Design Defects Detection and Correction by Example

Marouane Kessentini[1], Wael Kessentini[1], Houari Sahraoui[1], Mounir Boukadoum[2] and Ali Ouni[1]
[1]DIRO, Université de Montréal, Canada,
{kessentm, sahraouh}@iro.umontreal.ca,
[2] DI,Université du Québec à Montréal, Canada,
mounir.boukadoum@uqam.ca

Abstract— **Detecting and fixing defects make programs easier to understand by developers. We propose an automated approach for the detection and correction of various types of design defects in source code. Our approach allows to automatically find detection rules, thus relieving the designer from doing so manually. Rules are defined as combinations of metrics/thresholds that better conform to known instances of design defects (defect examples). The correction solutions, a combination of refactoring operations, should minimize, as much as possible, the number of defects detected using the detection rules. In our setting, we use genetic programming for rule extraction. For the correction step, we use genetic algorithm. We evaluate our approach by finding and fixing potential defects in four open-source systems. For all these systems, we found, in average, more than 80% 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 78%of detected defects.**

Keywords — design defects; software maintenance; search-based software engineering;by example.

I. INTRODUCTION

Many studies reported that software maintenance, traditionally defined as any modification made on a system after its delivery, consumes up to 90% of the total cost of a typical software project [1]. Adding new functionalities, correcting bugs, and modifying the code to improve its quality are major parts of those costs [3].

There has been much research focusing on the study of bad design practices, also called defects, antipatterns [2], smells [3], or anomalies [1] in the literature. Although these bad practices are sometimes unavoidable, they should be in general prevented by the development teams and removed from their code base as early as possible.

Detecting and removing these defects help developers to easily understand source code. However, it is a difficult, time-consuming, and to some extent, a manual process [6]. The number of outstanding 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 the 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 cope with this magnitude of activity, several automated detection techniques have been proposed [5] [6] [7] [4] [8].

The vast majority of existing work in defect detection 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 using combinations of mainly quantitative (metrics), structural, and/or lexical information. However, in an exhaustive scenario, the number of possible defects to manually characterize 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. Another work [5] proposes to generate detection rules using formal definitions of defects. This partial automation of rule writing helps developers concentrating 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 from the source code. Some work proposes “standard” refactoring solutions that can be applied by hand for each kind of defect [2]. However, no one can prove or ensure the generality of these solutions to any kind of defects or systems. The majority of other works start from the hypothesis that useful refactorings are those which improve the metrics [20]. Some limitations can be drawn from studying these metric-based approaches. First, how to determine the useful metrics for a given system. Second, how best to combine multiple metrics. Third, improving the metrics values do not means that specific defects are corrected.

These difficulties contrast with the availability of defect repositories in many companies where defects are manually identified, corrected and documented. These two observations are at the origin of the work described in this paper. Indeed, defect repositories contain valuable information that can be used to mine regularities about defect manifestations that can be translated into detection rules. More concretely, we propose a new automated approach to derive rules for design defect detection. Instead of specifying rules manually for detecting each defect type or semi-automatically using defect definitions, we extract them from valid instances of design defects. In our setting, we view the generation of design defect rules as an optimization problem where the quality of a detection rule is determined by its ability to conform to an example base. The example base contains instances of defects (classes) that were validated manually.

The generation process starts from an initial set of rules representing random combinations of metrics. Then this set is refined progressively according to its ability to detect defects present in the example base. Due to the very large number of possible rules (metric combinations), a heuristic method is used instead of an enumerative one to explore the space of possible solutions. To this end, we use a rule induction heuristic, called Genetic Programming (GP) [25] to find a near-optimal set of detection rules. GP is a variant of genetic algorithm (GA) [26] with a different solution representation (more suitable to rules generation).

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.

The remainder of this 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.

II. Research Problem

To understand better our contribution, it is important to define clearly the problems of defect detection and correction. In this section, we start by giving the definitions of important concepts. Then, we detail the specific problems that are addressed by our approach.

