Search-Based Design Defects Detection by Example

Marouane Kessentini[1], Houari Sahraoui[1], Mounir Boukadoum[2], and Manuel Wimmer[3]

[1] DIRO, Université de Montréal, Canada
{Kessentm, sahraouh}@iro.umontreal.ca
[2] DI, Université du Québec à Montréal, Canada
mounir.boukadoum@uqam.ca
[3] Vienna University of Technology, Austria
wimmer@big.tuwien.ac.at

Abstract. We propose an automated approach to detect 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). In our setting, we use and compare between different heuristic search algorithms for rule extraction: Harmony Search, Particle Swarm Optimization, and Simulated Annealing. We evaluate our approach by finding potential defects in two open-source systems. For all these systems, we found, in average, more than 75% 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.

Keywords: Design defects, software quality, metrics, search-based software engineering, by example.

1 Introduction

Many studies report 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 software project [2]. Adding new functionalities, correcting bugs, and modifying the code to improve its quality (by detecting and correcting design defects) are major parts of those costs [1]. There has been much research devoted to the study of bad design practices, also known in the literature as defects, antipatterns [1], smells [2], or anomalies [3]. Although bad practices are sometimes unavoidable, they should otherwise be prevented by the development teams and removed from the code base as early as possible in the design cycle.

Detecting and fixing defects is a difficult, time-consuming, and to some extent, manual process [5]. The number of outstanding software defects typically exceeds the resources available to address them [4]. In many cases, mature software projects are forced to ship with both known and unknown defects for lack of the development resources to deal with everyone. For example, in 2005, one Mozilla developer

D. Giannakopoulou and F. Orejas (Eds.): FASE 2011, LNCS 6603, pp. 401–415, 2011.
© Springer-Verlag Berlin Heidelberg 2011

claimed that "everyday, almost 300 bugs and defects appear . . . far too much for only the Mozilla programmers to handle". To help cope with this magnitude of activity, several automated detection techniques have been proposed [5, 7].

The vast majority of existing work in defect detection relies on declarative rule specification [5, 7]. In these settings, rules are manually defined to identify the key symptoms that characterize a defect, using combinations of mainly quantitative, structural, and/or lexical indicators. However, in an exhaustive scenario, the number of possible defects to characterize manually with rules can be very large. For each defect, the metric combinations that serve to define its detection rule(s) require a substantial calibration effort to find the right threshold value to assign to each metric. Alternatively, [5] proposes to generate detection rules using formal definitions of defects. This partial automation of rule writing helps developers to concentrate on symptom description. Still, translating symptoms into rules is not obvious because there is no consensual symptom-based definition of design defects [8]. When a consensus exists, the same symptom could be associated with many defect types, which may compromise the precise identification of defect types. These difficulties explain a large portion of the high false-positive rates found in existing research [8].

The previous difficulties contrast with the ease of finding defect repositories in several companies, where defects that are manually identified and corrected are documented. This observation is at the origin of the work described herein. We start from the premise that defect repositories contain valuable information that can be used to mine regularities about defect manifestations, subsequently leading to the generation of 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 a base of examples that contains instances of manually validated defects (classes).

The generation process starts from an initial set of rules that consists of random combinations of metrics. Then, the rules evolve progressively according to the set's ability to detect the documented defects in the example base. Due to the potentially huge number of possible metric combinations that can serve to define rules, a heuristic approach is used instead of exhaustive search to explore the space of possible solutions. To that end, we use and compare between three rule induction heuristics : Harmony Search (HS) [9], Particle Swarm Optimization (PSO) [28] and Simulated Annealing (SA) [27] to find a near-optimal set of detection rules.

We evaluated our approach on defects present in two open source projects: GANTTPROJECT [11] and XERCES [12]. We used an n-fold cross validation procedure. For each fold, six projects are used to learn the rules, which tested on the remaining seventh project. Almost all the identified classes in a list of classes tagged as defects (blobs, spaghetti code and functional decomposition) in previous projects [17] were found, with a precision superior to 75%.

The remainder of this paper is structured as follows. Section 2 is dedicated to the problem statement. In Section 3, we describe the overview of our proposal. Then Section 4 describes the principles of the different heuristic algorithms used in our approach and the adaptations needed to our problem. Section 5 presents and discusses

the validation results. A summary of the related work in defect detection is given in Section 6. We conclude and suggest future research directions in Section 7.

