# Example-based Design Defects Detection and Correction

Marouane Kessentini[1,2] , Wael Kessentini[1], and Abdelkarim Erradi[2]

[1] DIRO, Université de Montréal
Montréal, Canada
{kessentm}@iro.umontreal.ca
[2] Qatar University
Doha, Qatar
erradi@qu.edu.qa

**Abstract**. Software design 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 (development and maintenance stages). 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 design 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. Using a Genetic Algorithm, correction solutions are found by combining refactoring operations in such a way to minimize the number of detected defects. We evaluate our approach by finding and fixing potential defects in six open-source systems. For all these systems, we succeed in detecting, in average, more than 76% of known defects, a better result when compared to a state-of-the-art approach, where the detection rules are manually or semi-automatically specified. The proposed corrections fix, in average, more than 74% of detected defects.

**Keywords:** *Design defects; software maintenance; search-based software engineering; by example.*

## 1 Introduction

Many studies reported that software maintenance, traditionally defined as any modification made on a software code during its development process or after its delivery, consumes up to 90% of the total cost of a typical software project [1]. Adding new functionalities, detecting design defects, correcting them, and modifying the code to improve its quality are major activities of those maintenance tasks [3]. Although design defects are sometimes unavoidable, they

**ACKNOWLEDGEMENTS**
This research work was funded by Qatar National Research Fund (QNRF) under the National Priorities Research Program (NPRP) Grant No.: 09-1205-2-470.

should be in general prevented by the development teams and removed from their code base as early as possible.

There has been much research effort focusing on the study of design defects, also called antipatterns [2], smells [3], or anomalies [1] in the literature. Such defects include for example blobs, spaghetti code, functional decomposition, etc.

Detecting and fixing design defects is, to some extent, a difficult, time-consuming, and manual process [6]. 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. For example, one Mozilla developer claimed that "*everyday, almost 300 bugs and defects appear, ..., far too much for only the Mozilla programmers to handle*".

To insure detection of design defects, several automated detection techniques have been proposed [5], [6], [7], [4],[8].The vast majority of these techniques relies on declarative rule specification [5], [6], [7], [4]. 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 each defect, rules that are expressed in terms of metric combinations need substantial calibration efforts to find the right threshold value for each metric, threshold above which a defect is said to be detected.

Another work [5] 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 consensual symptom-based definition of design defects [1]. When consensus exists, the same symptom could be associated to many defect types, which may compromise the precise identification of defect types. These difficulties explain a large portion of the high false-positive rates reported in existing research.

After detecting design defects, the next step is to fix them. Some work proposes "standard" refactoring solutions that can be applied by hand for each kind of defect [2]. However, it is difficult to prove or ensure the generality of these solutions to any kind of defects or software codes. The majority of other works start from the hypothesis that useful refactorings are those which improve values of some metrics that describe “good” quality of software code [20]. Some questions arise from the study of these approaches: how to determine the useful metrics for a given system? What would be the best (or at least a good) way to combine multiple metrics? Does improving the metric values necessarily mean that specific defects are corrected? Existing approaches only bring limited answers to these issues.

Besides, one can notice the availability of defect repositories in many companies, where defects in projects under development are manually identified, corrected and documented. Despite its availability, this valuable knowledge is not used to mine regularities about defect manifestations. These regularities could be exploited both to detect defects, and to correct them.

In this paper, we propose a two step approach to overcome some of the above mentioned limitations. Our approach is based on the use of defect examples generally available in defect repositories of software developing companies. 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 design defects. This is achieved using Genetic Programming (GP). Then, we generate correction solutions based on combinations of refactoring operations that minimize the number of detected defects in the code to be corrected. This is achieved using a Genetic Algorithm (GA). Such proposal is very beneficial because:

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 rules manually;
3) it does not require to specify the metrics to use or their related threshold values;

The remainder of this paper develops our proposals and details how they are achieved. Therefore, the paper is structured as follows. Section 2 is dedicated to the problem statement. In Section 3, we give an overview of our proposal. Then, Section 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 Section 6. We conclude and suggest future research directions in Section 7.

## 2 Research Problem

To better understand our contribution, it is important to clearly 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 design defects, also called design anomalies, refer to design situations that adversely affect the development of a software. As stated by Fenton and Pfleeger [3], design 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 [2] and suggesting improvement solutions. The two types of defects that are commonly mentioned in the literature are code smells and anti-patterns. In [2], 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. [1] define another category of design defects that are documented in the literature, and named anti-patterns. In our approach, we focus on the three following concepts:

- *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 in finding code fragments that violate properties on internal attributes 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 [12]. The most widely used metrics are the ones defined by Chidamber and Kemerer [3]. These include:

- Depth of inheritance tree, DIT,
- Weighted methods per class, WMC,
- Coupling between objects, CBO.

