Autom Softw Eng (2013) 20:47–79
DOI 10.1007/s10515-011-0098-8

Maintainability defects detection and correction: a multi-objective approach

Ali Ouni · Marouane Kessentini · Houari Sahraoui · Mounir Boukadoum

Received: 30 March 2011 / Accepted: 23 December 2011 / Published online: 19 January 2012

Abstract Software defects often lead to bugs, runtime errors and software maintenance difficulties. They should be systematically prevented, found, removed or fixed all along the software lifecycle. However, detecting and fixing these defects is still, to some extent, a difficult, time-consuming and manual process. In this paper, we propose a two-step automated approach to detect and then to correct various types of maintainability defects in source code. Using Genetic Programming, our approach allows automatic generation of rules to detect defects, thus relieving the designer from a fastidious manual rule definition task. Then, we correct the detected defects while minimizing the correction effort. A correction solution is defined as the combination of refactoring operations that should maximize as much as possible the number of corrected defects with minimal code modification effort. We use the Non-dominated Sorting Genetic Algorithm (NSGA-II) to find the best compromise. For six open source projects, we succeeded in detecting the majority of known defects, and the proposed corrections fixed most of them with minimal effort.

A. Ouni · H. Sahraoui
DIRO, Université de Montréal, Montréal, Canada

A. Ouni
e-mail: ouniali@iro.umontreal.ca

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

M. Kessentini (✉)
CS, Missouri University of Science and Technology, Rolla, USA
e-mail: marouanek@mst.edu

M. Boukadoum
DI, Université du Québec à Montréal, Montréal, Canada
e-mail: Boukadoum.mounir@uqam.ca

Keywords Maintainability defects · Software maintenance · Search-based software engineering · Multi-objective optimization · By example · Effort

1 Introduction

Many studies report that software maintenance, traditionally defined as any modification made to software code during its development or after its delivery, consumes up to 90% of the total cost of a typical software project (Brown et al. 1998). Adding new functionalities, detecting maintainability defects, correcting them, and modifying the code to improve its quality are major activities of the maintenance effort (Fenton and Pfleeger 1997). Although maintainability defects are sometimes unavoidable, they should be prevented in general by the development teams, and removed from the code base as early as possible.

There has been much research effort focusing on the study of maintainability defects, also called anomalies (Brown et al. 1998), antipatterns (Fowler 1999), or smells (Fenton and Pfleeger 1997) in the literature. Such defects include, for example, blobs (very large classes), spaghetti code (tangled control structure), functional decomposition (a class representing only a single function), etc.

Detecting and fixing maintainability defects is, to some extent, a difficult, time-consuming, and manual process (Liu et al. 2009). The number of software defects typically exceeds the resources available to address them. In many cases, mature software projects are forced to ship with both known and unknown defects for lack of development resources to deal with every defect.

To insure the detection of maintainability defects, several automated detection techniques have been proposed (Moha et al. 2009; Liu et al. 2009; Marinescu 2009; Khomh et al. 2009; Kessentini et al. 2010). The vast majority of these techniques relies on declarative rule specification (Moha et al. 2009; Liu et al. 2009; Marinescu 2009; Khomh et al. 2009). In these settings, rules are manually defined to identify the key symptoms that characterize a defect. These symptoms are described using quantitative metrics, structural, and/or lexical information. For example, large classes have different symptoms like the high number of attributes, relations and methods that can be expressed using quantitative metrics. However, in an exhaustive scenario, the number of possible defects to be manually characterized with rules can be very large. For example, Brown et al. (1998), Fowler (1999), Fenton and Pfleeger (1997) describe more than sixty defect types. In addition, this list is not exhaustive because different defects are not documented. For each defect, rules that are expressed in terms of metric combinations need substantial calibration efforts to find the right threshold value for each metric, above which a defect is said to be detected. Since there is no consensus in defining defects, different threshold values should be tested to find the best one.

Another work (Moha et al. 2009) proposes to use formal definitions of defects to generate detection rules. This partial automation of rule writing helps developers concentrate on symptom description. Still, translating symptoms into rules is not obvious because there is no consensualsymptom-based definition of maintainability defects (Brown et al. 1998). When consensus exists, the same symptom could be associated to many defect types, which may compromise the precise identification of

defect types. In fact, translating defect definitions from the natural language to metric combinations is a subjective task. For this reason, different defects are characterized by the same metrics. Thus, it is difficult to identify some defect types. These difficulties explain a large portion of the high false-positive rates reported in existing research.

After detecting maintainability defects, the next step is to fix them. Some authors, such as in Fowler (1999), Liu et al. (2009), Mens and Tourwé (2004), Sahraoui et al. (2000), propose "standard" refactoring solutions that can be applied by hand for each kind of defect. However, it is difficult to prove or ensure the generality of these solutions to all defects or software codes. In fact, the same defect type can have different possible corrective solutions. Automated approaches are used in the majority of existing works (O'Keeffe and Cinnéide 2008; Harman and Tratt 2007; Seng et al. 19091916) to formulate the refactoring problem as a single-objective optimization problem that maximizes code quality. However, many proposed refactoring solutions can ameliorate code with the same quality, but with different code-modification (effort) degrees. Thus, applying a particular suggested refactoring sequence may require an effort that is comparable to the one of re-implementing part of the system from scratch. In these conditions, it is important to minimize code changes, and then effort, when we improve the code quality.

To conclude, some questions arise from the study of the existing approaches: how to determine the useful metrics that characterize the quality of a given system? What would be the best (or at least a good) way to combine multiple metrics for detecting defects? Does improving the metric values necessarily mean that specific defects are corrected? How to find the best refactoring solution that maximizes code quality while minimizing effort? The approaches in Fowler (1999), Liu et al. (2009), Mens and Tourwé (2004), Moha et al. (2009), Sahraoui et al. (2000) only bring limited answers to these questions.

Beside the previous approaches, one notices the availability of defect repositories in many companies, where defects in projects under development are manually identified, corrected and documented. However, this valuable knowledge is not used to mine regularities about defect manifestations, although these regularities could be exploited both to detect and correct defects.

In this paper, we propose to overcome some of the above-mentioned limitations with a two-step approach based on the use of defect examples generally available in defect repositories of software developing companies: (1) detection-identification of defects, and (2) correction of detected defects. In fact, we translate regularities that can be found in such defect examples into detection rules and correction solutions. Instead of specifying rules manually for detecting each defect type, or semi-automatically using defect definitions, we extract these rules from instances of maintainability defects. This is achieved using Genetic Programming (GP).

We generate correction solutions based on combinations of refactoring operations, taking in consideration two objectives: (1) maximizing code quality by minimizing the number of detected defects using detection rules generated by genetic programming (2) minimizing the effort needed to apply refactoring operations. Thus, we propose to consider refactoring as a multi-objective optimization problem instead of a single-objective one. To search for refactoring solutions, we selected and

adapted, from the existing multi-objective evolutionary algorithms (MOEAs) (Zitzler and Thiele 1998), the Non-dominated Sorting Genetic Algorithm (NSGA-II) (Deb et al. 2002). NSGA-II aims to find a set of representative Pareto optimal solutions in a single run. In our case, the evaluation of these solutions is based on two usually conflicting criteria's: maximizing quality and minimizing effort.

The proposed approach can be very beneficial since:

(1) it does not require to define the different defect types, but only to have some defect examples;
(2) it does not require an expert to write detection or correction rules manually;
(3) it does not require to specify the metrics to use or their related threshold values;
(4) it takes in consideration the effort to perform refactoring solutions;

The remainder of this paper develops our proposed steps and details how they are achieved. Section 2 is dedicated to the problem statement. In Sect. 3, we give an overview of our proposal. Then, Sect. 4 details our adaptations to the defect detection and correction problem. Section 5 presents and discusses the validation results. The related work in defect detection and correction is outlined in Sect. 6. We conclude and suggest future research directions in Sect. 7.

2 Research problem

To better understand our contribution, it is important to first define the problems of defect detection and correction. In this section, we first introduce definitions of important concepts related to our proposal. Then, we emphasize the specific problems that are addressed by our approach.

2.1 Definitions

Software maintainability defects, also called design anomalies, refer to design situations that adversely affect the maintenance of software. As stated by Fenton and Pfleeger (1997), maintainability defects 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 defects, presenting a variety of symptoms, have been studied in the intent of facilitating their detection (Fowler 1999) and suggesting improvement solutions. The two types of defects that are commonly mentioned in the literature are code smells and anti-patterns. In Fowler (1999), Beck defines 22 sets of symptoms of common defects, named code smells. These include large classes, feature envy, long parameter lists, and lazy classes. Each defect type is accompanied by refactoring suggestions to remove it. Brown et al. (1998) define another category of maintainability defects named anti-patterns. In our approach, we focus on the following three defect types:

- *Blob*: It is found in designs where one large class monopolizes the behavior of a system (or part of it), and other classes primarily encapsulate data.
- *Spaghetti Code*: It is a code with a complex and tangled control structure.

- *Functional Decomposition*: It occurs when a class is designed with the intent of performing a single function. This is found in code produced by non-experienced object-oriented developers.

The defect detection process consists of finding code fragments that violate structure or semantic properties on code elements such as coupling and complexity. In this setting, internal attributes are captured through software metrics, and properties are expressed in terms of valid values for these metrics (Gaffney 1981). The most widely used metrics are the ones defined by Chidamber and Kemerer (1994).

The detected defects can be fixed by applying refactoring operations. For example, to correct the blob defect many operations can be used to reduce the number of functionalities in a specific class: move methods, extract class, etc. Opdyke (1992) defines refactoring as the process of improving a code after it has been written by changing its internal structure without changing its external behavior. The idea is to reorganize variables, classes and methods in order to facilitate future extensions. This reorganization is used to improve different aspects of software-quality: reusability, maintainability, complexity, etc. (Mens and Tourwé 2004).

