Analyzing Code Evolution to Uncover Relations between Bad Smells

Angela Lozano
Vrije Universiteit Brussel
Brussels, Belgium
alozanor@vub.ac.be

Kim Mens
Université catholique de Louvain
Louvain-la-Neuve, Belgium
kim.mens@uclouvain.be

Jawira Portugal
–
La Paz, Bolivia
jawitugal@gmail.com

***Abstract*—This paper reports on evidence found of five possible relations (Plain Support, Mutual Support, Rejection, Common Refactoring, and Inclusion) among four bad smells (God Class, Long Method, Feature Envy, and Type Checking). We analyzed several releases of three open-source applications (16 for Log4j, 34 for Jmol, and 45 for JFreeChart) using four direct and two indirect metrics. This analysis uncovered correlations between three of these bad smells, namely, Feature Envy, Long Method, and God Class. The strongest correlation discovered was between Feature Envy and Long Method, followed by a mild correlation between Long Method and God Class, and between Feature Envy and God Class. These findings seem to provide initial evidence of the co-existence of bad smells and therefore, the need for bad smell removal plans to take into account these correlations in order to minimize code improvement efforts.**

I. Introduction

Structural guidelines for implementing software are difficult to follow because local and global optimization are usually in conflict. An example of this conflict is cohesion (local) and coupling (global). The assignment of functionality among classes should be such that cohesion is high for each class, and coupling is low for the whole application. However, while a perfect coupling can be achieved by putting all functionality in a single class, it would result in poor cohesion. Similarly, making a class per feature needed would result in perfect cohesion but rather high coupling. Although it is not possible to ensure that the functionality of an application is perfectly distributed, there are several indicators of structural issues. These indicators, called *bad smells*, **point to parts of the code that may need to be refactored**. One example of these bad smells is having a long method. In object oriented languages, long methods are typically considered harmful because they are complex or they might have too many responsibilities. Dividing the method into several methods, each one with a single responsibility would reduce the complexity of each one of these methods, and facilitate their usage in different contexts.

Recent studies have shown that bad smells can indeed have a negative impact on program comprehension [1], on change proneness [2], and on fault proneness [2]–[5]. This paper presents a study on the evolution of the source code to look for co-occurences of bad smells. In particular, we want to measure to what extent bad-smells co-occur, and whether or not there is evidence of over-time degradation of the source code due to the presence of bad smells.

II. Approach

This paper presents an empirical study that analyzes the following five (out of the seven) inter-smell relations proposed by Pietrzak & Walter [6]:

- `Plain Support` from A to B occurs when a source code entity having the bad smell A implies with high certainty that the source code entity also has the smell B. We represent plain support with a single arrow that indicates the direction of the relation (i.e., $A \rightarrow B$).
- `Mutual Support` occurs when there is symmetrical plain support, which means that is impossible to identify the direction of the relation. We represent mutual support with a single arrow with two ends (i.e., $A \leftrightarrow B$).
- `Rejection` is the contrary of plain support. Rejection from A to B means that having the bad smell A implies with high likelihood that the smell B is not present in the source code entity. We represent rejection with a single negated arrow that indicates the direction of the relation (i.e., $A \not\rightarrow B$).
- `Common Refactoring` indicates that a single refactoring could remove both bad smells. We represent it with a single arrow that indicates the direction of the relation where both bad smells are negated. This indicates that both smells can co-occur, but whenever one is removed then the other one will likely cease to exist as well (i.e., $!A \rightarrow !B$).
- `Inclusion` occurs when there are no occurrences of a bad smell (A) that are not accompanied by another bad smell (B). In other words, for every source code entity where A is present, B is present too. We represent inclusion with a double arrow that indicates the direction of the relation (i.e., $A \Rightarrow B$)

A. Metrics Extracted

For each source code entity, at each version, and for each bad smell analyzed we record the following metrics:

- `exist(BS)`: number of source code entities (Classes or Methods) where the bad smell BS was detected
- `co-exist(BS1, BS2)`: number of SCEs (Classes or Methods) where both bad smells are detected

