Code-Smell Detection as a Bilevel Problem

DILAN SAHIN, MAROUANE KESSENTINI, and SLIM BECHIKH, University of Michigan, Dearborn
KALYANMOY DEB, Michigan State University

Code smells represent design situations that can affect the maintenance and evolution of software. They make the system difficult to evolve. Code smells are detected, in general, using quality metrics that represent some symptoms. However, the selection of suitable quality metrics is challenging due to the absence of consensus in identifying some code smells based on a set of symptoms and also the high calibration effort in determining manually the threshold value for each metric. In this article, we propose treating the generation of code-smell detection rules as a bilevel optimization problem. Bilevel optimization problems represent a class of challenging optimization problems, which contain two levels of optimization tasks. In these problems, only the optimal solutions to the lower-level problem become possible feasible candidates to the upper-level problem. In this sense, the code-smell detection problem can be treated as a bilevel optimization problem, but due to lack of suitable solution techniques, it has been attempted to be solved as a single-level optimization problem in the past. In our adaptation here, the upper-level problem generates a set of detection rules, a combination of quality metrics, which maximizes the coverage of the base of code-smell examples and artificial code smells generated by the lower level. The lower level maximizes the number of generated artificial code smells that cannot be detected by the rules produced by the upper level. The main advantage of our bilevel formulation is that the generation of detection rules is not limited to some code-smell examples identified manually by developers that are difficult to collect, but it allows the prediction of new code-smell behavior that is different from those of the base of examples. The statistical analysis of our experiments over 31 runs on nine open-source systems and one industrial project shows that seven types of code smells were detected with an average of more than 86% in terms of precision and recall. The results confirm the outperformance of our bilevel proposal compared to state-of-art code-smell detection techniques. The evaluation performed by software engineers also confirms the relevance of detected code smells to improve the quality of software systems.

Categories and Subject Descriptors: D.2 [**Software**]: Software Engineering

General Terms: Algorithms, Reliability

Additional Key Words and Phrases: Search-based software engineering, software quality, code smells

ACM Reference Format:
Dilan Sahin, Marouane Kessentini, Slim Bechikh, and Kalyanmoy Deb. 2014. Code-smell detection as a bilevel problem. ACM Trans. Softw. Eng. Methodol. 24, 1, Article 6 (September 2014), 44 pages.
DOI: http://dx.doi.org/10.1145/2675067

6

1. INTRODUCTION

Large-scale software systems exhibit high complexity and become difficult to maintain. In fact, it has been reported that software cost dedicated to maintenance and evolution activities is more than 80% of the total software costs [Bennett and Rajlich 2000]. In addition, it is shown that software maintainers spend around 60% of their time in understanding the code [Abran and Nguyenkim 1993]. In particular, object-oriented software systems need to follow some traditional set of design principles, such as data

Author's addresses: D. Sahin, M. Kessentini (corresponding author), and S. Bechikh, University of Michigan; emails: {dilan, marouane, slim}@umich.edu; K. Deb, Michigan State University; email: kdeb@egr.msu.edu.

DOI: http://dx.doi.org/10.1145/2675067

abstraction, encapsulation, and modularity. However, some of these nonfunctional requirements can be violated by developers for many reasons, such as inexperience with object-oriented design principles, deadline stress, and much focus on only implementing main functionality. This high cost of maintenance activities could potentially be reduced greatly by providing automatic or semiautomatic solutions to increase a system's comprehensibility, adaptability, and extensibility to avoid bad practices.

As a consequence, there has been much research focusing on the study of bad design practices, also called code smells, defects, anti-patterns, or anomalies [Fowler et al. 1999], in the literature. Although these bad practices are sometimes unavoidable, they should be in general prevented by the development teams and removed from their code base as early as possible. In fact, detecting and removing these code smells help developers to easily understand source code [Brown et al. 1998]. In this work, we focus on the detection of code smells.

The vast majority of existing work in code-smell detection relies on declarative rule specification [Mika and Lassenius 2006]. In these settings, rules are manually defined to identify the key symptoms that characterize a code smell using combinations of mainly quantitative (metrics), structural, and/or lexical information. However, in an exhaustive scenario, the number of possible code smells used to manually characterize with rules can be large. For each code smell, rules that are expressed in terms of metric combinations need substantial calibration efforts to find the right threshold value for each metric. Another important issue is that translating symptoms into rules is not obvious, because there is no consensual symptom-based definition of code-smells [Mika 2010]. When consensus exists, the same symptom could be associated to many code-smell types, which may compromise the precise identification of code-smell types. These difficulties explain a large portion of the high false-positive rates reported in existing research. Recently, a Search-Based Software Engineering (SBSE) approach [Harman et al. 2012], based on genetic programming [Goldberg 1989], was used to generate code-smell detection rules from a set of examples of code smells identified manually by developers [Kessentini et al. 2011]. However, such approaches require a high number of code-smell examples (data) to provide efficient detection rules solutions. In fact, code smells are not usually documented by developers (unlike bugs report). Thus, it is time consuming and difficult to collect code smells and manually inspect large systems. In addition, it is challenging to ensure the diversity of the code-smell examples to cover most of the possible bad practices.

In this work, we start from the observation that the generation of efficient code-smell detection rules heavily depends on the coverage and diversity of the used code-smell examples. In fact, both mechanisms for the generation of detection rules and the generation of code smells examples are dependent. Thus, the intuition behind this work is to generate examples of code smells that cannot be detected by some possible detection rules solutions, then adapting these rule-based solutions to be able to detect the generated code-smell examples. These two steps are repeated until reaching a termination criterion (e.g., number of iterations). To this end, we propose, for the first time, considering the code-smell detection problem as a bilevel one [Colson et al. 2005b]. Bilevel optimization problems (BLOPs) are a class of challenging optimization problems, which contain two levels of optimization tasks [Kolstad 1985]. In these problems, the optimal solutions to the lower-level problem become possible feasible candidates to the upper-level problem. In our adaptation, the upper level generates a set of detection rules, a combination of quality metrics, which maximizes the coverage of the base of code-smell examples; and artificial code smells are generated by the lower level. The lower level maximizes the number of generated "artificial" code smells that cannot be detected by the rules produced by the upper level. The overall problem appears as a BLOP task, where for each generated detection rule, the upper level observes how

the lower level acts by generating artificial code smells that cannot be detected by the upper-level (leader) rule, and then chooses the best detection rule which suits it the most, taking the actions of the code-smell generation process (lower level or follower) into account. The main advantage of our bilevel formulation is that the generation of detection rules is not limited to some code-smell examples identified manually by developers that are difficult to collect, but it allows the prediction of new code-smell behaviors that are different from those in the base of examples.

We implemented our proposed bilevel approach and evaluated it on several open-source systems: JFreeChart, GanttProject, ApacheAnt, Nutch, Log4J, Lucene, Xerces-J, and Rhino. In addition, we evaluated our proposal on one industrial system provided by our industrial partner, that is, the Ford Motor Company. We found that, on average, the majority of seven types of code smells were detected with more than 86% of precision and 90% of recall. The statistical analysis of our experiments over 31 runs shows that BLOP performed significantly better than two existing search-based approaches [Boussaa et al. 2013; Kessentini et al. 2011] and a practical code-smell detection technique [Moha et al. 2010]. The software developers considered in our experiments confirm the relevance of the detected code-smells for several maintenance activities.

The primary contributions of this article can be summarized as follows.

(1) The article introduces a novel formulation of code-smell detection as a bilevel problem.
(2) The article reports the results of an empirical study with an implementation of our bilevel approach. The obtained results provide evidence to support the claim that our proposal is more efficient, on average, than existing techniques based on a benchmark of nine open-source systems and one industrial project. The article also evaluates the relevance and usefulness of the detected code smells for software engineers to improve the quality of their systems.

The remainder of this article is as follows: Section 2 presents the relevant background and the motivation for the presented work; Section 3 describes the search algorithm; an evaluation of the algorithm is explained and its results are discussed in Section 4; a small industrial case study and the relevance of detected code smells are discussed in Section 5; the different threats that affect our experimentations are described in Section 6; Section 7 is dedicated to related work. Finally, concluding remarks and future work are provided in Section 8.

2. BACKROUND

In this section, we first provide the necessary background of detecting code smells and discuss the challenges and open problems that are addressed by our proposal. Then, we describe the necessary background related to BLOP.

2.1. Code Smells

A *code smell* is defined as bad design choices that can have a negative impact on the code quality, such as maintainability, changeability, and comprehensibility, which could introduce bugs [Brito e Abreu and Melo 1996]. Code smells classify shortcomings in software that can decrease software maintainability. They are also defined as structural characteristics of software that may indicate a code or design problem that makes software hard to evolve and maintain and trigger refactoring of code [Brown et al. 1998]. Code smells are not limited to design flaws, since most of them occur in code and are not related to the original design. In fact, most code smells can emerge during the evolution of a system and represent patterns or aspects of software design that may cause problems in the further development and maintenance of the system. As stated by Fowler et al. [1999], code smells are unlikely to cause failures directly, but

may do it indirectly. In general, they make a system difficult to change, which may in turn introduce bugs. It is easier to interpret and evaluate the quality of systems by identifying code smells than the use of traditional software quality metrics. In fact, most of the definitions of code smells are based on situations that are faced daily by developers. Most of the code-smell identify locations in the code that violate object-oriented design heuristics, such as the situations described by Riel [1996] and Coad and Yourdon [1991]. The 22 code smells identified and defined informally by Fowler et al. [1999] aim to indicate software refactoring opportunities and 'give you indications that there is trouble that can be solved by a refactoring.' Zhang et al. [2011] identified in their survey the code smells that attracted more attention in current literature.

Van Emden and Moonen [2002] developed one of the first automated code-smell detection tools for Java programs. Mika and Lassenius studied the manner of how developers detect and analyze code-smells [2006]. Previous empirical studies have analyzed the impact of code smells on different software maintainability factors, including defects [Monden et al. 2002] and effort [Deligiannis et al. 2003]. In fact, software metrics (quality indicators) are sometimes difficult to interpret and suggest some actions (refactoring), as noted by Anda et al. [2007] and Ratiu et al. [2004]. Code smells are associated with a generic list of possible refactorings to improve the quality of software systems. In addition, Yamashita and Moonen [2012] show that the different types of code smells can cover most of maintainability factors [Brito e Abreu and Melo 1996]. Thus, the detection of code smells can be considered a good alternative to the traditional use of quality metrics to evaluate the quality of software products. Brown et al. [1998] define another category of code smells that are documented in the literature, and named anti-patterns.

In our experiments, we focus on the seven following code-smell types.

—*Blob*. It is found in designs where one large class monopolizes the behavior of a system (or part of it), and the other classes primarily encapsulate data.
—*Feature Envy (FE)*. It occurs when a method is more interested in the features of other classes than its own. In general, it is a method that invokes several times accessor methods of another class.
—*Data Class (DC)*. It is a class with all data and no behavior. It is a class that passively stores data.
—*Spaghetti Code (SC)*. It is a code with a complex and tangled control structure.
—*Functional Decomposition (FD)*. It occurs when a class is designed with the intent of performing a single function. This is found in a code produced by non-experienced object-oriented developers.
—*Lazy Class (LC)*. A class that is not doing enough to pay for itself.
—*Long Parameter List* (*LPL*). Methods with numerous parameters are a challenge to maintain, especially if most of them share the same data type.

We choose these code-smell types in our experiments because they are the most frequent, hard to detect, and fix based on a recent empirical study [Palomba et al. 2013], cover different maintainability factors, and also due to the availability of code-smell examples. However, the proposed approach in this article is generic and can be applied to any type of code smell.

The code-smell detection process consists of finding code fragments that violate structural or semantic properties, such as the ones related to coupling and complexity. In this setting, internal attributes used to define these properties, are captured through software metrics, and properties are expressed in terms of valid values for these metrics. This follows a long tradition of using software metrics to evaluate the quality of the design, including the detection of code smells. The most widely-used metrics are the ones defined by Chidamber and Kemerer [1994] and other studies [Brito e Abreu

Table I. List of Quality Metrics

Metrics	*Description*
Weighted Methods per Class (WMC)	WMC represents the sum of the complexities of its methods.
Response for a Class (RFC)	RFC is the number of different methods that can be executed when an object of that class receives a message.
Lack of Cohesion of Methods (LCOM)	Chidamber and Kemerer define Lack of Cohesion in Methods as the number of pairs of methods in a class that does not have at least one field in common minus the number of pairs of methods in the class that does share at least one field. When this value is negative, the metric value is set to 0.
Number of Attributes (NA)	
Attribute Hiding Factor (AH)	AH measures the invisibilities of attributes in classes. The invisibility of an attribute is the percentage of the total classes from which the attribute is not visible.
Method Hiding Factor (MH)	MH measures the invisibilities of methods in classes. The invisibility of a method is the percentage of the total classes from which the method is not visible.
Number of Lines of Code (NLC)	NLC counts the lines but excludes empty lines and comments.
Coupling Between Object classes (CBO)	CBO measures the number of classes coupled to a given class. This coupling can occur through method calls, field accesses, inheritance, arguments, return types, and exceptions.
Number of Association (NAS)	
Number of Classes (NC)	
Depth of Inheritance Tree (DIT)	DIT is defined as the maximum length from the class node to the root/parent of the class hierarchy tree and is measured by the number of ancestor classes. In cases involving multiple inheritances, the DIT is the maximum length from the node to the root of the tree.
Polymorphism Factor (PF)	PF measures the degree of method overriding in the class inheritance tree. It equals the number of actual method overrides divided by the maximum number of possible method overrides.
Attribute Inheritance Factor (AIF)	AIF is the fraction of class attributes that are inherited.
Number of Children (NOC)	NOC measures the number of immediate descendants of the class.

and Melo 1996; Brito e Abreu 1995]. In this article, we use variations of these metrics and adaptations of procedural ones as well [Abran and Nguyenkim 1993; Briand et al. 1999]. The list of metrics is described in Table I.

In the following, we introduce some issues and challenges related to the detection of code smells. Overall, there is no general consensus on how to decide if a particular design violates a quality heuristic. In fact, there is a difference between detecting symptoms and asserting that the detected situation is an actual code smell. For example, an object-oriented program with a hundred classes from which one class implements all the behavior and all the other classes are only classes with attributes and accessors. No doubt, we are in the presence of a *blob*. Unfortunately, in real-life systems, we can find many large classes, each one using some data classes and some regular classes. Deciding which classes are blob candidates heavily depends on the interpretation of each analyst. In some contexts, an apparent violation of a design principle may be consensually accepted as normal practice. For example, a "Log" class responsible for maintaining a log of events in a program, used by a large number of classes, is a common and acceptable practice. However, from a strict code-smell definition, it can be

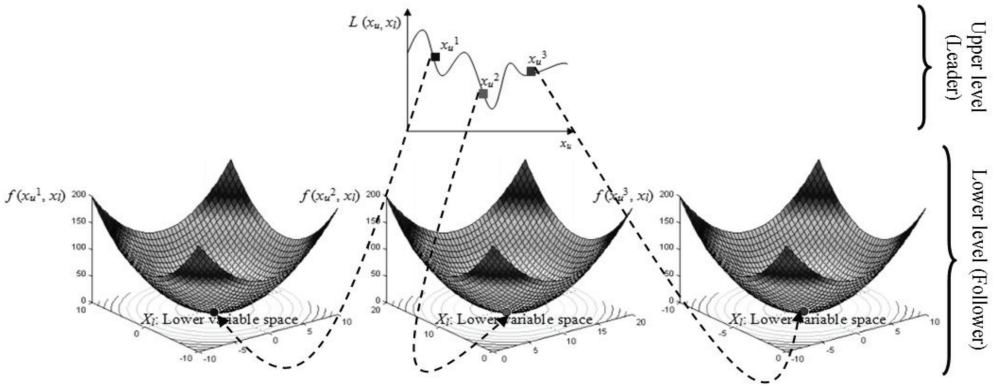


Fig. 1. Illustration of the two levels of an exemplified bilevel single-objective optimization problem.

considered as a class with an abnormally large coupling. Another issue is related to the definition of thresholds when dealing with quantitative information. For example, blob detection involves information such as class size. Although we can measure the size of a class, an appropriate threshold value is not trivial to define. A class considered large in a given program/community of users could be considered average in another. All these issues make the process of manually defining code-smell detection rules challenging. The generation of detection rules requires a huge code-smell example set to cover most of the possible bad-practice behaviors. Code smells are not usually documented by developers (unlike bugs report). Thus, it is time consuming and difficult to collect code smells and inspect manually large systems [Riel 1996]. In addition, it is challenging to ensure the diversity of the code-smell examples to cover most of the possible bad practices, then using these examples to generate good quality of detection rules.

To address these challenges, we propose considering the code-smell detection problem as a bilevel optimization problem. The principles of BLOP are described in the next section; then we present our adaptation in Section 3.