During the search for the optimal refactoring solution, we need to minimize the effort to perform refactoring operations. In general, efforts are related to software development. However, additional costs are added during the maintenance stage to correct detected anomalies, for example. In particular, this maintenance effort corresponds to the cost of modifying the code to improve his quality (Boehm et al. 2000). This cost can be estimated using different techniques, especially using metrics to predict complexity or some impact change techniques (Kapser and Godfrey 2006; Mehta and Heineman 2002; Boehm 1981).

Increasing the quality and reducing the changes (effort) associated to refactoring of software are complementary. Our work aims to solve those problems by automatically finding the optimal refactoring from an exhaustive list of candidate refactorings and apply this refactorings in order to improve the quality of the specified system.

2.2 Detection and correction issues

Although there exists a consensus that it is necessary to detect and fix design anomalies, our experience with industrial partners showed that there are many open issues that need to be addressed when defining a detection and correction tool. In the following, we introduce some of these open issues. Later, in Sect. 5, we discuss these issues in more detail with respect to our approach.

2.2.1 Detection issues

Unlike software bugs, there is no general consensus on how to decide if a particular design violates a quality heuristic. There is a difference between detecting symptoms and asserting that the detected situation is an actual defect.

Furthermore, detecting dozens of defect occurrences in a system is not always helpful except if the list of defects is sorted by priority. In addition to the presence of false positives that may create a rejection reaction from development teams, the

process of using the detected lists, understanding the defect candidates, selecting the true positives, and correcting them is long, expensive, and not always profitable.

In addition, there is a general agreement on extreme manifestations of maintainability defects. For example, consider an OO program with a hundred classes such that one class implements all the behavior and the others are only classes with attributes and accessors. There is no doubt that we are in the presence of a Blob. Unfortunately, we can find many large classes in real-life systems that make use of data and regular classes. Hence, deciding which classes are Blob candidates depends heavily on the analyst's interpretation. For instance, 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.

Moreover, 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 defect definition, it can be considered as a class with abnormally large coupling.

Another observation is that, in the majority of situations, code quality can be improved without fixing maintainability defects; we only need to identify if the code modification effectively corrects some specific defects. In addition, the code quality is estimated using quality metrics, but different problems are related to this: how to determine the useful metrics for a given system and how to best combine multiple metrics to detect or correct defects.

2.2.2 Correction issues

The corrective solutions should not be specific to some defect types. Indeed, manually specifying a "standard" refactoring solution for each design defect can be a difficult task. In the majority of cases, these "standard" solutions can remove all symptoms of a defect, but this does not necessarily mean that the defect is corrected.

Another issue that the correction process should consider is related to the impact of refactoring. Correcting a defect may produce other defects. For example, moving methods to reduce the size/complexity of a class may increase the global coupling. To avoid these situations, it is important not to correct defects separately and consider the correction as a global process. In addition, different correction strategies should be defined for the same type of defect. Indeed, the list of all possible correction strategies, for each defect type, can be very large.[1] Thus, the process of manually defining correction strategies for each defect from an exhaustive list of refactoring-sis complex, time-consuming and error-prone. The manual evaluation of refactoring strategies needs a large effort to choose the best one. Finally, improving the quality and reducing the costs associated to refactoring of software are complementary and sometimes conflicting considerations. In some cases, correcting some anomalies corresponds to re-implementing a large portion of the system. In such situations, the best refactoring solution that improves code quality is not necessarily the most useful due

[1] http://www.refactoring.com/catalog/.

to the high modification cost. For this reason, we need to find a compromise between improving code quality and reducing the modification effort. Although there is no consensual way to select the best refactoring from large lists of candidates, existing approaches do not fully and explicitly consider the refactoring effort.

These observations are at the origin of the work described in this paper as described in the next section.

3 Approach overview

To address or at least circumvent the above-mentioned issues, we propose an approach in two steps:

(1) Detection of defects: we use examples of already detected software design defects to derive automatically detection rules.
(2) Correction of defects: we use the derived detection rules in the first step to select the best refactoring solution from a list, which maximizes the quality (by minimizing the number of detected defects) and at the same time minimizes the effort.

The general structure of our approach is illustrated in Fig. 1. The following two subsections give more details about the two steps.

3.1 Detection of maintainability defects

In this step, the knowledge from defect examples is used to generate detection rules. The detection step takes as inputs a base (i.e. a set) of defect examples and takes as controlling parameters a set of quality metrics (the expressions and the usefulness of these metrics were defined and discussed in the literature (Gaffney 1981)). This step generates a set of rules. The rule generation process chooses randomly from the

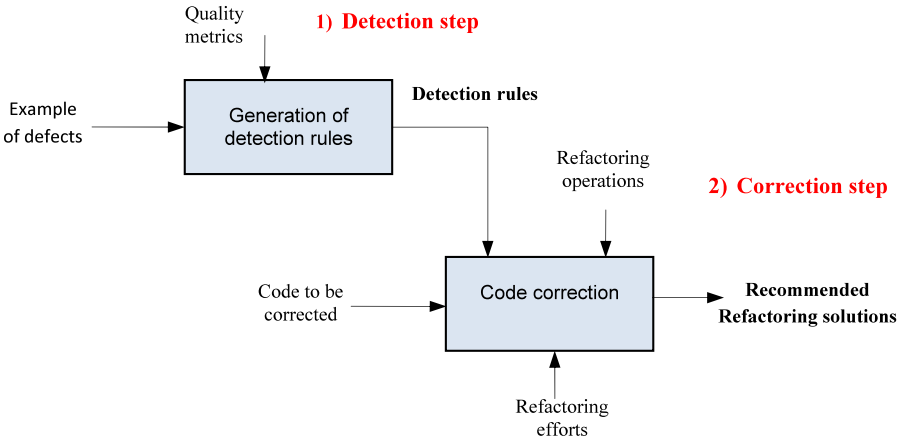


Fig. 1 Overview of the approach general architecture

metrics list a combination of quality metrics (and their threshold values) to detect a specific defect. Consequently, a solution to the defect detection problem is a set of rules that best detect the defects of the base of examples. For example, the following rule states that a class c having more than 10 attributes and 20 methods is considered as a blob defect:

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

In this rule example, 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 defect. A class will be detected as a blob whenever both thresholds of 10 attributes and 20 methods are exceeded.

Defect examples are in general available in repositories of new software projects under development, or previous projects under maintenance. Defects are generally documented as part of the maintenance activity, and can be found in version control logs, incident reports, inspection reports, etc. The use of such examples has many benefits. First, it allows deriving defect detection rules that are closer to, and more respectful of the programming "traditions" of software-development teams of specific companies. These rules will be more precise and more context faithful, yet almost without loss of generality, than more general rules, generated independently of any context. Second, it solves the problem of defining the values of the detection thresholds since these values will be found during the rule generation process. These thresholds will then correspond more closely to the company best practices. Finally, learning from examples allows reducing the list of detected defect candidates.

The rule generation process is executed periodically over large periods of time using the base of examples. The generated rules are used to detect the defects of any system that needs to be evaluated (in the sense of defect detection and correction). The rule generation step needs to be re-executed only if the base of examples is updated with new defected instances. In fact, adding new defect examples improves the quality of the detection rules.

3.2 Correction of maintainability defects

The correction step takes as controlling parameters the generated detection rules and a set of refactoring operations in the scientific literature (Fowler 1999). Each refactoring operation has a weight that corresponds to the cost of applying it. The correction step takes as input a software code to be corrected and recommends a refactoring solution consisting of refactoring operations to apply in order to correct the input code.[2] The correction step first uses the detection rules to detect defects in the input software code. Then, the process of generating a correction solution can be viewed as the mechanism that finds the best way to combine some subset (to be fixed during the search), among all available refactoring operations, in such a way to best reduce the number of detected defects and the cost of applying the refactoring sequence. This

[2] http://www.refactoring.com/catalog/.

process can be seen as a compromise between maximizing the quality and minimizing the code adaptability effort.

We use logic predicates (Bratko and Muggleton 1995) to represent refactoring operations. For example, the following predicate indicates that the method "division" is moved from the class "department" to class "university":

MoveMethod(division, department, university)

The correction step is aimed to be executed more frequently than the rule generation step, i.e., each time the evaluation (in the sense of defect detection and correction) of a software code is needed.

3.3 Problem complexity

In the detection step, our approach assigns a threshold value randomly to each metric and combines these threshold values within logical expressions (union OR; intersection AND) to create rules. The number m of possible threshold values is usually very large and the rule generation process consists of finding the best combination between n metrics. In this context, the number NR of possible combinations that have to be explored is given by:

$$NR = (n!)^m \tag{1}$$

This value quickly becomes huge. For example, a list of 5 metrics with 6 possible thresholds necessitates the evaluation of up to 120^6 combinations.

Consequently, the rule generation process is a combinatorial optimization problem. Due to the huge number of possible combinations, a deterministic search is not practical, and the use of a heuristic search is warranted. To explore the search space, we use a global heuristic search by means of genetic programming (Koza 1992). This algorithm will be detailed in Sect. 4.

As far as the correction step is concerned, the size of the search space is determined not only by the number of possible refactoring combinations, but also by the order in which they are applied. Formally, if k is the number of available refactoring operations, then the number NS of possible refactoring solutions is given by all the permutations of all the possible subsets and is at least equal to:

$$NS = (k!)^k \tag{2}$$

This number can also be huge since the same refactoring operations can be applied several times in different code fragments.

Due to the large number of possible refactoring solutions, another heuristic-based optimization method is used to generate solutions. As any solution must satisfy two criteria (quality and effort), we propose to consider the refactoring search as a multi-objective optimization problem instead of a single-objective one. To this end, we propose an adaptation of the non-dominated sorting genetic algorithm (NSGA-II) proposed in Deb et al. (2002). This algorithm and its adaptation to the refactoring problem are described in Sect. 4.

