Deviance from Perfection is a Better Criterion than Closeness to Evil when Identifying Risky Code

Marouane Kessentini, Stéphane Vaucher, Houari Sahraoui
DIRO, Université de Montréal, CANADA
{kessentm,vauchers,sahraouh}@iro.umontreal.ca

ABSTRACT

We propose 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. We evaluated our approach by finding potential defects in two open-source systems (Xerces-J and Gantt). We used the library JHotDraw as the code base representing good design/programming practices. In both systems, we found that 90% of the riskiest classes were defects, a precision far superiour to state of the art rule-based approaches.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Maintenance—*Restructuring, reverse engineering, and reengineering*

General Terms

Design

Keywords

Maintenance, design defects, artificial immune systems

1. INTRODUCTION

In order to limit maintenance costs and improve the quality of their software systems, companies try to both enforce good design/development practices and prevent bad practices. As a result, these practices have been studied by professionals and researchers alike with a special attention given to design-level problems.

There has been much research focusing on the study of bad design practices sometimes called defects, antipatterns [3], smells [11], or anomalies [10] in the literature [1]. Although these bad practices are sometimes unavoidable, in most cases, development teams should try to prevent them and remove them from their code base as early as possible. Hence, many fully-automated detection techniques have been proposed [17, 22, 19].

Several problems limit the effectiveness of existing techniques. Indeed, the vast majority of existing work relies on rule-based detection [21, 22]. Different rules identify key symptoms that characterize a defect using combinations of mainly quantitative (metrics), structural, and/or lexical information. Therefore, to identify and remove defects in a system, all possible defects should be known and their symptoms characterized with rules. Moreover, the rules must to be applied equally to any system in any context. This is not reasonable considering the variety of software systems and the difficulty of expressing some types of symptoms. These difficulties explain a large portion of the high false-positive rates mentioned in existing research [19].

In this article, we propose an automated detection approach that is completely different from the state of art. Instead of characterizing each symptom of each possible defect type, we apply the principle of negative selection, the process used by biological immune systems [2] to identify antigens. An immune system does not try to detect specific bacteria and viruses. Rather, it starts by detecting what is abnormal, *i.e.,* what is different from the healthy cells of the body. The more something is different, the more it is considered risky.

We apply the same principle to the detection of design defects by, first, defining what is normal. Normality is defined using a code base containing examples of well designed and implemented software elements. Then, we create a set of detectors that represent different ways that a code can diverge from the good code. Finally, elements of assessed systems that are similar to detectors are considered as risky.

To evaluate our approach, we used classes from the JHotDraw library as our examples of well-designed and implemented code. Two systems, Xerces-J and Gantt, were then analyzed using our approach. Almost all the identified riskiest classes (with levels > 90%) were found in a list of classes tagged as defects (blobs, spaghetti code and functional decomposition) in another project [22].

Our contributions to automation are as follows. First, our technique is fully automatable from the creation of detectors to the evaluation of classes. Second, our technique

[1]In the remainder of this article, we use the generic term *defect* to refer to an occurrence of a bad practice in the code

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ASE'10, September 20–24, 2010, Antwerp, Belgium.
Copyright 2010 ACM 978-1-4503-0116-9/10/09 ...$10.00.

does not require an expert to write rules for every defect type, and adapt them to different systems. Finally, using only standard algorithms to measure similarity, our technique not only outperforms rule-based techniques in terms of precision, but we are also able to find a good mix of defects types. The major limitation of the approach is that we require a code base representing of good design practices. Our results indicate however that JHotdraw seems to be usable and could serve as a starting point for a company wishing to use our approach..

The remainder of this paper is structured as follows. Section 2 is dedicated to the problem statement. In Section 3, we describe the principles of the Artificial Immune System that inspires our approach and the adaptations of these principles to the detection of design defects. Section 4 presents and discusses the validation results. A summary of the related work in defect detection is given in Section 5. We conclude and suggest future research directions in Section 6.

2. PROBLEM STATEMENT

In this section, we describe the problem of defect detection. We start by defining important concepts. Then, we detail the specific problems that are addressed by our approach.

2.1 Basic Concepts

Design defects, also called **design anomalies** refer to design situations that adversely affect the development of a software. 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 with the intent of improving their detection [27] and suggesting improvements paths. The two following types of defects are commonly mentioned. In [11], 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 some refactoring suggestions to remove them. Brown *et al.* [3] define another category of design defects named **anti-patterns**, which includes blob classes, spaghetti code, and cut & paste programming. In both books, the authors focus on describing the symptoms to look for in order to identify specific defects.

Regarding our detection approach, we use the following concepts:

- A **code fragment** represents a software element that is evaluated. This could be a class, method, or package in an object-oriented code. Although our approach could be applied to evaluate any of these entities, in this paper, we use code fragment to refer essentially to a class.

- A **design risk** is a code fragment that is dissimilar (unusual) from known good code. It could be a design defect or simply an unusual design/developpement practice.

- The process of **discovering design defects** consists of finding high-risk code fragments in the system without relying on specific knowledge on the known defect types.

