International Journal of Computer Science and Software Engineering (IJCSSE), Volume 5, Issue 6, June 2016
ISSN (Online): 2409-4285 www.IJCSSE.org Page: 114-118

A Survey on Bad Smells in Codes and Usage of Algorithm Analysis

Aylin GÜZEL[1] and Özlem AKTAŞ[2]

[1, 2] Dokuz Eylül University, Computer Engineering Department Tınaztepe Campus, Buca İzmir Turkey

[1]*aylin.guzel@ogr.deu.edu.tr*, [2]*ozlem@cs.deu.edu.tr*

ABSTRACT

Bad smells are indications of potential problems in the system. Bad smells in the code reduce the quality of the software. They occur in some cases, such as, wrong analysis, incorrect integration new modules into the system, ignoring the software development principles, writing codes in complex way, etc. Design problems in the codes are seen as a bad smell. This survesy focuses on the definition of bad smell in codes, types of bad smells and occurence reasons. Also, some sorting algorithms periods were compared and their relationships with bad smells in code were explained. Additionally, the relationship between algorithm analysis and bad smells in code has been described. Performances of some sorting algorithms were compared by using runtime calculations. Finally, in this article, comparison of the certain recursive and iterative sorting algorithms was made.

Keywords: *Bad smell, Refactoring, Software Engineering, Algorithm Analysis, Code Review, Good Code, Optimization.*

1. INTRODUCTION

Bad smells in the code must be destroyed for better quality, high-performance, low-cost, re-use, modification, implementation and easy development of software by using Refactoring. Refactoring is simple but has an enormous impact on software quality [1]. Refactoring modifies software to improve its readability, maintainability and extensibility without changing what it actually does. It does not alter the external behavior of the code, yet improves its internal structure. The goal of Refactoring is simply not adding any new functionalities in the software. Some Refactoring methods are basically grouped as follows: Composing Methods, Moving Features Between Objects, Organizing Data, Simplifying Conditional Expressions, Making Method Calls Simpler, Dealing with Generalization.

Some Composing Methods are: Extract Method, Inline Method, Inline Temp, Split Temporary Variable, etc.

Some "Moving Features Between Objects" methods are: Move Method, Move Field, Extract Class, Hide Delegate, Remove Middle Man, etc.

Some "Organizing Data" methods are: Encapsulate Collection, Encapsulate Field, Replace Array with Object, Self Encapsulate Field, Change Unidirectional Association to Bidirectional, Change Bidirectional Association to Unidirectional, etc.

Some "Simplifying Conditional Expressions" methods are: Replace Conditional with Polymorphism, Consolidate Conditional Expression, Decompose Conditional, etc.

Some "Making Method Calls Simpler" methods are: Rename Method, Add Parameter, Remove Parameter, Replace Parameter with Explicit Methods, Preserve Whole Object, Replace Parameter with Method, Hide Method, etc.

Some "Dealing with Generalization" methods are: Pull Up Field, Pull Up Method, Push Down Method, Push Down Field, Extract Subclass, Extract Superclass, Extract Interface, etc.

2. BAD SMELLS IN CODE

In this section, some important bad smells are examined. Code smells are detected with the help of a software engineer's point of view or the software tool and bad smells can be solved manually or using the software tool in Refactoring method. Number of code smells are increasing continuously, yet tool support for detecting bad smells is not enough. Therefore, software engineer's point of view is used for detecting and correcting bad smells in codes commonly.

2.1 Duplicated Code

Duplicated code is a really big problem in terms of having good design. "Duplicated code can be definable as a same code structure in more than one place in application or code." [2] The program will be better by unifying the code smells in an efficiently way.

The dublicated code divided into some categories: same expression in two methods of the same class, the same expression in two identical subclasses etc. The first one can be corrected by using Extract Method and the code can be called from both places. Second one can be eliminated using Extract Method in both classes then Pull Up Field.

2.2 Long Method

The best programs are written by using not long methods. Short methods are important for good Refactoring. The longer procedure makes more difficult to understand the code so, causes the bad smells. Therefore, short the method by using Extract Method approach. This approach aims at more readeable and more clear code for current application [2].

2.3 Large Class

Large Class means a class with a huge amount of variables. In other words, Large class means that a class is trying to do too much unexpected tasks without necessary. Solution of large class problem is to eliminate redundancy in the class itself by using Extract Class or Extract Subclass approach [2].