- `disappear(BS)`: number of SCEs (Classes or Methods) where the bad smell was present in the previous analyzed version of the source code but is no longer present in the currently analyzed version
- `co-disappear(BS1, BS2)`: number of SCEs (Classes or Methods) where both both bad smells BS1 and BS2 disappeared from the analyzed entity with respect to the previous analyzed version of the source code

1) Indirect Metrics: Using on these basic metrics we derive additional metrics:

$$exist-overlap(BS1, BS2) = \frac{co-exist(BS1, BS2)}{exist(BS1)}$$

$$disappear-overlap(BS1, BS2) = \frac{co-disappear(BS1, BS2)}{disappear(BS1)}$$

Note that these metrics are not symmetric e.g., exist-overlap(BS1, BS2) is not necessarily the same as exist-overlap(BS2, BS1)

2) Metrics Used to Identify Relations between Bad Smells: Relation between metrics and inter-smell relations

- `Plain Support` $(A \rightarrow B)$`: if` exist-overlap(A, B) $>$ exist-overlap(B, A)
- `Mutual Support` $(A \leftrightarrow B)$`: if` exist-overlap(A, B) $\approx$ exist-overlap(B, A)
- `Rejection` $(A \not\rightarrow B)$`: if` co-exist(A, B) ≈ 0
- `Common Refactoring` $(!A \rightarrow !B)$`: if` disappear-overlap(A, B) $>$ disappear-overlap(B, A)
- `Inclusion` $(A \Rightarrow B)$ `: if` exist-overlap(A, B) ≈ 1

B. Bad Smells Analyzed

JDeodorant can detect bad smells by trying to identify possible refactorings. We decided to use JDeodorant's bad smells [7]–[9] because its detection approach is reliable for analyses over time[1]. The bad smells that JDeodorant can detect are Feature Envy (FE), God Class (GC), Long Method (LM), and Type Checking (TC).

C. Case Studies

We analyzed three open source Java applications over several releases. Log4j, a logging library that is part of the Apache Foundation. Log4j was analyzed from version 1.2.1 to 1.2.17 (i.e., sixteen releases). Jmol is a viewer for chemical structures (including crystals, bio-molecules and DNA sequences) in 3D. Versions 1 until 11.0 (34 releases) were analyzed for Jmol. Finally, JFreeChart, a chart library (bar charts, pie charts, line charts, etc.). We analyzed forty-five releases of JFreeChart (from version 0.5.6 to 1.0.14).

[1]Note that the source code entities that are above/below a threshold may change from one version to the other. Also note that avoiding metrics make the results more likely to be replicable across different case studies.

III. Results

Figure 1 shows the amount of bad smells accumulated trough all versions analyzed. The most common bad smells in all case studies were long method (LM) and god class (GC). The least frequent bad smell is type checking (TC).

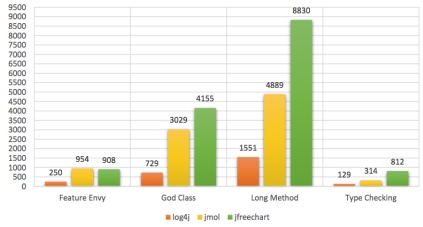

Fig. 1: Cumulative number of bad smells found per case study

Table I shows that the number of instances per type of bad smell are similar across case studies, and also that they tend to increase over time.

TABLE I: Number of bad smells per case study in the first and last versions analyzed

App	Version	TC	FE	GC	LM
Log4j	1st.	8	16	45	96
	Last	8	17	53	114
Jmol	1st.	10	16	47	79
	Last	19	35	130	200
JFreechart	1st.	17	6	21	38
	Last	13	35	153	199

However when looking at both (Figure 1 and Table I) we can see that the growth of bad smells varies across applications. While Log4j seems to have a controlled growth of their bad smells, Jmol and JFreechart degrade over time. In particular, there seems to be an explosion of long methods and god classes. The only case of bad smells being reduced in comparison with the first version are type checkings in JFreechart.[2]

A. Evidence for Inter-smell Relations

Table II shows the median result for the metrics that can identify relations between all the pairs of bad smells analyzed.