Variations of these metrics, adaptations of procedural ones, as well as new metrics were also proposed [3] including:

- Number of lines of code in a class, LOCCLASS,
- Number of lines of code in a method, LOCMETHOD,
- Number of attributes in a class, NAD,
- Number of methods, NMD,
- Lack of cohesion in methods, LCOM5,
- Number of accessors, NACC,
- Number of private fields, NPRIVFIELD.

The detected defects can be fixed by applying some refactoring operations. William Opdyke [27] defines refactoring as the process of improving a code after it has been written by changing the internal structure of the code without changing the 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 [28]. Some examples of refactoring operations include:

- *Push down field* moves a field from some class to those subclasses that require it.
- *Add parameter* adds new parameter to a method
- *Push down method* moves a method from some class to those subclasses that require it.
- *Move method* moves a method from class A to class B.

A complete list of refactoring operations can be found in [29].

## 2.2 Detection Issues

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.

**How to decide if a defect candidate is an actual defect?**
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.

**Are long lists of defect candidates really useful?**
Detecting dozens of defect occurrences in a system is not always helpful. 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.

**What are the boundaries?**
There is a general agreement on extreme manifestations of design defects. For example, consider an OO program with a hundred classes from which one class implements all the behavior and all the other classes are only classes with attributes and accessors. There is no doubt that we are in presence of a Blob.

Unfortunately, in real life systems, we can find many large classes, each one using some data classes and some regular classes. Deciding which classes are Blob candidates depends heavily on the interpretation of each analyst.

**How to define detection thresholds when dealing with quantitative information?**

For example, the Blob detection involves information such as class size. Although we can measure the size of a class, an appropriate threshold value is not trivial to define. A class considered large in a given program/community of users could be considered average in another.

**How to deal with the context?**

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.

**Does improving code quality really mean that detected defects are fixed?**

In the majority of situations, code quality can be improved without fixing design defects. We need to identify if the code modification corrects or not some specific defects. 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.

**How to generalize correction solutions?**

The correction solutions should not be specific to only some defect types. In fact, specifying manually 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 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 [28] and each defect type requires specific rules.

## 3 Approach Overview

To address or 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 automatically derive detection rules.
2) Correction of defects: we find the refactoring solution that minimizes the number of detected defects.

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

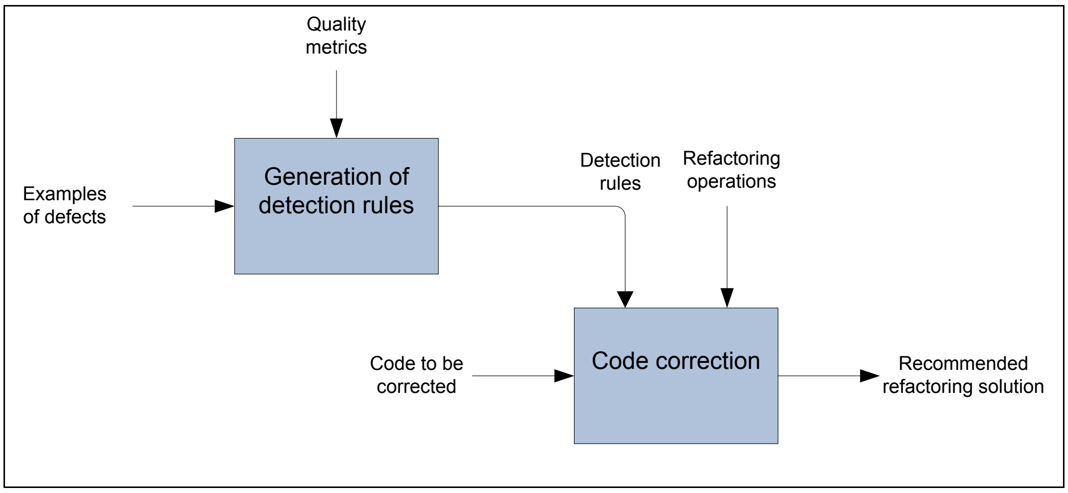

**Fig 1.** Overview of the approach general architecture

### 3.1 Detection of Design Defects

In this step, 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 [12]). This step generates as output a set of rules. The rule generation process combines quality metrics (and their threshold values) within rule expressions. 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:

*R1: IF NAD(c)≥10 AND NMD(c)≥20 Then Blob(c)*

In this example of a rule, the number of attributes (NAD) and the number of methods (NMD) of a class correspond to two quality metrics that are used to detect a blob 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 in particular companies. These rules will be more precise and more context faithful, yet almost without loss of genericity, 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 is required to be evaluated (in the sense of defect detection and correction). The rules generation step needs to be re-executed only if the base of examples is updated with new defect instances.

