On the Detection of God Class in Aspect-Oriented Programming: An Empirical Study

Gustavo Vale, Luiz Paulo Ferreira, Eduardo Figueiredo

Software Engineering Lab (LabSoft) – Federal University of Minas Gerais (UFMG)
Belo Horizonte, MG – Brazil

{gustavovale, luizpcf, figueiredo}@dcc.ufmg.br

Abstract. *Software metrics have been used to detect design flaws, such as code smells, in software systems. This work investigates the use of a set of metrics to detect God Class in Aspect-Oriented Programming (AOP). God Class can be defined as a module that knows too much or does too much in the system. We aim to identify if a set of metrics is efficient to identify God Classes in aspect-oriented software systems. We design an experiment involving 29 participants organized in pairs and divided in two groups: the first one analyzes metrics for AOP and the second one did the same for Object-Oriented Programming. We rely on two equivalent software systems, which have been developed using both AspectJ and Java languages. We measured recall and precision for each group of subjects and, based on the results, we created a reference list of metrics that subjects used to detect the God Class code smell. We therefore identify a set of metrics that seems to be effective to detect God Class in AOP systems.*

Keywords — *empirical study; code smells; god class; metrics.*

1. Introduction

Nowadays, different technologies have been used to develop systems, such as aspects and objects. Software metrics are the key means for assessing the system maintainability [Chidamber and Kemerer, 1994; Fenton and Pfleeger, 1998]. The community of software metrics has traditionally explored quantifiable properties of modules, such as class coupling, cohesion, and source code size, in order to identify maintainability problems in software systems [Chidamber and Kemerer, 1994; Ferrari et al., 2010; Marinescu, 2004; Nguyen et al., 2011]. More specifically, software measurement has be seen as a pragmatic solution to find symptoms of particular design flaws, such as code smells [Lanza and Marinescu, 2006; Marinescu, 2004]. Code smells were proposed by Kent Beck in Fowler's book [Fowler et al., 1999] as a mean to diagnose symptoms that may be indicative of something wrong in the system code. It describes a situation where there are hints that suggest a design flaw.

God Class is one of the most well-known code smells aim to diagnose symptoms that may be indicative of something wrong in the system code [Fowler et al., 1999]. They describe a situation where there are hints that suggest a design flaw. Along the years, different code smells have been defined and catalogued [Zhang et al., 2011]. In this work, we explore only the God Class code smell. God Class describes a component that knows too much or does too much [Riel, 1996]. It represents a class that has grown

beyond all logic to become the class that does almost everything in the system [Riel, 1996]. In a different view, we can say that God Class implements too many concerns and, so, has too many responsibilities [Carneiro et al., 2010]. A concern is any important property or area of interest of a system that we want to treat in a modular way [Robillard and Murphy, 2007].

Chidamber and Kemerer (1994) (CK) proposed a set of metrics for object-oriented systems. These authors claim that their metrics can aid users in understanding design flaws and in predicting certain project outcomes and external software qualities, such as software defects, testing, and maintenance effort [Subramanyan and Krishnan, 2003]. CK metrics and others metrics are gradually growing acceptance in industry [Subramanyan and Krishnan, 2003]. For instance, Marinescu (2004) proposes a technique, called detection strategies, that combines software metrics, such as CK ones, to detect God Class in object-oriented systems.

In this paper, we investigate the use of source code metrics to detect a code smell, namely God Class, in aspect-oriented systems. We design an empirical study involving 29 participants that analyzed measurements in two systems, one developed in AOP and another in Object-Oriented Programming (OOP). These systems have the same functions. The OOP version of the system was used as baseline. We rely on a set of 9 metrics obtained by using the AOPMetrics tool [Ceccato and Tonella, 2004]. We selected AOPMetrics because it collects all metrics we need in AOP code. In fact, AOPMetrics is able to count more than 15 metrics divided in three groups: simple metrics, CK metrics, and package dependencies [Wroclaw, 2005]. Our set considers only metrics of the first and second groups. The metrics available in AOPMetrics apply to both classes and aspects; the term module is used to indicate either of the two modularization units.

