Code-Smells Detection using Good and Bad Software Design Examples

Usman Mansoor[1], Marouane Kessentini[1], Slim Bechikh[1], and Kalyanmoy Deb[2]
[1]University of Michigan,
Michigan, USA
firstname@umich.edu
[2]Michigan State University,
Michigan, USA
kdeb@egr.msu.edu

Computational Optimization and Innovation (COIN) Laboratory
http://www.egr.msu.edu/~kdeb/reports.shtml

COIN Report Number 2014009

Abstract. Code-Smells refer to design situations that may adversely affect the maintenance of software. They make a system difficult to change, which may in turn introduce bugs. Code-smells are detected, in general, using quality metrics that formalize some symptoms based on detection rules. We propose in this work to consider the problem of code-smells detection as a multi-objective problem where examples of code-smells and well-designed code are used to generate detection rules. To this end, we use multi-objective genetic programming (MOGP) to find the best combination of metrics that maximizes the detection of code-smells examples and minimizes the detection of well-designed code examples. We evaluated our proposal on seven large open source systems and found that, on average, most of the different three code-smells types were detected with an average of 86% of precision and 91% of recall. Statistical analysis of our experiments over 51 runs shows that MOGP performed significantly better than state-of-the-art code-smell detectors.
Keywords: Search-based software engineering, refactoring, software metrics.

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. In addition, it is shown that software maintainers spend around 60% of their time in understanding the code [4]. This high cost could potentially be greatly reduced by providing automatic or semi-automatic solutions to increase their understandability, adaptability and extensibility to avoid bad-practices. Thus, there has been much research focusing on the study of bad design practices, also called code-smells, design defects, anti-patterns or anomalies [5] 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, these code-smells refer to design situations that may adversely affect the maintenance of software. They make a system difficult to change, which may in turn introduce bugs. In this paper, we focus on the detection of code-smells.

The majority of studies related to code-smells detection relies on declarative rule specification [6][7][8][9]. 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. In addition, the translation of symptoms into rules is not obvious due to the ambiguities related to the definition of some code-smells. In fact, the same symptom could be associated to many code-smells types, which may compromise the precise identification of code-smell types.

To address the above-mentioned limitations, we propose in this paper a multi-objective search-based approach for the generation of code-smells detection rules from code-smells and well-designed examples. The process aims at finding the combination of quality metrics, from an exhaustive list of metric combinations, that: 1) maximizes the coverage of a set of code-smell examples collected from different systems; and 2) minimizes the detection of examples of good-design practices. To this end, a multi-objective genetic programming (MOGP) [1] is used to generate the code-smells detection rules that find trade-offs between the two mentioned objectives. MOGP is a powerful evolutionary metaheuristic which extends the generic model of learning to the space of programs. The primary contributions of this paper can be summarized as follows:

- The paper introduces a novel formulation of the code-smells detection problem as a multi-objective problem that takes into account both good and bad design practices to generate detection rules. To the best of our knowledge, and based on a recent survey [9], this is the first work to use *multi-objective evolutionary algorithms* for code-smells detection.
- The paper gives an evaluation of our multi-objective approach on seven open source systems [10][11][12][13][14] using existing benchmarks [15] to detect three different types of code-smells. We report the statistical analysis of MOGP results on the efficiency and effectiveness of our approach over 51 runs [16]. We compared our approach to random search, a MOAIS (Multi-Objective Artificial Immune System) called NNIA (Non-dominated Neighbor-based Immune Algorithm) [18], and two existing code-smells approaches [8][9]. Our results indicate that our proposal has great promise and outperforms two existing studies [8][9] by detecting most of the expected code-smells with an average of 86% of precision and 91% of recall.

The remainder of this paper is structured as follows. Section 2 provides the background required to understand our approach and the nature of the code-smells detection challenge. In Section 3, we describe multi-objective optimization and explain how we formulate code-smells detection as an optimization problem. Section 4 presents and discusses the results obtained by applying our approach to seven large open-source projects. Related work is discussed in Section 5, while in Section 6 we conclude and suggest future research directions.

2 Background and Problem Statement

Code-smells are unlikely to cause failures directly, but may do it indirectly [6]. 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. A list of code-smells is defined in the literature with their potential symptoms. These include large classes, feature envy, long parameter lists, lazy classes, etc. In our approach, we focus on the following *three* code-smell types: 1) *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. 2) *Spaghetti Code*: It is a code with a complex and tangled control structure. 3) *Functional Decomposition*: It occurs when a class is designed with the intent of performing a single function. This is found in code produced by non-experienced object-oriented developers. We choose these code-smell types in our experiments because they are *the most frequent and hard to detect and fix* based on recent empirical studies [5][6][[7][8][9][15].

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 these 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 [17]. We selected and used in our experiments the following quality metrics, 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 Associations (NAS), Number of Classes (NC), Depth of Inheritance Tree (DIT), Polymorphism Factor (PF), Attribute Inheritance Factor (AIF) and Number Of Children (NOC).

The manual definition of rules to identify is difficult and can be time-consuming. One of the main issues 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. Thus, the manual definition of detection rules sometimes requires a high calibration effort. Furthermore, the manual selection of the best combination of metrics that formalize some symptoms of code-smells is challenging. In fact, different metrics can used to identify the same symptom thus different possible metric combinations need to be explored manually. In addition, the translation of code-smell definitions into metrics is not straightforward. Some definitions of code-smells are confusing and it is difficult to determine which metrics to use to identify such design problems. To address these challenges, we describe in the next section our approach based on the use of multi-objective genetic programming to generated code-smells detection rules using not only bad design practice examples but also good ones.

