Code-Smells Detection as a Bi-Level Problem

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

COIN Report Number 2013006
Computational Optimization and Innovation (COIN) Laboratory
Department of Electrical and Computer Engineering
Michigan State University, East Lansing, MI 48824, USA
http://www.egr.msu.edu/~kdeb/COIN.shtml

Code-Smells represent design situations that can affect the maintenance and evolution of software. They make a 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 to identify some code-smells based on a set of symptoms and also the high calibration effort to determine manually the thresholds value for each metric. In this paper, we propose, for the first time, to consider the generation of code-smells detection rules as a bi-level optimization problem. Bi-level optimization problems are a new 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 is truly a bi-level optimization problem, but due to lack of suitable solution techniques, it was attempted to solve using a single-level optimization problem in the past. In our adaptation here, the upper level problem generates a set of detection rules, combination of quality metrics, which maximizes the coverage of a 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 bi-level formulation is that the generation of detection rules is not limited to some code-smell examples identified manually by developers which are difficult to collect but it allows the prediction of new code-smells behavior that are different from those in the base of examples. The statistical analysis of our experiments over 31 runs on 9 open source systems shows that 7 types of code-smell were detected with an average of more than 85% in terms of precision and recall. The results confirm the outperformance of our bi-level proposal compared to state-of-art code-smells detection techniques.

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 total software costs [32]. In addition, it is shown that software maintainers spend around 60% of their time in understanding the code [1]. In particular, object oriented software systems need to follow some traditional set of design principles such as data abstraction, encapsulation, and modularity. However, some of these non-functional requirements can be violated by developers for many reasons like 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 greatly reduced by providing automatic or semi-automatic solutions to increase system's understandability, 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 [16][6] in

Author's addresses: Dilan Sahin, Marouane Kessentini and Slim Bechikh, University of Michigan, firstname@umich.edu. Kalyanmoy Deb, Michigan State University, kdeb@egr.msu.edu.

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 [6]. In this work, we focus on the detection of code-smells.

The vast majority of existing work in code-smells detection relies on declarative rule specification [42][44][43][35][33]. 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 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 [33]. When consensus exists, the same symptom could be associated to many code-smells 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, Search-Based Software Engineering [19] is used by the mean of genetic programming [17] to generate code-smells detection rules from a set of examples of code-smells identified manually by developers [33]. 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 inspect manually large systems. In addition, it is challenging to ensure the diversity of the code-smell examples to cover most of possible bad-practices. To this end, we propose, for the first time, to consider the code-smells detection problem as a bi-level one [12]. Bi-level optimization (BLOP) problems are a new class of challenging optimization problems, which contain two levels of optimization tasks [12][21][8]. 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, combination of quality metrics, which maximizes the coverage of a 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 overall problem appears as a bi-level optimization task, where for each generated detection rules, 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 suits it the most, taking the actions of the code-smells generation process (lower level or follower) into account. The main advantage of our bi-level formulation is that it is a natural way to detect code-smells in a software and that the generation of detection rules is not limited to some code-smell examples identified manually by developers which are difficult to collect but it allows the prediction of new code-smells behaviour that are different from those in the base of examples. This paper presents the very first study in the software engineering literature that proposes the use of bi-level optimization to address a critical software engineering problem [19].

We implemented our proposed bi-level approach and evaluated it on nine large open source systems [27][26][22][25][23][24][29][30] and found that, on average, the majority of seven types of code-smells were detected with more than 88% 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

[33][5] and a practical code-smell detection technique [44] based on existing corpuses [47][44][33][34][46][38].

The primary contributions of this paper can be summarized as follows:

(1) The paper introduces a novel formulation of the code-smells detection as a bi-level problem. To the best of our knowledge, this is the first paper in the literature that formulates a software engineering problem in a natural way as a bi-level one;
(2) The paper reports the results of an empirical study with an implementation of our bi-level approach. The obtained results provide evince to support the claim that our proposal is more efficient, in average, than existing techniques based on a benchmark of nine large open source systems.

The remainder of this paper 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; the different threats that affect our experimentations are described in Section 5; Section 6 is dedicated to related work. Finally, concluding remarks and future work are provided in Section 7.

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. The, we describe the principles of bi-level optimization.

2.1 Code-Smells

Code-smells, also called design anomalies or design defects, refer to design situations that adversely affect the software maintenance. As stated by [16], bad-smells are unlikely to cause failures directly, but may do it indirectly. In general, they make a system difficult to change, which may in turn introduce bugs. Different types of code-smells, presenting a variety of symptoms, have been studied in the intent of facilitating their detection and suggesting improvement solutions [38]. In [16], Beck defines 22 sets of symptoms of code smells. These include large classes, feature envy, long parameter lists, and lazy classes. Each code-smell type is accompanied by refactoring suggestions to remove it. Brown et al. [6] define another category of code-smells that are documented in the literature, and named anti-patterns. In our experiments, we focus on the five 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 store 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 code produced by non-experienced object-oriented developers.
- *Lazy Class (LC)*: A class that isn't 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 and hard to detect and fix based on a recent empirical study [15][47].

The code-smells detection process consists in finding code fragments that violate structure 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 [9]. In this paper, we use variations of these metrics and adaptations of procedural ones as well, namely Weighted Methods per Class (WMC), Response for a Class (RFC), Lack of Cohesion of Methods (LCOM), Cyclomatic Complexity (CC), Number of Attributes (NA), Attribute Hiding Factor (AH), Method Hiding Factor (MH), Number of Lines of Code (NLC), Coupling Between Object Classes (CBO), Number of Association (NAS), Number of Classes (NC), Depth of Inheritance Tree (DIT), Polymorphism Factor (PF), Attribute Inheritance Factor (AIF) and Number of Children (NOC).

In the following, we introduce some code-smells' detection issues and challenges. 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 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 is 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 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, the 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 the above issues make the process of manually defining code smells detection rules manually challenging. The generation of detection rules requires a huge examples set of code smells to cover most of possible bad-practices behavior. 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 [39]. In addition, it is challenging to ensure the diversity of the code-smell examples to cover most of possible bad-practices then using these examples to generate good quality of detection rules.

To address the above-mentioned challenges, we propose to consider code-smells detection problem as a bi-level optimization problem. The principles of bi-level are described in the next sub-section; then we present our adaptation in Section 3.

