Experimental Evaluation of Code Smell Detection Tools

Thanis Paiva[1], Amanda Damasceno[1], Juliana Padilha[1], Eduardo Figueiredo[1], Claudio Sant'Anna[2]

[1]Computer Science Department – Federal University of Minas Gerais (UFMG)

[2]Computer Science Department – Federal University of Bahia (UFBA)

{thpaiva, amandads, juliana.padilha, figueiredo}@dcc.ufmg.br, santanna@dcc.ufba.br

Abstract. *Code smells are code fragments that can hinder the evolution and maintenance of software systems. Their detection is a challenge for developers and their informal definition leads to the implementation of multiple detection techniques by tools. This paper aims to evaluate and compare three code smell detection tools, namely inFusion, JDeodorant and PMD. These tools were applied to different versions of a same system. The results were analyzed to answer four research questions related to the evolution of the smells, the relevance of the smells detected, and the agreement among tools. This study allowed us to understand the evolution of code smells in the target system and to evaluate the accuracy of each tool in the detection of three code smells: God Class, God Method, and Feature Envy.*

1. Introduction

Code smells are symptoms that something may be wrong in the system code [Fowler 1999]. They can degrade system quality aspects, such as maintainability, and potentially introduce flaws [Yamashita and Counsell 2013]. Different studies report that the cost of software maintenance is about 70-80% of the total project cost [Dehagani and Hajrahimi 2013]. Therefore, it is important to detect code smells throughout the software development. However, code smell detection is a complex, tedious and error prone task. In this context, tools for automatic detection of code smells can assist developers in the identification of affected entities, facilitating the detection task. This goal is accomplished by the implementation of different detection techniques that allow the tools to highlight the entities most likely to present code smells.

Nowadays, there are an increasing number of software analysis tools available for detecting bad programming practices [Fontana et al. 2012] [Murphy-Hill and Black 2010] [Tsantalis et al. 2008] and, in general, there is increasing awareness of software engineers about the structural quality of features under development. The question is how to assess and compare tools to select which is more efficient in a given situation. However, evaluation of the effectiveness of tools for detecting code smells presents some challenges [Fontana et al. 2012]. First, there are different interpretations for each code smell, given the ambiguity and sometimes vagueness of their definitions. This lack of formalism prevents different tools to implement the same detection technique for a specific code smell. Second, these techniques generate different results, since they are usually based on the computation of a particular set of combined metrics, ranging from standard object-oriented metrics to metrics defined in ad hoc way for the smell detection purpose [Lanza and Marinescu 2006]. Finally, even if the same metrics are used, the threshold values might be different because they are established considering different factors, such as system domain and its size, organizational practices and the experience

and understanding of software engineers and programmers that define them. Changing the threshold has a large impact on the number of code smells detected.

The previous factors make it difficult to validate the results generated by different techniques, which is made only for small systems and few code smells [Mäntylä 2005] [Moha et al. 2010] [Murphy-Hill and Black 2010]. For instance, Fontana [2012] investigated six code smells in one software system, named GanttProject. Their study showed that a more effective analysis requires a greater understanding of the system as well as its code smells. The difficulty lies not only in the different interpretations of code smells, but also in the manual identification of the code smells, that is also a challenge. Therefore, it is difficult to find open-source systems with validated lists of code smells to allow further analysis.

Using an approach similar to Fontana [2012], this paper presents an in depth analysis of the identification and evolution of three code smells in different versions of a software system, named MobileMedia [Figueiredo et al. 2008]. We compare the accuracy in terms of recall and precision of three code smell detection tools, namely inFusion[1], JDeodorant[2] [Tsantalis et al. 2008], and PMD[3]. This paper also presents a measurement of agreement among the analyzed tools. We focus our investigation on three code smells detected by these tools: God Class [Riel 1996], God Method [Fowler 1999], and Feature Envy [Fowler 1999]. We found that in spite of a high agreement among the tools, their individual accuracy varies depending on the code smell being analyzed.