Note that type checkings did not exist or disappear with any other bad smell. Therefore it seems that having type checking is a good indicator that the entity does not suffer from other of the bad smells analyzed. That is, $TC \not\rightarrow FE$, $TC \not\rightarrow LM$, and $TC \not\rightarrow GC$.

In contrast, feature envy seems to have a very strong relation with long methods. Results indicate that whenever a feature envy is found, that method is very likely to be a long method as well (i.e. $FE \Rightarrow LM$).

Feature envy existence overlaps more (30%, 58%, and 60%) with god classes than on the other way around. Therefore we conclude that the existence of feature envy increases the likelihood of that class to being a god class i.e., $FE \rightarrow GC$.

[2]Although there are several versions in which the number of long methods and god classes were reduced in JFreechart, the final count is higher than the initial count for these two bad smells.

TABLE II: Median of the metrics used to identify relations

Bad Smells	App	exist-overlap(A, B)	exist-overlap(B, A)	co-exist(A, B)	disappear-overlap(A, B)	disappear-overlap(B, A)
A = FE, B = LM	Log4J	0.94	0. 15	15	1	0
	Jmol	1	0	27	0	0
	JFreechart	0.92	0.1	22	0.14	0
A = FE, B = TC	Log4J	0	0	0	0	0
	Jmol	0	0	0	0	0
	JFreechart	0	0	0	0	0
A = FE, B = GC	Log4J	0.44	0.16	7	0	0
	Jmol	0.85	0.27	28	0.17	0
	JFreechart	0.76	0.17	16	0	0
A = GC, B = LM	Log4J	0.75	0.34	33	0	0
	Jmol	0.83	0.51	82	0.27	0.39
	JFreechart	0.75	0.4	73	0	0.06
A = GC, B = TC	Log4J	0	0	0	0	0
	Jmol	0	0	0	0	0
	JFreechart	0	0	0	0	0
A = LM, B = TC	Log4J	0	0	0	0	0
	Jmol	0	0	0	0	0
	JFreechart	0	0	0	0	0

This difference between exist-overlaps is lower (40%, 30%, and 35%) for god classes co-existing with long methods, so it is difficult to decide whether god classes support long methods ($GC \rightarrow LM$) or if there is mutual support ($GC \leftrightarrow LM$). We concluded that the relation between these two bad smells is mutual support because there is not enough evidence to conclude a common refactoring (even though some refactorings eliminate both of them).

B. Discussion

This study ignores the analysis of bad smells that cannot be found by JDeodorant (and their corresponding relations). That is, bad smells for which it could not find an automatic refactoring, or whose refactoring does not 'improve' the source code entity analyzed.

Moreover, the fact that these bad smells are located by proposing a refactoring alternative must be taken into account to understand the results. Therefore, a method with feature envy should not be interpreted as a method that uses more methods of other classes than its own, but rather a method that should be moved to a specific class. JDeodorant's long methods can be divided, while the control and data flow of long methods in the general case may be tangled so that the amount of long methods analyzed might be just a fraction than those that have issues. Similarly, the god classes analyzed here correspond to classes that have sets of methods that are disjoint among themselves (i.e., a class whose methods can be separated into cohesive subsets). However, the definition commonly used is more vague: classes that represent multiple abstractions or have too many responsibilities. Finally, methods with type checkings are those methods that can be implemented with a strategy or template pattern. In this case, JDeodorant's definition is again more precise than using too many switches or checks against the type of objects before calling the appropriate methods.