4 Refactoring by example

This section describes how genetic programming (GP) can be used to generate rules to detect maintainability defects, and how the non-dominated sorting genetic algorithm (NSGA-II) (Deb et al. 2002) can be adapted to find refactoring solutions. To apply GP and NSGA-II to a specific problem, the following elements have to be defined:

- Representation of the individuals,
- Creation of a population of individuals,
- Definition of the fitness function to evaluate individuals for their ability to solve the problem under consideration,
- Selection of the individuals to transmit from one generation to another,
- Creation of new individuals using genetic operators (crossover and mutation) to explore the search space,
- Generation of a new population.

The next sections explain the adaptation of the design of these elements for both the automatic generation of detection rules using GP, and suggestion of refactoring solutions using NSGA-II.

4.1 Defects detection using genetic programming

Genetic programming is a powerful heuristic search optimization method inspired by the Darwinian theory of evolution (Goldberg 1989). The basic idea is to explore the search space by making a population of candidate solutions, also called individuals, evolve toward a "good" solution of a specific problem.

In genetic programming, a solution is a (computer) program which is usually represented as a tree, where the internal nodes are functions, and the leaf nodes are terminal symbols. Both the function set and the terminal set must contain elements that are appropriate for the target problem. For instance, the function set can contain arithmetic operators, logic operators, mathematical functions, etc.; whereas the terminal set can contain the variables (attributes) of the target problem. Every individual of the population is evaluated by a fitness function that determines a quantitative measure of its ability to solve the target problem. The exploration of the search space is achieved by the evolution of candidate solutions using selection and genetic operators such as crossover and mutation. The selection operator insures selection of individuals in the current population proportionally to their fitness values, so that the fitter an individual is, the higher the probability is that it be allowed to transmit its features to new individuals by undergoing crossover and/or mutation operators. The crossover operator insures the generation of new children, or offspring, based on parent individuals. The crossover operator allows transmission of the features of the best fitted parent individuals to new individuals. This is usually achieved by replacing a randomly selected sub tree of one-parent individual with a randomly chosen sub tree from another parent individual to obtain one child. A second child is obtained by inverting parents. Finally, the mutation operator is applied, with a probability which is usually inversely proportional to its fitness value, to modify some randomly selected nodes in a single

Algorithm: DefectDetection

Input:
Set of quality metrics
Set of defect examples
Process:
1. I := rules(R, Defect_Type)
2. P := set_of(I)
3. initial_population(P, Max_size)
4. repeat
5. for all I in P do
6. detected_defects := execute_rules(R, I)
7. fitness(I) := compare(detected_defects, defect_examples)
8. end for
9. best_solution := best_fitness(I);
10. P := generate_new_population(P)
11. $it := it + 1$;
12. until $it = \max\_it$
13. return best_solution
Output:
best_solution: detection rule

Fig. 2 High-level pseudo-code for GP adaptation to our problem

individual. The mutation operator introduces diversity into the population and allows escaping from local solutions found during the search.

Once the selection, mutation and crossover operators have been applied with given probabilities, the individuals in the newly created generation are evaluated using the fitness function. This process is repeated iteratively, until a stopping criterion is met. The criterion usually corresponds to a fixed number of generations. The result of genetic programming (the best solution found) is the fittest individual produced along all generations.

4.1.1 GP algorithm overview

A high level view of the GP approach to the defect detection problem is summarized in Fig. 2. As this figure shows, the algorithm takes as input a set of quality metrics and a set of defect examples that were manually detected in some systems, and finds a solution that corresponds to the set of detection rules that best detect the defects in the base of examples.

Lines 1–3 construct the initial GP population which is a set of individuals that define possible detection rules. The function rules(R, Defect_Type) returns an individual I by randomly combining a set of metrics/thresholds that corresponds to a specific defect type, e.g., blob, spaghetti code or functional decomposition. The function set_of(I) returns a set of individuals, i.e., detection rules, that corresponds to a GP population. Lines 4–13 encode the main GP loop, which explores the search space and constructs new individuals by combining metrics within rules. During each

R1: IF (LOCCLASS(c) $\geq$ 1500 AND LOCMETHOD(m, c) $\geq$ 129) OR (NMD(c) $\geq$ 100) THEN blob(c)
R2: IF (LOCMETHOD(m, c) $\geq$ 151) THEN spaghetti code(c)
R3: IF (NPRIVFIELD(c) $\geq$ 7 AND NMD(c) $=$ 16) THEN functional decomposition(c)

Fig. 3 Rule interpretation of an individual

iteration, we evaluate the quality of each individual in the population, and save the individual having the best fitness (line 9). We generate a new population ($p+1$) of individuals (line 10) by iteratively selecting pairs of parent individuals from population p and applying the crossover operator to them; each pair of parent individuals produces two children (new solutions). We include both the parent and child variants in the new population p. Then, we apply the mutation operator with a probability score for both parent and child to ensure the solution diversity; this produces the population for the next generation. The algorithm terminates when the termination criterion (maximum iteration number) is met, and returns the best set of detection rules (best solution found during all iterations).

4.1.2 Genetic programming adaptation

The following three subsections describe more precisely our adaption of GP to the defect detection problem.

I. *Individual representation*

An individual is a set of IF–THEN rules. For example, Fig. 3 shows the rule interpretation of an individual.

Consequently, a detection rule has the following structure:

IF "Combination of metrics with their threshold values" THEN "Defect type"

The IF clause describes the conditions or situations under which a defect type is detected. These conditions correspond to logical expressions that combine some metrics and their threshold values using logic operators (AND, OR). If some of these conditions are satisfied by a class, then this class is detected as the defect figuring in the THEN clause of the rule. Consequently, THEN clauses highlight the defect types to be detected. We will have as many rules as types of defects to be detected. In our case, mainly for illustrative reasons, and without loss of generality, we focus on the detection of three defect types, namely blob, spaghetti code and functional decomposition. Consequently, as it is shown in Fig. 4, we have three rules, R1 to detect blobs, R2 to detect spaghetti codes, and R3 to detect functional decomposition.

One of the most suitable computer representations of rules is based on the use of trees (Davis et al. 1977). In our case, the rule interpretation of an individual will be handled by a tree representation which is composed of two types of nodes: terminals and functions. The terminals (leaf nodes of a tree) correspond to different quality metrics with their threshold values. The functions that can be used between these metrics correspond to logical operators, which are Union (OR) and Intersection (AND).

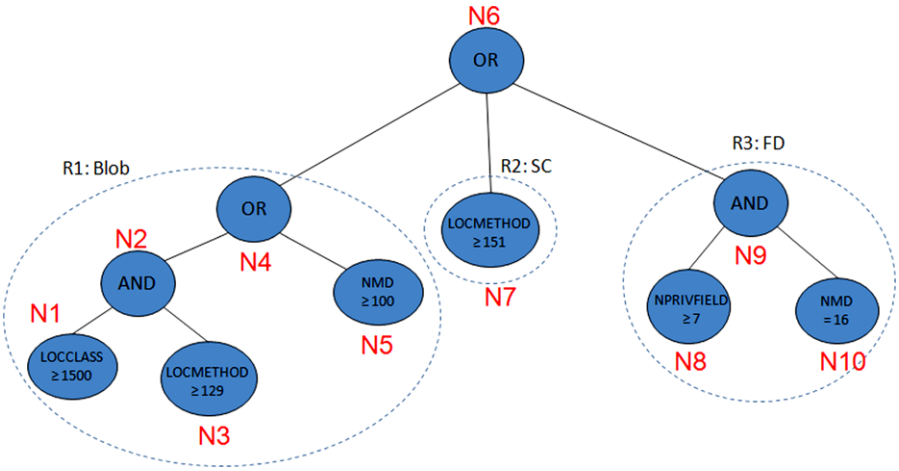


Fig. 4 A tree representation of an individual

Consequently, the rule interpretation of the individual of Fig. 3 has the following tree representation of Fig. 4. This tree representation corresponds to an OR composition of three sub-trees, each sub tree representing a rule: R1 OR R2 OR R3.

For instance, rule R1 is represented as a sub-tree of nodes starting at the branch (N1–N5) of the individual tree representation of Fig. 3. Since this rule is dedicated to detect blob defects, we know that the branch (N1–N5) of the tree will figure out the THEN clause of the rule. Consequently, there is no need to add the defect type as a node in the sub-tree dedicated to a rule.

II. *Generation of an initial population*

To generate an initial population, we start by defining the maximum tree length including the number of nodes and levels. The actual tree length will vary with the number of metrics to use for defect detectionNotice that a high tree length value does not necessarily mean that the results are more precise since, usually, only a few metrics are needed to detect a specific defect. These metrics can be specified either by the user or determined randomly. Because the individuals will evolve with different tree lengths (structures), with the root (head) of the trees unchanged, we randomly assign for each one:

- one metric and threshold value to each leaf node
- a logic operator (AND, OR) to each function node

Since any metric combination is possible and correct semantically, we do need to define some conditions to verify when generating an individual.