2 Problem Statement

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

2.1 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 [1] 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 both 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 [3]. The most widely used metrics are the ones defined by Chidamber and Kemerer [14]. 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.

2.2 Problem Statement

Although there is a consensus that it is necessary to detect design anomalies, our experience with industrial partners showed that there are many open issues that need to be addressed when defining a detection tool. Design anomalies have definitions at different levels of abstraction. Some of them are defined in terms of code structure, others in terms of developer/designer intentions, or in terms of code evolution. These definitions are in many cases ambiguous and incomplete. However, they have to be mapped into rigorous and deterministic rules to make the detection effective.

In the following, we discuss some of the open issues related to the detection.

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.

How to deal with the context? In some contexts, an apparent violation of a design principle is considered as a consensual practice. For example, a class Log 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.

In addition to these issues, the process of defining rules manually is complex, time-consuming and error-prone. Indeed, the list of all possible defect types can be very large. And each 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 example 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 candidates.

3 Approach Overview

This section shows how, under some circumstances, design defects detection can be seen as an optimization problem. We also show why the size of the corresponding search space makes heuristic search necessary to explore it.

3.1 Overview

We propose an approach that uses knowledge from previously manually inspected projects in order to detect design defects, called defects examples, to generate new detection rules based on a combinations of software quality metrics. More specifically, the detection rules are automatically derived by an optimization process that exploits the available examples.

Figure 1 shows the general structure of our approach. The approach 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 metrics that best detect the defects examples. In other words, the best set of rules is that who detect the maximum number of defects.

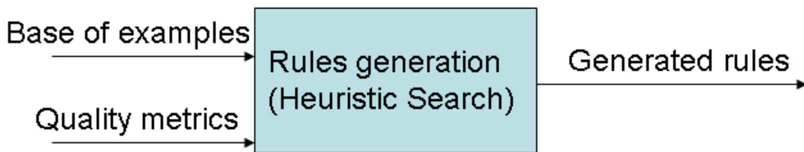

Fig. 1. Approach overview

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

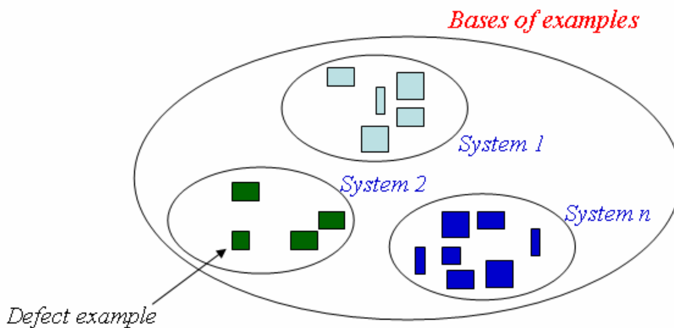

Fig. 2. Base of examples

As many metrics combinations are possible, the rules generation 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, hence the use of heuristic search. The dimensions of the solution space are the metrics and some operators between them: union (metric1 OR metric2) and intersection (metric1 AND metric2). A solution is determined by the assignment of a threshold value to each metric. The search is guided by the quality of the solution according to the number of detected defects comparing to expected ones in the base.

To explore the solution space, we use different heuristic algorithms that will be detailed in Section 4.

3.2 Problem Complexity

Our approach assigns to each metric a corresponding threshold value. The number m of possible threshold value is 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 can quickly become huge. A list of 5 metrics with 6 possible thresholds necessitates exploring at least 120^6 combinations. Considering these magnitudes, an exhaustive search cannot be used within a reasonable time frame. This motivates the use of a heuristic search if a more formal approach is not available or hard to deploy.

4 Search-Based Design Defect Detection by Example

We describe in this section the adaptation of three different heuristic algorithms to the design defects rules generation problem. To apply it to a specific problem, one must specify the encoding of solutions and the fitness function to evaluate a solution's quality. These two elements are detailed in subsections 4.1 and 4.2 respectively.

4.1 Solution Representation

One key issue when applying a search-based technique is finding a suitable mapping between the problem to solve and the techniques to use, *i.e.*, in our case, generating design defects rules. As stated in Section 3, we view the set of potential solutions as points in a *n*-dimensional space where each dimension corresponds to one metric or operator (union or intersection). Figure 3 shows an illustrative example which describes this rule: if (WMC≥4) AND (TCC≥7) AND (ATFD≥1) Then Defect_Type(1)_detected. The WMC, TCC and ATFD are metrics defined as [14]:

- *Weighted Method Count* (WMC) is the sum of the statical complexity of all methods in a class. We considered the McCabe's cyclomatic complexity as a complexity measure.
- *Tight Class Cohesion* (TCC) is the relative number of directly connected methods.
- *Access to Foreign Data* (ATFD) represents the number of external classes from which a given class accesses attributes, directly or via accessor-methods.

