


RESEARCH

Open Access



# On the evaluation of code smells and detection tools

Thanis Paiva<sup>1\*</sup> , Amanda Damasceno<sup>1</sup>, Eduardo Figueiredo<sup>1</sup> and Cláudio Sant'Anna<sup>2</sup>

\* Correspondence:

thanis.paiva@gmail.com

<sup>1</sup>Department of Computer Science,  
Federal University of Minas Gerais,  
Av. Antônio Carlos, 6627, Belo  
Horizonte 31270-901, Brazil  
Full list of author information is  
available at the end of the article

## Abstract

Code smells refer to any symptom in the source code of a program that possibly indicates a deeper problem, hindering software maintenance and evolution. Detection of code smells is challenging for developers and their informal definition leads to the implementation of multiple detection techniques and tools. This paper evaluates and compares four code smell detection tools, namely inFusion, JDeodorant, PMD, and JSplRIT. These tools were applied to different versions of the same software systems, namely MobileMedia and Health Watcher, to calculate the accuracy and agreement of code smell detection tools. We calculated the accuracy of each tool in the detection of three code smells: God Class, God Method, and Feature Envy. Agreement was calculated among tools and between pairs of tools. One of our main findings is that the evaluated tools present different levels of accuracy in different contexts. For MobileMedia, for instance, the average recall varies from 0 to 58% and the average precision from 0 to 100%, while for Health Watcher the variations are 0 to 100% and 0 to 85%, respectively. Regarding the agreement, we found that the overall agreement between tools varies from 83 to 98% among all tools and from 67 to 100% between pairs of tools. We also conducted a secondary study of the evolution of code smells in both target systems and found that, in general, code smells are present from the moment of creation of a class or method in 74.4% of the cases of MobileMedia and 87.5% of Health Watcher.

**Keywords:** Code smells, Software metrics, Code smell detection

## 1 Background

Code smells refer to any symptom in the source code of a program that possibly indicates a deeper problem (Fowler 1999). They are symptoms of poor design and implementation choices that may cause problems for further development, maintenance, and evolution of software systems (Lanza and Marinescu 2006) (Yamashita and Counsell 2013). Once code smells are located in a system they can be removed by refactoring the source code (Fowler 1999). However, detection in large software systems is a time and resource-consuming, error-prone activity (Travassos et al. 1999), and manual inspection is slow and inaccurate (Langelier et al. 2005).

Tools for automatic or semi-automatic detection of code smells support developers in the identification of “smelly” entities. The implementation of detection techniques allows the tools to highlight the entities that most likely present code smells. Fortunately, there are many software analysis tools available for detecting code smells (Fernandes et al. 2016) (Murphy-Hill and Black 2010) (Tsantalis et al. 2008). In general, this fact

indicates an awareness of the software engineering community about the importance of controlling the structural quality of features under development (Fontana et al. 2012). On the other hand, it brings a new challenge on how to assess and compare tools and to select the most efficient tool in specific development contexts.

In fact, the evaluation of the effectiveness of tools for detecting code smells presents some problems (Fontana et al. 2012). For instance, the ambiguity and sometimes vagueness of code smell definitions lead to different interpretations for each code smell. This lack of precise definitions implies on tools that implement different detection techniques for the same code smell. In addition, 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 ways 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 defined considering different factors, such as system domain and its size, organizational practices, and the experience of software engineers that define them. Changing the thresholds has a large impact on the number of code smells detected (Vale et al. 2015).

In the context of the above problems, it is hard to interpret the results generated by different techniques. In fact, some preliminary studies (Mäntylä 2005) (Moha et al. 2010) (Murphy-Hill and Black 2010) tried to address some of these problems only for small systems and few code smells. Fontana et al. (2012), for instance, investigated six code smells in one software system, named GanttProject. Their study showed that a more effective evaluation of code smell detection techniques 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 serve as a basis for evaluating the techniques and tools.

Following up our previous studies (Fernandes et al. 2016) (Paiva et al. 2015), this paper aims at increasing our knowledge about code smells and their detection tools. In a previous work (Paiva et al. 2015), we compared three detection tools, namely iFusion, JDeodorant (Tsantalis et al. 2008), and PMD, in one software system. We also performed a comparative study of tools in detecting two code smells (Fernandes et al. 2016): Large Class and Long Method. This paper extends previous ones by analyzing an additional tool, named JSPIRIT, in a different system, named Health Watcher. Furthermore, we also conducted a secondary study of the evolution of code smells in MobileMedia and in Health Watcher. Therefore, this paper presents the results of a comparative study of four code smell detection tools in two software systems, namely Health Watcher (Soares et al. 2006) and MobileMedia (Figueiredo et al. 2008). We investigate recall, precision, and agreement of tools in detecting three code smells: God Class, God Method, and Feature Envy.

One of the main findings of this paper is that the tools accuracy varies mostly depending on the code smell being analyzed. A higher recall captures most of the affected entities providing a good coverage of the smells present in the system. On the other hand, higher precisions reduce the validation effort by reporting less false positives. Considering all smells, for MobileMedia the average recall varies from 0 to 58% and the average precision from 0 to 100%, while for Health Watcher the variations are

0 to 100% and 0 to 85%, respectively. The overall agreement among tools varies from 83 to 98% considering all smells in both systems. Between pairs of tools, the overall agreement varies from 67 to 100%. A closer analysis of the results revealed that the high agreement is when classes and methods do not present code smells. Regarding our secondary study on the evolution of code smells, we found that the majority of code smells in both systems originate with the class or method creation. That is the case for 74.4% of the smells in MobileMedia and 87.5% in Health Watcher, confirming the findings of Tufano et al. (2015). Another observation is that the number of smells does not necessarily grow with the size of the system, even though there was an increase of 2057 lines of code in MobileMedia and of 2706 lines of code in Health Watcher. For instance, the number of God Methods is 6 in all versions of the Health Watcher system.

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 focusing on the target systems, code smell reference list, and research questions. Section 4 compares the code smell detection tools by analyzing their accuracy and agreement. Section 5 presents a secondary study on the evolution of code smells in the systems MobileMedia and Health Watcher. Section 6 discusses the main threats to the study. Section 7 presents the related work while Section 8 concludes this paper and points out directions for future work.

## 2 Code smells and detection tools

Code smells were defined 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. They refer to any symptom in the source code of a program that possibly indicates a deeper problem. Code smell detection tools can help developers to maintain software quality by employing different techniques for detecting code smells, such as object-oriented metrics (Lanza and Marinescu 2006) and program slicing (Tsantalis et al. 2008). Section 2.1 briefly discusses code smells. Section 2.2 presents the tools evaluated in this paper.

### 2.1 Code smell definitions

Code smells are code fragments that suggest the possibility of refactoring. Originally, 22 code smells were described by Fowler (1999), along with the suggested refactorings. Other smells have also been proposed in the literature, such as Spaghetti Code (Brown et al. 1998), and Swiss Army Knife (Moha et al. 2010). Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure (Fowler 1999). The necessity of refactoring is indicated by the presence of code smells. Code smells can degrade quality aspects of the software system, such as maintainability and comprehensibility, hindering the software development process by making the code difficult to understand and, consequently, to maintain (Fowler 1999) (Lanza and Marinescu 2006) (Yamashita and Counsell 2013).

In this paper, we focus on three code smells: God Class, God Method, and Feature Envy. *God Class* defines a class that centralizes the functionality of the system. That is, "a class that knows or does too much" (Riel 1996). In other words, a God Class violates

the single responsibility principle (DeMarco 1979) of object-oriented programming that states that a class should have only a single responsibility. *God Method* happens when more and more functionality is added to a method until it becomes out of control and difficult to maintain and extend (Fowler 1999). Therefore, God Method tends to centralize functionalities of a class in the same way that a God Class tends to centralize the functionality of a subsystem, or even an entire system. *Feature Envy* indicates a method that seems more interested in the functionalities of a class other than the one it actually is in (Fowler 1999). These code fragments access directly or indirectly several data from other classes. It can indicate that the method is badly located and should be transferred to another class (Fowler 1999). These three code smells were selected because (i) they are among the most frequent smells detected by code smell detection tools (Fernandes et al. 2016), allowing us to compare the tools recall, precision, and agreement, (ii) they can also be detected without the assistance of tools, which is necessary to compile the code smell reference list, and (iii) they are the most frequent code smells in the target systems, namely MobileMedia and Health Watcher (Section 3.1). In addition, they can be detected by at least two of the evaluated tools.

## 2.2 Detection tools

In this paper, we evaluate four code smell detection tools, namely inFusion, JDeodorant, PMD, and JSPIRIT, selected from the tools available for download that are free or have a trial version (Fernandes et al. 2016). These tools were selected because they analyze Java programs, they can be installed and setup from the provided downloaded files, they detect the analyzed smells in both target systems and their output is a list of occurrences of code smells, allowing calculating recall, precision, and agreement. Other tools were discarded for different reasons. For instance, Checkstyle was discarded because it has not detected instances of smells in any of the target systems, while Stench Blossom was discarded for not providing a concrete list of code smells. Instead it supports only visualization features.

Table 1 summarizes the basic information about the evaluated tools. The column *Tool* contains the names of the analyzed tools as reported in the tools corresponding websites. The column *Version* is the version of the tools that were used in the experiments. The column *Type* indicates if the tool is available as a plugin for the Eclipse IDE or as a standalone tool. The column *Languages* contains the programming languages of the source code that can be analyzed by the tools, with Java being the common language among them. The column *Refactoring* indicates whether the tool provides the feature of refactoring the code smell detected, which is available only in JDeodorant. The column *Export* indicates if the tool allows exporting the results to a file, a feature present only in inFusion and JDeodorant that export the results in a HTML file and a XML file,

**Table 1** Code smell detection tools

Tool	Version	Type	Languages	Refactoring	Export	Detection Techniques
inFusion	1.8.6 2015	Standalone	Java, C, C++	No	Yes	Software Metrics
JDeodorant	5.0.0 2015	Eclipse Plugin	Java	Yes	Yes	Refactoring opportunities
PMD	5.3.0 2015	Eclipse Plugin	Java, C, C++ and others	No	No	Software Metrics
JSPIRIT	1.0.0 2014	Eclipse Plugin	Java	No	No	Software Metrics

respectively. Finally, the column *Detection Techniques* contain a general description of the techniques used by each tool, with software metrics being the most common.

