSize$, $KELossRate$, $MoleColl$, $buffer$, $InitialKE$, $\alpha$, and $\beta$
Create $PopSize$ molecules
/*Iterations*/
While the stopping criteria not met do
  Generate $b \in [0, 1]$
  If ($b > MoleColl$) Then
    Randomly select one molecule $M_\omega$
    If (Decomposition criterion met) Then
      Trigger Decomposition
    Else
      Trigger OnwallIneffectiveCollision
    End If
  Else
    Randomly select two molecules $M_{\omega 1}$ and $M_{\omega 2}$
    If (Synthesis criterion met) Then
      Trigger Synthesis
    Else
      Trigger IntermolecularIneffectiveCollision
    End If
  End If
  Check for any new minimum solution
End While
```

Fig. 1 Basic CRO pseudocode

its parent(s) within the realization of the corresponding chemical reaction. Figure 1 illustrates the pseudocode of the CRO. According to this figure, the CRO algorithm begins by initializing the different parameters that are as follows:

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

For more details about the role of each of these parameters, the interested reader is invited to confer to (Lam et al. 2012b; Yu et al. 2012). In a more recent study, the effects of each of these parameters on the global behavior of CRO in addition to their interaction between themselves have been studied. The interested reader could refer to (Lam et al. 2013) for mathematical details behind the convergence analysis of CRO.

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

(1) *On-wall ineffective collision*: This reaction corresponds to the situation when a molecule collides with a wall of the container and then bounces away remaining in one single unit. In this collision, we only perturb the existing molecule structure (which captures the structure of the solution) ω to ω'. This could be done by any neighborhood operator $N(\cdot)$.
(2) *Decomposition*: It corresponds to the situation when a molecule hits a wall and then breaks into several parts (for simplicity, we consider two parts in this work). Any mechanism that can produce ω_1' and ω_2' from ω is allowed. The goal of decomposition is to allow the algorithm to explore other regions of the search space after enough local search by the ineffective collisions.
(3) *Inter-molecular ineffective collision*: This reaction takes place when multiple molecules collide with each other and then bounce away. The molecules (assume two) remain unchanged before and after the process. This elementary reaction is very similar to the unimolecular ineffective counterpart since we generate ω_1' and ω_2' from ω_1 and ω_2 such that $\omega_1' = N(\omega_1)$ and $\omega_2' = N(\omega_2)$. The goal of this reaction is to explore several neighborhoods simultaneously each corresponding to a molecule.
(4) *Synthesis*: This reaction is the opposite of decomposition. A synthesis happens when multiple (assume two) molecules hit against each other and fuse together. We obtain ω' from the fusion of ω_1 and ω_2. Any mechanism allowing the combination of solutions is allowed, where the resultant molecule is in a region farther away from the existing ones in the solution space. The idea behind synthesis is diversification of solutions.

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

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

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

For more details about the role of each of these attributes in CRO, the reader is invited to refer to Lam and Li (2010).

The CRO has been recently applied successfully to different combinatorial and continuous optimization problems (Xu et al. 2011; Sun et al. 2012; Lam et al. 2012a). Several nice properties for the CRO have been detected. These properties are as follows:

- The CRO framework allows deploying different operators to suit different problems.
- Its variable population size allows the system to adapt to the problems automatically, thereby minimizing the number of required function evaluations.
- Energy conversion and energy transfer in different entities and in different forms make CRO unique among metaheuristics. CRO has the potential to tackle those problems that have not been successfully solved by other metaheuristics.
- Other attributes can easily be incorporated into the molecule. This gives flexibility to design different operators.
- CRO enjoys the advantages of both SA and GA.
- CRO can be easily programmed in object-oriented programming language, where a class defines a molecule and methods define the elementary reactions.

Based on all these observations, the CRO seems to be an interesting metaheuristic ready to use for tackling SE problems. This work represents the first attempt to exploit CRO within the SBSE community. Our choice to use CRO was mainly due to the fact that the problem is so complex that standard metaheuristics may struggle. On the other hand, CRO is a newly established metaheuristics for optimization, and recent results demonstrate that it has superior performance when compared with other existing optimization algorithms in solving different optimization problems in different area (Lam and Li 2010; Lam et al. 2012b). In addition, CRO is distinguished from other existing metaheuristics that were already used in SBSE by the following three points that we have already used in our paper. Firstly, CRO inherits several features from GA and SA. For example, the variation operations performed by chemical reaction operators could be seen as new versions of crossover/mutation operations. Moreover, the use of KE allows CRO to accept worse solutions over better ones with a controlled probability in the same way as does the Metropolis rule in SA. CRO inherits the features of well balancing between intensification and diversification of the search from hybrid EAs (Jourdan et al. 2009). In fact, EAs were hybridized due to their lack of ability in intensifying the search toward promising regions of the search space. CRO does not have such shortcoming thanks to their four elementary chemical reactions that ensure both intensification (by means of on-wall ineffective collision and inter-molecular ineffective collision) and diversification (by means of decomposition and synthesis). Furthermore, CRO is the sole metaheuristic that performs the environmental selection within the variation operators. In fact, when the KE does not exceed a particular threshold, the chemical reaction (variation) is aborted. This fact represents a way to avoid driving the population toward non-interesting regions. When KE does not satisfy the threshold, this means that the defined search direction by the new generated offspring individual(s) is not promising. Such characteristic increases the efficiency of CRO since we do not waste time/effort in exploring non-interesting regions. For more details, please confer to (Lam et al. 2013).

3.2 Approach overview

In this paper, we propose an approach to support automated code-smells correction according to a prioritization schema where important, riskiest, and severe code-smells are prioritized taking into consideration the preferences of developers. In practice, a suitable prioritization scheme can significantly improve and maximize the efficiency of allocating maintenance efforts. Our approach aims at finding, from an exhaustive list of possible refactorings (http://www.refactoring.com/catalog/), the suitable refactoring solutions that should fix as much as possible the number of detected code-smells according to a prioritization schema.

Thus, to find the suitable refactoring solution (i.e., sequence of refactoring operations), a huge search space should be explored. In fact, the search space is determined not only by the number of possible refactorings (http://www.refactoring.com/catalog/), their possible combinations, and the order in which they should be applied, but also by the software size (number of packages, classes, methods, fields, etc.). To this end, we see the refactoring suggestion problem as a search-based optimization problem to best explore this huge search space, in order to find the suitable refactoring solutions by the means of CRO (Lam and Li 2010).

The general structure of our approach is sketched in Fig. 2. It takes as inputs a source code of the program to be refactored, a list of possible refactorings that can be applied (http://www.refactoring.com/catalog/), a set of bad smells detection rules (Ouni et al. 2012a), risk and severity score for each detected code-smell, software maintainer prioritization/preferences, and a history of code changes applied to the system during it lifecycle. Our approach generates as output the optimal sequence of refactorings, selected from an exhaustive list of possible refactorings that should improve software quality by minimizing as much as possible the number of detected code-smells.

3.3 Problem formulation

In this section, we describe our formulation of the refactoring task taking into consideration a set of criteria and also software engineers' preferences. Our goal is to ensure that riskiest code-smells are fixed first. More concretely, let us consider the example of a software system that contains 5 code-smells: 1 blob, 2 data classes (DC), and 2 functional decompositions (FD). Many possible refactoring solutions can fix these code-smells with the same score. For example, we can have 2 different solutions S_1 and S_2. After applying S_1, both DC and FD are fixed (the correction score $\mathrm{CCR}(s_1) = 4/5 = 0.8$), and after

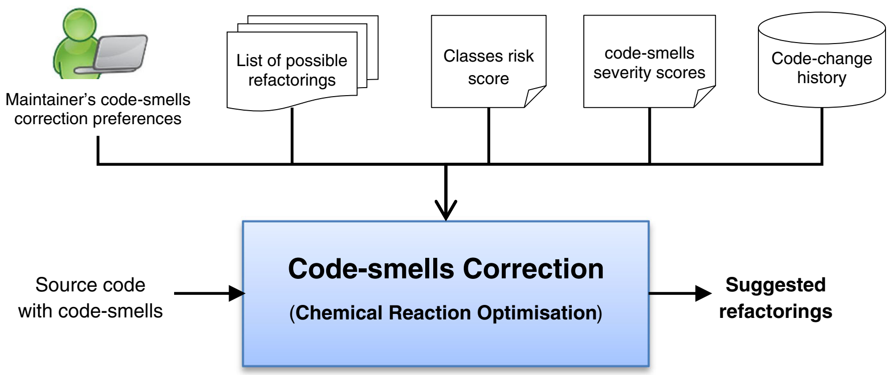

Maintainer's code-smells correction preferences
List of possible refactorings
Classes risk score
code-smells severity scores
Code-change history
Source code with code-smells
Code-smells Correction
(Chemical Reaction Optimisation)
Suggested refactorings

Fig. 2 Approach overview

applying S_2, the blob, 1 DC and 2 FD are fixed (the correction score $\mathrm{CCR}(s_2) = 4/5 = 0.8$). The same correction score is obtained by both s_1 and s_2. However, the blob class is known to be the severest and it may significantly affect the design of the whole system since it tends to centralize the functionalities of the system into one class. From this perspective, we consider that the solution s_2 is better than s_1. Prioritizing the correction of important code-smells is the main idea behind this paper. Furthermore, sometimes changing extensively a software system by applying refactorings may perturb the initial design. To preserve the initial design, software maintainers my ignore some of the suggested refactorings. To this end, it is very interesting to start first by fixing important code-smells. Hence, results can be of interest to software engineers, who perform development or maintenance activities and need to take into account and forecast their effort. In the following, we describe a set of four prioritization measures (priority, severity, risk, and importance) we use in our formulation of the refactoring task to correct code-smells.

(1) *Priority:* developers typically give more importance to some code-smell types that can occur with different impacts on the system's quality. Developers can rank different types of detected code-smell according to their preferences. Using such prioritization scheme, designers can save their time and maximize the efficiency of allocating maintenance effort in their software project. For instance, based on our experience in the field of software refactoring and code-smells correction (Ouni et al. 2012a, b, 2013; Kessentini et al. 2010, 2011), we choose to give a priority score of 7 for the blob code-smell, 6 for functional decomposition, 5 for shotgun surgery, 4 for spaghetti code, 3 for feature envy, 2 for schizophrenic class, and 1 for data classes, so that fixing blob code-smells instances will be more prioritized.