A. *Definitions*

Design defects, also called design anomalies, refer to design situations that adversely affect the development of 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 paths. 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, named anti-patterns, that are documented in the literature. In this section, we define the following three that will be used to illustrate our approach and in the detection tasks of our validation study.

- 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.

For the different types of defects, the initial authors focus on describing the symptoms to look for, in order to identify occurrences of these defects.

From the detection standpoint, the process consists of finding code fragments in the system 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 the depth of inheritance tree DIT, weighted methods per class WMC and coupling between objects CBO. Variations of this metrics, adaptations of procedural ones as well as new metrics were also used such as the 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, and number of private fields NPRIVFIELD.

The detected defects can be fixed by applying some refactoring operations. William Opdyke define 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 [27]. 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]. Following, some examples of refactoring operations: *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].

B. *Problem Statement*

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 discuss some of the open issues related to the detection and correction problems.

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 implements all the behavior and all the others 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 ones are Blob candidates depends heavily on the interpretation of each analyst.

How to define 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.

Is improving code quality really means that detected defects are fixed? In the majority of situations, we can improve the code quality without fixing design defects. In addition, different problems are related to: how to determine the useful metrics for a given system and how best to combine multiple metrics.

How to generalize correction solutions? The correction solutions should not be specific to only some defect types.

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.

To address or circumvent the above mentioned issues, we propose to use examples of manually found design defects to derive detection rules. Such examples are in general available and documents as part of the maintenance activity (version control logs, incident reports, inspection reports, etc.). The use of examples allows to derive rules that are specific to a particular company rather than rules that are supposed to be applicable to any context. This includes the definition of thresholds that correspond to the company best practices. Learning from examples aims also at reducing the list of detected defect candidate. These learned rules will be used to evaluate the quality of the recommended refactoring solutions.

III. Approach Overview

This section shows how, under some circumstances, design defects detection and correction can be seen as an optimization problem. We also show why the size of the corresponding search space makes heuristic search necessary in order to explore it and generate the sought rules and refactoring solutions.

A. *Overview*

We propose an approach that uses knowledge from previously manually inspected projects, called defects examples, in order to detect design defects that will serve to generate new detection rules based on combinations of software quality metrics. In short, the detection rules are automatically derived by an optimization process that exploits the available examples. Another optimization process is used to generate a refactoring strategy that fixes detected defects. The search process is guided by an evaluation function that minimizes the number of detected defects using learned detection rules.

Figure 1 shows the general structure of our approach. We distinguish between two important steps: 1) defects detection and 2) correction. The detection step takes as inputs a base of examples (i.e., a set of defects examples) and a set of quality metrics, and generates as output a set of rules. The generation process can be viewed as the combination of the metrics that best detect the defects examples. In other words, the best set of rules is the one that detect the maximum number of defects.

The correction step takes as inputs the generated detection rules and a set of refactoring operations, and generates as output a sequence of refactorings. The generation process can be viewed as the combination of the refactorings that best correct (eliminate) the detected defects. In short, the best set of refactorings is the one that minimizes the number of detected defects using the detection rules.

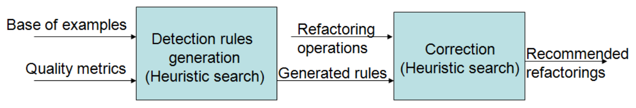


Fig 1. Approach overview

As showed in Figure 2, the base of examples contains projects (systems) that were manually inspected to detect possible defects. During a training stage, these systems are iteratively evaluated using rules generated by the algorithm. A fitness functions calculates the quality of each solution (rules) by comparing the list of detected defects with the expected ones from the base of examples.

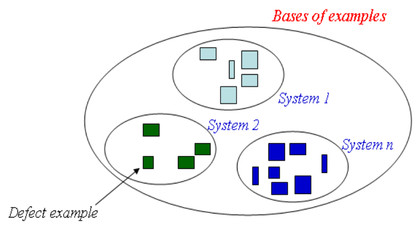


Fig 2. Base of examples

As many metrics combinations are possible, the detection rules generation process is a combinatorial optimization problem. The number of possible solutions quickly becomes huge as the number of metrics increases. A deterministic search is not practical in such cases, and the use of heuristic search is warranted (see Problem Complexity below). The dimensions of the solution space are set by the metrics and logical operations between them: union (metric1 OR metric2) and intersection (metric1 AND metric2). A solution is determined by assigning a threshold value to each metric. The search is guided by the quality of the solution according to the number of detected defects in comparison to the expected ones form the base of examples. The same things for the correction step. In fact, the number of refactorings combination is very high. A heuristic search is needed to explore this large number of combination.

