A New Design Defects Classification: Marrying Detection and Correction

Rim Mahouachi[1,*], Marouane Kessentini[2], and Khaled Ghedira[1]

[1] SOIE, University of Tunis, Tunisia
rim.mahouachi@gmail.com, khaled.ghedira@isg.rnu.tn
[2] CS, Missouri University of Science and Technology, USA
marouanek@mst.edu

Abstract. Previous work classify design defects based on symptoms (long methods, large classes, long parameter lists, etc.), and treat separately detection and correction steps. This paper introduces a new classification of defects using correction possibilities. Thus, correcting different code fragments appending to specific defect category need, approximately, the same refactoring operations to apply. To this end, we use genetic programming to generate new form of classification rules combining detection and correction steps. We report the results of our validation using different open-source systems. Our proposal achieved high precision and recall correction scores.

Keywords: Software maintenance, refactoring, search-based software engineering, genetic programming, design defects.

1 Introduction

Software systems are evolving by adding new functions and modifying existing functionalities over time. Through this evolution process, software design becomes more complex. Thus, the understandability and maintainability are difficult and fastidious tasks. So, perfective maintenance [21], defined as software product modification after delivery to improve its performance, is an important maintenance activity. However, perfective maintenance is the most expensive activity in software development [21]. 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. As a result, these practices have been studied by professionals and researchers alike with a special attention given to design-level problems, also known in the literature as defects, antipatterns [9], smells [21], or anomalies [15].

There has been much research devoted to the study these bad design practices [16, 3, 18, 7, 22]. Although bad practices are sometimes unavoidable, they should otherwise be prevented by the development teams and removed from the code base as early as possible in the design cycle. Refactoring is an effective technique to remove

[*] Corresponding author.

J. de Lara and A. Zisman (Eds.): FASE 2012, LNCS 7212, pp. 455–470, 2012.
© Springer-Verlag Berlin Heidelberg 2012

defects. It is defined as “the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure [9].” In [9], several refactoring patterns are described. It is necessary to identify the refactoring candidates that contain “bad-smells” in order to apply refactoring patterns.

There has been much work on different techniques and tools for detecting and correcting defects, with over 300 publications [14]. However, there is no existing works on using refactoring solutions to classify defects. The vast majority of these works identify key symptoms that characterize a defect using combinations of mainly quantitative, structural, and/or lexical information. Thus, the different defects are classified based on their symptoms. Then, different possible standard refactoring solutions, for each defect category, are proposed. Completely different refactoring solutions can be used to correct the same defect type.

This paper presents a novel approach to classify defects using correction possibilities. Thus, correcting different code fragments appending to specific defect category need, approximately, the same refactoring operations to apply. Our approach is based on the use of defect examples generally available in defect repositories of software development companies. In fact, we translate regularities that can be found in such defect examples into detection-correction rules. To this end, we use genetic programming [5] to generate new form of classification rules combining detection and correction steps.

The primary contributions of the paper can be summarised as follows:

1. We introduce a new defects classification approach based on correction solutions. Our proposal does not require to define the different defect types, but only to have some refactoring examples; it does not require an expert to write detection or correction rules manually; it combines detection and correction steps; and each defect category has, approximately, the same refactoring operations to apply. However, different limitations are discussed in the discussion section.
2. We report the results of an evaluation of our approach; we used classes from six open source projects, as examples of badly-designed and corrected code. We used a 6-fold cross validation procedure. For each fold, one open source project is evaluated by using the remaining two systems as bases of examples. Almost all the identified classes were found, with a precision more than 70%.
3. We report the comparison results of our new defects classification and an existing work [16].We also report the results of a further comparison between genetic programming (GP) and another heuristic search algorithm [8].

The rest of this paper is organised as follows. Section 2 is dedicated to the problem statement, while Section 3 describes our approach details. Section 4 explains the experimental method, the results of which are discussed in Section 5. Section 6 introduces related work, and the paper concludes with Section 7.

2 Problem Statement

In this section, we emphasize the specific problems that are addressed by our approach.

Although there is 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 section 5, we discuss these issues in more detail with respect to our approach.

In general, existing classification of defects are based on symptoms without taking in consideration the correction step. The two detection and correction steps are treated separately. Thus, each defect type could have different correction possibilities that are completely different: how to choose the good correction strategy?

