ISSN (Online) : 0974-5645
ISSN (Print) : 0974-6846
Indian Journal of Science and Technology, *Vol 8(S2), 23-28, January 2015*
DOI: 10.17485/ijst/2015/v8iS2/57796

Sequential Ordering of Code Smells and Usage of Heuristic Algorithm

T. Pandiyavathi [1*] and T. Manochandar [2]

[1]Anna University, Chennai, India; pands1991@gmail.com
[2]VRS College of Engineering & Technology, Arasur, India

Abstract

The process of removing the bad smell results in introduction of new smells due to dependency between the codes in the program. This process increases human effort and time. Automated tools are used for detecting the bad smells in the program. This problem is called as ripple effect and we aim in reducing and removing this effect in the program. We apply refactoring process for reducing the amount of bad smells in the code. Since there exist more number of code smells in the program, we generate a sequence in which the refactoring has to be applied by which the evolution of new bad smells is enormously decreased. The refactoring methods that have to be applied to the source code are also ordered using a heuristic algorithm.

Keywords: Code Smells, Heuristic Algorithm, Refactoring, Ripple Effect

1. Introduction

According to Fowler[1], design problems appear as "bad smells" at code or design level and the process of removing them is called refactoring where the software structure is improved without any modification in the behavior. It can be briefly defined as "Restructuring of internal structure of object oriented software to improve the quality while the software's external behavior remains unchanged" - Fowler[1].

Refactoring improves the design of software and makes software easier to understand. It also helps us to find bugs. Bad smells can be detected using various kinds of automated tools. When the smell is refactored due to dependency there is high possibility of increasing other kind of smell which in turn results in increased effort and time. A smell being resolved may affect the presence of an existing smell or introduces some more conflicts into the system. The design of software systems can exhibit several problems which can be either due to inefficient analysis and design during the initial construction of the software or due to software ageing since quality degenerates with time.

1.1 Proposed Work

In our approach single tool is used for calculating the metrics. We use metrics of the given source code to detect the defects in the source code and refactoring list elements are stored based on the detected smell. This sequence is given as input to an algorithm which gives the proper ordered sequence to perform refactoring which reduces human effort. Complexity of the current approach lies in finding the fitness function based on which the crossover and mutation in the genetic algorithm are done. It is suggested that often manual refactoring will be the most effective one among all the others. Since it increases the time factor, we detect the smells using different strategy and finally apply the sequence of refactoring methods to

* *Author for correspondence*

the code which involves manual checking along with the defect resolution.

2. Literature Survey

Bad smells are detected using various tools. But only limited methods are available for removing them.

2.1 Code Smells

The key issue can be solved by a kind-level scheme that arranges the detection and resolution sequences of different kinds of bad smells. Arranging detection and resolution sequences[5] can be done by analyzing the relationship among different bad smells. Based on the analyzed sequence, smells are detected and resolved using several kind of automated tools like JDeodorant(Feature envy), PMD(duplicate code) based on the type of smell. This greatly minimizes human effort but the tool may miss some bad smells in some cases. Since there exists only 8 important code smells being analyzed, in the proposed system some more smells can be introduced and sequencing is done to improve the quality of the code. Metrics can also be calculated to look into the results of the proposed system. This approach can be evaluated on application in future work for validation. In the proposed system smells namely move method, move field and dead code can be added as additional smell detection and evaluation.

Fowler[1] proposed the concept of bad smells. He proposed and described 22 bad smells in object-oriented systems. They also associated refactoring rules with these bad smells, suggesting how to resolve these bad smells. Bad smells in specific domains have also been proposed. Srivisut and Muenchaisri defined some bad smells in aspect-oriented software, and proposed approaches to detect them. Van Deursen Test Smells indicating problems in test code. The impact of bad smells has also been analyzed.