2.2. Bilevel Optimization

Most studied real-world and academic optimization problems involve a single level of optimization. However, in practice, several problems are naturally described in two levels. These are called BLOPs [Kolstad 1985]. In such problems, we find a nested optimization problem within the constraints of the outer optimization one. The outer optimization task is usually referred to as the *upper-level problem* or the *leader problem*. The nested inner optimization task is referred to as the *lower-level problem* or the *follower problem*, thereby referring to the bilevel problem as a leader-follower problem or as a Stackelberg game [Sinha et al. 2013b]. The follower problem appears as a constraint to the upper level, such that only an optimal solution to the follower optimization problem is a possible feasible candidate to the leader one. A BLOP contains two classes of variables: (1) the upper-level variables $x_u \in X_U \subset \Re^n$, and (2) the lower-level variables $x_l \in X_L \subset \Re^m$. For the follower problem, the optimization task is performed with respect to the variables x_l, and the variables x_u act as fixed *parameters*. Thus, each x_u corresponds to a different follower problem, whose optimal solution is a function of x_u and needs to be determined. All variables (x_u, x_l) are considered in the leader problem, but x_l are not changed (cf. Figure 1). In what follows, we give the formal definition of BLOP.

Definition 1. Assuming $L : \Re^n \times \Re^m \rightarrow \Re$ to be the leader problem and $f : \Re^n \times \Re^m \rightarrow \Re$ to be the follower one, analytically, a BLOP could be stated as follows:

$$\underset{x_u \in X_U, x_l \in X_L}{Min} L(x_u, x_l) \text{ subject to } \begin{cases} x_l \in ArgMin\left\{ f(x_u, x_l), g_j(x_u, x_l) \leq 0, j = 1, \ldots, J \right\} \\ G_k(x_u, x_l) \leq 0, k = 1, \ldots, K. \end{cases}$$

BLOPs are intrinsically more difficult to solve than single-level problems, so it is not surprising that most existing studies to date have tackled the simplest cases of BLOPs, that is, problems having nice properties, such as linear, quadratic, or convex objective and/or constraint functions. In particular, the most studied instance of BLOPs, for a long time, has been the linear case in which all objective functions and constraints are linear with respect to the decision variables.

Although the first works on bilevel optimization date back to the seventies, it was not until the early eighties that the usefulness of these mathematical programs in modeling hierarchical decision processes and engineering problems prompted researchers to pay close attention to BLOPs. A first bibliographical survey on the subject was written by Kolstad [1985]. BLOPs being intrinsically more difficult than single-level problems, it is not surprising that most algorithmic research to date has tackled the simplest cases of BLOPs, that is, problems having nice properties such as linear, quadratic, or convex objective and/or constraint functions [Bracken and McGill 1973; Bard 1998]. In particular, the most studied instance of BLOPs has been, for a long time, the linear case in which all objective functions and constraints are linear with respect to the decision variables [Vicente and Calamai 1994; Mathieu et al. 1994].

Existing methods to solve BLOPs could be classified into two main families: (1) classical methods and (2) evolutionary methods. The first family includes extreme point-based approaches [Candler and Townsley 1992], branch-and-bound [Bard and Falk 1982], complementary pivoting [Bialas et al. 1980], descent methods [Aiyoshi and Shimizu 1981], penalty function methods [Aiyoshi and Shimizu 1981], trust region methods [Colson et al. 2005a], etc. The main shortcoming of these methods is that they heavily depend on the mathematical characteristics of the BLOP at hand. The second family includes metaheuristic algorithms that are mainly Evolutionary Algorithms (EAs). Recently, several EAs have demonstrated their effectiveness in tackling such types of problems thanks to their insensibility to the mathematical features of the problem in addition to their ability to tackle large-size problem instances by delivering satisfactory solutions in a reasonable time. Some representative works are Sinha et al. [2013a, 2013b], Legillon et al. [2012], and Koh [2013].

To the best of our knowledge and based on recent surveys [Harman 2007; Harman et al. 2012], there is no work in the software engineering literature that considers a software engineering problem as bilevel. In the next section, we describe our adaptation of bilevel optimization to the problem of code-smell detection.

3. CODE-SMELL DETECTION AS A BILEVEL OPTIMIZATION PROBLEM

This section shows how the previously-mentioned issues can be addressed using bilevel optimization and describes the principles that underlie the proposed method for detecting code smells. We first present an overview of our bilevel code-smell detection approach, and then we describe the details of our bilevel formulation, including the adaptation of both lower and upper levels.

3.1. Approach Overview

The concept of bilevel is based on the idea that the main optimization task is usually termed as the upper-level problem, and the *nested* optimization task is referred to as the lower-level problem. The detection rules generation process has a main objective, which

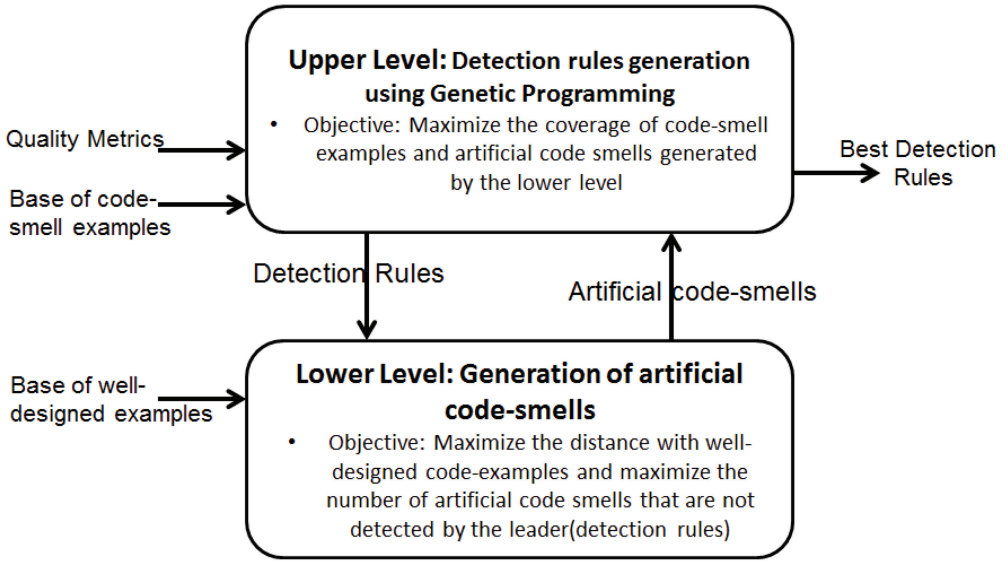


Fig. 2. Approach overview.

is the generation of detection rules that can cover, as much as possible, the code smells in the base of examples. The code-smell generation process has one objective: that is maximizing the number of generated artificial code smells that cannot be detected by the detection rules and that are dissimilar from the base of well-designed code examples. There is a hierarchy in the problem, which arises from the manner in which the two entities operate. The detection rule generation process has higher control of the situation and decides which detection rules for the code-smell generation process to operate in. Therefore, in this framework, we observe that the detection rule generation process acts as a leader (the important output of the problem), and the code-smell generation process acts as a follower.

The overall problem appears as a bilevel optimization task, where for each generated detection rule, the upper level observes how the lower level acts by generating artificial code smells that cannot be detected by the upper level (leader) rules and then chooses the best detection rules which suit it, taking the actions of the code-smell generation process (lower level or follower) into account. It should be noted that in spite of different objectives appearing in the problem, it is not possible to handle such a problem as a simple multiobjective optimization task. The reason for this is that the leader cannot evaluate any of its own strategies without knowing the strategy of the follower, which it obtains only by solving a nested optimization problem.

As described in Figure 2, the leader (upper level) uses knowledge from code-smell examples (input) to generate detection rules based on quality metrics (input). It takes as inputs a base (i.e., a set) of code-smell examples, and takes, as controlling parameters, a set of quality metrics and generates as output a set of rules. The rule generation process chooses randomly, from the provided metrics list, a combination of quality metrics (and their threshold values) to detect a specific code smell. Consequently, the solution is a set of rules that best detects the code smells of the base of examples. For example, the following rule states that a class c having more than $NAD = 10$ attributes and more than $NMD = 20$ methods is considered a blob smell:

$$\text{R1} : \text{IF } NAD(c) \geq 10 \text{ AND } NMD(c) \geq 20, \text{ THEN } Blob(c).$$

In this exemplified sample rule, the number of attributes (NAD) and the number of methods (NMD) of a class correspond to two quality metrics that are used to detect a blob. The detected code smells can be method(s) or class(es) depending on the type of the code smells to detect. Given this detection rule, it is not obvious what set of diverse code smells exist in the software. In the bilevel formulation of the code-smell detection problem, the lower-level problem allows us to find just that. An upper-level detection rules solution is evaluated based on the coverage of the base of code-smell examples (input) and also the coverage of generated "artificial" code smells by the lower-level problem. These two measures are used to be maximized by the population of detection rules solutions. The follower (lower level) uses well-designed code examples to generate "artificial" code smells based on the notion of deviation from a reference (well-designed) set of code fragments. The generation process of artificial code-smell examples is performed using a heuristic search that maximizes, on one hand, the distance between generated code-smell examples and reference code examples and, on the other hand, maximizes the number of generated examples that are not detected by the leader (detection rules).

There is no parallelism in our bilevel formulation. The upper level is executed for a number of iterations, then the lower level for another number of iterations. After that, the best solution found in the lower level will be used by the upper level to evaluate the associated solution (detection rules), and then this process is repeated several times until reaching a termination criterion (e.g., number of iterations). Thus, there is no parallelism, since both levels are dependent. In Boussaa et al. [2013], we proposed using Co-Evolutionary (Co-Evol) algorithms for code-smell detection, where the first population generates detection rules, and the second generates artificial code-smell. Both populations are executed in parallel *without hierarchy*. The problem with the Co-Evol approach is that one population may converge before the other. Contrariwise, in our bilevel approach, there is a *hierarchy* that allows avoiding the problem of premature convergence of one population over the other. Indeed, the evaluation of every detection rule solution (upper level) requires running a search algorithm to find the best undetectable artificial code smells by the upper-level solution. This concept avoids driving the search towards *uninteresting* directions. In addition, co-evolution treats the two populations independently; however, in BLOP, the evaluation of solutions in the upper level depends on the lower level (both populations cannot be executed in parallel). Furthermore, the two populations in co-evolution are considered with the same importance; however the upper level is more important than the lower level in any bilevel formulation. We will compare later in the experimentation section, with more details, the difference between our bilevel formulation for code-smell detection and our previous work based on co-evolution [Boussaa et al. 2013].

Next, we describe our adaptation of bilevel optimization to the code-smell detection problem in more detail.

3.2. Problem Formulation

The code-smells detection problem involves searching for the best metric combinations among the set of candidate ones, which constitutes a huge search space. A solution of our code-smell detection problem is a set of rules (metric combinations with their threshold values), where the goal of applying these rules is to detect code smells in a system.

Our proposed bilevel formulation of the code-smell detection problem is described in Figure 3. Consequently, we have two levels as described in the previous section. At the upper level, the objective function is formulated to maximize the coverage of code-smell examples (input) and also maximize the coverage of the generated artificial code smells at the lower level (best solution found in the lower level). Thus, the objective function

Upper level algorithm: GPSmellDetection

```
01.  Inputs: Quality metrics M, Defect example base B, Well-designed
     code example base D, Number of best upper solutions that are
     considered for lower level optimization nbs, Upper population size
     N1, Lower population size N2, Upper number of generations G1,
     Lower number of generations G2
02.  Output: Best detection rule BDR
03.  Begin
04.    P0 ← Initialization (N1, M);
05.    For each DR0 in P0 do   /*DR means Detection Rule*/
06.      BCS0 ← GASmellGeneration (DR0, D, N2, G2);   /*Call lower level
07.      DR0 ← Evaluation (DR0, B, BCS0);
08.    End For
09.    t ← 1;
10.    While (t < G1) do
11.      Qt ← Variation (Pt-1);
12.      For each DRt in Qt do   /*Evaluate each rule based on upper fitness
     function*/
13.        DRt ← UpperEvaluation (DRt, B);
14.      End For
15.      For each of the best nbs rules DRt in Qt do   /*Only nbs rules
     are used to*/
16.        BCSt ← GASmellGeneration (DRt, D, N2, G2);
17.        DRt ← EvaluationUpdate (DRt, BCSt);   /*Update based on lower
     level*/
18.      End For
19.      Ut ← Pt ∪ Qt;
20.      Pt+1 ← EvironmentalSelection (N1, Ut);
21.      t ← t+1;
22.    End While
23.    BDR ← FittestSelection (Pt);
24.  End
```

(a)

Lower level algorithm: GASmellGeneration

```
01.  Inputs: Upper level detection rule UDR, Well-designed code
     example base D, Population size N, number of generations G
02.  Output: Best artificial code-smells BCS
03.  Begin
04.    P0 ← Initialization (N, D);
05.    P0 ← Evaluation (P0, D, UDR);   /*Evaluation depends of UDR*/
06.    t ← 1;
07.    While (t < G) do
08.      Qt ← Variation (Pt-1);
09.      Qt ← Evaluation (Qt, D, UDR);   /*Evaluation depends of UDR*/
10.      Ut ← Pt ∪ Qt;
11.      Pt+1 ← EvironmentalSelection (N, Ut);
12.      t ← t+1;
13.    End While
14.    BCS ← FittestSelection (Pt);
15.  End
```

(b)

Fig. 3. Pseudocode of the bilevel adaptation for code-smell detection.

at the upper level is defined as follows:

$$Maximize \ \ f_{upperLevel} = \frac{\frac{\mathrm{Precision}(S,\ baseOfExamples) + \mathrm{Recall}(S,\ BaseOfExamples)}{2} + \frac{\#\mathrm{detectedArtificialCodeSmells}}{\#\ artificialCodeSmells}}{2}.$$

It is clear that the evaluation of solutions (detection rules) at the upper level depends on the best solutions generated by the lower level (artificial code smells). Thus, the fitness function of solutions at the upper level is calculated after the execution of the optimization algorithm in the lower level at each iteration.

At the lower level, for each solution (detection rule) of the upper level, an optimization algorithm is executed to generate the best set of artificial code smells that cannot be detected by the detection rules at the upper level. An objective function is formulated at the lower level to maximize the number of undetected artificial code smells that are generated and also maximize the distance with well-designed code examples. Formally,

$$Maximize \ f_{lower} = t + Min\left(\sum_{j=1}^{w}\sum_{k=1}^{l}\left|M_k(cArtificialCS) - M_k(c\mathrm{Re}ferenceCode)\right|\right),$$

where w is the number of code elements (e.g., classes) in the reference code, l is the number of structural metrics used to compare between artificial code smells and the well-designed code examples, M is a structural metric (such as number of methods, number of attributes, etc.), and t is the number of artificial code smells uncovered by the detection rule solution defined at the upper level.

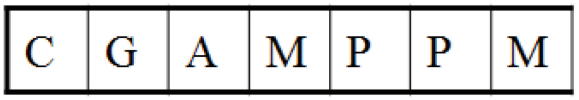
3.3. Solution Approach

The solution approach proposed in this article lies within the SBSE field. As noted by Harman et al. [2012], a generic algorithm like bilevel optimization cannot be used 'out of the box'—it is necessary to define problem-specific genetic operators to obtain the best performance. To adapt bilevel optimization to our code-smell detection problem, the required steps are to create for both levels (algorithms): (1) solution representation, (2) solution variation, and (3) solution evaluation. We examine each of these next.

3.3.1. Solution Representation. One key issue when applying a search-based technique is to find a suitable mapping between the problem to be solved and the techniques to be used, that is, detecting code smells.

For the *upper-level* optimization problem, a genetic programming (GP) algorithm is used [Goldberg 1989]. In GP, a solution is composed of terminals and functions. Therefore, when applying GP to solve a specific problem, they should be carefully selected and designed to satisfy the requirements of the current problem. After evaluating many parameters related to the code-smell detection problem, the terminal set and the function set are decided as follows. The terminals correspond to different quality metrics with their threshold values (constant values). The functions that can be used between these metrics are Union (OR) and Intersection (AND). More formally, each candidate solution S in this problem is a sequence of detection rules, where each rule is represented by a binary tree such that the following hold.

(1) Each leaf-node (Terminal) L belongs to the set of metrics (such as number of methods, number of attributes, etc.) and their corresponding thresholds generated randomly.
(2) Each internal-node (Functions) N belongs to the Connective (logic operators) set C = {AND, OR}.



Class(C12,public);
Generalisation(C12,C9);
Attribute(C12, a254,long,static);
Attribute(C12, a54, short,static);
Method(C12, m154, void,Y, public);
Parameter(C12, m154, p47,short);
Parameter(RangeExceptionImpl,RangeExceptionImpl,message, String);
Method(C12, m129, void,Y,private);

Fig. 4. Solution representation: vector (GA) to generate artificial code smell.

The set of candidate's solutions (rules) corresponds to a logic program that is represented as a forest of AND-OR trees.