In addition, different issues are related to defects classification based on symptoms:

- 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.
- The goal of identifying the type of defects is to help the maintainer in the correction step. However, with actual identification of defects based on symptoms only, large number of correction strategies can be proposed for the same defect type. Thus, the question is: how to choose the best refactoring solution?
- In the majority of situations, code quality can be improved without fixing maintainability defects. We need to identify if the code modification corrects some specific defects or not. In addition, the code quality is estimated using quality metrics but different problems are related to: how to determine the useful metrics for a given system and how to combine in the best way multiple metrics to detect or correct defects.
- The correction solutions should not be specific to only some defect types. In fact, specifying manually a “standard” refactoring solution for each maintainability defect can be a difficult task. In the majority of cases, these “standard” solutions can remove all symptoms for each defect. However, removing the symptoms does not mean that the defect is corrected.

In addition to these issues, the process of defining rules manually for detection or correction is complex, time-consuming and error-prone. Indeed, the list of all possible defect types or maintenance strategies can be very large [13] and each defect type requires specific rules.

3 Defects Classification Using Genetic Programming

This section shows how the above mentioned issues can be addressed using our proposal.

3.1 Overview

The general structure of our approach is introduced in Figure 1. The following two subsections give more details about our proposals.

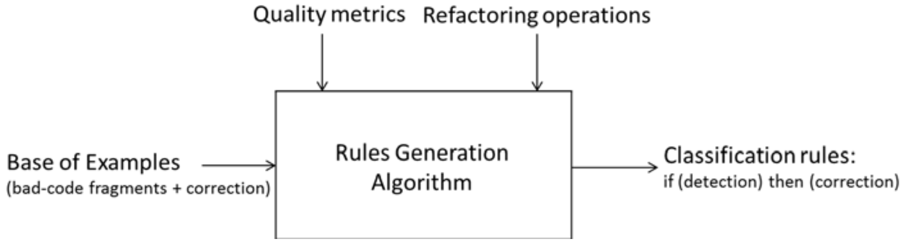


Fig. 1. Overview of the approach : general architecture

As described in Figure 1, knowledge from defect examples and their correction is used to generate our classification rules. In fact, our approach takes as inputs a base (i.e. a set) of defect examples (bad-designed code) with correction (refactorings to fix this bad designed code), 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 [15]) and an exhaustive list of refactoring operations [20]. This step generates a set of rules as output.

The rule generation process combines quality metrics (and their threshold values) and refactoring operations within rule expressions. Consequently, a solution to the defect detection and correction problem is a set of rules that best detect the defects of the base of examples and fix them. For example, the following rule states that a class having more than 10 attributes and 20 methods is fixed using different refactoring operations (Move method and Encapsulate field):

R1: IF NOA≥10 AND NOM≥20 Then MoveMethod ≥6 AND EncapsulateField<18

In this example of a rule, the number of attributes (NOA) and the number of methods (NOM) of a class correspond to two quality metrics that are used to detect a defect. A class will be detected as a defect whenever both thresholds of 10 attributes and 20 methods are exceeded. The second part of the rule fixes the corrected defect by applying more than 6 move method operations and less than 18 encapsulate field.

The rule generation process is executed periodically over large periods of time using the base of examples. The generated rules are used to detect and correct the defects of any system that is required 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 defect instances.

Our approach assigns a threshold value randomly to each metric and refactoring operation, 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. The rule generation process consists of finding the best combination between n metrics and k refactorings. In this context, the number NR of possible combinations that have to be explored is given by: $NR = ((n+k)!)^m$

This value quickly becomes huge. 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 [5]. This algorithm will be detailed in the next section.

3.2 Design Defects Classification Using Genetic Programming

This section describes how Genetic programming (GP) can be used to generate rules to detect and correct design defects.

3.2.1 Genetic Programming Overview

Genetic programming is a powerful heuristic search optimization method inspired by the Darwinian theory of evolution [1]. 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 symbols 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. Each individual of the population is evaluated by a fitness function that determines a quantitative measure of its ability to solve the target problem.

Exploration of the search space is achieved by 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 that it is allowed to transmit its features to new individuals by undergoing crossover and/or mutation operators. The crossover operator insures 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, 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 individual.