3 Code-Smells Detection as a Multi-Objective Problem

In this section, we describe first the principle of multi-objective genetic programming (MOGP) then we give details about our adaptation of MOGP to the problem of code-smells detection.

3.1 MOGP

Genetic Programming (GP) is a powerful evolutionary metaheuristic which extends the generic model of learning to the space of programs [1]. Differently to other evolutionary approaches, in GP, population individuals are themselves programs following a tree-like structure instead of fixed length linear string formed from a limited alphabet of symbols. GP can be seen as a process of program induction that allows automatically generating programs that solve a given task. Most exiting work on GP makes use of a single objective formulation of the optimization problem to solve using only one fitness function to evaluate the solution. Differently to single-objective optimization problems, the resolution of Multi-objective Optimization Problems (MOPs) yields a set of trade-off solutions called non-dominated solutions and their image in the objective space is called the Pareto front. In what follows, we give some background definitions related to this topic:

Definition 1 (MOP). A MOP consists in minimizing or maximizing a set of objective functions under some constraints [1]. An MOP could be expressed as:

$$\begin{cases} Min\, f(x) = [f_1(x), f_2(x), \ldots, f_M(x)]^T \\ g_j(x) \geq 0 \qquad\qquad j = 1, \ldots, P; \\ h_k(x) = 0 \qquad\qquad k = 1, \ldots, Q; \\ x_i^L \leq x_i \leq x_i^U \qquad\qquad i = 1, \ldots, n. \end{cases} \tag{1}$$

where M is the number of objective functions, P is the number of inequality constraints, Q is the number of equality constraints, x_i^L and x_i^U correspond to the lower and upper bounds of the variable x_i. A solution x_i satisfying the (P+Q) constraints is said feasible and the set of all feasible solutions defines the feasible search space denoted by Ω. In this formulation, we consider a minimization MOP since maximization can be easily turned to minimization based on the duality principle by multiplying each objective function by -1. The resolution of a MOP consists in approximating the whole Pareto front.

Definition 2 (Pareto optimality). A solution $x^* \in \Omega$ is Pareto optimal if there does not exist any solution x such that $f_m(x) < f_m(x^*)$ for all m.

The definition of Pareto optimality states that x^* is Pareto optimal if no feasible vector x exists which would improve some objective without causing a simultaneous worsening in at least another one. Other important definitions associated with Pareto optimality are essentially the following:

Definition 3 (Pareto dominance). A solution $u = (u_1, u_2, \ldots, u_n)$ is said to dominate another solution $v = (v_1, v_2, \ldots, v_n)$ (denoted by $f(u) \preceq f(v)$) if and only if $f(u)$ is partially less than $f(v)$. In other words, $\forall m \in \{1, \ldots, M\}$ we have $f_m(u) \leq f_m(v)$ and $\exists m \in \{1, \ldots, M\}$ where $f_m(u) < f_m(v)$.

Definition 4 (Pareto optimal set). For a MOP $f(x)$, the Pareto optimal set is $P^* = \{x \in \Omega \mid \neg \exists x' \in \Omega, f(x') \preceq f(x)\}$.

Definition 5 (Pareto optimal front). For a given MOP $f(x)$ and its Pareto optimal set P^*, the Pareto front is $PF^* = \{f(x), x \in P^*\}$.

In this work, we use a MOGP that follows the same evolutionary process of NSGA-II [25]. As described in Figure 1, the first step in MOGP is to create randomly a population P_0 of individuals encoded as trees. Then, a child population Q_0 is generated from the population of parents P_0 using genetic operators such as crossover and mutation. The crossover operator is based on sub-trees exchange and the mutation is based on random change in the tree. Both populations are merged into an initial population R_0 of size N, and a subset of individuals is selected, based on the dominance principle and crowding distance [1] to create the next generation. This process will be repeated until reaching the last iteration according to stop criteria. In the section, we describe the adaptation of MOGP to our problem.

```
01.  Begin
02.   While stopping criteria not reached do
03.     R_t ← P_t ∪ Q_t ;
04.     F ← fast-non-dominated-sort (R_t) ;
05.     P_t+1 ← Ø ; i ← 1 ;
06      While | P_t+1| + |F_i| ≤ N do
07.         Apply crowding-distance-assignment (F_i) ;
08.         P_t+1 ← P_t+1 ∪ F_i ;
09          i ← i + 1 ;
10.      End While
11.      Sort (Fi, ≺n ) ;
12.    P_t+1 ← P_t+1 ∪ F_i [1 : N – |P_t+1|] ;
13.     Q_t+1 ← create-new-population (P_t+1) ;
14.     t ← t+1 ;
15.   End While
16.  End
```

Figure 1. High-level pseudo-code of MOGP.

3.2 MOGP Adaptation for Code-Smells Detection

3.2.1 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-smells detection problem is a set of rules (metric combination with their thresholds values) where the goal of applying these rules is to detect code smells in a system. We propose a multi-objective formulation of the code-smells rules generation problem. Consequently, we have two objective functions to be optimized: (1) maximizing the coverage of code-smell examples, and (2) minimizing the detection of good design-practice examples. The collected examples of well-design code and code-smells on different systems are taken as an input for our approach. Analytically speaking, the formulation of the multi-objective problem can be stated as follows:

$$\begin{cases} \max f_1(x) = \dfrac{\dfrac{|DCS(x)| \cap |ECS|}{|ECS|} + \dfrac{|DCS(x)| \cap |ECS|}{|DCS(x)|}}{2} \\ \min f_2(x) = \dfrac{\dfrac{|DCS(x)| \cap |EGE|}{EGE} + \dfrac{|DCS(x)| \cap |EGE|}{|DCS(x)|}}{2} \end{cases} \quad (2)$$

where $|DCS(x)|$ is the cardinality of the set of Detected Code-Smells by the metric combination x, $|ECS|$ is the cardinality of the set of Existing Code-Smells, and |GDE| is the cardinality of the set of Existing Good Examples.