The rest of this paper is organized as follows. Section 2 introduces the code smells and the detection tools evaluated. Section 3 describes the study settings. Section 4 presents the results and tries to answer research questions based on statistical analysis. Section 5 discusses the main threats to the validity of this study and the related work while Section 6 concludes this paper and points out directions for future work.

2. Code Smell Definitions and Detection Tools

Smell Definitions. Code smells were proposed by Kent Beck in Fowler's book [Fowler 1999] as a mean to diagnose symptoms that may be indicative of something wrong in the system code. In this paper, we focus on three code smells. *God Class* describes an object that knows too much or does too much [Riel 1996]. *God Method* represents a method that has grown too much [Fowler 1999] and tends to centralize the functionality of a class. *Feature Envy* represents a method that seems more interested in a class other than the one it actually is in [Fowler 1999]. These smells were selected because they are the three predominant smells in the target system [Padilha et al. 2014] that can be detected by at least two of the following evaluated tools.

Detection Tools. In this paper we evaluate three code smell detection tools: inFusion[1], JDeodorant[2], and PMD[3]. They were selected because they are available for download and actively developed and maintained. Their results are easily accessible and they are able to detect the three predominant code smells in our target system: MobileMedia. More information about the tools evaluated is available on the project website[4].

[1] Available at http://www.intooitus.com/products/infusion
[2] Available at http://jdeodorant.com/
[3] Available at http://pmd.sourceforge.net/
[4] Available at http://homepages.dcc.ufmg.br/~thpaiva/vem2015

inFusion[1]. This is a commercial standalone tool for Java, C, and C++ that detects 22 code smells, including the three smells of our interest: God Class, God Method, and Feature Envy. The detection techniques for all the smells were initially based in the detection strategies defined by Lanza and Marinescu [2006], and then successively refined using real code.

JDeodorant[2] [Tsantalis et al. 2008]. This is an open-source Eclipse plugin for Java that detects four code smells: God Class, God Method, Feature Envy, and Switch Statement. The detection techniques are based in refactoring opportunities, for God Class and Feature Envy and slicing techniques, for God Method [Fontana et al. 2012].

PMD[3]. This is an open-source tool for Java and an Eclipse plugin that detects many problems in Java code, including two of the smells of our interest: God Class and God Method. The detection techniques are based on metrics. For God Class, it relies on the detection strategy defined by Lanza and Marinescu [2006] and, for God Method, a single metric is used: NLOC.

3. Study Settings

Target System. Our study involved the MobileMedia system [Figueiredo et al. 2008], a software product line (SPL) for applications that manipulate photo, music, and video on mobile devices. This system was selected because it has been previously used in other maintainability-related studies [Figueiredo et al. 2008] [Macia et al. 2012], and we have access to their developers and experts. Therefore, we were able to recover its reference list of actual code smells. Our study involved nine versions object-oriented (1 to 9) of MobileMedia, ranging from 1 KLOC to a little over 3 KLOC. Table 1 shows for each version: the number of classes, methods, and lines of code. It also contains the total number of God Classes (GC), God Methods (GM), and Feature Envy (FE) found in each system version.

Table 1. MobileMedia System Information

	Size Metrics of MobileMedia			Code Smells in the Reference List			
V	# of Classes	# of Methods	LOC	GC	GM	FE	Total
1	24	124	1159	3	9	2	14
2	25	143	1316	3	7	2	12
3	25	143	1364	3	6	2	11
4	30	161	1559	4	8	2	14
5	37	202	2056	5	8	2	15
6	46	238	2511	6	9	2	17
7	50	269	3015	7	7	2	16
8	50	273	3167	9	7	2	18
9	55	290	3216	7	6	3	16

Reference List Protocol. Two experts used their own strategy for detecting, individually and manually, the code smells in the system's classes and methods. As a result, two lists of entities were created. These lists were merged and the findings discussed with a developer of the system to achieve a consensus and validate the entities that present a code smell. The result of this discussion generated the final reference list used in this paper.