*inFusion* 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. As a commercial product, *inFusion* is no longer available for download at this moment. However, the open source version of the tool, called *iPlasma*,<sup>1</sup> is still available. The detection techniques for all smells were initially based on the detection strategies defined by Lanza and Marinescu (2006), and then successively refined using source code from multiple open source and commercial systems. *JDeodorant*<sup>2</sup> is an open source Eclipse plugin for Java that detects four code smells: God Class, God Method, Feature Envy, and Switch Statement (Tsantalis et al. 2008). The detection techniques are based in the identification of refactoring opportunities of *Extract Class* for God Class, *Extract Method* for God Method and *Move Method* for Feature Envy. *PMD*<sup>3</sup> is an open source tool for Java and an Eclipse plugin that detects many problems in Java code, including two of the code smells of our interest: God Class and God Method. The detection techniques are based on metrics. For God Class, it relies on the detection strategies defined by Lanza and Marinescu (2006) and for God Method, a single metric is used: LOC (lines of code). Finally, *JSPIRIT*<sup>4</sup> is an Eclipse plugin for Java that identifies and prioritizes ten code smells, including the three smells of our interest: God Class, God Method, and Feature Envy (Vidal et al. 2015). The detection techniques consist in the implementation of the detection strategies inspired by the work from Lanza and Marinescu (2006).

### 3 Study settings

This study aims at investigating code smells in software systems and evaluating the effectiveness of tools to detect them. We first present in Section 3.1 the selected software systems. Section 3.2 summarizes the reference lists of code smells identified in both systems. Section 3.3 defines the research questions we aim to answer. Section 3.4 presents the experimental measures used to answer the research questions.

#### 3.1 Target systems

This study analyzed two Java systems, namely MobileMedia and Health Watcher, with nine and ten object-oriented versions, respectively. We describe below these two target systems used in this study. We selected MobileMedia and Health Watcher as target systems mainly because the code smell experts responsible for manually analyzing the code to identify code smells were already familiar with these systems. The manual identification of code smells is a difficult task. Therefore, intimate knowledge of the system and its domain facilitates the comprehension of the source code. This allowed the experts to focus on identifying code smell instances instead of trying to understand the system, its dependencies, and other domain-related specificities. We also had other reasons for choosing the two systems: (i) we have access to their source code, allowing us to manually retrieve code smells, (ii) their code is readable, facilitating for instance, the task of identifying the functionalities implemented by classes and methods, (iii) these systems were previously used in other maintainability-related studies (Figueiredo et al. 2008) (Macia et al. 2012) (Soares et al. 2006) (Kulesza et al. 2006) (Greenwood et al. 2007).

### 3.1.1 MobileMedia (MM)

This system is a software product line (SPL) for applications that manipulate photo, music, and video on mobile devices (Figueiredo et al. 2008). Our study involved nine object-oriented versions (1 to 9) of MobileMedia, ranging from 1 to over 3 KLOC. Table 2 shows for each version of MobileMedia the number of classes, methods, and lines of code. We can observe that from versions 1 to 9 there was an increase of 2057 lines of code, 31 classes, and 166 methods.

### 3.1.2 Health Watcher (HW)

This application is a typical Web-based information system that allows citizens to register complaints regarding health issues (Soares et al. 2006). It is a real and non-trivial system that uses technologies common in day-to-day software development, such as GUI, persistence, concurrency, RMI, Servlets, and JDBC (Greenwood et al. 2007). Our study involved ten object-oriented versions (1 to 10) of Health Watcher, ranging from 5 KLOC to almost 9 KLOC. Table 2 shows for each version of Health Watcher the number of classes, methods, and lines of code. We can observe that from version 1 to version 10 there was an increase of 2706 lines of code, with the addition of 41 classes and 270 methods.

## 3.2 Code smell reference lists

The code smell reference list is a document containing the code smells identified in the source code of a software system. Table 3 contains the number of code smells for each version and the number of entities identified as God Class, God Method or Feature Envy in MobileMedia (MM) and Health Watcher (HW). To identify these code smells, we manually analyzed the source code of each system. The final code smell reference lists for each system was created in three phases.

In the first phase, two experts in code smells analyzed the systems independently to find code smells. Both experts analyzed each class and method individually using Fowler's description of code smells (Fowler 1999). In the second phase, the two experts discussed every potential code smell identified to resolve divergences. The entities classified by both experts as a code smell were registered in the final reference list for each system. In the third phase, the entities for which the experts disagreed

**Table 2** Metrics for MobileMedia (MM) and Health Watcher (HW)

Version	# of Classes		# of Methods		LOC	
	MM	HW	MM	HW	MM	HW
1	24	77	124	401	1159	5996
2	25	80	143	424	1316	6369
3	25	80	143	424	1364	6369
4	30	92	161	566	1559	7046
5	37	93	202	581	2056	7231
6	46	97	238	406	2511	7293
7	50	99	269	606	3015	7316
8	50	103	273	611	3167	7355
9	55	115	290	659	3216	8800
10	n/a	118	n/a	671	n/a	8702

**Table 3** Code smell reference list of MobileMedia (MM) and Health Watcher (HW)

Version	God Class		God Method		Feature Envy		Total	
	MM	HW	MM	HW	MM	HW	MM	HW
1	3	1	9	6	2	0	14	7
2	3	1	7	6	2	0	12	7
3	3	1	6	6	2	0	11	7
4	4	1	8	6	2	0	14	7
5	5	1	8	6	2	0	15	7
6	6	1	9	6	2	0	17	7
7	7	1	7	6	2	0	16	7
8	9	1	7	6	2	0	18	7
9	7	2	6	6	3	0	16	8
10	n/a	2	n/a	6	n/a	0	n/a	8
Total	47	12	67	60	19	0	133	72

were analyzed by a more experienced code smell expert that did not participate in the previous two phases. The code smells approved by this expert were registered in the final reference list for each system, along with the entities classified as code smells in the first and second phases.

### 3.3 Research questions

In this study, we analyze and compare the four code smell detection tools, namely inFusion, JDeodorant, JSPIRIT, and PMD. It aims at answering two research questions to compare the accuracy and agreement of these tools in detecting code smells. The research questions are presented and discussed below.

*RQ1. What is the accuracy of each tool in identifying relevant code smells?*

We aim to assess the accuracy of each tool in detecting code smells that were previously identified by experts and registered in the code smell reference list of each system. The results generated by the analysis of the source code of both systems by all tools were compared to the reference list to determine the precision and recall of each tool in the detection of God Class, God Method, and Feature Envy.

*RQ2. Do different detection tools agree on the same code smells?*

Different tools implement different detection techniques and sometimes the same technique can be implemented with variations specific to a particular tool, such as different threshold values. Therefore, it is expected that different tools identify different classes and methods as code smells. We aim to assess how much the tools agree when classifying a class or method as a code smell.

### 3.4 Experimental measures

This study evaluates and compares four code smell detection tools regarding their accuracy in detecting code smells and their agreement for the same system. We rely on recall and precision to measure their accuracy, while agreement is

measured by calculating the overall agreement and the  $AC_1$  statistic. Details are discussed in the following.

### 3.4.1 Recall and precision

To calculate recall and precision, we considered that *true positives* are instances (classes or methods) present in the code smell reference list that are also reported by the tool being assessed. *False positives* are instances that are not present in the reference list, but they were reported by the tool. *False negatives* are instances present in the reference list that were not reported by the tool. Finally, *true negatives* are instances that are not present in the reference list and were also not reported by the tool. Therefore, recall is the number of true positives divided by the number of instances in the reference list (true positives + false negatives), while precision is the number of true positives divided by the number of instances reported by the tool (true positives + false positives).

### 3.4.2 Overall agreement

The overall agreement or percentage agreement (Hartmann 1977) between two or more tools is the proportion of instances (classes or methods) that were classified in the same category by both tools, for the overall agreement between pairs, or by all tools, in the case of overall agreement between multiple tools. For each instance of a target system, there are two possible classifications: *smelly* or *non-smelly*. Therefore, the overall agreement is the number of instances classified in the same category (smelly and non-smelly) by the pair or set of tools, divided by the total number of instances in the target system. The total number of God Class instances is related to the total number of classes in the system, while the total number of instances for God Method and Feature Envy is related to the total number of methods in the system.

### 3.4.3 $AC_1$ statistic

The  $AC_1$  statistic (Gwet 2001), or first-order agreement coefficient, corrects the overall percentage agreement for chance agreement. That is, it minimizes the cases when tools randomly agree by classifying the same instances simultaneously as smelly or as non-smelly. Therefore, the  $AC_1$  statistic is the relative number of instances upon which tools are expected to agree from a set where instances classified by chance in identical categories were already removed. The  $AC_1$  statistic is calculated as  $(p - e(y))/(1 - e(y))$ , where  $p$  is the overall percentage agreement and  $e(y)$  is the chance agreement probability (Gwet 2001). The complete calculation and explanation of  $e(y)$  can be found in the book of Gwet (2001).

## 4 Comparative study of code smell detection tools

One of the aims of this study is to evaluate and compare four code smell detection tools, namely JDeodorant, inFusion, PMD and JSPIRIT. Section 4.1 presents a summary of detected code smells by experts and by the tools. Section 4.2 analyzes the tools accuracy in detecting code smells from the reference list. Section 4.3 analyzes the agreement among tools.



#### 4.1 Summary of detected code smells

This section summarizes the code smells detected in the two target systems using the four analyzed tools. The complete results for MobileMedia and Health Watcher are available on Additional files 1 and 2.

##### 4.1.1 MobileMedia (MM)

Table 4 shows the total number of code smells identified by each tool in the nine versions of MobileMedia. For God Class, JDeodorant reports the highest number of classes, reporting 85 classes, while the reference list has 47 classes and the other tools report less than 9. For God Method, JDeodorant reports 100 methods, while the reference list has 67 methods. On the other hand, JSpIRIT reports 27 God Methods, while PMD and inFusion report similar numbers, 16 and 17, respectively. For Feature Envy, JSpIRIT reports 74 methods, the highest number, followed by JDeodorant reporting 69 methods, while the reference list contains 19 methods. Lastly, inFusion reports only 9 instances of Feature Envy. Considering the total of smells reported, inFusion and PMD report similar totals of smells. However, inFusion is the most conservative tool, with a total of 28 code smell instances for God Class, God Method, and Feature Envy. PMD is less conservative, detecting a total of 24 instances for God Class and God Method, in contrast with the 20 instances detected by inFusion. PMD does not detect Feature Envy. JSpIRIT and JDeodorant report more smells than the previous tools. JDeodorant is by far the most aggressive in its detection strategies by reporting 254 instances. That is, JDeodorant detects more than nine times the amount of smells of the most conservative tools, namely inFusion and PMD. However, JSpIRIT is the tool that reports a total amount of code smells that is closer to the actual amount of 133 instances of the reference list for the nine versions of the MobileMedia system.