For the *lower-level* optimization problem, a genetic algorithm (GA) is used to generate artificial code smells. The generated artificial code fragments are composed of code elements. Thus, they are represented as a vector, where each dimension is a code element. We represent these elements as sets of predicates. Each predicate type corresponds to a construct type of an object-oriented system: *class (C), attribute (A), method (M), parameter (P), generalization (G), and method invocation relationship between classes (R)*. For example, the sequence of predicates *CGAMPPM* corresponds to a class with a generalization link, containing two attributes and two methods (Figure 4). The first method has two parameters. Predicates include details about the associated constructs (visibility, types, etc.). These details (thereafter called parameters) determine ways a code fragment can deviate from a notion of normality. The sequence of predicates must follow the specified order of predicate types (class, attribute, method, generalization, association, etc.) to ease the comparison between predicate sequences and then reduce the computational complexity. When several predicates of the same type exist, we order them according to their parameters.

To generate an initial population for both GP and GA, we start by defining the maximum tree/vector length (max number of metrics/code-elements per solution). The tree/vector length is proportional to the number of metrics/code-elements to use for code-smell detection. Sometimes, a high tree/vector length does not mean that the results are more precise. These parameters can be specified either by the user or chosen randomly. Figure 4 shows an example of a generated code smell composed of one class, one generalization link, two attributes, two methods, and two parameters. The parameters of each predicate contain information generated randomly describing each code element (type, visibility, etc.).

3.3.2. Solution Evaluation. The encoding of an individual should be formalized as a mathematical function called the "fitness function." The fitness function quantifies the quality of the proposed detection rules and generated artificial code smells. The goal is to define efficient and simple fitness functions in order to reduce the computational cost. For our GP adaptation (upper level), we used the fitness function f_{upper} defined in the previous section to evaluate detection-rules solutions. For the GA adaptation (lower level), we used the fitness function f_{lower} defined in the previous section to evaluate generated artificial code smells.

3.3.3. Evolutionary Operators: Selection. One of the most important steps in any evolutionary algorithm (EA) is the selection phase. There are two selection phases in EAs: (1) parent selection (also named mating pool selection), and (2) environmental selection (also named replacement). In this work, we use an elitist scheme for both selection phases with the aim to (1) exploit good genes of fittest solutions, and (2) preserve the best solutions along the evolutionary process. The two selections schemes are described as follows. Regarding the parent selection, once the population individuals are evaluated, we select the $|P|/2$ best individuals of the population P to fulfill the mating pool, whose size is equal to $|P|/2$. This allows exploiting the past experience of the EA in discovering the best chromosomes' genes. Once this step is performed, we apply genetic operators (crossover and mutation) to produce the offspring population Q, which has the same size as P ($|P| = |Q|$). Since crossover and mutation are stochastic operators, some offspring individuals can be worse than some of P individuals. In order to ensure elitism, we merge both populations P and Q into U (with $|U| = |P| + |Q| = 2|P|$), and then the population P for the next generation is composed of the $|P|$ fittest individuals from U. By doing this, we ensure that we encourage the survival of better solutions. We can say that this environmental selection is elitist, which is a desired property in modern EAs [Sinha et al. 2013a]. We use this elitist operation in both upper- and level-level EAs.

3.3.4. Evolutionary Operators: Mutation. For GP (upper level), the mutation operator can be applied to a function node or to a terminal node. It starts by randomly selecting a node in the tree. Then, if the selected node is a terminal (quality metric), it is replaced by another terminal (metric or another threshold value); if it is a function (AND-OR), it is replaced by a new function; and if tree mutation is to be carried out, the node and its subtree are replaced by a new randomly generated subtree.

For GA (lower level), the mutation operator consists of randomly changing a predicate (code element) in the generated predicates.

3.3.5. Evolutionary Operators: Crossover. For GP (upper level), two parent individuals are selected, and a subtree is picked on each one. Then crossover swaps the nodes and their relative subtrees from one parent to the other. This operator must ensure the respect of the depth limits. The crossover operator can be applied with only parents having the same rule category (code-smell type to detect). Each child thus combines information from both parents. In any given generation, a variant will be the parent in at most one crossover operation.

For GA (lower level), the crossover operator allows creating two offspring o_1 and o_2 from the two selected parents p_1 and p_2, as follows.

(1) A random position k is selected in the predicate sequences.
(2) The first k elements of p_1 become the first k elements of o_1. Similarly, the first k elements of p_2 become the first k elements of o_2.
(3) The remaining elements of, respectively, p_1 and p_2 are added as second parts of, respectively, o_2 and o_1 (hence having a crossing-over operation between parents).

For instance, if $k = 3$ and $p_1 =$ CAMMPPP and $p_2 =$ CMPRMPP, then $o_1 =$ CAMRMPP and $o_2 =$ CMPMPPP.

4. VALIDATION

In order to evaluate our approach for detecting code smells using the proposed bilevel optimization (BLOP) approach, we conducted a set of experiments based on different versions of open-source systems: JFreeChart, GanttProject, ApacheAnt, Nutch, Log4J, Lucene, Xerces-J and Rhino. Each experiment is repeated 31 times, and the obtained results are subsequently statistically analyzed with the aim to compare our bilevel

Table II. Software Studied in our Experiments

Systems	*Release*	*#Classes*	*#Smells*	*KLOC*
JFreeChart	v1.0.9	521	82	170
GanttProject	v1.10.2	245	67	41
ApacheAnt	v1.5.2	1,024	163	255
ApacheAnt	v1.7.0	1,839	159	327
Nutch	v1.1	207	72	39
Log4J	v1.2.1	189	64	31
Lucene	v1.4.3	154	37	33
Xerces-J	V2.7.0	991	106	238
Rhino	v1.7R1	305	78	57

proposal with a variety of existing code-smell detection approaches. In this section, we first present our research questions and then describe and discuss the obtained results.

4.1. Research Questions

We defined five research questions that address the applicability, performance comparison with existing code-smell detection approaches, and the scalability of our bilevel code-smell detection approach. The five research questions are as follows.

RQ1: Search Validation. To validate the problem formulation of our approach, we compared our BLOP formulation with Random Search (applied to both levels). If Random Search outperforms a guided search method, we can thus conclude that our problem formulation is not adequate. Since outperforming a random search is not sufficient, the next four questions are related to performance and scalability of BLOP, and a comparison with the state-of-the-art code-smell detection approaches.

RQ2: How Does BLOP Perform in Detecting Different Types of Code Smells? It is important to quantitatively assess the completeness and correctness of our code-smell detection approach.

RQ3.1: How Does BLOP Perform Compared to Existing Search-Based Code-Smell Detection Algorithms? Our proposal is the first work that treats a software engineering problem as a bilevel problem. A comparison with existing search-based code-smell detection approaches is helpful to evaluate the benefits of the use of our bilevel approach in the context of code-smell detection.

RQ3.2: How Does BLOP Perform Compared to the Existing Code-Smell Detection Approaches Not Based on the Use of Metaheuristic Search? While it is very interesting to show that our proposal outperforms existing search-based code-smell detection approaches, developers will consider our approach useful if it can outperform other existing tools that are not based on optimization techniques.

RQ4: How Does Our Bilevel Formulation Scale? There is a cost in solving every lower-level optimization problem in each iteration. An evaluation of the execution time is required to discuss the ability of our approach to detect code-smells within a reasonable timeframe.

4.2. Software Projects Studied

In our experiments, we used a set of well-known and well-commented open-source Java projects. We applied our approach to nine open-source Java projects. Table II presents the list and some relevant statistics of the software systems for our code-smell detection purpose.

JFreeChart is a powerful and flexible Java library for generating charts. GanttProject is a cross-platform tool for project scheduling. ApacheAnt is a build tool and library

specifically conceived for Java applications. Nutch is an open-source Java implementation of a search engine. Log4j is a Java-based logging utility. Lucene is a free/open-source information retrieval software library. Xerces-J is a family of software packages for parsing XML. Finally, Rhino is a JavaScript interpreter and compiler written in Java and developed for the Mozilla/Firefox browser. Table II provides some descriptive statistics about these nine programs. We selected these systems for our validation because they range from different sizes that have been actively developed over the past 10 years, and include a large number of code smells. In addition, these systems are well studied in the literature, and their code smells have been detected and analyzed manually.

In the nine studied open-source systems, the seven code-smell types, described in Section 2.1, were identified manually. Moha et al. [2010] asked three groups of students to analyze the libraries to tag instances of specific code smells to validate their detection technique, DECOR. For replication purposes, they provided a corpus of describing instances of different code smells. In our previous work [Ouni et al. 2012; Harman 2007], we asked eighteen graduate students and four software engineers to extend the existing corpus proposed by Moha et al. To calculate the recall, the different participants analyzed different quality metrics based on the definition of code smells. Recently, Palomba et al. [2013] proposed another corpus including several new code-smell types. The selected types of code smells in our validation are not similar (unilateral), diversified and cover different high-level design quality (maintainability) attributes such as reusability, flexibility, understandability, functionality, extendibility, and effectiveness [Bansiya and Davis 2002].

4.3. Evaluation Metrics Used

To assess the accuracy of our approach, we compute two measures, *precision (PR)* and *recall (Rc)*, originally stemming from the area of information retrieval. When applying precision and recall in the context of our study, the precision denotes the fraction of correctly detected code smells among the set of all detected code smells. The recall indicates the fraction of correctly detected code smells among the set of all manually identified code smells (i.e., how many code smells are undetected). In general, the precision denotes the probability that a detected code smell is correct, and the recall is the probability that an expected code smell is detected. Thus, both values range between 0 and 1, where a higher value is better than a lower one. We performed a nine-fold cross-validation, thus removing the expected code smells for detecting in the system from the base of code-smell examples when executing our BLOP algorithm, and then precision and recall scores are calculated automatically based on a comparison between the detected code smells and the expected ones. Thus, one project is evaluated by using the remaining systems as a base of code-smell examples to generate detection rules. We also use another measure *computational time* (*CT*) to evaluate the execution time required by our proposal to generate optimal detection rules.

4.4. Inferential Statistical Test Methods Used

Since metaheuristic algorithms are stochastic optimizers, they can provide different results for the same problem instance from one run to another. For this reason, our experimental study is performed based on 31 independent simulation runs for each problem instance, and the obtained results are statistically analyzed by using the Wilcoxon rank sum test [Arcuri and Fraser 2013] with a 99% confidence level ($\alpha = 1\%$). The Wilcoxon signed-rank test is a nonparametric statistical hypothesis test used when comparing two related samples to verify whether their population mean-ranks differ or not. The latter verifies the null hypothesis H_0 that the obtained results of two algorithms are samples from continuous distributions with equal medians, as against

the alternative that they are not, H_1. The p-value of the Wilcoxon test corresponds to the probability of rejecting the null hypothesis H_0 while it is true (type I error). A p-value that is less than or equal to α (≤ 0.01) means that we accept H_1, and we reject H_0. However, a p-value that is strictly greater than α (> 0.01) means the opposite. In this way, we could decide whether the outperformance of BLOP over one of each of the other detection algorithms (or the opposite) is statistically significant or just a random result.

To answer the first research question RQ1, an algorithm was implemented, where at each iteration, rules were randomly generated in the upper level, and artificial code smells were randomly created. The obtained best detection rule solution was compared for statistically significant differences with BLOP using *PR, Rc*, and *CT*.

To answer RQ2, we used the open-source systems described in Section 4.2 and calculated precision and recall scores for the different types of code smells. To answer RQ3.1, we compared BLOP with two existing search-based code-smell detection approaches—GP [Kessentini et al. 2011] and a co-evolutionary approach [Boussaa et al. 2013]. Kessentini et al. used genetic programming (GP) to generate detection rules from manually collected code-smell examples which correspond to the upper level of our approach (without lower level). Thus, the generation of artificial code smells is not considered, only the coverage of manually identified examples. Boussaa et al. proposed the use of co-evolutionary (Co-Evol) algorithms for code-smell detection, where two populations were evolved in parallel. The first population generates detection rules, and the second population generates artificial code-smell examples. Both populations are executed in parallel without hierarchy. Thus, the second-population solutions are independent of the solutions in the first population, which is one of the main differences with our bilevel extension. We considered the three metrics *PR, Rc*, and *CT* to compare bilevel with these search-based techniques based on 31 independent executions. To answer RQ3.2, we compared our results with DECOR [Moha et al. 2010]. Moha et al. started by describing code-smell symptoms using a domain-specific language (DSL) for their approach called DECOR. They proposed a consistent vocabulary and DSL to specify anti-patterns based on the review of existing work on design code smells found in the literature. To describe code-smell symptoms, different notions are involved, such as class roles and structures. Symptom descriptions are later mapped to detection algorithms based on a set of rules. We compared the results of this tool with BLOP using *PR* and *Rc* metrics.

To answer the last question RQ4, we evaluated the execution time *CT* required by our BLOP proposal based on different scenarios (parameters setting) on a large-scale system.

4.5. Parameter Tuning

An often-omitted aspect in metaheuristic search is the tuning of algorithm parameters [Bialas et al. 2010]. In fact, parameter settings significantly influence the performance of a search algorithm on a particular problem. For this reason, for each search algorithm and each system (cf. Table III), we performed a set of experiments using several population sizes: 10, 20, 30, 40, and 50. The stopping criterion was set to 750,000 fitness evaluations for all algorithms in order to ensure fairness of comparison. We used a high number of evaluations as a stopping criterion, since our bilevel approach involves two levels of optimization. Each algorithm was executed 31 times with each configuration, and then comparison between the configurations was performed based on precision and recall using the Wilcoxon test. Table III reports the best configuration obtained for each couple (algorithm, system). Since the replication of the bilevel experiment for 31 times is computationally expensive, the parameter setting experiments are performed

Table III. Best Population Size Configurations

System	Release	BLOP	CO-EVOL	GP
JFreeChart	v1.0.9	30	30	40
GanttProject	v1.10.2	30	50	30
ApacheAnt	v1.5.2	30	30	50
ApacheAnt	v1.7.0	30	30	30
Nutch	v1.1	30	40	30
Log4J	v1.2.1	30	30	50
Lucene	v1.4.3	30	40	50
Xerces-J	V2.7.0	30	40	40
Rhino	v1.7R1	30	30	30

on a cluster of 30 machines. In this way, each 30 experiments are performed in parallel with a termination criterion of 750,000 evaluations.

The other parameters' values were fixed by trial and error and are as follows: (1) crossover probability = 0.8; mutation probability = 0.5, where the probability of gene modification is 0.3; stopping criterion = 750,000 fitness evaluations. For our bilevel approach, both lower-level and upper-level EAs are run, each with a population of 30 individuals and 50 generations. The following reasons can explain the use of reduced population size at both levels. According to our formulation, detection rules are evaluated at the upper level based not only on its performance with respect to the upper fitness function but also on its performance on detecting associated generated code smells by the lower level. In this way, the lower level helps the upper in (1) discovering uninteresting upper search directions that should be ignored, and (2) favoring interesting ones; thereby we reduce the number of required evaluations at the upper level. This fact explains the reduced population size (30 individuals) at the upper level. For the lower level, we also used a reduced population size (30 individuals), since our goal is to have an idea about the performance of the detection rule at the lower level. We note that recent studies are based on the use of local search in order to predict the behavior of upper solutions at a lower level with an accepted computational cost (cf. Sinha et al. 2013a).

It should be noted that the lower-level routine is not called for all upper-level population members. To control the high computational cost of our bilevel approach, only *nbs*% of the best upper-level population members are allowed to call the lower-level optimization algorithm. Based on a parametric study, the value of 10% for *nbs* is found to be adequate empirically in our experiments. The *nbs* parametric study will be discussed later. For our experiment, we generated up to 125 artificial code smells from deviation with JHotDraw (about a quarter of the number of examples). JHotdraw was chosen as an example of reference code because it contains very few known code smells. In fact, previous work [Boussaa et al. 2013] could not find any blob in JHotdraw. In our experiments, we used all the classes of JHotdraw as our example set of well-designed code.

4.6. Results

This section describes and discusses the results obtained for the different research questions of Section 4.1.

4.6.1. Results for RQ1. Concerning RQ1, Table IV confirms that BLOP is better than random search based on the two metrics *PR* and *Rc* on all nine systems. The Wilcoxon rank sum test showed that in 31 runs, BLOP results were significantly better than random search.

Table IV. Significantly Best Algorithm Among Random Search, BLOP, GP, and Co-Evol over 31 Independent Runs

System	Release	Precision	Recall
JFreeChart	v1.0.9	BLOP	BLOP
GanttProject	v1.10.2	BLOP	BLOP
ApacheAnt	v1.5.2	BLOP	BLOP
ApacheAnt	v1.7.0	BLOP	BLOP
Nutch	v1.1	BLOP	BLOP
Log4J	v1.2.1	No. Sign.	No. Sign.
Lucene	v1.4.3	No. Sign.	No. Sign.
Xerces-J	V2.7.0	BLOP	BLOP
Rhino	v1.7R1	BLOP	BLOP

Note: "No. Sign." means no method is significantly better than another.