This process is repeated iteratively, until a termination criterion is met. This 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.

3.2.2 Genetic Programming Adaptation

A high level view of our Genetic Programming approach to the defect detection and correction problem is introduced by Figure 2.

Lines 1–3 construct an initial GP population, which is a set of individuals that stand for possible solutions representing classification rules. Lines 4–14 encode the main GP loop, which explores the search space and constructs new individuals by combining metrics and refactorings within rules. During each iteration, we evaluate the quality of each individual in the population, and save the individual having the best fitness (line 10). We generate a new population (p+1) of individuals (line 11) 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 classification rules (best solution found during all iterations). The following three subsections describe more precisely our adaption of GP to the defect classification problem.

```
Input: Set of quality metrics
Input: Set of refactoring operations
Input: Set of examples (bad-designed code fragments and their appropriate correction)
Output: Classification rules
1: I:= rules(R)
2: P:= set_of(I)
3: initial_population(P, Max_size)
4: repeat
5:      for all I ∈ P do
6:           (detected_defects, proposed_refactorings) := execute_rules(R)
7:           fitness(I) := compare(detected_defects,  defect_examples)   +
8:                   compare(proposed_refactorings, expected_refactorings)
9:      end for
10:     best_solution := best_fitness(I);
11:     P := generate_new_population(P)
12:     it:=it+1;
13: until it=max_it
14: return best_solution
```

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

3.2.2.1 Individual Representation. An individual is a set of IF – THEN rules. For example, Figure 3 shows the rule interpretation of an individual (NOA=Number of Attribute, and NOM=Number of Methods).

R1: IF (NOA ≥ 3 AND NOM ≤ 5) THEN (MoveField = 1 AND MoveClass = 1)
R2: IF (CBO≥ 1) THEN (MoveField = 1 AND ExtractClass = 1)

Fig. 3. Rule interpretation of an individual

Consequently, a detection rule has the following structure:

IF "Combination of metrics with their threshold values" THEN "Combination of Refactorings to apply"

The IF clause describes the conditions or situations under which a defect category 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 a defect. Consequently, THEN clauses highlight the correction to apply in order to fix the detected defect. 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 and correction of two defect types (categories). Consequently, as it is shown in Figure 4, we have two rules to detect and correct two categories of defects.

One of the most suitable computer representations of rules is based on the use of trees [17]. 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 or refactorings with their threshold values. The functions that can be used between these metrics correspond to logical operators, which are Union (OR) and Intersection (AND).

Consequently, the rule interpretation of the individual of Figure 3 has the following tree representation of Figure 4.

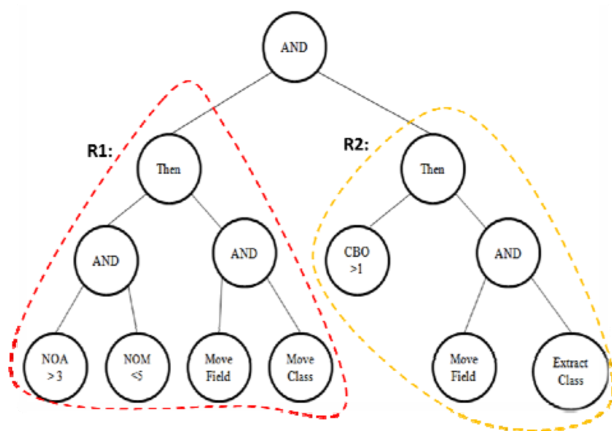


Fig. 4. A tree representation of an individual

3.2.2.2 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. These parameters can be specified either by the user or randomly. Thus, the individuals have different tree length (structure). Then, for each individual we randomly assign: (1) one metric or refactoring, with its threshold value, to each leaf node, and (2) a logic operator (AND, OR) to each function node.

The root (head) of the tree is unchanged. Since any metric combination is possible and correct semantically, we do need to define some semantic conditions to verify when generating an individual.