2.2 Bi-Level 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 latter are called Bi-Level Optimization Problems (BLOPs) [12]. In such problems, we find a nested optimization problem within the constraints of the outer optimization one. The outer optimization task is usually referred as the *upper level problem* or the *leader problem*. The nested inner optimization task is referred as the *lower level problem* or the *follower problem*, thereby referring the bi-level problem as

a leader-follower problem or as a Stackelberg game [cite]. 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 $L : \Re^n \times \Re^m \to \Re$ to be the leader problem and $f : \Re^n \times \Re^m \to \Re$ to be the follower one, analytically, a BLOP could be stated as follows:

$$\begin{aligned} &\underset{x_u \in X_U, x_l \in X_L}{Min} \; L(x_u, x_l) \\ &\text{Subject to } \begin{cases} x_l \in ArgMin\{f(x_u, x_l), g_j(x_u, x_l) \le 0, j = 1,..., J\} \\ G_k(x_u, x_l) \le 0, k = 1,..., K \end{cases} \end{aligned} \quad (1)$$

An example of an application of BLOP is the bi-level Multi-Depot Vehicle Routing Problem (MDVRP) where two different companies in supply chain are involved [2]. The leader transports goods from depots to retailers, thereby meeting to the retailer demands. The follower manages plants producing goods for the leader. The leader starts by deciding which depots should deliver goods. Then, the follower decides how to manufacture the goods. Both decisions influence the overall cost of each solution. More precisely, the leader controls a fleet of vehicles to deliver items from several depots to retailers, on the same principle as a classical MDVRP. The follower controls a set of plants, and has to produce the items and deliver them to the depots according to the demand of the retailers it serves, thus corresponding to a flow problem. The leader tries to minimize the total distance of his routes and the buying cost of the resources (depending on the lower-level decision). The follower minimizes the production cost and the distance travelled by the produced goods. The follower has to directly transport from plants to depots.

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

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 [7], branch-and-bound [31], complementary pivoting [4], descent methods [49], penalty function methods [2], trust region methods [11], 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 meta-heuristic algorithms which are mainly Evolutionary Algorithms (EAs). Recently, several EAs have demonstrated their effectiveness in tackling such type 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 by Sinha et al. [50], [51], Legillon et al. [40] and Koh [36].

To the best of our knowledge and based on recent surveys, there is no work in the software engineering literature that considers a software engineering problem as a bi-level one. In the next section, we describe our adaptation of bi-level optimization to the problem of code-smells detection.

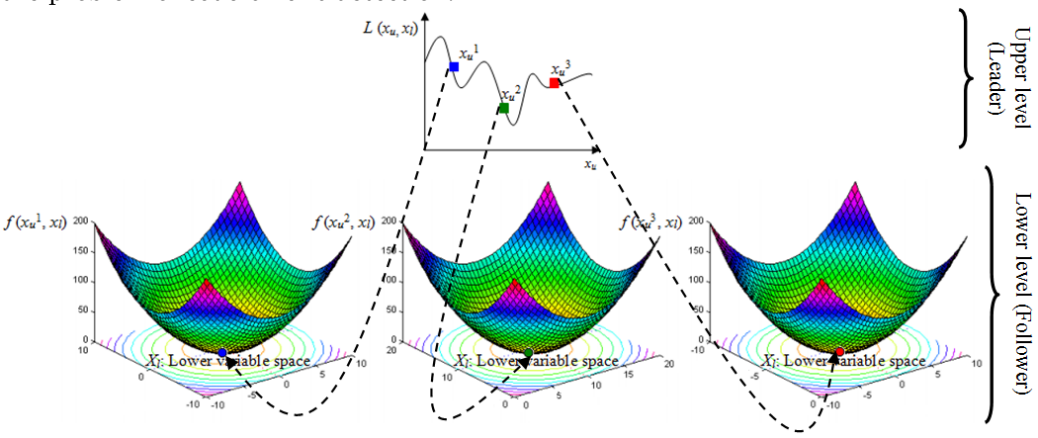


Figure 1. Illustration of the two levels of an exemplified bi-level single-objective optimization problem.

3. CODE-SMELLS DETECTION AS A BILEVEL OPTIMIZATION PROBLEM

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

3.1 Approach Overview

The concept of bi-level 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 is the generation of detection rules that can cover as much as possible the code-smells in the base of examples. The code-smells generation process has one objective which 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 rules generation process has higher control of the situation and decides which detection rules for the code-smells generation process to operate in. Therefore, in this framework, we observe that the detection rules generation process acts as a leader (the important output of the problem), and the code-smells generation process acts as a follower.

The overall problem appears as a bi-level optimization task, where for each generated detection rules, 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 that the best detection rules which suit it the most, taking the actions of the code-smells 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 multi-objective 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.

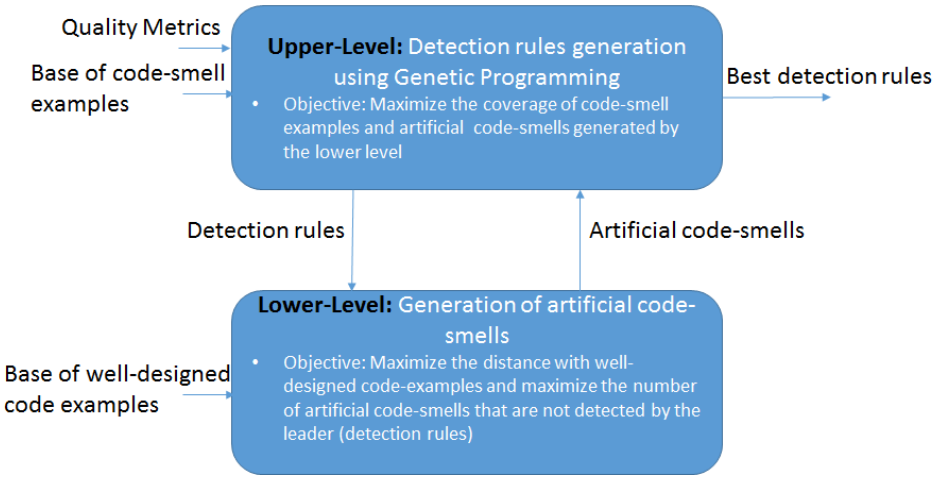


Figure 2. Approach overview

As described in Figure 2, the leader (upper level) uses knowledge from code-smells examples (input) to generate detection rules based on quality metrics (input). It takes as inputs a base (i.e. a set) of code smells' 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 metrics provided list, a combination of quality metrics (and their threshold values) to detect a specific code-smell. Consequently, a solution is a set of rules that best detect 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 as a blob smell:

R1: IF $NAD(c) \geq 10$ AND $NMD(c) \geq 20$, 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. Given the above detection rule, it is not obvious what set of diverse code-smells exist in the software. In the bi-level 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 maximize 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).