## 3.1 Correction of Design Defects

The correction step takes as controlling parameters, the generated detection rules, and a set of refactoring operations that were defined and discussed in the scientific literature [2]. This step takes as input a software code to be corrected. As output, this step recommends a refactoring solution, which suggests a set of refactoring operations that should be applied in order to correct the input code [29]. 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.

We use logic predicates [34] to represent refactoring operations. For example, the following predicate indicates that the method "division" is moved from class "department" to class "university":

*MoveMethod(division, departement, university)*

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

## 3.2 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. The rules 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:

$$NR = (n!)^m \qquad (1)$$

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 [25]. This algorithm will be detailed in section 4.

As far as the correction step is concerned, in addition to the number of possible refactoring combinations, the order of applying them is important. If k is the number of available refactoring operations, then the number NS of possible refactoring solutions is given by:

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

Due to the large number of possible refactoring solutions, another heuristic optimization method, which relies on a Genetic Algorithm [26], is used to generate refactoring solutions. This algorithm is described in section 4.

## 4 Refactoring by Example

This section describes how Genetic programming (GP) can be used to generate rules to detect design defects, and how a Genetic Algorithm (GA) can be derived to find refactoring solutions. To apply GP and GA to a specific problem, the following elements have to be defined:

- Representation of the individuals,
- Creation of a population of individuals,
- Evaluation of individuals (using a fitness function) to determine a quantitative measure of 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 GA.

## 4.1 Defects Detection Using Genetic Programming

Genetic programming is a powerful heuristic search optimization method inspired by the Darwinian theory of evolution [26]. 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. The mutation operator introduces diversity into the population and allows escaping local optima found during the search.
Once selection, mutation and crossover have been applied according to given probabilities, individuals of the newly created generation are evaluated using the fitness function. This process is repeated iteratively, until a stopping 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.

### *4.1.1. GP Algorithm overview*

A high level view of our Genetic Programming approach to the defect detection problem is introduced by Figure 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, which corresponds to the set of detection rules that best detect the defects in the base of examples.

```
Input: Set of quality metrics
Input: Set of defect examples
Output: Detection rules
1: I:= rules(R, Defect_Type)
2: P:= set_of(I)
3: initial_population(P, Max_size)
4: repeat
5:      for all I ∈ P do
6:            detected_defects := execute_rules(R)
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
```

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

Lines 1–3 construct an initial GP population, which is a set of individuals that stand for possible solutions representing detection rules. Lines 4–13 encode the main GP loop, which explores the search space and constructs new individuals by combining metrics within rules. During each 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.

#### 4.1.2.1 Individual Representation

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

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

**Fig 3.** 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 figure 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 [35]. 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).

