2019 IEEE Jordan International Joint Conference on Electrical Engineering and Information Technology (JEEIT)

Code Smells Analysis Mechanisms, Detection Issues, and Effect on Software Maintainability

Mohammed Lafi
Department of Software Engineering
AL- Zaytoonah University of Jordan
Amman, Jordan
lafi@zuj.edu.jo

Joseph Wassily Botros
Department of Software Engineering
Alzaytoonah University of Jordan
Amman, Jordan
jwasily@hotmail.com

Hamzah Kafaween
Department of Software Engineering
Alzaytoonah University of Jordan
Amman, Jordan
hamzahtk@hotmail.com

Ahmad Bassam Al-Dasoqi
Department of Software Engineering
Alzaytoonah University of Jordan
Amman, Jordan
ahmadbassam1995@yahoo.com

Abdelfatah Al-Tamimi
Department of Software Engineering
Al-Zaytoonah University of Jordan
Amman, Jordan
drtamimi@zuj.edu.jo

Abstract— **Software evolution is an inevitable need in most of the modern businesses, software that doesn't accommodate changes is hard to survive the market needs. Also, software changes can affect the overall design of the software and sometimes in a corrupting way, affecting the maintainability and evolvability of the software, which introduces technical debt that needs to be solved by continuous refactoring and restructuring of software. Code smells are useful indicators to identify the parts of the code to be refactored to improve the overall maintainability of the software. We present an overview of software code smells, detection and analysis mechanisms and difficulties. Also, we address the effect of refactoring on software maintainability and error-proneness of software.**

Keywords— Code Smells, Refactoring, Design Patterns, Technical Debt

I. Introduction

Software code undergoes many changes after its delivery for many different reasons, however if these changes weren't conducted properly then they're usually accompanied with deterioration of software quality leading to software that is very hard to maintain and read.

Software maintenance activities that cause the modification of software product after its delivery can be categorized into four types: adaptive, perfective, corrective and preventive.

Most of these maintenance activities are inevitable through the course of software production. However, some changes sometimes are accompanied with a decline in quality by introducing problematic design structures and code smells, which leads to the increase of the technical debt and the need for future corrective, preventive maintenance activities.

To Avoid that, Code smells must be detected early by exposing and removing them through refactoring which is the process of restructuring the program without changing the external behavior to eliminate code smells and to make sure that any further development is possible.

Many approaches have been proposed for automating the process of code smells detection. Some of these approaches use combination of object-oriented metrics such as Line of Code (LOC), Coupling Between Object (CBO), Depth of Inheritance Tree (DIT), with variety of thresholds values to detect code smells in code.

The Rest of the paper is organized as follows code smells definition is given in section II, code smells effect on defects and maintainability is given in section III, code smells detection mechanisms are discussed in section IV, the conclusion is given in section V.

II. Background

Software development Community is always interested in the quality of the developed software artifacts, from the beginning of software engineering principles formulation and the famous critical analysis by Dijkstra for the GOTO statement [1] and its effect on the overall structure of code, leading to the creation of an unmaintainable type of software.

Software engineering established the needed constraints to create a usable software that is resistant to continuous changing. By defining the process of development and the needed activities to improve the maintainability of software

Code smells term were first coined by Riel [2] and Brown et al. [3]. To refer to any symptom for bad designed parts of code that can cause serious problems while maintaining the software.

Fowler et. al. defined the code smells formally as certain structures in the code that suggests (sometimes they scream for) the possibility and need for refactoring [4].

Fowler et. al. [4] originally provided a set of most common code structures that should be avoided or refactored (restructured) if found, they gave informal descriptions of twenty-two code smells and the associated refactoring strategies that are needed to eliminate them, those code smells are language-independent but mostly targets object-oriented paradigm-based languages.

Mäntylä et al [5] formulated another definition for code smells as a term that refers to a somewhat a subjective indicator of poor design or coding style in specific parts of the code and categorized the 22-code smell into five groups according to each characteristic for better understanding and evaluation (Table I).

TABLE I. CODE SMELLS TAXONOMY [5]

Group	Smells in group
Bloaters	Long Method, Large Class, Primitive Obsession, Long Parameter List, Data Clumps.
Object-Orientation Abusers	Switch Statement, Temporary Field, Refused Bequest, Alternative Class with Different Interface.
Change Preventers	Divergent Change, Shotgun Surgery, Parallel Inheritance, Hierarchies.
Dispensables	Lazy Class, Data Class, Duplicate Code, Dead Code, Speculative Generality.
Couplers	Feature Envy, Inappropriate Intimacy, Message Chains, Middle Man.