Next we describe our adaptation of bi-level optimization to the code-smells detection problem in more details.

3.2 Problem Formulation

The code-smells detection problem involves searching for the best metric combinations among the set of candidate ones, which constitute a huge search space.

A solution of our code-smells 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 bi-level 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, an 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 at the upper level is defined as follows:

$$Maximize \quad f_{upperLevel} = \frac{\frac{\Pr ecision(S, baseOfExamples) + \mathrm{Re}call(S, BaseOfExamples)}{2} + \frac{\#\det ectedArtificialCodeSmells}{\#artificialCodeSmells}}{2}$$

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 un-detected artificial code-smells that are generated and maximize also the distance with well-designed code-examples. Formally,

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

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 rules solution defined at the upper level.

Upper level algorithm: GPSmellDetection

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 N_1, Lower population size N_2, Upper number of generations G_1, Lower number of generations G_2
Output: Best detection rule BDR
Begin
  P_0 □ Initialization (N_1, *M*);
  For each DR_0 **in** P_0 **do** */*DR means Detection Rule*/*
    BCS_0 □ GASmellGeneration (DR_0, *D*, N_2, G_2); */*Call lower level routine*/*
    DR_0 □ Evaluation (DR_0, *B*, BCS_0);
  End For
  t □ 1;
  While ($t < G$) **do**
    Q_t □ Variation (P_{t-1});
    For each DR_t **in** Q_t **do** */*Evaluate each rule based on upper fitness function*/*
      DR_t □ UpperEvaluation (DR_t, *B*);
    End For
    For each of the best *nbs* **rules** DR_t **in** Q_t **do** */*Only nbs rules are used to*/*
      BCS_t □ GASmellGeneration (DR_t, *D*, N_2, G_2); */*call lower level routine*/*

```
17.          DR_t □ EvaluationUpdate (DR_t, BCS_t);  /*Update based on lower level*/
18.        End For
19.        U_t □ P_t ∪ Q_t;
20.        P_t+1 □ EvironmentalSelection (N_1, U_t);
21.        t □ t+1;
22.      End While
23.      BDR □ FittestSelection (P_t);
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 smell BCS
03.  Begin
04.    P_0 □ Initialization (N, D);
05.    P_0 □ Evaluation (P_0, D, UDR);  /*Evaluation depends of UDR*/
06.    t □ 1;
07.    While (t < G) do
08.      Q_t □ Variation (P_t-1);
09.      Q_t □ Evaluation (Q_t, D, UDR);  /*Evaluation depends of UDR*/
10.      U_t □ P_t ∪ Q_t;
11.      P_t+1 □ EvironmentalSelection (N, U_t);
12.      t □ t+1;
13.    End While
14.    BCS □ FittestSelection (P_t);
15.  End
```

(b)

Figure 3. Pseudo-code of the bi-level adaptation for code-smells detection.

3.3 Solution Approach

The solution approach proposed in this paper lies within the SBSE field. As noted by Harman et al. [19], a generic algorithm like bi-level optimization cannot be used 'out of the box' – it is necessary to define problem-specific genetic operators to obtain the best performance. To adapt bi-level optimization to our code-smells 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 in the coming paragraphs.

3.3.1 Solution Representation

One key issue when applying a search-based technique is to find a suitable mapping between the problem to solve and the techniques to use, i.e., detecting code-smells.

For the *upper-level* optimization problem, a genetic programming (GP) algorithm is used [18]. 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-smells 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:

(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}.

The set of candidates 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 by 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 reducing the computational complexity. When several predicates of the same type exist, we order them according to their parameters.

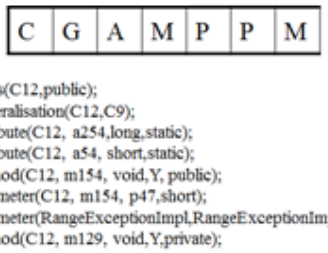


Figure 4. Solution representation: vector (GA) to generate artificial code-smell.

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-smells 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. Figures 4 shows an example of a generated code-smell composed by 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 selection. 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. Concerning parent selection, once the population individuals are evaluated, we select the $|P|/2$ best individuals of the population P to fulfill the mating pool, which 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 population P and Q into U (with $|U| = |P| + |Q| = 2|P|$), and then the population P for the next generation is composed by the $|P|$ fittest individuals from U. By doing this, we ensure that we do not encourage the survival of better solutions. We can say that this environmental selection is elitist, which is a desired property in modern EAs [49]. 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 sub-tree are replaced by a new randomly generated sub-tree.

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 sub-tree is picked on each one. Then crossover swaps the nodes and their relative sub-trees 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 to create two offspring o1 and o2 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. EMPIRICAL STUDY

In order to evaluate our approach for detecting code smells using the proposed bi-level optimization (BLOP) approach, we conducted a set of experiments based on different versions of large open source systems [27][26][22][25][23][29][30]. Each experiment is repeated 31 times, and the obtained results are subsequently statistically analyzed with the aim to compare our bi-level proposal with a variety of existing code-smells detection approaches [33][5][44]. In this section, we first present our research questions and then describe and discuss the obtained results. Finally, we present various threats to the validity of our experiments.

4.1 Research Questions

We defined five research questions that address the applicability, performance comparison with existing refactoring approaches, and the usefulness of our bi-level code-smells detection approach. The six 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 an intelligent search method thus we can conclude that our problem formulation is not adequate. Since outperforming a random search is not sufficient, the next four questions are related to performance of BLOP and the comparison with the state-of-the-art code-smells detection approaches.

RQ2: How does BLOP perform to detect different types of code-smells? It is important to quantitatively assess the completeness and correctness of our code-smell detection approach when applied in real-world settings.

RQ3.1: How do BLOP perform compared to existing search-based code-smells detection algorithms? Our proposal is the first work that treats a software engineering problem as a bi-level problem. A comparison with existing search-based code-smells detection approaches [33][41][5] is helpful to evaluate the benefits of the use of bi-level approach in the context of code-smell detection.

