Contents lists available at Science-Gate

International Journal of Advanced and Applied Sciences

Journal homepage: http://www.science-gate.com/IJAAS.html

Automatic multiprogramming bad smell detection with refactoring

Amit Verma *, Ashish Kumar, Iqbaldeep Kaur

Department of Computer Science and Engineering, CGC Landran, Mohali-140307, Punjab, India

ARTICLE INFO

Article history:
Received 9 June 2017
Received in revised form
2 August 2017
Accepted 3 August 2017

Keywords:
Index terms- Code smell (CS)
Lazy class
Long class
Optimization algorithm
FAR and FRR

ABSTRACT

A code smell detection and refactor is one of the very hot concepts in these days. A Lot of researcher worked on it to create an automatic bad smell detection and refactoring system. Main purpose behind the development of these type of systems is to create automatic for enhance the development quality of software systems. In the previous research the smell detection system perform detection on specific areas or specific language. Due to this companies needs to use more than one detector for software testing for large projects. The system is combination of various modules which can be developed in various languages. Our proposed method which is helpful their users to test their code and detect bad smell on more than one language. It acts as a bridge with some optimization techniques which provide highly accurate working for smell detection along with refactoring. Proposed approach uses optimization along with fact and rule programming to detect and refactor the bad smell from input programs. Various bad smells like long methods, dead code, lazy class, long class, etc. are used to check the quality of the code. The proposed approach is also working for Java, c++ and c#.net codes for the test all these bad smell and refactor c++ and Java code. The performance of the proposed approach is also better than other existing algorithms in terms of accuracy for detection and refactoring of bad smells. Some other challenges that the proposed approach faced to find the smells in the code also affect the performance. One of the main challenges is the way of writing code is different for everyone. So it's difficult to detect and refactor the thing on smell detection tool. Proposed approach used fact and rule processing for detection and eliminates unwanted entries with the help of the optimization process. The performance in terms of accuracy and FAR, FRR are stable and better for all the test cases in the comparison of existing methods and proposed approach.

1. Introduction

Now a day's software has become a major part of everyone's life. The principle of the software is it reliability to create our lives easier, efficiency and gets better productivity.

Although, some efficiencies come at the most expensive of all-encompassing viewers. The software feature that is an achievement that human kind shall never dis-remembers. The main area of the code is some code smells that symbol of major architecture, rules and adversely impact of structure quality. CS is usually not exceptions, nor or neither are they precisely not correct and do reverify the program (Hazelwood and Smith, 2003). A code smells are less better design that might be speed slower down along with maximizing the high risk if exceptions or errors in the future. The code smell has been distinct as symbols of weak plan and program run able selection. In some situations, such symbols might be designed by events performed by developers while at a distance, such as, design urgent patch / simplest making sub choices (Chatzigeorgiou and Manakos, 2010).

Refactoring is definite that the easy and meaningful design of existing code, without modifying its behavior. It adds and changing their code a lot by creating reappearances. But they don't use regular refactoring as it isn't easy. This is since a factored code inclines to rot. Several forms are created by a factored / class, duplicate code and some other phases of mixed up and discontinuous (Counsell et al., 2011). All intervals of time the modification of code without refactoring it, degrades and dispersals. Code decay frustrates us along with

* Corresponding Author.
Email Address: dramitverma.cu@gmail.com (A. Verma)
https://doi.org/10.21833/ijaas.2017.09.010
2313-626X/

the expensive us time and smallest the lifetime of useful system.

Detected code smells will differ depending on the preferred likelihood threshold (Munro, 2005). Growing the probability too much will reason more false negative, while feeling it in excess will grounds more false positives. It will be up to the developer to fine adjust the threshold to get the sufficient level of advice with respect to the occurrence of code smells. It will also be up to the developer to choose the sufficiency to relate a given refactoring to eliminate a detected code smell.

2. Code smell classes types

Bad smells in software testing significant features of testing module which are sometime causes hard to maintain and design software systems. Here the Fowler offers a new concept; in this concept various types of bad smell are detected (Fokaefs et al., 2007). Later, other authors detect other bad smells in the program. The bad code smell in software testing exactingly linked with exercise of refactoring bad smells from the code, it helps to enhance the working and quality of software systems. Various developers are working for single software system (Table 1). Here they identify some problem with the software system which is known as bad smells, they should appraise whether these problems can affect the software system in terms of quality, their functional work, performance and maintenance. This metaphor technique can make the smell finding technique easier and refector the smells from the code (Tsantalis et al., 2008). In Software system the bad smells are some problems which indicate problems. It normally designates that software system should refactor and system re-checked for performance enhancement. This processing scheme offered by Kent Beck. This process increases the software quality by refactoring various bad smells. The list of bad smells in software system areas.