2.2 Problem Statement

Any technique to detect design defect should address/ circumvent many difficulties inherent to the nature of defects. Here is the description of the most important difficulties and how they affect an automation process.

- There is **no exhaustive list** of all possible types of design defects. Although there has been significant work to classify defect types [27, 20, 28], programming practices, paradigms and languages evolve making unrealistic to support the detection of all possible defect types. Furthermore, there might be company or application-specific (bad) design practices.

- For those design defects that are documented, there is *no consensual definition of symptom detections*. Defects are generally described using natural language and their detection relies on the interpretation of the developers. This limits the automation of the detection.

- The majority of detection methods do not provide **an efficient manner to guide the manual inspection of the candidate list**. Potential defects are generally not listed in an order that helps developers addressing in priority the most severe ones. There is little work, such as the one of Khomh *et al* [17], where probabilities are used to order the results.

3. AIS-BASED DETECTION ALGORITHM

Our approach is based on the metaphor of biological immune systems. In this section, we present the principles of this metaphor, and our adaptation to the problem of detecting design defects.

3.1 Principles of Artificial Immune Systems

The role of a biological immune system (IS) is to protect its host organism against foreign elements such as pathogens (*e.g.,* bacteria and viruses) and/or malfunctioning cells (*e.g.,* cancerous cells). This is performed following three phases: (1) *discovery*, (2) *identification*, and (3) *elimination* of foreign elements. *Discovery* is the phase that interests us in particular for our work. Therefore, we explain its principle in the following paragraphs.

There is no central organ that fully controls the IS. Instead, *detectors* wander in the body searching for harmful elements. Any element that can be recognised by the immune system is called an *antigen*. The cells that originally belong to our body and are harmless to its functioning are termed *self* (for self antigens) while the disease causing elements are named *nonself* (for nonself antigens). The IS classifies cells that are present in the body as self and nonself cells.

The immune system produces a large number of randomly created detectors. A negative selection mechanism eliminates detectors that match cells present in a protected environment (bone marrow and the thymus) where only self cells are assumed to be present. Non-eliminated ones become naive detectors; they die after some time unless they match an element assumed to be a pathogen. Detectors that do match a pathogen are quickly cloned; this is used to accelerate the response to future attacks. Since the clones are not exact replicates (they are mutated), this provides a more

focused response to pathogens. This process, called *affinity maturation*, provides an efficient adaptation to a changing non-self environment. A detailed presentation of the biological immune system can be found in books such as [16]).

The success of immune systems at keeping a living organism healthy inspired the emergence of artificial immune systems (AIS) as a generic solution to problems in several domains, such as scheduling, computer security, optimization, or robotics [7]. AIS can be adapted to the problem of defect detection. The following mappings shows the similarity between our problem and the AIS concepts.

- **Body**: the evaluated system, more precisely, its code;
- **Detector**: an artificial code fragment that is very different from a well-designed code base;
- **Self Cells**: well-designed code fragments in the system to evaluate (without design defects);
- **Non-Self Cells**: code fragments in the system to evaluate that present a risk of being design defects;
- **Affinity**: the similarity between detectors and code fragments to evaluate.

3.2 Approach Overview

Figure 1 gives an overview of our approach. The detection process has two main steps: detector generation and risk estimation. Detectors are generated from a collection of code fragments coming from one or more well-designed systems. These code fragments define the reference of what is considered normal code. The generation process of detectors is performed using a heuristic search that maximizes on one hand, the distance between detectors and normal code and, on the other hand, the distance between the detectors themselves. The same set of detectors could be used to evaluate many systems, and it could be updated as the *normal* code base or development practices evolves.

The second step of the detection process consists of comparing the code to evaluate to the detectors. A code fragment that exhibits a similarity with a detector is considered as a risky element. The higher the similarity, the more a code fragment is considered risky. Both the detector generation and risk estimation steps use similarity scores. Before detailing the two steps, we first describe the similarity functions used in this work.

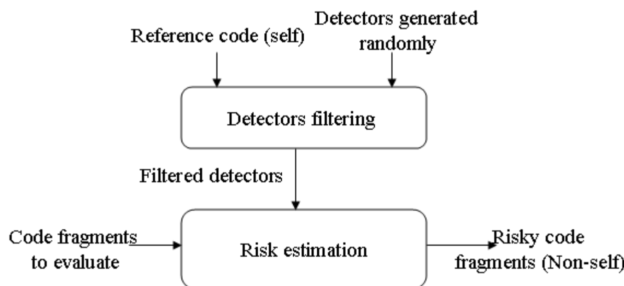


Figure 1: Approach Overview

3.3 Similarity between Code Fragments

To calculate the similarity between two code fragments, we adapted the Needleman-Wunsch alignment [23] algorithm to our context. It is a dynamic programing algorithm used in bioinformatics to efficiently find similar regions between two sequences of DNA, RNA or protein [4]. An example of the algorithm is presented in Figure 2.