Table V. Median PR and Rc Values on 31 Runs for BLOP, Random Search (RS), GP 0, and Co-Evol 0

System	*PR-BLOP*	*PR-GP*	*PR-Co-Evol*	*PR-RS*	*Rc-BLOP*	*Rc-GP*	*Rc-Co-Evol*	*Rc-RS*
JFreeChart	89% (77/86)	78% (71/92)	84% (74/89)	26% (34/129)	93% (77/82)	86% (71/82)	90% (74/82)	41% (34/82)
GanttProject	88% (62/71)	80% (57/73)	82% (58/71)	28% (29/106)	89% (62/67)	83% (57/67)	85% (58/67)	43% (29/67)
ApacheAnt v1.5.2	90% (152/169)	84% (146/174)	86% (148/171)	26% (51/189)	93% (152/163)	89% (146/163)	90% (148/163)	31% (51/163)
ApacheAnt v 1.7.0	91% (149/164)	82% (142/173)	85% (144/169)	28% (54/184)	94% (149/159)	90% (142/159)	91% (144/159)	33% (54/159)
Nutch	89% (67/76)	73% (64/88)	76% (65/86)	34% (37/131)	92% (67/72)	89% (64/72)	90% (65/72)	51% (37/72)
Log4J	89% (59/67)	71% (52/74)	77% (54/71)	32% (34/127)	91% (59/64)	81% (52/64)	85% (54/64)	53% (34/64)
Lucene	91% (35/39)	70% (31/42)	79% (33/42)	12% (11/88)	95% (35/37)	84% (31/37)	89% (33/37)	29% (11/37)
Xerces-J	91% (101/111)	75% (94/126)	80% (96/119)	17% (31/179)	95% (101/106)	88% (94/106)	91% (96/106)	29% (31/106)
Rhino	90% (75/84)	74% (69/93)	79% (71/89)	14% (23/167)	95% (75/78)	87% (69/78)	91% (71/78)	29% (23/78)

Note: The results were statistically significant on 31 independent runs using the Wilcoxon rank sum test with a 99% confidence level ($\alpha < 1\%$).

Table V also describes the outperformance of BLOP against random search. The average precision and recall values of random search on the nine systems are lower than 25%. This can be explained by the huge search space to explore at both levels to generate detection rules and artificial code smells. We conclude that there is empirical evidence that our bilevel formulation surpasses the performance of random search, thus our formulation is adequate (this answers RQ1).

4.6.2. Results for RQ2. In this section, we evaluate the performance of our BLOP adaptation on the detection of seven different types of code smells. Table V summarizes our findings. The expected code smells were detected with an average of more than 90% of precision and recall on the nine open-source systems. The highest precision was found in JDI-Ford, Xerces-J, and Lucene, where 91% of code smells were detected. The lowest precision was found in GanttProject with 88% of detected code smells. This can confirm that the list of returned code smells did not contain high numbers of false positives, thus the developer will not waste a lot of their time on manual inspections. We found similar facts when analyzing the recall scores of BLOP on the different system, where an average of more than 90% of expected code smells was detected. The highest and

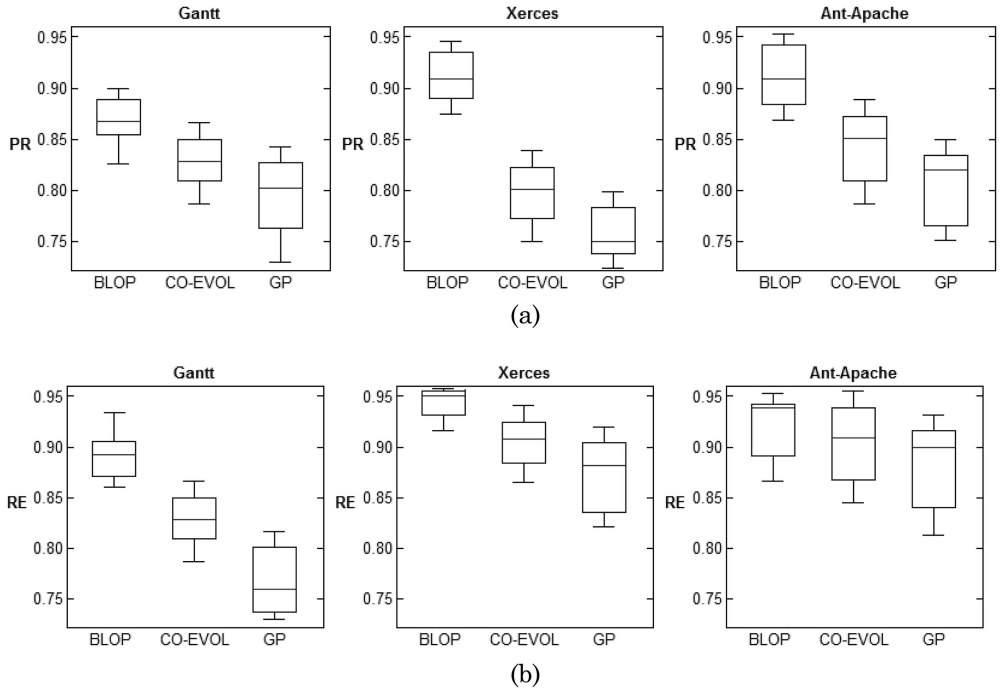


Fig. 5. Box plots on three different systems (Gantt: small, Xerces: medium, Ant-Apache 1.7.0: large) of (a) precision values, and (b) recall values.

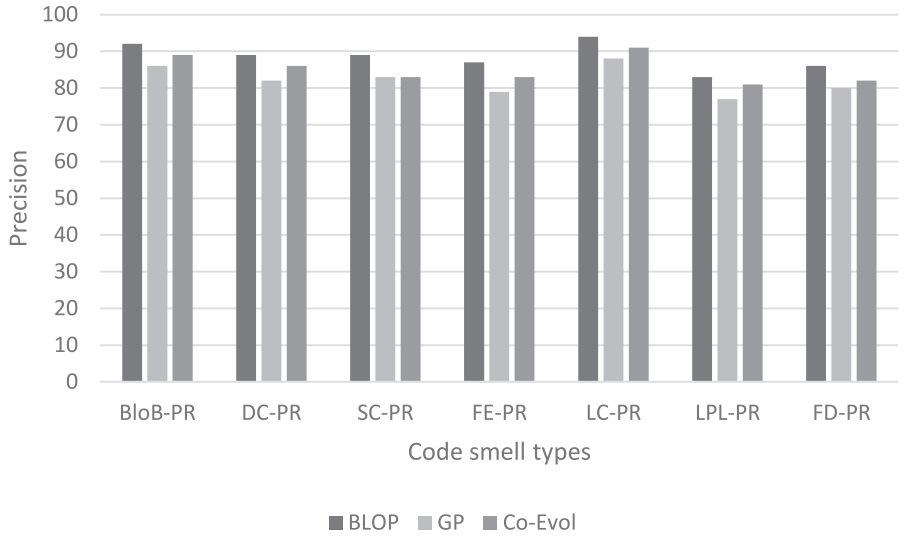
lowest recall scores were, respectively, 95% (Rhino) and 89% (GanttProject). An interesting observation that the performance of bilevel in terms of *PR* and *Rc* is independent of the size and number of code smells in the system to analyze. The *PR* and *Rc* scores of ApacheAnt are among the highest ones, with more than 90% better than the detection results of a smaller system, such as GanttProject. A more qualitative evaluation is presented in Figure 5, illustrating the box plots obtained for the *PR* and *Rc* metrics on three different projects: GanttProject (small), Xerces-J (medium), and Ant-Apache (large). We see that for almost all problems, the distributions of the *PR* and *Rc* values for BLOP are the best ones.

We noticed that our technique does not have a bias towards the detection of specific code-smell types. As described in Figure 6, in all systems, we had an almost equal distribution of each code-smell type. Having a relatively good distribution of code smells is useful for a quality engineer. Overall, all the seven code-smell types are detected with good precision and recall scores in the different systems (more than 80%).

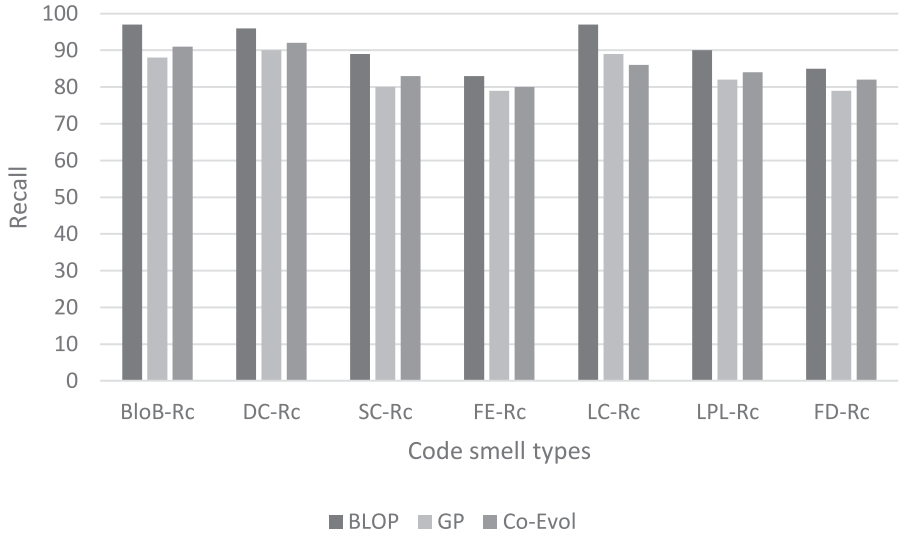
This ability to identify different types of code smell underlines a key strength to our approach. Most other existing tools and techniques rely heavily on the notion of size to detect code smells. This is reasonable considering that some code smells, like the blob, are associated with the notion of size. For code smells like FDs, however, the notion of size is less important, and this makes this type of anomaly hard to detect using structural information.

To conclude, our BLOP approach detects well all seven types of considered code smells (RQ2).

4.6.3. Results for RQ3. In this section, we compare our BLOP adaptation to the current, state-of-the-art code-smell detection approaches. To answer RQ3.1, we compared BLOP



(a)



(b)

Fig. 6. Median *PR* (a) and *Rc* (b) scores for every code-smell type over 31 runs on the different nine open-source systems.

to two other existing search-based techniques: GP [Kessentini et al. 2011] and Co-Evol [Boussaa et al. 2013]. Table V shows the overview of the results of the significance tests comparison between all these algorithms. It is clear that BLOP outperforms GP and Co-Evol in 100% of the cases in terms of precision (*PR*) and recall (*Rc*). However, as will be discussed in RQ4, the execution time (*CT*) of our BLOP algorithm is much higher than GP and Co-Evol. In addition, the improvements of *PR* and *Rc* scores are significant using BLOP comparing to GP and Co-Evol. A more qualitative evaluation is presented

Table VI. Adjusted p-Values of Comparisons Related to Table V

	BLOP	GP	Co-Evol
JFreeChart	0.00353 (GP) 0.00463 (Co-Evol) 0.00157 (RS)	0.00379 (Co-Evol) 0.00272 (RS)	0.00377 (RS)
GanttProject	0.00116 (GP) 0.00249 (Co-Evol) 0.00039 (RS)	0.00296 (Co-Evol) 0.00128 (RS)	0.00138 (RS)
ApacheAnt v1.5.2	0.00273 (GP) 0.00479 (Co-Evol) 0.00282 (RS)	0.00386 (Co-Evol) 0.00053 (RS)	0.00240 (RS)
ApacheAnt v 1.7.0	0.00179 (GP) 0.00485 (Co-Evol) 0.00079 (RS)	0.00216 (Co-Evol) 0.00138 (RS)	0.00228 (RS)
Nutch	0.00226 (GP) 0.00391 (Co-Evol) 0.00111 (RS)	0.00431 (Co-Evol) 0.00294 (RS)	0.00321 (RS)
Log4J	0.00243 (GP) 0.00405 (Co-Evol) 0.00071 (RS)	0.00412 (Co-Evol) 0.00147 (RS)	0.00195 (RS)
Lucene	0.00211 (GP) 0.00458 (Co-Evol) 0.00196 (RS)	0.00259 (Co-Evol) 0.00175 (RS)	0.00267 (RS)
Xerces-J	0.00480 (GP) 0.00328 (Co-Evol) 0.00017 (RS)	0.00217 (Co-Evol) 0.00119 (RS)	0.00226 (RS)
Rhino	0.00325 (GP) 0.00467 (Co-Evol) 0.00239 (RS)	0.00391 (Co-Evol) 0.00183 (RS)	0.00249 (RS)

in Figure 5 illustrating the box plots obtained for the *PR* and *Rc* metrics on three different projects: GanttProject (small), Xerces-J (medium), and Ant-Apache (large). This is clear from the box plots presented in Figure 5 that the outperformance of BLOP compared to GP and Co-Evol in all three different systems based on the statistical analysis of 31 independent runs. For GP, this can be explained by the fact that the use of manually identified code-smell examples is not enough to cover all the possible bad-practice behaviors, since it is a fastidious task to collect them (most of the code smells are not well documented, unlike bugs report). However, our BLOP algorithm can generate artificial code smells based on a deviation with good practices in addition to the coverage of code-smell examples. For Co-Evol, the two populations are executed in parallel, and the problem is that there is no dependency between both populations (unlike BLOP that creates a hierarchy between two levels), thus one population can converge before the second one.

The statistical tests are based on multiple pairwise comparisons using the Wilcoxon test. Thus, we have to adjust the p-values. To achieve this task, we used the Holm method that is reported to be more accurate than the Bonferroni one [Holm 1979]. Table VI presents these adjusted p-values, confirming that the results are statistically significant with 99% confidence level ($\alpha = 1\%$).

We found that the main reason explaining the outperformance of BLOP against Co-Evol is the diversity/quality of the generated artificial code smells. In fact, the lower level of our BLOP formulation generates artificial code-smell examples for every good solution (detection rules) in the upper level, then these examples are used to evaluate the solutions in the upper level. Thus, the generated artificial code-smell examples depend on the associated solution (rules) in the upper level. However, both populations, in Co-Evol, are executed in parallel and independently (without the hierarchy of BLOP).

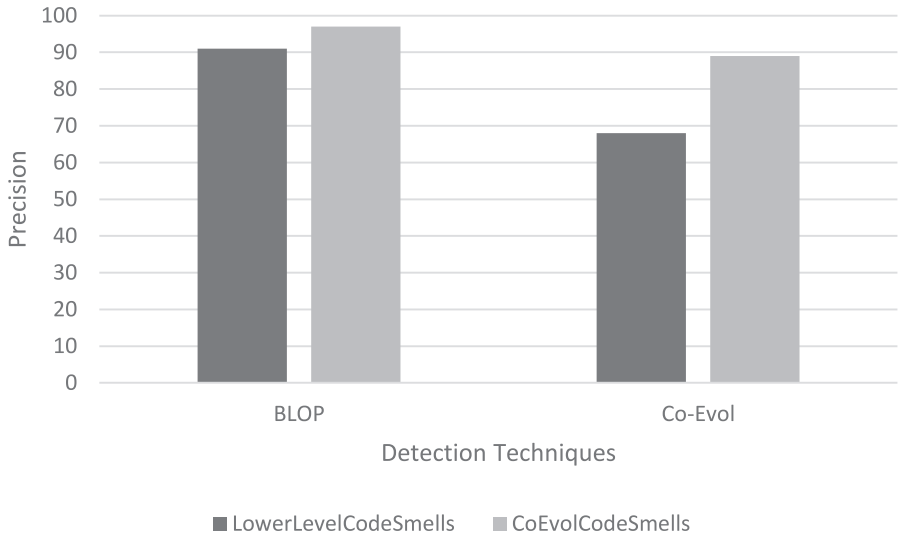


Fig. 7. The median precision score of detected artificial code smells by BLOP and Co-Evol.

Thus, the generated artificial code smells by the second population can be evaluated, at every iteration, by the best solution (detection rule) of the first population that is not "mature" enough. In fact, one population in Co-Evol can converge before the other, however, this is addressed by BLOP, because the two levels are executed in sequence after a number of iterations. We executed the best solution (code-smell detection rules) of BLOP to detect code smells on the best set of artificial code smells generated Co-Evol and vice-versa. As described in Figure 7, it is clear that the rules generated by BLOP can detect most of the artificial code-smell examples generated by Co-Evol with a precision of more than 90%, however, Co-Evol detects only around 70% of code smells generated by the lower level of BLOP. The quality of generated rules depends heavily on the diversity of the artificial code smells. Thus, the main reason explaining the outperformance of BLOP is the hierarchy that exists between the two levels, allowing the lower level to generate intelligently the artificial code smells that can improve the quality of detection rules.

We analyzed the results of our comparison between the different search techniques based on a final target fitness function value and not on a maximum number of iterations. Due to the space limitation, we present in Figure 8 the results of the comparison based on precision and recall only on Xerces-J, but similar facts were found on the remaining systems. The four different targeted fitness functions value (0.7, 0.75, 0.8, 0.85) confirms the outperformance of BLOP compared to the other search techniques in terms of precision and recall. An interesting observation is the fact that the fitness function values are correlated with the precision and recall scores. An improvement of the fitness function value leads to better precision and recall scores. This is can be a good indication about whether our fitness functions are well formulated and adapted to the code-smell detection problem. We also noticed that all the targeted fitness functions values are reached by all the algorithms with a lower number of evaluations than 750,000, which the termination criterion selected in our experiments.