RQ3.2: How does BLOP perform compared to the existing refactoring approaches not based on the use of metaheuristic search? While it is very interesting to show that our proposal outperforms existing search-based code-smells detection approaches, developers will consider our approach useful, if it can outperform other existing tools that are not based on optimization techniques [44][46].

RQ4: How does our bi-level 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 time-frame.

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 large and medium sized open source Java projects: JFreeChart [27], GanttProject [26], ApacheAnt [22], Nutch [25], Log4J [23], Lucene [25], Xerces-J [29] and Rhino [30]. Table 1 presents the list and some relevant statistics of the softwares for our code-smell detection purpose.

Table 1. 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	1024	163	255
ApacheAnt	v1.7.0	1839	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

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 1 provides some descriptive statistics about these nine programs. We selected these systems for our validation because they range from medium to large-sized open source projects 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 [47][44][33][34][46][38].

In the nine studied open source systems, the seven following code-smell types were identified manually [47][44][33][34]: *Blob* (one large class monopolizes the behavior of a system or part of it, and the other classes primarily encapsulate data); *Data Class* (a class that contains only data and performs no processing on these data); *Spaghetti Code* (code with a complex and tangled control structure); *Feature Envy* (Feature Envy occurs when a method is more interested in the features (methods and fields) of other classes than its own); *Lazy Class* (a class that isn't doing enough to pay for itself), *Long Parameter List* and *Functional Decomposition* (when a class performs a single function, rather than being an encapsulation of data and functionality). We chose these code smell types in our experiments because they are the most frequent ones detected and corrected in recent studies and existing corpuses [47][44][33][34].

4.3 Evaluation Metrics Used

To assess the accuracy of our approach, we compute two measures, *precision (PR)* and *recall (RE)*, 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 (that is, 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, whereas a higher value is better than a lower one. We performed a 9-fold cross validation thus we removed the expected code-smells to detect in the system to evaluate from the base of code-smell examples when executing our BLOP algorithm, then precision and recall scores are calculated automatically based on a comparison between the detected code-smells and expected ones. Thus, one open source project is evaluated by using the remaining systems as a base of code-smells' 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 [3] with a 95% confidence level (= 5%). 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 (≤ 0.05) means that we accept H_1 and we reject H_0. However, a p-value that is strictly greater than (> 0.05) means the opposite. In this way, we could decide whether the outperformance of BLOP over one of each of the others 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 rules solution was compared for statistically significant differences with BLOP using *PR, RE* 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-smells detection approaches – GP [33] and a co-evolutionary approach [5]. In [33], Kessentini et al. used genetic programming (GP) to generate detection rules from manually collected code-smell examples which corresponds to the upper level of our approach (without a lower level). Thus, the generation of artificial code-smells is not considered but only the coverage of manually identified examples. In [5], we 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 from the solutions in the first population which is one of the main differences with our bi-level extension. We considered the three metrics *PR, RE* and *CT* to compare bi-level with these search-based techniques based on 31 independent executions. To answer RQ3.2, we compared our results with DECOR [44]. Moha et al. [44] 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 antipatterns 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 *RE* metrics. To answer the last question RQ4 we evaluated the execution time *CT* required by our BLOP proposal based on different scenarios (parameters setting).

4.5 Parameter tuning and setting

An often-omitted aspect in metaheuristic search is the tuning of algorithm parameters [4]. In fact, parameter setting influences significantly the performance of a search algorithm on a particular problem. For this reason, for each search algorithm and for each system (cf. Table 1), 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 large number of evaluations as stopping criterion since our bi-level approach requires involves two levels of optimization. Each algorithm was executed 51 times with each configuration and then comparison between the configurations was performed based on precision and recall using the Wilcoxon test. Table 2 reports the best configuration obtained for each couple (algorithm, system).

Table 2. 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

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 bi-level approach, both lower-level and upper-level EAs are run each with a population of 30 individuals and 50 generations. 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 bi-level approach, only *nbs*% of the best upper-level population members are allowed to call the lower-level optimization algorithm. Based on a parametric study, a value of 10% for *nbs* is found to be adequate empirically in our experiments. The *nbs* parametric study will be discussed in the next section (RQ4). For our experiment, we generated up to 125 artificial code-smells from deviation with JHotDraw (about a quarter of the number of examples) [28]. JHotdraw [28] was chosen as an example of reference code because it contains very few known code-smells. In fact, previous work [56] 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

We do not focus too much in answering the first research question, RQ1, which involves comparing our BLOP approach with random search. The remaining research questions will reveal more about the performance, insight, and usefulness of our approach. Table 3 confirms that BLOP is better than random search based on the two metric *PR* and *RE* on all the nine open source systems. The Wilcoxon rank sum test showed that in 31 runs BLOP results were significantly better than random search.

Table 3. The significantly best algorithm among random search, BLOP, GP and Co-Evol over 31 independent runs. "No. Sign." Means no method is significantly better than another.

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

Table 4 describes also 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 is 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 bi-level formulation surpasses the performance of random search thus our formulation is adequate (this answers RQ1).

Table 4. Median PR and RE values on 31 runs for BLOP, random search (RS), GP [33] and Co-Evol [5]. The results were statistically significant on 31 independent runs using the Wilcoxon rank sum test with a 99% confidence level (α < 1%).

System	*PR-BLOP*	*PR-GP*	*PR-Co-Evol*	*PR-RS*	*RE-BLOP*	*RE-GP*	*RE-Co-Evol*	*RE-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)

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 4 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 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 is can confirm that the list of returned code-smells did not contain high number of false positive thus developer will not waste a lot of their time for manual inspections. We found similar observation 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 lowest recall scores were respectively 95% (Rhino) and 89% (GanttProject). An interesting observation that the performance of bi-level in terms of PR and RE is independent from the size and number of code-smells in the system to analyze. The PR and RE scores of ApacheAnt (largest system) are among the highest ones with more than 90% which are better than the detection results of smaller system such as GanttProject. A more qualitative evaluation is presented in Figure 5 illustrating the box plots obtained for the PR and RE 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 RE values for BLOP are the best ones.