##### 4.1.2 Health Watcher (HW)

Table 4 shows the total number of code smell instances identified by each tool in the ten versions of Health Watcher. The reference list has only 12 God Classes, while the tools report more instances, except inFusion that reports none. JDeodorant reports the highest number of God Classes, reporting 98 instances, while PMD and JSpIRIT report lower numbers of classes, 33 and 20. For God Method, JDeodorant reports a very high number, 599 methods, while the reference list contains 60 methods. The other tools report fewer methods, with JSpIRIT reporting 30 methods, PMD reporting 13, and inFusion reporting none. There are no instances of Feature Envy in Health Watcher. However, JSpIRIT reported the highest number of methods, reporting 111 methods, while JDeodorant reported 90 and inFusion reported 48. Considering the total of smells reported, inFusion is once again the most conservative tool, reporting only instances of

**Table 4** Total number of code smells detected by each tool

Code Smell	inFusion		JDeodorant		JSpIRIT		PMD		Reference List	
	MM	HW	MM	HW	MM	HW	MM	HW	MM	HW
God Class	3	0	85	98	9	20	8	33	47	12
God Method	17	0	100	599	27	30	16	13	67	60
Feature Envy	8	48	69	90	74	111	–	–	19	0
Total	28	48	254	787	110	161	24	46	133	72

Feature Envy and none of God Class or God Method. PMD is the second more conservative, detecting a total of 46 instances of God Class and God Method. JSPIRIT and JDeodorant also detect more instances in Health Watcher than the other tools. JDeodorant is again the more aggressive in its detection strategy by reporting 787 instances. That is, it detects about 16 times the amount of smells of the most conservative tools, inFusion and PMD. JSPIRIT detects a little over twice the amount of actual instances of smells according to the reference list for Health Watcher.

#### 4.2 Analysis of accuracy

This section aims to answer the first research question (RQ1). We calculated the precision and recall based on the code smell reference list to assess the tools accuracy to detect smells from the reference list, i.e., relevant code smells. Tables 5 and 6 show the average of recall and precision considering all versions for each tool and code smell analyzed in MobileMedia and Health Watcher.

We decided to calculate recall and precision because they determine the accuracy of the tools. Tools with high recall can detect the majority of the smells of the systems when compared to tools with lower recall. Smells should be detected as soon as possible, because as the system evolves and becomes more complex, undetected smells may become increasingly harder to refactor. Tools with high precision report less false positives, i.e., most of the reported entities are actual code smells. Since the results have to be validated by the programmer, less false positives imply that the programmer spends less time inspecting non-smelly entities. In the literature, there is no common sense for recall and precision because the acceptable or desirable values depend on several factors, such as the system, the level of quality required, and the maintenance needs of the programmer. For instance, in a system that is critical, finding the highest number of smells is more important than reducing the manual validation effort. Therefore, recall is more relevant and a higher recall is desirable. On the other hand, if there are time constraints, it can be more important to reduce manual validation effort. Therefore, tools with higher precision and, therefore, that report less false positives are more desirable. However, the acceptable values for recall and precision have to be determined by the programmer that intends to use code smell detection tools.

##### 4.2.1 MobileMedia

Table 5 shows the tools average recall (R) and precision (P) for all nine versions of MobileMedia. An overview of data shows that the minimum average recall is 0% and the maximum is 58%, while the minimum average precision is 0% and the maximum 100%. From the results of Table 5, we made the following observations. For God Class, JSPIRIT and PMD have similar accuracy, i.e., lower average recalls of 17%, but higher precisions of 67 and 78% when compared to JDeodorant, with a 58% average recall and

**Table 5** Average recall and precision for MobileMedia

Code Smell	inFusion		JDeodorant		JSPIRIT		PMD	
	R	P	R	P	R	P	R	P
God Class	9%	33%	58%	28%	17%	67%	17%	78%
God Method	26%	100%	50%	35%	36%	93%	26%	100%
Feature Envy	0%	0%	48%	13%	0%	0%	n/a	n/a

**Table 6** Average recall and precision for Health Watcher

Code Smell	inFusion		JDeodorant		JSplRIT		PMD	
	R	P	R	P	R	P	R	P
God Class	0%	undef.	70%	8%	10%	10%	100%	36%
God Method	0%	undef.	82%	8%	17%	33%	17%	85%
Feature Envy	undef.	0%	undef.	0%	undef.	0%	n/a	n/a

28% average precision. Despite the highest average recall, JDeodorant reports many false positives, increasing the effort to validate the results by programmers. inFusion has the lowest average recall of 9%, however, it has an average precision of 33%, 5% higher than JDeodorant (28%).

For God Method, PMD and inFusion have the same accuracy, with an average recall of 26% and an average precision of 100%. Despite having the lowest recall for God Method, when compared to JDeodorant (50%) and JSplRIT (36%), inFusion and PMD have an average precision of 100%. Higher precision reduces greatly the validation effort of the programmer, but it can also increase the risk of missing relevant code smells. JDeodorant has the highest average recall of 50% and the lowest precision of 35%, values that are further away from the averages of the other tools. JSplRIT on the other hand, has averages closer to inFusion and PMD, with an average recall of 36% and an average precision of 93%.

For Feature Envy, inFusion and JSplRIT have the worst overall accuracy with 0% recall and 0% precision. JDeodorant has a better accuracy for Feature Envy when compared to both tools, with an average recall of 48% and an average precision of 13%. However, 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.

#### 4.2.2 Health Watcher

Table 6 shows the tools average recall (R) and precision (P) for all the ten versions of Health Watcher. An overview of the tables shows that the minimum average recall is 0% and the maximum is 100%, while the minimum average precision is 0% and the maximum 85%. From the results of Table 6, we made the following observations. For God Class, PMD has the best accuracy, with an average recall of 100% and the highest average precision of 36%. JDeodorant has the second highest average recall of 70% and the lowest average precision of 8%, with the exception of inFusion. inFusion has the worst accuracy, with an average of 0% for recall and undefined precision, since it did not detect any instance of God Class in any version of the system. Therefore, JSplRIT presents a better accuracy when compared to inFusion, with an average recall and precision of 10%.

For God Method, JDeodorant had the highest average recall of 82%. PMD and JSplRIT have the same average recall of 17%. However, PMD has a higher average precision of 85% when compared to JSplRIT (33%). inFusion has once again the worst average of recall (0%), since it did not detect any instances of God Method. In Health Watcher, there are no instances of Feature Envy. Nonetheless, all tools reported false positives; hence, they all present a 0% precision. Since there are no false negatives or true positives, recall is undefined.

### 4.3 Analysis of agreement

This section aims to answer RQ2. That is, it investigates the level of agreement among tools when applied to the same software system. We calculated the overall agreement and the  $AC_1$  statistic (Gwet 2001), which adjusts the overall agreement probability for chance agreement, considering all tools and pairs of tools.

#### 4.3.1 Overall agreement

Table 7 summarizes the results for overall agreement (OA) considering the agreement among all tools simultaneously. The values for percentage agreement vary between 0 and 100%. In the literature, the definition is that average agreement at or above 70% is *necessary*, above 80% is *adequate*, and above 90% is *good* (House et al. 1981) (Hartmann 1977). Since most averages for overall agreement between tools are higher than 80%, we considered that values equal or greater than 80% are *high*. Therefore, the overall agreement (OA) among tools is high, ranging from 83.33 to 98.03% for MobileMedia and is even higher for Health Watcher, ranging from 91.34 to 98.13%. Observing the standard deviation in Table 7, we can see that the results of the overall agreement (OA) found for each code smell in both systems do not present much variation, with standard deviation ranging from 0.609 to 2.041.

#### 4.3.2 Overall agreement between pairs of tools

The overall agreement was also calculated considering the agreement between pairs of tools for all versions of MobileMedia and Health Watcher. In general, in both systems, there was a higher average of agreement between tools that implemented the same detection technique. However, the agreement remained high even between tools with distinct techniques, indicating that the results obtained from different techniques are distinct, but still similar enough to yield high agreement values. Table 8 summarizes the overall agreement calculated between each pair of tools that allowed us to make the following observations for MobileMedia and Health Watcher. We use the following key in Table 8: inFusion (inf), JDeodorant (jde), PMD (pmd) and JSpIRIT (jsp). In addition, GC stands for God Class, GM for God Method, and FE for Feature Envy (FE).

**4.3.2.1 God Class** In MobileMedia, the pair PMD-JSpIRIT has the highest average agreement (99.15%), followed by the pairs inFusion-PMD (98.79%) and inFusion-JSpIRIT (97.94%). In Health Watcher, the same pairs have the highest averages, nevertheless, the ordering differs, with the pair inFusion-JSpIRIT (97.90%) first, followed by the pairs PMD-JSpIRIT (96.76%) and inFusion-PMD (96.59%). These three pairs of

**Table 7** Overall agreement (OA) for MobileMedia and Health Watcher

System	Code Smell	Mean	Standard Deviation	Median	Minimal	Maximal
Mobile Media	God Class	86.62%	2041	86.67%	83.33%	89.33%
	God Method	96.68%	0,634	96.35%	95.97%	97.63%
	Feature Envy	95.14%	1327	95.06%	93.55%	98.03%
Health Watcher	God Class	92.82%	0,794	93.03%	91.34%	93.79%
	God Method	94.89%	0,609	94.97%	93.58%	95.98%
	Feature Envy	97.39%	0,851	97.66%	95.74%	98.13%