One of the advantages of using our BLOP adaptation is that the developers do not need to provide a huge set of code-smell examples to generate the detection rules. In fact, the lower-level optimization can generate examples of code smells that are used to evaluate the detection rules at the upper level. Figure 9 shows that BLOP requires

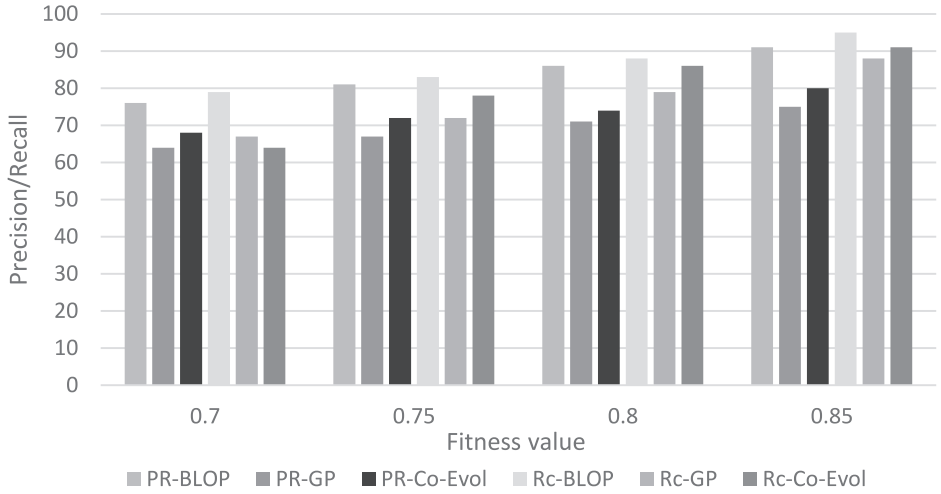


Fig. 8. The median precision and recall scores of detected code smells by BLOP, GP, and Co-Evol on Xerces-J, based on target final fitness function values.

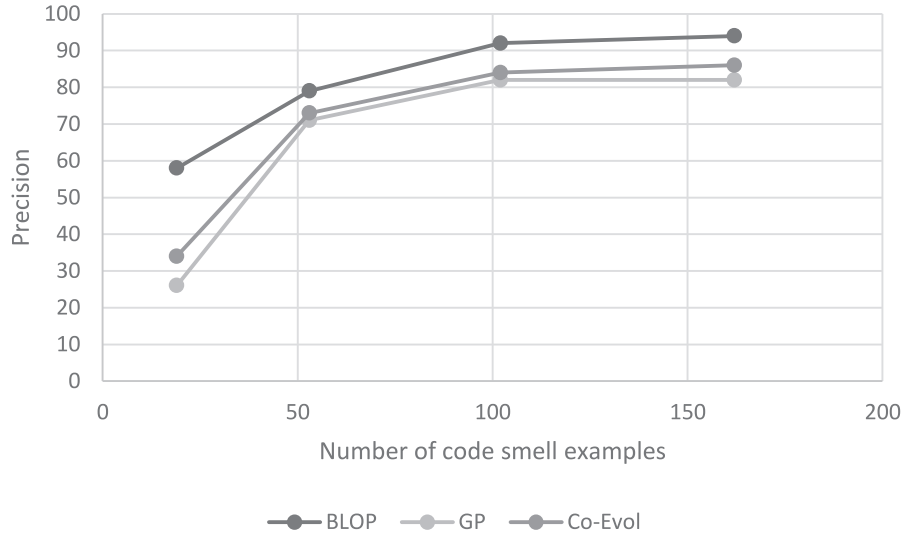


Fig. 9. The impact of the number of code-smell examples on the quality of the results (PR on Xerces-J).

a low number of manually identified code smells to provide good detection rules with reasonable precision and recall scores. GP and Co-Evol require a higher number of code-smell examples than BLOP to generate good code-smell detection rules. In addition, the reliability of the proposed BLOP approach requires an example set of good code and code-smell examples. In our study, we showed that by using JHotdraw directly, without any adaptation, the BLOP method can be used out of the box, and this will produce good detection results for the detection of code smells for all the studied systems. In an industrial setting, we could expect a company to start with JHotDraw and gradually transform its set of good code examples to include context-specific data. This might be essential if we consider that different languages and software infrastructures have different best/worst practices.

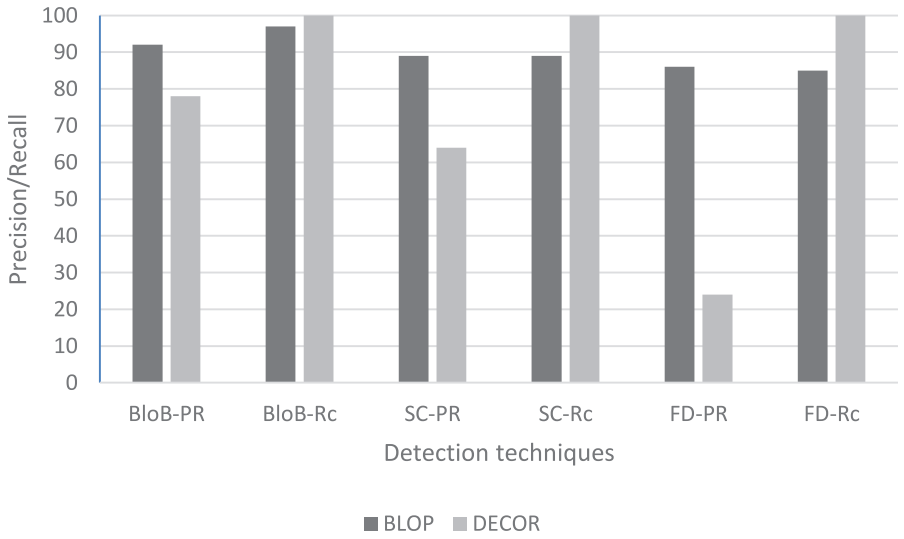


Fig. 10. The median precision and recall scores of BLOP and DECOR obtained on GanttProject, Nutch, Log4J, Lucene, and Xerces-J based on three code-smell types (Blob, SC, and FD).

In conclusion, we answer RQ3.1 by concluding that the results in our experiments confirm that our proposed BLOP is adequate, and it outperforms two existing search-based code-smell detection work—GP [Kessentini et al. 2011] and co-evolutionary GA [Boussaa et al. 2013].

Since it is not sufficient to compare our proposal with only search-based work, we compared the performance of BLOP with DECOR [Moha et al. 2010]. DECOR was mainly evaluated based on three types of code smells using metrics-based rules that are identified manually: blob, functional decomposition, and spaghetti code. The results of the execution of DECOR are only available for the following systems: GanttProject, Nutch, Log4J, Lucene, and Xerces-J. Figure 10 summarizes the results of the precision and recall obtained on these five systems. The recall for DECOR in all the systems is 100%, signifying that it is better than BLOP; however the precision scores are lower than our proposal on all systems. In fact, the higher recall achieved by DECOR can be easily explained by the use of relatively permissive constraints. For example, the average precision to detect functional decompositions using DECOR is lower than 25%, however, we can detect this type of code smell with more than 80% precision. The same observation for the spaghetti code (SC): BLOP can detect SC with more than 85% in terms of precision, however, DECOR detected the same type of code smell with a precision lower than 65%. This can be explained by the high calibration effort required to manually define the detection rules in DECOR for some specific code-smell types and the ambiguities related to the selection of the best metrics set. The recall of BLOP is lower than DECOR, on average, but it is an acceptable recall average, which is higher than 80%. To conclude, our BLOP adaption also outperforms, on average, an existing approach not based on metaheuristic search (RQ3.2).

4.6.4. Results for RQ4. Since our proposal is based on bilevel optimization, it is important to evaluate the execution time (*CT*). It is evident that BLOP requires higher execution time than GP, Co-Evol, and DECOR, since BLOP has an optimization algorithm to be executed at the lower level. To reduce the computational complexity of our BLOP adaptation, we selected only best solutions (*nbs%*) at the upper level to update

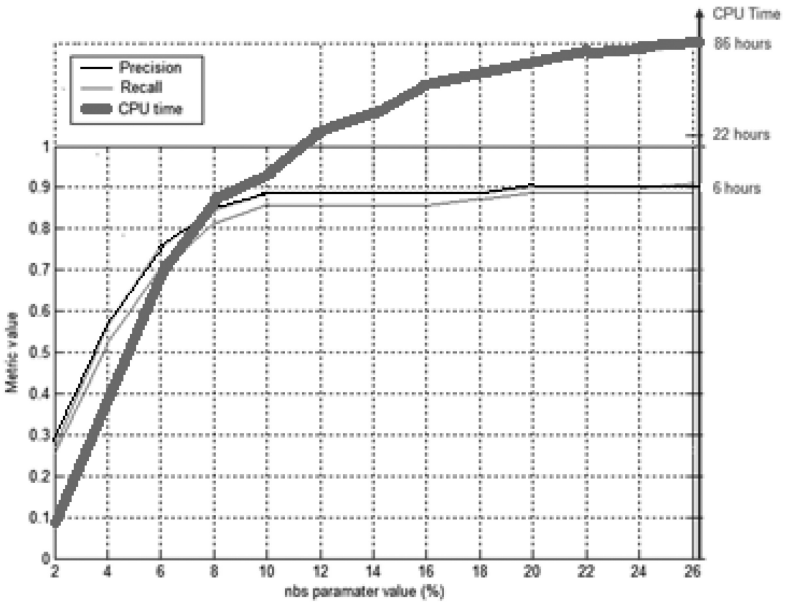


Fig. 11. The impact of the number of selected solutions at the upper level on the quality of the results (*PR*, *Rc*, and *CT*) using JFreeChart v1.0.9.

their fitness evaluations based on the coverage of artificial code smells that are generated by the optimization algorithms executed at the lower level for every selected solution. All the search-based algorithms under comparison were executed on machines with Intel Xeon 3GHz processors and 4GB RAM. We recall that all algorithms were run for 750,000 evaluations. This allows us to make a fair comparison between CPU times. Overall, the GP and Co-Evol algorithms were faster than BLOP. In fact, the average execution time for BLOP, GP, and Co-Evol were, respectively, 6.2, 1.4, and 2.1 hours. However, the execution for BLOP is reasonable, because the algorithm is executed only once, then the generated rules will be used to detect code smells. There is no need to execute BLOP again except in the case that the base of examples will be updated with a high number of new code-smell examples.

An important parameter that reduced the execution time of our BLOP adaptation is the number of selected good solutions at the upper level. Figure 11 shows that the performance of our approach improves as we increase the percentage of best solutions selected from the upper level at each iteration. However, the results become stable after 10% (percentage of selected solutions from the upper-level population). For this reason, we considered this threshold in our experiments that represent a good trade-off between the quality of detection solutions and the execution time. As described in Figure 11, the *PR* and *Rc* scores become almost stable after the 10% threshold value, and the execution time increases dramatically, since a high number of optimization algorithms are executed at the lower level. The evaluation was performed on JFreeChart v1.0.9, but similar facts were found on the remaining systems.

Figure 12 shows the number of evaluations required to generate good code-smell detection rules. We used the *f-measure* metric, defined as the harmonic mean of precision and recall, to evaluate the quality of the best solution at each iteration for each algorithm (BLOP, Co-Evol, and GP). We considered an *f-measure* value higher than 0.7 as an indication of an acceptable detection rules solution based on our corpus. We evaluated which algorithm could reach that threshold value of *f-measure (0.7)* faster.

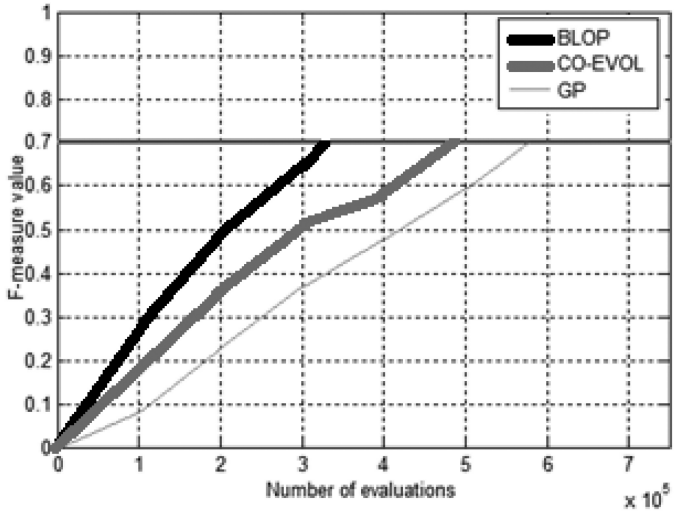


Fig. 12. The number of evaluations required by the different algorithms (BLOP, Co-Evol and GP) to reach acceptable results (f-measure = 0.7) using Xerces-J v 2.7.0.

We selected a threshold value of 0.7, since it represents a good balance between precision and recall that can lead to acceptable detection solutions. We found that our bilevel adaptation required a fewer number of evaluations than Co-Evol and GP to generate good code-smell detection solutions. In fact, after around 325,000 evaluations, BLOP generated detection rules that have 0.7 as the *f-measure* value on Xerces-J. Co-Evol requires at least more than 480,000 evaluations to reach similar solution quality, and GP needs more than 560,000 evaluations to generate similar detection solution. Thus, we can conclude that the lower level helped the upper level to quickly generate good quality of detection solutions by producing in an intelligent manner efficient artificial code smells. Although BLOP needs higher execution time than Co-Evol and GP, as described in Figure 11, it is clear from Figure 12 that the good solutions provided by a single-level approach can be reached quickly by our bilevel adaptation.

To further evaluate the scalability of the performance of bilevel evolutionary algorithms for systems of increasing size, we executed our bilevel tool on Eclipse without assessing the precision and recall scores. Eclipse is an open-source integrated development environment (IDE) written in Java and widely used to develop applications. We considered three versions of Eclipse that contain more than 3.5 MLOCs. Figure 13 describes the execution time of our bilevel approach on seven different versions of Eclipse. We believe that an execution time of seven hours is acceptable and reasonable, since the developers will not use our tool in their daily activities, they just need to execute it once to extract the rules. A new execution of the bilevel algorithm is required when major updates are performed on the base of examples used by the upper level.

5. INDUSTRIAL CASE STUDY AND RELEVANCE OF THE DETECTED CODE SMELLS

The goal of this study is to evaluate the usefulness and effectiveness of our code-smell detection tool in practice. We conducted a nonsubjective evaluation with potential developers who could use our tool related to the relevance of our approach for software engineers.

We performed a small industrial case study, described in Table VII, based on one industrial project JDI-Ford. It is a Java-based software system that helps our industrial partner, the Ford Motor Company, analyze useful information from the past sales of

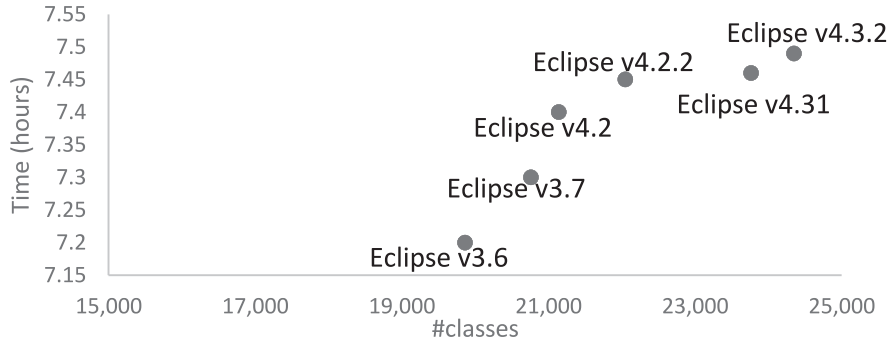


Fig. 13. Scalability of our bilevel approach for code-smell detection on three different versions of Eclipse.

Table VII. Software Studied in Our Experiments

Systems	*Release*	*#Classes*	*#Smells*	*KLOC*
JDI-Ford	v5.8	638	88	247

dealerships data and suggests which vehicles to order for their dealer inventories in the future. This system is the main key software application used by the Ford Motor Company to improve their vehicle sales by selecting the right vehicle configuration to the expectations of customers. Several versions of JDI were proposed by software engineers at Ford during the past 10 years. Due to the importance of the application and the high number of updates performed during a period of 10 years, it is critical to make sure that all the JDI releases are within a good quality to reduce the time required by developers to introduce new features in the future. The software engineers from Ford evaluated the JDI system to manually find code smells based on their knowledge of the system, since they are some of the original developers. We considered those that are detected by the majority of the software engineers to calculate the precision and recall. Our study focused on the usefulness of the detected code smells and the performance of our detection technique in an industrial setting. We also evaluated the relevance of some of the detected code smells on the different open-source systems described in the previous section.

In this section, we will answer to the following question.

How can our bilevel code-smell detection BLOP approach and the detected code smells be useful for software engineers?