B. *Problem Complexity*

Our approach assigns a threshold value to each metric. The number m of possible threshold values is usually very large. Furthermore, the rules generation process consists of finding the best combination between n metrics. In this context, $(n!)^m$ possible solutions have to be explored. This value quickly becomes huge. For example, a list of 5 metrics with 6 possible thresholds necessitates the evaluation of up to 1206 combinations. The same complexity is need for the correction step. In addition the number of possible refactorings combination, the order of applying them is important. Thus, the complexity is $(k!)^k$ where k is the number possible refactorings.

Considering these magnitudes, an exhaustive search cannot be used within a reasonable time frame. In such situations, or when a formal approach is not possible, heuristic search can be used.

IV. Refactoring by Example

We describe in this section the adaptation of GP and GA to automate, respectively, design defects detection and correction. To apply them to a specific problem, one must specify the encoding of solutions, the operators that allow movement in the search space so that new solutions are obtained, and the fitness function to evaluate a solution's quality. These three elements are detailed in the next subsections.

A. *Defects Detection Using Genetic Programming*

When designing a GP system, some elements have to be considered: the set of functions and terminals (defined below) that will be used to create the GP population, the representation of the individuals, and the fitness function used to measure the quality of the candidate solutions. In addition, crossover and mutation operators have to be designed according to the individual's representation, and a method for selection of the best individuals has to be implemented. The next sections explain the main concepts involved in the design of these elements for the automatic generation of design defects detection rules.

1) Genetic Programming Overview

Genetic programming is a powerful search method inspired by natural selection. The basic idea is to make a population of candidate "programs" evolve toward the solution of a specific problem. A program (an individual of the population) is usually represented in the form of a tree, where the internal nodes are functions (operators) 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 its ability to solve the target problem. Then, it is subjected to the action of genetic operators such as reproduction and crossover. The reproduction operator selects individuals in the current population in proportion to their fitness values, so that the more fit an individual is, the higher the probability that it will take part in the next generation of individuals. The crossover operator replaces a randomly selected subtree of an individual with a randomly chosen subtree from another individual.

