2010 Seventh International Conference on the Quality of Information and Communications Technology

IDS: An Immune-inspired Approach for the Detection of Software Design Smells

Salima Hassaine*, Foutse Khomh*, Yann-Gaël Guéhéneuc†, and Sylvie Hamel*
*DIRO, Université de Montréal, Québec, Canada
Email: {hassaisa,foutsekh,hamelsyl}@iro.umontreal.ca
†Ptidej Team, DGIGL, École Polytechnique de Montréal, Québec, Canada
Email: {yann-gael.gueheneuc}@polymtl.ca

***Abstract*—We propose a parallel between object-oriented system designs and living creatures. We suggest that, like any living creature, system designs are subject to diseases, which are design smells (code smells and antipatterns). Design smells are conjectured in the literature to impact the quality and life of systems and, therefore, their detection has drawn the attention of both researchers and practitioners with various approaches. With our parallel, we propose a novel approach built on models of the immune system responses to pathogenic material. We show that our approach can detect more than one smell at a time. We build and test our approach on GanttProject v1.10.2 and Xerces v2.7.0, for which manually-validated and publicly-available smells exist. The results show a significant improvement in detection time, precision, and recall, in comparison to the state–of–the–art approaches.**

***Index Terms*—System design; Reverse engineering; Code smells; Antipatterns; Artificial Immune Systems.**

I. Introduction

Design smells [1] and antipatterns [2], collectively called in the following smells, are poor practices to recurring software design and implementation problems. They occur generally in object-oriented systems when developers lack knowledge and–or experience in solving a design problem or applying some patterns. They are conjectured to have a negative impact on the quality and life-time of systems [1], [2]. Consequently, their detection has received attention from both researchers and practitioners with approaches ranging from manual inspections to rule-based detection algorithms.

To the best of our knowledge, all previous approaches require expert's knowledge and interpretation of the smells for their implementation. They focus on detecting one smell at a time, while some smells share similar characteristics, and exclude classes that are not identical to the smell (given some thresholds). Yet, in the course of our experiments with various detection approaches (based on rules [3], Bayesian Beliefs Networks [4], and B-Splines [5]), we noticed that:

- Several smells have similarities. For example, the Blob and Spaghetti Code antipatterns both describe classes that are too large and too complex, the Blob further describing that these classes should relate to data classes.
- Classes similar but not equal to some design smell are also of interest to developers and quality assurance personnel because they could, in the future, emerge as smells themselves in the "submarine" effect [5].

Moreover, previous approaches have somewhat limited performances in time, precision, and recall. They are also somewhat limited in the interpretation of the detected occurrences of the smells, being either too restrictive or to lax.

In this paper, we present an approach to systematically detect classes whose characteristics violate some established design rules; rules inferred from sets of instances (*i.e.*, manually-validated occurrences) of smells reported in the literature and freely-available [3], [4]. Our approach detects smells in general: although we train our approach on only three kinds of smells, it can detect any number and any kind of smells specified during the training. Moreover, it reports classes similar but not identical to the smells, which are of interest to developers and quality-assurance personnel.

Our approach stems from a parallel between object-oriented software systems and living bodies, which constantly fight invading pathogens, such as viruses, bacteria, and so on, through their immune system defense mechanisms. A natural immune system is able to protect the body by identifying, learning from, and defending against invading pathogens. It recognises pathogens after having fought the disease once or by the use of vaccines. Vaccines work by stimulating the immune system using small amounts of disactivated, disease-causing organisms. They cause the immune system to produce antibodies matching the pathogens. Antibodies react concretely to the presence of antigens carried by pathogens. Once antibodies are developed, the immune system is able to respond quickly to the infection of a similar or identical disease-causing organism entering the body, *i.e.*, pathogens carrying similar or identical antigens.

