Metric and Rule Based Automated Detection of Antipatterns in Object-Oriented Software Systems

Mehmed Taha Aras
Computer Engineering Department
Yildiz Technical University
Istanbul/Turkey
taha_aras@hotmail.com

Asst. Prof. Dr. Yunus Emre Selçuk
Computer Engineering Department
Yildiz Technical University
Istanbul/Turkey
yunus@ce.yildiz.edu.tr

***Abstract*—Patterns are techniques to improve design and enhance reusability. Design patterns are general solutions which are used for common problems in object oriented systems. Code and design smells are symptoms of weak design and development, problems that reside deep in code and reduce the quality of software. The antipattern concept is also introduced as poor solutions to solve recurring problems, even though developers think that they practice a design pattern. It is proven that antipatterns have negative effects on maintainability, flexibility and readability of object oriented software systems. In this research, we propose a metric and a rule based automated antipattern detection system for object oriented software. This system consists of three main mechanisms to detect an antipattern. These mechanisms are "Metric Analyzer", "Static Code Analyzer" and "Filtering Mechanism". We specified three antipatterns to analyze; namely Blob, Swiss Army Knife and Lava Flow. Thresholds that are used to detect antipatterns are determined considering six reference projects' results and averages of the analyzed project itself. Detection algorithms have been applied on a set of hand-crafted Java classes and accuracy percentages are measured according to the produced results.**

Keywords—antipattern; code smell; automated detection; metric based detection; rule based detection;

I. INTRODUCTION

Code smells are defined as the recurring signs of weak design and coding by Fowler [1]. Code smells decrease readability, flexibility and increase error-proneness of projects [2], [3], [4]. Thus code smells are needed to be analyzed, detected and refactored to prevent building complex and costly software systems. Code smells are also the indicators of possible design smells [5] which disturb basic design standards and decrease the quality of design. Fowler introduced 22 code smells and pointed to the need of refactoring operations [5].

Software projects mostly deal with large products that contain too many components, making their structures more complex and hard to resolve [6]. One of the most powerful techniques to prevent these messy and complicated architectures is the usage of design patterns. Design patterns are ordinary and efficient techniques used for enhancing the design and supporting the maintainability, reusability and reverse engineering [7]. Adhering to design patterns improves understandability and maintainability as well [8].

Antipatterns resemble patterns; they are used as right solutions but opposite to patterns they commonly provide wrong and hazardous solutions to existing problems [9]. The word antipattern is coined by Koenig in 1995 [10] and the notion was firstly introduced by Akroyd in 1996 as a reaction of pattern, which is commonly used even though it is a wrong practice [11]. Antipatterns are poor solutions to recurring design problems. These poor solutions negatively affect development and maintenance phases by decreasing comprehensibility of the system, decreasing readability of the source code and reducing flexibility of the software [12]. Code and design smells contain low-level or local problems and they are significant symptoms of antipatterns which are more common design smells [1], [13]. Due to harmful effects of antipatterns, they need to be carefully detected and eliminated.

We present a metric and rule based automated antipattern detection system which aims to detect three antipatterns. These antipatterns are Blob, Swiss Army Knife and Lava Flow. Our approach considers both metric values and rule validations. We have three different mechanisms to decide the existence of antipatterns in source code. Firstly, there is a metric analyzer mechanism which extracts some basic code metrics as well as the Chidamber and Kemerer (CK) metrics [14] from the source code and tries to give meanings to them. Second one is a static code analyzer mechanism which parses the source code according to certain rules, extracts some code usages and tries to give them meanings. After these two mechanisms complete their jobs, some raw data show up but it is not enough yet for deciding the suspicious classes. Intrinsically deciding suspicious classes is a very troublesome and complex process in antipattern detection. Because results may differ easily according to some factors such as the size of project, coding style of developer and richness of the content of programming language. Thus, our third mechanism is a filtering mechanism which filters the raw data in order to understand the coding style of programmer and considering the size of project.