- **Long Parameter List:** when the method is too long means more number of lines of code.
- **Large Class**: Classes that have large numbers of instance variables and large number of lines of code. Due to large software system the large class suffer from duplication of software codes.
- **Long Parameters:** In this lists the parameters are very difficult to detect from a software system. These are those methods which having parameters more than the limit (Pérez et al., 2013).
- **Comments:** If the comments are present in the code more than the lines of code.
- **Switch Statements:** Switch statements may produce duplication. Sometime the similar switches are placed on various locations in the code. It can increase the maintenance cost of the code module (Bakota et al., 2006).
- **Lazy Class:** Classes that are not doing much work and a number of methods is null.
- **Temporary Field:** When some of the instance variables in a class are only used occasionally.
- **Duplicate Code:** This smell is also very important in software testing. Duplicate the code in a software system make the update process harder. Same code are placed more than once in the code can increase the time of detection issues and improvement of software systems.
- **Dead Code:** Sometime the software developer designs some coding modules but didn't use them inside the system. Those modules which are not used inside the code are known as dead code. Dead code is increase the memory consumption and operation cost of software system (Kim and Oh, 1997).

Table 1: Comparison of the various code smell detection tools

Code Smell	Definition	Variable used	Results
LM	MWMLOC	CC,LOC,#M	LOC >50 then no variable used CC > 50
LPL	MCS	#P, ∑n POM, AP[10]	NOP > 7 ∑n POM = 148 M in C = 88 AP = 3 #P > AP
LC	NETH	LOC, IV, DOI	LOC > 300 LM > 5 DIP> 3 coupling >10
DC	DNRC	UBoD[11]	UBoD =24
LaC	LaC should PR	#T, LOC	SoM ==0 LOC<=300 #M <=2
LCB	DECB	#UCB	TUCB = 5
DuC	DuC exists if CMWO	#DCB[12]	T#DCB =19

LM- Long Method, MWMLOC-Method with Maximum Lines of Code, CC- Cyclomatics Complexity, LOC-Lines of Code, #M-Number of Methods, LPL-Long Parameter List, MCS- Many

Constraints Passed, #P-Number of Parameter, POM-Parameter of Method, AP-Average Parameter, LC-Large Class, NETH-Not Enough To Handle, IV-Instance Variable, DOI- Depth of Inheritance, UBoD-Unused Block of Data, DC-Dead Code, DNRC-Delete Not Required Code, LaC-Lazy Class, PR-Predominantly Request, #T-Number of Technique, SoM-Several of Method, WMC-Weighted Method Count, #UCB-Number of Unused Catch Block, TUCB-Total number of Unused Catch Block, LCB-Lazy Catch Block, DECB- Discover Empty Catch Block, DuC-Duplicate Code, CMWO- Code Match With Other, #DCB-Number of Duplicate Code Block, T#DCB-Total Number of Duplicate Code Block.

3. Literature review

dos Reis et al. (2016) presented conducted a quasi-experiment by one hundred eighteen software systems. Here the author detects various smells in six different domains. Author used various testing methods to find them from a large software system. The ANOVA and Kruskal-Wallis tests are performed for find the smells from a software system. Palomba et al. (2013) worked on bug detection from a software system. This process is prediction based which are used to identify the smelly classes and list them to make a list of various smells in the code. The new method compared with other in terms of various performance matrixes in this research. Vidal et al. (2016) prioritizing the smells are process according to their groups which might cause the problems in software system. The detection of bad smells is performed on JSpirit tool in this research. Here more than 23 issues which identify the various structural problems from the code. Fowler and Beck (1999) identified various bad smells with some refactoring processes. This process provides the code modification on the basis of detected spots from the code. This process used to improve the performance of code with modification in the external part and the internal part modified automatically. This process eliminates the various bad smells and reduces the unwanted memory consumption and bugs. Van Emden and Moonen (2002) detected some bad smells form the code which is used to enhance the design and development of software system. The author worked on java platform with development of a prototype known as jCOSMO. This process make the software process lesser interms of cost and efforts along with easy and efficient correction of bad smells.