A useful parallel can be drawn from the natural immune system: a system design is comparable to a body, we wish to protect it from pathogens, such as design smells. Design smell detection approaches are defense mechanisms of the system design. A design smell is a pathogen. A "vaccine" could be build using instances of some smells, from which the system design should be protected. Occurrences of a smell are classes with characteristics similar or identical to the smell, *i.e.*, cells contaminated by some pathogen. Antigen that should trigger a response of the defense mechanism can be any characteristics of classes, *e.g.*, metrics, binary class relationships, and so on.

Like pathogens, smells come in a variety of forms with some smells being only slightly different from others. A

978-0-7695-4241-6/10 $26.00 © 2010 IEEE
DOI 10.1109/QUATIC.2010.61

natural immune system can handle such similar pathogens with good precision. This good precision is essential for the body and have inspired a family of classification algorithms name Artificial Immune Systems (AIS) algorithms. Oda and White commented that "if the immune system were inaccurate, the lifespan of the average human would be much shorter as the system would mistakenly attack vital cells or fail to attack viruses and other dangerous pathogens" [6]. Therefore, an AIS-based approach could potentially overcome the limitations of previous approaches regarding the detection of similar but not identical smells as well as the performance in time, precision, and recall of current state-of-the-art approaches. We propose a novel detection approach, called IDS (Immune-based Detection Strategy), based on AIS.

In this paper, we apply IDS to detect three smells and show that its performance in time, precision, and recall are comparable or superior to that of previous approaches. We show that IDS recognise similar but not identical smells using a unique data set. We also discuss IDS other advantages: generalisation, parameter stability, adaptability, portability across systems, simplicity, self-regulation, and performance.

The paper is organised as follows. Section II summarises and discusses previous work. Section III describes our approach. Section IV provides details on the design of the experiments carried out to evaluate our approach. Section V reports and discusses the results. Section VI concludes and suggests future work.

II. Related Work

Webster [7] wrote the first book on antipatterns in object-oriented development; his contribution covers conceptual, political, coding, and quality-assurance problems. Riel [8] defined 61 heuristics characterising good object-oriented programming to assess software quality manually and improve design and implementation. Fowler [1] defined 22 code smells, suggesting where developers should apply refactorings. Mäntylä [9] and Wake [10] proposed classifications of code smells. Brown *et al.* [2] described 40 antipatterns, including the well-known Blob.

These books provide in-depth views on heuristics, code smells, and antipatterns aimed at a wide academic and industrial audience. We build upon this work to propose an approach to characterise design smells and identify classes with similar characteristics. We use the term smell to acknowledge that, in certain contexts, a code smell or an antipattern may be unavoidable and the best way to design and–or implement (part of) a system, *e.g.*, parsers are often Spaghetti Code.

Several approaches to specify and detect design smells have been proposed in the literature. They range from manual approaches, based on inspection techniques [11], to metric-based heuristics [3], [12], [13], where smells are detected according to sets of rules and thresholds defined on various metrics. Manual approaches were defined, for example, by Travassos *et al.* [11], who introduced manual inspections and reading techniques to detect code smells.

Marinescu [12] presented a metric-based approach to detect smells with detection strategies, which capture deviations from good design principles and consist of combining metrics with set operators and comparing their values against absolute and relative thresholds. Similarly to Marinescu, Munro [13] proposed metric-based heuristics to detect code smells; the heuristics are derived from template similar to the one used for design patterns [14]. He also performed an empirical study to justify the choice of metrics and thresholds.

Moha *et al.* [3] proposed the DECOR method to specify and automatically generate detection algorithms. DECOR includes a domain-specific language based on a literature review of existing work. It also includes algorithms and a platform to automatically convert specifications into detection algorithms and apply these algorithms on any system. DECOR produces detection algorithm with good precision and perfect recall while allowing quality assurance personnel to adapt the specifications to their context.