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

5.1. Subjects

Our study involved seven subjects from the University of Michigan and eight software engineers from Ford Motor Company. Subjects include three master students in Software Engineering, three Ph.D. students in Software Engineering, one faculty member in Software Engineering, six junior software developers, and two senior projects manager. Six subjects are female, and nine are male. All 15 subjects are familiar with Java development, software maintenance activities including refactoring. The experience of these subjects on Java programming ranged from 2 to 16 years. All the graduate students had an industrial experience of at least two years with large-scale object-oriented systems. The eight software engineers from Ford evaluated the code-smell detection results only on the JDI-Ford system. They were selected, as part of a project funded by

Ford, based on having similar development skills, their motivations to participate in the project, and their availability. They are part of the original developer team of the JDI system.

5.2. Questionnaire, Instructions, and Pilot Study

Subjects were first asked to fill out a pre-study questionnaire containing five questions. The questionnaire helped to collect background information, such as their role within the company, their programming experience, their familiarity with code smells, and software refactoring.

We divided the subjects into four groups to evaluate the relevance of some detected code smells according to the number of studied systems, the number of detected code-smells to evaluate, and the results of the pre-study questionnaire. Due to the high number of code smells to be evaluated, we pick at random a subset of the detected code smells to evaluate their relevance for software engineers, as described in Table VIII. In Table VIII, we summarize how we divided subjects into four groups. All the participants from Ford (Group D) evaluated the relevance of the detected code smells only on the JDI-Ford system. They were also asked to find the other code smells in JDI not detected by our BLOP technique and evaluate the correctness of detected ones.

Each group of subjects who accepted an invitation to participate in the study received a questionnaire, a manuscript guide that helps to fill the questionnaire, and the source code of the studied systems in order to evaluate the relevance of the suggested code smells to fix. The questionnaire is organized in an Excel file with hyperlinks to visualize the source code of the affected code elements easily. The study questionnaire is composed by three main sections and a specific fourth section only for Group D (Ford's developers).

The first part of the questionnaire includes questions to evaluate the relevance of some detected code smells using the following scale: 1. Not at all relevant; 2. Slightly relevant; 3. Moderately relevant; and 4. Extremely relevant. If a detected code smell is considered relevant, then this means that the developer considers it is important to fix it.

The second part of the questionnaire includes questions for those code smells that are considered at least "moderately relevant." We asked the subjects to specify their usefulness to perform some maintenance activities: 1. Refactoring guidance; 2. Quality assurance; 3. Bug prediction; 4. Effort prediction; and 5. Code inspection.

In the third part of the questionnaire, we asked our subjects to fix some of the detected code smells by suggesting and applying some refactorings [Holm 1979].

In the last part dedicated to only Group D (developers from Ford), we asked them to explore the JDI source code, analyze the metrics value of the system, and use their knowledge of the existing implementation (since they are the original developers) in order to identify the remaining code smells not detected by our BLOP technique. In addition, they evaluated the correctness of detected ones by BLOP. We considered the majority of votes (more than four votes) to evaluate the correctness of detected code-smells.

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

During the entire process, subjects were encouraged to think aloud and to share their opinions, issues, detailed explanations, and ideas with the organizers of the study (one

Table VIII. Survey Organization to Study the Relevance of Some Detected Code Smells

Subject Groups	Systems	Selected Code Smells
Group A	JFreeChart v1.0.9	*Blob*: 4 *Data Class*: 8 *Spaghetti Code*: 4 *Feature Envy*: 6 *Lazy Class*: 4 *Long Parameter List*: 6 *Functional Decomposition*: 3
	GanttProject v1.10.2	*Blob*: 4 *Data Class*: 3 *Spaghetti Code*: 4 *Feature Envy*: 2 *Lazy Class*: 2 *Long Parameter List*: 3 *Functional Decomposition*: 4
	ApacheAnt v1.5.2	*Blob*: 5 *Data Class*: 2 *Spaghetti Code*: 3 *Feature Envy*: 3 *Lazy Class*: 2 *Long Parameter List*: 2 *Functional Decomposition*: 2
	ApacheAnt v1.7.0	Blob: 5 Data Class: 2 Spaghetti Code: 3 Feature Envy: 3 Lazy Class: 2 Long Parameter List: 2 Functional Decomposition: 3
Group B	Nutch v1.1	Blob: 4 Data Class: 3 Spaghetti Code: 3 Feature Envy: 2 Lazy Class: 4 Long Parameter List: 3 Functional Decomposition: 6
	Log4J v1.2.1	Blob: 2 Data Class: 3 Spaghetti Code: 2 Feature Envy: 4 Lazy Class: 1 Long Parameter List: 3 Functional Decomposition: 3
	Lucene v1.4.3	Blob: 2 Data Class: 2 Spaghetti Code: 3 Feature Envy: 2 Lazy Class: 2 Long Parameter List: 3 Functional Decomposition: 2
Group C	Xerces-J V2.7.0	Blob: 3 Data Class: 2 Spaghetti Code: 4 Feature Envy: 4 Lazy Class: 2 Long Parameter List: 4 Functional Decomposition: 4

(Continued)

Table VIII. Continued

Subject Groups	Systems	Selected Code Smells
	Rhino v1.7R1	Blob: 2 Data Class: 2 Spaghetti Code: 2 Feature Envy: 4 Lazy Class: 4 Long Parameter List: 2 Functional Decomposition: 4
Group D	JDI-Ford v5.8	Blob: 6 Data Class: 8 Spaghetti Code: 8 Feature Envy: 8 Lazy Class: 12 Long Parameter List: 12 Functional Decomposition: 12

graduate student and one faculty from the University of Michigan), not only answering the questions.

A brief tutorial session was organized for every participant around code smells and refactoring to make sure that all of them had a minimum background to participate in the study. The instructions indicate also that the developers need to inspect the source code to evaluate the detected code smells and their relevance and not by evaluating the quality metric values. In addition, all the developers performed the experiments in a similar environment: similar configuration of the computers, tools (Eclipse, Excel, etc.), and facilitators of the study. Because some support was needed for the installation of our Eclipse plugin and the other detection techniques considered in our experiments, we added a short description of this instruction for the participants. These sessions were also recorded as audio, and the average time required to finish all the questions was five hours. The average time to answer the first three parts of the questionnaire was three hours, but the participants from Ford spent a considerable time finding the code smells not detected by our BLOP technique.

Prior to the actual experiment, we did a pilot run of the entire experiment with two subjects, one average-performing student and one software engineer from Ford. We performed this pilot study to verify whether the assignments were clear and if our estimation of the required time to finalize the experiments evaluation were realistic, thus all the assignments could be completed in an afternoon session by the subjects. The pilot study pointed out that the assignments and the questions in the questionnaire form were clear and relevant, and that they could be executed as offered by the subjects of the pilot study within four hours. The pilot study also pointed out that the description of code smells and the examples were clear and sufficient to understand the different types of code smells considered in our experiments. However, the pilot study showed that subjects had problems efficiently evaluating the usefulness of the identified code smells, especially for the open-source systems, since they are not the original developers and could not understand all the design decisions. For this reason, we extended our experiments to include the industrial project from Ford and some of their original developers.

5.3. Results of the Industrial Case Study and Relevance of Detected Code Smells

For the Ford system, the original developers conducted the study (group D). Hence, they are almost sure whether a code fragment is affected by code smell or not, whether it should be refactored, or whether, although a tool says this is a smell, there are good reasons for the design and implementation choices made. Unfortunately, this is not

Table IX. Manual Validation of the Detected Code Smells on JDI-Ford

System	*PR-BLOP*	*PR-GP*	*PR-Co-Evol*	*PR-RS*	*Rc-BLOP*	*Rc-GP*	*Rc-Co-Evol*	*Rc-RS*
JDI-Ford v5.8	92% (76/82)	67% (63/94)	78% (69/88)	18% (27/148)	86% (76/88)	72% (63/88)	78% (69/88)	31% (27/88)

Table X. Adjusted p-Values of Comparisons Related to Table IX

	BLOP	GP	Co-Evol
JDI-Ford v5.8	0.00327 (GP) 0.00167 (Co-Evol) 0.00112 (RS)	0.00398 (Co-Evol) 0.00213 (RS)	0.00187 (RS)

possible for the other subjects and open-source systems, who have evaluated code not familiar to them. For this reason, we describe the results of the evaluation in two separate sections.

5.3.1. Industrial Case Study : Detection Results and Relevance of Detected Code Smells. In this section, we evaluate the performance of our BLOP detection technique on the detection of seven different types of code smells in an industrial setting and compare it with some existing code-smell detection techniques. As described in Table IX, the expected code smells were detected with more than 90% of precision on the JDI-Ford system that corresponds to the highest precision comparing to random search, GP, and Co-Evol. The lowest precision was found by the random search of only 18%. The precision of Co-Evol is higher than Co-Evol and random search but lower than BLOP on the JDI industrial system. This confirms the importance and efficiency of the lower level considered in our BLOP adaptation for improving the quality of detection solutions generated by the upper level. Similar facts were found when analyzing the recall scores of BLOP on the JDI-Ford, where 86% of expected code smells were detected. The recall score of BLOP confirms its outperformance compared to random search, GP, and Co-Evol.

Table X confirms that the comparison results of the different detection techniques on JDI-Ford are statistically significant with a 99% confidence level ($\alpha = 1\%$).

To better investigate the relevance of the detected code smells for software engineers, we asked group D to fix some of the detected code smells (described in Table XIII) in JDI-Ford by manually suggesting and applying some refactorings [Holm 1979] on JDI-Ford. Then, we used the QMOOD (Quality Model for Object-Oriented Design) model [Bansiya and Davis 2002] to estimate the effect of the fixed code smells on quality attributes. QMOOD has the advantage of defining six high-level design quality attributes (reusability, flexibility, understandability, functionality, extendibility, and effectiveness) that can be calculated using 11 lower-level design metrics. In our study we consider the following quality attributes.

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

Table XI and Table XII summarize the QMOOD formulation of these quality attributes [Bansiya and Davis 2002].

The improvement in quality can be assessed by comparing the quality before and after refactoring applied to fix a number of code smells. Hence, the total gain in quality G for each of the considered QMOOD quality attributes q_i before and after refactoring

Table XI. QMOOD Quality Factors [Bansiya and Davis 2002]

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

Table XII. QMOOD Metrics for Design Properties [Bansiya and Davis 2002]

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

can be easily estimated as

$$Gq_i = q_i' - q_i,$$

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

We see in Figure 14 that all the quality attributes of both systems are improved after fixing several code smells from the JDI-Ford system. Understandability is the quality factor that has the highest gain value; whereas the effectiveness quality factor has the lowest. This can be explained by the types of fixed code smells that are known to increase the coupling within classes that heavily affect the quality index calculation of the effectiveness factor. Another reason relates to the types of manually suggested refactoring, such as move method, move field, and extract class, that are known to have a high impact on coupling, cohesion, and the design size in classes that serve to calculate the understandability quality factor. To conclude, fixing the detected code smells by our tool can lead to better code quality and improve the developers' understandability, the reusability, the flexibility, and the effectiveness of the refactored system.

As described in Table XIII, we asked group D to evaluate the relevance of a randomly selected set of selected code smells on the JDI-Ford system. Figure 15 illustrates that only less than 18% of detected code smells are considered not at all relevant by software engineers. Around 65% of the code smells are considered as moderately or extremely relevant by software developers. This confirms the importance of the detected code smells for developers that need to be fixed for better quality of their systems.

It is also important to evaluate the usefulness of the detected code smells for software maintainers in their daily maintenance activities. To this end, we asked the Ford software developers to justify the usefulness of the code smells ranked as moderately or extremely relevant. Figure 16 describes the obtained results. The main usefulness is related to refactoring guidance. In fact, most of the software engineers we interviewed found that the detected code smells give relevant advice about where refactorings should be applied and what are the commonly used refactorings to fix these defects.

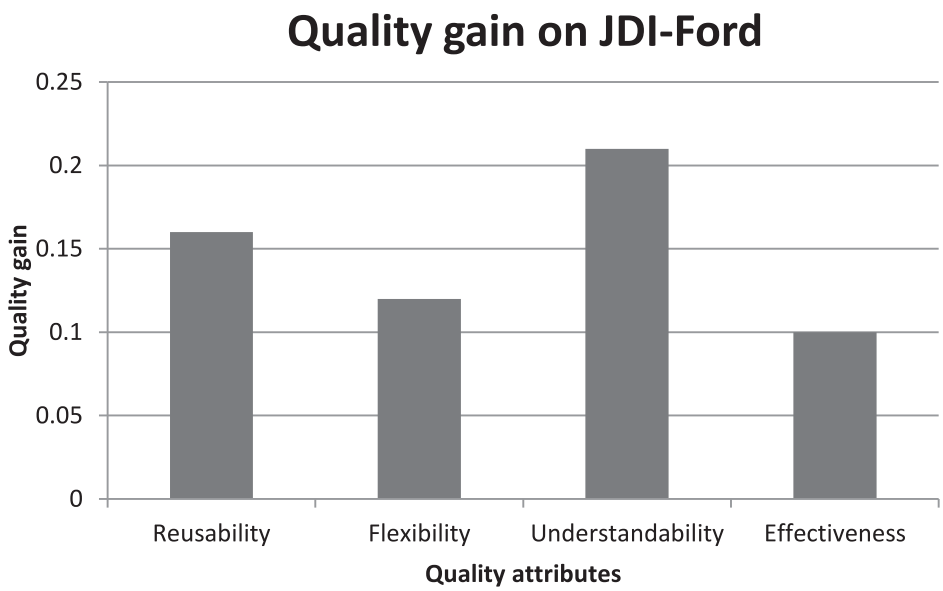


Fig. 14. The impact of fixing a number of code smells (refactorings) on QMOOD quality attributes for JDI-Ford

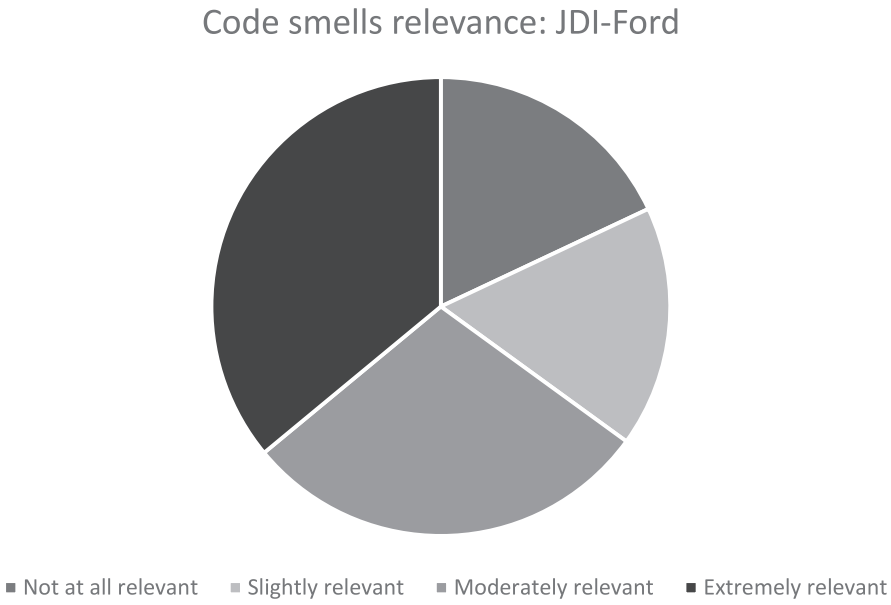


Fig. 15. The relevance of detected code smells on the JDI-Ford system evaluated by the original developers.

In addition, they found that the code-smell detection process is much more helpful than the traditional analysis of software metrics to find refactoring opportunities. They consider the use of traditional software metrics for quality assurance as a time-consuming process, and it is easier to interpret the results of detected code smells and apply the appropriate refactorings to improve the overall quality of the system.

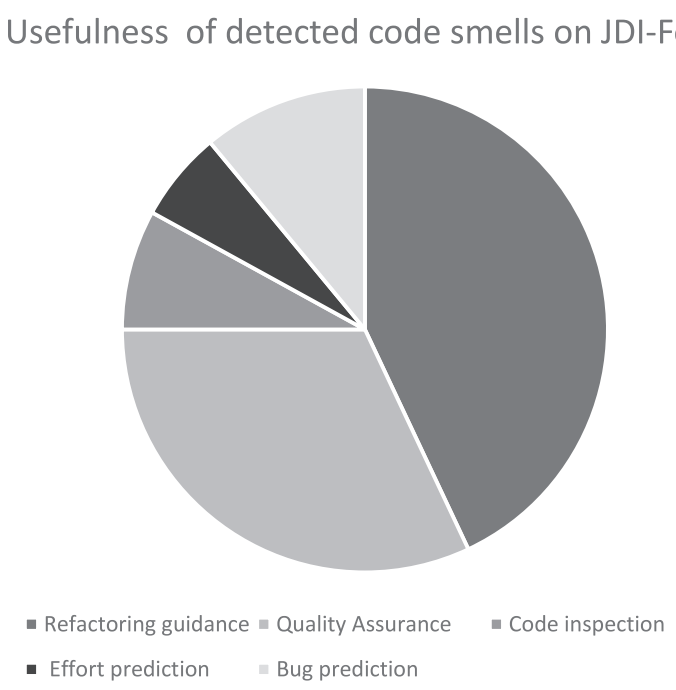


Fig. 16. The usefulness of detected code smells on the JDI-Ford system evaluated by the original developers.