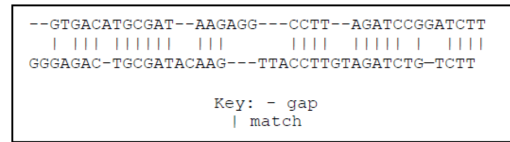


Figure 2: Global alignment of two strings

As we are manipulating code elements and not sequences (strings), we represent these elements by sets of predicates. Each predicate type corresponds to a construct type of an object-oriented system: Class (C), attribute (A) , method (M), parameter (P), generalization (G), and method invocation relationship between classes (R). For example, in Figure 3, the sequence of predicates CGAAMPPM corresponds to a class with a generalization link, containing two attributes and two methods. The first method has two parameters.

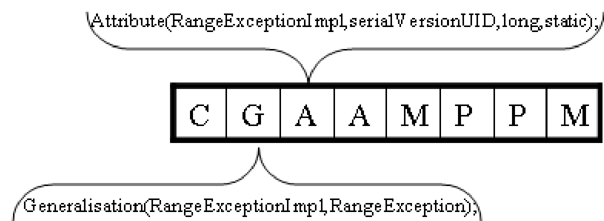


Figure 3: Encoding

Predicates include details about the associated constructs (visibility, types, *etc.*). These details (thereafter called parameters) determine ways a code fragment can deviate from a notion of normality. The example of Figure 3 is a representation of class *RangeExceptionImpl* from Xerces-J. The corresponding predicate set, extracted using our inhouse eclipse plugin is as follows:

```
Class(RangeExceptionImpl,public);
Generalisation(RangeExceptionImpl,RangeException);
Attribute(RangeExceptionImpl,serialVersionUID,long,static);
Attribute(RangeExceptionImpl,serialImplUID,short,static);
Method(RangeExceptionImpl,RangeExceptionImpl,void,Y,public);
Parameter(RangeExceptionImpl,RangeExceptionImpl,code,short);
Parameter(RangeExceptionImpl,RangeExceptionImpl,message,
String);
Method(RangeExceptionImpl,implSerial,void,Y,private);
```

As described below, the Needleman-Wunsch global alignment algorithm [23] is described recursively. . When aligning two sequences (a_1,...,a_n) and (b_1,...,b_m). Each position $s_{i,j}$ in the matrix corresponds to the best score of alignment considering the previously aligned elements of the sequences. The algorithm can introduce gaps (represented by "-") to improve the matching of subsequences.

$$
s_{i,j} = Max \begin{cases} s_{i-1,j} - g & \text{// insert gap for } b_j \\ s_{i,j-1} - g & \text{// insert gap for } a_i \\ s_{i-1,j-1} + sim_{i,j} & \text{// match} \end{cases}
$$

where $s_{i,0} = g * i$ and $s_{0,j} = g * j$

At any given point, algorithm considers two possibilities. First, it considers the case when a gap should be inserted. When a gap is inserted for either a or b, the algorithm applies a penalty of g. Second, it tries to match predicates. The similarity function $sim_{i,j}$ returns the reward or cost of matching a_i to b_j. The final similarity is contained in $s_{n,m}$.

Our adaptation of the algorithm is straightforward. We define the gap penalty g and the similary function to match individual predicates (sim). We do not seek perfect matches in terms of number of predicates. A class with 4 is not necessarily different from one with 6 methods if the methods are similar. To eliminate the sensitivity of the algorithm to size, we thus set the gap penalty to 0.

We define a predicate-specific function to measure the similarity. First, if the types differ, the similarity is 0. As we manipulate sequences of complex predicates and not strings, $sim_{i,j}$ is defined as a predicate-matching function, PM_{ij}. PM_{ij}, measures the similarity in terms of the elements of the predicates associated to a_i and b_j. This similarity is the ratio of common parameters in both predicates.

$$
PM_{ij} = \frac{\forall_{p \in a_i, q \in b_i} \cap (p, q)}{max(|a_i|, |b_j|)|}
$$

where a_i and b_j are treated as sets of predicates. The equivalence between predicate parameters depends on each type of parameter. For visibility and element types, it means equality. Specific names are not considered. Instead, they are used to indicate a common reference by other predicates. For example, if a class defines an attribute and its related getter method. They will both share the same class name.

To illustrate an example for the local alignment algorithm, let us consider the class *RangeExceptionImpl*, described previously, as a code fragment C_{32} and *Options*,C_{152} as a second code fragment to compare with C_{32}. The code fragments are sequentially numbered. C_{152} is defined as follows :

```
Class(Options,public);
Method(Options,isFractionalMetrics,boolean,N,public);
Method(Options,isTextAntialiased,boolean,N,private);
Parameter(Options,isTextAntialiased,id,String);
Relation(AbstractFigure;getFontRenderContext;isFractionalMetrics,
Options,N);
```

According to the coding mentioned previously, the predicate sequence for C_{32} is CGAAMPPM and one of C_{152} is CMMPR. The alignment algorithm finds the best alignment sequence as shown in Figure 4.