Once reproduction and crossover have been applied according to given probabilities, the newly created generation of individuals is evaluated by the fitness function. This process is repeated iteratively, usually for a fixed number of generations. The result of genetic programming (the best solution found) is the fittest individual produced along all generations.

```
Input: Set of quality metrics
Input: Set of systems S evaluated manually (defects
examples).
Output: Detection rules.
1: initial_population(P, Max_size)
2: P:= set_of(I)
3: I := rules(R, Defect_Type)
4: repeat
5:        for all I ∈ P do
6:                detected_defects := execute_rules(R)
7:                fitness(I) := compare(detected_defects,
                  defects_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 3. High-level pseudo-code for GP adaptation to our problem

The pseudocode for the algorithm is given in Figure 3. As the Figure shows, the algorithm takes as input a set of quality metrics that compose rules and a set of manually detected defects examples in some systems. Lines 1–3 construct an initial GP population, based on a specific representation, using the quality metrics given at the input. The population stands for a set of possible solutions representing detection rules (metrics combination). Lines 4–13 encode the main GP loop, which searches for the best metrics combination. During each iteration, we evaluate the quality of each solution (individual) in the population, and the solution having the best fitness (line 9) is saved. We generate a new population of solutions using the crossover operator (line 10) to the selected solutions; each pair of parent solutions produces two children (new solutions). We include the parent and child variants in the population and then apply the mutation operator to each variant; this produces the population for the next generation. The algorithm terminates when it achieve the termination criteria (maximum iteration number), and return the best set of detection rules (solution).

2) *Genetic Programming Adaptation*

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

a) *Program Representation*

Functions and terminals

In GP, an individual (i.e., solution) is composed of terminals and functions. Therefore, when applying GP to solve a specific problem, they should be carefully selected and designed to satisfy the requirements of the current problem. After evaluating many parameters related to the design defects detection problem, the terminal set and the function set that are used in our algorithm are described next.

In our problem, the terminals correspond to different quality metrics with their threshold values. The functions that can be used between these metrics are Union (OR) and Intersection (AND).

Individual representation

The set of candidates solutions (rules) corresponds to a logic program that is represented as a forest of AND-OR trees. For example, consider the following logic program:

C1: defect(blob) :- locClass(upper, 1500), locMethod(upper,129).
C2: defect(blob) :- nmd(upper, 100).
C3: defect(spaghettiCode) :- locMethod(upper,151).
C4: defect(functionalDecomposition) :- nPrivField(upper, 7), nmd(equal,16).

The predicate symbols (e.g. defect, locClass, locMethod, etc.) correspond to metrics names or defect specifier. When associated with subjects such as blob, upper, 1500, etc., they define propositional variables that can serve to build the defect detection rules. The subjects specify the proprieties of each predicate. For example, the defect predicate contains only one parameter corresponding to the type. All other predicates contain two parameters describing the operator used (upper, lower, equal) and the threshold value. The threshold value domain depends to the metric.

The set of rules C1-C4 can be described as follows:

R1 : IF (LOCCLASS ≥ 1500 AND LOCMETHOD ≥ 129) OR (NMD ≥ 100) THEN defect = blob

R2 : IF (LOCMETHOD ≥ 151) THEN defect = spaghetti code

R3 : IF (NPRIVFIELD ≥ 7 AND NMD = 16) THEN defect = functional decomposition

Thus, the first rule is represented as a sub-tree of nodes (AND-OR, metrics…) as shown in Figure 4. The main program tree will be a composition of three sub-trees: R1 AND R2 AND R3.

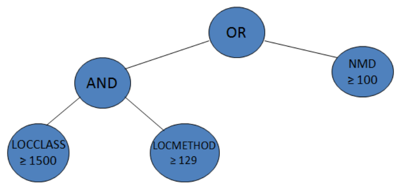


Fig 4. Tree representation (blob detection rule)

b) *Selection and Genetic Operators*

Selection

There are many selection algorithms where more fit individuals are allocated more copies in the next generations than less fit ones. For our initial prototype, we used stochastic universal sampling (SUS) [25], in which each individual's probability of selection is directly proportional to its relative fitness in the population. We use SUS to select pop size/2 new members of the population.

Mutation

The mutation operator can be applied to either a function node, a terminal node or a tree. It starts by randomly selected a node in the tree. Then, if the selected node is a terminal (quality metric), it is replaced by another terminal (metric or another threshold value); if it is a function (and-or), it is replaced by a new function; and if tree mutation is to be carried out, the node and its subtree are replaced by a new randomly generated subtree.