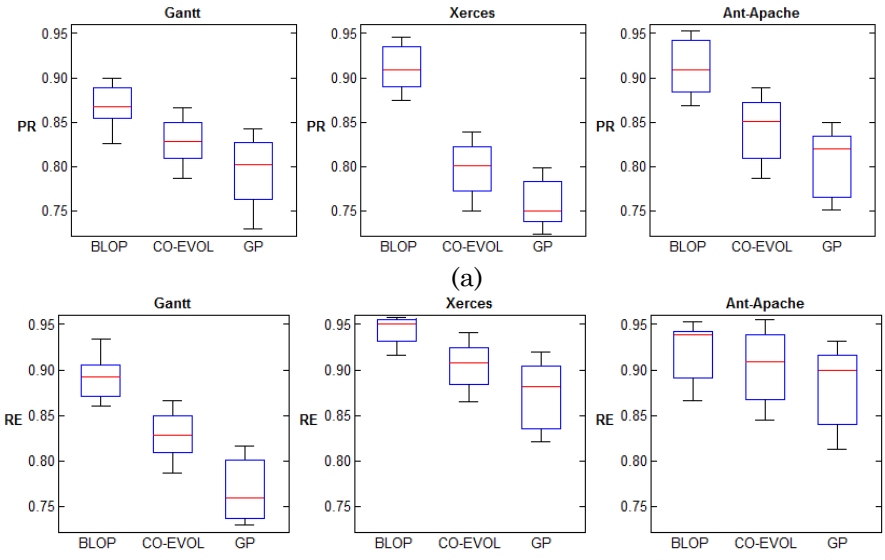
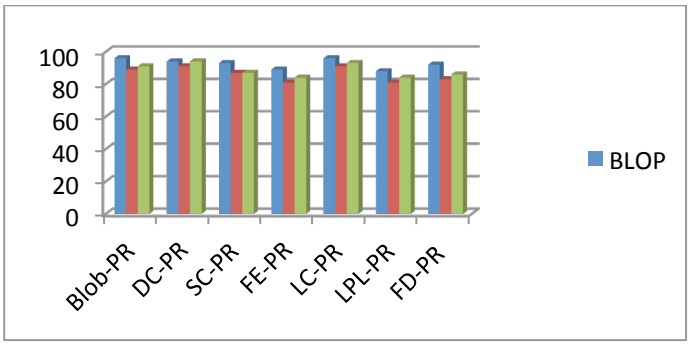


Figure 5. Box plots on three different systems (Gantt: small, Xerces: medium, Ant-Apache 1.7.0: large) of: (a) precision values, and (b) recall values.

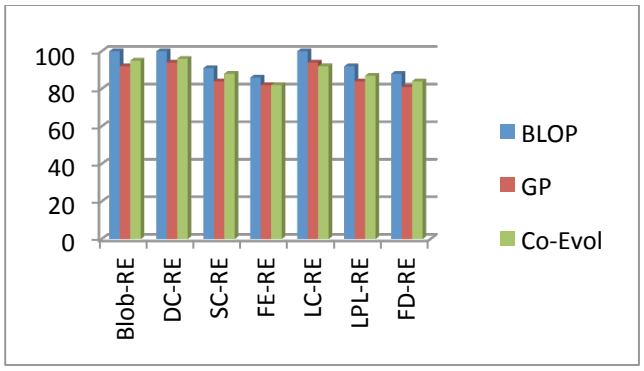
We noticed that our technique does not have a bias towards the detection of specific code-smells types. As described in Figure 6, in all systems, we had an almost equal distribution of each code-smell types. Having a relatively good distribution of code-smells is useful for a quality engineer. Overall, 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-smells 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 a 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 the seven types of considered code-smells (RQ2).



(a)



(b)

Figure 6. Average PR (a) and RE (b) scores for every code-smells type over 31 runs on the different systems.

4.6.3 Results for RQ3

In this section, we compare our BLOP adaptation to the current, state-of-the-art refactoring approaches. To answer RQ3.1, we compared BLOP to two other existing search-based techniques: GP [33] and Co-Evol [5]. Table 4 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 (RE). However, as it 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 RE scores are significant using BLOP comparing to GP and Co-Evol. A more qualitative evaluation is presented in Figure 5 illustrating the box plots obtained for the PR and RE metrics on three different projects GanttProject (small), Xerces-J (medium) and Ant-Apache (large). It is clear from the box plots presented in Figure 5 that the outperformances of BLOP comparing to GP and Co-Evol in all the three different systems based on the statistical analysis of 31 independent runs. For GP, this is 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 fastidious task to collect them (most of 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.

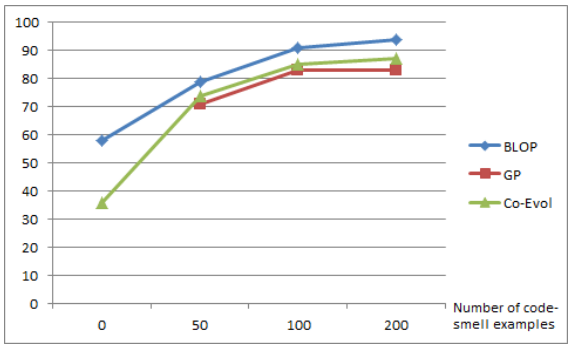


Figure 7. The impact of number of code-smell examples on the quality of the results (PR on Xerces-J).

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 7 shows that BLOP requires a low number of manually identified code-smells to provide good detection rules with reasonable precision and recall scores. GP and Co-Evol require higher number of code-smell examples than BLOP to generate good code-smells 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 [28] 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.

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-smells detection work – a GP [33] and a co-evolutionary GA [5].

Since it is not sufficient to compare our proposal with only search-based work, we compared the performance of BLOP with DECOR [44]. 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 8 summarizes the results of the precision and recall obtained on the above-mentioned 5 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. For example, the average precision to detect functional decompositions using DECOR is lower than 25% however we can detect this type of code-smells with more than 80% of precision. The same observation for the spaghetti-code (SC), BLOP is able to 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 is can be explained by the high calibration effort required to define manually 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, in average, but it is an acceptable recall average which is higher than 80%. To conclude, our BLOP adaption also outperforms, in average, an existing approach not based on meta-heuristic search (RQ4).