Khomh *et al.* [4] argued that threshold-based approaches do not handle the uncertainty of the detection results and, therefore, miss borderline classes, *i.e.*, classes with characteristics of design smells "surfacing" slightly above or "sinking" slightly below the thresholds because of minor variations in their characteristics. Consequently, they proposed a Bayesian Belief Network (BBN) for the detection of design smells in systems, which output is the probability that a class exhibiting the characteristics of a smell be truly a smell. Thus, their approach handles the degree of uncertainty for a class to be a smell. They also showed that BBNs can be calibrated using historical data both from a similar and from a different context.

Oliveto *et al.* [5] proposed ABS, an approach to detect design smells in systems using signatures of the classes and of the smells. The signature of a smell is computed as the average of the signatures of a set of known classes participating to that smell. For each class in a system, using B-splines, they compared the signature of the class with that of a smell and computed their similarity to detect occurrences of the smell. They reported a case study and claimed that ABS outperforms previous approaches in precision and recall while being simpler in practice.

Some visualisation techniques, for example [15], were used to find a compromise between fully-automatic detection techniques, which are efficient but lose track of the context, and manual inspections, which are slow and subjective. Other approaches perform fully-automatic detection and use visualisation to present the detection results [16], [17].

Catal *et al.* [18] used several machine learning algorithms to predict the defective modules. They investigated the effects of dataset size, metrics set, and feature selection techniques for software fault prediction problem. They employed several algorithms based Artificial Immune Systems.

Kessentini *et al.* [19] independently used an AIS to estimate the risks of classes to deviate from "normality", *i.e.*, a set of classes representing a "good" design. They used structural data to describe a design, *i.e.*, classes, fields, methods... They showed that 90% of the more riskiest classes in GanttProject

and Xerces are smells but do not discuss recall even though there is a balance between precision/recall [3].

Previous approaches advanced the state-of-the-art in the specification and detection of design smells but all require experts' knowledge and interpretation. Moreover, they focus on detecting one kind of design smells at a time, while some smells are similar and classes with characteristics similar but no identical to some smells are also of interest to developers and quality assurance personnel. In IDS, we use metrics computed on instances of smells as input to the AIS following our parallel between object-oriented software systems and living bodies. We analyses IDS in two distinct, industrial-like scenarios. We also discuss all the advantages of an AIS over previous approaches, including precision and recall.

III. Artificial Immune Systems

A. Biological Background

Innate immunity defends the body from any pathogens that enter the body. Adaptive immunity allows the immune system to attack any foreign pathogens that the innate system cannot destroy. It can distinguish between the body's own cells and foreign cells.

The adaptive immune system is directed against specific invaders and is modified by exposure to such invaders. It is made up of *lymphocytes* (B cells and T cells). These cells aid in the process of recognizing and destroying specific antigens. Immune responses are normally directed against the antigen that provoked them and are said to be antigen-specific. The immune system learns to react to particular patterns of antigens. On the surface, each lymphocyte cell have receptors to recognize antigens, which are specialized: each can match only one specific antigen.

B. Computer Models

An artificial immune system (AIS) is a classification algorithm that mimics the immune system defense mechanisms. It can accept patterns of arbitrary length and it has the ability to maintain and exploit previously learned data efficiently for improved performance in future encounters with pathogens.

An AIS produces a large number of randomly-created *detectors* to recognise the antigens located on the surface of the foreign pathogens. A *negative selection* mechanism is applied to eliminate detectors that match the body's own cells. Kept detectors become *naive* detectors; they die after some time, unless they match some antigens; in case of such a matching, they become memory cells. Detectors that match a pathogen are quickly multiplied via the *clonal selection* to accelerate the response to further attacks. Also, because the clones are not exact copies (they are mutated, the mutation rate being an increasing function of affinity between detectors and antigens), they can both better focus on pathogens and handle similar pathogens (*affinity maturation*).

We draw a parallel between the immune system and the detection of smells. A system design is similar to a living body. It is protected from design smells by a detection approach, as a body is by its immune system. The detection approach identify smelly classes, *i.e.*, pathogens, using some characteristics of the classes—in the following, metrics values—by comparing them to sets of metrics values of smelly classes. Our novel detection approach, IDS (Immune-based Detection Strategy), can classify cells (system classes) that are present in the body (system design) as body's own cells (well-design classes) and foreign cells (smell-prone classes). We choose to characterise the body's and foreign cells with a set of metrics[1] [20].