Once the bi-objective trade-off front is obtained, the developer can navigate through this front in order to select his/her preferred solution (metric combination).

3.2.2 Solution Approach

We use a MOGP that follows the same evolutionary scheme as the well-known multi-objective evolutionary algorithm NSGA-II [1][25] to try to solve the code-smells detection problem. As noted by Harman et al. [9], a multi-objective algorithm cannot be used "out of the box" – it is necessary to define problem-specific genetic operators to obtain the best performance. To adapt MOGP to our problem, the required steps are to create: (1) solution representation, (2) solution variation, and (3) solution evaluation.

Solution representation. In our MOGP, a solution is composed of terminals and functions. Therefore, when applying MOGP 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. The functions that can be used between these metrics are Union (*OR*) and Intersection (*AND*). More formally, each candidate solution x in this problem is a set 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. Each sub-tree corresponds to a rule for the detection of specific code-smell (e.g. blob, functional decomposition, etc.). Figure 2 illustrates an example of a solution according to our formulation including two rules. The first rule is to detect Blob using the two metrics NLC and NA and the second rule detects Spaghetti Code (SC) using one metric NOC. The thresholds value are selected randomly along with the comparison and logic operators.

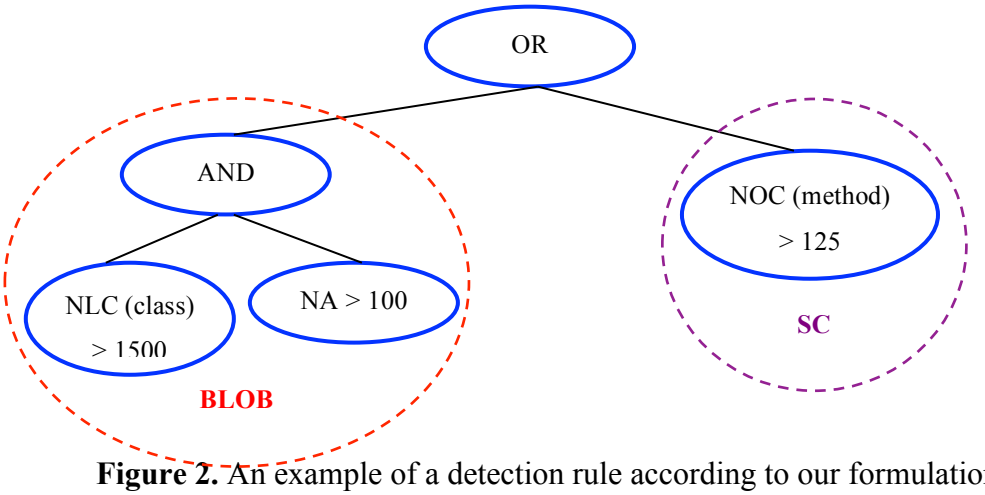


Figure 2. An example of a detection rule according to our formulation.

Solution variation. The mutation operator can be applied to a function node or 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. If a tree mutation is to be carried out, the node and its sub-tree are replaced by a new randomly generated sub-tree. Figure 3 illustrates an example of a mutation operation. One node is deleted from the tree representation in Figure 3 to generate a new other possible solution. For the crossover, 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 rules category (code-smell type to detect). Each child thus obtains information from both parents. Figure 4 describes an example of a crossover operation. Two sub-trees are exchanges between two solutions to generate two new ones.

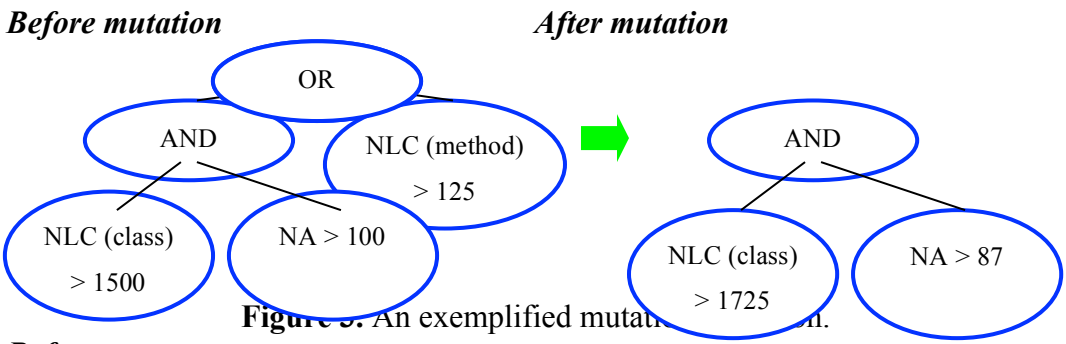
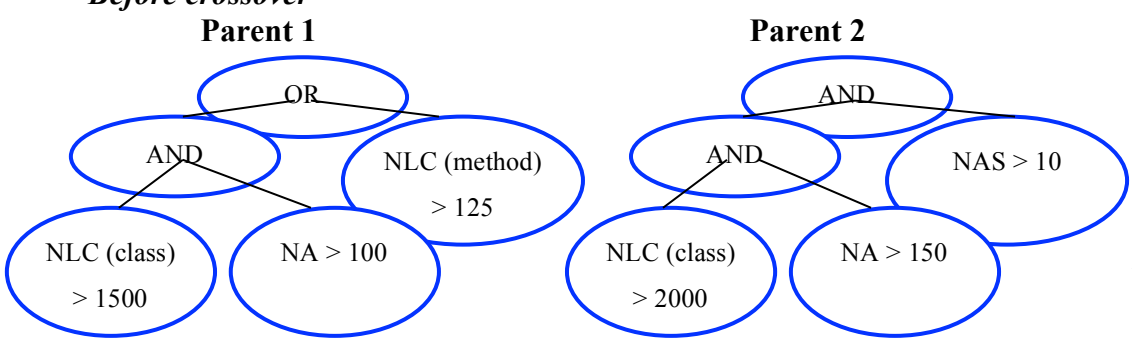


Figure 3. An exemplified mutati[illegible]n.