III. *Genetic operators*

Selection To select the individuals that will undergo the crossover and mutation operators, we used stochastic universal sampling (SUS) (Koza 1992), in which the probability to select an individual is directly proportional to its relative fitness

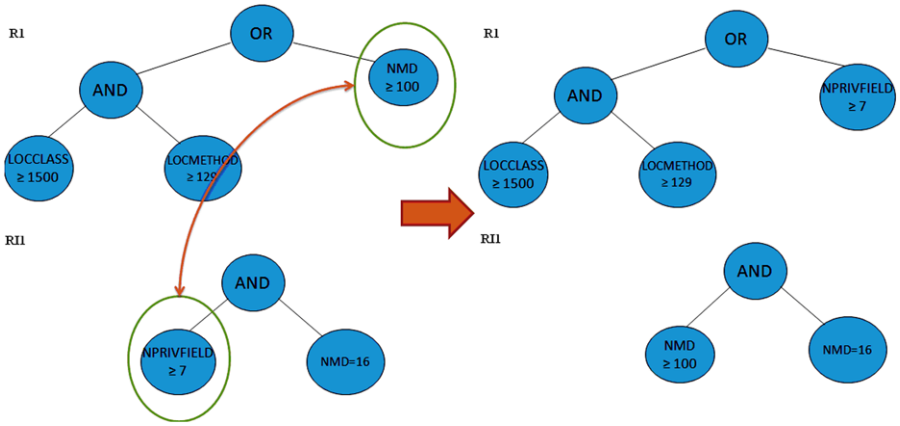


Fig. 5 Crossover operator

in the population. For each iteration, we used SUS to select *population_size/2* individuals from population p to form population $p+1$. These (population_size/2) selected individuals will "give birth" to another (population_size/2) new individuals using crossover operator.

Crossover Two parent individuals are selected, and a sub tree is picked on each one. Then, the crossover operator swaps the nodes and their relative sub trees from one parent to the other. The crossover operator can be applied only on parents having the same type of defect to detect. Each child thus combines information from both parents.

Figure 5 shows an example of the crossover process. In fact, the rule R1 and a rule RI1 from another individual (solution) are combined to generate two new rules. The right sub tree of R1 is swapped with the left sub tree of RI1.

As result, after applying the cross operator the new rule R1 to detect blob will be:

R1: IF (LOCCLASS(c) $\geq$ 1500 AND LOCMETHOD(m, c) $\geq$ 129)) OR (NPRIVFIELD(c) $\geq$ 7) THEN blob(c)

Mutation The mutation operator can be applied either to function or terminal nodes. This operator can modify one or many nodes. Given a selected individual, the mutation operator first randomly selects a node in the tree representation of the individual. Then, if the selected node is a terminal (threshold value of a quality metric), it is replaced by another terminal. The new terminal either corresponds to a threshold value of the same metric or the metric is changed and a threshold value is randomly fixed. If the selected node is a function (AND operator, for example), it is replaced by a new function (i.e. AND becomes OR). If a tree mutation is to be carried out, the node and its sub trees are replaced by a new randomly generated sub tree.

Fig. 6 Mutation operator

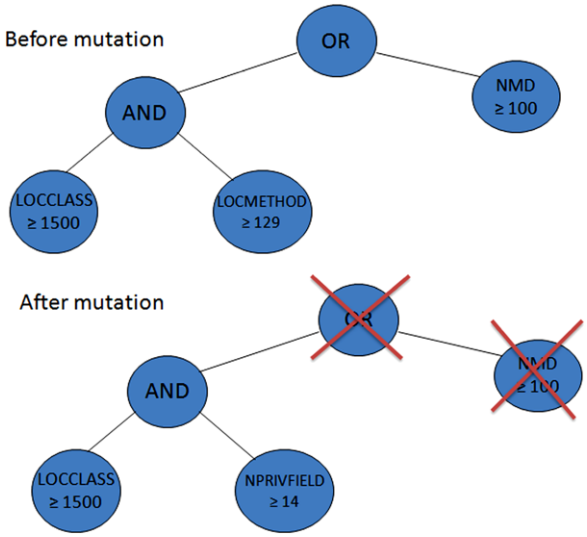
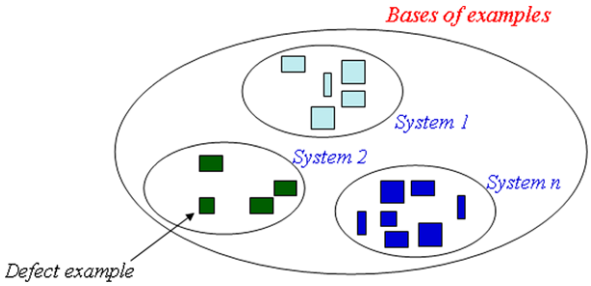


Fig. 7 Base of examples



To illustrate the mutation process, consider again the example that corresponds to a candidate rule to detect blob defects. Figure 6 illustrates the effect of a mutation that deletes the node NMD, leading to the automatic deletion of node OR (no left sub tree), and that replaces the node LOCMETHOD by node NPRIVFIELD with a new threshold value. Thus, after applying the mutation operator the new rule R1 to detect blob will be:
R1: IF (LOCCLASS(c) $\geq$ 1500 AND NPRIVFIELD(c) $\geq$ 14)) THEN blob(c)

IV. *Decoding of an individual*

The quality of an individual is proportional to the quality of the different detection rules composing it. In fact, the execution of these rules on the different projects extracted from the base of examples detects various classes as defects. Then, the quality of a solution (set of rules) is determined with respect to the number of detected defects in comparison to the expected ones in the base of examples. In other words, the best set of rules is the one that detects the maximum number of defects.

Considering the example of Fig. 7, let us suppose that we have a base of defect examples having three classes X, W, T that are considered respectively as blob,

Table 1 Defects example

Class	Blob	Functional decomposition	Spaghetti code
Student		X	
Person		X	
University		X	
Course	X		
Classroom			X
Administration	X		

Table 2 Detected classes

Class	Blob	Functional decomposition	Spaghetti code
Person		X	
Classroom	X		
Professor		X	

functional decomposition and another blob. A solution contains different rules that detect only X as blob. In this case, the quality of this solution will have a value of $1/3 = 0.33$ (only one detected defect over three expected ones).

V. *Evaluation of an individual*

The encoding of an individual should be formalized in a fitness function that quantifies the quality of the generated rules. The goal is to define an efficient and simple (in the sense of not computationally expensive) fitness function in order to reduce computational complexity.

As discussed in Sect. 3, the fitness function aims to maximize the number of detected defects in comparison to the expected ones in the base of examples. In this context, we define the fitness function of a solution, normalized in the range [0, 1], as:

$$f_{norm} = \frac{\frac{\sum_{i=1}^{p} a_i}{t} + \frac{\sum_{i=1}^{p} a_i}{p}}{2} \tag{3}$$

Where t is the number of defects in the base of examples, p is the number of detected classes with defects, and a_i has value 1 if the ith detected class exists in the base of examples (with the same defect type), and value 0 otherwise.

To illustrate the fitness function, we consider a base of examples containing one system evaluated manually. In this system, six (6) classes are subject to three (3) types of defects as shown in Table 1.

The classes detected after executing the solution generating the rules R1, R2 and R3 of Fig. 4 are described in Table 2.

Thus, only one class corresponds to a true defect (Person). Classroom is a defect but the type is wrong and Professor is not a defect. The fitness function

has the value:

$$f_{norm} = \frac{\frac{1}{3} + \frac{1}{6}}{2} = 0.25$$

With $t = 3$ (only one defect is detected over 3 expected defects), and $p = 6$ (only one class with a defect is detected over 6 expected classes with defects).

4.2 Maintainability defects correction using multi-objective optimization

In this section, we describe the Non-dominated Sorting Genetic Algorithm NSGA II used to correct the detected defects. This algorithm takes into consideration two objectives: (1) maximizing the code quality (minimizing the number of defects); (2) minimizing code changes (effort).

4.2.1 NSGA-II overview

NSGA-II (Deb et al. 2002) is a powerful search method inspired from natural selection. The basic idea is to make a population of candidate solutions evolve toward the best solution in order to solve a multi-objective optimization problem. NSGA-II was designed to be applied to an exhaustive list of candidate solutions, which creates a large search space.

The main idea of NSGA-II is to calculate the Pareto front that corresponds to a set of optimal solutions called non-dominated solutions or Pareto set. A non-dominated solution is one which provides a suitable compromise between all objectives without degrading any of them. Indeed, the concept of Pareto dominance consists of comparing each solution x with every other solution in the population until it is dominated by one of them. If any solution does not dominate it, the solution x will be considered non-dominated and will be selected by the NSGA-II to be a member of the Pareto front. If we consider a set of objectives f_i, $i \in 1, \ldots, n$, to maximize, a solution x dominates x'

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

The first step in NSGA-II is to randomly create the initial population P_0 of individuals encoded using a specific representation. Then, a child population Q_0 is generated from the population of parents P_0 using genetic operators such as crossover and mutation. Both populations are merged and a subset of individuals is selected, based on the dominance principle to create the next generation. This process will be repeated until reaching the last iteration according to stop criteria.

To be applied, NSGA-II needs to specify some elements for its implementation: (1) the representation of individuals used to create a population; (2) a fitness function to evaluate the candidate solutions according to each objective; (3) the crossover and mutation operators that have to be designed according to the individual's representation. In addition, a method to select the best individuals has to be implemented to create the next generation of individuals. The result of NSGA-II is the best individuals (with highest fitness scores) produced along all generations. In the following subsections, we show how we adapted all of these concepts to guide our search-based refactoring approach.