C. Implementations

In general, any AIS algorithm has five steps: initialization, antigen training, competition for limited resources, memory cell selection, and classification [21]. The first step and the last step are applied only once, but Steps 2, 3, 4 are used for each sample (antigens) in the dataset.

Carter [22] developed the first AIS-based classification algorithm. It is a supervised learning system based on a high-level abstraction of *T cells*, *B cells*, *antibodies*, and an *amino-acid library*. The artificial T cells control the production of B-cell populations, which compete for recognition of the *unknowns*. The amino-acid library acts as a library of *epitopes* or characteristics of antigens currently in the system. When a new *antigen* is introduced into the system, its variables are entered into this library. The T cells then use the library to create their receptors that are used to identify the new antigen [23]. Brownlee [21] developed another algorithm, called Immunos-99, to combine the benefits of AIS-based classification algorithm with clonal selection classification algorithm [24]. Each step of this algorithm is explained below:

```
1:  Divided data into antigen-groups (by classification labels)
2:  Prepare a B-cell population
3:  for each antigen-group do
4:     Create an initial B-cell population for an antigen-group
5:     for each generation do
6:        Create an initial B-cell population for an antigen-group.
7:        Calculate fitness scorings
8:        Perform population pruning
9:        Perform Affinity maturation
10:       Insert randomly selected Antigens of the same group
11:    end for
12: end for
13: Perform final pruning for each B-cell population
14: Present the final B-cell populations as the classifier
```

First, the algorithm divides the provided antigens into groups. Once prepared, a new B-cell population is created from each antigen group. The initial size of each B-cell population equals the number of antigens in the original group. The population is then exposed to all antigens, one antigen at a time and an affinity value is calculated for each B-cell to the antigen. The B-cell populations are then sorted in descending order of affinities. Once the training steps completed, the resulting B-cell populations form the classifier for new (invading) antigens. Each B-cell population is exposed to a new antigen and populations compete for "ownership" of the new antigen using their affinity. The population that have the highest affinity classifies the new antigen with its label.

Feature selection can be useful in reducing the dimensionality of the data to be processed by the classifier: reducing the dimensionality of the data reduces the sizes of the

[1] http://www.ptidej.net/downloads/experiments/quatic10/

	Numbers of				
	Classes	KLOCs	Blobs	FDs	SCs
Gantt Project	188	31	4	4	4
Xerces	589	240	15	15	18
Total	777	271	19	19	22

TABLE I
SYSTEM CHARACTERISTICS.

hypothesis space and, thus, results in faster execution time and improving predictive accuracy (inclusion of irrelevant features can introduce noise into the data, thus obscuring relevant features). Therefore, in our context of metric-based software quality classification, we could need a subset of metrics that can discriminate between the non-smell-prone and smell-prone classes. However, our approach does not need any feature selection, because its classification algorithm uses data reduction: deletion of irrelevant data (antigens) during the generation of B-cells.

IV. STUDY DEFINITION AND DESIGN

We perform a series of experiments to assess the performance in time, precision, and recall of our novel approach for design-smells detection. Following the Goal Question Metric (GQM) methodology [25], the *goal* of our experiments is to analyse the performance of our approach and understand whether it performs better than previous approaches. The *purpose* is to provide an approach for design-smells detection. The *quality focus* is to provide a set of smell occurrences (*i.e.*, classes with characteristics violating design principles) with good precision and recall and in a reasonable time. The *perspective* is both of developers and quality assurance personnel, who perform evaluation activities and are interested in locating accurately parts of a system that need improvements; and researchers, who want to study design smells. The *context* of our experiments is both development and maintenance.