3.2.2.3 Operators

Selection

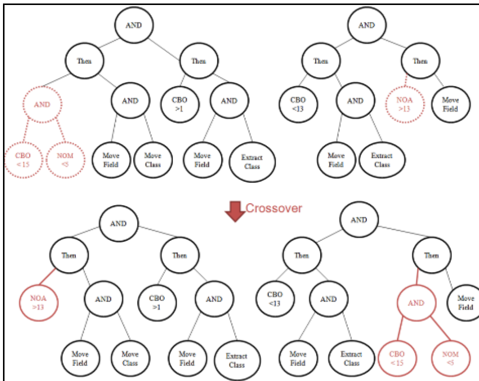
To select the individuals that will undergo the crossover and mutation operators, we used the stochastic universal sampling (SUS) [1], in which the probability of selection

of an individual is directly proportional to its relative fitness in the population. For each iteration, we use SUS to select (population_size/2) individuals from population p for the new populationp+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 crossover swaps the nodes and their relative sub trees from one parent to the other. 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 from another individual (solution) are combined to generate two new rules.



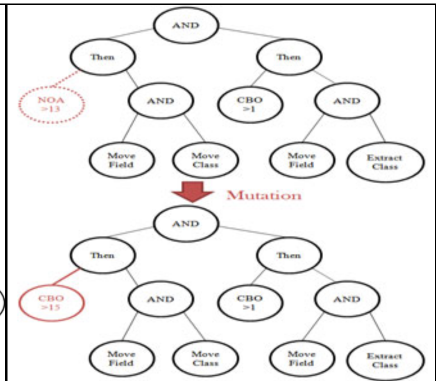


Fig. 5. Crossover operator

Fig. 6. Mutation operator

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

R1: IF (NOA ≥ 13) THEN (MoveField = 1 AND MoveClass = 1)

Of course, the crossover can be applied to the second part of the rules (refactorings).

Mutation

The mutation operator can be applied either to function or terminal nodes. This operator can modify one or more 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 or refactoring), it is replaced by another terminal. 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.

To illustrate the mutation process, consider again the example that corresponds to a candidate rule R1. Figure 6 illustrates the effect of a mutation that replaces node NOA by node CBO with a new threshold value. Thus, after applying the mutation operator the new rule R1 will be: *R1: IF (CBO > 15) THEN (MoveField = 1 AND MoveClass = 1).*

3.2.2.4 Decoding of an Individual. The quality of an individual is proportional to the quality of the different rules composing it. In fact, the execution of these rules, on the different projects extracted from the base of examples, detect and fix various classes. Then, the quality of a solution (set of rules) is determined with respect to 1) the number of detected defects in comparison with the expected ones in the base of examples, and 2) the number of proposed refactorings with those in the base of examples. In other words, the best set of rules is the one that detects and corrects the maximum number of defects.

3.2.2.5 Evaluation of an Individual. The decoding of an individual should be formalized as a mathematical function called “fitness function”. The fitness function quantifies the quality of the generated rules. The goal is to define an efficient and simple (in the sense not computationally expensive) fitness function in order to reduce the complexity.

As discussed previously, the fitness function aims to maximize the number of detected defects and proposed refactorings 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:

$$f_{norm} = \frac{\sum_{r=1}^{nbr} \frac{DQ+CQ}{2}}{nbr} \quad (3), \text{ where } DQ = \frac{\frac{\sum_{i=1}^{dd} a_i}{ed} + \frac{\sum_{i=1}^{dd} a_i}{dd}}{2} \text{ and } CQ = \frac{\frac{\sum_{j=1}^{pr} b_j}{er} + \frac{\sum_{j=1}^{pr} b_j}{pr}}{2}$$

nbr is the number of rules (defects categories) in the solution (individual), *DQ* represents detection quality, *CQ* is correction quality, *ed* is the number of expected defects in the base of examples, *dd* is the number of detected defects with defects, *pr* is the number of proposed refactoring and *er* is the number of expected refactoring in the base of examples.

a_i has value 1 if the ith detected class exists in the base of examples, and value 0 otherwise. b_j has value 1 if the jth proposed refactoring is used (exist in the base of examples) to correct the defect example.

4 Validation

To evaluate the feasibility of our approach, we conducted an experiment with different open-source projects. We start by presenting our research questions. Then, we describe and discuss the obtained results.

4.1 Research Questions

Our study asks three research questions, which we define here, explaining how our experiments are designed to address them. 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: To what extent our new defects classification is different from an existing classification work?
RQ3: To what extent can the proposed approach correct detected defects?