Algorithm: DefectCorrection

Input:
Code with design defects,
Set of refactoring operations RO,
Effort values for each operation in RO,
Defect detection rules D,

Process:
1. initial_population(P, Max_size)
2. P_0 := set_of(S)
3. S := sequence_of(RO)
4. code := defect_Code
5. $t := 0$
6. repeat
7. Q_t := Gen_Operators(P_t)
8. $R_t := P_t U Q_t$
9. for all $S_i \in R_t$ do
10. code := execute_refactoring(S, defect_Code);
11. $Quality(S_i)$:= calculate_Quality(D, code);
12. $Effort(S_i)$:= calculate_Effort();
13. end for
14. F := fast-non-dominated-sort(R_t)
15. $P_{t+1} := \emptyset$
16. while $|P_{t+1}|$ <Max_size
17. F_i := crowding_distance_assignment(F_i)
18. $P_{t+1} := P_{t+1} + F_i$
19. end while
20. $P_{t+1} := P_{t+1}[0 : \text{Max\_size}]$
21. $t := t + 1$;
22. until $t = \max\_it$
23. best_solution = First_front(R_t)
24. return best_solution

Output:
best_solution: Near-Optimal refactoring sequences

Fig. 8 High-level pseudo-code for NSGA-II adaptation to our problem

4.2.2 Adaptation

We adapted the NSGA-II to the problem of correcting detected design defects that can exist in a given system, taking into consideration both quality and effort dimensions. As our aim is to maximize the quality and minimize the correction effort, we consider each one of these criteria as a separate objective for NSGA-II. The pseudo-code for the algorithm is given in Fig. 8.The algorithm takes as input defective code fragments to be corrected, a set of possible refactoring operations RO and its associated efforts, and a set of defect detection rules D. Lines 1–5 construct an initial solution population based on a specific representation, using the list of RO given at the input. This initial

RO1	moveMethod
RO2	pullUpAttribute
RO3	deleteGeneralization
RO4	extractClass
RO5	addRelationShip
RO6	inlineClass
RO7	extractSuperClass
RO8	inlineMethod
RO9	extractClass
RO10	PushDownMethod
RO11	pullUpMethod
RO12	ExtractClass

Fig. 9 Representation of an NSGA-II individual

population stands for a set of possible defect-correction solutions returned by the function set_of(S), each one representing sequences of RO selected and combined randomly using the function sequence_of(RO).

Lines 6–22 encode the main NSGA-II loop whose goal is to make a population of candidate solutions evolve toward the best sequence of RO, i.e., the one that minimizes as much as possible the number of defects and code changes (effort). During each iteration t, a child population Q_t is generated from a parent generation P_t (line 7) using genetic operators. Then, Q_t and P_t are assembled in order to create a global population R_t (line 8). After that, each solution S_i in the population R_t is evaluated using two fitness functions one for quality and the other one for effort (lines 11 and 12):

- The quality fitness function (line 11) represents the number of detected defects after applying the proposed refactoring sequence.
- The effort fitness function (line 12) calculates the effort invested to execute the refactoring sequence.

Once the quality and effort of its solution are evaluated, the solutions are sorted and a list of non-dominated fronts F is produced (line 14). Once all of the current population is sorted, the next population is created using solutions that are selected from the sorted fronts F (lines 16–19). When two solutions are in the same front, i.e. they have equal dominance, they are sorted by the crowding distance (Deb et al. 2002), a measure of density in the neighborhood of a solution. The algorithm terminates when it achieves the termination criterion (maximum iteration number; line 22). The algorithm then returns the best solutions that are extracted from the first front of the last iteration (line 23).

We give more details in the following sub-sections about the representation of solutions, genetic operators, and the fitness functions.

I. *Solution representation*

To represent a candidate solution (individual), we used a vector representation where each vector dimension represents a refactoring operation. When created, a solution will be executed in the vector order to be evaluated. An example of a solution is given in Fig. 9.

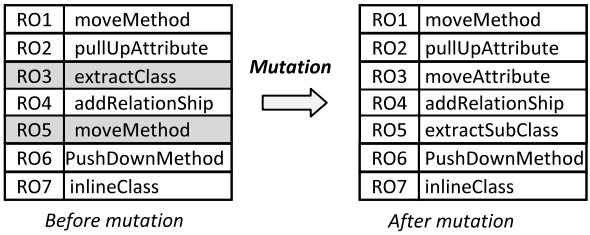


Fig. 10 Mutation operator

II. *Selection and genetic operations*

Selection There are many selection strategies where the fittest individuals are allocated more copies in the next generation than the rest of the population. To guide the selection process, NSGA-II uses a comparison operator based on the calculation of the crowding distance to select potential individuals to construct new population P_{t+1}. Furthermore, for our initial prototype, we used Stochastic Universal Sampling (SUS) (Koza 1992) to derive a child population Q_t from a parent population P_t, in which each individual's probability of selection is directly proportional to its relative overall fitness value (average score of the two fitness functions defined before) in the population. We use SUS to select elements from P_t that represents the best elements to be reproduced in the child population Q_t. The procedure uses genetic operators mutation and crossover in similar fashion to when detecting defects.

Mutation Mutation of a solution starts by randomly selecting one or more operations from its associated sequence. Then, the selected operations are replaced by other ones from the initial list of in the RO set. An example is shown in Fig. 10.

Crossover We use a single, random, cut-point crossover. The two selected parent solutions are split in two parts. Then crossover creates two child solutions by putting, for the first child, the first part of the first parent with the second part the second parent, and, for the second child, the first part of the second parent with the second part of the first parent. This operator must ensure the respect of the length limits by randomly eliminating some refactoring operations. As illustrated in Fig. 11, each child combines some refactoring operations from a parent with others from the second parent. In any given generation, a particular solution will be a parent in at most one crossover operation.

III. *Multi-criteria evaluation (fitness functions)*

In the majority of other works, one fitness function evaluates a generated solution by verifying its ability to improve the quality of the source code. In our case, two fitness functions are included in our NSGA-II adaptation: (1) the quality of the corrected code and (2) the change effort.

Quality criterion The Quality criterion is evaluated using the fitness function given in (4). The quality value increases when the number of defects in the code is reduced after correction. Quality returns a real value between 0 and 1 that

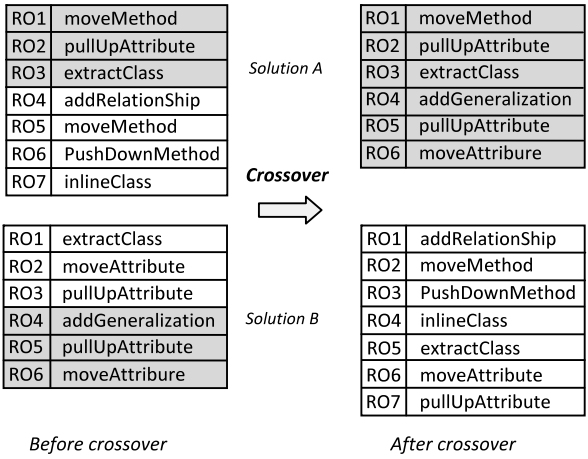


Fig. 11 Crossover operator

represents the proportion of defected classes (detected using design defects detection rules D) compared to the total number of possible defects that can be detected.

$$\text{Quality} = 1 - \frac{\text{number\_Detected\_Classes}}{\text{Total\_number\_classes} * \text{number\_Design\_defects\_Type}} \quad (4)$$

Effort criterion One important dimension that needs to be taken into account, when applying code refactoring, is the required effort. However, there exists no consensual way on how to calculate the effort to concretely apply the refactoring operations. The vast majority of the existing work proposed some model or expert-based approaches using historical data to predict/estimate effort (Menzies et al. 2006; Boehm 1981; Mehta and Heineman 2002; Boehm et al. 2000). In our approach, we propose an alternative method to estimate effort according to code-complexity and code-changes when applying a refactoring-operation sequence. To this end, we use a model that considers two categories of operations in RO. Therefore, an operation can be a Low-Level Refactoring (LLR) or a High-Level Refactoring (HLR). HLR is composed by the combination of two or more operations; LLR is an elementary refactoring.

For each low-level operation, we manually attribute an effort value (equal to 1, 2, or 3) based on our expertise on performing such an operation. This value reflects the amount of code that needs to be created, modified or inspected. We consider, in particular, the type of the involved fragment (parameter, field, method, class, etc.) and the possible change impact. Then, for each high-level refactoring, we derive the effort by cumulating the effort values of its contained low-level operations.

Table 1 shows our RO classification. We classify the following basic operations in the LLR category (columns): Create_class, delete_method, add_field, move_method, rename_method, create_relationship, etc. The effort value is given for each LLR. HLR category (rows) contains Extract_class, Extract_subclass, Pull up method, Push down Field, etc. Each of these elements is de-

scribed in terms of LLR together with their frequencies (cells). The derivation of the effort for a HLR is done according to (4).

$$\text{Effort}_{\text{HLR}} = \sum_{i=1}^{p} (n_i * \text{Effort}_{\text{LLR}_i}) \tag{5}$$

Where p is the number of different low-level refactorings to execute and n_i is the number of times that we need to execute the LLR_i. As stated before and described in Table 3, the effort LLR_i is given (between 1, 2, or 3) regarding different criteria that include the type of the involved fragment (parameter, field, method, class, etc.) and the possible change impact.

5 Validation

To test our approach, we studied its usefulness to guide quality assurance efforts for some open-source programs. In this section, we describe our experimental setup and present the results of an exploratory study.

5.1 Goals and objectives

The goal of the study is to evaluate the efficiency of our approach for the detection and correction of maintainability defects from the perspective of a software maintainer conducting a quality audit. We present the results of the experiment aimed at answering the following research questions:

RQ1: To what extent can the proposed approach detect maintainability defects?
RQ2: What types of defects does it locate correctly?
RQ3: To what extent can the proposed approach correct detected defects with a minimal effort?

To answer RQ1, we used an existing corpus of known maintainability defects (Moha et al. 2009) to evaluate the precision and recall of our approach. We compared our results to those produced by an existing rule-based strategy (Moha et al. 2009). To answer RQ2, we investigated the type of defects that were found. To answer RQ3, we validated manually if the proposed corrections are useful to fix detected defects with low efforts.

5.2 Systems studied

We used six open-source Java projects to perform our experiments: GanttProject (Gantt for short) v1.10.2, Quick UML v2001, AZUREUS v2.3.0.6, LOG4J v1.2.1, ArgoUML v0.19.8, and Xerces-J v2.7.0. Table 3 provides some relevant information about the programs. The base of defect examples contains more than examples as presented in Table 3.