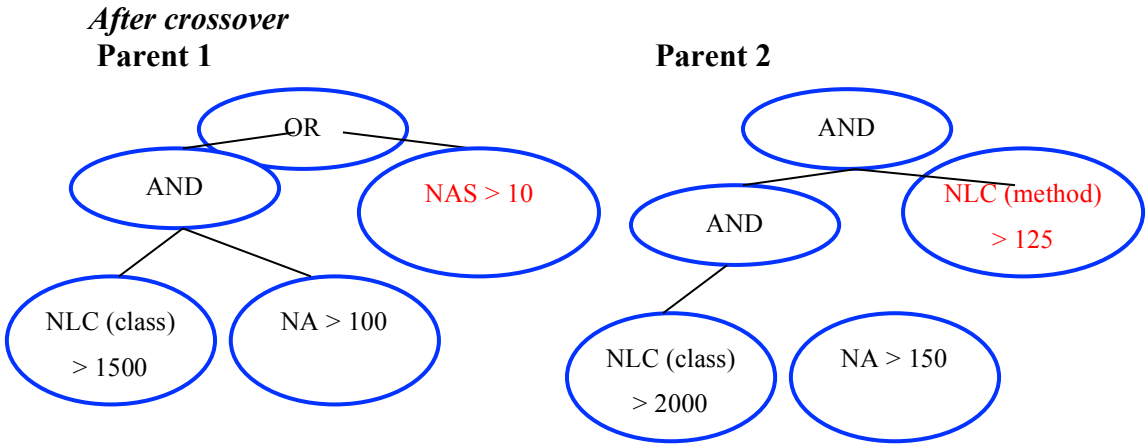


Figure 4. An exemplified corssover operation.

Solution evaluation. The solution is evaluated based on the two objective functions defined in the previous section. Since we are considering a bi-objective formulation, we use the concept of Pareto optimality to find a set of compromise (Pareto-optimal) solutions. The fitness of a particular solution in MOGP corresponds to a couple (*Pareto Rank*, *Crowding distance*). In fact, MOGP classifies the population individuals (of parents and children) into different layers, called non-dominated fronts. Non-dominated solutions are assigned a rank of 1 and then are discarded temporary from the population. Non-dominated solutions from the truncated population are assigned a rank of 2 and then are discarded temporarily. This process is repeated until the entire population is classified with the domination metric. After that, a diversity measure, called *crowding distance*, is assigned front-wise to each individual. The crowding distance is the average side length of the cuboid formed by the nearest neighbors of the considered solution. Once each solution is assigned its Pareto rank, mating selection and environmental selection are performed. This is based on the crowded comparison operator ($\prec n$) that favors solutions having better Pareto ranks and, in case of equal ranks, it favors the solution having larger crowding distance. In this way, convergence towards the Pareto optimal bi-objective front and diversity along this front are emphasized simultaneously. The output of MOGP is the last obtained parent population containing the best of the non-dominated solutions found. When plotted in the objective space, they form the Pareto front from which the user will select his/her preferred code-smells detection rules solution.

4 Empirical Evaluation

4.1 Design of the Experimental Study

4.1.1 Research Questions

We defined five research questions that address the applicability, performance in comparison to existing refactoring approaches, and the usefulness of our multi-objective code-smells detection approach. The five research questions are as follows:

RQ1: MOGP versus Random Search (this serves as sanity check). If random search outperforms an intelligent search method, then we can conclude that our problem formulation is not adequate.
RQ2: Comparison to start of the art code-smell detectors.
RQ 2.1: How does MOGP perform compared to MOAIS (quality Indicators, precision and recall)? It is important to justify the use of MOGP for the problem of multi-objective code-smells detection. We compare MOGP with another multi-objective algorithm, MOAIS [18] using the same adaptations.
RQ 2.2: How does MOGP perform compared to mono-objective genetic programming (GP) with aggregation of both objectives (precision and recall)? This comparison is required to ensure that the detection rules solutions provided by MOGP and MOAIS provide a better trade-offs than a mono-objective approach. Otherwise, there is no benefit to our multi-objective adaptation.
RQ 2.3: How does MOGP perform compared to some existing code-smells detection approaches (precision and recall)? It is important to determine if considering both code-smell examples and well-designed ones performs better than existing approaches [8][9].
RQ3: To what extent can our approach generates rules that detect different code-smell types? It is important to discuss the ability of our approach to detect different types of code-smells to evaluate the quality of each detection rule separately.

4.1.2 Studied Systems

Our study considers the extensive evolution of different open source Java systems analyzed in the literature [5] [7][8][15]. The corpus used includes releases of Apache Ant [10], ArgoUML [11], Gantt [13], Azureus [12] and Xerces-J [14]. Apache Ant is a build tool and library specifically conceived for Java applications. ArgoUML is an open-source UML modeling tool. Xerces is a family of software packages that implement a number of standard APIs for XML parsing. GanttProject is a tool for creating project schedules in the form of Gantt charts and resource-load charts. Azureus is a peer-to-peer file-sharing tool. Table 1 reports the size in terms of classes of the analyzed systems. The table also reports the number of code smells identified manually in the different systems, more than 700 in total. Indeed, in several works [5][7][8][15], the authors asked different groups of developers to analyze the libraries to tag instances of specific code smells to validate their detection techniques. For replication of experimental results, they provided a corpus describing instances of different code smells including blob, spaghetti code, and functional decomposition. In our study, we verified the capacity of our approach to fix classes that correspond to instances of these code smells. We choose the above-mentioned open source systems because they are medium/large-sized open-source projects and were analyzed in related work. . JHotDraw [19] was chosen as an example of reference code because it contains very few known code-smells and it is well-know that JHotDraw is one of the best well-designed open source systems.