The paper is organized as follows: Section II presents related works and discusses their pros and cons. Section III describes our approach and mechanisms inside. Section IV presents and discusses the results of our study. Section V concludes and mentions future work.

II. RELATED WORKS

Several different approaches were proposed to detect antipatterns. These approaches have different advantages and disadvantages depending on the type of detection approach.

A. Detection Techniques

Three types of detection techniques will be discussed in this part. They are Manual and Auto Detection, Rule Based Detection and BBN Based Detection.

1) Manual and Auto Detection

Travassos et al. [15] presented their approach which aims to determine design smells using manual review and reading techniques. Their technique relies on only manual inspection and smells are not specified. Because of these reasons it is not proper to use this approach on large systems.

Marinescu [16] proposed an approach which depends on metric-based evaluation to detect design smells and implemented it in the IPLASMA tool. This approach needs deep knowledge of metrics to successfully detect an antipattern instance. Also changes in threshold values may cause very different results. Thus determining thresholds is critical and does not tolerate any mistake.

Dhambri et al. [17] tried to find a trade-off between manual and fully automated detection techniques. The aim of their approach is obtaining a technique which does not suffer from high time and effort consumption in large systems, while avoiding uncertainty problems.

Munro [18], Alikacem [19] and Ciupke [20] proposed their manual approaches which depend on mostly manual assessments to detect antipatterns.

Lanza and Marinescu [21] and van Emden and Moonen [22] introduced fully automated approaches to prevent uncertainty and long list problems. They used visualization techniques to present detection results. Their studies ignore analyst opinion.

Moha et al. [5] presented a DSL-based (domain specific language) approach, DECOR, depending on some set of rules which describes antipatterns and created an antipattern – smell taxonomy. They defined rule cards and a mechanism to convert those rule cards into antipattern detection algorithms. They focused on and identified some well-known antipatterns and tried to detect these antipatterns with auto-generated algorithms.

Studies of Simon [23], Rao [24] and Khomh [25] are other automated approaches which were conducted to detect antipatterns.

2) Rule Based Detection

Informative rule descriptions are the keystone of most of the antipattern detection approaches. These rules are manually defined by analysts and aim to identify the indications that characterize smells. They are formed as combinations of essentially quantitative, structural and/or lexical symptoms [26]. Each smell needs its own detection rule and a right threshold value which is a very critical decision. Thus, the number possible antipattern instances could be very large to detect the antipatterns manually using these rules [9].

3) BBN Based Detection

BBN (Bayesian belief network) based detection approaches preclude uncertainty problems and past results can be used to improve efficiency [25]. Probabilities of classes to be antipatterns are used, but this process is expensive and needs high time and knowledge [27]. Mentioned process requires expert decisions, thus lots of candidates will be detected as antipattern instances which are actually not antipattern instances. BBN models are compatible only with their given context and cannot be generalized to other contexts. This is a factor that decreases the flexibility of the approach and requires extra work to solve that issue [7].

B. Problems Encountered

Dhambri et al. [17] introduced six problems in antipattern detection processes and must be carefully considered to present an efficient detection approach [12]:

Problem 1: Making a decision that a class exists as an antipattern or not, may be an uncertain operation. Because a class which shows all symptoms of an antipattern may be a regular class. Thus analyst opinion is needed in this process to determine antipatterns more accurately.

Problem 2: Listing large set of candidate classes can be an inefficient process. Results may have too many false positives thus detection technique may be discarded. Assessments of an analysts are needed to produce a list of candidates.

Problem 3: Considering the assessments of analysts is a complex process. A design or class which is interpreted strong, regular or poor by analyst depends on knowledge, experience and background of analyst.

Problem 4: Detection framework needs to be considered while detecting antipatterns. Classes which are specified "good" by most of quality analysts may violate some design concepts in a specific environment.

Problem 5: Determining thresholds is a significant process in antipattern detection. Different threshold values may be interpreted differently by quality analysts.

Problem 6: Improving and using semantic data are problematic operations. Some antipattern symptoms include semantic data and an automated detection technique is required to consider them.