We conduct our experiments using two open-source Java systems: GanttProject v1.10.2 and Xerces v2.7.0, which characteristics are summarised in Table I. GanttProject[2] is a system for creating project schedules by means of Gantt charts and resource-load charts. It enables breaking down projects into tasks and establishing dependencies among tasks. Xerces[3] is a family of packages for parsing and manipulating XML files. It implements a number of standard API for XML parsing, including DOM, SAX, and SAX2. We chose these systems because they are medium-size systems and manually-validated occurrences of Blob, Functional Decomposition (FD), and Spaghetti Code (SC) are available [3], [4].

We want to answer the two research questions:

- **RQ1**: To what extent an AIS-based approach can detect design smells in a system?
- **RQ2**: Is our approach better than state of-the-art approaches, such as DECOR, and BBNs?

We answer **RQ1** in the following two scenarios:

- *intra-system identification*: In this first scenario, we study how knowledge of previously-detected Blobs in a given system, Xerces v2.7.0, can help predict occurrences of other design smells in the same system. We divide the classes of Xerces in three subsets with 16 occurrences of Blobs in each subset. Then, we train IDS on two of the subsets and apply it on the third subset (of the same system) in a 3-fold cross-validation.
- *extra-system identification*: In this second scenario, we study the performance of our approach using heterogeneous data. We assume that a developer has access to historical data from one system, *e.g.*, GanttProject. We use this data to detect occurrences of design smells in the other system, Xerces. We also perform the same study in the other direction, *i.e.*, using design smells in Xerces to detect occurrences in GanttProject.

[2]http://ganttproject.biz/index.php
[3]http://xerces.apache.org/

	Numbers of			
	Design Smells	False Positives	Precisions	Recalls
Subset 1	16	1	94.11%	100%
Subset 2	16	2	88.23%	100%
Subset 3	16	2	88.23%	100%
Average			90.19%	100%

TABLE II
INTRA-SYSTEM DETECTION ON XERCES: 3-FOLD CROSS VALIDATION.

We answer **RQ2** by comparing the performance of IDS in precision, and recall, as computed in Scenario 2, against that of previous approaches.

In each scenario and research question, we use publicly-available data [3], [4] as oracle. We collect the numbers of true and false positive occurrences of the design smells detected by our approach and compare them with the oracle using the following IR metrics defined in [26]:

$$precision = \frac{|correct \cap detectedd|}{|detected|}$$

$$recall = \frac{|correct \cap detected|}{|correct|}$$

where *correct* represents the set of known instances of the design smells and *detected* that of candidate occurrences detected by an approach.

V. STUDY RESULTS, ANALYSES AND DISCUSSIONS

We now discuss the results of our experiments.

A. Answer to RQ1

In this first scenario, we use 3-fold cross validation. Table II shows the precisions corresponding to each fold. The average precision is 90.19% and the recall is 100%. The results also confirmed that IDS is not limited to the detection of a specific design smell: although we train our approach on instances of Blob, FD, and SC, it was also able to detect LargeClass and LongMethod. Overall, IDS detects all smelly classes, *i.e.*, classes deviating from specific *good* design rules, *exemplified* by some design smells.

In the second scenario, we trained our approach on GanttProject v1.10.2 and applied it on Xerces v2.7.0 and vice-versa. Table III shows the results. On Xerces, our approach achieved a precision above 80%. The precision for GanttProject, although slightly lower at 65.0%, is still interesting

	GanttProject (Trained on Xerces' Blobs, FDs, SCs)	Xerces (Trained on GanttProject's Blobs, FDs, SCs)
# of Classes	188	589
# of Design Smells Instances	20	54
# of False Positives	7	10
Precision	65.0%	81.48%
Recall	100%	100%

TABLE III
INTER-SYSTEM DETECTION.

considering that the approach was trained on a system from a different context. Moreover, a review of the false positives show that these classes have characteristics similar to the smells and, therefore, may eventually degenerate in design smells in the near future. Hence, they are also interesting to developers as they may be interested in preventing their further decay. Our approach in both cases achieved 100% recall, succeeding in returning all smelly classes in the systems.