We chose the Xerces-J, ArgoUML, LOG4J, AZUREUS, Quick UML and Gantt libraries because they are medium-sized open-source projects and were analyzed in

Table 3 Refactoring operations composition and its associated effort values

High level refactoring	Low level refactorings												
	Create class	Delete class	Add method	Delete method	Add field	Delete field	Create relationship	Delete relationship	Hide method	Add parameter	Remove parameter	Modify cardinality	Rename method
Effort value	**2**	**3**	**1**	**3**	**1**	**2**	**1**	**2**	**1**	**1**	**2**	**1**	**1**
Encapsulate_Field			*n*		*n*	*n*							
Extract Class	1		*n*	*n*	*n*	*n*	*n*	*n*					
Extract interface	1		*n*	*n*	*n*	*n*	*n*	*n*				*n*	
Extract method			1	*n*									
Extract subclass	*n*		*n*		*n*		*n*		*n*				*n*
Extract superclass	1		*n*		*n*		*n*		*n*				*n*
Inline class		1	*n*	*n*	*n*	*n*		1	*n*				
Inline method				1						*n*	*n*		
Move class			*n*	*n*	*n*	*n*	*n*	*n*				*n*	
Move Method					1	1							
Move Field			1	1									*n*
Parametrize_Method			*n*	*n*						*n*			
Pull up Field					*n*	*n*							
Pull up Method			*n*	*n*									*n*
Push down Field					*n*	*n*							
Push_down_Method			*n*	*n*									*n*
Replace_Data_Value_with_Object	1				*n*		1						
Replace_subclass_with_field		*n*	*n*	*n*	*n*	*n*		*n*					
Replace_Type_Code_With_SubClass	*n*				*n*	*n*	*n*						

Table 4 Program statistics

Systems	Number of classes	KLOC	Number of defects
GanttProject v1.10.2	245	31	41
Xerces-J v2.7.0	991	240	66
ArgoUML v0.19.8	1230	1160	89
Quick UML v2001	142	19	11
LOG4J v1.2.1	189	21	17
AZUREUS v2.3.0.6	1449	42	93

related work. The version of Gantt studied was known to be of poor quality, which led to a major revised version. ArgoUML, Xerces-J, LOG4J, AZUREUS and Quick UML, on the other hand, has been actively developed over the past 10 years and their design has not been responsible for a slowdown of their developments.

In Moha et al. (2009), the authors asked three groups of students to analyze the libraries to tag instances of specific antipatterns to validate their detection technique, DECOR. For replication purposes, they provided a corpus of describing instances of different antipatterns that includes blob classes, spaghetti code, and functional decompositions. As described in Sect. 2, blobs are classes that do or know too much; spaghetti Code (SC) is a code that does not use appropriate structuring mechanisms; finally, functional decomposition (FD) is a code that is structured as a series of function calls. These represent different types of design risks. In our study, we verified the capacity of our approach to locate classes that corresponded to instances of these antipatterns. We used a 6-fold cross validation procedure. For each fold, one open source project is evaluated by using the remaining five systems as the base of examples. For example, Xerces-J is analyzed using detection rules generated from some defect examples from ArgoUML, LOG4J, AZUREUS, Quick UML and Gantt.

The obtained results were compared to those of DECOR. Since (Moha et al. 2009) reported the number of antipatterns detected, the number of true positives, the recall (number of true positives over the number of maintainability defects) and the precision (ratio of true positives over the number detected), we determined the values of these indicators when using our algorithm for every antipattern in Xerces-J, AZUREUS, LOG4J, Quick UML, ArgoUML and Gantt.

To validate the correction step, we verified manually the feasibility of the different proposed refactoring sequences for each system. We applied the proposed refactorings using ECLIPSE. Some semantic errors (programs behavior) were found. When a semantic error is found manually, we consider the operations related to this change as a bad recommendation. We calculate a correctness precision score (ratio of possible refactoring operations over the number of proposed refactoring) as performance indicator of our algorithm.

5.3 Comparative results

Table 4 summarizes our findings. For Gantt, our average antipattern detection precision was 94%. DECOR, on the other hand, had a combined precision of 59% for the

same antipatterns. The precision for Quick UML was about 86%, over twice the value of 43% obtained with DECOR. In particular, DECOR did not detect any spaghetti code in contradistinction with our approach. For Xerces-J, our precision average was 90%, while DECOR had a precision of 67% for the same dataset. Finally, the comparison results were mixed for ArgoUML, AZUREUS and LOG4J; still, our precision was consistently higher than 75% in comparison to DECOR. On the negative side, our obtained recall score for the different systems was systematically less than that of DECOR. In fact, the rules defined in DECOR are large and this may explain the lower score in terms of precision. However, in such situations DECOR has better results. The main reason that our approach finds better precision results is that the threshold values are well-defined using our genetic algorithm. Indeed, with DECOR the user should test different threshold values to find the best ones. Thus, it is a fastidious task to find the best threshold combination for all metrics. The blob defect is detected better using DECOR because it is easy to find the thresholds and metrics combination for this kind of defects. The hypothesis to have 100% of recall justifies low precision, sometimes, to detect defects. In fact, there is a compromise between precision and recall. The detection of FDs by only using metrics seems difficult. This difficulty is alleviated in DECOR by including an analysis of naming conventions to perform the detection process. However, using naming conventions lead to results that depend on the coding practices of the development team. We obtained comparable results without having to leverage lexical information. We can also mention that fixed defects correspond to the different defect types.

In the context of this experiment, we can conclude that our technique is able to identify design anomalies, in average, more accurately than DECOR (answer to research question RQ1 above) (Table 5).

As described in Table 6, we compared our genetic algorithm (GP) detection results with those obtained by another local search algorithm, simulated annealing (SA). The detection results for SA are also acceptable. For small systems, the precision when using SA is even better than with GP. In fact, GP is a global search that performs best in a large search space (which corresponds to large systems). In addition, the solution representation used in GP (tree) is suitable for rule generation, while SA uses a vector-based representation that is not. Furthermore, SA takes a lot of time, comparing to GP, to converge to an optimal solution (more than 10 minutes).

In this work, we adapted the NSGA-II to our search-based refactoring approach with bi-objective optimization. We have also obtained very good results for the correction step. As shown in Table 7, the majority of proposed refactorings is feasible and improves the code quality. For example, for Gantt, 34 refactoring operations are feasible over the 41 proposed. After applying by hand the feasible refactoring operations for all systems, we evaluated manually that, on average, more than 75% of detected defects was fixed. The majority of non-fixed defects were related to the blob type. Indeed, this type of defect requires a large number of refactoring operations, and it is very difficult to correct.

Since the search space is two-dimensional (quality and effort); the results can be visualized as illustrated in Fig. 12.

NSGA-II does not produce a single solution as GA, but a set of solutions. As shown in Fig. 12, NSGA-II converges to Pareto-optimal solutions that are considered

Table 5 Detection results

System	Precision	Precision-DECOR	Recall	Recall-DECOR
GanttProject	Blob: 100%	90%	100%	
	SC: 93%	71.4%	97%	
	FD: 91%	26.7%	94%	
Xerces-J	Blob: 97%	88.6%	100%	
	SC: 90%	60.5%	88%	
	FD: 88%	51.7%	86%	
ArgoUML	Blob: 93%	86.2%	100%	
	SC: 88%	86.4%	91%	
	FD: 82%	38.6%	89%	
QuickUML	Blob: 94%	100%	98%	100%
	SC: 84%	0%	93%	
	FD: 81%	30%	88%	
AZUREUS	Blob: 82%	92.7%	94%	
	SC: 71%	81.7%	81%	
	FD: 68%	38.6%	86%	
LOG4J	Blob: 87%	100%	90%	
	SC: 84%	66.7%	84%	
	FD: 66%	54.5%	74%	

Table 6 Detection results

System	Precision-GP	Precision-SA
GanttProject	Blob: 100%	100%
	SC: 93%	94%
	FD: 91%	90%
Xerces-J	Blob: 97%	83%
	SC: 90%	69%
	FD: 88%	79%
ArgoUML	Blob: 93%	83%
	SC: 88%	84%
	FD: 82%	67%
QuickUML	Blob: 94%	100%
	SC: 84%	88%
	FD: 81%	83%
AZUREUS	Blob: 82%	91%
	SC: 71%	63%
	FD: 68%	54%
LOG4J	Blob: 87%	100%
	SC: 84%	88%
	FD: 66%	73%

Table 7 Correction results

System	Number of proposed refactorings	Precision
GanttProject	41	82% (34|41)
Xerces-J	66	80% (49|66)
ArgoUML	82	65% (54|82)
QuickUML	29	75% (22|29)
AZUREUS	102	71% (73|102)
LOG4J	38	81% (31|38)