To answer RQ1, we asked eighteen graduate students to evaluate the precision and recall of our approach. To answer RQ2, we compared our results to those produced by an existing work [16]. Furthermore, we calculate a classification deviation score between the two algorithms. To answer RQ3, eighteen graduate students validated manually if the proposed corrections are useful to fix detected defects.

4.2 Setting

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 1 provides some relevant information about the programs. The base of examples contains more than 317 examples as presented in Table 1.

Table 1. Program statistics

Systems	Number of classes	KLOC	Number of Defects	Number of applied refactoring operations
GanttProject v1.10.2	245	31	41	34
Xerces-J v2.7.0	991	240	66	41
ArgoUML v0.19.8	1230	1160	89	82
Quick UML v2001	142	19	11	29
LOG4J v1.2.1	189	21	17	102
AZUREUS v2.3.0.6	1449	42	93	38

We used a 6-fold cross validation procedure. For each fold, one open source project is evaluated by using the remaining five systems as base of examples. For example, Xerces-J is analyzed using detection-correction rules generated from the base of examples containing ArgoUML, LOG4J, AZUREUS, Quick UML and Gantt.

The obtained detection results were compared to those of DECOR based on the recall (number of true positives over the number of maintainability defects) and the precision (ratio of true positives over the number detected). We also calculate a deviation score (number of common detected classes over the number of detected ones) between our new classification and DECOR classification (F.D.: Functional Decomposition, S.C.: Spaghetti Code, and S.A.K.: Swiss Army Knife).

To validate the correction step, eighteen graduate students verified manually the feasibility of the different proposed refactoring sequences for each system. We asked students to apply the proposed refactorings using ECLIPSE. Some semantic errors (programs behavior) were detected. When a semantic error is detected manually, we consider the refactoring 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.

Finally, we compared our genetic algorithm (GP) detection and correction results with a local search algorithm, called simulated annealing (SA) [8].

4.3 Results

In this sub-section we present the answer to each research question in turn, indicating how the results answer each.

Table 2 shows obtained detection precision and recall scores for each of the 6 folds. Furthermore, this table describes comparison results between our defects classification and DECOR. For Gantt, our average detection precision was 89%. DECOR, on the other hand, had a combined precision of 59% for the same expected bad-classes. The precision for Quick UML was about 62%, more than the value of 53% obtained with DECOR. For Xerces-J, our precision average was 95%, while DECOR had a precision of 68% for the same dataset. Finally, for ArgoUML, AZUREUS and LOG4J our precision was more than 75. However, the recall score for the different systems was less than that of DECOR. In fact, the rules defined in DECOR are large and this is explained by the lower score in terms of precision. Indeed, the hypothesis to have 100% of recall justifies low precision, sometimes, to detect defects. 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.

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

Table 2. Detection results

Systems	GP-Detection Precision	DECOR-Detection Precision	GP-Detection Recall	DECOR-Detection Recall	Number of Defect types (categories)
GanttProject	89% (33|37)	59% (41|69)	81% (33|41)	100%	6
Xerces-J	95% (47|49)	68% (66|97)	72% (47|66)	100%	7
ArgoUML	77% (74|96)	63% (89|143)	83% (74|89)	100%	6
Quick UML	62% (8|13)	53% (11|21)	73% (8|11)	100%	8
LOG4J	79% (15|19)	60% (17|28)	89% (15|17)	100%	6
AZUREUS	82%(87|106)	67% (93|138)	93% (87|93)	100%	7

For RQ2, we calculate, based on execution of our rules on ArgoUML, a similarity score between a well-known classification of defects [16] and our classification. This similarity score represents the number of common detected classes between a defect type *i* and our defect category *j*. In Table 3 we take the most similar category for each defect type. We can mention that only category 4 can be classified as a specific defect type which is the blob. Indeed, the explanation is that large classes has, in general, the same refactoring operations to be corrected (move methods, extract classes, etc.). Since our classification is based on correction criteria's and not symptoms, table 3 confirms our findings that there is a dissimilarity comparing to DECOR classification.