(2) *Severity:* In practice, not all code-smells have equal effects/importance (http://www.intooitus.com/products/infusion, Olbrich et al. 2010). Each individual instance has its severity score that allows designers to immediately spot and fix the most critical instances of each code-smell. Concretely, the same code-smell type can occur in different code fragments but with different impact scores on the system design (http://www.intooitus.com/products/infusion). This impact score represents the relative severity of the code-smell, as well as the absolute negative impact on overall quality. For example, two code-smell instances having, respectively, 27 and 36 methods can be detected both as blob, but each of them has different impact scores on the system quality (number of methods, coupling, cohesion, etc.). In this paper, we use the inFusion tool (http://www.intooitus.com/products/infusion), which classifies code-smells based on a set of "design properties" such as size and complexity, encapsulation, coupling, cohesion, and hierarchy (Chidamber and Kemerer 1994). Moreover, these design properties are the most useful ones in existing code-smells detection approaches (Marinescu 2004; Moha et al. 2010; Kessentini et al. 2011).

(3) *Risk:* An important score to consider is the risk score. Thus, we consider that the more code deviates from good practices, the more it is likely to be risky (Kessentini et al. 2010). Consequently, the riskiest code-smells should be prioritized during the correction step. A risk score is associated with each detected code-smell that corresponds to the deviance from well-designed code (Kessentini et al. 2010).

(4) *Importance*: Generally, developers need to know which code fragments (e.g., classes and packages) are important in the whole software system in order to focus their effort on improving their quality. In a typical software system, important code fragments are those who change frequently during the development/maintenance process to add new functionalities, to accommodate new changes or to improve its

structure. Moreover, as reported in the literature (Olbrich et al. 2009; Khomh et al. 2009a, b), classes participating in design problems (e.g., code-smells) are significantly more likely to be subject to changes and to be involved in fault-fixing changes (bugs) (Khomh et al. 2009a, b). Indeed, if a class undergoes many changes in the past and it is still smelly, it needs to be fixed as soon as possible. On the other hand, not every code-smell is assumed to have negative effects on the maintenance/evolution of a system. It has been shown that in some cases, a large class might be the best solution (Olbrich et al. 2010). Moreover, it is reported that if a code-smell (e.g., God Class) is created intentionally and remains unmodified or hardly undergo changes, the system may not experience any problems (Olbrich et al. 2010; Ratiu et al. 2004). For these reasons, code-smells related to more frequently changed classes should be prioritized during the correction process.

3.4 CRO design

In order to adapt a metaheuristic search technique to a specific problem, a number of elements have to be defined and different decisions have to be made. To apply CRO, the following elements have to be defined: the way in which solutions (molecules) should be encoded so that they can be manipulated by the search process, creation of a population of solutions (a container of molecules), evaluation function to determine a quantitative measure of the ability of candidate solutions to solve the problem under consideration, selection of solutions for elementary reaction operations, creation/modification of new solutions using elementary reaction operations (on-wall ineffective collision, decomposition, synthesis, and intermolecular ineffective collision) to explore the search space.

In the following, we describe the design of these elements for the code-smells correction problem using CRO.

3.4.1 Solution coding

(a) Solution representation

In our CRO design, we use a vector-based solution coding. Each vector's dimension represents a refactoring operation. When created, the order of applying these refactorings corresponds to their positions in the vector. In addition, for each refactoring, a set of controlling parameters, e.g., actors and roles, as illustrated in Table 2, are randomly picked from the program to be refactored. An example of a solution is given in Fig. 3. Hence, we construct a refactoring solution incrementally. First, we create an empty vector that represents the current refactoring solution. Then, we randomly select (1) a refactoring operation from the list of possible refactorings and (2) its controlling parameters (i.e., the code elements), after that, (3) we apply this refactoring operation to an intermediate model that represents the original source code. The model will be updated after applying each refactoring operation and the process will be repeated n times until reaching the maximal solution length (n). This means that in each iteration i, we have a different model according to the $(i - 1)$ applied refactoring operations, that is, in each iteration, the controlling parameters will be selected from the current version of the model. For this reason, the order of applying the refactoring sequence heavily influences the refactoring results.

Table 2 Refactorings and its controlling parameters

Ref.	Refactorings	Controlling parameters
MM	Move method	(source class, target class, method)
MF	Move field	(source class, target class, field)
PUF	Pull-up field	(source class, target class, field)
PUM	Pull-up method	(source class, target class, method)
PDF	Pushdown field	(source class, target class, field)
PDM	Pushdown method	(source class, target class, method)
IC	Inline class	(source class, target class)
EC	Extract class	(source class, new class)
EI	Extract interface	(Source class, interface)
ESuC	Extract super class	(Source class, super class)
ESC	Extract subclass	(Source class, subclass)

RO1	move field (Person, Employee, salary)
RO2	extract class(Person, Adress, streetNo, city, zipCode, getAdress(), updateAdress())
RO3	move method (Person, Employee, getSalary())
RO4	push down field (Person, Student, studentId)
RO5	inline class (Car, Vehicle)
RO6	move method (Person, Employee, setSalary())
RO7	move field (Person, Employee, tax)
RO8	extract class (Student, Course, courseName, CourseCode, addCourse(), rejectCourse()

Fig. 3 CRO solution coding

(b) Refactoring feasibility

Although we propose a recommendation system and we do not apply refactorings automatically, it is important to guarantee that they are feasible and that they can be legally applied. The first work in the literature was by Opdyke (1992) who introduced a way of formalizing the preconditions that must be imposed before a refactoring can be applied in order to preserve the behavior of the system. Opdyke created functions, which could be used in predicate expressions to formalize constraints. These are analogous to the analysis functions used later by Ó Cinneide (2000) and Roberts (1999) who developed an automatic refactoring tool to reduce program analysis. In our approach, we used a system to check a set of simple conditions, inspired from the one developed by Ó Cinnéide (2000). Similarly to Ó Cinnéide (2000), our search-based refactoring tool simulates refactorings using pre- and post-conditions that are expressed in terms of conditions on the code model. To express these conditions, we defined a set of functions. These functions include the following:

- ***isClass(c)***: c is a class.
- ***isInterface(c)***: c is an interface.
- ***isMethod(m)***: m is a method.
- ***Sig(m):*** the signature of the method m.

- ***isField(f)***: f is a field.
- ***defines(c,e)***: the code element e (method or field) is implemented in the class/interface c.
- ***exists(e)***: the code element c exists in the current version of the code model.
- ***inheritanceHierarchy(c1, c2)***: both classes c1 and c2 belong to the same inheritance hierarchy
- ***isSuperClassOf(c1,c2)***: c1 is a superclass of c2.
- ***isSubClassOf(c1,c2)***: c1 is a subclass of c2.
- ***fields(c)***: returns the list of fields defined in the class or interface c.
- ***methods(c)***: returns the list of methods implemented in class or interface c.

For each refactoring operation, we specify a set of pre- and post-conditions to ensure the feasibility of applying them using a static analysis. For example, to apply the refactoring operation *move method(Person, Employee, getSalary())*, a number of necessary preconditions should be satisfied, e.g., *Person* and *Employee* should exist and should be classes; *getSalary()* should exist and should be a method; the classes *Person* and *Employee* should not be in the same inheritance hierarchy; the method *getSalary()* should be implemented in *Person*; the method signature of *getSalary()* should not be present in class *Employee*. As post-conditions, *Person*, *Employee*, and *getSalary()* should exist; *getSalary()* declaration should be in the class *Employee*; and *getSalary()* declaration should not exist in the class *Person*.

Table 3 describes for each refactoring operation its pre- and post-conditions that should be satisfied.

For composite refactorings, such as extract class and inline class, the overall pre- and post-conditions should be checked. For a sequence of refactorings, which may be of any length, we simplify the computation of its full precondition by analyzing the precondition of each refactoring in the sequence and the corresponding effects on the code model (post-conditions). For more details about the pre- and post-conditions, the interested reader is invited to confer to (Opdyke 1992; Mel 2000; Roberts 1999).

(c) Creation of the initial population of solutions

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

3.4.2 Elementary reaction operators

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

(a) On-wall Ineffective Collision

For on-wall ineffective collision, many possible changes can be applied to a given refactoring solution. To apply this change operator, *n* (one or more) refactorings are first

Table 3 Specification of pre- and post-conditions for refactoring operations

Refactorings	Pre- and post-conditions
Move method(c1,c2,m)	**Pre**: exists(c1) AND exists(c2) AND exits(m) AND isClass(c1) AND isClass(c2) AND isMethod(m) AND NOT(inheritanceHierarchy(c1,c2)) AND defines(c1,m) AND NOT(defines(c2, sig(m)) **Post**: exists(c1) AND exists(c2) AND exits(m) AND defines(c2,m) AND NOT(defines(c1, m))
Move field(c1,c2,f)	**Pre**: exists(c1) AND exists(c2) AND exits(f) AND isClass(c1) AND isClass(c2) AND isField(f) AND NOT(inheritanceHierarchy(c1,c2)) AND defines(c1,f) AND NOT(defines(c2, f)) **Post**: exists(c1) AND exists(c2) AND exits(f) AND defines(c2,f) AND NOT(defines(c1, f)
Pull-up field(c1,c2,f)	**Pre**: exists(c1) AND exists(c2) AND exits(f) AND isClass(c1) AND isClass(c2) AND isField(f) AND isSuperClassOf(c2,c1) AND defines(c1,f) AND NOT(defines(c2, f)) **Post**: exists(c1) AND exists(c2) AND exits(f) AND defines(c2,f) AND NOT(defines(c1, f))
Pull-up method(c1,c2,m)	**Pre**: exists(c1) AND exists(c2) AND exits(m) AND isClass(c1) AND isClass(c2) AND isMethod(f) AND isSuperClassOf(c2,c1) AND defines(c1,m) AND NOT(defines(c2, sig(m)) **Post**: exists(c1) AND exists(c2) AND exits(m) AND defines(c2,m) AND NOT(defines(c1, m))
Pushdown field(c1,c2,f)	**Pre**: exists(c1) AND exists(c2) AND exits(m) AND isClass(c1) AND isClass(c2) AND isField(f) AND isSubClassOf(c2,c1) AND defines(c1,f) AND NOT(defines(c2, f)) **Post**: exists(c1) AND exists(c2) AND exits(m) AND defines(c2,m) AND NOT(defines(c1, m))
Pushdown method(c1,c2,m)	**Pre**: exists(c1) AND exists(c2) AND exits(m) AND isClass(c1) AND isClass(c2) AND isMethod(f) AND isSubClassOf(c2,c1) AND defines(c1,m) AND NOT(defines(c2, sig(m)) **Post**: exists(c1) AND exists(c2) AND exits(m) AND defines(c2,m) AND NOT(defines(c1, m))
Inline class(c1,c2)	**Pre**: exists(c1) AND exists(c2) AND isClass(c1) AND isClass(c2) **Post**: exists(c1) AND NOT(exists(c2))
Extract class(c1,c2)	**Pre**: exists(c1) AND NOT(exists(c2)) AND isClass(c1) AND $\lvert$methods(c1)$\rvert \geq 2$ **Post**: exists(c1) AND exists(c2) AND isClass(c2)
Extract interface(c1,c2)	**Pre**: exists(c1) AND NOT(exists(c2)) AND isInterface(c1) AND $\lvert$methods(c1)$\rvert \geq 2$ **Post**: exists(c1) AND exists(c2) AND isInterface(c2)
Extract super class(c1,c2)	**Pre**: exists(c1) AND NOT(exists(c2)) AND isClass(c1) AND $\lvert$methods(c1)$\rvert \geq 2$ **Post**: exists(c1) AND exists(c2) AND isClass (c2) AND isSuperClass(c1,c2)
Extract subclass(c1,c2)	**Pre**: exists(c1) AND NOT(exists(c2)) AND isClass(c1) AND $\lvert$methods(c1)$\rvert \geq 2$ **Post:** exists(c1) AND exists(c2) AND isClass (c2) AND isSubClass(c1,c2)

picked at random from the vector representing the refactoring solution (sequence of refactorings). Then, for each of the selected refactorings, we apply one of the following possible changes using a "probabilistic select" (Lam and Li 2010):

- ***Refactoring type change (RTC)*** 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.