Lozano[5] assessed the impact of bad smells, i.e., the extent to which different bad smells influence software maintainability. They argued that it is possible to analyze the impact of bad smells by analyzing historical information. With the impact in mind, it is possible to assess code quality by detecting and visualizing bad smells. Van Emden and Moonen[6] implemented a code browser for detecting and visualizing code smells, and assessed the quality of code according to the visual representation. Detecting bad smells is critical and time-consuming. Therefore, automating detection is essential. Tsantalis[7] proposed an approach to identifying and removing type-checking bad smells which is implemented in an prototype tool named JDeodorant. Fokaefs[8] proposed an Eclipse plug-in to identify and resolve feature envy bad smells. Clones, one of the most common bad smells, have been investigated for a long time, and dozens of detection algorithms have been proposed to detect them. Mohamed[2] proposed a language for formalizing bad smells, and a framework for automatically generating detection algorithms for the formalized bad smells.

2.2 Dependency among Bad Smells

Wake classified bad smells into two categories: bad smells within classes and bad smells between classes. Meszaros[10] classified test smells into code smells, behavior smells, and project smells. Mantyla[11] analyzed the correlations among bad smells by investing the frequency with which each pair of bad smells appears in the same module. They found that bad smells within the same category are more likely to appear together. The work aimed to simplify the comprehension of bad smells, instead of refactoring activities.

Pietrzak and Walter[12] investigated the intersmell relationships to facilitate the detection of bad smells. They argued that detected or rejected bad smells might imply the existence or absence of other bad smells.

2.3 Heuristic Algorithm

Genetic algorithm was first *proposed by* Goldberg et al in 1989. In the computer science field of artificial intelligence, a genetic algorithm is a search heuristic that mimics the process of natural selection. This heuristic is routinely used to generate useful solutions to optimization and search problems. To insure the detection of maintainability defects, several automated detection techniques have been proposed by Mohamed[2]. The vast majority of these techniques rely on declarative rule. In

these settings, rules are manually defined to identify the key symptoms that characterize a defect. These symptoms are described using quantitative metrics, structural, and/or lexical information.

Beside the previous approaches, one notices the availability of defect repositories in many companies, where defects in projects under development are manually identified, corrected and documented. However, this valuable knowledge is not used to mine regularities about defect manifestations, although these regularities could be exploited both to detect and correct defects.

2.4 Detailed Work

We propose a method to overcome some of the above-mentioned limitations with a two-step approach based on the use of defect examples generally available in defect repositories of software developing companies:

1. Detection-identification of defects, and
2. Correction of detected defects.

Instead of specifying rules manually for detecting each defect type, or semi automatically using defect definitions, we extract these rules from instances of maintainability defects. This is achieved using Genetic Programming. We generate correction solutions based on combinations of refactoring operations, taking in consideration two objectives:

1. Maximizing code quality by minimizing the number of detected defects using detection rules generated
2. Minimizing the effort needed to apply refactoring operations.

This is a multi-objective approach. In all previous works discussed above, discussion is done based on resolution sequences of bad smells, but no evaluation or discussion is presented.

3. Refactoring

The steps involved in refactoring are as follows,

1. Perform pair wise analysis among each selected code smells
2. Draw directed graph based on the analysis made above
3. Apply topological sorting to obtain ordered code smells
4. Generate detection rules using combinations of metrics and thresholds
5. Collect refactorings methods to be processed after the defect detection
6. Generate/ frame list of possible refactoring methods like pull up, move method, extract method
7. Apply natural evolution techniques like genetic algorithm with input as the outcomes of steps 4,5,6
8. Perform crossover and mutation along with the elitism property in the above algorithm
9. Obtain Optimal solution with sequenced refactoring plans
   a. Sequencing Code Smells
   Study of smells selected for the problem and analyzing its complexities, Pair wise analysis, Generate DAG and Sequence the code smells using topological sorting algorithm.
   b. Detection of Smells Using Automated Tools
   Select fragments of code, Inject smells into the code, Use automated tools to detect the smells in the code where PMD detects dead code, duplicate code, long method, long parameter list and Checkstyle detects feature envy.
   c. Metric Calibration and Refactoring Plans
   It aims on implementing our current ideas on detecting the code smells. We use a tool named “metrics” which is an eclipse plugin to find the metrics in the code which is followed by metrics calibration where the detected/calculated metrics is compared against the threshold values. Initially generate Design defects rules and then generate list of refactoring plans.
   d. Extracting Optimal Refactoring Solution
   Encoding involves conversion of Array to a feasible input value in which the processing is going to be done and Selection involves selecting individuals for the population and finally evaluates the individuals through fitness function.