Providing explanations for the results requires to consider the specific definitions of these bad smells. It is easy to conjecture possible relations between two bad smells. For instance, if a method contains type-checks it is likely to be longer than the average method, therefore, type-checks support long methods. Nevertheless, where the specific definitions are taken into account the results obtained seem evident. For instance, there is no reason for methods that should be implemented with the template or the strategy pattern to be longer, require more services of classes other than its own, or be located in classes that handle multiple abstractions/responsibilities ($TC \not\rightarrow FE$, $TC \not\rightarrow LM$, and $TC \not\rightarrow GC$). Whenever a method is located in the wrong class it is likely to be dealing with separable data/control flow paths ($FE \Rightarrow LM$). Methods that are wrongly located are likely to be a part of a disjunct set of methods within their class ($FE \rightarrow GC$). Finally, separable methods are likely to belong to classes with disjunct sets of methods and classes with disjunct sets of methods are likely to have methods that are handling more than one data/control flow path ($GC \leftrightarrow LM$).

C. Threats to Validity

The following aspects may affect the validity of these results:

1) Internal Validity: As mentioned in the discussion section, the bad smells detected are just a subset of those that a developer (or another tool) may have encounter. Therefore, it is likely that with another experiment setup the relations among bad smells analyzed are found to be stronger. Notice that JDeodorant, relies on eclipse's AST which implies that only the revisions that compiled were analyzed. However, given than the variability among the metrics analyzed was low, we are confident that the revisions missing do not have a significant impact in the results. This study does not perform origin analysis which means that for each version we count which entities had each bad smell. In order, to identify which bad smells were removed, a simple name comparison is performed against the entities with bad smells from the previous revision. Therefore, if there were source code entities that were renamed or moved and their bad smells remained unchanged, they are identified twice as 'deleted' and 'added' bad smells.

2) External Validity: Although the applications analyzed belong to different domains these results may only hold for small to medium sized, Java, open source applications. Analyzing the limitations of these results for similar projects, larger projects, in other programming languages, or within an industrial setting remains as future work.

3) Construct Validity: We think this study has little concerns regarding the relation between the evidence gathered and

the conclusion for several reasons. First, the data collection is automatic so the chance of random errors is low[3]. Second, there were no a priori hypotheses so we considered every possible relation (in both directions) which makes it less susceptible of researcher bias. Third, the identification of bad smells does not rely on trespassing thresholds which may change over time.

IV. Related work

Yamashita and Moonen [10] analyzed twelve bad smells in four industrial applications implemented in Java with feedback from the developers in order to identify problems (during maintenance) related to the interaction among bad smells located in the same source code entity. Using principal component analysis, they were able to determine whether the problems were due to single or multiple bad smells. In contrast, our study focuses on the prevalence of inter-smell relations and not on their effect. However, the results are converging. Yamashita and Moonen also found interactions between FE, GC, and LM (*hoarders*). In addition, they showed that the effects of collocated smells are equivalent to having these bad smells in different but coupled files. Moreover, other studies on the impact of bad smells have also reported the relation found between GC and LM [1], [4].

Even though there is little evidence on the prevalence of co-existing bad smells, Liu et al [11] proposed an algorithm to reduce the number of refactorings (and their level of difficulty) required to eliminate bad smells from a source code entity. By analyzing the dependencies among bad smells in case they are inside the same source code entity, the authors develop a tree with the resolution order to tackle co-occurring bad smells.

Moreover, there are a couple of studies on the evolution of bad smells. Peters and Zaidman [12] analyzed the evolution of bad smells to identify to what extent they are refactored. They analyzed three industrial and five open source java applications using JDeodorant and Ptdej. Finally, Chatzigeorgiou and Anastasios [13] analyze two OSS and JDeodorant's bad smells in order to identify deliberate from accidental bad smell removal.