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

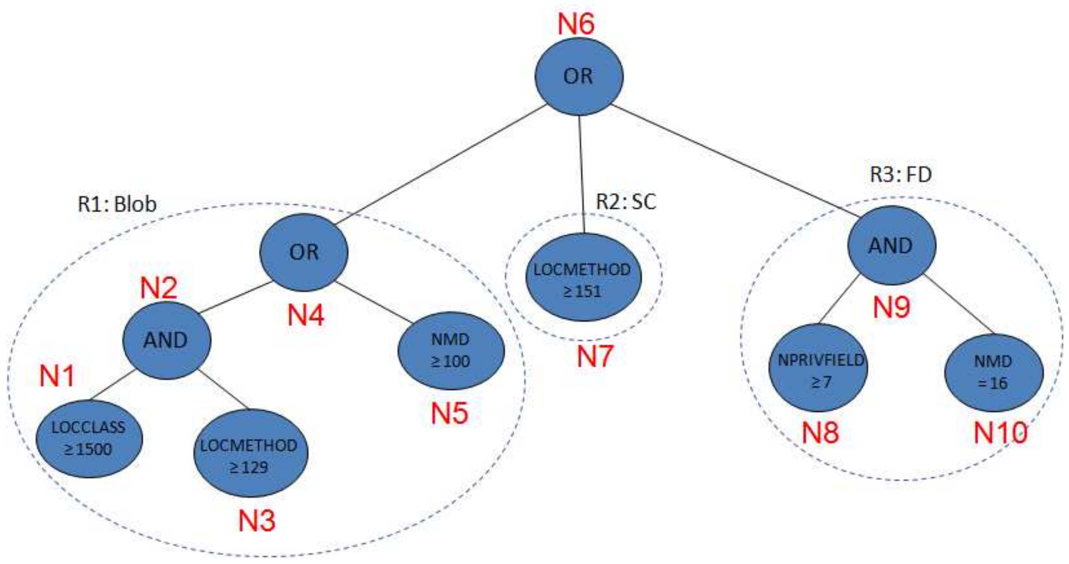

**Fig 4.** A tree representation of an individual

For instance, the first rule R1 is represented as a sub-tree of nodes starting at the branch (N1 – N5) of the individual tree representation of Figure 5. 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.

### *4.1.2.2 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:

- one metric, with its threshold value, to each leaf node
- 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 conditions to verify when generating an individual.

*4.1.2.3 Genetic Operators*

### Selection

To select the individuals that will undergo the crossover and mutation operators, we used the stochastic universal sampling (SUS) **[25]**, 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. 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.

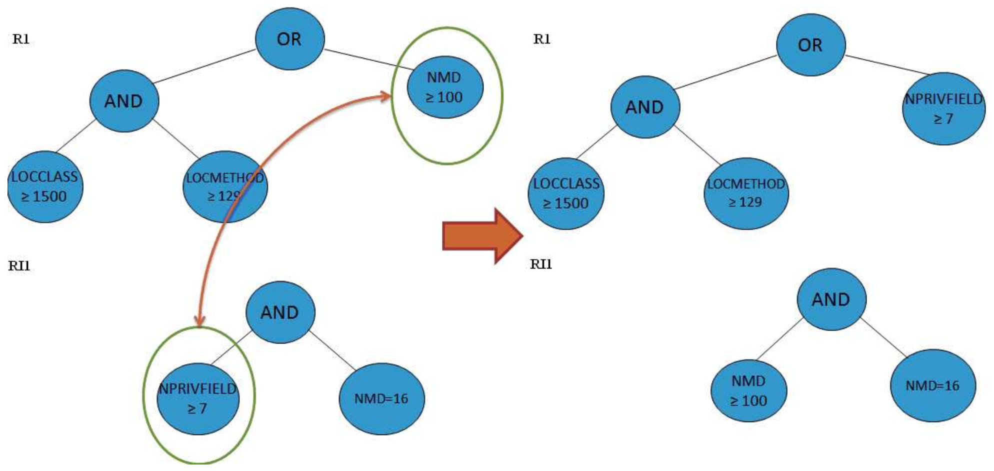

**Fig 5.** Crossover operator

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

R1: IF (LOCCLASS(c) ≥ 1500 AND LOCMETHOD(m,c) ≥ 129)) OR (NPRIVFIELD(c) ≥ 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.

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 node NMD, leading to the automatic deletion of node OR (no left sub tree), and that replaces 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) ≥ 1500 AND NPRIVFIELD(c) ≥ 14)) THEN blob(c)

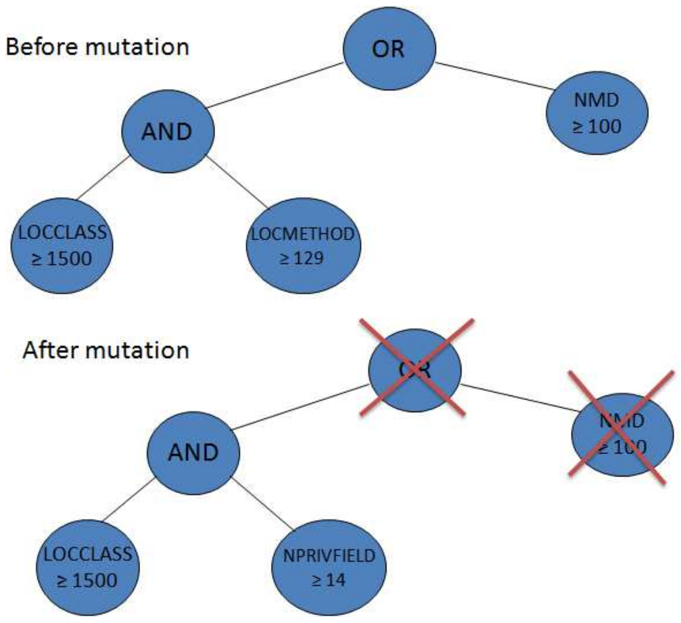

**Fig 6.** Mutation operator

#### *4.1.2.4 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, detect various classes as defect. 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.