III. CODE SMELLS EFFECTS

A. *Effect of code smells on Software defects:*

Many researchers studied the relationship between code smells and design flaws in the software, other researches focused on empirical investigations of the effect of code smells on maintenance effort and future software defects.

Li et al. [12] Investigated the correlation between code smells and number of software defects by observing defects in industrial-strength systems and concluded that the presence of code smells namely the shotgun surgery code smell was positively associated with the number of software defects in the classes

Monden et. al.. [13] performed an analysis of legacy systems written in COBOL and concluded that cloned (duplicated) code is more reliable (less prone to errors) but requires more effort for maintaining than not cloned code.

Rahman et. al. [8] found similarly that cloned code is less prone to errors.

Olbrich et al. [11] reports that smelly structures can contain more than defects than other kinds of code structures.

B. *Effect of Code Smells on Software Maintenance:*

Researchers conducted studies to show the empirical evidence of code smells effect on maintainability of the system and empirical analysis of the effectiveness of code smells refactoring on maintainability the system

Marinescu [14] defined three factors when quantifying the effect of code smells (design flaws):

a) Influence: How strongly the design flaw affects the good quality according to Coad and Yourdon's four criteria of good quality (i.e low coupling, high cohesion, moderate complexity and proper encapsulation) using three-level scale (high, medium, low).

b) Granularity: a flaw that affects a method has a smaller impact on the overall quality than the flaws that affect the classes

c) Severity: the first two factors refer to design flaw types, which means that all evaluated smells are considered the same type, however this factor is concerned with evaluating each code smell using severity score (low 1 to high 10)

B'an and Ferenc [15] found out by studying 228 subject systems antipatterns and relationship with overall maintainability of the system using ColumbusQM model that the more antipatterns (code smells) a system contains the less maintainable it will be, meaning that it will most likely cost more time and resources to execute any changes

IV. CODE SMELLS DETECTION

Detecting code smells is a very important and exhausting task, however Fowler et. al. [4] didn't provide any mechanism to detect code smells using any metrics other than subjective evaluation of the code and indicators-based search method, which requires manual review process to find the described code smells.

Fontana et. al. [6] proposed a metrics-based approach that uses a set of metrics (CK metric suite) at code level to detect using specific thresholds the existence of code smells.

Sharma and Spinellis [7] summarized the approaches in literature to detect code smells under five categories of smells detection:

a) Metrics-Based smell detection

b) Rules/Heuristic-based smell detection

c) History-Based smell detection

d) Machine-Learning based smell detection

e) Optimization Based smell detection

Rasool et.al [16] classified detection mechanisms into

a) Dynamic source analysis: Examines the cause-effect relationship of smells during execution

b) Static source analysis: Technique that examines the properties of smells & their impact without executing the system

And then divided the *static source analysis* classification into:

a) Behavioural

b) Empirical

c) Methodological

d) Linguistic

e) Algorithm-based analysis

All detecting processes in the literature studied uses one of the following methods:

a) Automated Based assessment

b) Semi-Automated Based assessment

c) Manual Subjective asssessment

But still "There are many smells that cannot be detected by the currently available metrics alone. For example, we cannot detect rebellious hierarchy, missing abstraction, cyclic hierarchy and empty catch block smells using commonly used metrics." [7]

Even with the literature and market advancement in providing solutions that employ an automated based approach that reduce the amount of time /expertise needed to conduct a thorough evaluation of software code base, still there are limitations regarding the evaluation of the result of such tools and taking the right refactoring decisions.

Mansoour et. al. [9] found some *issues* in code smells detection using metrics which can be summarized as the following:

a) Threshold problem: An appropriate threshold value is not trivial to be defined.

b) Metrics combination: Manual selection of the best combination of metrics that formalize some symptoms of code-smells is challenging.

c) Definitions translation: Translation of code-smell definitions into metrics is not straightforward.

d) Imprecise Metrics: Same metrics can identify different code smells which undermines precision.

e) False positives: detection of well-designed parts of the code as smelly structures.

Fontana et.al. [10] also found that the interpretation of threshold values and measures is challenging across different tools. the research also observed the results from using different tools with different detection mechanism will always differ, potentially because of the discrepancy of threshold and metrics definitions.

Kessintini et. al. [17] proposed a search-based defects detection by example, which is an approach that uses knowledge from previously manually inspected projects in order to detect design defects by generating new detection rules based on the combination of software quality metrics (Fig.1).

The rules generation is a combinatorial optimization problem. And since the number of possible combinations quickly becomes huge with the increase of quality metrics using *deterministic search* is not practical, hence the use of heuristic search.

