Understanding the Evolution of Code Smells by Observing Code Smell Clusters

Ahmad Tahmid*, Nadia Nahar† and Kazi Sakib‡
Institute of Information Technology, University of Dhaka, Bangladesh
**bit0332@iit.du.ac.bd, †bit0327@iit.du.ac.bd, ‡sakib@iit.du.ac.bd*

***Abstract*—Code smells are more likely to stay inter-connected in software rather than remaining as a single instance. These code smell clusters create maintainability issues in evolving software. This paper aims to understand the evolution of the code smells in software, by analyzing the behavior of these clusters such as size, number and connectivity. For this, the clusters are first identified and then these characteristics are observed. The identification of code smell clusters is performed in three steps - detection of code smells (God Class, Long Method, Feature Envy, Type Checking) using smell detection tools, extraction of their relationships by analyzing the source code architecture, and generation of graphs from the identified smells and their relationships, that finally reveals the smelly clusters. This analysis was executed on JUnit as a case study, and four important cluster behaviors were reported.**

I. Introduction

In a large software, a significant percentage of code smells are inter-connected and co-occurred [1]. Studies, such as [2], showed that a combination of code smells is more difficult to manage compared to a single instance of code smell. In this study, these inter-related smelly code components are referred as smell clusters. These smell clusters can have a serious impact on the manageability and overall quality of software, specially if the software is evolving. Because, as a system grows older, instances of smells in its source code go through a complex process of evolution [3] and often lead to bigger architectural problems known as anti-patterns [4].

Despite the problems due to smell clusters in the evolution of software, it has not gained enough focus in research due to the complex connectivity of smells. Although researchers have focused on evolution of individual smells, to the best of authors' knowledge no study have been conducted on the evolution of the inter-connected smells. Therefore, to achieve greater knowledge about software maintainability, it is needed to observe smell clusters, and understand how these clusters evolve in different versions of the software.

Many studies, such as JSNOSE, Decor, have been conducted on code smell detection and elimination. However, refactoring all the smells is not always a feasible option due to time and budget constraints. Therefore some studies rather focused on identifying consequences of different code smells. Olbrich et. al. [5] studied impact of smells on the change behavior of evolving software in terms of change frequency and size. This study proved that software change history and source control can help understanding the impact of different code smells in software evolution. According to [1], clusters and relation between smells give better insight of software quality. This paper focused on the detection of code smells and in particular on their relations and co-occurrences. However, it did not analyse how these related clusters of code smells change with time. In [3] and [6], the authors showed the evolution process of individual code smells. It was observed that occurrences of specific code smell increase over time. However, it did not explain how the relation between different smells affects this process.

For understanding the relationship between code smells, the behavior of code smell clusters is observed in this paper. Thus, the methodology is to identify the smelly clusters and examine their evolution in different software versions to identify their characteristics. The smelly clusters are identified following three major steps – detecting smelly classes and methods in source code using a smell detection tool (JDeodorant), extracting the relationship between these smelly components from the source code architecture, and generating a descriptive graph using these components as nodes and their relationships as edges to create clusters of the smelly components. Using this process, smell clusters are identified for various versions of software leading to the identification of their evolution pattern.

In this study, a case study was conducted to understand how the relationships between smelly components of a source code change with the evolution of software. JUnit was selected for the study. For 10 different versions of JUnit smell clusters were detected. By comparing the clustering properties such as cluster size, cluster count, connectivity etc. for all the versions few characteristics were identified - number of connected code smell clusters increases steadily with time, smell clusters can be categorized in three types (mega cluster, single-node cluster, and small cluster), smelly components tend to create a mega cluster, and the size of the mega cluster increases steadily with time.

II. Methodology

The purpose of this study is to understand how relationships between different code smells evolve with time by observing the behaviors of code smell clusters. First, smelly clusters are identified for different releases of software and their properties are extracted. By comparing these properties, evolution patterns for smell clusters are detected. A greater understanding of code smell relationships is achieved by examining the evolution patterns of smell clusters. An overview of the whole process is shown in Fig. 1.