- We used three types of defects : (1) blob, (2) spaghetti code and (3) functional decomposition.

The operator used as default is the intersection (and). The other operator (union) can be used as a dimension. The vector presented in Figure 3 generates only one rule. However, a vector may contain many rules separated by the dimension "Type".

WMC	TCC	ATFD	Type
HigherThan(4)	HigherThan(7)	HigherThan(1)	Equal(1)

Fig. 3. Solution Representation

4.2 Evaluating Solutions

The *fitness function* quantifies the quality of the generate rules. As discussed in Section 3, the fitness function must consider the following aspect:

- Maximize the number of detected defects comparing to expected ones in the base of examples

In this context, we define the fitness function of a solution as

$$f = \sum_{i=1}^{p} a_i \quad (1)$$

Where p represents the number of detected classes. a_i has value 1 if the ith detected classes exists in the base of examples, and value 0 otherwise.

4.3 Search Algorithms

4.3.1 Harmony Search (HS)

The HS algorithm is based on natural musical performance processes that occur when a musician searches for a better state of harmony, such as during jazz improvisation [9]. Jazz improvisation seeks to find musically pleasing harmony as determined by an aesthetic standard, just as the optimization process seeks to find a global solution as determined by a fitness function. The pitch of each musical instrument determines the aesthetic quality, just as the fitness function value is determined by the set of values assigned to each dimension in the solution vector.

In general, the HS algorithm works as follows:

Step 1. Initialize the problem and algorithm parameters.

The HS algorithm parameters are specified in this step. They are the harmony memory size (HMS), or the number of solution vectors in the harmony memory; harmony memory considering rate (HMCR); bandwidth (bw); pitch adjusting rate (PAR); and the number of improvisations (K), or stopping criterion.

Step 2. Initialize the harmony memory.
The initial harmony memory is generated from a uniform distribution in the ranges $[x_{imin}, x_{imax}]$ (i = 1,2, . . .,N) , as shown in Equation 1 :

$$HM = \begin{bmatrix} x_1^1 & x_2^1 & . & . & x_N^1 \\ . & & & & \\ . & & & & \\ x_1^{HMS} & x_2^{HMS} & & . & x_N^{HMS-1} \end{bmatrix} \quad (2)$$

Step 3. Improvise a new harmony.
Generating a new harmony is called improvisation. The new harmony vector x' = (x'$_1$, x'$_2$,…, x'$_N$) is determined by the memory consideration, pitch adjustment and random selection.
Step 4. Update harmony memory.
If the fitness of the improvised harmony vector x' = (x'$_1$, x'$_2$,…, x'$_N$) is better than the worst harmony, replace the worst harmony in the IHM with x'.
Step 5. Check the stopping criterion: If the stopping criterion (maximum number of iterations K) is satisfied, computation is terminated. Otherwise, step 3 is repeated.

4.3.2 Particle Swarm Optimization (PSO)

PSO is a parallel population-based computation technique [31]. It was originally inspired from the flocking behavior of birds, which emerges from very simple individual conducts. Many variations of the algorithm have been proposed over the years, but they all share a common basis. First, an initial population (named swarm) of random solutions (named particles) is created. Then, each particle flies in the *n*-dimensional problem space with a velocity that is regularly adjusted according to the composite flying experience of the particle and some, or all, the other particles. All particles have fitness values that are evaluated by the objective function to be optimized. Every particle in the swarm is described by its position and velocity. A particle position represents a possible solution to the optimization problem, and velocity represents the search distances and directions that guide particle flying. In this paper, we use basic velocity and position update rules defined by [31]:

$$V_{id} = W * V_{id} + C_1 * rand() * (P_{id} - X_{id}) + C_2 * Rand() * (P_{gd} - X_{id}) \quad (3)$$

$$X_{id} = X_{id} + V_{id} \quad (4)$$

At each time (iteration), V_{id} represents the particle velocity and X_{id} its position in the search space. P_{id} (also called *pbest* for local best solution), represents the ith particle's best previous position, and P_{gd} (also called *gbest* for global best solution), represents the best position among all particles in the population. w is an inertia term; it sets a balance between the global and local exploration abilities in the swarm. Constants c_1 and c_2 represent cognitive and social weights associated to the individual and global behavior, respectively. There are also two random functions *rand()* and *Rand()*