There are three matched predicates between C_{32} and C_{152}: one class, one method, and one method parameter. If we consider the second matched predicates $p1_5 =$ `Method(RangeExceptionImpl,RangeExceptionImpl,void,Y,public)` from C_{32} and $p2_2 =$ `Method(Options,isFractionalMetrics,boolean, N,public);` from C_{152}. The predicates have two common parameters out of a possible five. The resulting similarity is consequently 40%. We normalize this absolute similarity

		C	*G*	*A*	*A*	*M*	*P*	*P*	*M*
	0	0	0	0	0	0	0	0	0
C	0	**1**	0	0	0	0	0	0	0
M	0	**1**	**1**	1	1	1.6	1.6	1.6	1.6
M	0	1	**1**	**1**	**1**	1.6	1.6	1.6	1.6
P	0	1	1	**1**	**1**	**1.6**	2	2.6	2.6
R	0	1	1	1	**1**	**1.6**	**2**	**2.6**	**2.6**

```
C32:   C G A A M  -  P P M -
       |          |      |
C152:  C  -  - -  M  M - P - R
```

Figure 4: Best alignment sequence between C_{32} and C_{152}

measure, $s_{n,m}$, by the maximum number of predicates to produce our similarity measure:

$$
Sim(A, B) = \frac{s_{n,m}}{max(n, m)} \quad (1)
$$

3.4 Detectors Generation

This section describes how a set of detectors is produced starting from the reference code. The generation, inspired by the work of Gonzalez and Dasgupta [13], follows a genetic algorithm [12]. The idea is to produce a set of detectors that best covers the possible deviations from the reference code. As the set of possible deviations is very large, its coverage may require a huge number of detectors, which is infeasible in practice. For example, pure random generation was shown to be infeasible in [15] for performance reasons. We therefore consider the detector generation as a search problem. A generation algorithm should seek to optimize the following two objectives:

- Maximize the generality of the detector to cover the non-self by minimizing the similarity with the self;
- Minimize the overlap (similarity) between detectors.

These two objectives define the cost function that evaluates the quality of a solution and, then guides the search. The cost of a solution D (set of detectors) is evaluated as the average costs of the included detectors. We derive the cost of a detector d_i as a weighted average between the scores of respectively, the lack of generality and the overlap. Formally,

$$
cost(d_i) = \frac{LG(d_i) + O(d_i)}{2} \quad (2)
$$

Here, we give equal weight to both scores. The lack of generality is measured by a matching score $LG(d_i)$ between the predicate sequence of a detector d_i and those of all the classes s_j in the reference code (call it S). It is defined as the average value of the alignment scores $Sim(d_i, s_j)$ between d_i and classes s_j in S. Formally,

$$
LG_{d_i} = \frac{\sum_{s_j \in S} Sim(d_i, s_j)}{|S|}
$$

Similarly, the overlap O_i, is measured by the average value of the individual $Sim(d_i, d_j)$ between the detector d_i and all the other detectors d_j in the solution D. Formally,

$$O_{d_i} = 1 - \frac{\sum_{d_j, j \neq i} Sim(d_i, d_j)}{|D|}$$

The cost function defined above is used in our genetic-based search algorithm. Genetic algorithms (GA) implement the principle of natural selection [12]. Roughly speaking, a GA is an iterative procedure that generates a population of individuals from the previous generation using two operators: crossover and mutation. Individuals having a high fitness have higher chances to reproduce themselves (by crossover), which improves the global quality of the population. To avoid falling in local optima, mutation is used to randomly change individuals. Individuals are represented by chromosomes containing a set of genes.

For the particular case of detector generation, we reuse the predicate sequences as chromosomes. Each predicate represents a gene. We start by randomly generating an initial population of detectors. The size of this population is a parameter that will be discussed later in Section 4. This size is maintained constant during the evolution. The fitness of each detector is evaluated by the inverse function of cost. The fitness determines the probability of being selected for the crossover. This process is called a wheel-selection strategy [12].

In fact, for each crossover, two detectors are selected by applying twice the wheel selection. Even though detector 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 p_1 and p_2. It is defined as follows:

- A random position k, is selected in the predicate sequences.
- The first k elements of p_1 become the first k elements of o_1. Similarly, he first k elements of p_2 become the first k elements of o_2.
- The remaining elements of, respectively, p_1 and p_2 are added as second parts of, respectively, o_2 and o_1.

For instance, if k = 3 and p1 = CAMMPPP and p2 = CMPRMPP, then o1 = CAMRMPP and o2 = CMPMPPP.

The mutation operator operator consists of randomly changing a predicate.

3.5 Risk Estimation

The second step of our defect discovery is the assessment of risk for the different code fragments evaluated. These are also represented by predicate sequences. Each sequence is compared using the alignment algorithm to the detectors obtained in the previous step. The risk of being a defect, associated to a code fragment e_i is defined as the average value of the alignment scores $Sim(e_i, d_j)$ obtained by comparing e_i to respectively all the detectors of a set D. Formally,

$$risk_{e_i} = \frac{\sum_{d_j \in D} Sim_l(e_i, d_j)}{|D|}$$

The code fragments can then be ranked according to their risks to be inspected by the maintainers.

4. EVALUATION

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

4.1 Goals and Objectives

The goal of the study is to evaluate the efficiency of our AIS approach for the discovery 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 discover 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 of our approach. We ranked classes in order of decreasing risk and compared results to produced by a rule-based strategy [22]. To answer RQ2, we investigated the type of defects that were found.