Table 1. Software projects features.

Systems	Number of classes	Number of code smells
ArgoUML v0.26	1358	138
ArgoUML v0.3	1409	129
Xerces v2.7	991	82
Ant-Apache v1.5	1024	103
Ant-Apache v1.7.0	1839	124
Gantt v1.10.2	245	41
Azureus v2.3.0.6	1449	108

4.1.3 Evaluation Metrics

We use the two following performance indicators (which are among the most used in multi-objective optimization) when comparing MOGP and MOAIS:

−*Hypervolume* (*IHV*) [20]: It corresponds to the proportion of the objective space that is dominated by the Pareto front approximation returned by the algorithm and delimited by a reference point. Larger values for this metric mean better performance.

−*Inverted Generational Distance* (*IGD*) [20]: It is a combined convergence and diversity measure that corresponds to the average Euclidean distance between the Pareto front Approximation *PA* provided by the algorithm and the Reference Front *RF* (RF is the set of non-dominated solutions obtained over all runs). The distance between *PA* and *RF* in an *M*-objective space is calculated as the average *M*-dimensional Euclidean distance between each solution in *PA* and its nearest neighbour in *RF*. Lower values for this indicator mean better performance (convergence).

To assess the accuracy of our approach, we compute two measures: (1) precision and (2) recall, originally stemming from the area of information retrieval:

−*Precision (PR)*: It denotes the fraction of correctly detected code-smells among the set of all detected code-smells. It could be seen as the probability that a detected code-smell is correct.

−*Recall (RE)*: It corresponds to the fraction of correctly detected code-smells among the set of all manually identified code-smells (i.e., how many code-smells have not been missed). It could be seen as the probability that an expected code-smell is detected.

4.1.4 Used Inferential Statistical Methodology

Since metaheuristic algorithms are stochastic optimizers, they usually provide different results for the same problem instance from one run to another. For this reason, our experimental study is performed based on 51 independent simulation runs for each problem instance and the obtained results are statistically analyzed by using the Wilcoxon rank sum test [16] with a 95% confidence level ($\alpha = 5\%$). This statistical test 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 (≤ 0.05) means that we

accept H_1 and we reject H_0. However, a p-value that is strictly greater than α (> 0.05) means the opposite.

4.1.5 Parameter Tuning and Setting

Parameter setting has a significant influence on the performance of a search algorithm on a particular problem instance. For this reason, for each multi-objective algorithm and for each system (cf. Table 2), we perform a set of experiments using several population sizes: 50, 100, 200, 500 and 1,000. The stopping criterion was set to 250,000 fitness evaluations for all algorithms in order to ensure fairness of comparison. The other parameter values were fixed by trial and error and are as follows: (1) crossover probability = 0.8; mutation probability = 0.2 where the probability of gene modification is 0.3. For MOAIS, the maximum size of the dominant population, and the clone population size are set to 100 and 20, respectively.

Table 2. Best population size configurations.

System	MOGP	MOAIS	Mono-GP
ArgoUML v0.26	500	500	500
ArgoUML v0.3	500	200	500
Xerces v2.7	200	200	100
Ant-Apache v1.5	200	500	500
Ant-Apache v1.7.0	100	100	200
Gantt v1.10.2	100	100	100
Azureus v2.3.0.6	500	500	500

4.2 Analysis of Results

4.2.1 Results for RQ1

We do not dwell long in answering the first research question (RQ1) that involves comparing our MOGP approach based on NSGA-II with random search. The remaining research questions will reveal more about the performance, insight, and usefulness of our approach. Table 3 confirms that MOGP and MOAIS are better than random search based on the two quality indicators IHV and IGD on all seven open source systems. The Wilcoxon rank sum test showed that in 51 runs both MOGP and MOAIS results were significantly better than random search. We conclude that there is empirical evidence that our multi-objective formulation surpasses the performance of random search thus our formulation is adequate (this answers RQ1).

4.2.2 Results for RQ2

In this section, we compare our MOGP adaptation to the current, state-of-the-art code-smells detection approaches. To answer the second research question, RQ2.1, we compared MOGP to another multi-objective algorithm, MOAIS, using the same adaptations. Table 4 shows the overview of the results of the significance tests comparison between MOGP and MOAIS. MOGP outperforms MOAIS in most of the cases: 11 out of 14 experiments (78%).

Table 3. The significantly best algorithm among random search, MOGP and MOAIS (No sign. diff. means that MOGP and MOAIS are significantly better than random, but not statistically different).