Table 3. Classification variation

Defect types	Most similar new category %
Blob	88 % (7|9, Category 4)
Functional Decomposition	36% (5|14, Category 1)
Spaghetti Code	54% (6|11, Category 6)
Swiss Army Knife	50% (8|16, Category 2)

Table 4. Correction results

Systems	GP-Correction Precision	GP-Correction Recall
GanttProject v1.10.2	84% (26|31)	76% (26|34)
Xerces-J v2.7.0	62% (29|47)	71% (29|41)
ArgoUML v0.19.8	75% (57|76)	67% (57|82)
Quick UML v2001	81% (21|26)	72% (21|29)
LOG4J v1.2.1	52% (63|123)	61% (63|102)
AZUREUS v2.3.0.6	74% (31|42)	81% (31|38)

We have also obtained very good results for the correction step. As showed in Table 4, the majority of proposed refactoring are feasible and improve the code quality. For example, for Gantt, 26 refactoring operations are feasible over the 31 proposed ones. After applying by hand the feasible refactoring operations for all systems, we evaluated manually that more than 75%, in average, of detected defects was fixed.

As described in Figure 7, we compared our genetic algorithm (GP) detection results with another local search algorithm, simulated annealing (SA). The detection and correction results for SA are also acceptable. Especially, with small systems the precision is better using SA than GP. In fact, GP is a global search that gives good results when the search space is large. For this reason, GP performs with large systems. In addition, the solution representation used in GP (tree) is suitable for rule generation. However, SA used a vector-based representation which is not suitable for rules. Furthermore, SA takes a lot of time, comparing to GP, to converge to an optimal solution (more than 1 hour).

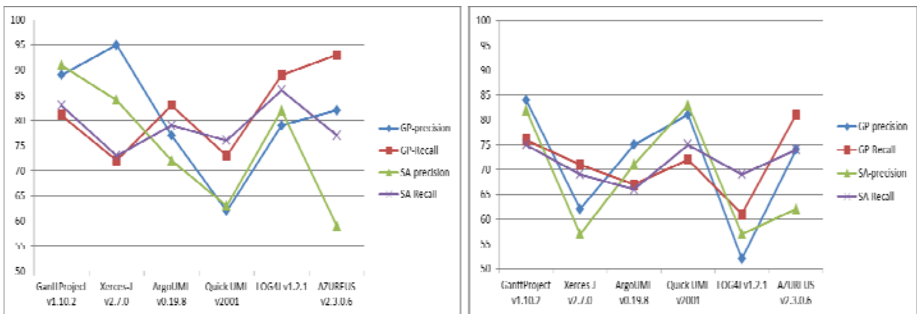


Fig. 7. (a) Detection results comparison: GP and SA, and (b) Correction results comparison: GP and SA

5 Discussions

An important consideration is the impact of the example base size on detection-correction quality. In general, our approach does not needs a large number of examples to obtain good detection results. The reliability of the proposed approach requires an example set of bad code and his correction (refactoring operations). 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 results for the studied systems. However, we agree that, sometimes, with specifying programming languages and contexts it is difficult to find bad code fragments with the correction version.

The performance of detection was superior to that of DECOR. In an industrial setting, we could expect a company to start with some few open source projects, and gradually migrate its set of bad code examples to include context-specific data. This might be essential if we consider that different languages and software infrastructures have different best/worst practices.

Another issue is the rule generation process. The detection and correction results might vary depending on the rules used, which 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 rule generation. We consequently believe that our technique is stable, since the precision and recall scores are approximately the same for different executions. In addition, our classification algorithm generates, approximately, the same defects categories as showed in Table .

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 [4].

Finally, since we viewed the maintainability defects detection problem 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 (i7 CPU running at 3 GHz with 4GB of RAM). The execution time for rule generation with a number of iterations (stopping criteria) fixed to 10000 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, refactorings and the size of the base of examples.

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. Nevertheless, few works focused on combining detection and correction steps to classify defects based on correction possibilties.

Marinescu [18] 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 et al. [6] 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 et al. [2] 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. [16], 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.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 post condition), or graph transformation. The use of invariants has been proposed to detect parts of program that require refactoring by [22]. Opdyke [20] 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. [19] 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 [8], 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 [7] 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 another work [11], we generate detection rules defined as combinations of metrics/thresholds that better conform to known instances of design defects (defect examples). Then, the correction solutions, a combination of refactoring operations, should minimize, as much as possible, the number of defects detected using the detection rules. Our previous work treats the detection and correction as two different steps. In this paper, we generate also new form of detection-correction rules that are completely different from [11].