4.2 System Studied

We used three open-source Java projects to perform our experiments: GanttProject v1.10.2, Xerces v2.7.0, and JHotdraw v7.1. Table 1 summarizes facts on these programs. GanttProject[2] is a tool for creating project schedules by means of Gantt charts and resource-load charts. GanttProject enables breaking down projects into tasks and establishing dependencies between these tasks. Xerces[3] is a family of software packages for parsing and manipulating XML. It implements a number of standard APIs for XML parsing. JHotdraw v7.1[4] is a framework used to build graphic editors. It was first built as an example of the use of design patterns. JHotdraw was chosen because it contains very few known design defects. In fact, previous work [6] could not find any *Blob* defects. In our experiments, we used all of the classes in JHotdraw as our example set of good code. We chose the Xerces 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 [22], 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[5] 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

[2]http://ganttproject.biz/index.php
[3]http://xerces.apache.org/
[4]http://jhotdraw.org
[5]http://www.ptidej.net/research/decor/index_html

approach to locate classes that corresponded to instances of these antipatterns.

Systems	♯ classes	♯ Predicates	KLOC
GanttProjectv1.10.2	245	16640	31
Xerces v2.7.0	991	67810	240
JHotdraw v7.1	471	40354	91

Table 1: Program statistics

4.3 Experimental Setting

For our experiment, we randomly generated 100 detectors for JHotDraw (about a quarter of the number of examples) with a maximum size of 256 characters. The same set of detectors was used on both Xerces and Gantt. The obtained results[6] were compared to those of DECOR [22], a state of the art rule-based detection technique. For every antipattern in Xerces and Gantt, they published the number of antipatterns detected, the number of true positives, and the precision (ratio of true positives over the number detected). Our comparison is consequently done using precision. We would have liked to consider recall, but they did not publish clean, complete data describing all existing antipatterns. We therefore could not perform systematic comparisons of the recall of our approach. Instead, we discuss the proportion of "known antipatterns" detected.

4.4 Results

Tables 2 and 3 summarize our findings. Each class is presented with its risk, its size, and the associated defect types. We only presented classes with a risk level of >= 70%; this corresponds to about 5% of the classes in the system. For Gantt, our precision over the top 20 classes is 95% with the eight riskiest classes being true positives. DECOR on the other hand has a combined precision of 59% for its detection on the same set of antipatterns. For Xerces, our precision is of 90% with the top 30 classes correctly identified as defects. For the same dataset, DECOR had a precision of 67%. 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, we had an almost equal distibution of each antipattern (14 SCs, 13 Blobs, and 13 FDs). On Gantt, the distribution is not as balanced. This is principally due to the number of actual antipatterns in the system. We found all four known Blobs and all 11 SCs in the system. We found 6/17 FDs, two more than DECOR.

Having a relatively good distribution of antipatterns is useful for a quality engineer as he can focus on the notion of riskiest classes regardless of the type. Furthermore, since the results are ranked he can efficiently use his time unlike DECOR.

This ability to identify different types of antipatterns underlines a key strength to our approach: the similarity function is able to abstract out the importance of size. Most other tools and techniques rely heavily on the notion of size to detect defects. This is reasonable considering that some antipatterns like the Blob are associated to a notion of size.

[6] http://www.iro.umontreal.ca/~sahraouh/papers/ASE2010/

Class	Risk	S.C.	Blob	F.D.
GanttOptions	0.96	✓		
GanttTree	0.96		✓	
GregorianTimeUnitStack	0.93			✓
GanttDialogPerson	0.92	✓		
CSVSettingsPanel	0.9	✓		
GanttProject	0.9	✓	✓	
GanttTaskPropertiesBean	0.9	✓		
NewProjectWizard	0.87			✓
TimeUnitGraph	0.87			
ResourceLoadGraphicArea	0.85	✓	✓	
GanttCSVExport	0.82	✓		
GanttGraphicArea	0.82	✓	✓	
FindPossibleDependeesAlgo...	0.82			✓
GanttXFIGSaver	0.81	✓		
GanttApplet	0.79			✓
GraphicPrimitiveContainer	0.75	✓		
Shape	0.75			
GanttXMLSaver	0.75	✓		
RecalculateTaskCompletion...	0.71			✓
TaskHierarchyManagerImpl	0.71			✓
Precision	95%			

Table 2: Results for Gantt

For antipatterns like FDs however, the notion of size is irrelevant and this makes this type of anomaly hard to detect using structural information. 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.

4.5 Discussion

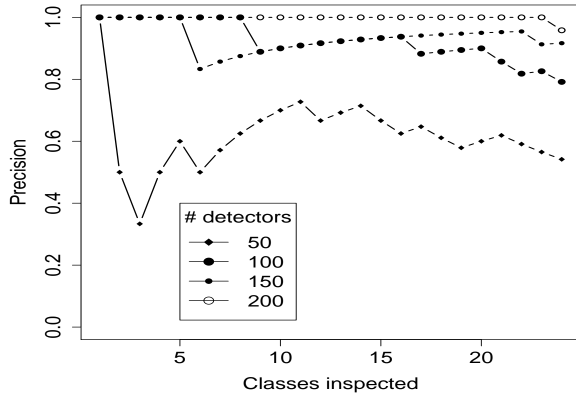
In this section, we discuss different issues concerning the detection of design risks.