V. Conclusion

This paper presents an analysis of bad smells over time. We have recorded to what extent different bad smells are located in the same source code entity, and whether or not they are removed in the same version. These events allowed us to identify six relations between bad smells. Methods with type checkings are very unlikely to have feature envy, be a long method, or to belong to a god class. This is, type checkings reject feature envy, long method, and god class. Almost all methods with feature envy were located in long methods, which means feature envy methods are included in long methods. Also several methods with feature envy were located in god classes, which means feature envy methods support the existence of god classes. And finally we found that the percentage of god classes have long methods is similar to the percentage of long methods that belong to god classes, i.e., god classes and long methods support each other. These results can be used to plan the order in which bad smells detection should be tackled. For instance, if type checking is confirmed first, it might be unnecessary to check for other bad smells within the same source code entity. This planing can increase the performance of tools that detect bad smells, with a low cost in precision. Moreover, this planing could also be used to analyze which bad-smells interact, and for which sets of bad smells it is worth to prioritize the order in which they are removed. For future work we expect to reproduce these results in other case studies, with other bad smells detection tools, and using additional metrics. Moreover, we would like to check whether these results hold for a commit based analysis instead of a release based analysis. Finally, we would also like to analyze the effect of inter-smell relations on the prioritization of refactorings.

[3]An error in the data collection or analysis would be systematic for all bad smells and applications analyzed. So, it would not affect the relations found.

Acknowledgment

Angela Lozano is funded by the Cha-Q SBO project (IWT-Vlaanderen), Belgium.

References

[1] 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 *Proc. of the European Conf. on Software Maintenance and Reengineering*, ser. CSMR '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 181–190.

[2] F. Khomh, M. D. Penta, Y.-G. Guéhéneuc, and G. Antoniol, "An exploratory study of the impact of antipatterns on class change- and fault-proneness," *Empirical Softw. Engg.*, vol. 17, no. 3, pp. 243–275, Jun. 2012.

[3] H. Aman, S. Amasaki, T. Sasaki, and M. Kawahara, "Empirical analysis of fault-proneness in methods by focusing on their comment lines," in *Proc. the 2nd Int'l Workshop on Quantitative Approaches to Software Quality*, ser. QuASoQ'14, 2014, pp. 51–56.

[4] N. Zazworka, A. Vetro', C. Izurieta, S. Wong, Y. Cai, C. Seaman, and F. Shull, "Comparing four approaches for technical debt identification," *Software Quality Control*, vol. 22, no. 3, pp. 403–426, Sep. 2014.

[5] A. Vetro, N. Zazworka, F. Shull, C. Seaman, and M. A. Shaw, "Investigating automatic static analysis results to identify quality problems: An inductive study," in *Proc. of the 35th Annual IEEE Software Engineering Workshop*, ser. SEW '12, 2012, pp. 21–31.

[6] B. Pietrzak and B. Walter, "Leveraging code smell detection with inter-smell relations," in *Extreme Programming and Agile Processes in Software Engineering*. Lecture Notes in Computer Science, 2006, pp. 75–84.

[7] M. Fokaefs, N. Tsantalis, and A. Chatzigeorgiou, "Jdeodorant: Identification and removal of feature envy bad smells," in *Proc. of the Int'l Conf. on Software Maintenance (ICSM)*, 2007, pp. 519–520.

[8] N. Tsantalis, T. Chaikalis, and A. Chatzigeorgiou, "Jdeodorant: Identification and removal of type-checking bad smells," in *Proc. of the European Conf. on Software Maintenance and Reengineering (CSMR)*, 2008, pp. 329–331.

[9] M. Fokaefs, N. Tsantalis, E. Stroulia, and A. Chatzigeorgiou, "Jdeodorant: Identification and application of extract class refactorings," in *Proc. of the Int'l Conf. on Software Engineering (ICSE)*, 2011, pp. 1037–1039.

[10] A. Yamashita and L. Moonen, "Exploring the impact of inter-smell relations on software maintainability: An empirical study," in *Proc. of the Int'l Conf. on Software Engineering (ICSE)*, 2013, pp. 682–691.

[11] H. Liu, Z. Ma, W. Shao, and Z. Niu, "Schedule of bad smell detection and resolution: A new way to save effort," *IEEE Trans. Softw. Eng.*, vol. 38, no. 1, pp. 220–235, 2012.

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

[13] A. Chatzigeorgiou and A. Manakos, "Investigating the evolution of code smells in object-oriented systems," *Innov. Syst. Softw. Eng.*, vol. 10, no. 1, pp. 3–18, 2014.