4. Proposed model

In proposed work the process of detecting bad smells is a combination of fact and rules with bees colony optimization to detect and optimize for verify various software modules to detect bad smells from the code. The fact and rules are used to process the code and make the software module identification for multiple programming languages. The proposed approach is also used some refactoring approach for enhance the performance of software systems. Programming languages are having different syntax and program structure. The proposed approach working with them on the basis of fact and rule processing and design a bridge between software module and detection network to enhance the evaluation of bad smell detection. The proposed method is able to work with C++, Java and .net software system with refactoring for various bad smells.

4.1. Bee colony algorithm

It consist group of employed bees, scout bees and onlooker bees. The bees whose food sources left or abandoned become scout and start searching the new food source. Onlooker sees and analyses the dance and select the food source accordingly. In starting all the food sources are given to employed bees. Then employed bees determine the other source and nectar is produces and start work on that hive. And then onlooker bees see the dance of employed bees and choose the food source accordingly. The empty food source is determined and gets replaced with the scout food source which has been searched by scout. By this process the best food source found and get registers

4.2. Proposed methodology

The proposed flow chart describes the working of smell detection for more than one language into single tool. The stepwise working shows uploading of software system and identifies modules from the large systems. The fact and rule programming apply on uploaded data and process with to design a bridge between software modules and smell processing system. Step 4 is used for analysis of various smells from the code and optimize in next step for verification of them. Verified data identified from the software system and provide a output to the user in terms of bad smell. Bad smells are refector to improve the performance of software system. Some limits are also used to calculate the performance of proposed model and compared with existing system to ensure the better performance.

Step 1: Code uploading for test cases. (Choose a project for case study which is in c++, java and .net)
Step 2: verify various modules of code
Step 3: calculate software metrics based on fact and rules technique.
Step 4: analysis of code on the behalf of software metrics
Step 5: Optimization of various detected software metrics using artificial bee's colony algorithm.
Step 6: Comparative studies with various results parameters.

5. Simulation results

The main page is a connection between all the modules in the proposed approach (Fig. 1). It works

as a linker to call data from other forms or transmit results from one to another form (Fig. 2). Project upload panel is here and user can upload their project for checking various test cases. Test window is also working as a linker between test modules and other parameter calculations. These are performing and transfer their calculation results on this window and here they are analyses and produce final results.

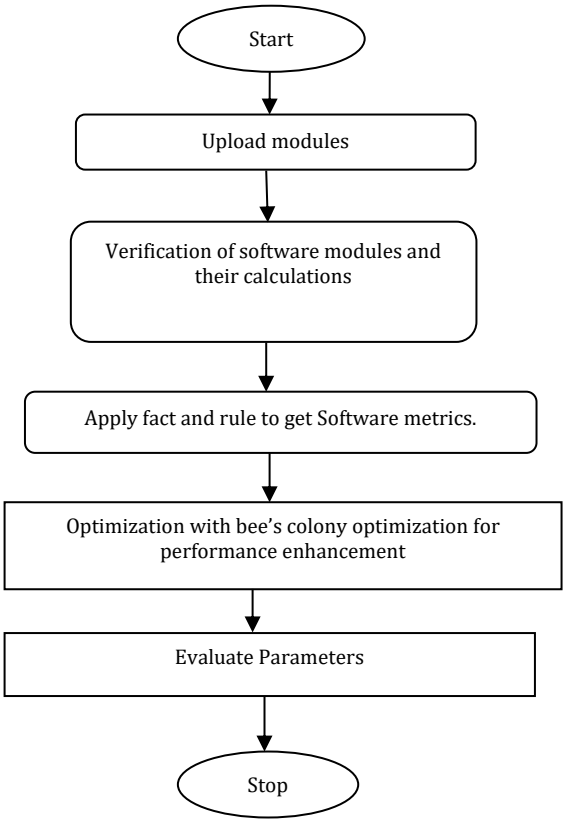

Fig. 1: Flow chart of proposed work

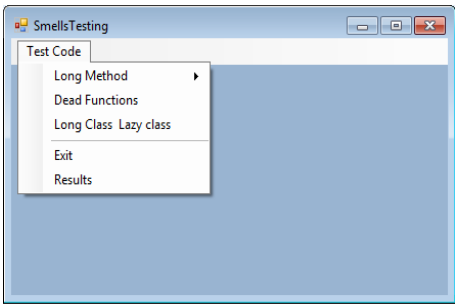

Fig. 2: Test window

The Fig. 3 shows that uploading source code divided into classes and methods. According to an abstract syntax tree; calculated object –oriented matrices: LOC number in a method, total number of variable, used variable, unused variable, Cyclomatic complexity and Halstead efforts. Compare these metrics with detection rules and threshold value.