Kaur and Kaur [12] also compared some manual and automated approaches to the problems suggested by Dhambri [17]. They visualized these comparisons Table 1 and Table 2 [12].

III. PROPOSED APPROACH

We propose a metric and rule based automated antipattern detection approach for object-oriented software systems. Our proposed method consists of three main steps to obtain the final results of suspicious classes.

TABLE I. THE PROBLEMS CONSIDERED BY SOME MANUAL DETECTION APPROACHES [12]

Problems	Manual Approaches					
	Travassos et al. [15]	*Marinescu [16]*	*Munro [18]*	*Alikacem & Sahraoui [19]*	*Dhambri et.al. [17]*	*Ciupke [20]*
Problem 1	Y					
Problem 2						
Problem 3	Y				Y	Y
Problem 4	Y				Y	Y
Problem 5		Y	Y	Y	Y	
Problem 6	Y	Y	Y			Y

TABLE II. THE PROBLEMS CONSIDERED BY SOME SEMI-AUTOMATED DETECTION APPROACHES [12]

Problems	Semi-automated Approaches						
	Simon et al. [23]	*van Emden & Moonen[22]*	*Lanza & Marinescu [21]*	*Rao & Reddy [24]*	*Moha [5]*	*F. Khomh et al. [25]*	*Our Approach*
Problem 1				Y		Y	
Problem 2						Y	Y
Problem 3	Y						Y
Problem 4	Y				Y	Y	
Problem 5	Y	Y	Y		Y	Y	Y
Problem 6					Y	Y	

We used the Java programming language to develop the detection algorithms and our target classes which will be analyzed are Java classes. In this study we focused on three antipatterns which are Blob, Swiss Army Knife and Lava Flow. We determined thresholds values considering accepted values in literature, average of big projects that we analyzed and dynamic code statistics which show us the developer coding style.

Reference projects, which were analyzed to determine thresholds values, are six large scale projects. Three of these projects are open source projects by the Apache foundation: BCEL, Commons IO and Maven Core. Other three projects belong to one of the biggest airline companies.

A. Inspected Antipatterns

1) Blob

Blob, also named God Class or God Object, is a well-known antipattern. Blob is a class which has different and too many responsibilities, generally characterized by having high number of attributes and methods. It performs different works and have many dependencies with data classes [28].

We specified four factors to detect Blob classes in source code. These factors are:

- LCOM (Lack of Cohesion in Methods - CK Metric)
- RFC (Response for a class - CK Metric)
- NAM (Number of Attributes and Methods)
- NADC (Number of Accessed Data Classes)

We determined NAM and NADC threshold values considering the distribution of results of reference projects. Also these values can be reconsidered dynamically if the size of project is very unusual. General thresholds for LCOM and RFC are specified by Ferreira [29] and Jhans et al. [30], respectively. These thresholds are specified in Table 3.

TABLE III. THRESHOLD VALUES OF BLOB ANTIPATTERN METRICS

	Good	Regular	Bad
LCOM	0-1	1-20	> 20
RFC	0-50	50-100	> 100
NAM	0-10	10-20	> 20
NADC	0-3	3-6	> 6

2) Swiss Army Knife

Swiss Army Knife antipattern occur when one tries to meet all possible needs in a single class. Therefore, these classes have large number of signatures and complicated interfaces [7]. This antipattern is different from Blob, as it is an indicator of different responsibilities and high complexity while a Blob class is a center of processing and data of the system.

We specified three factors to detect Swiss Army Knife antipattern on source code: LCOM, OPI (Out of Package Imports) and TSC (Total Signature Count). OPI and TSC threshold values were determined by analyzing the reference projects and given in Table 4.

TABLE IV. THRESHOLD VALUES OF SWISS ARMY KNIFE A.PATTERN METRICS

	Good	Regular	Bad
LCOM	0-1	1-20	>20
OPI	0-4	4-8	>8
TSC	0-5	5-10	>10

3) Lava Flow

Lava Flow, also named as Dead Code, is a class or project that has an important size of unused or nonfunctional code parts which no one has an opinion about those parts. In large projects which have many developers coding, dead code starts to accumulate when a developer decides to stop using a code part and leaves it like that.