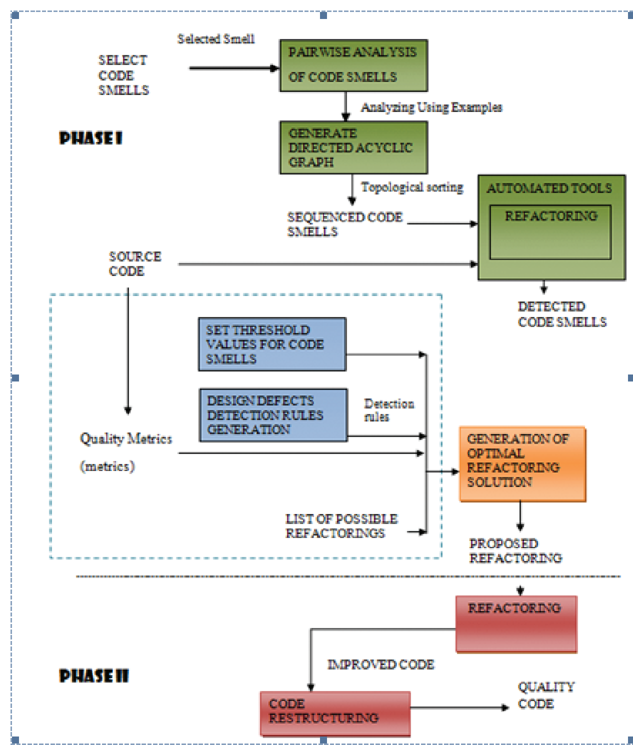

Figure 1. Architecture Diagram.

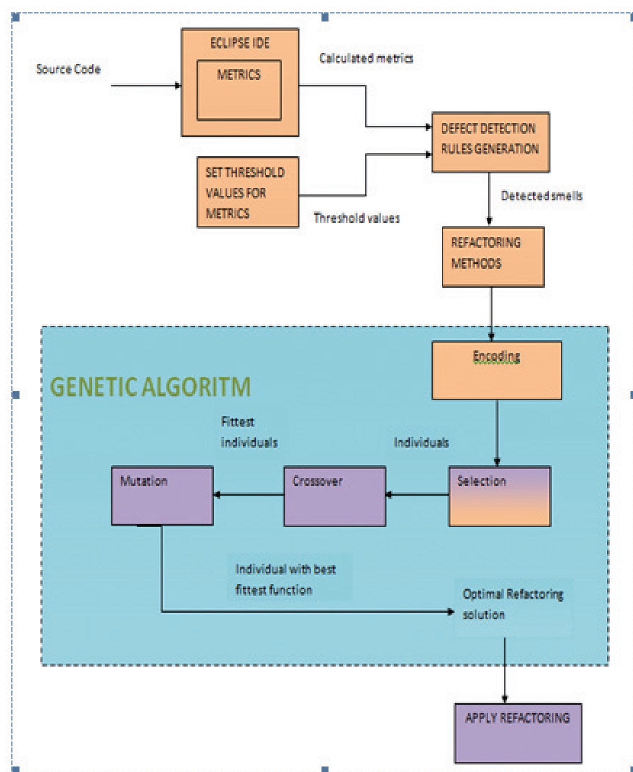

Figure 2. Genetic Algorithm.

Sequenced Code Using Sorting Algorithm:

Dead code→ Duplicate code→Feature envy→long method→ god class→ long parameter list

4. Refactoring with Automated Tools

Inject smells into the code and use automated tools to detect the smells in the code. Tools used here can find the code smells that exist in the code. But not all the code smells are detected using a single tool. There are more than 22 code smells as proposed by Fowler[1] and plug-in can detect not more than 5 smells in a code. The smells taken for study are 5 important code smells and detecting them using tool gives an idea about the code which facilitates in reusing of the code.

5. Optimal Solution

Genetic Algorithms (GAs) are an iterative approach which is described as analogous to evolutionary processes for solving search and optimization problems. We find the individuals and combine them to create a population with higher fitness.