Identification of smelly clusters is divided into three major tasks. A) Smell Detection – where smelly classes and methods are identified from source code. B) Relationship Identification – where relations between different smelly

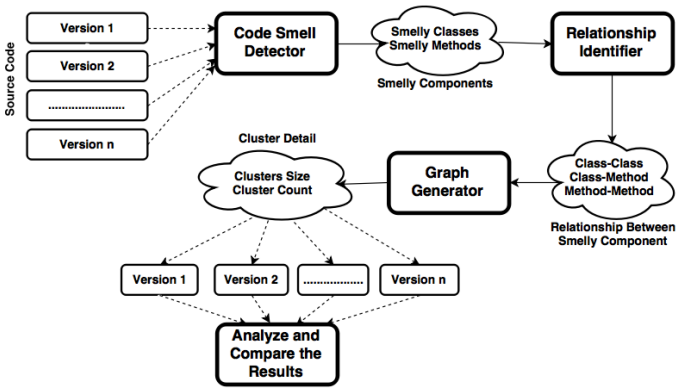

Figure 1: Overview of the Complete Methodology

components (classes or methods containing smells) are extracted. C) Graph Generation – where a graph is generated using the smelly components as nodes and their relationships as edges. This identification process is depicted in Fig. 2.

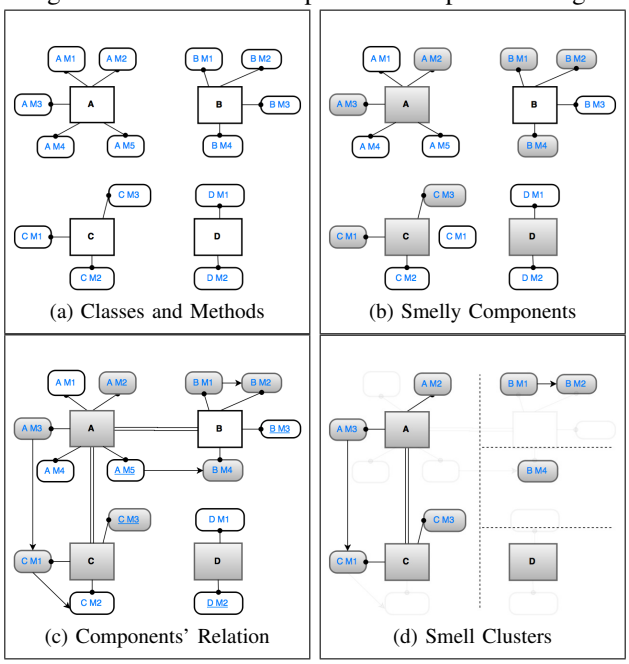

Figure 2: Smelly Cluster Detection Process

In Fig. 2(a), an example set of classes is shown along with methods, contained in those. Here, the square boxes represent classes and the oval-shaped ones represent methods. The relationships between a class and a method is shown as an undirected edge. Fig. 2(b) shows the identified smelly components as gray boxes. Then, in Fig. 2(c) the relationships between the components are depicted. Different relationships are illustrated as different types of edges such as directed edges represent a call from a method to another method, dual edges represent relationship between two classes. Finally, Fig. 2(d) shows the extracted smelly component clusters.

A. Smell Detection

There is no accepted standard for detecting code smells. For example – the exact number of lines needed for a class to be a god class is not defined. According to Fowler, the best way to detect code smells is using human intuition. However, for a large project, detecting code smells manually is highly inefficient and costly. Although the standards of smell detection is not established yet, there are tools such as JSNose, Decor, JDeodorant [7] which are widely used. In this study, JDeodorant is used. Any other tools can also be used as the key concern of this study is not to detect smells but to analyze relationships between those. So the standards of detection do not carry much significance in this case.