Most of studies have not dealt with Lava Flow antipattern and there is no enlightening information about detection of it. We proposed three factors to detect a Lava Flow antipattern:

- ICU (Is class used?)
- PMS (Percentage of methods suspicious)
- PFS (Percentage of fields suspicious)

We used average values and distribution of results of our reference projects to determine thresholds values of these factors. We also consider the average of subject project when determining PMS value. Because there are many different method usage styles and we see that PMS value can result wide range, but these values are consistent in the project itself.

ICU metric consists of boolean values as it is named. ICU results are used for inferring some logical results more than mathematical calculations. For example; if a class is used and none of its methods are used in the whole project, we can infer that there is an irrational situation. Thus necessary intervention is done to solve that issue as dynamically predicting true outcomes, accepting function as used/unused or ignoring the class from average. The thresholds of PMS and PFS are calculated by two factors:

1. Average of six reference projects
2. Average of the subject project itself

PMS and PFS values are not exact unused percentages. They show the suspicious methods and field percentages to decide if a class is antipattern or not. They have wide range PMS and PFS threshold values are specified in Table 5.

TABLE V. THRESHOLD VALUES OF LAVA FLOW ANTIPATTERN METRICS

	Good	Regular	Bad
PMS	0-10%	10-20%	>20%
PFS	0-6%	6-12%	>12%

B. Detection Mechanism

We proposed a detection mechanism which has three steps to obtain the final results of detection of antipattern classes.

1) Step 1: Metric Analyzer Mechanism

Aim of this mechanism is to extract of some code metrics and CK metrics from source code and to give meanings to them. Metric Analyzer is used for Blob and Swiss Army Knife antipatterns. LCOM, RFC, NAM, NADC, OPI and TSC metrics are determined by analyzing the project's source code. Ckjm [31] and JavaParser [32] libraries are used in the extraction process of mentioned metrics. After extracting some primitive values about classes such as declared functions, declared fields and declared imports; convertor algorithms process this raw data and obtain semi-meaningful metric data.

2) Step 2: Static Code Analyzer Mechanism

This mechanism is used only for the Lava Flow antipattern. Its aim is to detect classes which are suspected as being used or not used. This mechanism has its own file exploration process and compares each class with other classes in all packages in entire project. Basically, three steps are coded in this mechanism by defining some rules and searching matches according to those rules. These steps are:

- Determining whether a class is used in project by creating an instance of this class or statically.
- Finding whether suspicious methods are used locally in their owner classes or externally in the entire project.
- Detecting suspicious fields which have uncertainty of its local usage in its owner class.

All these steps have their own algorithms and these algorithms work according to some rules that we define in the beginning of our project. These rules provide us exact matchings of usage information as follows:

- Is class externally used by creating an instance of it?
- Is object created by upcasting or downcasting?
- Is function used statically?
- Is field locally used in its owner class?

After we get matching results by using these rules, we now have raw data of Lava Flow antipattern metrics. Some convertor methods process this raw data and produce semi-meaningful data.

3) Step 3: Filtering Mechanism

All semi-meaningful data generated in Step 1 and Step 2 are processed in a filtering mechanism to produce meaningful data. By using this data, we can understand suspicious antipattern classes more accurately. Our filtering mechanism aims to eliminate misleading factors such as coding style differences of developers, size and scope of the analyzed project and richness of content of programming language. All these factors may mislead interpreters of final results. Some antipatterns can be missed in detection. On the contrary, some regular classes can be marked as antipatterns. To prevent these unwanted conditions, we developed a filtering mechanism to increase accuracy of antipattern detection results.

First part of the filtering mechanism works dynamically during runtime and interfere the process if necessary. We defined some logical rules to decide whether an intervention is required. For example;

- If a class is used, at least a method has to be used externally.
- If a class has getter and setter methods, these methods should not be counted as suspicious because of the nature of data classes.