7 Conclusion

In this paper we introduced a new classification of defects based on correction criteria's. Existing work classify different types of common maintainability defects based on 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 and correction. Instead, we use examples of maintainability defects and their corrections to generate detection-correction classification rules. Our results show that our classification is able to achieve correction precision scores on different open-source projects.

As part of future work, we plan to compare our results with some existing approaches. In addition, we will modify our algorithm to generate sub-rules in order to specify code elements name (fully-automate the correction step). Furthermore, we need to extend our base of examples in order to improve precision scores.

References

1. Goldberg, D.E.: Genetic Algorithms in Search, Optimization and Machine Learning. Addison-Wesley Longman Publishing Co., Inc., Boston (1989)
2. Alikacem, H., Sahraoui, H.: Détection d'anomalies utilisant un langage de description de règle de qualité, in actes du 12e colloque LMO (2006)
3. 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 2009, pp. 265–268 (2009)
4. Bratko, I., Muggleton, S.: Applications of inductive logic programming. Commun. ACM 38(11), 65–70 (1995)
5. Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press, Cambridge (1992)
6. Erni, K., Lewerentz, C.: Applying design metrics to object-oriented frameworks. In: Proc. IEEE Symp. Software Metrics. IEEE Computer Society Press (1996)
7. Kessentini, M., Vaucher, S., Sahraoui, H.: Deviance from perfection is a better criterion than closeness to evil when identifying risky code. In: Proc. of ASE 2010. IEEE (2010)
8. Kirkpatrick, S., Gelatt Jr., C.D., Vecchi, M.P.: Optimization by simulated annealing. Sciences 220(4598), 671–680 (1983)
9. Fowler, M.: Refactoring – Improving the Design of Existing Code, 1st edn. Addison-Wesley (June 1999)
10. Harman, M., Clark, J.A.: Metrics are fitness functions too. In: IEEE METRICS, pp. 58–69 (2004)
11. Kessentini, M., Kessentini, W., Sahraoui, H., Boukadoum, M., Ouni, A.: Design Defects Detection and Correction by Example. In: Proc. ICPC 2011, pp. 81–90. IEEE (2011)
12. Mantyla, M., Vanhanen, J., Lassenius, C.: A taxonomy and an initial empirical study of bad smells in code. In: Proc. of ICSM 2003. IEEE Computer Society (2003)
13. O'Keeffe, M., Cinnéide, M.: Search-based refactoring: an empirical study. Journal of Software Maintenance 20(5), 345–364 (2008)
14. Mens, T., Tourwé, T.: A Survey of Software Refactoring. IEEE Trans. Softw. 30(2), 126–139 (2004)
15. Fenton, N., Pfleeger, S.L.: Software Metrics: A Rigorous and Practical Approach, 2nd edn. International Thomson Computer Press, London (1997)
16. Moha, N., Guéhéneuc, Y.-G., Duchien, L., Meu, A.-F.L.: DECOR: A method for the specification and detection of code and design smells. Transactions on Software Engineering (TSE), 16 pages (2009)

17. Davis, R., Buchanan, B., Shortcliffe, E.H.: Production Rules as a Representation for a Knowledge-base Consultation Program. Artificial Intelligence 8, 15–45 (1977)
18. Marinescu, R.: Detection strategies: Metrics-based rules for detecting design flaws. In: Proc. of ICM 2004, pp. 350–359 (2004)
19. Heckel, R.: Algebraic graph transformations with application conditions. M.S. thesis, TU Berlin (1995)
20. Opdyke, W.F.: Refactoring: A Program Restructuring Aid in Designing Object-Oriented Application Frameworks, Ph.D. thesis, University of Illinois at Urbana-Champaign (1992)
21. Brown, W.J., Malveau, R.C., Brown, W.H., McCormick III, H.W., Mowbray, T.J.: Anti Patterns: Refactoring Software, Architectures, and Projects in Crisis, 1st edn. John Wiley and Sons (March 1998)
22. 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 Computer Society (2001)