JDeodorant supports detection of four smells - God Class (GC), Long Method (LM), Feature Envy (FE), Type Checking (TC). These four types of smells are detected in each version of software. As GC and FE are class level smells, classes affected by these are identified. Similarly, as LM and TC are method level, the smelly methods are identified.

B. Relationship Identification

Different relationships exist between smelly classes and methods. As defined in [1], types of these relationships are:-

- Class-Class: Class using or being used by another class
- Class-Method: A class containing a method or a method contained in a class
- Method-Method: A method, calling or being called by another method.

These relationships are identified in two steps. The first step is to analyze software source code and extract its architecture. In the next step, JCallGraph is used to identify the above stated relationships by extracting software callgraph.

C. Graph Generation

If smelly components are considered as nodes and relationships between those are considered as edges, a graph can be generated. In this graph, different types of edges represent class-class, class-method, method-method relationships.

After graph generation, clusters are identified using BFS algorithm from JGraph library. Various characteristics of smells such as clustering behavior, connectivity, change of cluster size, are extracted by analysing the clusters.

Using this process, smell clusters are identified for various versions of software. Then, these clusters are analyzed to identify the evolution of the code smell relationships.

III. Case Study on JUnit

The goal of this case study is to understand how the relationships between the smelly components of a system change with the evolution of software. JUnit was selected for this study as it is one of the most popular libraries in Java. Code smells and the relationships between smells were extracted from all the versions of JUnit's source code. A graph was generated for each version where every node represented a smelly component and every edge represented a relation between two smelly components. From these

graphs various clustering information such as average cluster size, connection count, cluster count were extracted.

The analysis was performed on 10 different releases of JUnit (3.8.2-4.12), published during December 2004 to December 2014. Each step of the cluster identification on these versions are described in the following sections.

A. Smell Detection

This study employed JDeodorant (5.0.42) for identification of code smells. Using the output of JDeodorant, the number of smelly classes and smelly methods were calculated for the JUnit versions. The detailed smell characteristics are shown in Table I. It shows that the number of different smells increase steadily with time, which supports the assumption made by Chatzigeorgiou et. al. [6].

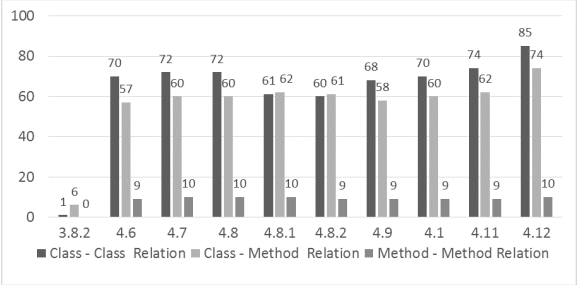

Figure 3: Relations between Smelly Components of JUnit

B. Relationship Identification

JCallGraph was used to extract the relationships between smelly components (smelly methods and smelly classes) for different versions of JUnit. Obtained relationships are shown in Table II. Results show that, the number of relationships between smelly classes increase with time (Table II Row 1). The number of relationships between two smelly methods does not differ much here. Fig. 3 presents relationship counts between smelly components for 10 releases of JUnit. This relationship data is used to generate smell graphs.

C. Graph Generation

In this study, graphs are generated for each version of JUnit to understand the relationship between code smells and how they evolve with time by observing clustering behavior. Detail of generated graphs are given in Table III.

D. Analysis of the result

As every vertex in the graph represents a smelly component in the code, it is very much expected that vertex count will rise with time [6], which was the case in this study. However, interesting results were obtained for clustering behavior. Cluster counts and size of different clusters show some patterns of change with time.