Research Questions. In this study, first we explore the presence of code smells in the system. For that purpose, we formulate the first two research questions (RQ1 and RQ2) presented below. We then analyze and compare the three detection tools based on two additional research questions (RQ3 and RQ4).

RQ1. *The number of code smells increase over time?*

RQ2. *How the code smells evolve with the system evolution?*

RQ3. *What is the accuracy of each tool in identifying relevant code smells?*

RQ4*. Do different detection tools agree for the same code smell when applied to the same software system?*

4. Statistical Analysis of Results and Discussions

4.1. Summary of Detected Code Smells

Table 2 shows the total number of code smell instances identified in the nine versions of MobileMedia by each tool. The most conservative tool is inFusion, with a total of 28 code smell instances. PMD is less conservative, detecting a total of 24 instances for God Class and God Method, in contrast with the 20 detected by inFusion. JDeodorant is more aggressive in its detections strategy by reporting 254 instances. That is, it detects more than nine times the amount of smells of the most conservative tools, inFusion and PMD. The reference list contains a total of 133 instances distributed among the nine versions of MobileMedia. The complete raw data is available on the project website[4].

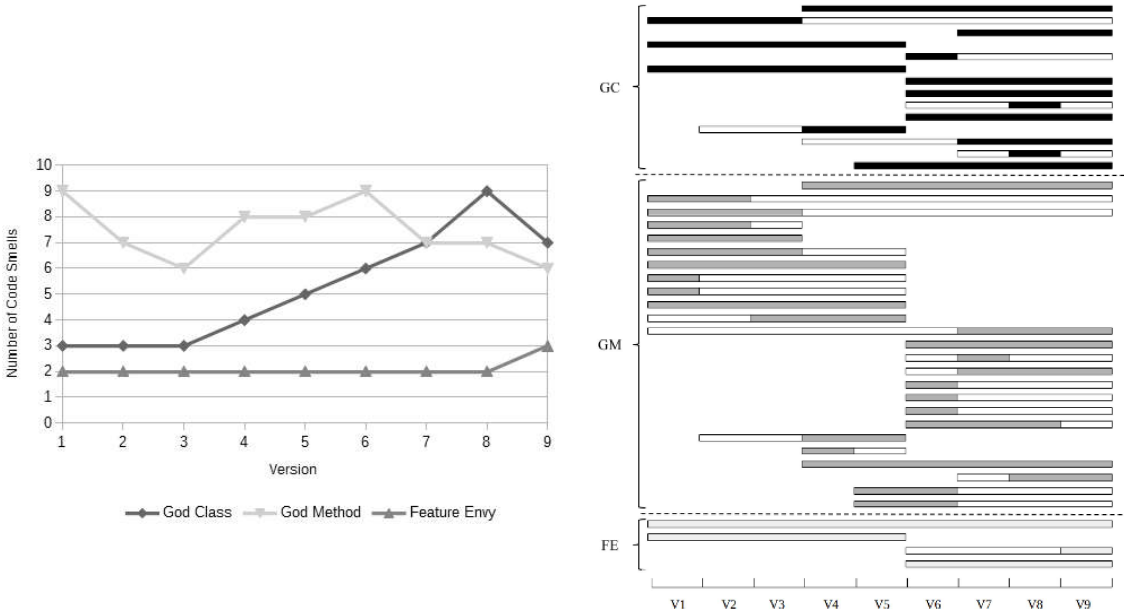
Table 2. Total number of identified code smells among versions by detection tools

Code Smell	inFusion	JDeodorant	PMD	Reference List
God Class	3	85	8	47
God Method	17	100	16	67
Feature Envy	8	69	-	19
Total	**28**	**254**	**24**	**133**

4.2. Evolution of Code Smells

This section aims to answer the research questions RQ1 and RQ2 using the code smell reference list. Focusing on RQ1, the results summarized in Figure 1a confirmed that only the number of God Class increases with the system's evolution. That was expected, since the evolution of the system includes new functionalities and God Classes tend to centralize them. The number of God Methods varies, but the last version has fewer instances than the first one. This variation is due to more frequent modification of methods between versions. For Feature Envy, the number of instances remained constant from version 1 up to 8. Only the final version has one additional instance.