5.1 Crossover Complexity

Problem exists in the algorithm since the input is converted to binary strings, after the computation of the algorithm on the bit strings, due to crossover and mutation the number of strings and the sequence is changed. Illegal results are produced which has been found on latter stage. To avoid this issue partially mapped crossover is used.

6. Discussion

6.1 Amended Techniques

Topological sorting algorithm for sequencing the major selected code smells.

Automated tools to define that the usage of tools has many flaws which has to be solved.

Metric calibration where the metrics calculated from the source code is used for detecting the defects and finding their related refactoring methods from the list of refactoring methods stored in an array list.

Genetic algorithm will encode, select individual, do crossover and mutation and finally produces the optimal solution to the problem.

6.2 Limitations

We have encountered a problem with the illegal child generation in the genetic algorithm. During study it is found that for optimal solution generation using binary encoding is the better way. But this condition holds badly for our solution domain. This resulted in rework of the genetic algorithm by assigned some char values or numbers to the refactoring methods. Problem is therefore analyzed and the crossover technique which produce legal children is found to be partially-mapped crossover technique and later on this technique is been implemented and results are obtained along with the fitness values.

Future work can be done using multi-objective evolutionary algorithm that adapts non-dominated sorting genetic algorithm (NSGA-II)[2]. If smells are introduced, monitoring by itself invokes smell detection tools to inform the developer to resolve the smells. This facilitates instant refactoring decisions being made as soon as the smell is been detected. This solution can reduce the total number of smells by 51 percent.

6.3 Accuracy

71.428.

7. Conclusion

Our work uses genetic algorithm to find an optimal solution to the problem. Some problem exists while doing Encoding and Crossover in genetic algorithm. In previous work, scheduling of the code smell is done. Refactoring of the code smells solely depends on the tool and in case of existence of code smells ever after refactoring leads in increasing the human effort.

```
31204; Fit = 105.62002441935006
02134; Fit = 105.62002441935006
12304; Fit = 103.93666659677497
03214; Fit = 103.93666659677497
12304; Fit = 103.93666659677497
30124; Fit = 105.07115674260575
12034; Fit = 108.75444221231609
32104; Fit = 95.83511235190642
30214; Fit = 108.75444221231609
30214; Fit = 108.75444221231609
32104; Fit = 95.83511235190642
12304; Fit = 103.93666659677497
30124; Fit = 105.07115674260575
30124; Fit = 105.07115674260575
```

Figure 3. Output of Genetic Algorithm.

Table 1. Accuracy Measures

tp(correct result)	fp(unexpected)
6	3
fn(missing)	tn(correct absence of result
1	4

7.1 Graph Representation

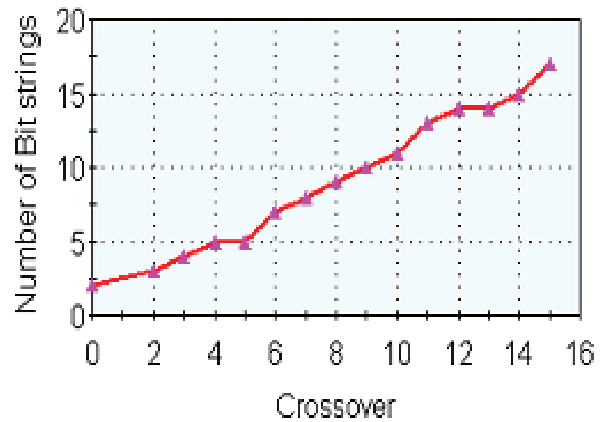

Figure 4. Cause of Illegal Children.