Second part of the filtering mechanism works after the results are processed. This operation mostly depends on statistics science. Some filtering operations are done in the result list, such as:

- Outlier analysis that eliminates extreme metric values which significantly affect accuracy of our results.

- Averages of some uncommon uses like interface signature metric are calculated in a sub-group of classes which implement one or more interfaces.

After those two filtering processes, our detection system produces the final accurate antipattern detection results. The overall diagram of the mechanism is given in Fig. 1.

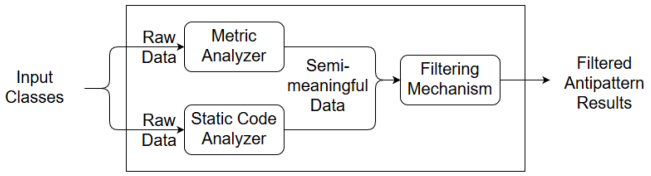

Fig. 1.Overall diagram

IV. RESULTS AND DISCUSSIONS

We tried our detection system on 36 java classes. This set of classes had 21 antipattern classes and 15 regular classes. In our subject set, each class has been developed as containing maximum 1 antipattern in order to analyze them separately. Each antipattern subgroup has 5 regular classes. Distribution of classes is specified below in Table 6.

TABLE VI. THE CLASS DISTRIBUTION OF SUBJECT SET

	Number of Classes
Regular	5 for each analyzed antipattern
Blob	8
Swiss Army Knife	6
Lava Flow	7

After these 36 classes are analyzed by our detection system, it marked classes as antipatterns or regular. Results of the detection of three antipatterns are visualized separately as confusion matrixes in Table 7, Table 8 and Table 9. In these tables, TP stands for "True Positive", meaning the class is actually an antipattern instance and our system has marked it as an antipattern instance. FP stands for "False Positive", meaning the class is not an antipattern instance but our system has marked it as an antipattern instance TN stands for "True Negative", meaning the class is actually not an antipattern instance and our system has marked it as not an antipattern instance. FN stands for "False Negative", meaning the class is actually an antipattern but our system has marked is as not an antipattern.

TABLE VII. CONFUSION MATRIX OF BLOB ANALYSIS RESULTS.

	Predicted: YES	Predicted: NO	Total:
Actual: YES	TP = 7	FN = 1	8
Actual: NO	FP = 1	TN = 4	5
Total:	8	5	

TABLE VIII. CONFUSION MATRIX OF SWISS ARMY KNIFE ANALYSIS RESULTS.

	Predicted: YES	Predicted: NO	Total:
Actual: YES	TP = 5	FN = 1	6
Actual: NO	FP = 2	TN = 3	5
Total:	7	4	

TABLE IX. CONFUSION MATRIX OF LAVA FLOW ANALYSIS RESULTS

	Predicted: YES	Predicted: NO	Total:
Actual: YES	TP = 5	FN = 2	7
Actual: NO	FP = 2	TN = 3	5
Total:	7	5	

Accuracy of detection of specific antipatterns can be calculated from these confusion matrixes according to (1).

$$\text{Accuracy} = (TP + TN) / (TP + TN + FP + FN) \quad (1)$$

Based on (1), accuracy results of antipattern detections are:

- Blob Detection Accuracy: 11/13 (84.6%)
- Swiss Army Knife Accuracy: 8/11 (72.7%)
- Lava Flow Accuracy: 8/12 (66.6%)

Lava Flow accuracy result is relatively low in comparison to other two antipatterns' results. We can infer that various usages of functions, objects and fields in object oriented programming languages originated this negative effect.

Average detection accuracy of the system is calculated as 74.6%. As can be seen in the satisfying accuracy results, our detection system completed efficiency tests successfully and produced meaningful data for the interpreter. These meaningful results can be acceptably used to determine and refactor the problematic classes.

A comparison of the accuracy of our approach with other most relevant works' is given in Table 10. Any two work is considered relevant if their detected antipattern types' sets intersect.