We thus answer **RQ1** positively: the results suggest that even in the absence of historical data on a specific system, developers or quality assurance personnel could use IDS on different systems and obtain good precision and recall.

B. Answer to RQ2

To answer RQ2, we compare the results of our approach against that of the state-of-the–art approaches: DECOR [3] and BBNs [4]. Table IV summarises the results achieved by each approach. Globally, IDS outperforms DECOR and BBNs in term of precision. Moreover, DECOR and BBNs require expensive tuning by experts (in time and knowledge) to have acceptable precision and recall. Indeed, DECOR relies on rule cards built by expert while BBNs need experts' knowledge to build their learning structure. For these two approaches, an incomplete experts' knowledge can cause a high number of false positives, resulting in a waste of time and resources for developers that must skim through the results. IDS does not rely on any experts' knowledge but on a set of metrics characterising known instances of smells. Therefore, IDS reduces the bias introduced in BBNs by the experts structuring the BBNs and in DECOR when crafting rule cards.

Moreover, contrary to DECOR and BBNs, IDS detects a larger set of design smells. As presented in Table IV, when trained only on instances of Blob, IDS was *still* able to detect the instances of the other smells: on GanttProject, it returned all the 4 true occurrences of Blob and also 11 true occurrences of SC; on Xerces, it returned again all the 15 true occurrences of Blob, 18 true occurrences of SC, and 14 occurrences of LargeClass. IDS also reported classes with borderline structure that may decay in design smells in the near future. Another strong point of IDS is its computation time, *i.e.*, in the order of milliseconds, *e.g.*, on Xerces, IDS detects Blob occurrences in 0.26s, while DECOR takes 2.4s.

We thus answer **RQ2** also positively: our approach has precisions and recalls superior than those of DECOR and BBNs and can detect similar smells as well as classes similar but not identical to some design smells.

C. Threats to Validity

The main threat to the *external validity* of our experiments that could affect the generalisation of the presented results relates to the analysed systems. We only used two medium-size systems, yet they support different activities and are open-source, thus available for replication. We plan to replicate our experiments on larger systems to confirm our results.

The subjective nature of specifying and detecting design smells and assessing detected classes is a threat to the *internal validity* of our experiments. Our understanding of design smells may differ from that of others. Our oracle, used to analyse our approach, was manually built by analysing the two systems used in the experiments. Three of the authors independently re-validated the publicly-available data [3], [4] to reduce the risk of classification errors. Finally, a candidate occurrence was classified as a real smell only when two out of three authors classified it as such. Such a process makes us quite confident about the accuracy of the oracle.

VI. CONCLUSION AND FUTURE WORK

The detection of design smells in object-oriented software systems is important to improve and assess the quality of the systems, to ease their maintenance and evolution, and, thus, to reduce the overall cost of their development and ownership.

Current automated detection approaches are difficult to develop and put into place because they require experts' knowledge and interpretation. Moreover, they focus on detecting one kind of design smells at a time, while some smells are similar and classes with characteristics **similar but not identical** to some smells are also of interest to developers and quality assurance personnel.

In this paper, we presented the first systematic parallel between artificial immune systems and the detection of design smells; a machine learning technique inspired from the immune system of the human body.

We performed experiments using GanttProject v1.10.2 and Xerces v2.7.0 and the Blob, FD, and SC design smells. The experiments showed that an AIS can detect smells in programs with good precision and recall and address the limitations of previous work: it does not require experts' knowledge and interpretation and it can report classes that are similar but not identical to the detected smell.