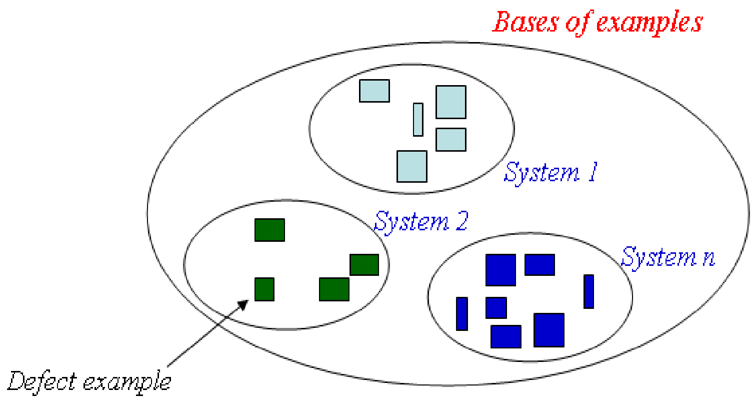

**Fig 7.** Base of examples

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

#### *4.1.2.5 Evaluation of an individual*

The encoding 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 computational complexity.

As discussed in Section 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:

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

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 $i^{th}$ 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.

TABLE I. DEFECTS EXAMPLE.

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

The classes detected after executing the solution generating the rules R1, R2 and R3 of Figure 5 are described in Table 2.

TABLE II. DETECTED CLASSES

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

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 Design Defects Correction Using Genetic Algorithm

Genetic programming, described in the previous section, is a branch of genetic algorithms [26]. The main difference between genetic programming and genetic algorithms is the representation of the solution. Genetic programming creates computer programs as the solution (tree representation). Genetic algorithms create a string of numbers that represent the solution. Since we have detailed the description of GP in the previous section, we describe directly in the next section the adaption of GA to our problem.

### *4.2.1 Genetic Algorithm Adaptation*

#### *4.2.1.1 Individual Representation*

One key issue when applying a search-based technique is to find a suitable mapping between the problem to solve and the techniques to use, i.e., in our case, correcting detected defect classes. We view the set of potential solutions as points in an n-dimensional space, where each dimension corresponds to one refactoring operation. Figure 8 shows an example where the ith individual (solutions) represents a combination of refactoring operations to apply. The sequence of applying the refactorings corresponds to their order in the table (dimension number). The execution of these refactorings are conformed to some pre and post conditions (to avoid conflicts).

| | | |
|---|---|---|
| MoveMethod(division, Departement, University | AddParameter(x, int, average, Student) | *PushDownMethod(average, Student, Person)* |

**Fig 8.** Individual representation

#### *4.2.1.2 Selection and Genetic Operators*

For each crossover, two detectors are selected by applying twice the wheel selection [26]. Even though individuals are selected, the crossover happens only with a certain probability.

The crossover operator allows to create two offspring o1 and o2 from the two selected parents p1 and p2. It is defined as follows: A random position k, is selected. The first k refactorings of p1 become the first k elements of o1. Similarly, the first k refactorings of p2 become the first k refactorings of o2.

The mutation operator operator consists of randomly changing a dimension (refactoring).

#### *4.2.1.3 Fitness Function*

The fitness function quantifies the quality of the proposed refactorings. In fact, the fitness function checks to minimize the number of detected defects using the detection rules. In this context, we define the fitness function of a solution as

$$f = \min(n) \qquad (4)$$

Where n is the number of detected classes.

# 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 design 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 design defects?

RQ2: What types of defects does it locate correctly?

RQ3: To what extent can the proposed approach correct detected defects?

To answer RQ1, we used an existing corpus of known design defects [5] to evaluate the precision and recall of our approach. We compared our results to those produced by an existing rule-based strategy [5]. 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.

### 5.2 System 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 1 provides some relevant information about the programs.

TABLE III. PROGRAM STATISTICS.