**Table 8** Overall agreement (OA) of pairs of the tools

			inf-jde	inf-jsp	inf-pmd	jde-jsp	jde-pmd	pmd-jsp
MobileMedia	GC	Avg	75.12%	97.94%	98.79%	74.36%	74.36%	99.15%
		Median	73.33%	97.83%	100%	73.33%	73.33%	100%
		Std Dev	3.684	1.682	1.269	3.729	3.508	1.572
		Min	70.27%	96%	96%	67.57%	67.57%	96.15%
		Max	82%	100%	100%	82%	80%	100%
	GM	Avg	94.29%	99.25%	99.66%	93.70%	93.94%	99.24%
		Median	94.09%	99.01%	99.64%	93.10%	93.43%	99.01%
		Std Dev	1.863	0.615	0.327	1.488	1.706	0.618
		Min	91.94%	98.33%	99.01%	91.94%	91.94%	98.52%
		Max	97.07%	100%	100%	95.93%	96.65%	100%
	FE	Avg	95.52%	96.79%	n/a	93.12%	n/a	n/a
		Median	97.04%	97.53%	n/a	92.59%	n/a	n/a
		Std Dev	3.127	1.370	n/a	1.635	n/a	n/a
		Min	91.13%	94.88%	n/a	91.67%	n/a	n/a
		Max	99.51%	98.58%	n/a	97.04%	n/a	n/a
Health Watcher	GC	Avg	89.75%	97.90%	96.59%	87.82%	88.10%	96.76%
		Median	89.80%	97.90%	96.74%	87.76%	88.78%	96.76%
		Std Dev	0.914	0.289	0.425	1.483	1.880	0.397
		Min	88.31%	97.40%	95.76%	85.71%	84.42%	96.10%
		Max	91.30%	98.31%	97.09%	90.68%	89.83%	97.46%
	GM	Avg	89.92%	99.49%	99.77%	90.44%	90.15%	99.60%
		Median	90.06%	99.52%	99.84%	90.66%	90.26%	99.68%
		Std Dev	1.185	0.096	0.108	1.138	1.110	0.130
		Min	87.39%	99.29%	99.56%	88.05%	87.83%	99.34%
		Max	92.06%	99.58%	99.86%	92.48%	92.20%	99.72%
	FE	Avg	97.78%	97.27%	n/a	97.12%	n/a	n/a
		Median	98.17%	97.51%	n/a	97.45%	n/a	n/a
		Std Dev	0.915	0.676	n/a	0.980	n/a	n/a
		Min	95.98%	95.98%	n/a	95.27%	n/a	n/a
		Max	98.44%	97.98%	n/a	98.16%	n/a	n/a

tools also present a low standard deviation, ranging from 1.269 to 1.682 in MobileMedia and from 0.289 to 0.425 in Health Watcher. A low standard deviation means that there is not much variation in the level of agreement from one version of the system to another. The high agreement among these tools is because they all use the detection strategy defined by Marinescu (Lanza and Marinescu 2006). Although these tools use the same detection technique and agree on most classes, they disagree on others. The disagreement is due to the variation of the threshold values for the software metrics in the detection strategy.

The lowest average agreements occur in pairs with JDeodorant. JDeodorant detects God Class by searching for refactoring opportunities (Fontana et al. 2012), while inFusion, JSpIRIT and PMD use Marinescu's detection strategy (Lanza and Marinescu 2006). The usage of different techniques explains the lower agreement between the other tools with JDeodorant. However, the overall agreement considering all versions

can still be considered high, ranging from 67.57 to 82% in MobileMedia and from 84.42 to 91.30% in Health Watcher. The standard deviation between JDeodorant and the other tools is also higher than the other pairs of tools, with a minimum of 3.508 and a maximum of 3.729 in MobileMedia and a minimum of 0.914 and a maximum of 1.880 in Health Watcher. The higher standard deviation indicates a greater variation in the agreement between the other tools and JDeodorant from one version to another, when compared with the other pairs of tools.

**4.3.2.2 God Method** In MobileMedia, the pair inFusion-PMD has the highest average agreement (99.66%), followed by the pairs inFusion-JSpIRIT (99.25%) and PMD-JSpIRIT (99.24%). In Health Watcher, the same pairs have the highest averages, nevertheless, the ordering differs, with the pair inFusion-PMD (99.77%) first, followed by the pairs PMD-JSpIRIT (99.60%) and inFusion-JSpIRIT (99.49%). Interestingly, inFusion and JSpIRIT use the same detection strategy, while PMD uses a single metric LOC to detect God Method. Although the techniques are distinct, the high average agreement seems to indicate that both techniques are similarly effective in the identification of God Method. The standard deviation is also low, with a minimum of 0.327 and maximum of 0.618 for MobileMedia, and a minimum of 0.096 and maximum of 0.130 for Health Watcher. The low standard deviation supports the fact that the agreement between tools remains high across versions of both systems.

The lower average agreements are once again in pairs with JDeodorant. JDeodorant detects God Method using slicing techniques (Fontana et al. 2012), while inFusion and JSpIRIT use the detection strategy of Marinescu (Lanza and Marinescu 2006), and PMD uses the metric LOC. The usage of different techniques explains the lower agreement between the other tools with JDeodorant. However, the overall agreement considering all versions can still be considered high, ranging from 91.94 to 97.07% for MobileMedia and 87.39 to 92.48% for Health Watcher.

**4.3.2.3 Feature Envy** For MobileMedia, the pair inFusion-JSpIRIT has the highest average agreement (96.79%), followed by the pairs inFusion-JDeodorant (95.52%) and JDeodorant-JSpIRIT (93.12%). Both inFusion and JSpIRIT use Marinescu's detection strategy (Lanza and Marinescu 2006), with variations in the threshold values of the metrics. Therefore, the high agreement between these tools was expected. On the other hand, JDeodorant detects Feature Envy by searching for refactoring opportunities (Fontana et al. 2012). The different detection techniques lead to a lower agreement between JDeodorant and the other two tools. However, the agreement is still high, indicating that the detection techniques have similar results for detecting Feature Envy.

In Health Watcher, the variations between the pairs of tools are subtler. That is, there is not much variation in the average agreement. The pair inFusion-JDeodorant has the highest agreement (97.78%), followed by inFusion-JSpIRIT (97.27%) and JDeodorant-JSpIRIT (97.12%). The standard deviation has a minimum of 0.676 and maximum of 0.980, meaning there is not much variation of the agreement across the versions of the system. Although inFusion and JSpIRIT use the same detection technique and JDeodorant does not, for Health Watcher they yielded similar results.

### 4.3.3 $AC_1$ agreement

The  $AC_1$  statistic, or first-order agreement coefficient, is the conditional probability that two randomly selected raters agree given that there is no agreement by chance (Gwet 2001). This is a robust alternative agreement coefficient to Kappa (Gwet 2001) that is more sensitive to minor disagreements among the tools. The  $AC_1$  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). Tables 9 and 10 show the  $AC_1$  statistic (Gwet 2001) measured with a 95% confidence interval (CI). The  $AC_1$  statistic is "Very Good" for all smells and versions in both systems. This result is compatible with the high agreement found for the overall agreement among tools and between pairs of tools.

### 4.3.4 $AC_1$ statistic between pairs of tools

The  $AC_1$  was also calculated to consider pairs of tools with a 95% confidence interval. Figure 1 summarizes the classifications in each level of the Altman's benchmark scale for all versions of MobileMedia and Health Watcher. We can observe that for Health Watcher, there is a high agreement between all pairs of tools for all smells, since for all versions of the system, the  $AC_1$  values are "Very Good". For MobileMedia, the same happens for God Method and Feature Envy. However, for God Class, none of the pairs with JDeodorant have versions with an  $AC_1$  "Very Good".

For God Class in MobileMedia, the pairs with JDeodorant have  $AC_1$  either "Good" or "Moderate". This fact seems to support our analysis that for God Class, the detection technique of JDeodorant, when compared to the other tools, leads to different levels of agreement. However, we still believe that the agreement can be considered high, just not as high as the agreement among the other pairs of tools that do not include JDeodorant.

### 4.3.5 Final remarks on agreement

The overall agreement and the  $AC_1$  statistic have been calculated considering the agreement among all tools simultaneously and between pairs of tools. We found that the agreement is high among tools that implement the same detection technique with variations in threshold values. Different thresholds influence the results, but that analysis is beyond our

**Table 9**  $AC_1$  statistics of the analyzed tools for MobileMedia

Version	God Class		God Method		Feature Envy	
	$AC_1$	95% CI	$AC_1$	95% CI	$AC_1$	95% CI
1	0,831	(0.702,0.961)	0,958	(0.931,0.984)	0,930	(0.890,0.971)
2	0,814	(0.677,0.953)	0,959	(0.935,0.984)	0,939	(0.903,0.974)
3	0,826	(0.693,0.959)	0,960	(0.936,0.984)	0,935	(0.899,0.971)
4	0,848	(0.739,0.956)	0,970	(0.950,0.990)	0,943	(0.911,0.975)
5	0,802	(0.687,0.916)	0,960	(0.940,0.981)	0,980	(0.963,0.996)
6	0,874	(0.792,0.955)	0,976	(0.961,0.990)	0,959	(0.938,0.981)
7	0,880	(0.802,0.958)	0,972	(0.957,0.987)	0,948	(0.925,0.971)
8	0,844	(0.755,0.932)	0,962	(0.944,0.979)	0,951	(0.929,0.974)
9	0,870	(0.793,0.945)	0,972	(0.958,0.986)	0,950	(0.928,0.973)

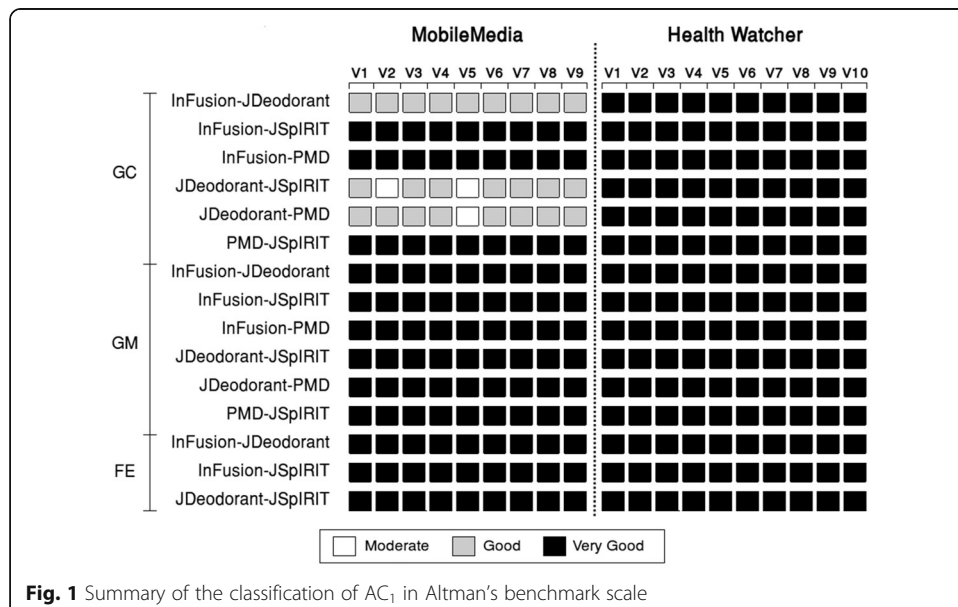
**Table 10** AC<sub>1</sub> statistics of the analyzed tools for Health Watcher