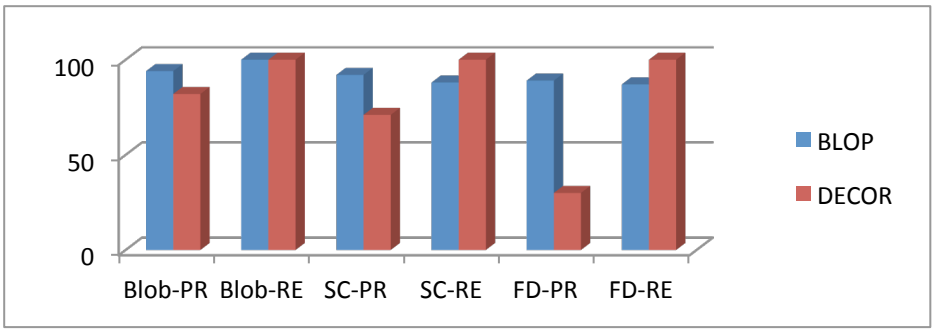


Figure 8. The average precision and recall scores of BLOP and DECOR obtained on GanttProject, Nutch, Log4J, Lucene and Xerces-J based on three code-smell types (Blob, SC and FD).

4.6.4 *Results for RQ3*

Since our proposal is based on bi-level 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 at each iteration an optimization algorithm is executed at the lower level. To reduce the computational complexity of our BLOP adaptation, we selected only best solutions (*nbs*%) at the upper level, after an evaluation based on the coverage of manually identified code-smell examples, to update their fitness evaluation 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 3 GHz processors and 4 GB RAM. We recall that all algorithms were run for 750 000 evaluations. This allows us to make a fair comparison between CPU times. Overall, 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 detection rules are generated only one time then used to detect code-smells. There is no need to execute BLOP again except the case that the base of examples was 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 9 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 represents a good trade-off between the quality of detection solutions and the execution time. As described in Figure 9, the PR and RE 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 observations were found on the remaining systems.

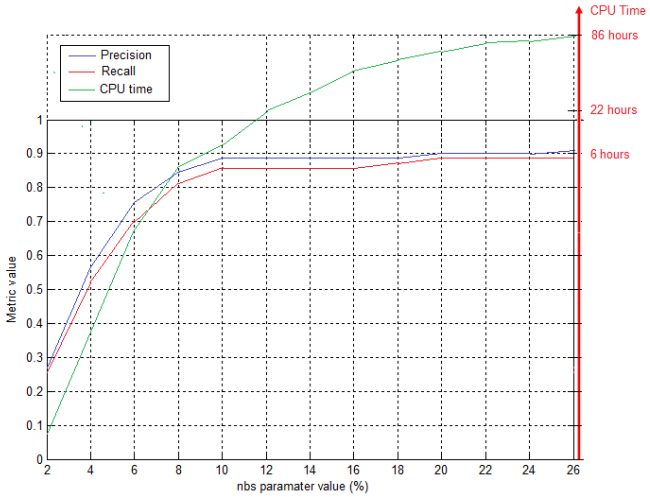


Figure 9. The impact of the number of selected solutions at upper level on the quality of the results (*PR, RE* and *CT*) using JFreeChart v1.0.9.

Figure 10 shows the number of evaluations required to generate good code-smells 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 can reach faster that threshold value of *f-measure (0.7)*. We found that our bi-level adaptation required a fewer number of evaluation than Co-Evol and GP to generate good code-smells detection solutions. In fact, after around 325000 evaluations BLOP generated detection rules that have 0.7 as *f-measure* value on Xerces-J. Co-Evol requires at least more than 480000 evaluations to reach similar solution quality and GP needs more than 560000 evaluations to generate similar detection solution. Thus, we can conclude that the lower level helped the upper level to generate quickly good quality of detection solutions by producing in an intelligent manner efficient artificial code-smells. Although the fact that BLOP needs higher execution time than Co-Evol and GP as described in Figure 9, it is clear from Figure 10 that the good solutions provided by a single-level approach can be reached quickly by our bi-level adaptation.

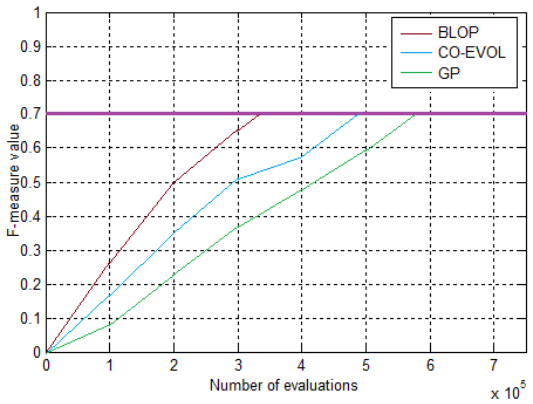


Figure 10. The number of evaluations required by the different algorithms (BLOP, Co-Evol and GP) to reach acceptable results (*f-measure=0.7*) using Xerces-J v 2.7.0.

5. 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 relation 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 bi-level techniques for code-smells detection. For that reason, we compare our proposal with two other search-based techniques [33][5] and DECOR [44]. Another threat to construct validity arises because, although we considered 7 well-known code-smell types, we did not evaluate the performance of our proposal with other code-smell types. In future work, we plan to use additional code-smell types and evaluate the results. Another construct threat can 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.

We take into consideration the internal threats to validity in the use of stochastic algorithms since our experimental study is performed based on 31 independent simulation runs for each problem instance and the obtained results are statistically analyzed by using the Wilcoxon rank sum test [3] with a 95% confidence level (= 5%). However, 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 additional experiments to evaluate the impact of upper and lower levels' parameters on the quality of the results.

External validity refers to the generalizability of our findings. In this study, we performed our experiments on nine different widely-used open-source systems belonging to different domains and with different sizes, as described in Table 1. However, we cannot assert that our results can be generalized to industrial applications, other programming languages, and to other practitioners. Future replications of this study are necessary to confirm the generalizability of our findings.

6. 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.

6.1 Interactive-based approaches

In [16], Fowler and Beck have described a list of design smells which may exist in a program. They suggested that software maintainers should manually inspect the program to detect existing design smells. In addition, they specify particular refactorings for each code-smell type. Travassos et al. [52] have also proposed a manual approach for detecting code-smells in object-oriented designs. The idea is to create a set of "reading techniques" which 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. So that, each reading technique helps the maintainer focusing on some aspects of the design, in such a way that an inspection team applying the entire family should achieve a high degree of coverage of the design code-smells. In addition, in [10], another proposed approach is based on violations of design rules and guidelines. This approach consists of analyzing legacy code, specifying frequent design problems as queries and locating the occurrences of these problems in a model derived from the source code. However, the majority of the detected problems were simple ones, since it is based on simple conditions with particular threshold values. As a consequence, this approach did not address complex design code-smells.