Moreover, our AIS-based approach has the following additional benefits with respect to previous approaches. **Generalisation**: It does not need all of the data set to detect similar or identical occurrences of the design smells. It has data reduction capability: it does not require feature selection, *i.e.*, choosing the set of metrics. **Parameter Stability**: Current freely-available implementation of AIS are not optimised for the detection of design smells but already provide good precision and recall. **Adaptability**: It is adaptable and, in some cases, self-organising and thus can automatically identify new patterns in the data to create a different representation of the data being learnt. **Portability across Systems**: It has a good precision and recall when applied to different systems while previous work require recalibration of the conditional

	GanttProject				Xerces			
	DECOR	BBNs	IDS		DECOR	BBNs	IDS	
			(Trained on Blob)	(Trained on Blob, FD, SC)			(Trained on Blob)	(Trained on Blob, FD, SC)
Blob	16 (8.5%)	7 (3.7%)	34		44 (7.6%)	41 (6.9%)	55	
	4 (2.1%)	4 (2.1%)	4	20 (10.6)	15 (2.5%)	15 (2.5%)	15	54 (9.1%)
	25 %	57.1 %	11.76%	13 (6.9%)	33.3%	36.5 %	27.27%	44 (7.4%)
FD	15 (8.0%)	–	–	65%	29 (5.6%)	–	–	81.48%
	4 (2.1%)	–	–		15 (2.9%)	–	–	
	26.7%	–	–		51.7%	–	–	
SC	14 (7.4%)	–	–		76 (14.8%)	–	–	
	4 (2.1%)	–	11		18 (3.0%)	–	18	
	28.5%	–	32.35%		23.68%	–	32.72%	
Average	26.73%	57.1 %		65%	36.22%	36.5%		81.48%

TABLE IV

RESULTS OF APPLYING THE DETECTION APPROACHES. (In each row, the first line is the number of detected classes, the second is the number of classes *being* design smells, the third is the precision. Numbers in parentheses are the percentages of classes being reported.)

probabilities [4] or changes of thresholds [3], [5]. **Simplicity and Self-regulatory**: It does not require a topology or a rule card: no experts' knowledge and interpretation. Thus, it does not embed the experts' subjective understanding of the design smells. However, it requires an oracle providing occurrences of some smells. **Performance**: It has good performance in terms of precision, and recall, and it computation is very fast. The results of the experiments showed that its precision and recall are comparable or superior to that of previous approaches.

We conclude that the application of an artificial immune system to detect design smells is valuable. The immune system provides an interesting metaphor for detecting design smells. In future work, we plan to compare our approach with other machine learning techniques, such as support vector machine, and to further study the parameters of the approach, including refining the choice of characteristics of classes. Finally, we will package our approach for the automated detection of design smells to provide it to developers and quality assurance personnel. We also plan to perform more in vivo (larger systems) and in vitro experiments on our approach.

Acknowledgment. This work was partially supported by the NSERC Research Chair in Software Patterns and Patterns of Software and the FQRNT.

REFERENCES

[1] M. Fowler, *Refactoring – Improving the Design of Existing Code*, 1^{st} ed. Addison-Wesley, June 1999.

[2] 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*, 1^{st} ed. John Wiley and Sons, March 1998.

[3] Naouel Moha, Y.-G. Guéhéneuc, L. Duchien, and A.-F. Le Meur, "DECOR: A method for the specification and detection of code and design smells," *Transactions on Software Engineering (TSE)*, 2009, 16 pages. [Online]. Available: http://www-etud.iro.umontreal.ca/~ptidej/yann-gael/Work/Publications/Documents/TSE09.doc.pdf

[4] Foutse Khomh, Stéphane Vaucher, Y.-G. Guéhéneuc, and H. Sahraoui, "A bayesian approach for the detection of code and design smells," in *Proceedings of the 9^{th} International Conference on Quality Software (QSIC)*, C. Byoung-ju, Ed. IEEE Computer Society Press, August 2009, 10 pages. [Online]. Available: http://www-etud.iro.umontreal.ca/~ptidej/yann-gael/Work/Publications/Documents/QSIC09.doc.pdf

[5] R. Oliveto, F. Khomh, G. Antoniol, and Y.-G. Guéhéneuc, "Numerical signatures of antipatterns: An approach based on b-splines," in *Proceedings of the 14^{th} Conference on Software Maintenance and Reengineering*, R. F. Rafael Capilla and J. C. Dueas, Eds. IEEE Computer Society Press, March 2010.