To answer RQ2, we investigate the evolution of each code smell in Figure 1b. In Figure 1b each rectangle in a row represents the lifetime of an entity (class or method) in the system. The shaded area indicates the presence of a code smell. Each column consists of a particular system version. A rectangle is omitted if the entity does not exist in that particular version. We found that for 10 out of 14 God Class instances, the problem originates with the class. This result is aligned with recent findings [Tufano et al. 2015]. For the other 4 instances, the class becomes a code smell in later versions. For God Method, there is a lot of variation among versions, some methods are created with a code smell (19 of 25) and others become a code smell with the evolution of the system (6 of 25). However, when a smell is introduced in a method, it tends to be removed permanently in subsequent versions. For Feature Envy, in 3 out of 4 instances, the smell originated with the method and persisted during its entire existence. Only one method was created without Feature Envy and evolved to present that code smell.



(a) Number of code smells per version. **(b) Evolution of code smells**

Figure 1. Code Smells in MobileMedia

4.3. Analysis of Precision and Recall

This section aims to answer RQ3 presented in Section 3. To assess the tools accuracy to detect relevant code smell, we calculated the precision and recall based on the code smell reference list. A relevant code smell is a smell from the code smell reference list. Table 3 shows the average of recall and precision considering all versions for each tool and code smell analyzed. In general, for all code smells, JDeodorant has the highest average recall, followed by PMD and inFusion. In spite of its high recall, JDeodorant is also the tool that reports the highest number of false positives. That is, it has a lower precision. Consequently, the results require a greater validation effort by the developer.

For God Class and God Method, PMD and inFusion have a similar accuracy, i.e., lower recall, but higher precisions when compared to JDeodorant. They achieve 100% precision for God Method. Higher precision reduces greatly the validation effort of the programmer, but at the risk of missing relevant code smell.

For Feature Envy, inFusion had the worst overall accuracy with 0% of recall and precision. JDeodorant had a better accuracy for Feature Envy when compared to inFusion, although these were the worst values for JDeodorant. This can be an indicator that Feature Envy is a more complex code smell to be automatically detected, when compared to seemingly less complex smells, such as God Class and God Method.

Table 3. Average Recall and Precision for inFusion, JDeodorant and PMD

	Recall			Precision		
Code Smell	**inFusion**	**JDeodorant**	**PMD**	**inFusion**	**JDeodorant**	**PMD**
God Class	9%	58%	17%	33%	28%	78%
God Method	26%	50%	26%	100%	35%	100%
Feature Envy	0%	48%	-	0%	13%	-

4.4. Analysis of Agreement

This section aims to answer RQ4. We calculated the agreement among tools to investigate whether different detection tools return the same results for different detection algorithms. God Class and God Method are detected by inFusion, JDeodorant, and PMD. Feature Envy is detected only by inFusion and JDeodorant.

Table 4 shows the calculated overall agreement (OA) of the tools and the AC_1 *statistic* [Gwet 2014], which adjusts the overall agreement probability for chance agreement with a 95% confidence interval (CI). The overall agreement (OA) among tools is high, ranging from 78.38 % to 99.51%. However, the high agreement is due to the greater agreement on true negatives, i.e., most entities are categorized as not containing any smell. Comparing the tools results with the code smell reference list, we concluded that the occurrence of a smell in the system is indeed rare, as indicated by the detection tools.