Project	*IHV*	*IGD*
ArgoUML v0.26	MOGP	MOGP
ArgoUML v0.3	MOGP	MOGP
Xerces v2.7	No sign. diff.	MOAIS
Ant-Apache v1.5	MOGP	MOGP
Ant-Apache v1.7.0	MOGP	MOGP
Gantt v1.10.2	MOGP	No sign. diff.
Azureus v2.3.0.6	MOGP	MOGP

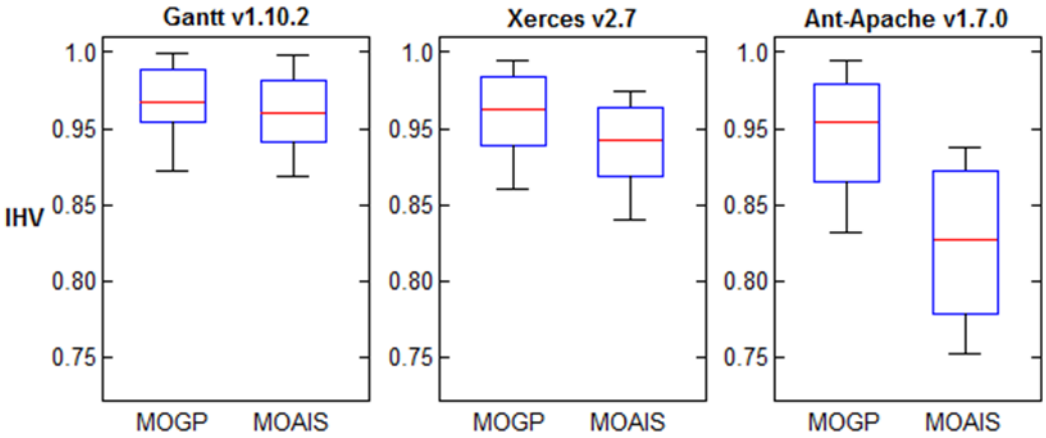


Figure 5. IHV boxplots on 3 projects having different sizes (Gantt v1.10.2: small, Xerces v2.7: medium, Ant-Apache v1.7.0: large).

A more qualitative evaluation is presented in Figures 5 and 6 illustrating the box plots obtained for the multi-objective metrics on the different projects. We see that for almost all problems the distributions of the metrics values for MOGP have smaller variability than for MOAIS. This fact confirms the effectiveness of MOGP over MOAIS in finding a well-converged and well-diversified set of Pareto-optimal detection rules solutions (RQ2.1).

Next, we use precision and recall measures to compare the efficiency of our MOGP approach compared to mono-objective GP (aggregating both objectives) and two existing code-smells detection studies [8][9]. We first note that the mono-objective approaches provide only one detection solution (set of detection rules), while MOGP generate a set of non-dominated solutions. In order to make meaningful comparisons, we select the best solution for MOGP using a *knee point* strategy as described in Figure 7 [20]. The knee point corresponds to the solution with the maximal trade-off between maximizing the coverage of code-smells and minimizing the number of detected well-designed code. Thus, for MOGP, we select the knee point from the Pareto approximation having the median IHV value. We aim by this strategy to ensure fairness when making comparisons

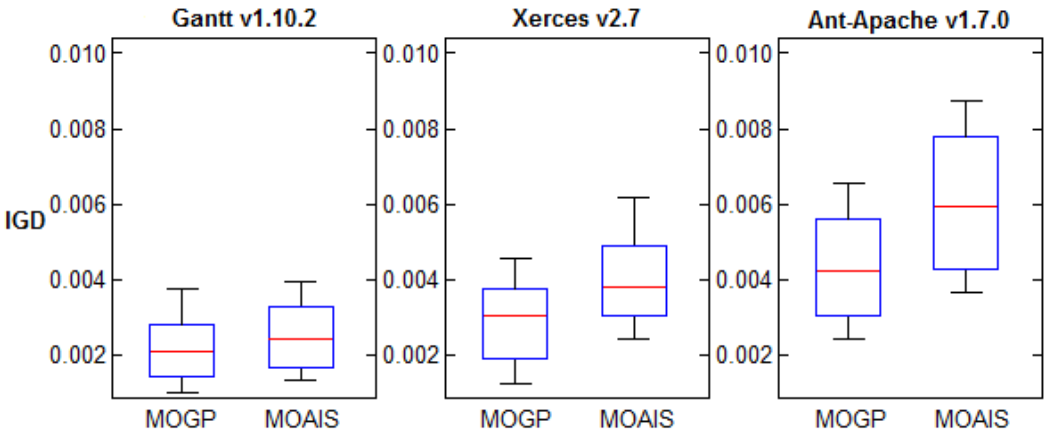


Figure 6. IGD boxplots on 3 projects having different sizes (Gantt v1.10.2: small, Xerces v2.7: medium, Ant-Apache v1.7.0: large).

against the mono-objective EA. For the latter, we use the best solution corresponding to the median observation on 51 runs.

The results from 51 runs are depicted in Table 4. It can be seen that MOGP provides better precision and recall scores for the detection of code-smells. For recall (RE), MOGP is better than GP in 100% of the cases. We have the same observation for the precision also where MOGP outperforms GP in all cases with an average of more than 90%. Thus, it is clear that a multi-objective formulation of our problem outperforms the aggregation-based approach. In conclusion, we answer RQ2.2 by concluding that the results obtained in Table 4 confirm that both multi-objective formulations are adequate and outperform the mono-objective algorithm based on an aggregation of two objectives related to use of good and bad software design examples. Table 4 also shows the results of comparing our multi-objective approach based on MOGP with two mono-objective refactoring approaches [8][9]. In [8], the authors used search-based techniques to detect code-smells from only code-smell examples. In [9], an artificial immune system approach is proposed to detect code-smells by deviation with well-designed code examples. It is apparent from Table 4 that MOGP outperforms both mono-objective approaches in terms of precision and recall in most of the cases. This is can be explained by the fact that our proposal takes into account both positive and negative examples when generating the detection rules. If only code-smell examples are used then it is difficult to ensure the coverage of all possible bad design behaviors. The same observation is still valid for the use only of well-designed code examples. The use of both types of examples represents a complementary way to formulate the problem of code-smells detection using a multi-objective approach. To answer RQ2.3, the results of Table 4 support the claim that our MOGP formulation outperforms, on average, the two code-smells existing approaches.