The Searching Algorithms that were used by Kessintini:

a) Harmony Search (HS)

b) Particle Swarm Optimization (PSO)

c) Simulated Annealing (SA)

The Kessintini search-based method was simulated on 2 open source projects to detect 3 code smells *Blob, Spaghetti code, Functional decomposition.*

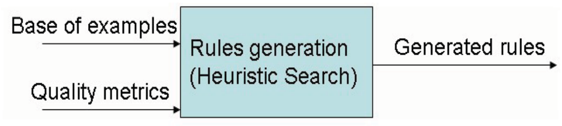

Fig. 1. Search-based by example approach [17]

Sahin et.al. [18] proposed tackling the problem as a Bi-Level optimization problem (BLOP) (Fig. 2) and proposed a two-level method for solving BLOP, evaluation for the method was conducted on seven code smells detection on nine open source projects with the result of 88% precision and more than 90% of recall (Table 2).

Mansour et. al. [9] proposed a multi-objective search-based approach for the generation of code-smells detection rules from code-smells and well-designed examples, aiming at finding the combination of metrics that:

a) Maximize the coverage of a set of code-smells examples collected from different systems.

b) Minimize the detection of examples of good design practices (false positives)

The approach was evaluated by using it on seven open source projects to detect three different code smells (Blob, Spaghetti code, Functional decomposition) and successfully detected most of the expected code smells with an average of 86% of precision and 91% of recall (Table 2).

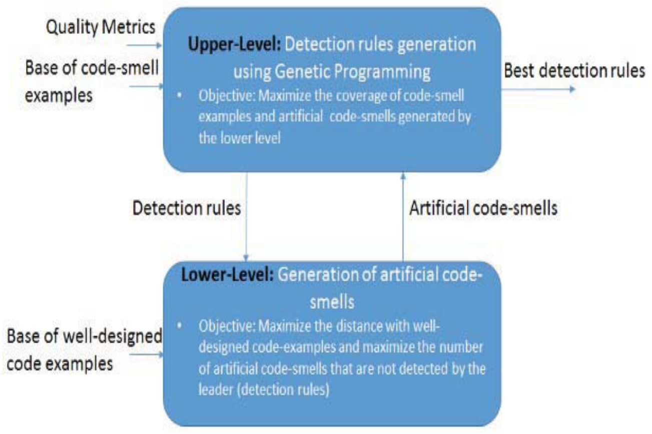
Quality Metrics
Base of code-smell examples
Upper-Level: Detection rules generation using Genetic Programming
Objective: Maximize the coverage of code-smell examples and artificial code-smells generated by the lower level
Best detection rules
Detection rules
Artificial code-smells
Lower-Level: Generation of artificial code-smells
Base of well-designed code examples
Objective: Maximize the distance with well-designed code-examples and maximize the number of artificial code-smells that are not detected by the leader (detection rules)

Fig. 2. Bi-Level Optimization approach [18]

TABLE II. COMPARISON OF CODE SMELLS DETECTION APPROACHES

Research	Detection Mechanism	Algorithm / Method	Avg. Precision	Avg. Recall
Kessintini et. al [17]	Optimization-Based	Examples Search-Based	94%	97%
Sahin et.al. [18]	Optimization-Based	BILevel Optimization	88%	90%
Mansour et. al. [9]	Machine-Learning Based	Multi-Objective Search-Based	86%	91%

V. DISCUSSION & ANALYSIS:

Many researchers proposed algorithms to tackle the difficulties of code smells detection. Code smells can be detected in software code through many techniques such as manual review, machine-learning, and optimization-based approached. Manual review of the code is the most basic and efficient but this approach is not productive and cannot be scaled.

To automate code smells detection process, the researchers depend on specific algorithms to detect code smells using combination of metrics. But, finding the optimal combination of metrics to cover all different types of software is impossible because the detection problem domain has many variables that cannot be covered by metrics indicators only.

Therefore, machine-learning, optimization-based approaches, and recommendation systems [18], [19], [20] are proposed to help tackle this difficulty. We surveyed three of these algorithms (Table 2), and we found that the empirical evidence show that such algorithms are performing well in detecting smells compared to metrics-based solutions such as DÉCOR.

However, machine learning algorithms depends on the *availability* and *accuracy* of training data and lack of such training data can be challenging and results in inefficient algorithms [19] and it is still unknown in literature if machine learning algorithms can scale to cover all known code smells [7].

Also, optimization algorithms depend on optimizing the metrics combination and their threshold values can limit the algorithms in the same way as metrics-based algorithms [7].

VI. Conclusion:

In this paper, we have analyzed the harmfulness of code smells as design defects that adversely affect the software maintenance and quality of software. We surveyed the effect of code smells on both software error-proneness and maintainability.