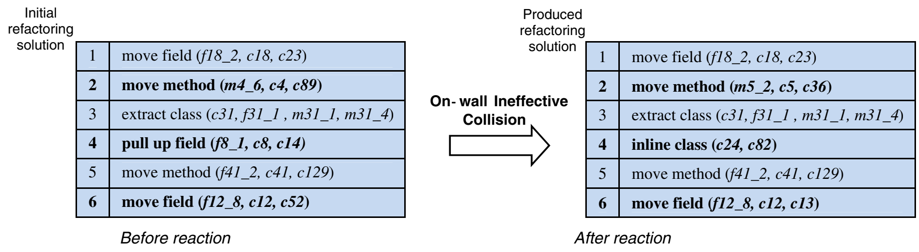
- ***Controlling parameters change (CPC)*** consists of replacing randomly, for the selected refactoring, only their controlling parameters. For instance, for a "move method," we can replace the source and/or target classes by other classes from the whole system.

An example is shown in Fig. 4. 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).

In our previous work (Ouni et al. 2012a, 2013; Kessentini et al. 2011), only refactoring type change is considered to create new refactoring solution from previous population. We will see in this paper how the diversification of change operators can significantly improve the refactoring results.

(b) Decomposition

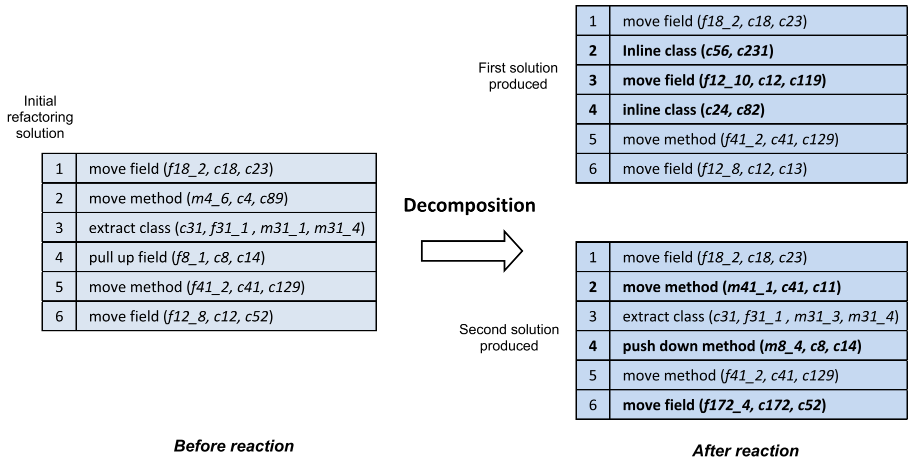
This operator is used to produce two new solutions far away from a given one. We apply "half-total-change" (Lam and Li 2010) to our implementation. We first duplicate the original solutions. Then, we add perturbations to *n/*2 dimensions of the original solution to create new solutions, where *n* is the size of vector representing the original solution. Each perturbation change could be performed through a *refactoring type change* but also *controlling parameters change*. An example is depicted in Fig. 5.


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

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

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

Fig. 4 Example of on-wall ineffective collision operator

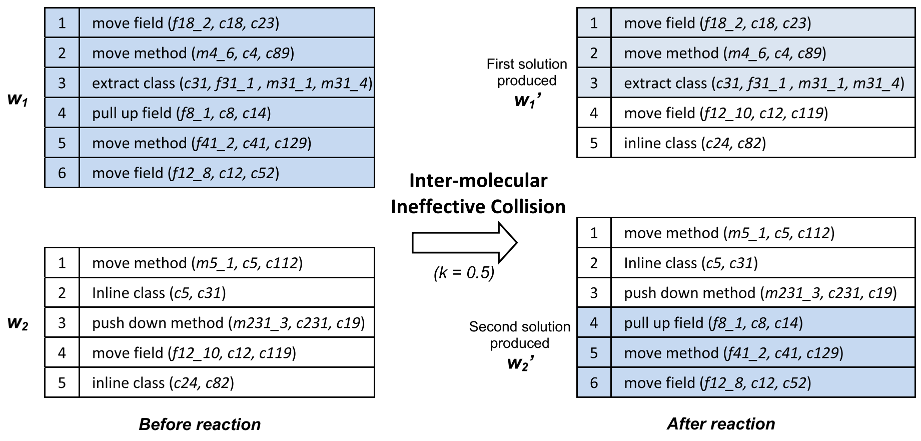


	Initial refactoring solution
1	move field (*f18_2, c18, c23*)
2	move method (*m4_6, c4, c89*)
3	extract class (*c31, f31_1 , m31_1, m31_4*)
4	pull up field (*f8_1, c8, c14*)
5	move method (*f41_2, c41, c129*)
6	move field (*f12_8, c12, c52*)

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

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

Fig. 5 Example of decomposition operator



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

Fig. 6 Example of inter-molecular ineffective collision operator

(c) Inter-molecular Ineffective Collision

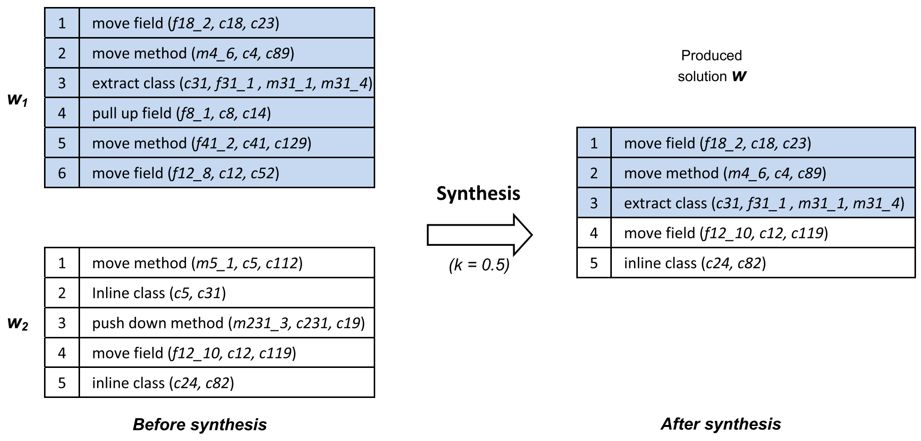
Inter-molecular ineffective collision is the process of two or more solutions to share information with each other and then produce two or more other different solutions. In our implementation, we apply inter-molecular ineffective collision between only two solutions (w_1 and w_2). To this end, two possible change mechanisms could be applied: (1) apply for each of solution on-wall ineffective collision, (2) exchange some dimensions between them using an operator similar to a single, random, cut-point crossover, in GA (Goldberg 1989). First, a random value k is chosen from [0,1]. Then, inter-molecular ineffective collision creates two new solutions by putting, for the first new solution, the first $k * n_1$ elements from the first parent (with length n_1), followed by the last $(1 - k) * n_2$ elements from the second parent (with length n_2). On the other hand, the second new solution contains the first $k * n_2$ elements from the second parent followed by last $(1 - k) * n_1$ element of the first parent. This operator ensures that the generated solutions will never have greater size than the biggest of the parents (Fraser and Arcuri 2013). As illustrated in Fig. 6, each child combines some of the refactoring operations of the first parent with some ones of the second parent.

(d) Synthesis

This operator is used to combine two refactoring solutions w_1 and w_2 into a new one w. In our approach, we are using two different mechanisms for synthesis operator: (1) crosscut combination and (2) probabilistic select (Lam and Li 2010). To apply synthesis operator, CRO selects randomly one of these two mechanisms.

For the first, an integer value k is randomly generated in the range of $[1, n]$, where n is the shortest vector length of the solutions w_1 and w_2 ($n = \text{Min}(|w_1|, |w_2|)$). Then, w is generated by picking the first k values from w_1 and the rest of the $(n - k)$ values from w_2. This operator must ensure that the length limits are respected. If so, some refactoring operations should be eliminated randomly. As shown in Fig. 7, a new refactoring solution w is formed by combining the first two set of refactorings from w_1 and the last set of refactorings from w_2.

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



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

Fig. 7 Example of synthesis operator

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

3.4.3 Fitness function

After creating a solution, it should be evaluated using an objective function to ensure its ability to solve the problem under consideration. We used a fitness function that calculates, according to prioritization schema described in the Sect. 3.3, the number of corrected code-smells using detection rules (Ouni et al. 2012a). Equation (1) allows calculating the quality of refactoring solution w.