We summarize briefly, in the following, the feedback of the Ford developers during the think-aloud sessions. Most of the participants mention that the detection rules generated by BLOP represents a faster solution than manually refactoring opportunities detection. The manual techniques represent a time-consuming process to find the locations where the refactoring should be applied or to calibrate the metrics threshold or the combination of metrics to identify a maintainability issue manually. The participants found the detection rules useful to maintaining a good quality of design and to making sure that some quality issues are fixed after refactoring. In addition, the developers liked the flexibility of modifying the rules (metrics or thresholds) if required. Some possible improvements for our detection techniques were also suggested by the participants. Some participants believe that it would be very helpful to extend the tool by adding a new feature to rank the detected code smells based on several criteria, such as risk, cost, and benefits. They believe that current refactoring tools do not provide any support for estimating the risk, cost, and benefits of fixing some maintainability issues. In fact, most of Ford's developers mentioned that they do not fix some code smells if they feel that it will require a high cost, if the benefit is not clear, or if the code is stable.

5.3.2. Relevance of the Detected Code Smells on the Open-Source Systems. We evaluated the relevance of the detected code smells on the different open-source systems by participants of groups A, B, and C, as described in Table XIII. Figure 17 illustrates that only less than 15% of detected code smells are considered not at all relevant by the software engineers. Around 70% of the code smells are considered as moderately or extremely relevant by the different groups, and this confirms the importance of the detected code smells for developers; the results are also similar to the industrial case study.

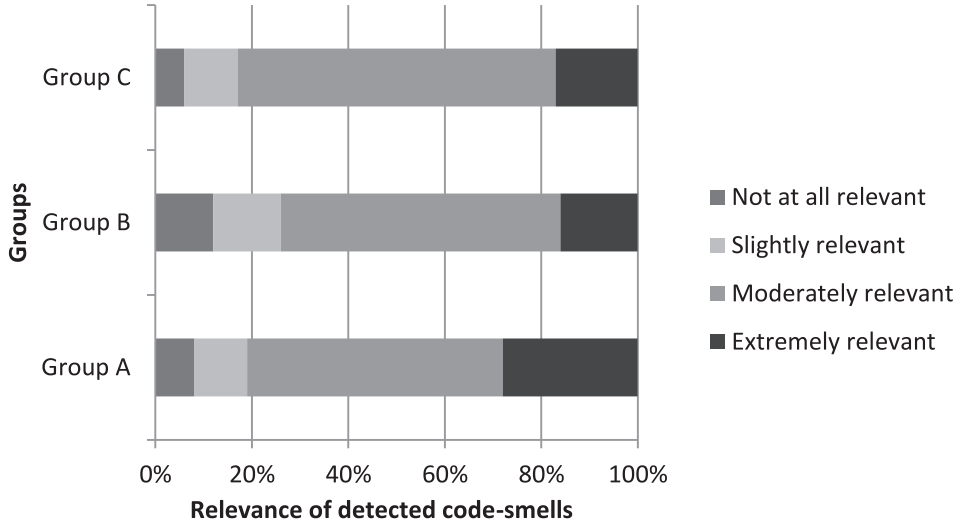


Fig. 17. The relevance of detected code smells on the different open-source systems evaluated by developers from groups A, B, and C.

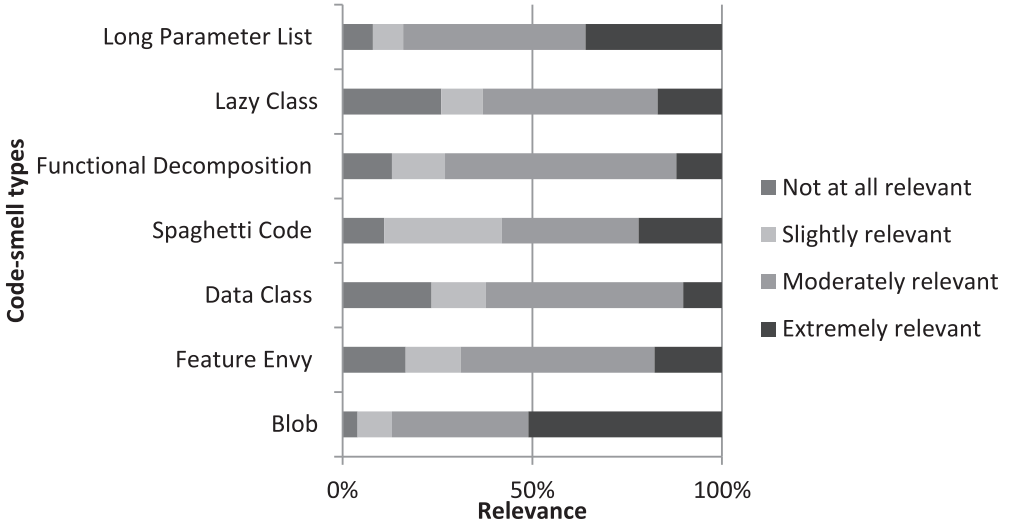


Fig. 18. The relevance of the types of detected code smells on the different open-source systems evaluated by developers from the groups A, B, and C.

To better evaluate the relevance of the detected code smells, we investigated the types of code smells that developers consider more or less important than others (e.g., blob, lazy class, etc.). Figure 18 summarizes our findings. It is clear that the detected blobs are considered very relevant for developers. One of the reasons can be the importance of distributing the behavior/functionalities of a program between different classes. In addition, it is very difficult for developers to understand existing functionalities to add new ones if most of them are implemented in one or a few classes. Another interesting observation is that lazy classes are not considered very relevant by developers. In fact, lazy classes do not implement, in general, any important feature, thus software engineers did not consider it an important concern.

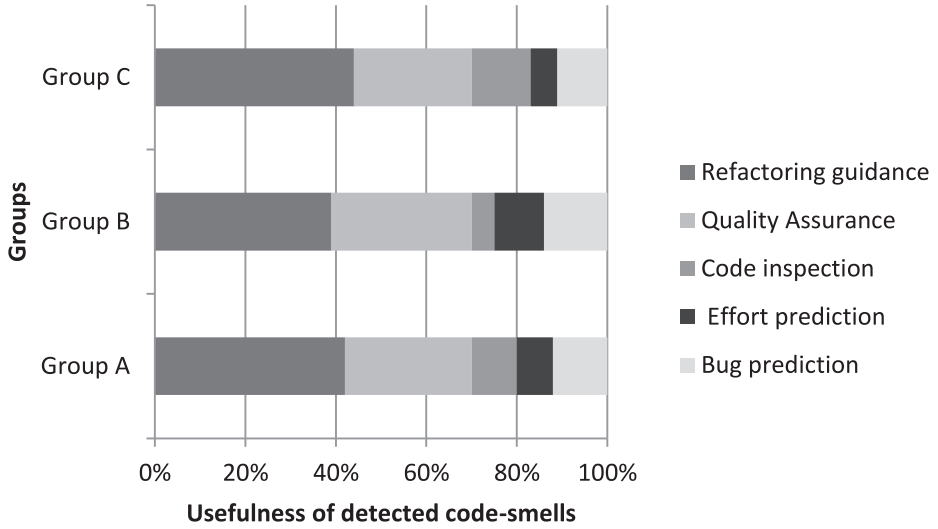


Fig. 19. The usefulness of detected code smells for software maintenance activities on the different open-source systems evaluated by developers from the A, B, and C groups.

We asked the different groups of software engineers (A–C) to justify the usefulness of the code smells ranked as moderately or extremely relevant. Figure 19 describes the different reasons of the importance of detected code smells. Similar to the industrial case study's results, the main usefulness is related to refactoring guidance and quality assurance.

6. THREATS TO VALIDITY

We explore, in this section, the factors that can bias our empirical study. These factors can be classified in three categories: construct, internal, and external validity. Construct validity concerns the relationing 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 threats are related to the absence of similar work that uses bilevel techniques for code-smell detection. For that reason, we compare our proposal with two other search-based techniques [Kessentini et al. 2011; Boussaa et al. 2013] and DECOR [Moha et al. 2010]. However, we are aware that there are other tools able to detect several types of code smells that also can be considered in our experiments [Palomba et al. 2013].

Another threat to construct validity arises because, although we considered seven well-known code-smell types, we did not evaluate the performance of our proposal with other code-smell types. We must further evaluate the performance and ability of our bilevel technique to detect other smells.

A construct threat can also be related to the corpus of manually detected code-smells, since developers do not all agree if a candidate is a code smell or not. We will ask some new experts to extend the existing corpus and provide additional feedback regarding the detected code smells. In addition, the recall score is challenging to calculate by the software engineers in our experiments and requires additional participants to check its accuracy.

A limitation related to our experiments is the difficulty in setting the thresholds for DECOR. In fact, we used the default thresholds used by DECOR's authors that can have an impact on the quality of the results generated by DECOR.

The evaluation of detected code smells for some participants is mainly based on the definitions of the code smells and the examples that we provided during the pilot study. However, the definition of code smells is subjective and depends on the programming behavior of the participants, thus this could affect the accuracy of the detection results.

A construct threat is related to the fact that our detection results depend on the examples of code smells and well-designed code. In addition, the generation of artificial code fragments can lead to several non-useful examples (generated by the lower level). Additional constraints should be defined to better guide the search at a lower level to refine the generation of artificial code-smell examples.

The same observation is valid for the used change operators at both the upper and lower levels that can generate invalid rules and code-smell examples (e.g., redundancy) may be avoided by the definition of additional constraints.

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 using the Wilcoxon rank sum test [Arcuri and Fraser 2013] with a 99% confidence level ($\alpha = 1\%$).

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 by applying additional experiments to evaluate the impact of upper- and lower-level parameters on the quality of the results.

Another internal threat is related to the fact that our tool is not able to rank the detected code smells. The software engineers considered in our experiments found that the type of detected code smells is helpful in determining its importance. In addition, our contribution in this article is mainly related to the detection of code smells. We will propose several measures that can be used to rank the detected code smells by our rules. Thus, we consider in our work the detection and ranking of code smells as two separate steps.

The participants considered in our experiments are not the original developers of the open-source systems. Thus, some of their evaluations of the detected code smells could be not very accurate. In fact, there are, sometimes, good reasons for the design and implementation choices made, and this can be mainly determined by the original developers. However, this is not the case for the Ford project, since some of the original developers of the system participated in our experiments. We are planning to integrate a few of the original developers from these open-source projects to evaluate the detected code smells as part of our future work.

Our experiments also lack the evaluation of the impact of removing the detected code smells on the productivity of the developers and long-term maintenance objectives. Another internal threat is the possibility that some developers did not report all the maintainability issues in the evaluated systems, thus some detected code smells are not considered very useful. The use of triangulation as suggested by Yamashita and Moonen [2012] based on the usage of three independent collection methods, such as interviews, direct observation, and think-aloud sessions, may reduce this threat.

The correction of code smells can be performed by different refactoring strategies, thus the quality improvements of the code after fixing some code smells depends on the applied refactoring. Consequently, further investigation is needed to evaluate the stability of the refactoring results.

For the selection threat, the subject diversity in terms of profile and experience could affect our study. We mitigated the selection threat by giving written guidelines and

examples of code smells already evaluated with arguments and justification. Additionally, each group of subjects evaluated different code smells from different systems using different techniques/algorithms. Randomization also helps to prevent the learning and fatigue threats. Only a few code smells per system were randomly picked for the evaluation.

Diffusion threat is related to the fact that most of the subjects are from the same university, and the majority know each other. However, they were instructed not to share information about the experience before a certain date.

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

We selected subjects who had similar programming skill levels to reduce the impact of skills on the results, but it is challenging to evaluate the background and skills of the participants objectively. However, a larger number of subjects is required to improve the evaluation of our detection technique. In addition, the manual validation of the detected code smells is limited to some samples of detected code smells due to the limited number of participants.

An evaluation of the usefulness of detected code smells is based only on four maintainability objectives from QMOOD, thus the consideration of additional objectives such as performance and the number of bugs after fixing the smells could lead to a better evaluation of the usefulness of detected code smells. In addition, the quality of detection rules depends on the quality metrics used as input by our bilevel technique, which are limited to 14 metrics in our experiments. Additional metrics may lead to a better quality of results with other programming languages.

An evaluation of the relevance of detected code smells in our experiments heavily depends on the opinion of the developers, and it is difficult to generalize due to the limited number of participants in our experiments.

7. RELATED WORK

There are several studies that have recently focused on detecting code smells in software using different techniques. These techniques range from fully automatic detection to guided manual inspection.

7.1. Interactive-Based Approaches

Fowler et al. [1999] described a list of design smells that may exist in the program. They suggested that the software maintainers should manually inspect the program to detect existing design smells. In addition, they specify particular refactorings for each code-smell type. Van Emden and Moonen [2002] have also proposed a manual approach for detecting code smells in object-oriented designs. The idea is to create a set of "reading techniques" that help a reviewer to "read" a design artifact for finding relevant information. These reading techniques give specific and practical guidance for identifying code smells in object-oriented design. In this way, each reading technique helps the maintainer focus on some aspects of the design in such a way that the inspection team applying the entire family should achieve a high degree of coverage of the design code smells. Ciupke [1999] is another proposed approach based on violations of design rules and guidelines. This approach consists of analyzing legacy code, specifying frequent design problems as queries, and locating the occurrences of these problems in a model derived from the source code.

The high rate of false-positives generated by the previously mentioned approaches encouraged other teams to explore semiautomated solutions. These solutions took the

form of visualization-based environments. The primary goal is to take advantage of the human capability to integrate complex contextual information in the detection process. Kothari et al. [2004] present a pattern-based framework for developing tool to detect software anomalies by representing potential code smells with different colors. Dhambri et al. [2008] have proposed a visualization-based approach to detect design anomalies by automatically detecting some symptoms and leaving others to a human analyst. The visualization metaphor was chosen specifically to reduce the complexity of dealing with a large amount of data. Although visualization-based approaches are efficient to examine potential code smells on their program and in their context, they do not scale to large systems easily. In addition, they require great human expertise, and thus they are still time-consuming and error-prone strategies. Moreover, the information visualized is mainly metric-based, meaning that complex relationships can be difficult to detect. Indeed, since visualization approaches and tools such as VERSO [Dhambri et al. 2005] are based on manual and human inspection, they are still not only slow and time-consuming, but also subjective.

The main disadvantage of existing manual and interactive-based approaches is that they are ultimately a human-centric process which requires a great human effort and strong analysis and interpretation effort from software maintainers to find design fragments that correspond to code smells. In addition, these techniques are time consuming, error prone, and depend on programs in their contexts.

7.2. Symptom-Based Detection

Moha et al. [2010] started by describing code-smell symptoms using a domain-specific-language (DSL) for their approach called DECOR. They proposed a consistent vocabulary and DSL to specify anti-patterns based on the review of existing work on design code smells found in the literature. Symptom descriptions are later mapped to detection algorithms. However, converting symptoms into rules needs a significant analysis and interpretation effort to find the suitable threshold values. In addition, this approach uses heuristics to approximate some notions, which results in an important rate of false positives. Indeed, this approach has been evaluated on only four well-known design code smells: the blob, functional decomposition, spaghetti code, and Swiss-Army knife, because the literature provides obvious symptom descriptions on these code smells. Recently, another probabilistic approach has been proposed by Khomh et al. [2009] extending the DECOR approach, a symptom-based approach, to support uncertainty and to sort the code-smell candidates accordingly using a Bayesian Belief Network (BBN). The detection outputs are probabilities that a class is an occurrence of a code-smell type, that is, the degree of uncertainty for a class to be a code smell. They also showed that BBNs can be calibrated using historical data from both similar and different context. Similarly, Munro [2005] proposed a template-based approach using a precise definition of bad smells from the informal descriptions given by the originators [Fowler et al. 1999]. The template consists of three main parts: a code-smell name, a text-based description of its characteristics, and heuristics for its detection.

In another category of work based on the use quality metrics, Marinescu [2004] proposed a mechanism called "detection strategy" for formulating metric-based rules that capture deviations from good design principles and heuristics. Detection strategies allow a maintainer to directly locate classes or methods affected by a particular design code smell. As such, Marinescu has defined detection strategies for capturing around 10 important flaws of object-oriented design found in the literature. After his suitable symptom-based characterization of design code smells, Salehie et al. [2006] proposed a metric-based heuristic framework to detect and locate object-oriented design flaws similar to those illustrated by Mika and Lassenius [2006]. It is accomplished by evaluating design quality of an object-oriented system through quantifying deviations from good

design heuristics and principles by mapping these design flaws to class-level metrics, such as complexity, coupling, and cohesion by defining rules. Legillon et al. [2012] introduced the concept of multimetrics as an n-tuple of metrics expressing a quality criterion (e.g., modularity). Unfortunately, multimetrics neither encapsulate metrics in a more abstract construct, nor do they allow a flexible combination of metrics.