2.4 Long Parameter List

Parameters should be used only if necessary. Also, using global data is not optimal for programs. Global data was alternative to parameters. Instead of global data or large parameter lists, use objects. Use small parameter lists with an object-oriented programs. Short parameter lists are easy to understand and useage. "If the parameter list is too long or changes too often, rethink dependency structure of code and redesign your own code"[2].

2.5. Shotgun Surgery

"Shotgun surgery is similar to divergent change, yet is the opposite" [2].Make a lot of little changes to a lot of different classes use Move Method and Move Field tto put all the changes into a single class for optimal solution. Inline Class approach is used for to bring a whole bunch of behavior together. "Divergent change is one class that suffers many kinds of changes, and shotgun surgery is one change that alters many classes"[2].

2.6 Divergent Change

"Divergent change occurs when one class is commonly changed in different ways for different
Reasons. Thus, each object is changed only as a result of one kind of change"[2]. For solution, use Extract Class method.

2.7 Data Class

Data classes have fields and getting and setting methods for the fields. Data classes are dumb data keepers. If getting and setting methods used by other classes, try to use Move Method to move behavior into the data class. If you can't move a whole method, use Extract Method to create a method that can be moved. Data classes are approved as a starting point, but to participate as a grownup object, they need to take some responsibility [2].

2.8 Switch Statements

The most common problem of switch statements is duplication. In other words, the same switch statement is found more than one places in the program. If you add a new clause to the switch, you have to find all these switch, statements and change them [2].

2.9 Comments

Refactoring not to say that people shouldn't write comments. "In early stage, comments aren't a bad smell; indeed they are a sweet smell" [2]. But now, supported that commented code used for hiding the bad smells and so the comments are bad smell, because the code is bad. If our code is good enough for reading, re-using, implementing to other systems, there is no need to write any comment into code.

2.10 Lazy Class

Understanding and maintaining classes always costs time and money. So if a class doesn't do enough to earn your attention, it should be deleted. Perhaps a class was designed to be fully functional but after some of the refactoring it has become ridiculously small or perhaps it was designed to support future development work that never got done. For subclasses with few functions, try Collapse Hierarchy. Advantage of this solution is reduced code size and easier maintenance. Ignore this approach when a Lazy Class is created in order to delineate intentions for future development, try to maintain a balance between clarity and simplicity in your code [3].

3. BAD SMELLS IN CODE AND ALGORITHM ANALYSIS

The algorithm is a sequential series of steps applied to solve a problem and bases of computer science. An algorithm is expected to work faster and covering less space in memory [4]. A good algorithm always aims to optimum performance. Algorithm analysis is used to measure algorithm's performance, compare different

algorithms, find answer of questions, such as "Is it possible to be better", "what is the best available solution". Performance of an algorithm is depends on internal (space and time) and external (the size of the input data, the computer speed, the quality of compiler) factors [5].

Bad smell in the code causes a loss of performance in the software. Loss of performance would give rise to increased costs, the motivation disorders; transportability, interchangeability and readability of code that are difficult. Refactoring should be used to improve the performance of the software.

Big-O notation is a mathematical approach used to define the algorithm's performance by looking at the internal details of the algorithm. Big-O notation indicates the growth rate of an algorithm and growth rate is the best indicator of the performance of an algorithm [5].

According to the Refactoring method code smells are detected with the help of a software engineer ’s point of view or the software tool and bad smells can be solved manually or using the software tool. Error is observed in the determination of the bad smell in the code in many studies. In order to overcome this deficiency, for many Refactoring rule, software engineer or software tool whether to determine bad smell in the the code correctly is proposed to be verified by using"Algorithm Analysis" approach. Very long cycles, used too much of nested loop structures, method with extreme parameter and prove that the God Class tire the program. Therefore, "Algorithm Analysis" approach will be used to solve this problem.

In this study, the detection of bad smells in the code has been viewed on some algorithms by calculating algorithm runtime. Bad smell detection and correction defends some rules for some methods such as recursive and iterative algorithms have better performance and also some of them have better code. This study compares some algorithm’s performance using runtime calculate approach. Also, this study investigates which method is better for certain sorting algorithms recursive or iteratives?

The first algorithm is “Bubble Sort Algorithm”. Bubble Sort is a simple sorting algorithm that repeatedly steps through the list to be sorted, compares each pair of adjacent items and swaps them if they are in the wrong order. The pass through the list is repeated until no swaps are needed, which indicates that the list is sorted [3]. Two different solutions such as Recursive Solution and Non-Recursive Solution were examined for Bubble Sort Algorithm. Running with an application written in C # programming language to analyze Bubble Sort Algorithm solutions.