Version	God Class		God Method		Feature Envy	
	AC <sub>1</sub>	95% CI	AC <sub>1</sub>	95% CI	AC <sub>1</sub>	95% CI
1	0,905	(0.853,0.958)	0,947	(0.931,0.963)	0,956	(0.938,0.973)
2	0,909	(0.859,0.960)	0,931	(0.913,0.949)	0,958	(0.943,0.974)
3	0,926	(0.882,0.969)	0,944	(0.930,0.958)	0,973	(0.962,0.984)
4	0,919	(0.874,0.965)	0,945	(0.931,0.959)	0,974	(0.963,0.984)
5	0,920	(0.876,0.965)	0,947	(0.934,0.960)	0,978	(0.968,0.988)
6	0,924	(0.881,0.967)	0,948	(0.935,0.961)	0,978	(0.969,0.988)
7	0,926	(0.884,0.967)	0,949	(0.936,0.962)	0,980	(0.971,0.989)
8	0,929	(0.889,0.969)	0,950	(0.937,0.962)	0,981	(0.972,0.990)
9	0,932	(0.895,0.968)	0,942	(0.929,0.955)	0,980	(0.971, 0.989)
10	0,933	(0.897,0.969)	0,958	(0.947,0.969)	0,974	(0.964,0.984)

scope. Comparing the agreement between tools with the same technique with different thresholds and the agreement between tools with different detection techniques, we observed that the latter is lower, but not significantly. Therefore, we can consider that the agreement remained high even when comparing tools with different detection techniques. Investigating the results, we found that the high agreement is on true negatives, i.e., non-smelly entities. Comparing the detected entities with the code smell reference list, we concluded that the presence of a smell in a system can be considered somewhat a rare occurrence and this seems to be confirmed by the results of tools.

### 5 Code smell evolution

In this second study, we use the code smell reference lists (Section 3.2) to analyze the evolution of code smells in MobileMedia and in Health Watcher. Section 5.1 presents the



**Fig. 1** Summary of the classification of AC<sub>1</sub> in Altman's benchmark scale



number of code smells identified manually by experts in all versions of each system. Section 5.2 relies on visual representations to show how the code smells evolved in the systems.

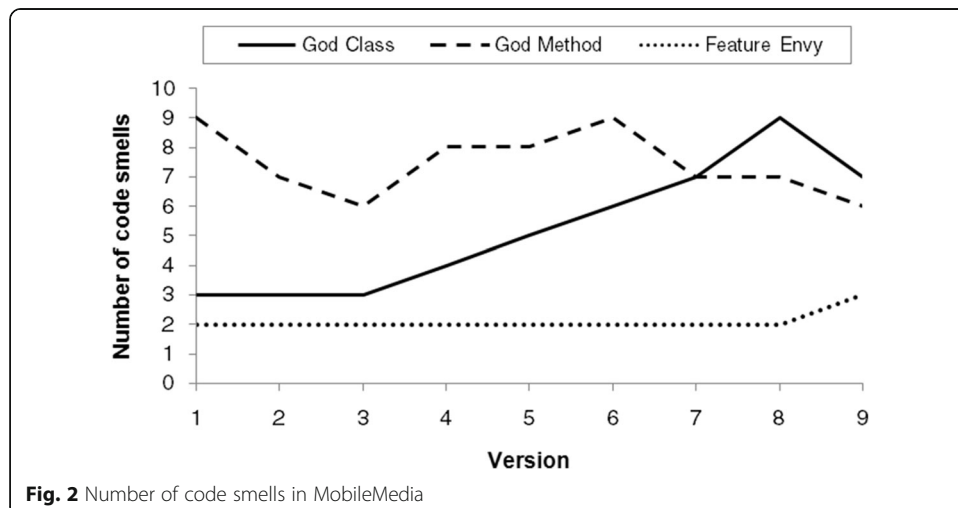
### 5.1 Analysis of the number of code smells

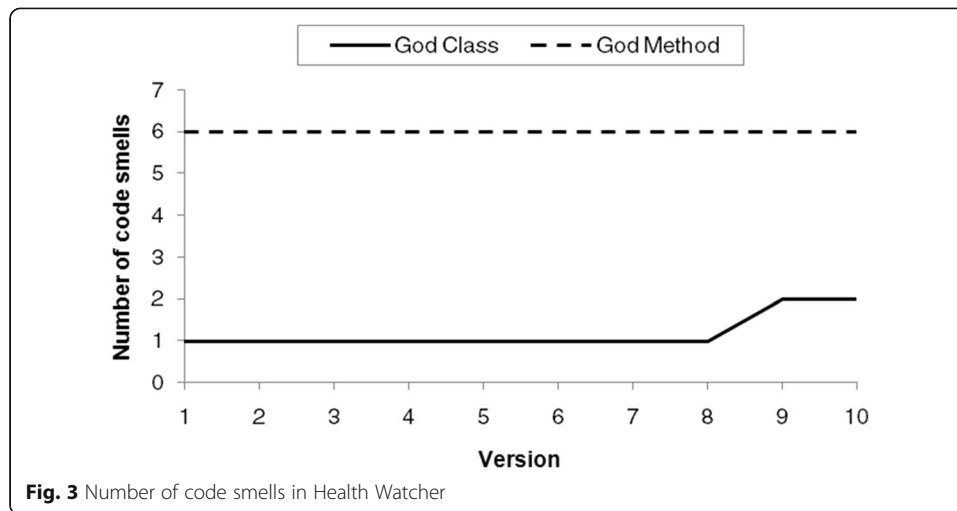
This section analyzes variations in the total number of code smell instances in the reference list of MobileMedia and Health Watcher as the systems evolved. Figures 2 and 3 present the number of code smells instances in the reference list per release of MobileMedia and Health Watcher, respectively.

#### 5.1.1 MobileMedia

According to Fig. 2, only the number of God Class increases with the system evolution. This result was expected, since the evolution of the system includes new functionalities and God Classes tends to centralize them. In version 1, three classes, namely `BaseController`, `ImageAccessor` and `ImageUtil`, were created smelly and remain God Classes in all versions. In addition, from versions 4 to 7, one God Class is introduced per version and two are added in version 8. These added God Classes are either new classes already created with the smell, or classes that were created in previous versions but only became smelly in subsequent versions. For instance, the `PhotoController` class was created in version 2 without any smell, but it became God Class in version 4 due to the addition of several new functionalities, such as displaying an image on screen and providing the image information. On the other hand, also in version 4, the new `AlbumController` class has already been created as God Class. Throughout the versions, some God Classes are eliminated by refactoring or by the removal of the class itself. However, more smelly classes were created than removed, leading to an increase in God Classes as the system evolves.

For God Method, the number of instances varies across versions, with intervals in which the total of smells increases or decreases. The first version contains the maximum number of God Methods, 9, when compared to any other version of the system, since there were only a few methods that concentrated the functionalities





of the system. However, as the system evolved, some of those methods were refactored, contributing to the decrease of God Methods, while others were either created as God Methods or became one with the addition of new functionality. For instance, the method `BaseController.handleCommand` was a God Method in versions 1 to 3. However, in version 4, the method was broken into other non-smelly methods, contributing to the decrease of smells. On the other hand, the initially non-smelly method `PhotoListController.handleCommand` in version 2 to 3 becomes smelly in version 4 due to the addition of functionalities, such as editing a photo label and sorting photos. Therefore, it contributed to the increase in the number of God Methods. Throughout the versions, the methods are frequently modified. The changes include: breaking a single method into multiple methods, adding functionalities, removing functionalities and merging methods. All these changes lead to the variations in the number of God Methods in the system, either increasing or decreasing the number of smells without a fixed pattern.

Finally, for Feature Envy, the number of instances remained constant from version 1 up to 8. Only the final version has one additional smell instance. The `ImageAccessor.updateImageInfo` and `MediaController.showImage` methods were already created with the smell, and only `MediaAccessor.updateMediaInfo` became smelly after creation. All these methods manipulate images and access directly data and methods from one or more classes that also manipulate images, such as the `ImageData` class. The similar role of image manipulation might have made it difficult for the developers to identify the correct class where the methods should have been placed and, consequently, they introduced Feature Envy instances in the system.

### 5.1.2 Health Watcher

Figure 3 show that, despite having more lines of code than `MobileMedia`, Health Watcher has no instances of Feature Envy. The number of God Classes and God Methods remains constant, with the addition of only one instance of God Class

in version 9. Analyzing the system code, we observed that the same God Classes and God Methods are present in all versions of the system. From one version to another, either the instances were not changed or functionality was added to them. However, no new smelly class or method was introduced. The only exception is the `HealthWatcherFacade` class that smells after version 9 with the addition of multiple new functionalities and, consequently, many lines of code.

## 5.2 Tracking code smells

This section aims to track instances of code smells from MobileMedia and Health Watcher in two steps. First, we selected the instances of the reference list. That is, classes or methods which at some point presented a code smell. We then tracked their states throughout the versions of both target systems.

### 5.2.1 Overall results

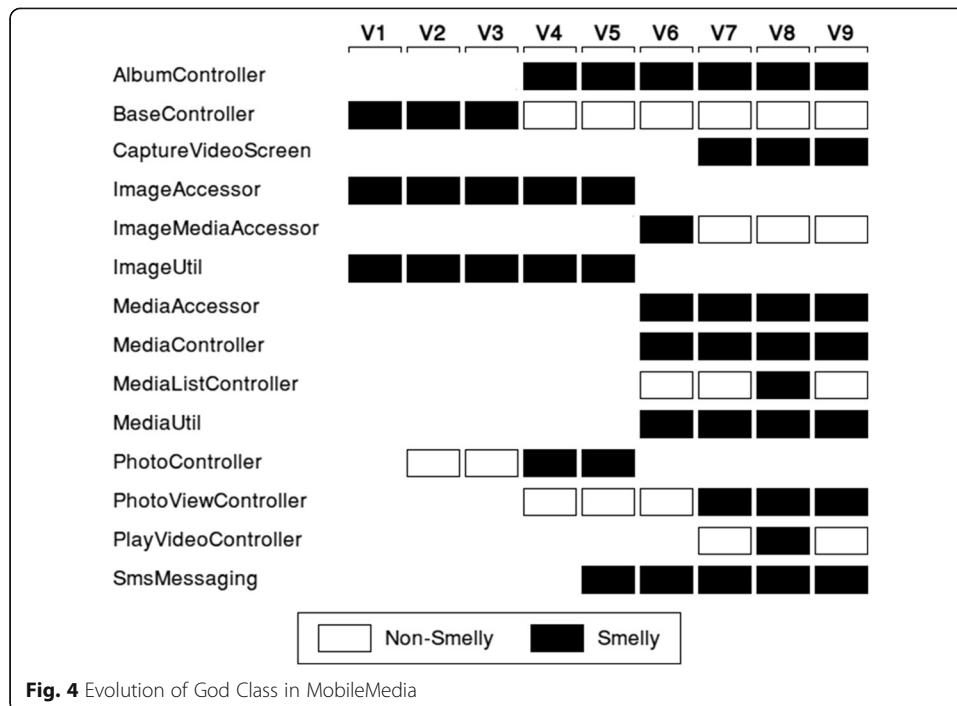
Table 11 summarizes quantitatively our findings of the evolution of code smells in MobileMedia (MM) and Health Watcher (HW). The columns “On Creation” indicate the number of classes or methods that were already introduced in the system as smelly. The columns “After Creation” indicate the number of classes or methods that became smelly in releases other than the one they were created. The columns “Total” indicate the total of smelly classes and methods considering all the versions of the system.