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

We chose the Xerces-J, ArgoUML, LOG4J, AZUREUS, Quick UML and Gantt libraries because they are medium-sized open-source projects and were analysed in related work. The version of Gantt studied was known to be of poor quality, which has led to a new 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 [5], Moha et al. asked three groups of students to analyse 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 describe in section 2, blobs are classes that do or know too much; spaghetti Code (SC) is code that does not use appropriate structuring mechanisms; finally, functional decomposition (FD) is 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 base of examples. For example, Xerces-J is analyzed using detection rules generated from some defects examples from ArgoUML, LOG4J, AZUREUS, Quick UML and Gantt.

The obtained results were compared to those of DECOR. Since [5] reported the number of antipatterns detected, the number of true positives, the recall (number of true positives over the number of design 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 calculate a correctness precision score (ratio of possible refactoring operations over the number of proposed refactoring) as performance indicator of our algorithm.

The complete lists of metrics, used to generate rules, and applied refactorings can be found in [16].

### 5.3 Results

| **Class** | F.D. | Blob | Spag |
|---|---|---|---|
| AbstractDOMParser | | | x |
| BaseMarkupSerializer | | X | |
| CoreDocumentImpl | | X | x |
| DFAContentModel | | | x |
| DocumentBuilderImpl | | | |
| DOMNormalizer | | X | |
| DOMSerializerImpl | | | x |
| DTDConfiguration | | X | |
| DTDGrammar | | X | |
| HTMLDOMImplementation | | | |
| LineSeparator | | | |
| NonValidatingConfiguration | | X | |
| RegexParser | x | | |
| SAXParser | x | | |
| SchemaDOM | x | | |
| ShortHandPointer | | | |
| Token | | | x |
| ValidatedInfo | | | |
| XIncludeHandler | | X | |

| | | | |
|---|---|---|---|
| XML11Configuration | | X | x |
| XML11DTDConfiguration | | X | |
| XML11NonValidatingConfiguration | | X | |
| XMLDTDValidator | | X | |
| XMLElementDecl | | | |
| XMLEntityManager | | X | |
| XMLNSDTDValidator | x | | |
| XMLParser | x | | |
| XMLSchemaValidator | | X | |
| XMLSerializer | | | x |

**Fig 9.** Xerces-J Results (Top-30 classes)

| **Class** | F.D. | Blob | Spag |
|---|---|---|---|
| GanttGraphicArea | | x | X |
| GraphicPrimitiveContainer | | | X |
| GanttOptions | | x | X |
| ResourceLoadGraphicArea | | | X |
| GanttXFIGSaver | | | X |
| GanttDialogPerson | | | X |
| NewProjectWizard | X | x | |
| GanttTree | | x | |
| GanttProject | | x | |
| TimeUnitGraph | X | | |
| GregorianTimeUnitStack | X | | |
| TipsDialog | | | X |
| RecalculateTaskCompletionPercentageAlgorithm | X | | |
| TaskHierarchyManagerImpl | X | | |

**Fig 10.** Gantt Results

TABLE IV. DETECTION RESULTS.

| System | Precision | Recall |
|---|---|---|
| GanttProject | Blob : 100%<br>SC : 93%<br>FD : 91% | 100%<br>97%<br>94% |
| Xerces-J | Blob : 97%<br>SC: 90%<br>FD: 88% | 100%<br>88%<br>86% |
| ArgoUML | Blob : 93%<br>SC: 88%<br>FD: 82% | 100%<br>91%<br>89% |
| QuickUML | Blob : 94%<br>SC: 84%<br>FD: 81% | 98%<br>93%<br>88% |
| AZUREUS | Blob : 82%<br>SC: 71%<br>FD: 68% | 94%<br>81%<br>86% |
| LOG4J | Blob : 87%<br>SC: 84%<br>FD: 66% | 90%<br>84%<br>74% |

TABLE V. CORRECTION RESULTS.

| System | Number of proposed refactorings | Precision |
|---|---|---|

|  |  |  |  |
|---|---|---|---|
| GanttProject | 39 | 84% (33 | 39) |
| Xerces-J | 47 | 78% (38 | 47) |
| ArgoUML | 73 | 81% (59 | 73) |
| QuickUML | 32 | 85% (27 | 32) |
| AZUREUS | 96 | 72% (69 | 96) |
| LOG4J | 41 | 77% (32 | 41) |

Figures 9-10 and Tables 4-5 summarize our findings. Figure 9 shows the top-30 detected classes including only 5 false-positive ones (highlighted with a different color). 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 42% 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, for ArgoUML, AZUREUS and LOG4J our precision was more than 75. However, the recall score for the different systems 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. In the context of this experiment, we can conclude that our technique was able to identify design anomalies more accurately than DECOR (answer to research question RQ1 above).