Result occurred in rule wise. The other button makes correction is used to refactor the code and produce optimal output.

In Fig. 4 shows that, dead code means, remove code i.e., not being second-hand. That's why used source switch systems. The proposed approach is detecting the dead code from a large code module very efficiently. It shows the code from all the classes in the project as shows in the code. The proposed approach is also able to work on the whole project at a time.

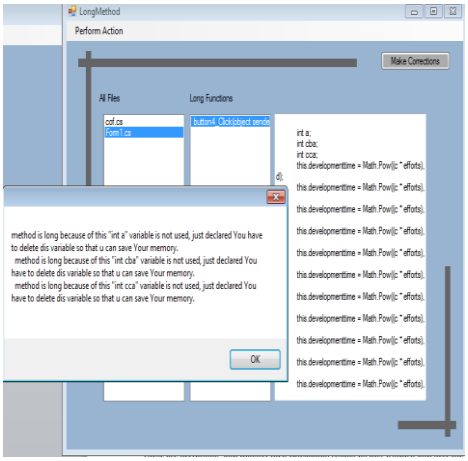

Fig. 3: Upload source code

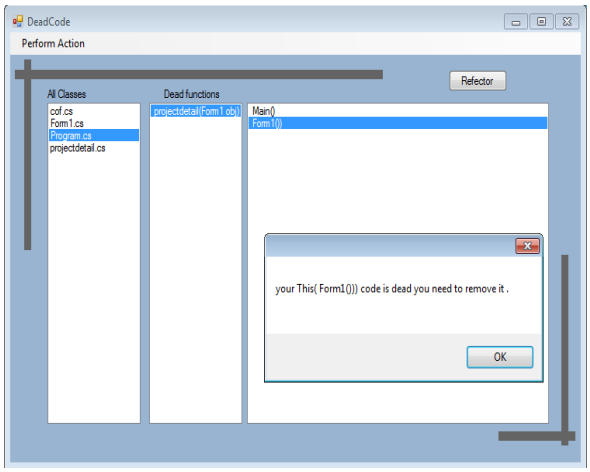

Fig. 4: Types of code

In Fig. 5 described that lazy classes should particularly requesting information from exacting source their heaviness. Each included class enhances the difficult of a project. A number of the method ==0, LOC<=300 and weighted method count or no. of method <=2.

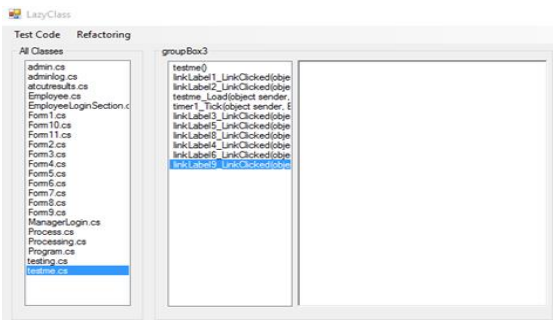

Fig. 5: Lazy class

The Fig. 6 shows the parameter calculation for processing code modules. The performance of the proposed approach is better as compared to the existing approach in terms of various bad smells detection and refectoring. Performance matrixes for this are shown below.

As in the Table 2 the features of proposed and existing approach are compared. The existing system is able to work with only on java. But as shown in Table 2 the proposed algorithm having capability to work with three different languages. net, java and c++. The other enhancement is the existing method is having capability to find and refector the long method bad smell but in proposed enhancement the proposed method working with four types of bad smells with refactoring of them. The proposed method is the proposed method is more accurate and performing multiple testing tool's functionality in a single program.

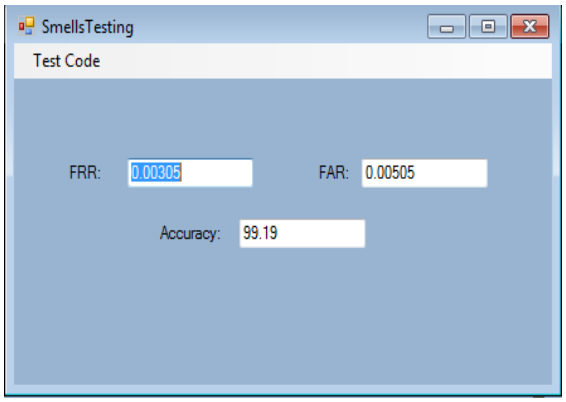

Fig. 6: Performance parameters

The proposed approach is tested on various code modules from java. C++ and .net programming. Because every programming language is having their own syntax to write a program so this process id become little difficult to detect the various bad smells from a coding module. The proposed algorithm is working on the basis of fact and rules programming to achieve this challenge for all the programs.