The high rate of false positives generated by the above-mentioned approaches encouraged other teams to explore semi-automated solutions. These solutions took the form of visualization-based environments. The primary goal is to take advantage of the human capability to integrate complex contextual information in the detection process. Kothari et al. [37] present a pattern-based framework for developing tool support to detect software anomalies by representing potential code-smells with different colors. Dhambri et al. [13] have proposed a visualization-based approach to detect design anomalies by automatically detecting some symptoms and letting others to human analyst. The visualization metaphor was chosen specifically to reduce the complexity of dealing with a large amount of data. Although visualization-based approaches are efficient to examine potential 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 [13] are based on manual and human inspection, they still, not only, slow and time-consuming, but also subjective.

The main disadvantage of exiting 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. Another important issue is that locating code-smells manually has been described as more a human intuition than an exact science.

6.2 Symptom-based detection

Moha et al. [44] 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. 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. [35] extending the DECOR approach [44], a symptom-based approach, to support uncertainty and to sort the code-smell candidates accordingly. This approach is managed by Bayesian belief network (BBN) that implements the detection rules of DECOR. The detection outputs are probabilities that a class is an occurrence of a code-smell type, i.e., 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 et al. [45] have proposed description and symptoms-based approach using a precise definition of bad smells from the informal descriptions given by the originators Fowler and Beck [16]. The characteristics of design code-smells have been used to systematically define a set of measurements and interpretation rules for a subset of design code-smells as a template form. This template consists of three main parts: a code smell name, a text-based description of its characteristics, and heuristics for its detection.

In another category of work, quality metrics were used to Marinescu [42] have proposed a mechanism called "detection strategy" for formulating metrics-based rules that capture deviations from good design principles and heuristics. Detection strategies allow to a maintainer to directly locate classes or methods affected by a particular design code-smell. As such, Marinescu has defined detection strategies for capturing around ten important flaws of object-oriented design found in the literature. After his suitable symptom-based characterization of design code-smells, Salehie et al. [48] proposed a metric-based heuristic framework to detect and locate object-oriented design flaws similar to those illustrated by Marinescu [42]. It is accomplished by evaluating design quality of an object-oriented system through quantifying deviations from good design heuristics and principles by mapping these design flaws to class level metrics such as complexity, coupling and cohesion by defining rules. Erni et al. [14] introduce the concept of multi-metrics, as an n-tuple of metrics expressing a quality criterion (e.g., modularity). Unfortunately, multi-metrics neither encapsulate metrics in a more abstract construct, nor do they allow a flexible combination of metrics.

In general, the effectiveness of combining metric/threshold is not obvious. That is, for each code-smell, rules that are expressed in terms of metric combinations need a significant calibration effort to find the fitting threshold values for each metric. Since there exists no consensus in defining design smells, different threshold values should be tested to find the best ones.

6.3 Search-based approaches

Our approach is inspired by contributions in the domain of Search-Based Software Engineering (SBSE) [19][20]. SBSE uses search-based approaches to solve optimization problems in software engineering. Once a software engineering task is framed as a search problem, many search algorithms can be applied to solve that problem. In [34], we have proposed another approach, based on search-based techniques, for the automatic detection of potential code-smells in code. The detection is based on the notion that the more code deviates from good practices, the more likely it is bad. In another work [33], 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 [19], the use of bi-level optimization is still very limited in software engineering. Indeed, this work represents the first attempt to use bi-level optimization to address a software engineering problem.

7. CONCLUSIONS AND FUTURE WORK

In this paper, we have addressed the problem of the absence of consensus in code-smells 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 bi-level 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 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. Firstly, the main problem when using bi-level optimization in SE 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 significantly the required number of function evaluations. Secondly, the idea of bi-level optimization seems interesting for several other SE problems. It would be challenging to model and then solve other interesting SE problems in a bi-level manner. We are currently working on extending our work by proposing a bi-level approach for the correction of code-smells.

REFERENCES

[1] Abran, A. and Hguyenkim, H. 1993. Measurement of the Maintenance Process from a Demand-Based Perspective*, Journal of Software Maintenance: Research and Practice*, *Vol 5, no 2.*

[2] Aiyoshi, E., and Shimizu, K. 1981. Hierarchical decentralized systems and its new solution by a barrier method. *IEEE Transactions on Systems, Man, and Cybernetics*, *11, 444–449.*

[3] Arcuri, A. and Fraser, G. 2013. Parameter tuning or default values? An empirical investigation in search-based software engineering*. Empirical Software Engineering. vol. 18.*.

[4] Bialas, W., Karwan, M., Shaw, J. 1980. A parametric complementarity pivot approach for two-level linear programming. *Technical Report 80-2, State University of New York at Buffalo, Operations Research Program.*

[5] Boussaa, M., Kessentini, W., Kessentini, M., Bechikh, S., Chikha, S. B. 2013. Competitive Coevolutionary Code-Smells Detection. 5th IEEE Symposium on Search-based Software Engineering SSBSE 2013, pp. 50-65.

[6] Brown, W. J., Malveau, R. C., Brown, W. H., and Mowbray, T. J. 1998. Anti Patterns: Refactoring Software,Architectures, and Projects in Crisis. *John Wiley and Sons, 1st edition* (March 1998)

[7] Candler, W., Townsley, R. 1982. A linear two-level programming problem. Computers and Operations Research, 9(1):59–76.

[8] 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, New Mexico, USA.

[9] Chidamber, S.R., Kemerer, C.F. 1994. A metrics suite for object-oriented design. *IEEE Trans. Softw. Eng.*20(6), 293–318,.

[10] Ciupke, O. 1999. Automatic detection of design problems in object-oriented reengineering, D. Firesmith (Ed.), Proceeding of 30th Conference on Technology of Object-Oriented Languages and Systems, IEEE Computer Society Press, pp. 18–32.

[11] Colson, B., Marcotte, P., and Savard, G. 2005. A trust-region method for nonlinear programming: algorithm and computational experience. *Computational Optimization and Applications*, 30.

[12] Colson, B., Marcotte, P., Savard, G.2005. Bilevel programming: *A survey*. 4OR, 3(2):87–107.

[13] Dhambri, K., Sahraoui, H.A., Poulin, P. 2008. Visual detection of design anomalies. *CSMR. IEEE*, pp. 279–283,.