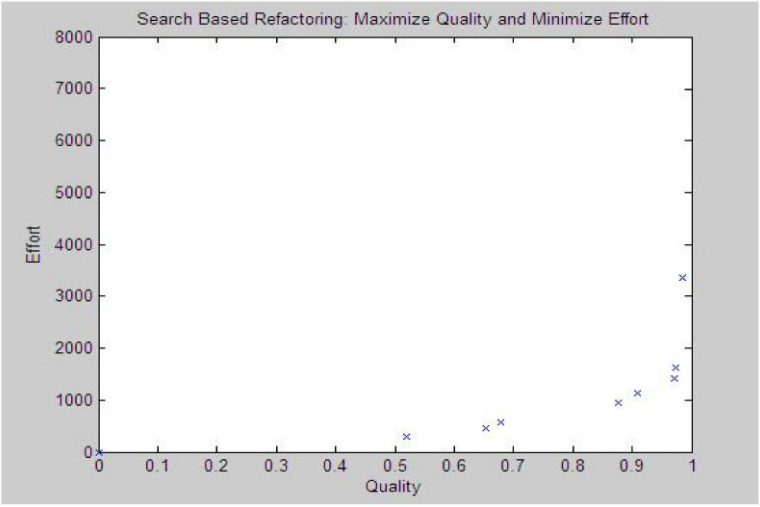


Fig. 12 Example of an NSGA-II execution

as good compromises between quality and effort as calculated using (4) and (5). In this figure, each point is a solution with the quality score represented in x-axis and the effort score in the y-axis. The best solutions exist in the right-bottom corner representing the Pareto-front that minimizes the effort value and maximizes the quality. The user can choose a solution from this front depending on his preferences in terms of compromise. However, at least for our validation, we only need to have one best solution that will be suggested by our approach. To this end, and in order to fully automate the approach, we propose to extract and suggest only one best solution from the returned set of solutions. Equation (6) is used to choose the solution that corresponds of the best compromise between Quality and Effort. In our case the ideal solution has the best quality value (equals to 1) and the best effort value (equals to 0). Hence, we select the nearest solution to the ideal one in terms of Euclidean distance. As quality and effort have different magnitudes, we normalize the effort between 0 and 1 (we divide by the maximum effort in the Pareto front) to obtain:

$$\text{bestSol} = \min_{i=0}^{n}\left(\sqrt{(1-\text{Quality}[i])^2 + (\text{Effort}[i]/\max\_\text{Eff})^2}\right) \tag{6}$$

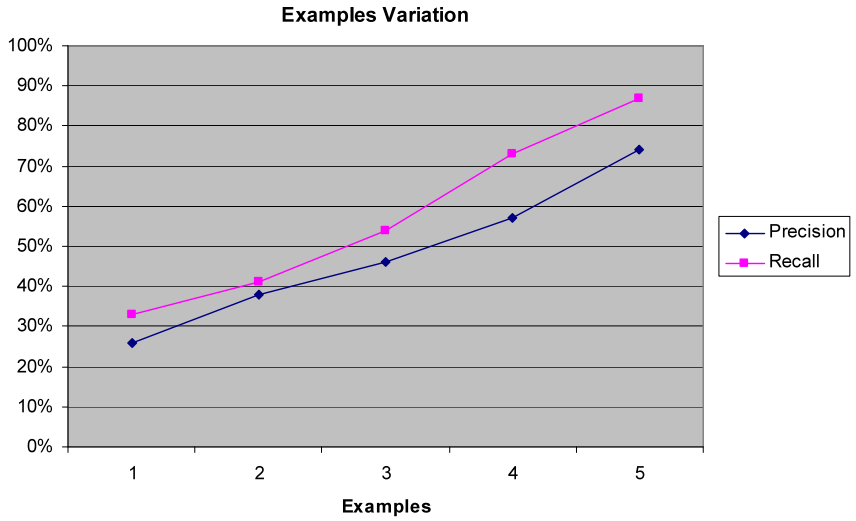


Fig. 13 Examples-size variation (example = system)

where n is the number of solutions in the Pareto front returned by NSGA-II, and max_Eff is the maximal effort value in the Pareto front.

In conclusion, our approach produces good refactoring suggestions both from the point of views of defect-correction ratio and feasibility. These results are comparable to those obtained by an approach that considers only the quality without considering the effort.

5.4 Discussion

We noticed that our technique does not have a bias towards the detection and correction of specific anomaly types. In Xerces-J, we had an almost equal distribution of each antipattern (14 SCs, 13 Blobs, and 14 FDs). On Gantt, the distribution was not as balanced, but this is principally due to the number of actual antipatterns in the system. We found all four known FDs and nine Blobs in the system, and eight of the seventeen FDs, four more than DECOR. In Quick UML, we found three out of five FDS; however DECOR detected three out of ten FDs.

An important consideration is the impact of the example base size on detection quality. Drawn for AZUREUS, the results of Fig. 13 shows that our approach also proposes good detection results in situations where only few examples are available. The precision and recall scores seem to grow steadily and linearly with the number of examples and rapidly grow to acceptable values (75%). Thus, our approach does not need a large number of examples to obtain good detection results.

The reliability of the proposed approach requires an example set of bad code. It can be argued that constituting such a set might require more work than identifying, specifying, and adapting rules. In our study, we showed that by using six open source projects directly, without any adaptation, the technique can be used out of the box and will produce good detection, correction and recall results for the detection of antipatterns for the studied systems.

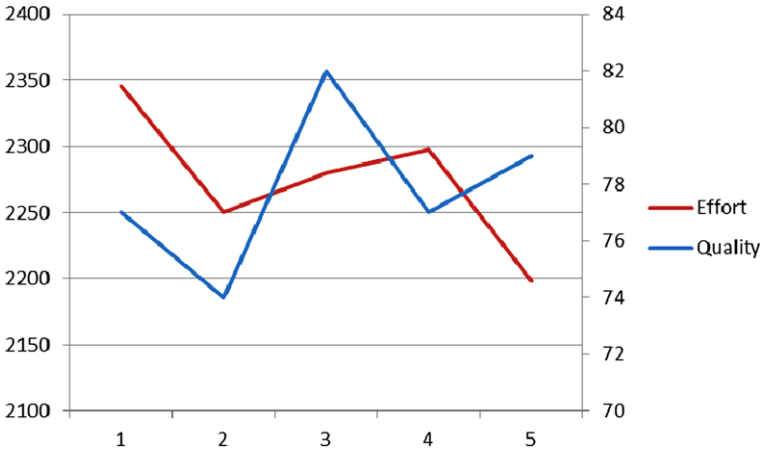


Fig. 14 NSGA-II Multiple Executions on GanttProject

In an industrial setting, we could expect that a company starts with some few open-source projects, and gradually evolves its set of bad code examples to include context-specific data. This might be essential if we consider that different languages and software environments have different best/worst practices.

The correction results might vary depending on search space exploration, since solutions are randomly generated, though guided by a meta-heuristic. To ensure that our results are relatively stable, we compared the results of multiple executions for NSGA-II as shown in Fig. 14. We, consequently, believe that our technique is stable, since the effort and quality scores are approximately the same for five different executions.

Another important advantage in comparison to machine learning techniques is that our GP algorithm does not need both positive (good code) and negative (bad code) examples to generate rules like, for example, Inductive Logic Programming (Raedt 1996).

Finally, since we viewed the problem of defect detection as a combinatorial problem addressed with heuristic search, it is important to contrast the results with the execution time. We executed our algorithm on a standard desktop computer (Pentium CPU running at 2 GHz with 3 GB of RAM). The execution time for rule generation with a number of iterations (stopping criteria) fixed to 1000 was less than fifty minutes for both detection and correction. This indicates that our approach is reasonably scalable from the performance standpoint. However, the execution time depends on the number of used metrics and the size of the base of examples. In any case, our approach is meant to apply mainly in situations where manual rule-based solutions are not easily available.

6 Related work

There are several studies that have recently focused on detecting and fixing design defects in software using different techniques. These techniques range from fully

automatic detection and correction to guided manual inspection. Other works are focused on efforts and costs estimation on maintenance stage. Nevertheless, there a very few contributions focused on taking effort into consideration when correcting design defects.

Design-defect detection and correction can be classified into three broad categories: rules-based detection-correction, detection and correction combination, and visual-based detection.

In the first category, Marinescu (2009) defined a list of rules relying on metrics to detect what he calls design flaws of OO design at method, class and subsystem levels. Erni and Lewerentz (1996) use metrics to evaluate frameworks with the goal of improving them. They introduce the concept of multi-metrics, n-tuples of metrics expressing a quality criterion (e.g., modularity). The main limitation of the two previous contributions is the difficulty to manually define threshold values for metrics in the rules. To circumvent this problem, Alikacem and Sahraoui (2006) express defect detection as fuzzy rules, with fuzzy labels for metrics, e.g., small, medium, large. When evaluating the rules, actual metric values are mapped to truth values for the labels by means of membership functions. Although no crisp thresholds need to be defined, still, it is not obvious to determine the membership functions. Moha et al. (2009), in their DECOR approach, they start by describing defect symptoms using an abstract rule language. These descriptions involve different notions, such as class roles and structures. The descriptions are later mapped to detection algorithms. In addition to the threshold problem, this approach uses heuristics to approximate some notions, which results in an important rate of false positives. Khomh et al. (2009) extended DECOR to support uncertainty and to sort the defect candidates accordingly. Uncertainty is managed by Bayesian belief networks that implement the detection rules of DECOR. The detection outputs are probabilities that a class is an occurrence of a defect type. In our approach, the above-mentioned problems related to the use of rules and metrics do not arise. Indeed, the symptoms are not explicitly used, which reduces the manual adaptation/calibration effort.

The majority of existing approaches to automate refactoring activities are based on rules that can be expressed as assertions (invariants, pre- and postconditions), or graph transformations. The use of invariants has been proposed to detect parts of program that require refactoring by Kataoka et al. (2001). Opdyke (1992) suggest the use of pre- and postcondition with invariants to preserve the behavior of the software. All these conditions could be expressed in the form of rules. Heckel (1995) considers refactorings activities as graph production rules (programs expressed as graphs). However, a full specification of refactorings would require sometimes large number of rules. In addition, refactoring-rules sets have to be complete, consistent, non redundant, and correct. Furthermore, we need to find the best sequence of applying these refactoring rules. In such situations, search-based techniques represent a good alternative. In Kessentini et al. (2010), we have proposed another approach, based on search-based techniques, for the automatic detection of potential design defects in code. The detection is based on the notion that the more code deviates from good practices, the more likely it is bad. The two approaches are completely different. We use in Kessentini et al. (2010) a good quality of examples in order to detect defects; however, in this work we use defect examples to generate rules. Both works do not need a formal definition of defects to detect them.