Number of Detectors.

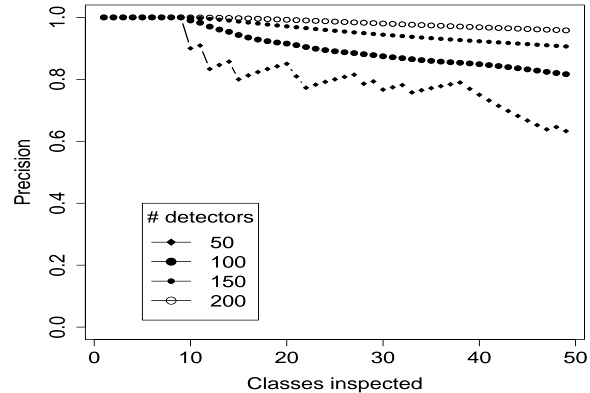
An important factor to our detection technique is the number of detectors generated. In Figure 5, we present the precision of our approach when varying the number of detectors (N_d) with $N_d = \{50,100,150,200\}$. The figure shows that the performance of our approach improves as we consider more detectors. When we use 200 detectors (50% of the total number of cases in JHotDraw), our performance is over 95% for both systems. Our technique requires the comparison of every class to every detector, this improved performance is at a negligible cost in terms of execution time. Indeed, the execution time for applying the detection on each system varies between 2 minutes for 50 detectors and 15 minutes for 200.

Variability in Detector Generation.

Another issue is our selection of interesting detectors. The detection results might vary depending on the detectors 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 detector generation. When we consider results up to 70% of risk, we observed an average precision of 92% for Gantt and 91% for Xerces. Furthermore, we found that the majority of defects detected are found in every execution (54% and 60% respectively for Gantt and Xerces). These unanimously detected defects were systematically the riskiest classes in every execution: in Gantt, the top 10 classes were common to all executions.



(a) Gantt



(b) Xerces

Figure 5: Effect of the number of detectors on detection precision vs. #classes inspected

In Xerces, there was only one non-unanimous class in the top ten classes returned which was a false-positive. The average rank for a class detected by a single execution was 18 and 34 for Gantt and Xerces respectively. We consequently believe that since the variability comes from the least risky classes, and that our technique is stable.

Metric-based Detection vs. Similarity-based Detection.

Our approach is significantly different from existing work that are rule-based. A key problem with these approaches is that these rules simplify the different notions that are useful for the detection of certain antipatterns. In particular, to detect blobs the notion of size is important. Most size metrics are highly correlated with one another, and the best measure of size can depend on the system itself. Our use of predicates allows for complex structures to be detected.

For example, we correctly detected TaskHierarchyManagerImpl in Gantt. It holds a reference to the root of the hierarchy, and controls creations of new children to the root.

```
public class TaskHierarchyManagerImpl {
    private TaskHierarchyItem myRootItem =
        new TaskHierarchyItem(null ,null);
    public TaskHierarchyItem getRootItem() {
        return myRootItem;
    }
    public TaskHierarchyItem createItem(Task task) {
        TaskHierarchyItem result =
           new TaskHierarchyItem(task, myRootItem);
        return result;
    }}
public class TaskManagerImpl implements TaskManager { ...
    private final TaskHierarchyManagerImpl myHierarchyManager
        = new TaskHierarchyManagerImpl();
    public TaskHierarchyManagerImpl getHierarchyManager() {
        return myHierarchyManager;
    }...}
```

It is detected for three reasons. First, it declares one attribute type (TaskHierarchyItem) on which it never invokes any methods. Second, it is used in a similar manner by TaskManagerImpl. Finally, apart from creating objects, it never uses any methods. It is consequently a datastructure. These types of relationships are hard to detect using metrics. On the other hand, our technique produced a detector that was almost a complete match (except the final parameter):

```
Attribute(X,aaaa,AA,N,private); # myRootItem
Attribute(X,aa,X,N,private);    # myHierarchyManager
Class(X,N,N,public);            # TaskHierarchyManagerImpl
Method(X,z,X,Y,N,N,public);     # createItem
Method(X,zzzz,AA,N,N,N,public); # getRootItem
Method(X,zxzzz,X,N,N,N,public); # getHierarchyManager
Parameter(X,z,zuwe,gsfg,declaration); # task
Parameter(X,z,xuqye,fzfgg,local); # result
Parameter(X,zzzz,jdajg,gffgs,declaration); # Match error
```

DECOR also successfully identified this class. However, it did so, not because of metrics, but because the name of the class contains the term *Manager*.

Building an Example Data Set.