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

where x_i is assigned to 0 if the actual class is detected as a code-smell using our code-smells detection rules (Ouni et al. 2012a), 1 otherwise, and $\alpha + \beta + \gamma + \delta = 1$ and their values express the confidence (i.e., weight) in each measure that can be fixed according to the preferences of developers. We have performed comprehensive experiments with different combinations of weights on each prioritization measure. For our experiments reported in this paper, we give equal weights (=0.25) to each of them.

3.4.4 Implementation details

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

The initial population/solution of CRO, GA, PSO, and SA are completely random. The stopping criterion is when the maximum number of function evaluations, set to 8000, is reached. After several trial runs of the simulation, the parameter values of the four

Table 4 Parameter settings used for the different algorithms

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

algorithms are fixed. There are no general rules to determine these parameters, and thus, we set the combination of parameter values by trial and error. Parameter settings of the four algorithms are shown in Table 4. For each algorithm, we repeat the simulation 31 times in each case and compute the median value.

Another issue is that our formulation of code-smells correction problem using prioritization schema is a maximization problem. However, CRO is originally designed to solve minimization problems and the objective function value should not be negative since it is interpreted as energy (Lam and Li 2010). Typically, to convert a maximization problem f to a minimization one, $-f$ is considered as objective function; however, this may not be appropriate for CRO (Lam et al. 2012b). Thus, to keep with CRO principles, we can consider $f' = 1 - f$, to make every possible f' nonnegative. After minimizing f', we can compute the corresponding f by $f = 1 - f'$. As such, CRO can be adapted to solve maximization problems.

The simulation codes are programmed in Java using Soot (Vallée-Rai et al. 2000), a java optimization framework, that provide a panel of functionalities helping on analyzing java programs, calculating metrics, simulating refactorings, etc.

4 Evaluation

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

4.1 Research questions and objectives

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

RQ1: (*Usefulness*) To what extent can the proposed approach correct code-smells?
RQ2: (*Precision*) To what extent can the proposed approach correct severest, riskiest, and important code-smells?
RQ3: (*Comparison to state of the art*) To what extent can the proposed approach improves the results of refactoring suggestion using prioritization compared to existing work that do not use it (Kessentini et al. 2011)
RQ4: (*Comparison with other metaheuristics*) How does the proposed approach using CRO perform compared to other popular search-based algorithms GA (Goldberg 1989), SA (Kirkpatrick et al. 1983), PSO (Kennedy and Eberhart 1995)?

4.2 Setting

4.2.1 Systems studied

We applied our approach to five large- and medium-sized well-known open-source java projects: Xerces-J (http://xerces.apache.org/xerces-j), JFreeChart (http://www.jfree.org/jfreechart/), GanttProject (www.ganttproject.biz), ArtOfIllusion (www.artofillusion.org/), and JHotDraw (http://www.jhotdraw.org/). 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-modeler, renderer, and raytracer written in Java. 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 developments. Table 5 provides some descriptive statistics about these five programs.

Previous code changes applied to previous versions (sixth column in Table 5) are used mainly to calculate the importance score. In general, open-source programs and their change history (e.g., change log in CVS or SVN Cederqvist 2003) are available through SourceForge.net (http://sourceforge.net/). However, for other software programs especially where code change history is not publicly available, code changes could be expressed in

Table 5 Programs statistics

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

terms of recorded refactorings that are applied to previous versions (i.e., how many times a class experienced a refactoring). In order to vary our experiment settings in this paper, we are using both change log history (for Xerces and JFreeChart) and recorded refactorings (for GanttProject, AntApsche, JHotDraw, and Rhino). To collect refactorings applied for each program, we use Ref-Finder (Prete et al. 2010). Ref-Finder, implemented as an Eclipse plug-in, can identify refactoring operations between two releases of a software system.

4.2.2 Analysis method

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

(1) CCR is given by Eq. (2) and calculates the number of corrected code-smells over the total number of code-smells detected before applying the proposed refactoring sequence.

$$\mathrm{CCR} = \frac{\#\,\text{corrected code\_ smells}}{\#\,\text{code\_smells before applying refactorings}} \in [0,1] \tag{2}$$

(2) For the refactoring precision (RP), we inspect manually the feasibility of the different proposed refactoring sequences for each system. We applied the proposed refactorings using ECLIPSE (https://netbeans.org/), and we checked the semantic coherence of modified code fragments. Some semantic errors (programs behavior) were found. When a semantic error is found manually, we consider the operations related to this change as a bad recommendation. We calculate a correctness precision score (ratio of possible refactoring operations over the number of proposed refactoring) as usefulness indicator of our approach. RP is given by (3)

$$\mathrm{RP} = \frac{\#\,\text{feasable refactorings}}{\#\,\text{proposed refactorings}} \in [0,1] \tag{3}$$

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

(1) The importance correction ratio (ICR) that corresponds to the sum of importance score of detected code-smells after applying a given refactoring solution w compared to the one before applying refactoring. ICR reflects the efficiency of a refactoring solution for correcting important code-smells, so that the higher the ICR is, the more a refactoring solution is considered as a good recommendation. ICR is given by Eq. (4).

$$\mathrm{ICR}\,(w) = 1 - \frac{\sum_{i=0}^{n-1}(x_i * \text{importance}(c_i))}{\sum_{j=0}^{m-1}\left(x_j * \text{importance}(c_j)\right)} \in [0,1] \tag{4}$$

where n and m are the number of classes in the system, respectively, after and before applying the refactoring solution w, the function importance(c_i) returns the importance score of the class $c_{i,}$ and x_i takes the value 0 if the actual class c_i is detected as code-smell using code-smells detection rules (Ouni et al. 2012a), 1 otherwise.

(2) The risk correction ratio (RCR) that corresponds to the sum of importance score of detected code-smells after applying a given refactoring solution w compared to the

one before applying refactoring. RCR reflects the efficiency of a refactoring solution for correcting riskiest code-smells, so that the higher the RCR is, the more a refactoring solution is considered as a good recommendation. RCR is given by Eq. (5).

$$\text{RCR}\,(w) = 1 - \frac{\sum_{i=0}^{n-1}(x_i * \text{risk}(c_i))}{\sum_{j=0}^{m-1}\left(x_j * risk(c_j)\right)} \in [0,1] \tag{5}$$

where n and m are the number of classes in the system, respectively, after and before applying the refactoring solution w, the function $\text{risk}(c_i)$ returns the risk score of the class c_i, and x_i takes the value 0 if the actual class c_i is detected as code-smell using code-smells detection rules (Ouni et al. 2012a), 1 otherwise.

(3) The severity correction ratio (SCR) corresponds to the sum of importance score of detected code-smells after applying a given refactoring solution w compared to the one before applying refactoring. SCR reflects the efficiency of a refactoring solution for correcting severest code-smells, so that the higher the SCR is, the more a refactoring solution is considered as a good recommendation. SCR is given by Eq. (5).

$$\text{SCR}\,(w) = 1 - \frac{\sum_{i=0}^{n-1}(x_i * \text{severity}(c_i))}{\sum_{j=0}^{m-1}\left(x_j * \text{severity}(c_j)\right)} \in [0,1] \tag{6}$$

where n and m are the number of classes in the system, respectively, after and before applying the refactoring solution w, the function $\text{severity}(c_i)$ returns the severity score of the class c_i, and x_i takes the value 0 if the actual class c_i is detected as code-smell using code-smells detection rules (Ouni et al. 2012a), 1 otherwise.

For **RQ3**, we compare our approach to two other different approaches: Kessentini et al. (2011) and CRO without the use of prioritization that consider the refactoring suggestion task only from the quality improvement standpoint without considering prioritization.

Finally, to answer **RQ4**, we assessed the performance of the CRO algorithm we use in our approach compared to three other popular metaheuristic algorithms GA, SA, and PSO. We selected these three metaheuristics because they range from global search (GA and PSO) and local search (SA). Moreover, these three metaheuristics are the most frequent ones demonstrating good performance in solving different software engineering problems according to recent surveys (Harman et al. 2012)

4.3 Experiment results and evaluation

Before delving into details, we provide a high-level view of the experimental approach we adopted and its rationale. We first compared our approach to two other techniques that do not use prioritization (CRO without prioritization and Kessentini et al. 2011), where the fitness function calculates the number of corrected code-smells, to ensure the effectiveness of using such prioritization schema. Then, we compare the performance of CRO to three popular metaheuristics (GA, SA, and PSO) using the same fitness function to justify the use of CRO. Thus, due to the stochastic nature of the algorithms/approaches we are studying, each time we execute an algorithm we can get slightly different results. To cater for this issue and to make inferential statistical claims, our experimental study is performed based on 31 independent simulation runs for each algorithm/technique studied. Wilcoxon rank sum test (Arcuri and Briand 2011) is applied between CRO-based approach and each of the other algorithms/techniques (CRO without prioritization, Kessentini et al. 2011) in terms of CCR, ICR, RCR, and CSR with a 99 % confidence level ($\alpha = 1$ %). Our tests show that the obtained results are statistically significant with p value <0.01 and not due to chance. 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 $d \geq 0.8$. We have computed the effect size values for the different comparisons and we concluded that our CRO approach with prioritization has mainly: (1) medium effect size values against population-based metaheuristics under comparison, and (2) large effect size values against single-solution-based ones.

As described in Table 6 and Fig. 8, the majority of suggested refactorings by our approach improve significantly the code quality with good code-smells correction scores compared to both CRO without prioritization and Kessentini et al. For the five studied systems, our approach proves significant performance by fixing, on average, 90 % of all existing code-smells, whereas only 84 and 82 % for the other two approaches while focusing on fixing the prioritized code-smells. For instance, for JFreeChart, 92 % (24 over 26) of blobs, 94 % (16 over 17) of spaghetti code, and 79 % of functional decomposition (11 over 14) are fixed. This score is higher than the one of the other approaches having, respectively, only 81, 73 % of blobs, 76, 82 % of spaghetti code, and 79, 79 % of functional decomposition in terms of CCR scores. Moreover, after applying the proposed refactoring operations, for all systems, we found that most of the fixed code-smells (87 %) are related to important code fragments; however, only 69 and 66 % of ICR score are obtained by both other approaches that do not use prioritization (Table 6; Fig. 9). We also found that most of the fixed code-smells lie with the riskiest ones having a RCR average score of 92 %, while both other approaches provide only an average of 85 and 84 % of RCR as shown in Table 7 and Fig. 9. Additionally, the obtained results demonstrate that using the proposed prioritization schema, 89 % of severe code-smells were fixed, while both other approaches succeeded on fixing less than 81 % of severe code-smells.