For all three analyzed smells in Mobile Media, 74.4% (32 of 43) of the smelly classes and methods were smelly from the beginning of their lifetime. Only about 25.6% (11 of 43) were initially non-smelly, but they acquired a smell in later versions. That is, only a fourth of the smells were introduced by changes in existing classes and methods, while the majority was created with a smell. Similarly, Table 11 also shows that there is no instance of Feature Envy in Health Watcher. In Health Watcher, for God Class and God Method, 7 out of 8 of the smelly classes and methods were smelly from the beginning of their lifetime. Therefore, in Health Watcher almost all smells were introduced at the creation of the class or method. Overall, our results showed that most of the identified code smells in MobileMedia and Health Watcher were already present at the creation of the affected class or method. That is the case for 74.4% of the smells in MobileMedia and 87.5% in Health Watcher, confirming the findings of Tufano et al. (2015). The complete analysis of the evolution of smelly classes and methods in all versions of MobileMedia and Health Watcher is discussed below using the Figs. 4, 5, 6, 7 and 8.

In Figs. 4, 5, 6, 7 and 8, each smelly class or method is represented by a row and each system version by a column. Each row is labeled with the corresponding class or

**Table 11** Number of code smells originated and created in classes and methods

Code Smell	On Creation		After Creation		Total	
	MM	HW	MM	HW	MM	HW
God Class	10 (71.4%)	1 (50%)	4 (28.6%)	1 (50%)	14	2
God Method	19 (76%)	6 (100%)	6 (24%)	0	25	6
Feature Envy	3 (75%)	0	1 (25%)	0	4	0
Total	32 (74.4%)	7 (87.5%)	11 (25.6%)	1 (12.5%)	43	8

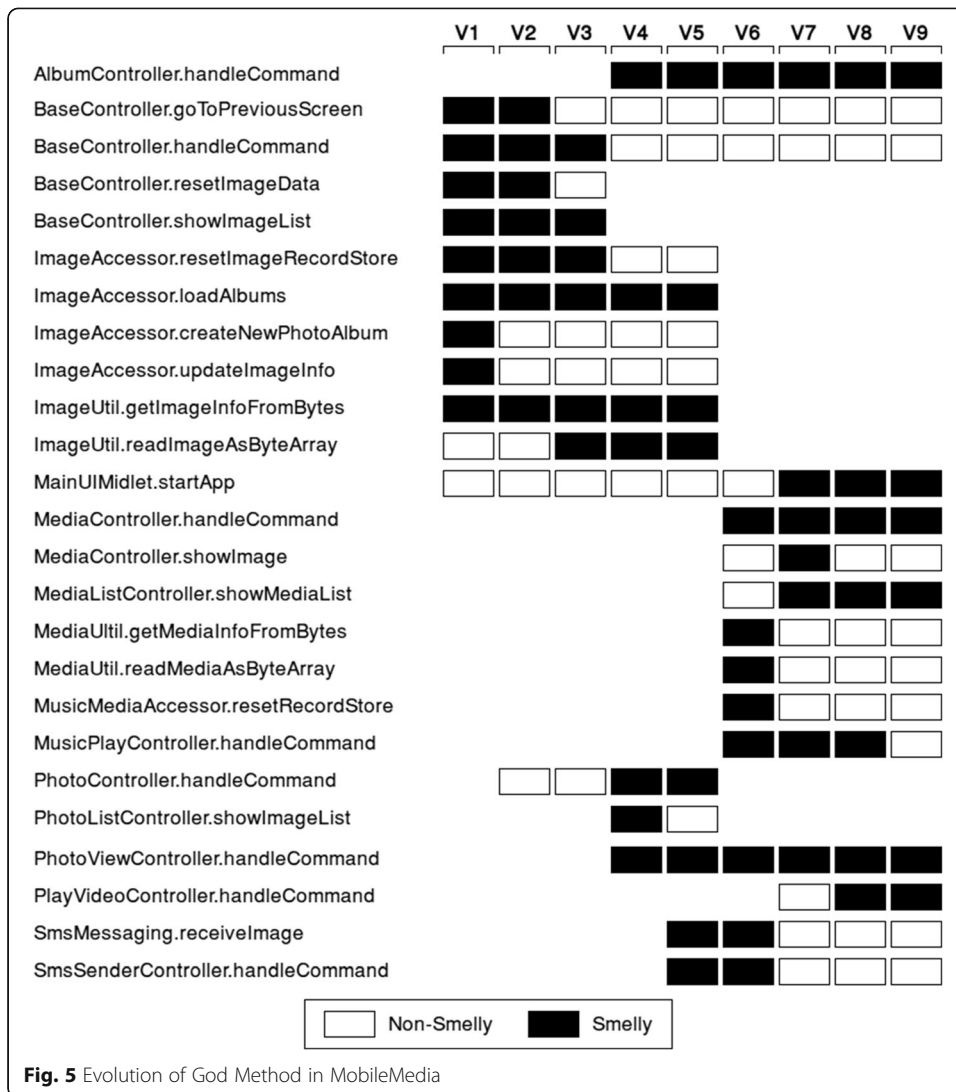


method name. Each rectangle represents the state of the class or method in the system version given by the column. There are two possible states: *white* and *black*. A *white* state indicates that the class or method is present in that system version, but it does not have a code smell. A *black* state indicates that the class or method is in fact a code smell in that system version. The absence of a square in a version means that the given class or method is not present in the respective version.

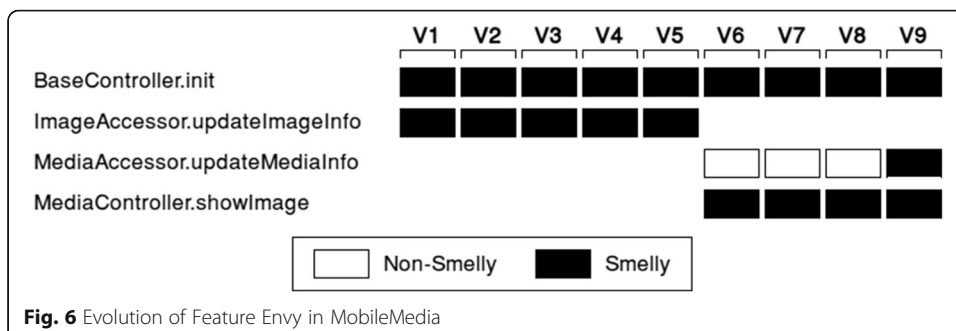
### 5.2.2 MobileMedia

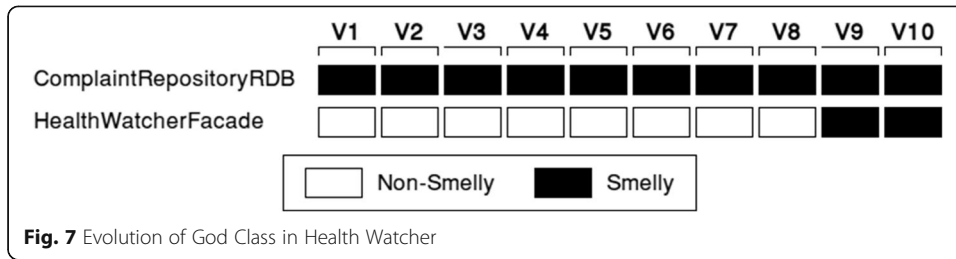
Figure 4 shows that for 10 out of 14 God Class instances, the code smells originate with the class. That is, the class is already created as a class that centralizes functionalities instead of a class to which functionalities are gradually included with each release of the system. This result is aligned with recent findings (Tufano et al. 2015). For instance, the `ImageAccessor` and `AlbumController` classes were created in versions 1 and 4, respectively, as God Classes and remained as such for as long as they are present in the system. On the other hand, 4 out of 14 classes were created non-smelly and became a God Class at some point of their lifetime. For instance, the `PhotoController` class is added in the second version of the system and it only became smelly in version 4, because of the incorporation of new features such as showing saved images, and updating the image information.

Figure 5 shows that some methods are created as God Method (19 of 25) and others become God Method with the evolution of the system (6 of 25). In the first case, the methods are introduced in the system already with much functionality. In the second case, methods are created with a single purpose, and as the system evolves, more functionality is added until the method becomes smelly. For instance, the method `BaseController.handleCommand` is introduced in the first version as a smelly



method, centralizing multiple functionalities, such as adding, saving and deleting photos and albums. By version 4, the method was refactored, and some of the previously mentioned functionalities alongside others were removed from the method, removing the smell. On the other hand, the `PhotoController.handleCommand` method is created in version 2 with a single functionality, saving photo labels. By version 4, multiple features were added, such as editing photo labels, sorting photos, and adding photos as favorite,



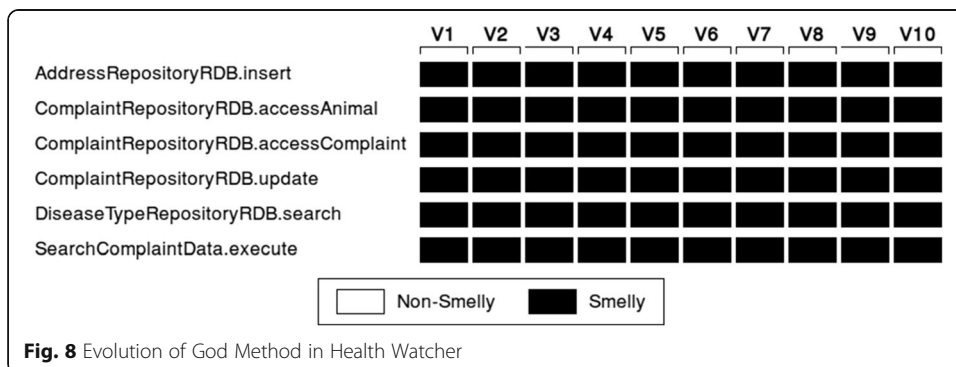


introducing a smell. One interesting observation is that when a smell is introduced in a method and then removed in a later version, the method remains non-smelly in the subsequent versions.