[6] T. Oda and T. White, "Increasing the accuracy of a spam-detecting artificial immune system," in *Proceedings of the Congress on Evolutionary Computation (CEC 2003)*, vol. 1, Canberra, Australia, 2003, p. 390396.

[7] B. F. Webster, *Pitfalls of Object Oriented Development*, 1^{st} ed. M & T Books, February 1995.

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

[9] M. Mantyla, "Bad smells in software - a taxonomy and an empirical study." Ph.D. dissertation, Helsinki University of Technology, 2003.

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

[11] G. Travassos, F. Shull, M. Fredericks, and V. R. Basili, "Detecting defects in object-oriented designs: using reading techniques to increase software quality," in *Proceedings of the 14^{th} Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM Press, 1999, pp. 47–56.

[12] R. Marinescu, "Detection strategies: Metrics-based rules for detecting design flaws," in *Proceedings of the 20^{th} International Conference on Software Maintenance*. IEEE CS Press, 2004, pp. 350–359.

[13] M. J. Munro, "Product metrics for automatic identification of "bad smell" design problems in java source-code," in *Proceedings of the 11^{th} International Software Metrics Symposium*, F. Lanubile and C. Seaman, Eds. IEEE Computer Society Press, September 2005.

[14] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns – Elements of Reusable Object-Oriented Software*, 1^{st} ed. Addison-Wesley, 1994.

[15] F. Simon, F. Steinbrückner, and C. Lewerentz, "Metrics based refactoring," in *Proceedings of the 5th European Conference on Software Maintenance and Reengineering (CSMR'01)*, 2001.

[16] M. Lanza and R. Marinescu, *Object-Oriented Metrics in Practice*. Springer-Verlag.

[17] E. van Emden and L. Moonen, "Java quality assurance by detecting code smells," in *Proceedings of the 9th Working Conference on Reverse Engineering (WCRE'02)*. IEEE CS Press, Oct. 2002.

[18] C. Catal and B. Diri, "Investigating the effect of dataset size, metrics sets, and feature selection techniques on software fault prediction problem," *Information Sciences, Elsevier*, vol. 179, no. 8, pp. 1040–1058, 2009.

[19] M. Kessentini, S. Vaucher, and H. Sahraoui, "Deviance from perfection is a better criterion than closeness to evil when identifying risky code," in *Proceedings of the 25th International Conference on Automated Software Engineering*. IEEE Computer Society Press, September 2010.

[20] Y.-G. Guéhéneuc, H. Sahraoui, and Farouk Zaidi, "Fingerprinting design patterns," in *Proceedings of the 11^{th} Working Conference on Reverse Engineering (WCRE)*, E. Stroulia and A. de Lucia, Eds. IEEE Computer Society Press, November 2004, pp. 172–181, 10 pages.

[21] J. Brownlee, "Artificial immune recognition system: a review and analysis," Swinburne University of Technology, Tech. Rep. 1-02, 2005.

[22] J. H. Carter, "The immune system as a model for pattern recognition and classification," *American Medical Informatics Association*, vol. 7, no. 1, pp. 28–41, 2000.

[23] J. Timmis and T. Knight, "Artificial immune systems: Using the immune system as inspiration for data mining," 2001.

[24] J. Brownlee, "Clonal selection theory clonalg. the clonal selection classification algorithm," Swinburne University of Technology, Tech. Rep. 2-02, 2005.

[25] R. Basili and D. M. Weiss, "A methodology for collecting valid software engineering data," *IEEE Transactions on Software Engineering*, vol. 10, no. 6, pp. 728–738, November 1984.

[26] R. Baeza-Yates and B. Ribeiro-Neto, *Modern Information Retrieval*. Addison-Wesley, 1999.