Another important observation to highlight is that the majority of non-fixed code-smells obtained with both CRO without prioritization and Kessentini et al. (2011) are related to the blob type, as shown in Fig. 8. This type of code-smell usually requires a large number of refactoring operations and is then very difficult to correct without a specific mechanism (e.g., prioritization). On the other hand, the obtained CCR score related to data class is acceptable (an average of 79 % in all systems); however, we noticed that this score is less than the ones obtained by both other approaches. Thus, the loss in the data class correction ratio is largely compensated by the significant improvement in terms of importance, risk, and severity scores as shown in Fig. 9. In fact, this low score is due to the fact that data class is not prioritized in our experiments, we assign data classes the lowest priority score (equals to 1) unlike the blob code-smell, as described in Sect. 3.3. This score is assigned according to the developers' preferences. Moreover, in general, data classes do not experience changes frequently during the development and maintenance since it contains mainly data and performs no processing on these data (contains mainly setters and getters). To this end, the importance score related to this code-smell is very low. On the contrary, as shown in Table 6, all the detected shotgun surgery code-smells are fixed (a CCR score of 100 %). This is mainly due to the fact that shotgun surgery is extensively connected to a

Table 6 Refactoring results: code-smells

Systems	Approach	Code-smells correction ratio (CCR)							CCR (all code-smells) (%)
		Blob (%)	Spaghetti code (%)	Functional decomposition (%)	Data class (%)	Feature Envy (%)	Schizophrenic class (%)	Shotgun Surgery (%)	
Xerces-J	CRO (our approach)	94 (29|31)	92 (12|13)	86 (12|14)	90 (26|29)	90 (65|72)	100 (10|10)	100 (2|2)	91 (156|171)
	CRO without prioritization	84 (26|31)	85 (11|13)	93 (13|14)	97 (28|29)	85 (61|72)	80 (8|10)	50 (1|2)	87 (148|171)
	Kessentini et al. (2011)	84 (26|31)	77 (10|13)	86 (12|14)	97 (28|29)	83 (60|72)	70 (7|10)	50 (1|2)	84 (144|171)
JFreeChart	CRO (our approach)	92 (24|26)	94 (16|17)	79 (11|14)	89 (24|27)	85 (17|20)	92 (11|12)	100 (0|0)	89 (103|116)
	CRO without prioritization	81 (21|26)	76 (13|17)	79 (11|14)	100 (27|27)	80 (16|20)	83 (10|12)	100 (0|0)	84 (98|116)
	Kessentini et al. (2011)	73 (19|26)	82 (14|17)	79 (11|14)	100 (27|27)	80 (16|20)	100 (12|12)	100 (0|0)	85 (99|116)
GanttProject	CRO (our approach)	100 (7|7)	83 (5|6)	100 (18|18)	79 (11|14)	86 (6|7)	100 (1|1)	100 (0|0)	91 (48|53)
	CRO without prioritization	71 (5|7)	83 (5|6)	100 (18|18)	93 (13|14)	57 (4|7)	0 (0|1)	100 (0|0)	85 (45|53)
	Kessentini et al. (2011)	57 (4|7)	83 (5|6)	94 (17|18)	93 (13|14)	71 (5|7)	0 (0|1)	100 (0|0)	83 (44|53)
ArtofIllusion	CRO (our approach)	88 (15|17)	92 (11|12)	100 (8|8)	87 (27|31)	95 (42|44)	86 (12|14)	100 (1|1)	91 (116|127)
	CRO without prioritization	76 (13|17)	75 (9|12)	100 (8|8)	97 (30|31)	82 (36|44)	86 (12|14)	100 (1|1)	86 (109|127)
	Kessentini et al. (2011)	71 (12|17)	75 (9|12)	88 (7|8)	94 (29|31)	84 (37|44)	86 (12|14)	0 (0|1)	83 (106|127)
JHotDraw	CRO (our approach)	100 (4|4)	100 (3|3)	100 (5|5)	50 (2|4)	100 (4|4)	80 (4|5)	100 (0|0)	88 (22|25)
	CRO without prioritization	75 (3|4)	100 (3|3)	80 (4|5)	100 (4|4)	75 (3|4)	60 (3|5)	100 (0|0)	80 (20|25)
	Kessentini et al. (2011)	50 (2|4)	67 (2|3)	100 (5|5)	100 (4|4)	50 (2|4)	60 (3|5)	100 (0|0)	72 (18|25)
Average (all code-smells)	CRO (our approach)	95	92	93	79	91	91	100	90
	CRO without prioritization	78	84	90	97	76	62	90	84
	Kessentini et al. (2011)	67	77	89	97	74	63	70	82

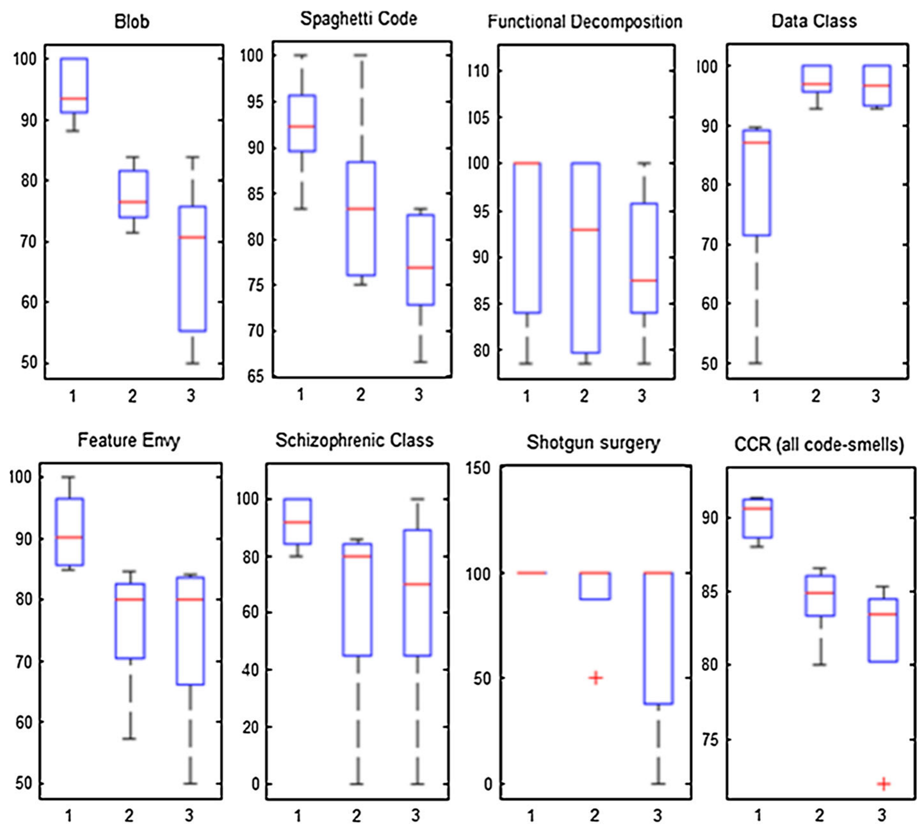


Fig. 8 Code-smells correction results for the five studied systems for each code-smell type for (1) CRO (our approach), (2) CRO without prioritization, and (3) Kessentini et al. 2011

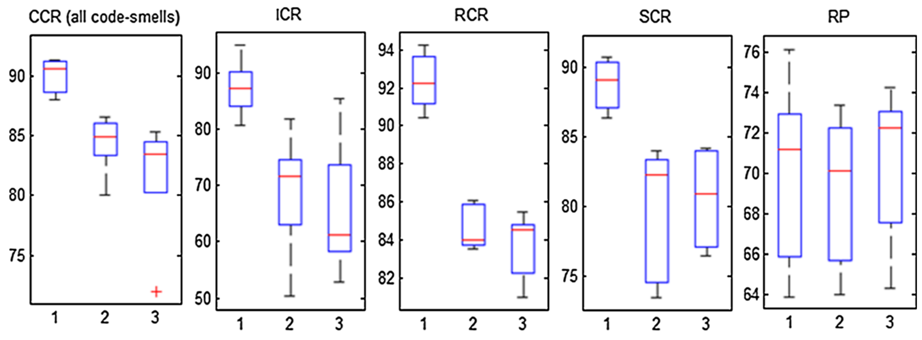


Fig. 9 Refactoring comparison results for the five systems for (1) CRO (our approach), (2) CRO without prioritization, and (3) Kessentini et al. 2011 in terms of ICR, RCR, SCR, and RP

large number of external methods calling it having large and widespread impact of a change. Consequently, its importance score is very high, and therefore, it will be more prioritized.

Table 7 Refactoring results: importance, risk, severity, and refactoring precision scores

Systems	Approach	ICR (%)	RCR (%)	SCR (%)	RP (%)
Xerces-J	CRO (our approach)	89	90	89	76 (230|302)
	CRO without prioritization	72	86	82	72 (245|341)
	Kessentini et al. (2011)	61	83	84	73 (261|359)
JFreeChart	CRO (our approach)	81	91	90	64 (152|238)
	CRO without prioritization	50	86	83	64 (151|236)
	Kessentini et al. (2011)	53	85	81	64 (155|241)
GanttProject	CRO (our approach)	87	93	87	67 (147|221)
	CRO without prioritization	67	83	75	66 (145|219)
	Kessentini et al. (2011)	60	81	76	69 (166|242)
ArtofIllusion	CRO (our approach)	85	92	91	71 (205|288)
	CRO without prioritization	72	84	84	70 (176|251)
	Kessentini et al. (2011)	70	85	84	72 (180|249)
JHotDraw	CRO (our approach)	95	94	86	72 (146|203)
	CRO without prioritization	82	84	73	73 (160|218)
	Kessentini et al. (2011)	85	85	77	74 (147|198)
Average (all systems)	CRO (our approach)	87	92	89	70
	CRO without prioritization	69	85	80	69
	Kessentini et al. 2011	66	84	81	70

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

To sum up, we present in Fig. 9 the metric scores for all systems using boxplots. The majority of code-smells (90 %), on average, were corrected using our approach that outperforms both CRO without prioritization and Kessentini et al. approach in terms of code-smells correction ratio. However, only for data classes, the obtained results are slightly less than other approaches. In general, this kind of code-smells is less risky/important than other code-smells and not need an extensive correction effort by software engineers compared to the blob. Hence, to fix data class, software maintainers can easily apply some refactorings such as inline class, move method/field to add new behavior/functionalities or merge data classes with other existing classes in the system. Although data classes are not prioritized in our approach, we obtained an acceptable correction score. This is due to the fact that blob are in general related to data classes (Brown et al. 1998); consequently, fixing blobs can implicitly fix its related data classes. We also had good results in terms of importance, risk, and severity correction scores. The majority of important, riskiest, and severest code-smells were fixed, and most of the proposed refactoring sequences (70 %) are coherent semantically.