Other parameters like accuracy, FRR, FAR are also calculated for evaluation of performance in terms of detection and refactoring various smells from a code.

The performance accuracy in proposed approach is maximum in the proposed graph as calculated from the various test modules (Fig. 7 and Table 3). The proposed approach is performing better than existing in all the cases as it is shown in the graph for software testing.

FRR is a false reject rate of an algorithm when it works with real time tasks (Fig. 8). The high rate of FRR causes less accuracy.

As in the proposed parameters the FRR is stable at 0.003 so it can show accuracy more than 98 as in the accuracy graph. The values in for proposed graphs are shown in the matrix below (Table 4).

The False acceptance rate is also used to calculate the performance of an algorithm (Fig. 9). Here on the graph the value of FAR is stable or below 0.005. As in the result the accuracy goes higher in overall results. The fewer acceptances of unwanted samples in test cases are optimizing the results of FAR in software testing. The result table of all these values is shown Table 5.

Table 2: Comparison of working features between base and proposed work

Bad Smells detection	Base	Proposed	.net	Java	C++	Refectoring Proposed	Refectoringbase
LM	Y	Y	N	Y	N	Y	Y
LC	N	Y	Y	Y	Y	Y	N
LoC	N	Y	Y	Y	Y	Y	N
DC	N	Y	Y	Y	Y	Y	N

BSD-Bad Smell Detection, LM-Long Method, LC-Lazy Class,LoC-Long Class, DC-Dead Code, Y-Yes, N-No

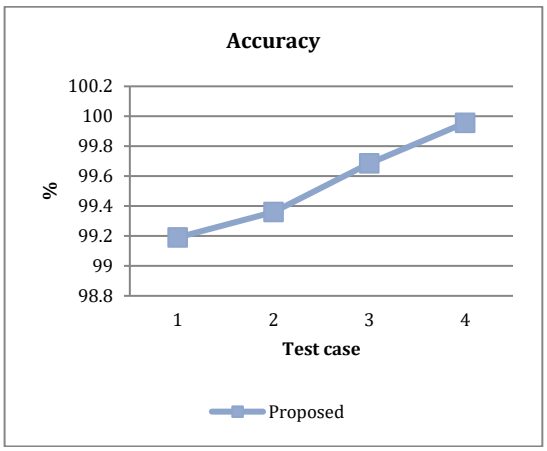

Fig. 7: Accuracy in proposed work

6. Conclusion and future scope

The proposed approach works with three different languages and detects the various bad smells from the code. The approach does refactoring for c++ and Java programming languages along with detection of code smell in .net projects. The proposed hybrid approach detects bad smells as long method, lazy class, dead code, long class from uploaded projects and refactors some of them. The proposed algorithm performs both detection and optimization for all the cases to enhance the output of these parameters.

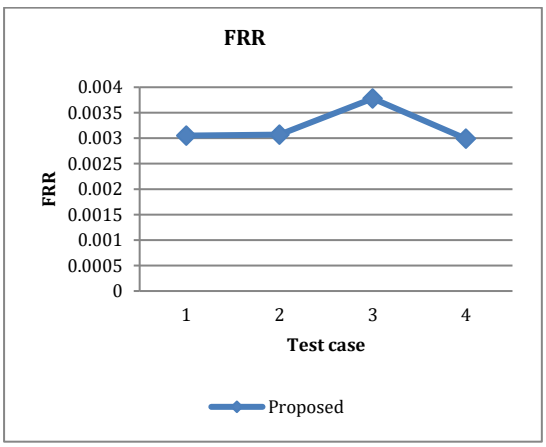

Fig. 8: False rejection rate

Table 3: Accuracy in proposed work

	1	2	3	4
Proposed	99.19	99.36	99.685	99.956

Table 4: False rejection rate in proposed work

	1	2	3	4
Proposed	0.00305	0.00307	0.00378	0.00299

The overall output of proposed approach has been better than other existing approaches and also performs better detection than existing tool in terms of programming languages.