We have also obtained very good results for the correction step. As showed in Table 5, the majority of proposed refactoring are feasible and improve the code quality. For example, for Gantt, 33 refactoring operations are feasible over the 39 proposed ones. After applying by hand the feasible refactoring operations for all systems, we evaluated manually that more than 75% or detected defects was fixed. The majority of non-fixed defects are related to blob type. In fact, this type of defect needs a large number of refactoring operations and it is very difficult to correct it.

The complete list of detected classes and proposed refactorings by our approach for the different systems can be found in [16].

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 blobs and nine SCs in the system, and eight of the seventeen FDs, four more than DECOR. In Quick UML, we found three out five FDS; however DECOR detected three out of ten FDs.

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 leads 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.

An important consideration is the impact of the example base size on detection quality. Drawn for AZUREUS, the results of Figures 11 shows that our approach also proposes good detection reults 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 do not needs a large number of examples to obtain good detection results.

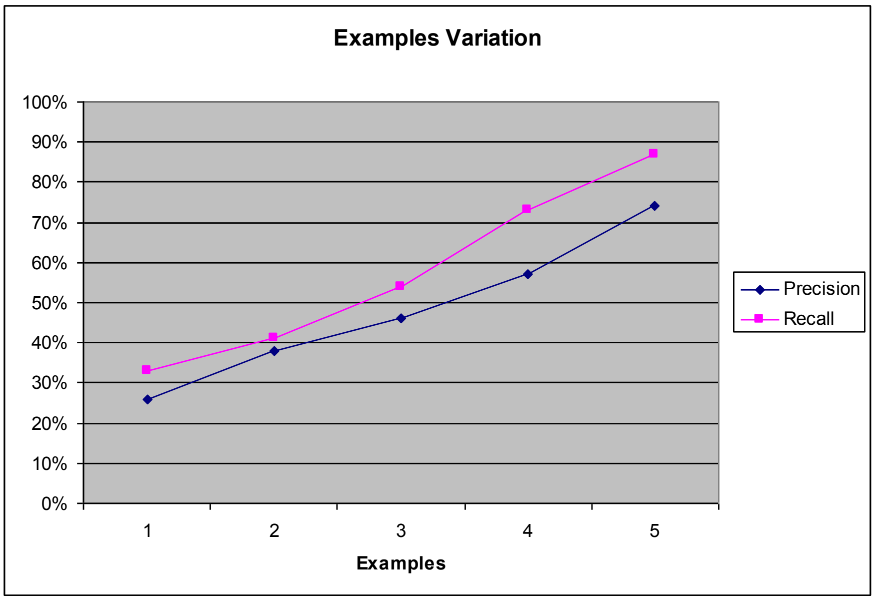

**Fig 11.** Examples-size variation

## 5.4 Discussion

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.

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 rules generation process. The detection 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 rules generation. We consequently believe that our technique is stable, since the precision and recall scores are approximately the same for 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 [17].

Finally, since we viewed the design 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 (Pentium CPU running at 2 GHz with 3GB of RAM). The execution time for rules generation with a number of iterations (stopping criteria) fixed to 350 was less than four minutes (3min27s) 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. It should be noted that more important execution times may be obtained than when using DECOR. In any case, our approach is meant to apply mainly in situations where manual rule-based solutions are not easily available.

## 6 Related Work

Several studies 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. The related work can be classified into three broad categories: rules-based detection-correction, detection and correction combination, and visual-based detection.

In the first category, Marinescu [7] 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. [18] 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. [19] 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. [5], 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. [4] 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 post condition), or graph transformation. The use of invariants has been proposed to detect parts of program that require refactoring by [30]. Opdyke [27] 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. [31] 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 [8] 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 [20], 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 [21]. 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. [33] present a pattern-based framework for developing tool support to detect software anomalies by representing potentials defects with different colors. Later, Dhambri et al. [32] 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 visualisation techniques is limited to the detection step.

## 7 Conclusion

In this article, we presented a novel approach to the problem of detecting and fixing design defects. Typically, researchers and practitioners try to characterize different types of common design defects and present symptoms to search for in order to locate the design 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 design 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. The best solution has the minimum fitness value. Due to the large number of refactoring combination, a genetic algorithm is used. Our study shows that our technique outperforms DECOR [5], 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 take into consideration more programming contexts.