Table I: Smelly Components in Different Versions of JUnit

Smelly Components	Versions									
	3.8.2	**4.6**	**4.7**	**4.8**	**4.8.1**	**4.8.2**	**4.9**	**4.10**	**4.11**	**4.12**
GC	2	31	31	32	32	32	32	34	36	46
FE (classes)	0	19	20	20	19	19	19	20	20	21
FE (methods)	0	35	37	37	39	38	36	37	38	43
LM	6	22	23	23	23	23	23	24	25	31
TC	0	1	1	1	1	1	1	1	1	2
Smelly classes	2	38	39	40	39	39	39	42	44	55
Smelly methods	6	57	60	60	62	61	58	60	62	74

Table II: Relations between Smelly Components

Smell Relations	Versions									
	3.8.2	**4.6**	**4.7**	**4.8**	**4.8.1**	**4.8.2**	**4.9**	**4.10**	**4.11**	**4.12**
Class-Class	1	70	72	72	61	60	68	70	74	85
Class-Method	6	57	60	60	62	61	58	60	62	74
Method-Method	0	9	10	10	10	9	9	9	9	10

Table III: Detail of Graphs, Generated by Smelly Components of each Version

Graph Detail	Versions									
	3.8.2	**4.6**	**4.7**	**4.8**	**4.8.1**	**4.8.2**	**4.9**	**4.10**	**4.11**	**4.12**
Vertex Count	8	95	99	100	101	100	97	102	106	129
Cluster Count	1	21	24	25	33	34	24	25	26	35
Nodes in MC	8	69	71	71	54	54	67	70	71	87
Nodes in SNC	0	16	20	21	26	28	19	19	19	29
Nodes in SC	0	10	8	8	21	18	11	13	16	13

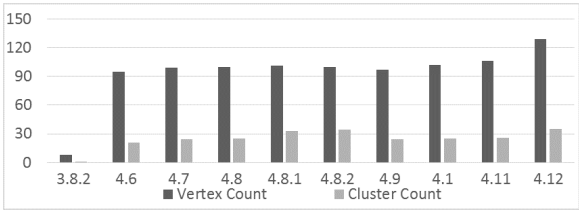

Figure 4: Evolution of Vertex Count and Cluster Count

1) Number of connected code smell clusters increases steadily with time: The cluster count in first release (3.8.2) was 1 which grew up to 35 in the final release (4.12). Average increase in cluster count is 3.78 per release. So a steady increase can be observed from the results. One decline in the cluster count is encountered in version 4.9. This inconsistency can be explained by the decline of vertex count in this version (shown in Fig. 4). Decline of vertex count indicates that a few smelly components might be removed or refactored. That can contribute in the decline of smelly cluster count. Now, this behavior of cluster increase means, if code smells exist in a component, its related components tend to be infected with new code smells.

2) Smell clusters can be categorized in three types: Based on the size and connectivity, the identified clusters can be divided in three categories. For each version of JUnit a single largest cluster is found that contains the highest amount of nodes (more than fifty percent of the smelly nodes). This is identified as the **Mega Cluster (MC)**. Another type of cluster is seen, **Single-Node Cluster (SNC)**. These type

of clusters hold only one smelly component that is not connected to any other smelly component. All the other identified clusters contain at least two connected smelly components. These are noted as **Small Clusters (SC)**. The number of nodes in different clusters are shown in Fig. 5.

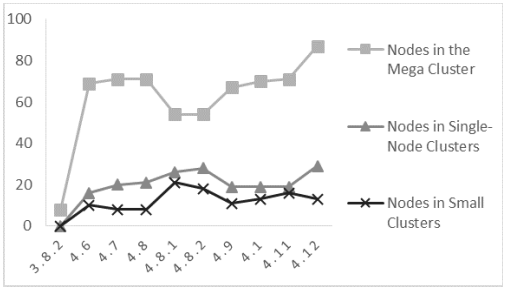

Figure 5: Number of nodes in different type of clusters

3) Smelly components tend to create a mega cluster: By observing the mega clusters in Fig. 5, it can be seen that it contains more than 50% of all smelly components. The case study shows that the mega cluster contains on an average 69.49% of all the smelly components. The lowest percentage is seen in JUnit 4.8.1 where it contained only 53.47% of all smelly nodes. While in JUnit 3.8.2 this percentage was maximum (100%). The study also found that on an average single-node clusters hold 19.04% and small cluster hold 11.47% of all smelly nodes. Now, as mega clusters can be interpreted as a large amount of badly designed code, it might be very difficult to refactor it.