To better evaluate our approach and to answer RQ4, we compare the results of the CRO-based approach with three different population and single-solution-based EAs (GA, SA, and PSO), which have been shown to have good performance in solving different software engineering problems (Harman 2007; Harman et al. 2012). For all algorithms, we use the same formulation given in Sect. 3.3 (objective function, solution representation, etc.) with the algorithms configuration described in Sect. 3.4.4. Table 8 shows the comparison results among the median of solution's quality for each pair of algorithms using Wilcoxon rank sum test (Arcuri and Briand 2011). As shown in Table 8, at 99 % of CI, the median values of CRO and GA, CRO and SA, as well as those of CRO and PSO are statistically different in terms of CCR, ICR, and RCR. However, in terms of RP, CRO, and GA, and CRO and PSO are not. The comparison results, sketched in Table 8 and Fig. 10, showthat CRO outperforms the other three algorithms in terms of CCR, ICR, and RCR while having similar performance in terms of RP (70 %). For instance, using CRO, an average of 90 % of code-smells is fixed, whereas only 84, 83, and 84 % are obtained with GA, SA, and PSO. Moreover, in terms of ICR, CRO succeeded on fixing, 87 % of important code-smells, while obtained ones for other algorithms are less than fixes less than 83 %. Based on these results, we can conjecture that CRO performs much better in comparison with GA, SA, and PSO. Moreover, we notice that SA turns out to be the worst algorithm.

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

Despite that the time constraints are not a real challenge in our proposal, it is relevant to analyze the convergence speed when we compare metaheuristics. To this end, we performed 31 independent simulation runs on the same PC with Intel Core i5-2450 M Processor and 4 GB of RAM for each of the algorithms CRO, GA, SA, and PSO. Hence, SA manipulates a single solution in each iteration, while GA and PSO control a population of solutions at a time. CRO is also a population-based metaheuristics; however, the number of manipulated solutions varies during a simulation run. Through 31 independent simulation runs, we found that PSO converges faster than the other algorithms (an average of 48 m 12 s). CRO is the second one in terms of convergence speed (an average of 48 m 18 s over 31 run) while we record an average of 1 h 32 m 47 s and 1 h 23 m 13 s for, respectively, GA and SA. However, code-smells fixing and refactoring activities are not related in general to real-time applications where time constraints are very important, that is, when done manually, code-smells correction is considered to be time-consuming, error-prone, and fastidious task for software engineers (Brown et al. 1998; Moha et al. 2010).

To sum up, we can conclude that CRO outperforms other popular metaheuristic algorithms (Kirkpatrick et al. 1983; Goldberg 1989; Kennedy and Eberhart 1995). In fact, there are two reasons for the high convergence speed of CRO. The first is the ability for CRO to jump out of a local minimum, by means of the four elementary reaction operators, and quickly search other possible better results. The second is due to the efficient encoding scheme and the variety of change operators, which greatly explores the search space.

4.4 Discussions

Our results indicate that our approach significantly outperforms two other approaches that do not use the prioritization for correcting code-smells. We also found that CRO performs much better than three other popular metaheuristic algorithms: GA, SA, and PSO. We also contrast the results of multiple executions with the execution time to evaluate the performance and the stability of our approach. Moreover, we evaluate the impact of the

Table 8 CCR, ICR, RCR, SCR, and RP median values of CRO, GP, SA, and PSO over 31 independent simulation runs

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

Table 8 continued

Systems	Algorithms	CCR (%)		ICR (%)		RCR (%)		SCR (%)		RP (%)	
		Score	*p* value	Score	*p* value	Score	*p* value	Score	*p* value	Score	*p* value
Average (all systems)	CRO	90 %		87 %		92 %		89 %		70 %	
	GA	85 %		83 %		89 %		87 %		70 %	
	SA	81 %		82 %		89 %		87 %		70 %	
	PSO	85 %		83 %		89 %		88 %		70 %	

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

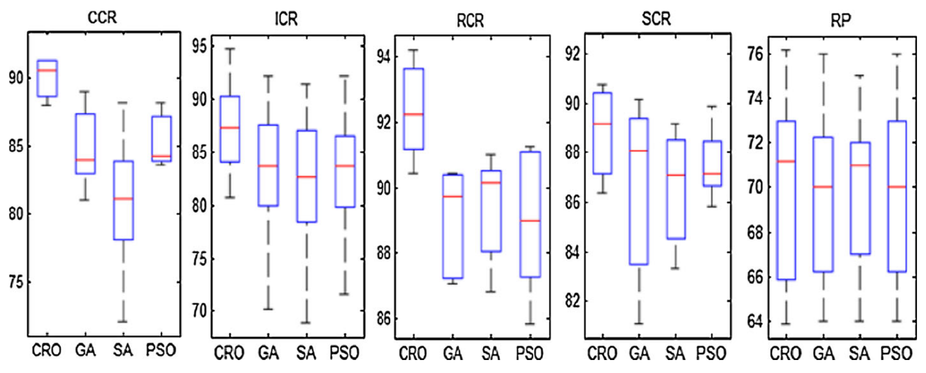
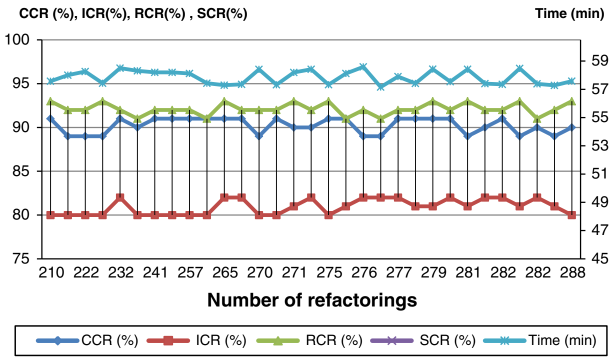


Fig. 10 CRO performance comparison with GA, SA, and PSO

Fig. 11 Impact of the number of refactorings on multiple simulation runs on JFreeChart



number of suggested refactorings on the CCR, ICR, RCR, and SCR scores and the execution time through 31 independent simulation runs. Over 31 independent simulation runs on JFreeChart, the average value of CCR, ICR, RCR, SCR, and execution time for finding the optimal refactoring solution with the suitable prioritization schema was, respectively, 90, 80, 92, 90 %, and 57 min 36 s as shown in Fig. 11. The standard deviation values was lower than 1. Moreover, the results of Fig. 11, drawn for JFreeChart, show that the number of suggested refactorings does not affect the refactoring results. Thus, a higher number of operations in a solution do not necessarily mean that the results will be better. Thus, we could conclude that our approach is scalable from the performance standpoint, especially that quality improvements are not related in general to real-time applications where time constraints are very important. In addition, the results accuracy is not affected by the number of suggested refactorings.

Another important consideration is the refactoring operations distribution. We contrast that most of the suggested refactorings are related to move method, move field, and extract class for almost all of the studied systems. For instance, in JFreeChart, we had different distribution of different refactoring types as illustrated in Fig. 12. We notice that the most suggested refactorings are related to moving code elements (fields, methods) and extract/inline class. This is mainly due to the code-smells types detected in JFreeChart and prioritized during the correction step. Most of code-smells are related to the blob, functional decomposition, and spaghetti code that need particular refactorings. For instance, to

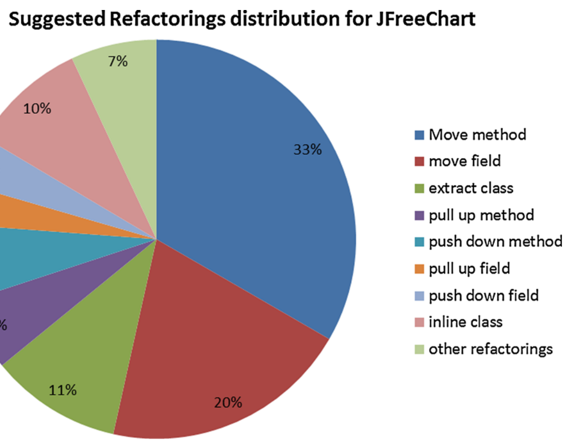


Fig. 12 Suggested refactorings distribution for JFreeChart

fix a blob code-smell, the idea is to move elements from the blob class to other classes (e.g., data classes) in order to reduce the number of functionalities from the blob and add behavior to other classes or to improve some quality metrics such as coupling and cohesion. As such, refactorings like move field, move method, and extract class are likely to be more useful to correct the bloc code-smell. Furthermore, before starting our experiments and analyzing our refactoring results, we expected that code-smells correction score for data classes will be very low; however, we found that most of them are corrected with a good score (an average of 79 %). This is mainly due to two reasons: (1) data classes are to some extent easy to fix and they do not need lot of refactorings to be fixed; it is sufficient to add some functionalities/methods to them from other related classes, and (2) in general, there is a structural relationship between data classes and blobs (Brown et al. 1998; Riel 1996), so that fixing blobs can implicitly fix data classes related to them. Enforcing the correction of blobs can implicitly increase the correction of data classes. As part of future work, we plan to conduct a large empirical study to investigate the relationship between different code-smells, and between code-smell types and refactoring types.

4.5 Threats to validity

Several threats can affect the validity of our experiments. We explore in this subsection the factors that can bias our experimental study. These factors can be classified into three categories: construct, internal, and external validity. Construct validity concerns the relationship between the theory and the observation. Internal validity concerns possible bias with the results obtained by our proposal. Finally, external validity is related to the generalization of observed results outside the sample instances used in the experiment.

In our experiments, construct validity is related to the code-smells detection rules (Kessentini et al. 2011) we use to measure CCR that could be questionable. To mitigate this threat, we manually inspect and validate each detected code-smell. Moreover, our refactoring tool configuration is flexible and can support other state-of-the-art detection rules.