Recursive and Non-Recursive solutions were analyzed by analysis of algorithms. Generally, Bubble Sort complexity is $O(n^2)$ and Bubble Sort works well with either linked lists or arrays $O(n^3)$. Also, Bubble sort does not work optimum with either Recursive or Non-Recursive functions. Due to dynamically changeable linked list, the performances of the software with recursive and non-recursive solution were not optimal in the current state. Bad smells in code were identified with the help of a software engineer perspective and “Algorithm Analysis” differences. ith this approach, performance of the method calculated for the Recursive and Non-Recursive function. As a result, the Recursive solution has been determined as more optimum than Non-Recursive one, after measuring the performance of these methods. The performance of these methods were calculated as;

- Normal method: 0,0203417 seconds.
- Recursive method: 0,011363 seconds.

The second algorithm is “Quick Sort Algorithm” which is a divide and conquer algorithm. Quicksort first divides a large array into two smaller sub-arrays: the low elements and the high elements. Then, quick sort can recursively sort the sub-arrays. Two different solutions such as Recursive Solution and Iterative Solution were examined for Quick Sort Algorithm. Running with an application written in C # programming language to analyze all sorting algorithm solutions.

Quicksort is a comparison sort, so it can sort items of any type for which a "less-than" relation. In efficient implementations it is not a stable sort, meaning that the relative order of equal sort items is not preserved. Quicksort can operate in-place on an array, requiring small additional amounts of memory to perform the sorting. Mathematical analysis of quicksort shows that, on average, the algorithm takes O(n log n) comparisons to sort n items. In the worst case, it makes O(n2) comparisons, though this behavior is not frequent [6]. Respectively, the result of recursive and iterative quick sort algorithm is shown as follows:

- Recursive method: 1,6E-06 seconds.
- Iterative method: 2,4E-06 seconds.

The third algorithm is “Merge Sort Algorithm”, which uses divide and conquer methods. This algorithm is an efficient, general-purpose, comparison-based sorting algorithm. The merge sort works as follows [7]:

i. Divide the unsorted list into n sublists, each containing 1 element (a list of 1 element is considered sorted).
ii. Repeatedly merge sublists to produce new

sorted sublists until there is only 1 sublist remaining. This will be the sorted list.

In sorting n objects, merge sort has an average and worst-case performance of O(n log n). If the running time of merge sort for a list of length n is T(n), then, the recurrence T(n) = 2T(n/2) + n follows from the definition of the algorithm. In the worst case, the number of comparisons merge sort makes is equal to or slightly smaller than (n ⌈log n⌉ - 2⌈log n⌉ + 1), which is between (n log n - n + 1) and (n log n + n + O (log n)). Respectively, the result of recursive and iterative merge sort algorithm is shown as follows:

- Recursive method: 1,2E-06 seconds.
- Iterative method: 1,6E-06 seconds.

Another application program was developed for calculating timing for Insertion Sort, Selection Sort, Bubble Sort, and Quick Sort. This application allows us to test various sizes of arrays [8]. This program uses randomly scrambled numbers. The program aims to demonstrate timing of Sorting Algorithms with an array size change option. In Table 1, the program tests each sorting algorithm with an 10000 array size. The program will calculate timing for each Sorting Algorithm by changing array size in each iteration (Each iteration begins with Selection Sort and ends after Insertion Sort).

Table 1: The array size is 10000 for each Sorting Algorithm.

	Array Size	Sort Time
Selection Sort	10000	1,154 seconds
Bubble Sort	10000	1,092 seconds
Quick Sort	10000	0,297 seconds

After that, the array size set 20000. The result for 20000 items is shown in Table 2.

Table 2: The array size is 20000 for each Sorting Algorithm.

	Array Size	Sort Time
Selection Sort	20000	6,584 seconds
Bubble Sort	20000	8,143 seconds
Quick Sort	20000	0,655 seconds

As a result, each Sorting Algorithm's total timing tested with different array size. The results shown in Table 3:

Table 3: Sorting Algorithm's timing with different array sizes.