TABLE X. ACCURACY OF RELEVANT WORKS

	Blob	Swiss Army Knife	Lava Flow
F. Palomba [28]	76%	N/A	N/A
A. Maiga [33]	94.49%	76.63%	N/A
N. Moha [5]	88.6%	41.1%	N/A
Our Approach	84.6%	72.7%	66.6%

Our approach considers the following problems identified by Dhambri [17]: Large set of candidate classes are eliminated by our filtering mechanism thus we deal with the high number of false positives issue indicated in Problem 2. Our detection system uses the unification of some expressions which consist of rule and metric validations to determine an antipattern and does not involve analyst assessments. Thus we consider Problem 3. The last problem that address is the threshold definition problem. Thresholds are defined according to three factors in our approach. These factors are the filtered statistical results of some reference projects, the average results of analyzed project and accepted threshold values of some known metrics. Therefore we address Problem 5.

V. FUTURE WORK

In the next steps of our study, we aim to add new antipatterns to our detection system's scope. New rules and

metrics will be defined for detection of new antipatterns. Also new algorithms can be developed for the filtering mechanism.

One of the possible improvements to our work is adding the capability to suggest refactoring options to the end user. Considering the meaningful results, our system can interact with end user and propose possible refactoring options. Another improvement can be in evaluating the effects of programmers' coding style differences in the detection of antipatterns. We aim to enlarge our code usage rules and decrease the error margin of our system.

References

[1] M. Fowler, "Refactoring: improving the design of existing code," Addison-Wesley, 1999

[2] M. Abbes, F. Khomh, Y.-G. Guéhéneuc, and G. Antoniol, "An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension," 15th European Conference on Software Maintenance and Reengineering, CSMR 2011, 1-4 March 2011, Oldenburg, Germany. IEEE Computer Society, 2011, pp. 181-190.

[3] F. Khomh, M. Di Penta, Y.-G. Guéhéneuc, and G. Antoniol, "An exploratory study of the impact of antipatterns on class change- and fault-proneness," Empirical Software Engineering, vol. 17, no. 3, pp. 243-275, 2012.

[4] F. Khomh, M. Di Penta, and Y.-G. Guéhéneuc, "An exploratory study of the impact of code smells on software change-proneness," 16th Working Conference on Reverse Engineering, WCRE 2009, 13-16 October 2009, Lille, France. IEEE Computer Society, 2009, pp. 75-84.

[5] N. Moha, Y.-G. Guéhéneuc, L. Duchien, and A.-F. L. Meur, "Decor: A method for the specification and detection of code and design smells," IEEE Transactions on Software Engineering, vol. 36, no. 1, pp. 20-36, 2010.

[6] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S. Halkidis, "Design pattern detection using similarity scoring," IEEE Transactions on Software Engineering, vol. 32, no. 11, pp. 10-19, 2006.

[7] J. Din, A. B. AL-Badareen, and Y. Y. Jusoh, "Antipattern detection approaches in object-oriented design: a literature review," Computing and Convergence Technology (ICCCT), 2012 7th International Conference, 2012, pp. 926-931.

[8] F. A. Fontana, S. Maggioni, and C. Raibulet, "Understanding the relevance of micro-structures for design patterns detection," Journal of Systems and Software, vol. 84, no. 12, pp. 2334-2347, 2011.

[9] E. Gamma, "Design patterns: elements of reusable object-oriented software," Addison-Wesley Professional, 1995.

[10] A. Koenig, "Patterns and antipatterns," Journal of Object-Oriented Programming 8 (1), pp. 46-48, 1995.

[11] M. Akroyd, "Anti patterns session notes," in Object World West, 1996.

[12] H. Kaur, P. J. Kaur, "A study on detection of anti-patterns in object oriented systems," International Journal of Computer Applications (0975 - 8887), Volume. 93, No. 5, May 2014

[13] W. J. Brown, R. C. Malveau, W. H. Brown, H. W. McCormick III, and T.J. Mowbray, "Anti patterns: refactoring software, architectures, and projects in crisis, 1st ed. John Wiley and Sons, March 1998.