We take into consideration the internal threats to validity in the use of stochastic algorithms since our experimental study is performed based on 31 independent simulation runs for each problem instance, and the obtained results are statistically analyzed by using the Wilcoxon rank sum test (Arcuri and Briand 2011) with a 99 % CI ($\alpha = 1$ %). However, despite we used the same stopping criteria, the parameter tuning of the different optimization algorithms used in our experiments creates another internal threat that we need to evaluate in our future work.

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

5 Related work

A large number of research works have addressed the problem of code-smells correction and software refactoring in recent years. In this section, we survey those works that can be classified mainly into two broad categories: manual and semiautomated approaches, and search-based approaches.

5.1 Manual and semiautomated approaches

We start by summarizing existing manual and semiautomated 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 the 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 bad smells and refactoring opportunities. For each bad smell, a predefined 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 start from the hypothesis that refactoring opportunities correspond of those that improves cohesion and coupling metrics to perform an optimal distribution of features over classes. Du Bois et al. analyze how refactorings manipulate coupling and cohesion metrics, and how to identify refactoring opportunities that improve these metrics. However, these two approaches are limited to only some possible refactoring operations with few number 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 the concept analysis aimed at identifying less cohesive classes. It also helps identify less cohesive methods, attributes, and classes in one go. Further, the approach guides refactoring opportunities identification such as extract class, move method, localize attributes, and remove unused attributes. In addition, Tahvildari 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 the rules that can be expressed as assertions (invariants, pre-, and post-conditions). The use of invariants has been proposed to detect parts of program that require refactoring by Kataoka et al. (2001). In addition, Opdyke (1992) has proposed the definition and the use of pre- and post-conditions 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-conditions. All these conditions are expressed in terms of rules.

The major limitation of these manual and semiautomated approaches is that they try to apply refactorings separately without considering the whole program to be refactored and its impact on the other artifacts. Indeed, these approaches are limited to only some possible refactoring operations and few number of quality metrics to asses quality improvement. In addition, improving some quality metrics does mean necessary that actual code-smells are fixed. Another important issue is that these approaches do not take into consideration the human effort needed to apply the suggested refactorings neither the software maintainer preferences nor the importance/severity of some code-smell types.

5.2 Search-based approaches

Search-based approaches are largely inspired by contributions in SBSE (Harman and Andjones 2001). Recently, new approaches have emerged where search-based techniques have been used recently to automate refactoring activities. These approaches cast the refactoring as an optimization problem, where the goal is to improve the design quality of a system based mainly on a set of software metrics. After formulating the refactoring as an optimization problem, several different techniques can be applied for automating refactoring, e.g., GAs, SA, and Pareto optimality. Hence, we classify those approaches into two main categories: mono-objective and multi-objective optimization approaches.

In the first category, the majority of existing work combines several metrics in a single fitness function to find the best sequence of refactorings. Seng et al. (2006) have proposed a single-objective optimization-based approach using a GA 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 preconditions for each refactoring. These conditions serve to preserve 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 SA to provide an automated refactoring support. Eleven weighted object-oriented design metrics have been used to evaluate the quality improvements. 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. However, 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 GA 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 in this work is to explore the addition of a refactoring step into the genetic programming iteration. There will be an additional loop in which refactoring steps drawn from a catalog of such steps will be applied to individuals of the population. Jensen et al. (Jensen and Cheng 2010) have proposed an approach that supports the 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, coupling between objects (CBO) and standard deviation of methods per class (SDMPC), in two separate objective functions. The authors start from the assumption that good design quality results from good distribution of features (methods) among classes. Their Pareto optimality-based algorithm succeeded in finding good sequence of "move method" refactorings that should provide the best compromise between CBO and SDMPC to improve code quality. However, one of the limitations of this approach is that it is limited to unique refactoring operation (move method) to improve software quality and only two metrics to evaluate the preformed improvements. 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.(2012a) 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. Furthermore, Ouni et al. (2012a) 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. Later, in Ouni et al. (2013), the authors have integrated a new objective to maximize the reuse of good refactorings applied to similar contexts to propose new refactoring solutions.

To conclude, in most of the existing search-based approaches, code-smells are treated with the same importance and risk. The fitness function, in general, minimizes the number of detected code-smells in the system after applying some refactoring operations. However, fixing the majority of code-smells did not mean that the riskiest ones are corrected.

Although in SBSE most of contributions are based on classic EAs (Kirkpatrick et al. 1983; Goldberg 1989; Kennedy and Eberhart 1995; Glover and Laguna 1997), the scope of this paper is to illustrate one of the first attempts of using CRO to solve software engineering problems and particularly the code-smells correction one. Indeed, to the best of our knowledge and based on recent surveys in SBSE (Harman et al. 2012), CRO is not yet explored in software engineering, and the idea of treating code-smells correction as an optimization problem to be solved by a SBSE approach was not studied before our proposal in (Kessentini et al. 2011). In this paper, we propose a new search-based approach using CRO with new formulation of refactoring task to correct code-smells.

6 Conclusion

This paper presented a novel CRO-based approach to suggest "good" refactoring solutions to fix code-smells while taking into consideration software maintainers' preferences. The suggested refactorings succeed in fixing the majority of important, riskiest, and severest code-smells. Experiments on five medium- and large-scale software systems and seven code-smell types show that the proposed approach is superior to two other approaches that do not use prioritization and maintainers' preferences to automate the refactoring task. Moreover, the proposed CRO-based approach was compared successfully to three different popular metaheuristics in SBSE. As future work, we are planning to conduct an empirical study to understand the correlation between correcting code-smells and introducing new ones or fixing other code-smells implicitly. We also plan to adapt our approach into a multi-objective one to take into consideration new objectives such as reducing the number of changes needed to apply the suggested refactoring solution, and improve the semantic coherence of the refactored program.

References

Alba, E., & Chicano, F. (2005). Management of software projects with GAs. In *Proceedings of the 6th metaheuristics international conference (MIC'05)*. Elsevier, 13–18, 2005.

Alikacem, H., & Sahraoui. H. (2006). Détection d'anomalies utilisant un langage de description de règle de qualité, in actes du 12e colloque LMO.

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 (ICSE '11)*, (pp. 1–10). New York, NY, USA: ACM.

Brown, W. J., Malveau, R. C., Brown, W. H., McCornnick, W. H, I. I. I., & Mowbray, T. J. (1998). *Anti patterns: Refactoring software, architectures, and projects in crisis*. New York: Wiley.

Canfora, G., Di Penta, M., Esposito, R., & Andvillani, M. L. (2005) An approach for QoS-aware service composition based on genetic algorithms. In *Proceedings of the conference on genetic and evolutionary computation (GECCO'05)*, (pp. 1069–1075). New York: ACM Press.

Cederqvist, P. (2003). *Version management with CVS*, December 2003. www.cvshome.org/docs/manual/.

Chidamber, S. R., & Kemerer, C. F. (1994). A metrics suite for object oriented design. *IEEE Transactions on Software Engineering, 20*(6), 476–493.

Cinnéide, M. Ó., Tratt, L., Harman, M., Counsell, S., & Moghadam, I. H. (2012). Experimental assessment of software metrics using automated refactoring. In *Proceedings of empirical software engineering and management (ESEM)*, pp. 49–58, September 2012.

Cohen, J. (1988). *Statistical power analysis for the behavioral sciences* (2nd ed.). New York, USA: Lawrence Earlbaum Associates.

Du Bois, B., Demeyer, S., & Verelst, J. (2004). Refactoring—Improving coupling and cohesion of existing code. In *Proceedings of 11th working conference reverse engineering (WCRE)*, 2004, pp. 144–151.

Erlikh, L. (2000). Leveraging legacy system dollars for e-business. *IT Professional, 02*(3), 17–23.

Fatiregun, D., Harman, M., & Hierons, R. (2004). Evolving transformation sequences using genetic algorithms. In *SCAM 04*, (pp. 65–74). Los Alamitos, CA, USA: IEEE Computer Society Press.

Fenton, N., & Pfleeger, S. L. (1997). *Software metrics: A rigorous and practical approach* (2nd ed.). London: International Thomson Computer Press.

Fowler, M., Beck, K., Brant, J., Opdyke, W., & Roberts, D. (1999). *Refactoring: Improving the design of existing code* (1st ed.). Reading, MA: Addison-Wesley.

Freitas, F. G., & Souza, J. T. (2011). Ten years of search based software engineering: A bibliometric analysis. In *3rd international symposium on search based software engineering* (SSBSE 2011), 10–12th September 2011, pp. 18–32.

Fraser, G., & Arcuri, A. (2013). Handling test length bloat. *Software Testing, Verification and Reliability, 23*(7), 553–582.

Glover, F., & Laguna, M. (1997). *Tabu search*. Boston, MA: Kluwer Academic Publishers.

Goldberg, D. E. (1989). *Genetic algorithms in search, optimization, and machine learning*. Reading, MA: Addison-Wesley.

GanttProject. [Online].

Harman, M. (2007). The current state and future of search based software engineering. In L. Briand & A. Wolf (Eds.), *Future of software engineering 2007* (pp. 342–357). Los Alamitos, CA: IEEE Computer Society Press.

Harman, M., & Jones, B. F. (2001). Search-based software engineering. *Information and Software Technology, 43*(14), 833–839.

Harman, M., & Tratt, L. (2007) Pareto optimal search based refactoring at the design level. In *Proceedings of the genetic and evolutionary computation conference* (GECCO'07), 2007, pp. 1106–1113.

Harman, M., Mansouri, S. A., & Zhang, Y. (2012). Search-based software engineering: Trends, techniques and applications. *ACM Computing Surveys, 45*, 61.

Infusion hydrogen: Design flaw detection tool. 2012.

Jensen, A., & Cheng, B. (2010). On the use of genetic programming for automated refactoring and the introduction of design patterns. In *Proceedings of GECCO*. ACM, July 2010.

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, 2009.

JFreeChart. [Online].

JHotDraw. [Online].

iPlasma.

Jourdan, L., Basseur, M., & Talbi, E.-G. (2009). Hybridizing exact methods and metaheuristics: A taxonomy. *European Journal of Operational Research, 199*(3), 620–629.

Kataoka, Y., Ernst, M. D., Griswold, W. G., & Notkin, D. (2001). Automated support for program refactoring using invariants. In *International conference on software maintenance (ICSM),* pp. 736–743, 2001.

Kennedy, J., & Eberhart, R. C. (1995). Particle swarm optimization. In *Proceedings of IEEE international conference neural networks*, Perth, Australia, November 1995, pp. 1942–1948.