Table 4. Overall Agreement (OA) and AC1 Statistics of the Analyzed Tools

	God Class			God Method			Feature Envy		
V	OA	AC1	95% CI	OA	AC1	95% CI	OA	AC1	95% CI
1	80.56%	0,764	[0.576,0.953]	94.62%	0,943	[0.907,0.979]	91.13%	0,903	[0.843,0.963]
2	81.33%	0,757	[0.565,0.949]	94.80%	0,946	[0.914,0.979]	92.20%	0,915	[0.862,0.967]
3	81.33%	0,757	[0.565,0.949]	94.91%	0,946	[0.914,0.979]	92.36%	0,916	[0.865,0.968]
4	82.22%	0,788	[0.631,0.945]	96.71%	0,966	[0.941,0.990]	93.83%	0,934	[0.891,0.976]
5	78.38%	0,732	[0.574,0.890]	95.40%	0,951	[0.925,0.978]	99.51%	0,995	[0.985,1.000]
6	85.51%	0,833	[0.724,0.942]	97.77%	0,977	[0.961,0.993]	97.49%	0,974	[0.953,0.995]
7	86.67%	0,848	[0.749,0.947]	96.79%	0,966	[0.948,0.985]	97.04%	0,970	[0.948,0.991]
8	82.67%	0,791	[0.672,0.911]	95.62%	0,954	[0.932,0.975]	98.18%	0,981	[0.965,0.998]
9	83.03%	0,797	[0.685,0.909]	97.04%	0,969	[0.952,0.986]	97.95%	0,979	[0.962,0.996]

The AC1 statistic is a robust agreement coefficient alternative to *Kappa* [Gwet 2014] and other common statistics for inter-rater agreement. It takes a value between 0 and 1, and communicates levels of agreement using the Altman's benchmark scale for Kappa [McCray 2013], that classifies agreement levels into Poor (< 0.20), Fair (0.21 to 0.40), Moderate (0.41 to 0.60), Good (0.61 to 0.80), and Very Good (0.81 to 1.00) [Altman 1991]. We found an AC1 “Very Good” for all smells and versions, with the exception of God Class in versions 6 and 7 with an AC1 “Good” with high precision, i.e., narrow confidence intervals.

The variation in the results is due to the implementation of different detection techniques or the variation in threshold values for the same technique. In our case, each tool uses a different detection technique, so we did not analyze the impact of different thresholds in the results. However, we are aware that they influence the results.

5. Threats to Validity and Related Work

Threats to Validity. Some limitations are typical of studies like ours, so we discuss the study validity with respect to common threats to validity. *Internal Validity* refers to threats of conclusions about the cause and effects in the study [Wohlin et al. 1999]. The main factors that could negatively affect the internal validity of the experiment are the size of the subject programs and possible errors in the transcription of the result of tool analysis. The subjects of our analysis are the nine versions of the MobileMedia that are small size programs. About transcription errors, the tools analyzed generate outputs in different formats. To avoid transcriptions errors we have reviewed all the data multiple

times. *External Validity* concerns the ability to generalize the results to other environments [Wohlin et al. 1999]. MobileMedia is a small open source system developed by a small team with an academic focus, therefore is not extendable to real large scale projects. This limits the generalization of our results. *Conclusion validity*, on the other hand, concerns the relation between the treatments and the outcome of the experiment [Wohlin et al. 1999]. This involves the correct analysis of the results of the experiment, measurement reliability and reliability of the implementation of the treatments. To minimize this threat, we discussed the results data to make a more reliable conclusion.

Related Work. Fontana [2012] presented an experimental evaluation of multiple tools, code smells and programs. Similarly, we evaluated tool agreement. However, we used the AC1 statistic, a more robust measure than the Kappa Coefficient. We also analyzed each tools precision and recall, unlike Fontana [2012]. Chatzigeorgiou and Manakos [2010] focused their analysis in evolution of code smells. In our work we also conduct this analysis, but at a higher level and not so focused in maintenance activities and refactoring.

6. Conclusions and Future Work

Comparing tools is a difficult task, especially when involves informal concepts, such as code smells. The different interpretations of code smell by researchers and developers leads to tools with distinct detection techniques, influencing the results and, consequently, the amount of time spent with validation.

In this paper we analyzed the code smells in the nine versions of MobileMedia to evaluate the accuracy of three code smell detection tools, namely inFusion, JDeodorant, and PMD to detect three code smells, God Class, God Method and Feature Envy. First we analyzed the evolution of MobileMedia code smells and found that the number of God Class increases, while the number of methods varies between versions and the number of Feature Envy remains constant. We also found that when a smell is when a smell is introduced in a method it tends to be removed permanently in subsequent versions.