To illustrate the mutation process, consider again the example sown in Figure 4, which corresponds to a candidate rule for blob detection. Figure 5 illustrate the effect of a mutation that deletes node NMD, leading to the automatic deletion of node OR (no left subtree), and that replaces node LOCMETHOD by node NPRIVFIELD with a new threshold value.

Crossover

Two parent individuals are selected and a subtree is picked on each one. Then crossover swaps the nodes and their relative subtrees from one parent to the other. This operator must ensure the respect of the depth limits. The crossover operator can be applied with only parents having the same rules category (defect type to detect). Each child thus combines information from both parents. In any given generation, a variant will be the parent in at most one crossover operation.

Figure 6 shows an example of the crossover process. In fact, the rule R1 and a rule RI1 form another individual (solution) are combined to generate new two rules. The right subtree of R1 is swapped with the left subtree of the second rule.

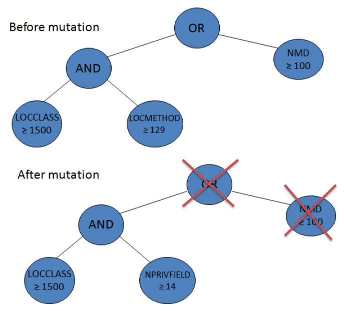


Fig 5. Mutation operator

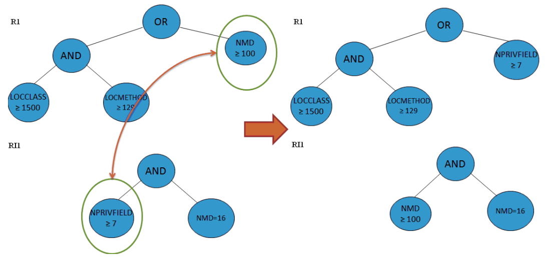


Fig 6. Crossover operator

c) Fitness Function

The fitness function quantifies the quality of the generated rules. The goal is to define an efficient fitness function that it is not complex to reduce the computational complexity.

As discussed in Section 3, the fitness function checks 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

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

where p is the number of detected classes, t is the number of defects in the base of example and a_i has value 1 if the ith detected classes 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. The true defects of this system are described 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 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 correspond 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:

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

B. Design Defects Correction Using Genetic Algorithm

1) Genetic Algorithm Overview

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.

2) Genetic Algorithm Adaptation

a) 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 a *n*-dimensional space, where each dimension corresponds to one refactoring operation. Figure 7 shows an example where the *i*th individual (solutions) represents a combination of refactoring operations to apply. To ease the manipulation of these operations, we use logic predicates to describe them. For example, the predicates *MoveMethod(division, Departement, University*) indicates that the method *division* is moved from class *Departement* to class *University*.

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 7. Individual representation

b) 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 o_1 and o_2 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).

c) 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

$$f = \min(n)$$

Where n is the number of detected classes.

V. 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.

A. 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 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.

B. System Studied

We used four open-source Java projects to perform our experiments: GanttProject (Gantt for short) v1.10.2, Quick UML v2001, 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

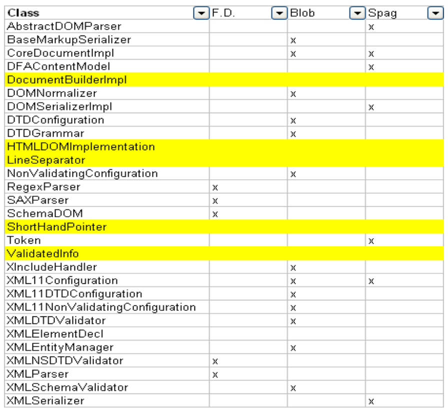
We chose the Xerces-J, ArgoUML, Quick UML and Gantt libraries because they are medium-sized open-source projects and were analysed in related work. The version of Gantt studied was known to be of poor quality, which has led to a new major revised version. ArgoUML, Xerces-J and Quick UML, on the other 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. 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. As previously mentioned in Introduction, we used a 4-fold cross validation procedure. For each fold, one open source project is evaluated by using the remaining two systems as base of examples. For example, Xerces-J is analyzed using some defects examples from Gantt, ArgoUML and Quick UML.

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, Quick UML, ArgoUML and Gantt.

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

C. 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 8. Xerces-J Results (Top-30 classes)

TABLE IV. DETECTION RESULTS.

System	Precision	Recall
GanttProject	Blob : 100% SC : 93% FD : 91%	100% 97% 94%
Xerces-J	Blob : 97% SC: 90% FD: 88%	100% 88% 86%
ArgoUML	Blob : 93% SC: 88% FD: 82%	100% 91% 89%
QuickUML	Blob : 94% SC: 84% FD: 81%	98% 93% 88%

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)

Figure 8 and Tables 4-5 summarize our findings. Figure 8 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, our precision was 86% upper than 63% DÉCOR precision. 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 80% 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 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.

D. 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 Gantt or ArgoUML Quick UML or Xerces-J 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 Xerces-J or ArgoUML or Quick UML or Gantt, 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.

VI. 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.

VII. 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.