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, ASE'10*, 2010.

Kessentini, M., Kessentini, W., Sahraoui, H., Boukadoum, M., & Ouni, A. (2011) Design defects detection and correction by example. In *19th IEEE ICPC11*, Kingston, Canada, pp. 81–90.

Khomh, F., Penta, M. D., & Gueheneuc, Y.-G. (2009a). An exploratory study of the impact of code smells on software change-proneness. In *Proceedings of WCRE 2009 (16th IEEE working conference on reverse engineering)*, pp. 75–84, 2009.

Khomh, F., Vaucher, S., Guéhéneuc, Y.-G., & Sahraoui, H. (2009b). A bayesian approach for the detection of code and design smells. In *Proceedings of the ICQS'09*.

Kilic, H., Koc, E., & Cereci, I. (2011). Search-based parallel refactoring using population-based direct approaches. In *Proceedings of the third international conference on search based software engineering, SSBSE'11*, pp. 271–272, 2011.

Kirkpatrick, S., Gelatt, C. D, Jr, & Vecchi, M. P. (1983). Optimization by simulated annealing. *Science, 220*(4598), 671–680.

Lam, A. Y. S., & Li, V. O. K. (2010). Chemical-reaction-inspired metaheuristic for optimization. *IEEE Transactions Evolutionary Computation, 14*(3), 381–399.

Lam, A. Y. S., Li, V. O. -K., & Wei, Z. (2012a). Chemical reaction optimization for the fuzzy rule learning problem. *Evolutionary Computation (CEC), 2012 IEEE Congress on* 10–15 June 2012, pp. 1–8.

Lam, A. Y. S., Li, V. O. K., & Yu, J. J. Q. (2012b). Real-coded chemical reaction optimization. *IEEE Transactions Evolutionary Computation, 16*(3), 339–353.

Lam, A. Y. S., Li, V. O. K., & Xu, J. (2013). On the convergence of chemical reaction optimization for combinatorial optimization. *IEEE Transaction Evolutionary Computation, 17*(5), 605–620.

Le Goues, C., Nguyen, T., Forrest, S., & Weimer, W. (2012). Genprog: A generic method for automatic software repair. *IEEE Transactions on Software Engineering, 38*(1), 54–72.

Marinescu, R. (2004). Detection strategies: Metrics-based rules for detecting design flaws. In *Proceedings of the 20th international conference on software maintenance* (pp. 350–359). IEEE Computer Society Press.

Martin, R. C. (2008). *Clean code: A handbook of agile software craftsmanship*. Englewood Cliffs: Prentice Hall.

McMinn, P. (2004). Search-Based software test data generation: A survey. *Software Testing, Verification and Reliability, 14*, 2.

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

Moha, N., Hacene, A., Valtchev, P. & Guéhéneuc, Y.-G. (2008). Refactorings of design defects using relational concept analysis. In R. Medina, & S. Obiedkov (Eds.), *Proceedings of the 4th international conference on formal concept analysis* (ICFCA 2008), February 2008.

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. *Transactions on Software Engineering (TSE) 36*(1), 20–36.

Munro, M. J. (2005). Product metrics for automatic identification of bad smell" design problems in java source-code. In F. Lanubile, & C. Seaman (Eds.), *Proceedings of the 11th international software metrics symposium*. IEEE Computer Society Press (2005).

Ó Cinnéide, M. (2000). *Automated application of design patterns: A refactoring approach*. Ph.D. dissertation, University of Dublin, Trinity College, Department of Computer Science, 2000.

O'Keeffe, M., & Cinnéide, M. O. (2006). Search-based refactoring for software maintenance. *Journal of Systems and Software, 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 *ESEM*, 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 *Software maintenance, ICSM 2010*, pp. 1–10, Timisoara, 2010.

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

Otero, F. E. B., Johnson, C. G., Freitas, A. A., & Thompson., S. J. (2010) Refactoring in automatically generated programs. In *Search based software engineering, international symposium on,* 0, 2010.

Ouni, A., Kessentini, M., Sahraoui, H., & Boukadoum, M. (2012a). Maintainability defects detection and correction: A multi-objective approach. *Journal of Automated Software Engineering*, *20*(1), 47–79.

Ouni, A., Kessentini, M., Sahraoui, H., & Hamdi, M. S. (2012b) Search-based refactoring: Towards semantics preservation. In *28th IEEE international conference on software maintenance (ICSM)*, (pp. 347–356), September 23–28, 2012.

Ouni, A., Kessentini, M., & Sahraoui, H. (2013). Search-based refactoring using recorded code changes. In *Proceedings of the 17th European conference on software maintenance and reengineering (CSMR)*, Genova, Italy, March 5–8, 2013.

Prete, K., Rachatasumrit, N., Sudan, N., & M. Kim. (2010) Template-based reconstruction of complex refactorings. In *Proceedings of the international conference on software maintenance (ICSM)*, 2010.

Qayum, F., & Heckel, R. (2009) Local search-based refactoring as graph transformation. In *Proceedings of 1st international symposium on search based software engineering*, pp. 43–46, 2009.

Ratiu, D., Ducasse, S., Gîrba, T., & Marinescu, R. (2004) Using history information to improve design flaws detection. In *CSMR*, pp. 223–232.

Riel, A. J. (1996). *Object-oriented design heuristics*. Reading, MA: Addison-Wesley.

Roberts, D. B. (1999). *Practical analysis for refactoring*. PhD thesis, Department of Computer Science, University of Illinois, 1999.

Sahraoui, H., Godin, R., & Miceli, T. (2000). Can metrics help to bridge the gap between the Improvement of OO design quality and its automation? *ICSM, 2000*, 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 (GECCO'06)*, 2006, pp. 1909–1916.

Sun, Y., Lam, A. Y. S., Li, V. O. -K., Xu, J., & Yu, J. J. Q. (2012). Chemical reaction optimization for the optimal power flow problem. *Evolutionary Computation (CEC), 2012 IEEE Congress on* 10–15 June 2012, pp. 1–8.

Tahvildari, L., & Kontogiannis, K. (2003). A metric-based approach to enhance design quality through meta-pattern transformation. In *Proceedings of the 7th European conference on software maintenance and reengineering*, Benevento, Italy, pp. 183–192, 2003.

Tsantalis, N., Chaikalis, T., & Chatzigeorgiou, A. (2008). JDeodorant: Identification and removal of type-checking bad smells. In *Proceedings of CSMR2008*, pp 329–331.

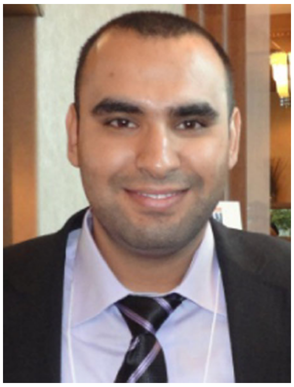
Vallée-Rai, R., Gagnon, E., Hendren, L. J., Lam, P., Pominville, P., & Sundaresan, V. (2000). Optimizing Java bytecode using the Soot framework: Is it feasible? In *International conference on compiler construction*, pp. 18–34, 2000.

Xu, J., Lam, A. Y. S., & Li, V. O. K. (2011). Chemical reaction optimization for task scheduling in grid computing. *IEEE Transactions on Parallel and Distributed Systems, 22*(10), 1624–1631.

Xerces-J. [Online].

Yu, J. J. Q., Lam, A. Y. S., & Li, V. O. -. (2012) Real-coded chemical reaction optimization with different perturbation functions. *Evolutionary Computation (CEC), 2012 IEEE Congress on* 10–15 June 2012, pp. 1–8.

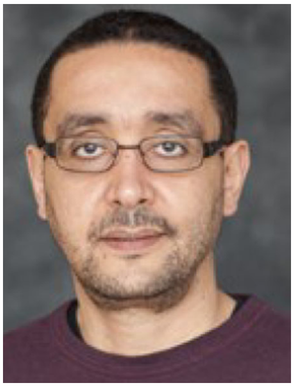
Zhang, Y., Finkelstein, A., & Andharman, M. (2008). Search based requirements optimisation: Existing work and challenges. In *Proceedings of the 14th international working conference, requirements engineering: Foundation for software quality (RefsQ'08).* Lecture notes in computer science, (Vol. 5025, pp. 88–94). New York: Springer.



Ali Ouni joined the University of Montreal in January 2011. He is currently a PhD student in computer science at the University of Montreal under the supervision of Prof. Houari Sahraoui and Prof. Marouane Kessentini (University of Michigan). He is a member of the GEODES software engineering laboratory, in the Department of Computer Science and Operational Research (DIRO) and member of the SBSE Laboratory, in the Computer and Information Science Department, University of Michigan. He received his bachelor degree (BSc) and his master degree diploma (MSc) in computer science from the University of Sousse, Tunisia, respectively, in 2008 and 2010. His research interests include the application of artificial intelligence techniques to software engineering (search-based software engineering). He serves as program committee member in several conferences and journals such as GECCO'14. He is a student member of the IEEE and the IEEE Computer Society.

Marouane Kessentini is Assistant Professor at University of Michigan. He is the founder of the research group: Search-based Software Engineering@Michigan. He holds a PhD 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 45 papers in conferences, workshops, books, and journals including four best paper awards. He has served as program committee member in several conferences and journals such as GECCO'13 and ICMT'13, and as organization member of many conferences and workshops. He is the co-chair of the SBSE track at GECOO'14.

Slim Bechikh received the BSc degree in computer science applied to management and the MSc degree in modeling from the University of Tunis, Tunisia, in 2006 and 2008, respectively. He also obtained the PhD degree in computer science applied to management from University of Tunis in January 2013. He worked, for four years, as an attached researcher within the Optimization Strategies and Intelligent Computing laboratory (SOIE), Tunisia. Actually, he is a postdoctoral researcher at the SBSE@Michigan laboratory, University of Michigan, USA. His research interests include multi-criteria decision making, evolutionary computation, multi-agent systems, portfolio optimization, and search-based software engineering. Since 2008, he published several papers in well-ranked journals and conferences. Moreover, he obtained the best paper award of the ACM Symposium on Applied Computing 2010 in Switzerland among more than three hundreds participants. Since 2010, he serves as reviewer for several conferences such as ACM SAC and GECCO and various journals such as Soft Computing and IJITDM.



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 PhD 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.