(normally uniform in the interval [0, 1]) that represent stochastic acceleration during the attempt to pull each particle toward the *pbest* and *gbest* positions. For a *n*-dimensional search space, the ith particle in the swarm is represented by a *n*-dimensional vector, $\boldsymbol{x}_i=(x_{i1},x_{i2},\ldots,x_{id})$. The velocity of the particle, *pbest* and *gbest* are also represented by *n*-dimensional vectors.

4.3.3 Simulated Annealing (SA)

SA [19] is a search algorithm that gradually transforms a solution following the annealing principle used in metallurgy.

After defining an initial solution, the algorithm iterates the following three steps:

1. Determine a new neighboring solution,
2. Evaluate the fitness of the new solution
3. Decide on whether to accept the new solution in place of the current one based on the fitness gain/lost ($\Delta cost$).

When $\Delta cost < 0$, the new solution has lower cost than the current solution and it is accepted. For $\Delta cost > 0$ the new solution has higher cost. In this case, the new solution is accepted with probability $\boldsymbol{e}^{-\Delta cost/T}$. The introduction of a stochastic element in the decision process avoids being trapped in a local minimum solution. Parameter *T*, called temperature, controls the acceptance probability of a lesser good solution. *T* begins with a high value, for a high probability of accepting a solution during the early iterations. Then, it decreases gradually (cooling phase) to lower the acceptation probability as we advance in the iteration sequence. For each temperature value, the three steps of the algorithm are repeated for a fixed number of iterations.

5 Validation

To test our approach, we studied its usefulness to guide quality assurance efforts on an open-source program. 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 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?

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.

5.2 System Studied

We used two open-source Java projects to perform our experiments: GanttProject (Gantt for short) v1.10.2, and Xerces-J v2.7.0.

Table 1. Program statistics

Systems	Number of classes	KLOC
GanttProject v1.10.2	245	31
Xerces-J v2.7.0	991	240

Table 1 summarizes facts on these programs. Gantt is a tool for creating project schedules by means of Gantt charts and resource-load charts. Gantt enables breaking down projects into tasks and establishing dependencies between these tasks. Xerces-J is a family of software packages for parsing and manipulating XML. It implements a number of standard APIs for XML parsing.

In our experiments, we used some of the classes in Gantt as our example set of design defects. These examples are validated manually by a group of experts [17]. We choose the Xerces-J 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 lead to a new major version. Xerces-J on the other hand has been actively developed over the past 10 years and its design has not been responsible for a slowdown of its development.

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 antipatters including: 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. 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. Thus, Xerces-J is then analyzed using some defects examples from Gantt and vice-versa.

The obtained results were compared to those of DECOR [5]. For every antipattern in Xerces-J and Gantt, they published 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). Our comparison is consequently done using precision and recall.

5.3 Results

Tables 2, 3 and 4 summarize our findings. The results show that HS performs comparing to PSO and SA. In fact, the two global search algorithms HS and PSO are suitable to explore large search space. For Gantt, our precision average is 87%. DECOR on the other hand has a combined precision of 59% for its detection on the

Table 2. HS results

System	Precision	Recall
Gantt	Spaghetti: 82% Blob: 100% F.D: 87%	Spaghetti: 90% Blob: 100% F.D: 47%
Xerces-J	Spaghetti:82% Blob:93% F.D:76%	Spaghetti:84% Blob:94% F.D:60%

Table 3. PSO results

System	Precision	Recall
Gantt	Spaghetti: 79% Blob: 100% F.D: 82%	Spaghetti: 94% Blob: 100% F.D: 53%
Xerces-J	Spaghetti:89% Blob:91% F.D:73%	Spaghetti:81% Blob:92% F.D:68%

Table 4. SA results

System	Precision	Recall
Gantt	Spaghetti: 81% Blob: 100% F.D: 80%	Spaghetti: 95% Blob: 100% F.D: 51%
Xerces-J	Spaghetti:77% Blob:91% F.D:71%	Spaghetti:80% Blob:92% F.D:69%

same set of antipatterns. For Xerces-J, our precision average is of 83%. For the same dataset, DECOR had a precision of 67%. However, the recall score for both systems is less than 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 is able to accurately identify design anomalies more accurately than DECOR (RQ1).