[14] Erni, K. and Lewerentz, C. 1996. Applying design metrics to object-oriented frameworks, *Proc. IEEE Symp. Software Metrics.*

[15] Fenton, N., and Pfleeger, S. L. 1997. Software Metrics: A Rigorous and Practical Approach, 2nd ed. London, UK*: International Thomson Computer Press*,.

[16] Fowler, M., Beck, K., Brant, J., Opdyke, W. and Roberts, D. 1999. Refactoring – Improving the Design of Existing Code, 1st ed. Addison-Wesley.

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

[18] Goldberg, D. E. 1989. Genetic algorithms in search, optimization and machine learning. Addison Wesley, ISBN 0-201-15767-5.

[19] Harman, M., Mansouri, S. A. and Zhang, Y. 2012. Search-based software engineering: Trends, techniques and applications. *ACM Comput. Surv.* 45, 61 pages.

[20] Harman, M.: The current state and future of search based software engineering, in Future of Software Engineering 2007, L. Briand and A. Wolf, Eds. IEEE Computer Society Press, 2007, pp. 342–357.

[21] Herminia I. Calvete, Carmen Galé. 2010. A Multiobjective Bilevel Program for Production-Distribution Planning in a Supply Chain. *Multiple Criteria Decision Making for Sustainable Energy and Transportation Systems*, pp. 155–165.

[22] http://ant.apache.org/

[23] http://logging.apache.org/log4j/2.x/

[24] http://lucene.apache.org/

[25] http://nutch.apache.org/

[26] http://www.ganttproject.biz/

[27] http://www.jfree.org/

[28] http://www.jhotdraw.org/

[29] http://xerces.apache.org/

[30] https://developer.mozilla.org/en-US/docs/Rhino

[31] Jonathan F. Bard, James E. Falk. 1982. An explicit solution to the multi-level programming problem. *Computers and Operations Research*, 9(1):77–100.

[32] Keith H. Bennett and Vclav T. R. 2000. Software maintenance and evolution: a roadmap. *In Proceedings of the Conference on The Future of Software Engineering* (ICSE '00). ACM, New York, NY, USA, 73-87.

[33] Kessentini, M., Kessentini, W., Sahraoui, H., Boukadoum, M., and Ouni, A. 2011. Design Defects Detection and Correction by Example. *19th IEEE International Conference on Program Comprehension (ICPC)*, (22-24 June 2011), pp 81-90, Kingston- Canada.

[34] Kessentini, M., Vaucher, S., Sahraoui, H. 2010. Deviance from Perfection is a Better Criterion than Closeness to Evil when Identifying Risky Code, *25th IEEE/ACM International Conference on Automated Software Engineering (ASE)*.

[35] Khomh, F., Vaucher, S., Guéhéneuc, Y.-G., Sahraoui, H. 2009. A Bayesian approach for the detection of code and design smells. *In: Proc. of the ICQS'09* .

[36] Koh, A.2013. A Metaheuristic Framework for Bi-level Programming Problems with Multi-disciplinary Applications. In: Metaheuristics for Bi-level Optimization. *Studies in Computational Intelligence, 482 . Springer, Berlin, Heidelberg , 153–87* . ISBN 3642378374

[37] Kothari, S.C., Bishop, L., Sauceda, J., Daugherty, G. 2004. A pattern-based framework for software anomaly detection. *Softw. Qual. J.*12(2), 99–120,.

[38] Mansoor, U., Kessentini, M., Bechikh, S. and Deb, K. 2013. Code-Smells Detection using Good and Bad Software Design Examples. Technical Report No. 2013005, University of Michigan.

[39] Langelier, G., Sahraoui, H.A., Poulin, P. 2005. Visualization-based analysis of quality for large-scale software systems, T. Ellman, A. Zisma (Eds.), *Proceedings of the 20th International Conference on Automated Software Engineering,* ACM Press.

[40] Legillon, F., Liefooghe, A., Talbi, E.G. 2012. CoBRA: A cooperative coevolutionary algorithm for bi-level optimization. *IEEE Congress on Evolutionary Computation*

[41] Liu, H., Yang, L., Niu, Z., Ma, Z., Shao, W. 2009. Facilitating software refactoring with appropriate resolution order of bad smells. *In: Proc. of the ESEC/FSE* '09, pp. 265–268

[42] Marinescu, R. 2004. Detection strategies: metrics-based rules for detecting design flaws, Proceedings of the 20th International Conference on Software Maintenance, IEEE Computer Society Press, pp. 350–359.

[43] Marinescu, R. Detection strategies: metrics-based rules for detecting design flaws. *In: Proc. of ICM'04,* pp. 350–359

[44] Moha, N., Guéhéneuc, Y.-G., Duchien, L., Le Meur, A.-F. 2010. DECOR: A Method for the Specification and Detection of Code and Design Smells. IEEE Trans. Software Eng. 36, 20-36, 2010.

[45] Munro, M. J. 2005. Product Metrics for Automatic Identification of "Bad Smell" Design Problems in Java Source-Code, *Proc. 11th Internatinal Software Metrics Symp., F. Lanubile and C. Seaman, eds.,*.

[46] Ouni, A., Kessentini, M., Sahraoui, H. and Boukadoum, M. 2012. Maintainability Defects Detection and Correction: A Multi-Objective Approach. *J. of Autmated Software Engineering, Springer*.

[47] Palomba, F., Bavota, G., Di Penta, M., Oliveto, R., De Lucia, A. and Poshyvanyk, D. 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 2013), to appear.

[48] Salehie, M., Li, S., Tahvildari L. 2006. A Metric-Based Heuristic Framework to Detect Object-Oriented Design Flaws, in Pro-ceedings of the 14th IEEE ICPC'06.

[49] Savard, G., and Gauvin, J. 1994. The steepest descent direction for the nonlinear bilevel programming problem. Operations Research Letters, 15, 265–272.

[50] Sinha, A., Malo, P. and Deb, K. 2013. Efficient Evolutionary Algorithm for Single-Objective Bilevel Optimization. KanGAL Report No. 2013003.

[51] Sinha, A., Malo, P., Frantsev, A., Deb, K. 2013. Multi-objective Stackelberg game between a regulating authority and a mining company: A case study in environmental economics. *IEEE Congress on Evolutionary Computation 2013:* 478–485.

[52] Travassos, G., Shull, F., Fredericks, M., Basili, V.R. 1999. Detecting defects in object-oriented designs: using reading techniques to increase software quality, *Proceedings of the 14th Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ACM Press, pp. 47–56,.