## References

[1] W. J. Brown, R. C. Malveau, W. H. Brown, H. W. McCormick III, and T. J. Mowbray: Anti Patterns: Refactoring Software, Architectures, and Projects in Crisis, 1st ed. John Wiley and Sons, March 1998.

[2] M. Fowler: Refactoring – Improving the Design of Existing Code, 1st ed. Addison-Wesley, June 1999.

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

[4] F. Khomh, S. Vaucher, Y.-G. Guéhéneuc, and H. Sahraoui: A Bayesian Approach for the Detection of Code and Design Smells, n Proc. of the ICQS'09.

[5] N. Moha, Y.-G. Guéhéneuc, L. Duchien, and A.-F. L. Meur: DECOR: A method for the specification and detection of code and design smells, Transactions on Software Engineering (TSE), 2009, 16 pages.

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

[7] R. Marinescu: Detection strategies: Metrics-based rules for detecting design flaws, in Proc. of ICM'04, pp. 350–359.

[8] Kessentini, M., Vaucher, S., and 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.

[9] http://ganttproject.biz/index.php

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

[11] A. J. Riel: Object-Oriented Design Heuristics. Addison-Wesley, 1996.

[12] Gaffney, J. E.: Metrics in software quality assurance, in *Proc. of the ACM '81 Conference*, ACM, 126-130, 1981.

[13] M. Mantyla, J. Vanhanen, and C. Lassenius: A taxonomy and an initial empirical study of bad smells in code, in Proc. of ICSM'03, IEEE Computer Society, 2003..

[14] W. C. Wake: Refactoring Workbook. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003.

[15] http://www.ptidej.net/research/decor/index_html

[16] http://www.marouane-kessentini/icpc11.zip

[17] Raedt, D.:Advances in Inductive Logic Programming, 1st. IOS Press, 1996.

[18] K. Erni and C. Lewerentz: Applying design metrics to object-oriented frameworks, in Proc. IEEE Symp. Software Metrics, IEEE Computer Society Press, 1996.

[19] H. Alikacem and H. Sahraoui: Détection d'anomalies utilisant un langage de description de règle de qualité, in actes du 12e colloque LMO, 2006.

[20] M. O'Keeffe and M. . Cinnéide: Search-based refactoring: an empirical study, Journal of Software Maintenance, vol. 20, no. 5, pp. 345–364, 2008.

[21] M. Harman and J. A. Clark: Metrics are fitness functions too. in IEEE METRICS. IEEE Computer Society, 2004, pp. 58–69.

[22] M. Kessentini, H.Sahraoui and M.Boukadoum Model Transformation as an Optimization Problem. In Proc.MODELS 2008, pp. 159-173 Vol. 5301 of LNCS. Springer, (2008)

[23] D.S. Kirkpatrick, Jr. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671680, (1983)

[24] R.C. Eberhart and Y. Shi : Particle swarm optimization: developments, applications and resources. In: Proc. IEEE Congress on Evolutionary Computation (CEC 2001), pp. 81–86 (2001)

[25] John R. Koza. 1992. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA.

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

[27] W. F. Opdyke, Refactoring: A Program Restructuring Aid in Designing Object-Oriented Application Frameworks, Ph.D. thesis, University of Illinois at Urbana-Champaign, 1992.

[28] Mens, T. and Tourwé, T. 2004. A Survey of Software Refactoring. *IEEE Trans. Softw. Eng.* 30, 2 (Feb. 2004), 126-139.

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

[30] Y. Kataoka, M. D. Ernst, W. G. Griswold, and D. Notkin, "Automated support for program refactoring using invariants," in Proc. Int'l Conf. Software Maintenance. 2001, pp. 736–743, IEEE Computer Society.

[31] Reiko Heckel, "Algebraic graph transformations with application conditions," M.S. thesis, TU Berlin, 1995

[32] K. Dhambri, H. A. Sahraoui, and P. Poulin, "Visual detection of design anomalies." in CSMR. IEEE, 2008, pp. 279–283.

[33] S. C. Kothari, L. Bishop, J. Sauceda, and G. Daugherty, "A pattern-based framework for software anomaly detection," Software Quality Journal, vol. 12, no. 2, pp. 99–120, June 2004.

[34] Ivan Bratko and Stephen Muggleton. 1995. Applications of inductive logic programming. *Commun. ACM* 38, 11 (November 1995), 65-70.

[35] R. Davis, B. Buchanan, and E.H. Shortcliffe, Production Rules as a Representation for a Knowledge-base Consultation Program, Artificial Intelligence 8 (1977), 15-45.