We noticed that our technique does not have a bias towards the detection of specific anomaly types. In Xerces-J, we had an almost equal distibution of each antipattern. On Gantt, the distribution is not as balanced. This is principally due to the number of actual antipatterns in the system.

The detection of FDs using only metrics seems difficult. This difficulty is why DECOR includes an analysis of naming conventions to perform its detection. Using naming convention means that their results depend on the coding practices of a development team. Our results are however comparable to theirs while we do not leverage lexical information. The complete results of our experiments, including the comparison with DÉCOR, can be found in [18].

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 Gantt or Xerces-J directly, without any adaptation, the technique can be used out of the box and this will produce good detection and recall results for the detection of antipatterns for the two systems studied.

The performance of this detection was superiour to that of DECOR. In an industrial setting, we could expect a company to start with Xerces-J 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 used rules which are generated randomly 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 observed an average recall and precision more than 80% for both Gantt and Xerces-J with the three different heuristic search algoithms. Furthermore, we found that the majority of defects detected are found in every execution. We consequently believe that our technique is stable, since the precision and recall scores are approximately the same for different executions.

Another important advantage comparing to machine learning techniques is that our search algorithms do not need both positive (good code) and negative (bad code) examples to generate rules like for example Inductive Logic Programming [19].

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 2GB of RAM). The execution time for rules generation with a number of iteration (stopping criteria) fixed to 500 is less than three minutes (2min36s). This indicates that our approach is scalable from the performance standpoint. However, the execution time depends to 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 in comparison with using DECOR. In any case, our approach is meant to apply to situations where manual rule-based solutions are normally not easily available.

6 Related Work

Several studies have recently focused on detecting design defects in software using different techniques. These techniques range from fully automatic detection to guided manual inspection. The related work can be classified into three broad categories: metric-based detection, detection of refactoring opportunities, visual-based detection.

In 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. [20] use metrics to evaluate frameworks with the goal of improving them. They introduce the concept of multi-metrics, as an n-tuple of metrics expressing a quality

criterion (e.g., modularity). The main limitation of the two previous contributions is the difficulty to define manually threshold values for metrics in the rules. To circumvent this problem, Alikacem et al. [21] express defect detection as fuzzy rules with fuzzy label for metrics, e.g., small, medium, large. When evaluating the rules, actual metric values are mapped to truth value for the labels by means of membership functions. Although no thresholds have to be defined, still, it is not obvious to decide for membership functions.

The previous approaches start from the hypothesis that all defect symptoms could be expressed in terms of metrics. Actually, many defects involve notions that could not quantified. This observation was the foundation of the work of Moha et al. [5]. In their approach, named DECOR, 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 with 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, all 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.

In the second category of work, defects are not detected explicitly. They are implicitly because, the approaches refactor a system by detecting elements to change to improve the global quality. For example, in [22], 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 maximize a function, which captures the variations of a set of metrics [23]. 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 phase. In [8, 26], we have proposed an approach 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. Taking inspiration from artificial immune systems, we generated a set of detectors that characterize different ways that a code can diverge from good practices. We then used these detectors to measure how far code in assessed systems deviates from normality.

7 Conclusion

In this article, we presented a novel approach for tackling the problem of detecting design defects. Typically, researchers and practitioners try to characterize different types of common design defects and present symptoms to use in order to locate them in a system. In our work, we show that we do not need this knowledge to perform detection. Instead, all we need is some examples of design defects to generate

detection rules. Interestingly enough, 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 were promising. The detection process uncovered different types of design defects was more efficiently than DECOR. The comparison between three heuristic algorithm shows that HS give better results than PSO and SA. Furthermore, as DECOR needed an expert to define rules, our results were achieved without any expert knowledge, relying only on the bad structure of Gantt to guide the detection process.

The benefits of our approach can be summarized as follows: 1) it is fully automatable; 2) it does not require an expert to manually write rules for every defect type and adapt them to different systems; 3) the rule generation process is executed once; then, the obtained rules can be used to evaluate any system.

The major limitations of our approach are: 1) the generated rules are based on metrics, and some defects may require additional or different knowledge to be detected; 2) the approach requires the availability of a code base that is representative of bad design practices, and where all the possible design defects are already detected.

As part of our future work, we plan to explore the second step: correction of detected design defects (refactoring). Furthermore, we need to extend our base of examples with other bad-designed code in order to take into consideration different programming contexts.