Table 4. Recall and precision median values of MOGP, GP, [7] and [8] over 51 independent simulation runs.

System	*RE-MOGP*	*RE-GP*	*RE-[7]*	*RE-[8]*	*PR-MOGP*	*PR-GP*	*PR-[7]*	*PR-[8]*
ArgoUML v0.26	91% (126/138)	84% (117/138)	82% (114/138)	84% (116/138)	88% (126/142)	77% (117/151)	74% (114/154)	77% (116/149)
ArgoUML v0.3	89% (116/129)	82% (107/129)	76% (98/129)	79% (102/129)	84% (116/137)	74% (107/144)	66% (98/147)	69% (102/146)
Xerces v2.7	89% (74/82)	84% (69/82)	77% (63/82)	77% (67/82)	86% (74/86)	73% (69/94)	64% (63/98)	67% (67/99)
Ant-Apache v1.5	87% (91/103)	79% (82/103)	86% (88/103)	81% (79/103)	90% (91/101)	78% (82/103)	76% (88/117)	70% (79/112)
Ant-Apache v1.7.0	93% (116/124)	79% (98/124)	73% (91/124)	70% (87/124)	97% (116/119)	71% (98/138)	62% (91/147)	64% (87/136)
Gantt v1.10.2	92% (38/41)	87% (36/41)	78% (32/41)	70% (29/41)	74% (38/51)	62% (36/58)	58% (32/56)	61% (29/47)
Azureus v2.3.0.6	93% (101/108)	81% (88/108)	71% (76/108)	76% (82/108)	87% (101/116)	72% (88/122)	62% (76/124)	69% (82/118)

4.2.3 Results for RQ3

We noticed that our technique does not have a bias towards the detection of specific code-smells types. As described in Figure 8, in all systems, we had an almost equal distribution of each code-smell types (SCs, Blobs, and FDs). Overall, all the three code smell types are detected with good precision and recall scores in the different systems (more than 85%). This ability to identify different types of code-smells underlines a key strength to our approach.

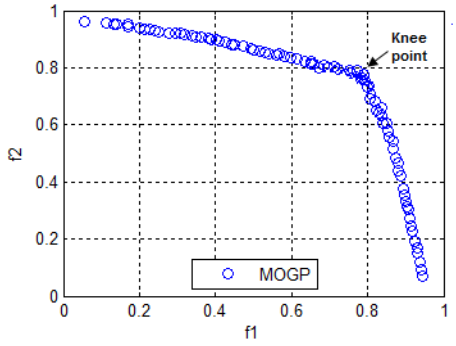


Figure 7. MOGP knee point for bi-objective code smells detection problem. f1 and f2 corresponds to the normalized values of both fitness functions.

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. This difficulty limits the performance of GP in well detecting this type of code-smells. Thus, we can conclude that our MOGP approach detects well all the three types of considered code-smells (RQ3).

4.3 Threats to Validity

In our experiments, a 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. Another construct threat is the base of good and bad design examples that we considered on our experiments that need to include a high number of examples. We can conclude that the use of open source systems can be a good starting point to use our approach in industrial settings.

We take into consideration the internal threats to validity in the use of stochastic algorithms since our experimental study is performed based on 51 independent simulation runs for each problem instance and the obtained results are statistically analyzed by using the Wilcoxon rank sum test with a 95% confidence level (α = 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. External validity refers to the generalizability of our findings. In this study, we performed our experiments on seven 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.

5 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.

In [7], 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. [26] 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 [27], 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

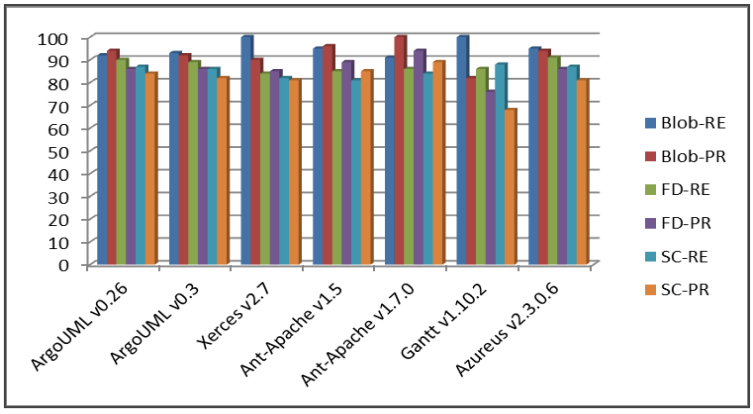


Figure 8. The median values of precision and recall on 51 runs for the three types of code-smell.

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. [28] present a pattern-based framework for developing tool support to detect software anomalies by representing potential code-smells with different colors. Dhambri et al. [15] 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 [30] 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.

Moha et al. [6] 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. [31] extending the DECOR approach [6], 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. [23] 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 [7]. 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 [32] 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. [27] proposed a metric-based heuristic framework to detect and locate object-oriented design flaws similar to those illustrated by Marinescu [32]. 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. [33] 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 is no consensus in defining design smells, different threshold values should be tested to find the best ones.

Our approach is inspired by contributions in the domain of Search-Based Software Engineering (SBSE) [10]. 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 [35], 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 [34], 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 [10], the use of multi-objective optimization is still limited in software engineering. Indeed, this work represents the first attempt to use multi-objective optimization to address a software engineering problem.