In Fig. 6, we observe that 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 it evolved to later present that code smell. The `MediaController.showImage`, `ImageAccessor.updateImageInfo`, and `MediaAccessor.updateMediaInfo` methods are smelly. They manipulate images and directly access data and methods from other classes that also manipulate images. Therefore, we believe that the similarity in the methods functionality may have led to confusion as to the correct class that each method should have been placed.

### 5.2.3 Health Watcher

There are no instances of Feature Envy in Health Watcher, therefore only the evolution of God Classes and God Methods is presented. Figure 7 shows the evolution of the only two God Classes in Health Watcher: `ComplaintRepositoryRDB` and `HealthWatcherFacade`. The former was created in the first version of the system, already as a God Class, and it remained as such throughout the entire evolution of the system. The latter is smelly only in versions 9 and 10, although the class was created in the first version. `ComplaintRepositoryRDB` is not modified from versions 1 to 6, suffering minor changes only in version 7, where a fragment of code that recovers information of a complaint in the database is reorganized, changing the order in which each field is displayed, while other fields became optional. Other minor changes are made in version 10, mainly in the order of statements and the inclusion of further information recovery means from the database. Nevertheless, the class remains smelly in all versions of the system. The class `HealthWatcherFacade` is created in version 1 and is modified in the versions 4 to 10. However, only the modifications in version 9 introduced a smell in



the class. This class is an implementation of the Façade design pattern (Gamma et al. 1994) that has the purpose of simplifying the access of underlying objects of the system. Therefore, it is expected to access data and methods from multiple classes. However, among the modifications in version 9, the inclusion of methods and the treatment of the exceptions in every method added multiple lines of code, making it a smelly class.

In Fig. 8, we observe that for God Method, all 6 instances were created with a code smell and the methods presented this smell during their entire existence. Analyzing the source code, we found that changes were minor, such as renaming variables, reordering statements and adding or removing types of exceptions caught or thrown by the methods. For instance, the method `AddressRepositoryRDB.insert` is only changed in version 10, where a few statements are placed in a different order from the previous versions. The same happens in version 10 with the methods `ComplaintRepositoryRDB.update` and `SearchComplaintData.execute`. Additionally, the `ComplaintRepositoryRDB.acesssComplaint` method stops throwing a `SQLException` in the version 10 of Health Watcher, while the method `ServletSearchComplaintData.execute` stops catching a `RemoteException`. In other words, the alterations in the methods affected only a few lines of code, without removing the smell.

## 6 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. 2012). The main factors that could negatively affect the internal validity of the experiment are the size of the subject programs, possible errors in the transcription of the result of tool analysis, and imprecision in the code smell reference lists. The subjects of our analysis are the nine versions of the MobileMedia and the ten versions of Health Watcher, which 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. Regarding the code smell reference lists, the manual detection of code smells is highly subjective and, therefore, it may vary from one expert to another. Even though this subjectivity can not be completely eliminated, we tried to reduce it by creating the code smell reference lists in well-defined stages and by discussing divergences between experts to reach a consensus.

*External Validity* concerns the ability to generalize the results to other environments (Wohlin et al. 2012). MobileMedia is a small open source system developed by a small team with an academic focus. Similarly, Health Watcher is a real-life system, but we only had access to a small portion of the source code. Therefore, these systems might not be representative of the industrial practice and our findings might not be directly extended to real large scale projects. However, to reduce this risk we selected systems from different domains, Mobile (MobileMedia) and Web (Health Watcher), which were developed to incorporate nowadays technologies, such as GUIs, persistence, distribution, concurrency, and recurrent maintenance scenarios of real software systems. Furthermore, these systems have been used and evaluated in previous research work (Figueiredo et al. 2008) (Macia et al. 2012) (Soares et al. 2006) (Kulesza et al. 2006) (Greenwood et al.

2007). We also selected these two systems because they have a comprehensible and familiar source code, allowing the experts to focus the analysis on code smell identification instead of code comprehension. However, additional investigation is necessary to determine if our findings can be generalized to other systems and domains.

*Conclusion validity* concerns the relation between the treatments and the outcome of the experiment (Wohlin et al. 2012). 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.

## 7 Related work

In the literature, there are many papers proposing new code smell detection tools (Marinescu et al. 2005) (Moha et al. 2010) (Murphy-Hill and Black 2010) (Zazworka and Ackermann 2010). A list of available detection tools was reported in a systematic literature review by Fernandes et al. (2016). In general, tools are evaluated individually and considering only a few smells. To our knowledge, Fontana et al. (2012) were the first to present a wider evaluation of several detection tools and code smells. They used six versions of a system to evaluate four tools, Checkstyle, inFusion, JDeodorant, and PMD. We also evaluate inFusion, JDeodorant, and PMD, calculating the agreement among these tools similarly to Fontana et al. (2012). However, we used the  $AC_1$  statistic, which is more robust than the Kappa Coefficient. Unlike Fontana et al. (2012) we also analyzed precision and recall of each tool.

Chatzigeorgiou and Manakos (2010) and Tufano et al. (2015) also analyzed multiple versions of systems to investigate the evolution of code smells. Chatzigeorgiou and Manakos (2010) investigated if code smells are removed naturally or by human intervention as the system evolves and if they are introduced with the creation of entities. Tufano et al. (2015) focused in identifying when and why smells are introduced in the system in a large empirical study of 200 open source projects. In our work, we also analyze the evolution of code smells, but at a higher level and neither focused on maintenance activities and refactoring, like Chatzigeorgiou and Manakos (2010), nor in the reasons why the smells were introduced, like Tufano et al. (2015).

Other studies proposed different approaches to detect code smells in software. Oizumi et al. (2016) proposed that code smells are related, appearing together in the source code to compose different design problems. In this approach, code smells are detected as agglomerations, unlike our work, where we focus on strategies that identify code smells individually. Another study by Fontana et al., (2015) applied 16 different machine-learning algorithms in 74 software systems to detect four code smells in an attempt to avoid some common problems of code smell detectors.

In our previous study (Paiva et al. 2015), we evaluated three code smell detection tools, namely inFusion, JDeodorant, and PMD and one target system, MobileMedia. This paper extends our previous work by including the tool JSPIRIT and the Health Watcher system to increase the confidence of our results and to favor generalization of our findings. Even though we included only one system, we found differences in the accuracy and agreement of tools when comparing the results of MobileMedia with those of Health Watcher.



In fact, after comparing the accuracy of MobileMedia and Health Watcher, we found that the precision of all tools for all smells is lower in Health Watcher than in MobileMedia. For instance, JDeodorant in MobileMedia has a precision of 28% for God Class, 35% for God Method, and 13% for Feature Envy, while in Health Watcher JDeodorant has a precision of 8% for God Class and God Method, and a precision of 0% for Feature Envy. The average recalls of inFusion and of JSpIRIT are lower in Health Watcher for all smells, and for PMD is lower only for God Method. For instance, for God Class, inFusion has a recall of 9% and JSpIRIT of 17% in MobileMedia, while in Health Watcher the average recall for inFusion is 0% and for JSpIRIT is 10%. On the other hand, recall of JDeodorant increased for God Class and God Method, from 58 and 50% in MobileMedia to 70 and 82% in Health Watcher. The recall of PMD for God Class also increased: while in MobileMedia the recall is 17%, in Health Watcher the recall is 100%. Regarding the overall agreement, in general the averages between tools are over 90% for MobileMedia and for Health Watcher. However, for God Class, pairs with JDeodorant have lower agreement in MobileMedia and higher agreement in Health Watcher. In MobileMedia the pairs inFusion-JDeodorant (75.12%), JDeodorant-JSpIRIT (74.36%), and JDeodorant-PMD (74.36%) have lower average overall agreements, while in Health Watcher, all pairs have an average agreement higher than 87.82%.

Variations in the tools results for MobileMedia and Health Watcher may be related with the fact that these systems are from different domains, Mobile (MobileMedia) and Web (Health Watcher). However, further investigation is necessary to determine the influence of the domain in the tools results. The comparison should be extended to include a larger number of systems from different sizes and domains. For instance, systems could be selected in a way that there are multiple sets of different domains and for each set there are multiple systems from the same domain. The results of the tools can then be compared inside the sets (same domain) and between sets (different domains).

## 8 Conclusions and future work

The comparison of code smell detection tools is a difficult task because these tools are based on informal definitions of the smells. The different interpretations of code smell by researchers and developers lead to tools with distinct detection techniques, results, and consequently, the amount of time spent with validation. In this paper, we used MobileMedia and Health Watcher as target systems, to evaluate the accuracy and the agreement of the tools inFusion, JDeodorant, JSpIRIT, and PMD. The accuracy was measured by calculating the recall and the precision of tools in detecting the code smells from the reference list. The agreement was measured by calculating the overall agreement and the  $AC_1$  statistic considering all tools simultaneously and between pairs of tools.

Considering the tools accuracy, we found that there is a trade-off between the number of correctly identified entities and time spent with validation. For all smells in both systems, JDeodorant identified most of the correct entities, but reports many false positives. A lower precision and a higher recall increase the validation effort, but capture most the affected entities. On the other hand, inFusion, JSpIRIT and PMD had higher precision, reporting more correct instances of smelly entities. A higher precision with a lower recall means that the tools do not report some of the affected entities. As the system evolves, it gets more complex, potentially increasing the level of difficulty to

refactor smells. Therefore, smells should be detected as soon as possible to ease refactoring activities. In that case, tools with higher recall can generate better results.