The reliability of the proposed approach requires an example set of good code. It can be argued that constituting such a set might require more work than identifying and adapting rules. In our study, we showed that by using JHotdraw directly, without any adaptation, the technique can be used out of the box and this will produce good detection results for the detection of antipatterns for the two systems studied. The performance of this detection (in terms of precision) was superiour to that of DECOR. In an industrial setting, we could expect a company to start with JHotDraw, and gradually migrate its set of good 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.

5. 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.

Class	Risk	S.C.	Blob	F.D.
DFAContentModel	0.97	✓		
XSFacets	0.96			✓
XMLSerializer	0.96	✓		
XMLVersionDetector	0.96			✓
XML11EntityScanner	0.93	✓		
XSDHandler	0.92		✓	✓
Token	0.91	✓		
XMLEntityManager	0.91		✓	
XSDAbstractTraverser	0.91			✓
XML11DTDValidator	0.91			✓
DOMNormalizer	0.91		✓	
XMLNSDTDValidator	0.88			✓
ParserConfigurationSettings	0.88			✓
SAXParser	0.85			✓
DTDGrammar	0.84		✓	
XML11NonValidatingConfiguration	0.84		✓	
XMLDTDValidator	0.84		✓	
XMLEntityScanner	0.84	✓		
XSAttributeGroupDecl	0.82	✓		
AbstractDOMParser	0.81	✓		
SchemaDOM	0.81			✓
XML11DTDConfiguration	0.81		✓	
XSDAttributeTraverser	0.81	✓		
ObjectFactory	0.8	✓		
XIncludeHandler	0.8		✓	
XSDFACM	0.78	✓		
NonValidatingConfiguration	0.78		✓	
XMLSchemaValidator	0.78		✓	
DTDConfiguration	0.77		✓	
CoreDocumentImpl	0.77	✓	✓	
XSAttributeChecker	0.77		✓	
CMNodeFactory	0.77			
RegexParser	0.75			✓
TimeDV	0.75			
XML11Configuration	0.74	✓		
XMLFilterImpl	0.73			
DOMSerializerImpl	0.71	✓		
XSFacets	0.71			✓
BaseMarkupSerializer	0.71			
ElementSchemePointer	0.71			✓
XMLParser	0.7			✓
XPathMatcher	0.7	✓		
Precision	90%			

Table 3: Results for Xerces

In first category, Marinescu [21] 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.* [9] 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 contribution is the difficulty to define threshold values for metrics in the rules. To circumvent this problem, Alikacem *et al.* [1] express defect detection as fuzzy rules with fuzzy label for metrics, *e.g., small, medium, large.* When evaluating the rules, actual metric value 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 foudation of the work of Moha *et al.* [22]. 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. Another limitation of DECOR is that all the detected defect candidate are listed without any rank that help the maintainers checking/addressing in priority the most severe ones.

Khomh *et al.* [17] 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 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 he global quality. For example, in [24], 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 [14]. 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.

The high rate of false positives generated by the automatic approaches encouraged other teams to explore semi-automatic 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. Kothari *et al.* [18] present a pattern-based framework for developing tool support to detect software anomalies by representing potentials defects with different colors. Later, Dhambri *et al.* [8] 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, the human intervention is needed for the inspection of the candidate only. This inspection is made easier because, the candidates are ranked by risk, and also because, by analysing the most similar detectors, it is possible to identify what part of the element was problematic.

The work that is closest to ours is by Catal and Diri [5]. The authors use a machine-learning version of AIS called an Artificial Immune Recognition System (AIRS) to learn a prediction model for defect-prone modules. The AIRS used was a generic package implemented in the machine-learning package Weka. This package cannot handle complex structures like predicates and does not implement the negative selection algorithm.

6. CONCLUSION

In this article, we presented a new approach to 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 a detection. Instead, all we need is a clear notion of what is good. What significantly diverges is often a defect. Interestingly enough, our study shows that our technique outperforms an DECOR [22], a state of the art, rule-based approach on its test corpus.

By ignoring the detection of specific defect types, we avoid problems with existing detection techniques. First, the detection of most defect is difficult to automate because their definitions are expressed informally. Second, even with a precise definition, some symptoms are context-specific and might or not be useful for a given system. There is consequently a non-negligeable effort to test and adapt a detection process to another system. Finally, by presenting all defects, regardless of types in order of risk, a development team can focus on the most urgent problems first.

This technique was tested on two open-source systems and the results were promising. The discovery process uncovered different types of design defects was more efficiently than DECOR. In fact, for Gantt, our precision is 95% with the eight riskiest classes being true positives. DECOR on the other hand has a combined precision of 59% for its detection of the same set of antipatterns. For Xerces, our precision is of 90% with the top 30 classes correctly identified as defects. For the same dataset, DECOR had a precision of 67%. Furthermore, as DECOR needed an expert to define rules, our results were achieved without any expert knowledge, relying only on the good structure of JHotdraw to guide the detection process.

In this work, we only looked at the first step of an immune systems: the discovery of risk. As part of our future work, we plan to explore the other two steps: identification and correction of detected design defects (refactoring). Furthermore, we need to extend our reference code base with other well-designed code in order to take into consideration different programming contexts. Specifically, we plan on:

- Adapting the AIS metaphor to *identify* discovered defects using immune memory and danger theory [26].
- Adapting the colonal selection algorithm [25] to find the best immune response that correspond the optimal refactorings sequence to apply.
- Using our approach for defect prediction using the estimation risk score.