6 Conclusions and Future Work

In this paper we have introduced a novel multi-objective approach to generate rules for the detection of code-smells. To this end, we have used MOGP to find the best trade-offs between maximizing the detection of examples of code-smells and minimizing the detection of well-designed code examples. We have evaluated our approach on seven large open source systems. All results were found to be statistically significant over 51 independent runs using the Wilcoxon rank sum test with a 99% confidence level ($\alpha < 1\%$). Our MOGP results have outperformed some existing studies [7][8] by detecting most of the expected code-smells with an average of 86% of precision and 91% of recall.

Future work should validate our approach with additional code-smell types in order to conclude about the general applicability of our methodology. Also, in this paper, we have focused on the detection of code-smells. We are planning to extend the approach by automating the correction of these code-smells. In addition, we shall consider the importance of code-smells during the detection step using previous code-changes, classes-complexity, etc. Thus, the detected code-smells will be ranked based on a severity score.

References

[1] Koza JR. Genetic Programming. MIT Press, Cambridge, MA, 1992.

[2] Deb K.. Multiobjective Optimization using Evolutionary Algorithms. John Wiley and Sons, Ltd, New York, USA, 2001.

[3] Langdon WB, Poli R, McPhee NF, Koza JR. Genetic Programming: An Introduction and Tutorial, with a Survey of Techniques and Applications. In: Computational Intelligence: A Compendium, Fulcher, John and Jain, Lakhmi C. Editors, Springer, pp. 927-1028, 2008.

[4] Coello Coello CA, Lamont GB, Van Veldhuizen DA. Evolutionary Algorithms for Solving Multi-objective Problems (second edition). Springer, New York, 2007.

[5] Abran A, Hguyenkim H. Measurement of the Maintenance Process from a Demand-Based Perspective. Journal of Software Maintenance: Research and Practice, vol. 5, no. 2, 63-90, 1993.

[6] Moha N, Guéhéneuc YG, Duchien L, Le Meur AF. DECOR: A Method for the Specification and Detection of Code and Design Smells. IEEE Transactions on Software Engineering, vol. 36, no. 1, 20-36, 2010.

[7] Fowler M, Beck K, Brant J, Opdyke W, Roberts D. Refactoring – Improving the Design of Existing Code. Addison-Wesley Professional, 1999.

[8] Kessentini M, Kessentini W, Sahraoui H, Boukadoum M, Ouni A. Design Defects Detection and Correction by Example. In Proceedings of the 19th IEEE International Conference on Program Comprehension (ICPC'11), pp. 81-90, 2011.

[9] Kessentini M, Vaucher S, Sahraoui H. Deviance from Perfection is a Better Criterion than Closeness to Evil when Identifying Risky Code. In proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering (ASE) , pp. 141-151, 2010.

[10] Harman M, Mansouri SA, Zhang Y. Search-based software engineering: Trends, techniques and applications. ACM Computing Surveys, 45, 61 pages, 2012.

[11] (ApacheAnt). [Online]. Available: http://ant.apache.org/

[12] (ArgoUML). [Online]. Available: http://argouml.tigris.org/

[13] (Azureus). [Online]. Available: http://vuze.com

[14] (GanttProject). [Online]. Available: www.ganttproject.biz

[15] (Xerces-J). [Online]. Available: http://xerces.apache.org/xerces-j/

[16] Palomba F, Bavota G, Di Penta M, Oliveto R, De Lucia A, Poshyvanyk D. Detecting Bad Smells in Source Code Using Change History Information. In Proceedings of 28th IEEE/ACM International Conference on Automated Software Engineering (ASE), to appear, 2013.

[17] Arcuri A, Briand LC. A practical Guide for using Statistical Tests to Assess Randomized Algorithms in Software Engineering. In Proceedings of the 33rd International Conference on Software Engineering (ICSE), pp. 1-10, 2011.

[18] Gong M, Jiao L, Du H, Bo L. Multiobjective Immune Algorithm with Nondominated Neighbor-Based Selection. Evolutionary Computation, vol. 6, no. 2, pp. 225-255, 2008.
[19] Fenton N, Pfleeger SL. Software Metrics: A Rigorous and Practical Approach (2nd edition). London, UK, International Thomson Computer Press, 1998.
[20] (JHotDraw). http://www.jhotdraw.org/
[21] Zitzler E, Thiele L, Laumanns M, Fonseca CM, da Fonseca VG. Performance Assessment of Multiobjective Optimizers: An Analysis and Review. IEEE Transaction on Evolutionary Computation. vol. 7, no. 2, pp. 117-132, 2003.
[22] Brown WJ, Malveau RC, Brown WH, Mowbray TJ. Anti-Patterns: Refactoring Software, Architectures, and Projects in Crisis. John Wiley and Sons, 1998.
[23] Munro MJ. Product Metrics for Automatic Identification of "Bad Smell" Design Problems in Java Source-Code. In Proceedings of the 11th International Software Metrics Symposium, page 15, 2005.
[24] Marinescu R. Detection Strategies: Metrics-based Rules for Detecting Design Flaws. In the Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM), pp. 350-359, 2004.
[25] Deb K, Agrawal S, Pratap A, Meyarivan T. A Fast and Elitist Multi-objective Genetic Algorithm: NSGA-II. IEEE Transactions on Evolutionary Computation, vol. 6, no. 2, 182-197, 2002.
[26] 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,.
[27] 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.
[28] 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,.
[29] Dhambri, K., Sahraoui, H.A., Poulin, P. 2008. Visual detection of design anomalies. CSMR. IEEE, pp. 279–283,.
[30] 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.
[31] 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 .
[32] Marinescu, R. Detection strategies: metrics-based rules for detecting design flaws. In: Proc. of ICM'04, pp. 350–359
[33] Erni, K. and Lewerentz, C. 1996. Applying design metrics to object-oriented frameworks, Proc. IEEE Symp. Software Metrics.
[34] 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.
[35] 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).