[14] S. Chidamber, C. Kemerer, "A metrics suite for object oriented design," IEEE Transactions on Software Engineering, June 1994.

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

[16] R. Marinescu, "Detection strategies: metric-based rules for detecting desing flaws," Proceedings of the 20th International Conference on Software Maintenance. IEEE Computer Society Press, pp. 350-359, 2004.

[17] K. Dhambri, H. Sahraoui, P. Poulin, "Visual detection of design anomalies," Proceedings of the 12th European Conference on Software Maintenance and Reengineering, IEEE Computer Society, pp. 279-283, 2008.

[18] M. J. Munro, "Product metrics for automatic identification of bad smell design problems in Java source-code," Proceedings of the 11th International Software Metrics Symposium, IEEE Computer Society Press, pp. 15, 2005.

[19] E. Alikacem, H. Sahraoui, Détection d'anomalies utilisant un langage de description de règle de qualité. In: Rousseau, R., Urtado, C., Vauttier, S. (Eds.), actes du 12e colloque Langages, Modèles, Objets. Hermès Science Publications, pp. 185–200.

[20] O. Ciupke, "Automatic detection of design problems in object-oriented reengineering," Proceeding of 30th Conference on Technology of Object-Oriented Languages and Systems, IEEE Computer Society Press, pp. 18-32, 1999.

[21] M. Lanza, R. Marinescu, "Object-oriented metrics in practice," Springer Berlin Heidelberg, 2006.

[22] E. van Emden, L. Moonen, "Java quality assurance by detecting code smells," Proceedings of the 9th Working Conference on Reverse Engineering (WCRE'02), IEEE Computer Society Press, 2002.

[23] F. Simon, F. Steinbrückner, C. Lewerentz, "Metrics based refactoring," Proceedings of the Fifth European Conference on Software Maintenance and Reengineering (CSMR '01), IEEE Computer Society, pp. 30-38, 2001.

[24] A. Rao, K. N. Reddy, "Detecting bad smells in object oriented design using design change propagation probability matrix," Proceedings of the International MultiConference of Engineers and Computer Scientists, 2008.

[25] F. Khomh, S. Vaucher, Y. Guhneuc, and H. Sahraoui, "Bdtext: A gqm-based bayesian approach for the detection of antipatterns," Journal of Systems and Software, vol. 84, no. 4, pp. 559-572, 2011.

[26] M. Kessentini, H. Sahraoui, M. Boukadoum, and M. Wimmer, "Search-based design defects detection by example," ser. Lecture Notes in Computer Science, Berling / Heidelberg: Springer, 2011, vol. 6603, pp. 401-415.

[27] R. Oliveto, F. Khomh, G. Antoniol and Y. Gueheneuc, "Numerical signatures of antipatterns: An approach based on b-splines," 14th European Conference on Software Maintenenance and Rengineering (CSMR), IEEE, 2010, pp. 248-251.

[28] F. Palomba, G. Bavota, R. Oliveto, A. de Lucia, "Antipattern detection: Methods, Challenges, and Open Issues," Advances in Computes, pp. 201-238, 2015.

[29] K. A. M. Ferreira, M. A. S. Bigonha, R. S. Bigonha, L. F. O. Mendes, H. C. Almeida, "Idetifying thresholds for object-oriented software metrics," The Journal of Systems and Software, vol. 85, pp. 244-257, 2012.

[30] I. K. Jhans, V.K. Priya, "Improved analysis of refactoring in forked project to remove the bugs present in the system," Internation Journal of Innovative Research in Science, Engineering and Technology, Vol.5, Issue. 2, February 2016.

[31] D. Spinellis, "Tool writing: A forgotten art?," IEEE Software, 22(4):9-11, July/August 2005.

[32] S. Viswanadha, J. V. Gesser, "Java Parser", https://github.com/javaparser/javaparser , Feb 2013.

[33] A. Maiga, et al., "SMURF: a SVM based incremental anti-pattern detection approach," Proc. 19th Working Conf. on Reverse Engineering (WCRE), IEEE Computer Society Press, pp. 466-475, 2012.