Using MobileMedia as target system, we evaluated the tools accuracy and realized that there is a trade-off between the number of correctly identified entities and time spent with validation. For all smells, JDeodorant identified most of the correct entities, but generated reports with more false positives. This increases the validation effort, but captures the majority of the affected entities. On the other hand, PMD and inFusion identified more correct entities, however, also generated reports without some affected entities. This reduces validation effort, but neglects to highlight potential code smells. In future work, we would like to investigate more the evolution of other code smells in a system and how their evolution is related to maintenance activities.

Acknowledgments. This work was partially supported by CAPES, CNPq (grant 485907/2013-5), and FAPEMIG (grant PPM-00382-14).

References

Altman, D. G. (1991). “Practical Statistics for Medical Research”, Chapman & Hall, London.

Chatzigeorgiou, A. and Manakos, A. (2010). “Investigating the Evolution of Bad Smells in Object-Oriented Code”, In: Seventh Intl Conf. Quality of Information and Communications Technology.

Dehagani, S.M.H. and Hajrahimi, N. (2013). “Which Factors Affect Software Projects Maintenance Cost More?”, Acta Informatica Medica, vol.21, no.1, p.63.

Figueiredo, E. et al. (2008). “Evolving Software Product Lines with Aspects: an Empirical Study on Design Stability”, Proc. of ICSE, p.261-270.

Fontana, F. A. et al. (2012). “Automatic Detection of Bad Smells in Code: An Experimental Assessment.”, Journal of Object Technology 11(2): 5: p.1-38

Fowler, M. (1999). “Refactoring: Improving the Design of Existing Code”, Addison Wesley.

Gwet, K (2014). “Handbook of Inter-Rater Reliability: The Definite guide to Measuring the Extent of Agreement Among Raters”, Advanced Analytics, USA.

Lanza, M. and Marinescu, R. (2006). “Object-Oriented Metrics in Practice”. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

Macia, I. et al.(2012) “Are Automatically-Detected Code Anomalies Relevant to Architectural Modularity?”, Proc. of AOSD, p.167-178

Mäntylä, M.V. (2005). “An Experiment on Subjective Evolvability Evaluation of Object-Oriented Software: Explaining Factors and Inter-rater Agreement.”, In: Proc. Intl Symposium on Empirical Software Engineering, p. 287–296.

McCray, G. (2013). “Assessing Inter-Rater Agreement for Nominal Judgement Variables”, Paper presented at the Language Testing Forum. Nottingham, p.15-17.

Moha, N. et al. (2010). “From A Domain Analysis to the Specification and Detection of Code and Design Smells”, Formal Aspects of Computing, 22: p.345–361.

Murphy-Hill, E. and Black, A. (2010). “An Interactive Ambient Visualization for Code Smells”, In: Proceedings of the 5th international symposium on Software visualization, (SoftVis), p.5–14.

Padilha, J. et al. (2014). “On the Effectiveness of Concern Metrics to Detect Code Smells: An Empirical Study”. In proc. of the International Conference on Advanced Information Systems Engineering (CAiSE). Thessaloniki, Greece.

Riel, A.J. (1996). “Object-Oriented Design Heuristics”, Addison-Wesley Profess.

Tsantalis, N.; Chaikalis, T.; Chatzigeorgiou, A. (2008). “JDeodorant: Identification and Removal of Type-Checking Bad Smells”. In proc. of European Conference on Software Maintenance and Reengineering (CSMR).

Tufano, M. et al. (2015). “When and Why Your Code Starts to Smell Bad”. In proc. of International Conference on Software Engineering (ICSE). Firenze, Italy.

Wohlin, C. et al. (1999). "Can the Personal Software Process be used for Empirical Studies?", In: Proceedings ICSE workshop on Empirical Studies of Software Development and Evolution, Los Angeles, USA.

Yamashita, A. and Counsell, S. (2013). “Code Smells as System-Level Indicators of Maintainability: an Empirical Study”, Journal Systems and Software, p.2639-2653.