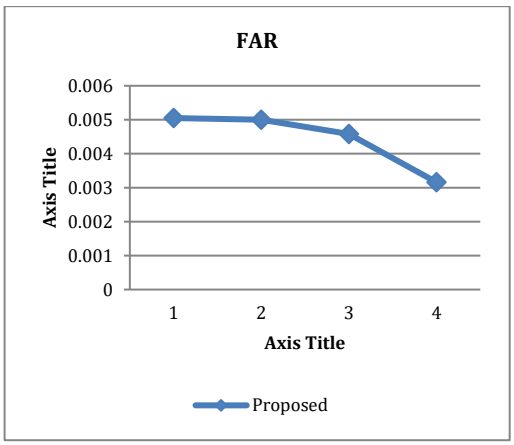

Fig. 9: False acceptance rate

Table 5: False acceptance rate

	1	2	3	4
Proposed	0.00505	0.005	0.00458	0.00316

In future, the performance can be optimized through some other optimization algorithms like Genetic algorithm and refactoring of bad smells for some other parameters of code can be done. The performance of proposed algorithm can also be enhanced through adding some other bad smells in the code as lots of other parameters are in the code smell testing.

References

Bakota T, Ferenc R, Gyimothy T, Riva C, and Xu J (2006). Towards portable metrics-based models for software maintenance problems. In the 22nd IEEE International Conference on Software Maintenance, IEEE, Philadelphia, USA: 483-486. https://doi.org/10.1109/ICSM.2006.69

Chatzigeorgiou A and Manakos A (2010). Investigating the evolution of bad smells in object-oriented code. In the 7th International Conference on Quality of Information and Communications Technology, IEEE, Porto, Portugal: 106-115. https://doi.org/10.1109/QUATIC.2010.16

Counsell S, Hierons RM, Hamza H, Black S, and Durrand M (2011). Exploring the eradication of code smells: An empirical and theoretical perspective. Advances in Software Engineering, 2010: Article ID 820103, 12 pages. https://doi.org/10.1155/2010/820103

dos Reis JP, e Abreu FB, and Carneiro GDF (2016). Code smells incidence: Does it depend on the application domain?. In the 10th International Conference on the Quality of Information and Communications Technology, IEEE, Lisbon, Portugal: 172-177. https://doi.org/10.1109/QUATIC.2016.044

Fokaefs M, Tsantalis N, and Chatzigeorgiou A (2007). Jdeodorant: Identification and removal of feature envy bad smells. In the IEEE International Conference on Software Maintenance, IEEE, Paris, France: 519-520. https://doi.org/10.1109/ICSM.2007.4362679

Fowler M and Beck K (1999). Refactoring: improving the design of existing code. Addison-Wesley Professional, Boston, Massachusetts, USA.

Hazelwood K and Smith MD (2003). Generational cache management of code traces in dynamic optimization systems. In the 36th annual IEEE/ACM International Conference on Microarchitecture, IEEE, San Diego, USA. https://doi.org/10.1109/MICRO.2003.1253193

Kim J and Oh S (1997). EM-code optimization algorithm using tree pattern matching. In the International Conference on Information, Communications and Signal Processing, IEEE, Singapore, Singapore, 2: 917-923. https://doi.org/10.1109/ICICS.1997.652113

Munro MJ (2005). Product metrics for automatic identification of" bad smell" design problems in java source-code. In 11th IEEE International Symposium Software Metrics, IEEE, Como, Italy: 15-15. https://doi.org/10.1109/METRICS.2005.38

Palomba F, Bavota G, Di Penta M, Oliveto R, De Lucia A, and Poshyvanyk D (2013). Detecting bad smells in source code using change history information. In the IEEE/ACM 28th International Conference on Automated Software Engineering, IEEE, Silicon Valley, USA: 268-278. https://doi.org/10.1109/ASE.2013.6693086

Pérez J, Murgia A, and Demeyer S (2013). A proposal for fixing design smells using software refactoring history. In RefTest 2013: International Workshop on Refactoring & Testing. Universitas Antwerpen Antwerpen, Belgium.

Tsantalis N, Chaikalis T, and Chatzigeorgiou A (2008). JDeodorant: Identification and removal of type-checking bad smells. In 12th European Conference on Software Maintenance and Reengineering, IEEE, Athens, Greece: 329-331. https://doi.org/10.1109/CSMR.2008.4493342

Van Emden E and Moonen L (2002). Java quality assurance by detecting code smells. In the 9th Working Conference on Reverse Engineering, IEEE, Richmond, USA: 97-106. https://doi.org/10.1109/WCRE.2002.1173068

Vidal SA, Marcos C, and Díaz-Pace JA (2016). An approach to prioritize code smells for refactoring. Automated Software Engineering, 23(3): 501-532.