Acknowledgment

This work has been partly funded by the Natural Sciences and Engineering Research Council of Canada (NSERC) and the Tunisian Ministry of Higher Education and Scientific Research.

7. REFERENCES

[1] H. Alikacem and H. Sahraoui. Détection d'anomalies utilisant un langage de description de règle de qualité. In LMO, editor, *actes du 12e colloque LMO*, 2006.

[2] F. Azuaje. Review of "artificial immune systems: a new computational intelligence approach" by l.n. de castro and j. timmis (eds) springer, london, 2002. *Neural Netw.*, 16(8):1229–1229, 2003.

[3] 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*. John Wiley and Sons, 1st edition, March 1998.

[4] M. Brudno. *Algorithms for comparison of dna sequences*. PhD thesis, Stanford, CA, USA, 2004. Adviser-Batzoglou, Serafim.

[5] C. Catal and B. Diri. Software defect prediction using artificial immune recognition system. In *SE'07: Proceedings of the 25th conference on IASTED International Multi-Conference*, pages 285–290, Anaheim, CA, USA, 2007. ACTA Press.

[6] I. G. Czibula and G. Czibula. Clustering based automatic refactorings identification. In *SYNASC '08: Proceedings of the 2008 10th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, pages 253–256, Washington, DC, USA, 2008. IEEE Computer Society.

[7] D. Dasgupta, Z. Ji, and F. Gonzalez. Artificial immune system (ais) research in the last five years. In *IEEE Congress on Evolutionary Computation (1)*, pages 123–130. IEEE, 2003.

[8] K. Dhambri, H. A. Sahraoui, and P. Poulin. Visual detection of design anomalies. In *CSMR*, pages 279–283. IEEE, 2008.

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

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

[11] M. Fowler. *Refactoring – Improving the Design of Existing Code*. Addison-Wesley, 1st edition, June 1999.

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

[13] F. A. González and D. Dasgupta. Anomaly detection using real-valued negative selection. *Genetic Programming and Evolvable Machines*, 4(4):383–403, 2003.

[14] M. Harman and J. A. Clark. Metrics are fitness functions too. In *IEEE METRICS*, pages 58–69. IEEE Computer Society, 2004.

[15] H. Hou and G. Dozier. An evaluation of negative selection algorithm with constraint-based detectors. In *ACM-SE 44: Proceedings of the 44th annual Southeast regional conference*, pages 134–139, New York, NY, USA, 2006. ACM.

[16] K. J. *Immunology*. by Richard A. Goldsby, Thomas J. Kindt, Barbara A. Osborne, W.H, 5th edition . edition, 2002.

[17] F. Khomh, S. Vaucher, Y.-G. Guéhéneuc, and H. Sahraoui. A Bayesian Approach for the Detection of Code and Design Smells. In D.-H. Bae and B. Choi, editors, *Proceedings of the 9th International Conference on Quality Software*. IEEE Computer Society Press, August 2009.

[18] S. C. Kothari, L. Bishop, J. Sauceda, and G. Daugherty. A pattern-based framework for software anomaly detection. *Software Quality Journal*, 12(2):99–120, June 2004.

[19] H. Liu, L. Yang, Z. Niu, Z. Ma, and W. Shao. Facilitating software refactoring with appropriate resolution order of bad smells. In *ESEC/FSE '09: Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 265–268, New York, NY, USA, 2009. ACM.

[20] M. Mäntylä, J. Vanhanen, and C. Lassenius. A taxonomy and an initial empirical study of bad smells in code. In *ICSM '03: Proceedings of the International Conference on Software Maintenance*, page 381, Washington, DC, USA, 2003. IEEE Computer Society.

[21] R. Marinescu. Detection strategies: Metrics-based rules for detecting design flaws. In *Proceedings of the International Conference on Software Maintenance*, pages 350–359, 2004.

[22] 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.

[23] L. Nanni and A. Lumini. Generalized needleman-wunsch algorithm for the recognition of t-cell epitopes. *Expert Syst. Appl.*, 35(3):1463–1467, 2008.

[24] M. O'Keeffe and M. . Cinnéide. Search-based refactoring: an empirical study. *Journal of Software Maintenance*, 20(5):345–364, 2008.

[25] W. Pang and G. M. Coghill. Modified clonal selection algorithm for learning qualitative compartmental models of metabolic systems. In *GECCO '07: Proceedings of the 2007 GECCO conference companion on Genetic and evolutionary computation*, pages 2887–2894, New York, NY, USA, 2007. ACM.

[26] S. Rawat and A. Saxena. Danger theory based syn flood attack detection in autonomic network. In *SIN '09: Proceedings of the 2nd international conference on Security of information and networks*, pages 213–218, New York, NY, USA, 2009. ACM.

[27] A. J. Riel. *Object-Oriented Design Heuristics.* Addison-Wesley, 1996.

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