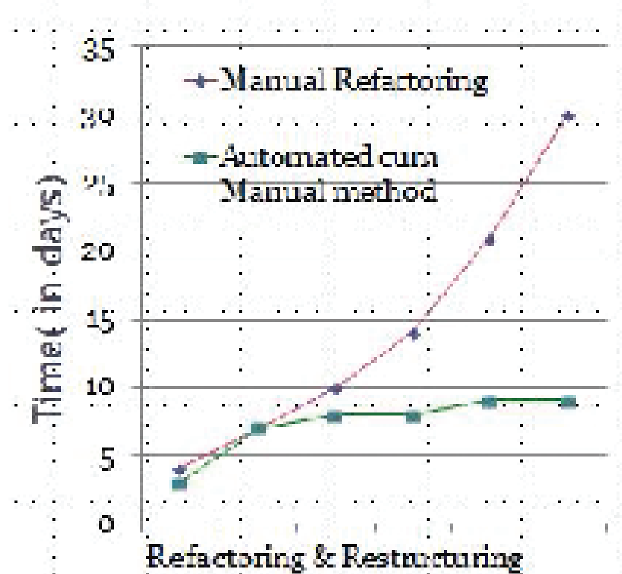

Figure 5. Time Vs Restructuring.

8. References

1. Fowler M, Beck K, Brant J, Opdyke W, Roberts D. Refactoring: Improving the Design of Existing Code. 2002.
2. Ouni A, Kessentini M, Sahraoui H, Hamdi MS. The use of development history in software refactoring using a multi-

objective evolutionary algorithm. Proceedings of GECCO'13. ACM; 2013. p.1461–8.
3. Liu H, Guo X and Shao W. Monitor-Based Instant Software Refactoring. IEEE Transactions of Software Engineering. 2013; 39(8):1112–26.
4. Liu H, Niu Z, Ma Z, Shao W. Identification of generalization refactoring opportunities. Automated Software Engineering. 2013; 20(1):81–110.
5. Lozano A, Wermelinger M and Nuseibeh B. Assessing the Impact of Bad Smells Using Historical Information. Proceedings of Ninth International Workshop on Principles of Software Evolution: In Conjunction with the Sixth ESEC/FSE Joint Meeting; 2007. p. 31–4.
6. van Emden E and Moonen L. Java Quality Assurance by Detecting Code Smells. Proceedings of Ninth Working Conference Reverse Engineering; 2002. p. 97–106.
7. Tsantalis N, Chaikalis T and Chatzigeorgiou A. Jdeodorant: Identification and Removal of Type-Checking Bad Smells. Proceedings of 12th European Conferenced Software Maintenance and Reengineering; 2008 Apr. p. 329–331.
8. Fokaefs M, Tsantalis N and Chatzigeorgiou A. Jdeodorant: Identification and Removal of Feature Envy Bad Smells. Proceedings of IEEE International Conference Software Maintenance; 2007 Oct. p. 519–20.
9. Mantyla M, Vanhanen J and Lassenius C. Bad Smells - Humans as Code Critics. Proceedings IEEE 20th International Conference Software Maintenance; 2004 Sept. p. 399–408.
10. Meszaros G. xUnit Test Patterns: Refactoring Test Code. Addison-Wesley; 2007.
11. Mantyla M, Vanhanen J and Lassenius C. A Taxonomy and an Initial Empirical Study of Bad Smells in Code. Proceedings International Conference Software Maintenance; 2003. p. 381–4.
12. Pietrzak B and Walter B. Leveraging Code Smell Detection with Inter-Smell Relations. Proceedings of Seventh International Conference Extreme Programming and Agile Processes in Software Engineering; 2007 Jun. p. 75–84.
13. Mens T, Taentzer G and Runge O. Analysing Refactoring Dependencies Using Graph Transformation. Software Syst Model. 2009 Sept; 6(3):269–285.
14. Liu H, Yang L, Niu Z, Ma Z and Shao W. Facilitating Software Refactoring with Appropriate Resolution Order of Bad Smells. Proceedings of Seventh Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium the Foundations of Software Engineering; 2009. p. 265–8.
15. Available from: http://www.eclipse.org/downloads
16. Horstmann CS and Cornell G. Core Java Fundamenals. 8th ed. 1.
17. Available from: http://www.Sourceforgenet.com- pmd, checkstyle.
18. Liu H, Ma Z, Shao W and Niu Z. Schedule of Bad Smell Detection and Resolution: A New Way to Save Effort. IEEE Transactions on Software Engineering. 2012 Jan-Feb; 38(1):220–35.
19. Nongpon K. Integrating Code Smells Detection with Refactoring Tool. 2012 Aug.
20. Griffith ID. Automated refactoring: a step towards enhancing the comprehensibility of legacy software systems.