4) Size of the mega cluster increases steadily with time: The study found that the size of the mega cluster increases with time. In the first version (3.8.2) of JUnit the mega cluster contained 8 nodes where, in the final version (4.12), it contained 87 nodes. An average increase 9.89 nodes is logged between two consecutive releases. In version 4.8.1, a significant decline in the size of the mega cluster is identified (71 to 54). The same version also had a sudden rise in number of nodes in small clusters (8 to 21). These two incidents might be the result of a major refactoring that eliminated some relations between smelly components and thus split few small clusters from the mega cluster. The semantics of this behavior is that ignoring the existence of mega clusters gradually increases the problem more, making it infeasible to ultimately refactor it.

After analyzing the results, few major behavior of code smell clusters have been identified. The number of related code smell clusters in source code increases steadily with time. For this case study the average increase in cluster count is 3.78 clusters per release. These related code smell clusters can be categorized in three groups. Among these, the mega cluster holds 69.49% of all the smelly nodes on an average. It has also been identified that the size of the mega cluster increases steadily with time (9.89 nodes per release). The results of this study gives a better understanding about how relations between code smells evolve in software.

IV. Conclusion

This paper analyses code smell clusters in evolving software to understand relationship between the smelly components of software. For the identification of clusters, it first detects classes and methods containing code smells in the software source code. Then, the relationship between these smelly components are extracted. Finally, a graph is generated for each version of software using these components and their relationships to get the clusters of each version.

A case study was conducted on the evolution of code smells in JUnit. 10 different releases of JUnit was analyzed to identify the code smell clusters in each version. A number of observations were made after examining the clusters of each version. These observations reveal significant characteristics of code smells in evolving software.

The future direction lies in generalizing the observations.

Acknowledgements

This research is supported by ICT Division, Ministry of Posts, Telecommunications and Information Technology, Bangladesh. 56.00.0000.028.33.028.15-214, 24-06-2015.

References

[1] F. A. Fontana, V. Ferme, and M. Zanoni, "Towards assessing software architecture quality by exploiting code smell relations," in *Proceedings of the Second International Workshop on Software Architecture and Metrics*. IEEE Press, 2015, pp. 1–7.

[2] M. Abbes, F. Khomh, Y.-G. Gueheneuc, and G. Antoniol, "An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension," in *Software Maintenance and Reengineering (CSMR), 2011 15th European Conference on*. IEEE, 2011, pp. 181–190.

[3] R. Peters and A. Zaidman, "Evaluating the lifespan of code smells using software repository mining," in *Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on*. IEEE, 2012, pp. 411–416.

[4] J. Garcia, D. Popescu, G. Edwards, and N. Medvidovic, "Identifying architectural bad smells," in *Software Maintenance and Reengineering, 2009. CSMR'09. 13th European Conference on*. IEEE, 2009, pp. 255–258.

[5] S. Olbrich, D. S. Cruzes, V. Basili, and N. Zazworka, "The evolution and impact of code smells: A case study of two open source systems," in *Proceedings of the 2009 3rd international symposium on empirical software engineering and measurement*. IEEE Computer Society, 2009, pp. 390–400.

[6] A. Chatzigeorgiou and A. Manakos, "Investigating the evolution of bad smells in object-oriented code," in *Quality of Information and Communications Technology (QUATIC), 2010 Seventh International Conference on the*. IEEE, 2010, pp. 106–115.

[7] N. Tsantalis, T. Chaikalis, and A. Chatzigeorgiou, "Jdeodorant: Identification and removal of type-checking bad smells," in *Software Maintenance and Reengineering, 2008. CSMR 2008. 12th European Conference on*. IEEE, 2008, pp. 329–331.