References

1. Brown, W.J., Malveau, R.C., Brown, W.H., McCormick III, H.W., Mowbray, T.J.: Anti Patterns: Refactoring Software, Architectures, and Projects in Crisis, 1st edn. John Wiley and Sons, Chichester (March 1998)
2. Fowler, M.: Refactoring – Improving the Design of Existing Code. 1st edn. Addison-Wesley, Reading (June 1999)
3. Fenton, N., Pfleeger, S.L.: Software Metrics: A Rigorous and Practical Approach, 2nd edn. International Thomson Computer Press, London (1997)
4. Khomh, F., Vaucher, S., Guéhéneuc, Y.-G., Sahraoui, H.: A Bayesian Approach for the Detection of Code and Design Smells. In: Proc. of the ICQS 2009 (2009)
5. Moha, N., Guéhéneuc, Y.-G., Duchien, L., Meur, A.-F.L.: DECOR: A method for the specification and detection of code and design smells. Transactions on Software Engineering (TSE), 16 pages (2009)
6. Liu, H., Yang, L., Niu, Z., Ma, Z., Shao, W.: Facilitating software refactoring with appropriate resolution order of bad smells. In: Proc. of the ESEC/FSE 2009, pp. 265–268 (2009)
7. Marinescu, R.: Detection strategies: Metrics-based rules for detecting design flaws. In: Proc. of ICM 2004, pp. 350–359 (2004)
8. Kessentini, M., Vaucher, S., Sahraoui, H.: Deviance from perfection is a better criterion than closeness to evil when identifying risky code. In: Proc. of the International Conference on Automated Software Engineering, ASE 2010 (2010)
9. Lee, K.S., Geem, Z.W.: A new meta-heuristic algorithm for continuous engineering optimization: harmony search theory and practice. Comput. Method Appl. M 194(36-38), 3902–3933 (2005)

10. Lee, K.S., Geem, Z.W., Lee, S.H., Bae, K.W.: The harmony search heuristic algorithm for discrete structural optimization. Eng Optimiz 37(7), 663–684 (2005)
11. http://ganttproject.biz/index.php
12. http://xerces.apache.org/
13. Riel, A.J.: Object-Oriented Design Heuristics. Addison-Wesley, Reading (1996)
14. Gaffney, J.E.: Metrics in software quality assurance. In: Proc. of the ACM 1981 Conference, pp. 126–130. ACM, New York (1981)
15. Mantyla, M., Vanhanen, J., Lassenius, C.: A taxonomy and an initial empirical study of bad smells in code. In: Proc. of ICSM 2003. IEEE Computer Society, Los Alamitos (2003)
16. Wake, W.C.: Refactoring Workbook. Addison-Wesley Longman Publishing Co., Inc., Boston (2003)
17. http://www.ptidej.net/research/decor/index_html
18. http://www.marou ane-kessentini/FASE10.zip
19. Raedt, D.: Advances in Inductive Logic Programming, 1st edn. IOS Press, Amsterdam (1996)
20. Erni, K., Lewerentz, C.: Applying design metrics to object-oriented frameworks. In: Proc. IEEE Symp. Software Metrics. IEEE Computer Society Press, Los Alamitos (1996)
21. Alikacem, H., Sahraoui, H.: Détection d'anomalies utilisant un langage de description de règle de qualité, in actes du 12e colloque LMO (2006)
22. O'Keeffe, M., Cinnéide, M.: Search-based refactoring: an empirical study. Journal of Software Maintenance 20(5), 345–364 (2008)
23. Harman, M., Clark, J.A.: Metrics are fitness functions too. In: IEEE METRICS, pp. 58–69. IEEE Computer Society Press, Los Alamitos (2004)
24. Kessentini, M., Sahraoui, H.A., Boukadoum, M.: Model Transformation as an Optimization Problem. In: Busch, C., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (eds.) MODELS 2008. LNCS, vol. 5301, pp. 159–173. Springer, Heidelberg (2008)
25. Kirkpatrick, D.S., Gelatt, Jr., Vecchi, M.P.: Optimization by simulated annealing. Science 220(4598), 671680 (1983)
26. Eberhart, R.C., Shi, Y.: Particle swarm optimization: developments, applications and resources. In: Proc. IEEE Congress on Evolutionary Computation (CEC 2001), pp. 81–86 (2001)