We discussed the code smells detection problems and concentrated on the issue of finding the optimal metrics combination for detecting code smells. We surveyed some of the researches that have been recently conducted to solve the problem of finding the optimal metrics combination for code smell detection among the set of candidate ones.

We also discussed three of machine-learning and optimization-based approaches to tackle this problem and their overall precision and recall percentages.

References

[1] E. W. Dijkstra, “Letters to the editor: go to statement considered harmful,” Communications of the ACM, vol. 11, no. 3, pp. 147–148, Mar. 1968.

[2] A. J. Riel, Object-oriented design heuristics. Reading, Mass: Addison-Wesley Pub. Co, 1996.

[3] W. J. Brown, Ed., AntiPatterns: refactoring software, architectures, and projects in crisis. New York: Wiley, 1998.

[4] M. Fowler and K. Beck, Refactoring: improving the design of existing code. Reading, MA: Addison-Wesley, 1999.

[5] M. V. Mäntylä and C. Lassenius, “Subjective evaluation of software evolvability using code smells: An empirical study,” Empirical Software Engineering, vol. 11, no. 3, pp. 395–431, Jul. 2007.

[6] F. Arcelli Fontana, P. Braione, and M. Zanoni, “Automatic detection of bad smells in code: An experimental assessment.,” The Journal of Object Technology, vol. 11, no. 2, p. 5:1, 2012.

[7] T. Sharma and D. Spinellis, “A survey on software smells,” Journal of Systems and Software, vol. 138, pp. 158–173, Apr. 2018.

[8] F. Rahman, C. Bird, and P. Devanbu, “Clones: What is that smell?,” (:unav), May 2010.

[9] U. Mansoor, M. Kessentini, B. R. Maxim, and K. Deb, “Multi-objective code-smells detection using good and bad design examples,” Software Quality Journal, vol. 25, no. 2, pp. 529–552, Jun. 2017.

[10] D. P. Tegarden, S. D. Sheetz, and D. E. Monarchi, “A software complexity model of object-oriented systems,” Decision Support Systems, vol. 13, no. 3–4, pp. 241–262, Mar. 1995.

[11] S. M. Olbrich, D. S. Cruzes, and D. I. K. Sjoberg, “Are all code smells harmful? A study of God Classes and Brain Classes in the evolution of three open source systems,” (:unav), Sep. 2010.

[12] W. Li and R. Shatnawi, “An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution,” Journal of Systems and Software, vol. 80, no. 7, pp. 1120–1128, Jul. 2007.

[13] A. Monden, D. Nakae, T. Kamiya, S. Sato, and K. Matsumoto, “Software quality analysis by code clones in industrial legacy software,” in Proceedings Eighth IEEE Symposium on Software Metrics, Ottawa, Ont., Canada, 2002, pp. 87–94.

[14] R. Marinescu, “Measurement and quality in object-oriented design,” in 21st IEEE International Conference on Software Maintenance (ICSM’05), Budapest, Hungary, 2005, pp. 701–704.

[15] D. Bán and R. Ferenc, “Recognizing Antipatterns and Analyzing Their Effects on Software Maintainability,” in Computational Science and Its Applications – ICCSA 2014, vol. 8583, B. Murgante, S. Misra, A. M. A. C. Rocha, C. Torre, J. G. Rocha, M. I. Falcão, D. Taniar, B. O. Apduhan, and O. Gervasi, Eds. Cham: Springer International Publishing, 2014, pp. 337–352.

[16] F. Sabir, F. Palma, G. Rasool, Y.-G. Guéhéneuc, and N. Moha, “A systematic literature review on the detection of smells and their evolution in object-oriented and service-oriented systems,” Software: Practice and Experience, vol. 49, no. 1, pp. 3–39, Jan. 2019.

[17] M. Kessentini, H. Sahraoui, M. Boukadoum, and M. Wimmer, “Search-Based Design Defects Detection by Example,” in Fundamental Approaches to Software Engineering, vol. 6603, D. Giannakopoulou and F. Orejas, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 401–415

[18] B. Hawashin, and A. Mansour, “An Efficient Agent-Based System to Extract Interests of User Groups,” Proceedings of the World Congress on Engineering and Computer Science ,vol.1 , 2016.

[19] A. M. Mansour, M. A. Obaidat, and B. Hawashin, “ Elderly people health monitoring system using fuzzy rule based approach”, International Journal of Advanced Computer Science and Applications (IJACSA), 6(12), 2014.

[20] B. Hawashin, A. M. Mansour, T. Kanan, and F. Fotouhi, “An efficient cold start solution based on group interests for recommender systems”, Proceedings of the First International Conference on Data Science, E-learning and Information Systems, pp. 26, October, 2018.