Array Size	Selection Sort	Bubble Sort	Quick Sort
10000	1,154	1,092	0,292
20000	6,584	8,143	0,655
23000	6,864	11,513	0,608
25000	6,801	7,114	0,546
5000	0,327	0,265	0,171
2600	0,109	0,093	0,047
30000	9,188	9,828	0,749
50000	25,569	43,914	1,201
67000	138,17	89,747	1,513
100000	210,477	356,384	5,304

4. CONCLUSIONS

This work emphasized that design defects known as bad smells may occur in system analysis, decision-making due to omissions and negligence in the application.
Defects in the code can be determined by deciding through the experience of using a software tool or software engineer. However, at this point, some problems are outstanding. First, there is a possibility of making wrong decisions by software engineers. Second, bad smells detecting and resolving process take so long by using software engineers' perspectives and experiences. One by one detection of bad smells by software engineers will take a lot of time. So, it causes loss of cost and labor. In previous studies, detecting bad smells in the code by using software tools that not contain all of the available Refactoring method, which can detect only a certain part and are seen to be a solution. The most important reason of this condition is that the development rate of software tools and the rate of emergence of new code defects are not at the same or close speed.
In this study, features of previous works have examined in details; detection of bad smells in the code has been tested on some algorithms by evaulating algorithm runtimes. It was seen that code smell detection and correction defends some rules for some algorithms have better performance and thus, some of them have better and more clear code. This study compared basic sorting algorithms' performance using runtime calculation approach. Also, it has been investigated that which method is better for certain sorting algorithms recursive or iteratives in terms of their own performance with the criteria of calculated runtime by using special feature. In addition, timing of Sorting Algorithms tested with different array size and compared.

REFERENCES

[1] A. Chatzigeorgiou, A. Manakos, "Investigating the evolution of code smells in object-orientedsystems", Innovations System Software Engineering, Vol. 10, 2014, pp. 3-18.

[2] M. Fowler, K. Beck, J. Brant, W. Opdyke, D. Roberts, "Refactoring: Improving the Design of Existing Code", Addison-Wesley Professional, 2002.
[3] https://sourcemaking.com/refactoring/smells/lazy-class, [Date Accessed: 3 Jan 2016].
[4] http://bilgisayarkavramlari.sadievrenseker.com/2010/09/24/algoritma-analizi-analysis-of-algorithms/, [Date Accessed: 3Oct 2014].
[5] http://www.erdalguvenoglu.com/veriyapilari/algoritma_analizi.pdf/, [Date Accessed: 3 Oct 2014].
[6] https://en.wikipedia.org/wiki/Quicksort, [Date Accessed: 4 Jan 2016].
[7] https://en.wikipedia.org/wiki/Merge_sort, [Date Accessed: 5 Jan 2016].
[8] https://www.exchangecore.com/blog/c-sharp-sorting-algorithms-performance-comparison-selection-sort-vs-insertion-sort-vs-bubble-sort-vs-quick-sort/, [Date Accessed: 8 Jan 2016].
[9] H. Liu, M. Zhiyi, W. Shao, Z. Niu, "Schedule of Bad Smell Detection and Resolution: A New Way to Save Effort ", IEEE Transactions on Software Engineering, Vol. 38, No. 1, 2012, pp. 220-235.
[10] R. Malhotra, N. Pritam, "Assessment of Code Smells for Predicting Class Change Proneness", Software Quality Professional, Vol 15, No. 1, 2012, pp. 33-40.
[11] K. Mehlhorn, P. Sanders, "Algorithms and Data Structures", Springer, 2008.
[12] T. Mens, T. Tourwe´, "A Survey of Software Refactoring" , IEEE Transactions on Software Engineering, Vol. 30, No. 2, 2004, pp. 126-139.
[13] M. Moha, "Detection and Correction of Design Defectsin Object-Oriented Designs", OOPSLA'07, Canada, 2007, pp. 949-950.
[14] J. Rech, W. Schäfer, "Visual Support of Software Engineers during Development and Maintenance", ACM SIGSOFT Software Engineering Notes, Vol 32, No. 2, 2007.
[15] J. Schumacher, N. Zazworka, F. Shull, C. Seaman,"Building Empirical Support forAutomated Code Smell Detection", ESEM'10, Italy, 2010.
[16] M. Kessentini, R. Mahaouachi, K. Ghedira, "What you like in design use to correct bad-smells", Software Qual J, Vol 21, 2013, pp. 551-571 .

AUTHOR PROFILES:

Aylin GÜZEL was graduated from Bilecik Şeyh Edebali University Department of Computer Engineering in 2013 in degree of BSc. She had lessons from Anadolu University Computer Engineering Department as Farabi Programme student in 2011-2012 Fall and Spring semesters. She is a student in Dokuz Eylul University Department of Computer Engineering MSc. Programme.

Özlem AKTAŞ was graduated from Dokuz Eylul University Department of Computer Engineering Department in 2003 in degree of BSc. She was graduated from the same department in 2005 in a degree of MSc. and in 2010 as Ph.D. She has been working as Assistant Professor in the Dokuz Eylul University Department of Computer Engineering Department since 2010