We also analyzed the agreement of the tools, calculating the overall agreement and the chance-corrected agreement using the  $AC_1$  statistic for all the tools and for pairs of tools. The overall agreement considering all the tools is high for all smells, with values over 80% in MobileMedia and over 90% in Health Watcher. Between pairs, the average overall agreement between the tools is also mostly in the acceptable range of 75 to 90% (Hartmann 1977). The  $AC_1$  statistic is also high with most values “Very Good”. By analyzing the results, we concluded that the high agreement was due to the agreement on non-smelly entities. Such observation is compatible with the code smell reference list and with the conclusions of Fontana et al. (2012).

Since we compiled the code smell reference list to measure the tools accuracy, we conducted a secondary study to analyze the evolution of code smells in nine versions of MobileMedia and in ten versions of Health Watcher. We found that for MobileMedia, only the number of God Classes increases as the system evolves, while for Health Watcher, the number is constant. We also found that most smelly classes and methods are already created with the smell. That is the case for 74.4% of the smells in MobileMedia and 87.5% in Health Watcher. This result confirms the findings of Tufano et al. (2015). Due to this result, Tufano et al. (2015) recommends that developers should apply code smell detection tools in classes and methods before *commits*, instead of running the detection tools in the entire system from time to time. However, in case running the tools during commit tasks is not possible, detection tools could be used before new releases to avoid or at least reduce the number of code smell instances introduced in the system.

In future work, we would like to expand our analysis to include other real-life systems from different domains and compare other code smell detection tools. Furthermore, we intend to investigate the influence of different domains in the analysis of detection tools. We also intend to investigate more the evolution of other code smells in a system and how their evolution is related to maintenance activities.

## 9 Endnotes

<sup>1</sup><http://loose.upt.ro/iplasma/>

<sup>2</sup><https://github.com/tsantalis/JDeodorant>

<sup>3</sup><https://pmd.github.io/>

<sup>4</sup><https://sites.google.com/site/santiagoavidal/projects/jspirit>

## 10 Additional files

**Additional file 1:** Complete results and evaluation for MobileMedia. (XLS 148 kb)

**Additional file 2:** Complete results and evaluation for Health Watcher. (XLS 222 kb)

### Abbreviations

$AC_1$ : First-order agreement coefficient; CI: Confidence interval; FE: Feature Envy; GC: God Class; GM: God Method; HW: Health Watcher; inf: inFusion; jde: JDeodorant; jsp: JSPLRIT; LOC: Lines of code; MM: MobileMedia; SPL: Software product line

### Acknowledgements

Not applicable.

**Funding**

No funding sources.

**Availability of data and materials**

Not applicable.

**Authors' contributions**

TP designed the study, collected the data, analyzed the data and drafted the manuscript. AD participated in the data collection, data analysis and in multiple quality assessments and revisions of the paper. EF provided guidance for the study design, for the analysis, and helped draft and review the manuscript. CS provided guidance for the study design, reviewed the manuscript and helped fine-tune the final draft. All authors read and approved the final manuscript.

**Ethics approval and consent to participate**

Not applicable.

**Consent for publication**

Not applicable.

**Competing interests**

The authors declare that they have no competing interests.

**Publisher's Note**

Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

**Author details**

<sup>1</sup>Department of Computer Science, Federal University of Minas Gerais, Av. Antônio Carlos, 6627, Belo Horizonte 31270-901, Brazil. <sup>2</sup>Department of Computer Science, Federal University of Bahia, Ondina, Salvador 40170-115, Brazil.

Received: 7 May 2017 Accepted: 24 September 2017

Published online: 06 October 2017

**References**

- Altman DG (1991) Practical statistics for medical research. Chapman & Hall, London
- Brown WJ, Malveau RC, Mowbray TJ, Wiley J (1998) AntiPatterns: Refactoring software, architectures, and projects in crisis. Wiley
- Chatzigeorgiou A, Manakos A (2010) Investigating the evolution of bad smells in object-oriented code. In: Proceedings of the 7th international conference on the quality of information and communications technology. IEEE, pp 106–115
- DeMarco T (1979) Structured analysis and system specification. Yourdon, New York
- Fernandes E, Oliveira J, Vale G, Paiva T, Figueiredo E (2016) A review-based comparative study of bad smell detection tools. In: Proceedings of the 20th international conference on evaluation and assessment in software engineering (EASE '16). ACM, article 18
- Figueiredo E, Cacho N, Sant'Anna C, Monteiro M, Kulesza U, Garcia A, Soares S, Ferrari F, Khan S, Castor F, Dantas F (2008) Evolving software product lines with aspects: an empirical study on design stability. In: Proceedings of the 30th international conference on software engineering. ACM, pp 261–270
- Fontana FA, Braione P, Zanoni M (2012) Automatic detection of bad smells in code: An experimental assessment. *J Object Technol* 11(2):1–38. doi:10.5381/jot.2012.11.2.a5
- Fontana FA, Mäntylä M, Zanoni M, Marino A (2015) Comparing and experimenting machine learning techniques for code smell detection. *Empir Softw Eng* 21(3):1143–1191. doi:10.1007/s10664-015-9378-4
- Fowler M (1999) Refactoring: improving the design of existing code. Addison-Wesley, Boston
- Gamma E, Vlissides J, Johnson R, Helm R (1994) Design patterns: elements of reusable object-oriented software. Addison-Wesley, Boston
- Greenwood P, Bartolomei TT, Figueiredo E, Dosea M, Garcia AF, Cacho N, Sant'Anna C, Soares S, Borba P, Kulesza U, Rashid A (2007) On the impact of aspectual decompositions on design stability: An empirical study. In: European conference on object-oriented programming. Springer, pp 176–200
- Gwet K (2001) Handbook of inter-rater reliability: how to measure the level of agreement between two or multiple raters. StatAxis Publishing Company, USA
- Hartmann D (1977) Considerations in the choice of inter-observer reliability estimates. *J Appl Behav Anal* 10:103–116. doi:10.1901/jaba.1977.10-103
- House AE, House BJ, Campbell MB (1981) Measures of interobserver agreement: Calculation formulas and distribution effects. *J Behav Assess* 3(1):37–57
- Kulesza U, Sant'Anna C, Garcia A, Coelho R, Staa A, Lucena C (2006) Quantifying the effects of AOP: a maintenance study. In: Proceedings of the 22nd international conference on software maintenance. ACM, pp 223–233
- Langellier G, Sahraoui HA, Poulin P (2005) Visualization-based analysis of quality for large-scale software systems. In: Proceedings of the 20th international conference on automated software engineering. ACM, pp 214–223
- Lanza M, Marinescu R (2006) Object-oriented metrics in practice. Springer, Heidelberg
- Macia I, Garcia J, Popescu D, Garcia A, Medvidovic N, von Staa A (2012) Are automatically-detected code anomalies relevant to architectural modularity?: an exploratory analysis of evolving systems. In: Proceedings of the 11th annual international conference on aspect-oriented software development. ACM, pp 167–178

- Mäntylä MV (2005) An experiment on subjective evolvability evaluation of object-oriented software: explaining factors and inter-rater agreement. In: Proceedings of the 2005 international symposium on empirical software engineering. IEEE, pp 287–296
- Marinescu C, Marinescu R, Mihancea PF, Ratiu D, Wetzel R (2005) iPlasma: an integrated platform for quality assessment of object-oriented design. In: Proceedings of the 21st IEEE international conference on software maintenance. IEEE, pp 25–30
- McCray G (2013) Assessing inter-rater agreement for nominal judgment variables. Paper presented at the Language Testing Forum, University of Nottingham, November 15-17 2013
- Moha N, Gueheneuc Y, Duchien L, Le Meur A (2010) DECOR: a method for the specification and detection of code and design smells. *Softw Eng IEEE Trans* 36:20–36. doi: 10.1109/tse.2009.50
- Murphy-Hill E, Black A (2010) An interactive ambient visualization for code smells. In: Proceedings of the 5th international symposium on software visualization. ACM, pp 5–14
- Oizumi W, Garcia A, Sousa LS, Cafeo B, Zhao Y (2016) Code anomalies flock together: exploring code anomaly agglomerations for locating design problems. In: Proceedings of the 38th international conference on software engineering. ACM, pp 440–445
- Paiva T, Damasceno A, Padilha J, Figueiredo E, Sant'Anna C (2015) Experimental evaluation of code smell detection tools. In: 3rd workshop on software Visualization, Evolution, and Maintenance (VEM), pp 17–24
- Riel AJ (1996) Object-oriented design heuristics. Addison-Wesley, Boston
- Soares S, Borba P, Laureano E (2006) Distribution and persistence as aspects. *Softw Pract Exp* 36(7):711–759. doi:10.1002/spe.715
- Travassos G, Shull F, Fredericks M, Basili VR (1999) Detecting defects in object-oriented designs: using reading techniques to increase software quality. In: Proceedings of the 14th conference on object-oriented programming, systems, languages, and applications. ACM, pp 47–56
- Tsantalis N, Chaikalis T, Chatzigeorgiou A (2008) JDeodorant: identification and removal of type-checking bad smells. In: Proceedings of the 12th European conference on software maintenance and reengineering. IEEE, pp 329–331
- Tufano M, Palomba F, Bavota G, Oliveto R, Di Penta M, De Lucia A, Shyvyanyk D (2015) When and why your code starts to smell bad. In: Proceedings of the 37th international conference on software engineering. IEEE press, pp 403–414
- Vale G, Albuquerque D, Figueiredo E, Garcia A (2015) Defining metric thresholds for software product lines: a comparative study. In: Proceedings of the 19th international conference on software product line. ACM, pp 176–185
- Vidal S, Vázquez H, Díaz-Pace A, Marcos C, Garcia A, Oizumi W (2015) JSplRIT: a flexible tool for the analysis of code smells. In: Proceedings of the 34th international conference of the chilean computer science society. IEEE, pp 35–40
- Wohlin C, Runeson P, Höst M, Ohlsson MC, Regnell B, Wesslén A (2012) Experimentation in software engineering. Springer, Heidelberg
- Yamashita A, Counsell S (2013) Code smells as system-level indicators of maintainability: An empirical study. *J Sys Softw* 86(10):2639–2653. doi: 10.1016/j.jss.2013.05.007
- Zazworka N, Ackermann C (2010) CodeVizard: a tool to aid the analysis of software evolution. In: Proceedings of the 4th international symposium on empirical software engineering and measurement. ACM, article 63

**Submit your manuscript to a SpringerOpen<sup>®</sup> journal and benefit from:**

- Convenient online submission
- Rigorous peer review
- Open access: articles freely available online
- High visibility within the field
- Retaining the copyright to your article

---

Submit your next manuscript at ► [springeropen.com](http://springeropen.com)

---