Van Emden and Moonen [2002] developed one of the first automated code-smell detection tools for Java programs. They developed a prototype code-smell browser that detects and visualizes code smells in Java programs. Mika [2010] studied the manner of how developers detect and analyze code smells. Previous empirical studies have analyzed the impact of code smells on different software maintainability factors [Monden et al. 2002]. In fact, software metrics (quality indicators) are sometimes difficult to interpret and suggest some actions (refactoring), as noted by Monden et al. [2002] and Ratiu et al. [2004]. In addition, Yamashita and Moonen [2012] show that the different types of code smells can cover different maintainability factors [Brito e Abreu and Melo 1996]. In other studies, Yamashita and Moonen [2013a, 2013b] analyzed the benefits of detecting code-smells on several industrial projects and evaluated their impact on different maintainability aspects. Recently, Palomba et al. [2013] used the history of changes to detect code smells instead of the use of quality metrics. For example, classes that change frequently are considered a blob. A comparison with DECOR confirms the outperformance of their approach and the benefits of the use of the history of changes for the detection of code smells.

7.3. Search-Based Approaches

SBSE [Harman et al. 2012] uses search-based approaches to solve optimization problems in software engineering. After the formulation of software engineering task as a search problem, many search algorithms can be applied to solve that problem. Harman [2007] proposed another approach, based on search-based techniques, for the automatic detection of potential code smells in code. We used the notion that the more code deviates from good practices, the more likely it is bad. In Kessentini et al. [2011], we generated detection rules defined as combinations of metrics/thresholds that better conform to known instances of bad smells (examples). Then, the correction solutions, a combination of refactoring operations, should minimize the number of bad smells detected using the detection rules. Thus, our previous work treats the detection and correction as two different steps.

Based on recent SBSE surveys [Harman et al. 2012; Harman 2007], the use of bilevel optimization is still very limited in software engineering. Indeed, this work represents the first attempt to use bilevel optimization to address software engineering problem.

8. CONCLUSIONS AND FUTURE WORK

In this article, we have addressed the problem of the absence of consensus in code-smell detection. In fact, choosing quality metrics to detect symptoms of code smells is not straightforward in software engineering and is usually a challenging task. In order to tackle this problem, we have proposed a bilevel evolutionary optimization approach. The upper-level optimization produces a set of detection rules, which are combinations of quality metrics, with the goal to maximize the coverage of not only a code-smell example base but also a lower-level population of artificial code smells. The lower-level optimization tries to generate artificial code smells that cannot be detected by the upper-level detection rules, thereby emphasizing the generation of broad-based and fitter rules. The statistical analysis of the obtained results over nine studied software systems have shown the competitiveness and the outperformance of our proposal in terms of precision and recall over a single-level genetic programming, co-evolutionary, and non-search-based methods.

Following this work, we have identified several avenues for future research. First, the main problem when using bilevel optimization in software engineering is the computational cost required for the lower-level search. Hence, it would be interesting to use regression methods for approximating the lower-level optimum for a given upper-level solution. In this way, we could minimize the required number of function evaluations significantly. Second, the idea of bilevel optimization seems interesting for several other SE problems. It would be challenging to model and then solve other interesting SE problems in a bilevel manner. We are currently working on extending our work by proposing a bilevel approach for the correction of code smells. Finally, our contribution in this article is mainly related to the detection of code smells. We will propose several measures that can be used to rank the detected code smells by our rules.

REFERENCES

Alain Abran and Hong Nguyenkim. 1993. Measurement of the maintenance process from a demand-based perspective. *J. Softw. Maint. Res. Pract.* 5, 2 (1993), 63–90.

Eitaro Aiyoshi and Kiyotaka Shimizu. 1981. Hierarchical decentralized systems and its new solution by a barrier method. *IEEE Trans. Syst. Man Cybernet.* 11, 6 (1981), 444–449.

Bente C. D. Anda. 2007. Assessing software system maintainability using structural measures and expert assessments. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'07)*. 204–213.

Andrea Arcuri and Gordon Fraser. 2013. Parameter tuning or default values? An empirical investigation in search-based software engineering. *Emp. Softw. Eng.* 18, 3 (2013), 594–623.

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

Jonathan F. Bard and James E. Falk. 1982. An explicit solution to the multi-level programming problem. *Comput. Oper. Res.* 9, 1, 77–100.

J. Bard. 1998. *Practical Bilevel Optimization: Algorithms and Applications*. Kluwer Academic Publishers, The Netherlands.

Victor R. Basili, Lionel C. Briand, and Walcélio L. Melo. 1996. A validation of object-oriented design metrics as quality indicators. *IEEE Trans. Softw. Eng.* 22, 10 (1996), 751–761.

Keith H. Bennett and Václav T. Rajlich. 2000. Software maintenance and evolution: A roadmap. In *Proceedings of the Conference on The Future of Software Engineering (ICSE'00)*. ACM, New York, NY, 73–87.

Wayne F. Bialas, Mark H. Karwan, and J. Shaw. 1980. A parametric complementarity pivot approach for two-level linear programming. Technical Report 80-2, Operations Research Program, State University of New York at Buffalo.

Mohamed Boussaa, Wael Kessentini, Marouane Kessentini, Slim Bechikh, and Soukeina Ben Chikha. 2013. Competitive coevolutionary code-smells detection. In *Proceedings of the 5th IEEE Symposium on Search-Based Software Engineering (SSBSE'13)*. 50–65.

J. Bracken and J. McGill. 1973. Mathematical programs with optimization problems in the constraints. *Oper. Res.* 21, 1 (1973), 37–44.

Lionel C. Briand, Khaled E. Emam, and Sandro Morasca. 1995. Theoretical and empirical validation of software metrics. In *Proceedings of the International Software Engineering Research Network.*

Lionel C. Briand, John W. Daly, and Jürgen K. Wüst. 1999. A unified framework for coupling measurement in object-oriented systems. *IEEE Trans. Softw. Eng.* 25, 1 (1999), 91–121.

Fernando Brito e Abreu and Walcelio Melo. 1996. Evaluating the impact of object-oriented design on software quality. In *Proceedings of the 3rd International Symposium on Software Metrics: From Measurement to Empirical Results (METRICS'96)*. IEEE Computer Society, 90–99.

Fernando Brito e Abreu. 1995. The MOOD metrics set. In *Proceedings of the European Conference on Object–Oriented Programming (ECOOP'95) Workshop on Metrics*.

William J. Brown, Raphael C. Malveau, Hays W. McCormick, III, and Thomas J. Mowbray. 1998. *Antipatterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley & Sons, Inc., New York, NY.

Herminia I. Calvete and Carmen Galé. 2010. A multiobjective bilevel program for production-distribution planning in a supply chain. In *Multiple Criteria Decision Making for Sustainable Energy and Transportation Systems*, Lecture Notes in Computer Science, vol. 634, Springer-Verlag, Berlin/ Heidelberg, 155–165.

Wilfred Candler and Robert Townsley. 1982. A linear two-level programming problem. *Comput. Oper. Res.* 9, 1, 59–76.

Syham R. Chidamber and Chris F. Kemerer. 1994. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.* 20, 6 (1994), 476–493.

Neville I. Churcher and Martin J. Shepperd. 1995. Comments on 'A Metrics Suite for Object Oriented Design'. *IEEE Trans. Softw. Eng.* 21, 3 (1995), 263–265.

Oliver Ciupke. 1999. Automatic detection of design problems in object-oriented reengineering. In *Proceedings of the Technology of Object-Oriented Languages and Systems (TOOLS'99)*. IEEE Computer Society, 18–32.

Peter Coad and Edward Yourdon. 1991. *Object-Oriented Design*. Yourdon Press, Upper Saddle River, NJ.

Benoit Colson, Patrice Marcotte, and Gilles Savard. 2005a. A trust-region method for nonlinear bilevel programming: Algorithm and computational experience. *Comput. Optim. Appl.* 30, 3 (2005), 211–227.

Benoit Colson, Patrice Marcotte, and Gilles Savard. 2005b. Bilevel programming: A survey. *4OR* 3, 2 (2005), 87–107.

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

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

Ignatios Deligiannis, Ioannis Stamelos, Lefteris Angelis, Manos Roumeliotis, and Martin Shepperd. 2004. A controlled experiment investigation of an object-oriented design heuristic for maintainability. *J. Syst. Softw.* 72, 2 (2004), 129–143.

Karim Dhambri, Houari A. Sahraoui, and Pierre Poulin. 2008. Visual detection of design anomalies. In *Proceedings of the 12th European Conference on Software Maintenance and Reengineering (CSMR'08)*. 279–283.

Karin Erni and Claus Lewerentz. 1996. Applying design metrics to object-oriented frameworks. In *Proceedings of the 3rd International Symposium on Software Metrics: From Measurement to Empirical Results (METRICS'96)*. 64.

Norman E. Fenton and Shari Lawrence Pfleeger. 1998. *Software Metrics: A Rigorous and Practical Approach* 2nd Ed. PWS Pub. Co., Boston, MA.

Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. 1999. *Refactoring: Improving the Design of Existing Code* 1st Ed. Addison-Wesley.

David E. Goldberg. 1989. *Genetic Algorithms in Search, Optimization and Machine Learning* 1st Ed. Addison-Wesley Longman Publishing Co., Inc., Boston, MA.

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

Mark Harman. 2007. The current state and future of search based software engineering. In *Proceedings of the Future of Software Engineering (FOSE'07)*. IEEE Computer Society, 342–357.

Rachel Harrison, Steve J. Counsell, and Reuben V. Nithi. 1998. An evaluation of the MOOD set of object-oriented software metrics. *IEEE Trans. Softw. Eng.* 24, 6 (1998), 491–496.

Sture Holm. 1979. A simple sequentially rejective multiple test procedure. *Scand. J. Stat.* 6, 2, 65–70.

Marouane Kessentini, Stéphane Vaucher, and Houari Sahraoui. 2010. Deviance from perfection is a better criterion than closeness to evil when identifying risky code. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE'10)*. ACM, New York, NY, 113–122.

Marouane Kessentini, Wael Kessentini, Houari A. Sahraoui, Mounir Boukadoum, and Ali Ouni. 2011. Design defects detection and correction by example. In *Proceedings of the 19th IEEE International Conference on Program Comprehension (ICPC'11)*. 81–90.

Foutse Khomh, Stéphane Vaucher, Yann G. Guéhéneuc, and Houari A. Sahraoui. 2009. A Bayesian approach for the detection of code and design smells. In *Proceedings of the 9th International Conference on Quality Software (QSIC'09)*.

Andrew Koh. 2013. A metaheuristic framework for bi-level programming problems with multi-disciplinary applications. In *Metaheuristics for Bi-level Optimization*, Studies in Computational Intelligence, Vol. 482, Springer-Verlag, Berlin/Heidelberg, 153–87.

Charles D. Kolstad. 1985. A review of the literature on bi-level mathematical programming. Technical Report LA-10284-MS, Los Alamos National Laboratory, Los Alamos, NM.

S. C. Kothari, Luke Bishop, Jeremias Sauceda, and Gary Daugherty. 2004. A pattern-based framework for software anomaly detection. *Softw. Qual. Control* 12, 2 (2004), 99–120.

Guillaume Langelier, Houari Sahraoui, and Pierre Poulin. 2005. Visualization-based analysis of quality for large-scale software systems. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE'05)*. ACM, New York, NY, 214–223.

François Legillon, Arnaud Liefooghe, and El-Ghazali G. Talbi. 2012. CoBRA: A cooperative coevolutionary algorithm for bi-level optimization. In *Proceedings of the IEEE Congress on Evolutionary Computation (CEC'12)*. 1–8.

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

Hui Liu, Limei Yang, Zhendong Niu, Zhyi Ma, and Weizhong Shao. 2009. Facilitating software refactoring with appropriate resolution order of bad smells. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'09)*. ACM, New York, NY, 265–268.

Radu Marinescu. 2004. Detection strategies: Metrics-based rules for detecting design flaws. In *Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM'04)*. IEEE Computer Society, 350–359.

Mäntylä Mika and Casper Lassenius. 2006. Subjective evaluation of software evolvability using code smells: An empirical study. *Emp. Softw. Eng.* 11, 3 (2006), 395–431.

Mäntylä Mika, Jari Vanhanen, and Casper Lassenius. 2003. A taxonomy and an initial empirical study of bad smells in code. In *Proceedings of the International Conference on Software Maintenance (ICSM'03)*. IEEE Computer Society, 381–384.

Mäntylä Mika. 2010. Empirical software evolvability—Code smells and human evaluations. In *Proceedings of the International Conference on Software Maintenance (ICSM'10)*. 1–6.

R. Mathieu, L. Pittard, and G. Anandalingam. 1994. Genetic algorithm based approach to bi-level linear programming. *Oper. Res.* 28, 1, 1–21.

Naouel Moha, Yann-Gaël Guéhéneuc, Laurence Duchien, and Anne F. Le Meur. 2010. DECOR: A method for the specification and detection of code and design smells. *IEEE Trans. Softw. Eng.* 36, 1 (2010), 20–36.

Akito Monden, Daikai Nakae, Toshihiro Kamiya, Shin-ichi Sato, and Ken-ichi Matsumoto. 2002. Software quality analysis by code clones in industrial legacy software. In *Proceedings of the 8th International Symposium on Software Metrics (METRICS'02)*. IEEE Computer Society, 87–94.

Matthew James Munro. 2005. Product metrics for automatic identification of "bad smell" design problems in Java source-code. In *Proceedings of the 11th IEEE International Software Metrics Symposium (METRICS'05)*. IEEE Computer Society.

Ali Ouni, Marouane Kessentini, Hauari A. Sahraoui, and Mounir Boukadoum. 2012. Maintainability defects detection and correction: A multi-objective approach. *J Autom. Softw. Eng.* 20, 1 (2013), 47–79.

Francesco Palomba, G. Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. 2013. Detecting bad smells in source code using change history information. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering (ASE'13)*. 268–278.

Foyzur Rahman, Christian Bird, and Premkumar Devanbu. 2012. Clones: What is that smell? *Emp. Softw. Eng.* 17, 4–5 (2012), 503–530.

Daniel Ratiu, Stéphane Ducasse, Tudor Gîrba, and Radu Marinescu. 2004. Using history information to improve design flaws detection. In *Proceedings of the 8th Euromicro Working Conference on Software Maintenance and Reengineering (CSMR'04)*. IEEE Computer Society, 223–232.

Arthur J. Riel. 1996. *Object-Oriented Design Heuristics* (1st ed.). Addison-Wesley, Boston, MA.

Mazeiar Salehie, Sen Li, and Ladan Tahvildari. 2006. A metric-based heuristic framework to detect object-oriented design flaws. In *Proceedings of the 14th IEEE International Conference on Program Comprehension (ICPC'06)*. 159–168.

Gilles Savard and Jacques Gauvin. 1994. The steepest descent direction for the nonlinear bilevel programming problem. *Oper. Res. Lett.* 15, 5 (1994), 265–272.

Ankur Sinha, Pekka Malo, and Kalyanmoy Deb. 2013a. Efficient evolutionary algorithm for single-objective bilevel optimization. KanGAL Report No. 2013003.

Ankur Sinha, Pekka Malo, Anton Frantsev, and Kalyanmoy Deb. 2013b. Multi-objective Stackelberg game between a regulating authority and a mining company: A case study in environmental economics. In *Proceedings of the IEEE Congress on Evolutionary Computation*. 478–485.

Dag I. K. Sjøberg, Aiko Fallas Yamashita, Bente Cecilie, Dahlum Anda, Audris Mockus, and Tore Dybå. 2013. Quantifying the effect of code smells on maintenance effort. *IEEE Trans. Softw. Eng.* 39, 8 (2013), 1144–1156.

Guilherme Travassos, Forrest Shull, Michael Fredericks, and Victor R. Basili. 1999. Detecting defects in object-oriented designs: Using reading techniques to increase software quality. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'99)*, A. Michael Berman (Ed.). ACM, New York, NY, 47–56.

Douglas A. Troy and Stuart H. Zweben. 1981. Measuring the quality of structured designs. *J. Syst. Softw.* 2, 2 (1981), 113–120.

E. Van Emden and L. Moonen. 2002. Java quality assurance by detecting code smells. In *Proceedings of the 9th Working Conference on Reverse Engineering (WCRE'02)*. IEEE Computer Society, 97–106.

Luís N. Vicente and Paul H. Calamai. 1994. Bilevel and multilevel programming: A bibliography review. *J. Global Optim.* 5, 3 (1994), 291–306.

Aiko F. Yamashita and Leon Moonen. 2013a. Do developers care about code smells? An exploratory survey. In *Proceedings of the 20th Working Conference on Reverse Engineering (WCRE'13)*. 242–251.

Aiko Yamashita and Leon Moonen. 2013b. To what extent can maintenance problems be predicted by code smell detection? An empirical study. *Inf. Softw. Technol.* 55, 12 (2013), 2223–2242.

Aiko F. Yamashita and Leon Moonen. 2012. Do code smells reflect important maintainability aspects? In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'12)*. IEEE Computer Society, 306–315.

Min Zhang, Tracy Hall, and Nathan Baddoo. 2011. Code bad smells: A review of current knowledge. *J. Softw. Maint. Evol.* 23, 3 (2011), 179–202.

Received November 2013; revised March, June 2014; accepted August 2014