In the second category of work, defects are not detected explicitly. They are so implicitly because the approaches refactor a system by detecting elements to change to improve the global quality. For example, in O'Keeffe and Cinnéide (2008), defect detection is considered as an optimization problem. The authors use a combination of 12 metrics to measure the improvements achieved when sequences of simple refactorings are applied, such as moving methods between classes. The goal of the optimization is to determine the sequence that maximizes a function, which captures the variations of a set of metrics (Harman and Clark 2004). The fact that the quality in terms of metrics is improved does not necessary means that the changes make sense. The link between defect and correction is not obvious, which make the inspection difficult for the maintainers. In our case, we separate the detection and correction phases, and the search-based adaptation is completely different.

The high rate of false positives generated by the automatic approaches encouraged other teams to explore semiautomatic solutions. These solutions took the form of visualization-based environments. The primary goal is to take advantage of the human ability to integrate complex contextual information in the detection process. Kothari et al. (2004) present a pattern-based framework for developing tool support to detect software anomalies by representing potential defects with different colors. Later, Dhambri et al. (2008) propose a visualization-based approach to detect design anomalies by automatically detecting some symptoms and letting others to the human analyst. The visualization metaphor was chosen specifically to reduce the complexity of dealing with a large amount of data. Still, the visualization approach is not obvious when evaluating large-scale systems. Moreover, the information visualized is, for the most part, metric-based, meaning that complex relationships can still be difficult to detect. In our case, human intervention is needed only to provide defect examples. Finally, the use of visualization techniques is limited to the detection step.

The different defect correction works mentioned previously do not take in consideration the effort when applying refactoring operations. However, some work already exists to predict effort in software maintenance. Menzies et al. proposed a model-based approach for effort estimation. This approach offers a solution to the large effort deviation problem that makes it difficult to distinguish the performance of different effort estimation methods. Hence, the large deviation problem cannot be solved by model-based methods or by expert-based methods. The authors proposed tool called COSEEKMO that take into consideration the list of possible existing causes of large deviations. Each cause will suggest operations that might reduce the deviation. The advantage of COSEEKMO was its fully automatic analysis. However, in terms of performance, COSEEKMO take a long execution time (more than twenty-four hours). Moreover, a specific input project data format that must be preserved. Boehm (1981), Boehm et al. (2000) developed a toolkit called COCOMO for the effort estimation. The core intuition behind COCOMO-based estimation is that, as a program grows in size, the development effort grows exponentially. Nevertheless, Boehm indicates that with less-than-great developers, refactoring effort increases with the number of requirements or stories. However, this work doesn't propose an approach to address the problem of refactoring effort minimization. In Mehta and Heineman (2002), were proposed an approach that consists of an evolution methodology for refactoring legacy systems into components. This methodology reduces the costs of future

maintenance. Cost reduction has been focused only on development processing time. Kapser and Godfrey (2006) proposed a classification of code cloning patterns that can be used to reduce maintenance efforts. In fact, cloning code can lead to unused code in the system that can cause diverse problems with code comprehensibility, readability, and maintainability over the life cycle of the software system. Consequently, maintenance efforts can be increased when bugs have to be fixed several times. Furthermore, this approach addresses the effort minimization problem over the maintenance stage, but does not cover the reduction of effort when applying refactorings for a given system.

7 Conclusion

In this article, we presented a novel approach to the problem of detecting and fixing maintainability defects. Typically, researchers and practitioners try to characterize different types of common maintainability defects and present symptoms to search for in order to locate the maintainability defects in a system. In this work, we have shown that this knowledge is not necessary to perform the detection. Instead, we use examples of maintainability defects to generate detection rules. After generating the detection rules, we use them in the correction step. In fact, we start by generating some solutions that represent a combination of refactoring operations to apply. A fitness function calculates, after applying the proposed refactorings, the number of detected defects, using the detection rules and taking in consideration the effort to apply these refactorings. Our study shows that our technique outperforms DECOR (Kapser and Godfrey 2006), a state-of-the-art metric-based approach, where rules are defined manually, on its test corpus.

The proposed approach was tested on open-source systems, and the results are promising. As part of future work, we plan to extend our base of examples with additional badly-designed code in order to consider more programming contexts.

References

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

Boehm, B.: Software Engineering Economics. Prentice Hall, New York (1981)

Boehm, B., Horowitz, E., Madachy, R., Reifer, D., Clark, B.K., Steece, B., Brown, A.W., Chulani, S., Abts, C.: Software Cost Estimation with Cocomo II. Prentice Hall, New York (2000)

Bratko, I., Muggleton, S.: Applications of inductive logic programming. Commun. ACM **38**(11), 65–70 (1995)

Brown, W.J., Malveau, R.C., Brown, W.H., McCormick, H.W. III, Mowbray, T.J.: Anti Patterns: Refactoring Software, Architectures, and Projects in Crisis, 1st edn. Wiley, New York (1998)

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

Davis, R., Buchanan, B., Shortcliffe, E.H.: Production rules as a representation for a knowledge-base consultation program. Artif. Intell. **8**, 15–45 (1977)

Deb, K., Pratap, A., Agarwal, S., Meyarivan, T.: A fast and elitist multiobjective genetic algorithm: NSGA-II. IEEE Trans. Evol. Comput. **6**, 182–197 (2002)

Dhambri, K., Sahraoui, H.A., Poulin, P.: Visual detection of design anomalies. In: CSMR. IEEE, pp. 279–283 (2008)

Erni, K., Lewerentz, C.: Applying design metrics to object-oriented frameworks. In: Proc. IEEE Symp. Software Metrics. IEEE Comput. Soc., Los Alamitos (1996)
Fenton, N., Pfleeger, S.L.: Software Metrics: A Rigorous and Practical Approach, 2nd edn. International Thomson Computer Press, London (1997)
Fowler, M.: Refactoring—improving the design of existing code, 1st edn. Addison-Wesley, Reading (1999)
Gaffney, J.E.: Metrics in software quality assurance. In: Proc. of the ACM '81 Conference, pp. 126–130. ACM, New York (1981)
Goldberg, D.E.: Genetic Algorithms in Search, Optimization and Machine Learning. Addison-Wesley/Longman, Boston (1989)
Harman, M., Clark, J.A.: Metrics are fitness functions too. In: IEEE METRICS, pp. 58–69. IEEE Computer Society, Los Alamitos (2004)
Harman, M., Tratt, L.: Pareto optimal search based refactoring at the design level. In: Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'07), pp. 1106–1113 (2007)
Heckel, R.: Algebraic graph transformations with application conditions, M.S. thesis, TU Berlin, (1995)
Kapser, C., Godfrey, M.W.: Cloning considered harmful considered harmful. In: Proceedings of the 13th Working Conference on Reverse Engineering (WCRE'06), pp. 19–28. IEEE Comput. Soc., Los Alamitos (2006)
Kataoka, Y., Ernst, M.D., Griswold, W.G., Notkin, D.: Automated support for program refactoring using invariants. In: Proc. Int'l Conf. Software Maintenance, pp. 736–743. IEEE Comput. Soc., Los Alamitos (2001)
Kessentini, M., Vaucher, S., Sahraoui, H.: Deviance from perfection is a better criterion than closeness to evil when identifying risky code. In: Proc. of the International Conference on Automated Software Engineering (ASE'10) (2010)
Khomh, F., Vaucher, S., Guéhéneuc, Y.-G., Sahraoui, H.: A Bayesian approach for the detection of code and design smells. In: Proc. of the ICQS'09 (2009)
Kothari, S.C., Bishop, L., Sauceda, J., Daugherty, G.: A pattern-based framework for software anomaly detection. Softw. Qual. J. **12**(2), 99–120 (2004)
Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press, Cambridge (1992)
Liu, H., Yang, L., Niu, Z., Ma, Z., Shao, W.: Facilitating software refactoring with appropriate resolution order of bad smells. In: Proc. of the ESEC/FSE '09, pp. 265–268 (2009)
Marinescu, R.: Detection strategies: metrics-based rules for detecting design flaws. In: Proc. of ICM'04, pp. 350–359
Mehta, A., Heineman, G.T.: Evolving legacy system features into fine-grained components. In: Proceedings of the 24th International Conference on Software Engineering, pp. 417–427. ACM Press, New York (2002)
Mens, T., Tourwé, T.: A survey of software refactoring. IEEE Trans. Softw. Eng. **30**(2), 126–139 (2004)
Menzies, T., Chen, Z., Hihn, J., Lum, K.: Selecting best practices for effort estimation. IEEE Trans. Softw. Eng. **32**(11), 883–895 (2006)
Moha, N., Guéhéneuc, Y.-G., Duchien, L., Meur, A.-F.L.: DECOR: A method for the specification and detection of code and design smells. IEEE Trans. Softw. Eng. **36**, 20–36 (2009)
O'Keeffe, M., Cinnéide, M.: Search-based refactoring: an empirical study. J. Softw. Maint. **20**(5), 345–364 (2008)
Opdyke, W.F.: Refactoring: a program restructuring aid in designing object-oriented application frameworks, Ph.D. thesis, University of Illinois at Urbana-Champaign (1992)
Raedt, D.: Advances in Inductive Logic Programming, 1st edn. IOS Press, Lansdale (1996)
Sahraoui, H., Godin, R., Miceli, T.: Can metrics help to bridge the gap between the improvement of OO design quality and its automation. In: Proc. of the International Conference on Software Maintenance (ICSM'00) (2000)
Seng, O., Stammel, J., Burkhart, D.: Search-based determination of refactorings for improving the class structure of object-oriented systems. In: Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'06), pp. 1909–1916 (2006)
Zitzler, E., Thiele, L.: Multiobjective optimization using evolutionary algorithms—a comparative case study. In: Eiben, A.E., Back, T., Schoenauer, M., Schwefel, H.-P. (eds.) Parallel Problem Solving from Nature, V, pp. 292–301. Springer, Berlin (1998)