Our experiment seems efficient to detect God Class in aspect-oriented systems. The high values of recall and precision show that the subjects agree with the detected God Classes. Some metrics are more useful to detect God Class in AOP. For instance our set of nine metrics can be reduced to six and high values of recall and precision are going to be obtained. The threshold value for different software orientation will be different because of the technology particularities.

The rest of this work is organized as follows. Section II describes the study settings. Section III reports and discusses the main results of this empirical study. Section IV report related works. Section V highlights the threats to this study validity and section VI concludes this study and points out directions for future work.

2. Study Settings

This study aims at evaluating the effectiveness of software metrics in detecting God Class in aspect-oriented systems. Therefore, we perform an experiment with a real software system developed with different code technologies in order to help developers to find this smell over the modules. This section describes the target system, the set metrics set used in the experiment, and the experimental tasks.

2.1. Target System

Our study involved a software system, called iBATIS [iBATIS, 2014]. This system was selected because it is a real software system and is a known system in academia. iBATIS was used in a previous study [Ferrari et al., 2010]. iBATIS is a persistence framework for object-relational mapping between SQL databases and objects in Java. For instance, suppose one has a database table PRODUCT and a Java class Product. iBATIS automates the integration between the two abstractions by allowing Java statements to retrieve a Java object directly from a SQL database.

iBATIS was originally implemented in Java and later refactored to AspectJ [Ferrari et al., 2010]. Therefore, it has both object and aspect oriented implementations. The system name has recently changed to MyBATIS[1]. It has an active support of 20 contributors over 340 classes (depends of the version). Its open-source code has passed through more than 200 releases over more than 13 years and the last version has about 6 KLOC. This longtime trajectory is followed by several changes in scope and architecture. In addition, the high developers turnover rates resulted in several misused practices and code smells scattered through the code.

2.2. Metrics Set

Nine different metrics were used in this study. These metrics can be organized in three categories: size, coupling and cohesion. Table 1 presents a brief definition of the metrics. We choose this set of metrics because they are well known and used in other studies [Carneiro et al., 2010; Chindamber and Kemerer, 1994; Figueiredo et al., 2008; Lanza and Marinescu, 2006; Marinescu, 2004; Padilha et al., 2013; Padilha et al., 2014. All nine metrics can be measured in a unique tool [Wroclaw, 2005].

Table 1. List of used metrics

Attributes	Metrics	Definitions
Size	Lines of Class Code (LOC)	It counts the lines of code per class.
	Weighted Operations in Module (WOM)	It counts number of operations (Methods) in a given module. It is an equivalent to WMC [Chidamber and Kemerer, 1994].
Coupling	Depth of Inheritance Tree (DIT)	It is a length of the longest path from a given module to the class/aspect hierarchy root.
	Number Of Children (NOC)	It is a number of immediate subclasses or sub-aspects of a given module.
	Coupling on Method Call (CMC)	It is a number of modules declaring methods that are possibly called by a given module.
	Coupling on Field Access (CFA)	It is a number of modules or interfaces declaring fields that are accessed by a given module.
	Coupling between Modules (CBM)	It is a number of modules or interfaces called by a given module. It is an equivalent of the CBO [Chidamber and Kemerer, 1994]. This metric is a combination of CFA and CMC.
	Response For a Module (RFM)	It is number of methods and advices potentially executed in response to a message received by a given module.
Cohesion	Lack of Cohesion in Operations (LCO)	It is the number of pairs of operations that do not access common fields minus the pairs of operations that do (zero if negative). It is equivalent to LCOM [Chidamber and Kemerer, 1994].

[1] http://blog.mybatis.org/.

2.3. Experimental Tasks

This study involved a set of 29 participants, named P1 to P29, from Federal University of Minas Gerais – Brazil. Participants were 19 undergraduate students and 10 graduated students. We organized participants in pairs such a way, if possible, one graduated and one undergraduate. This way we had 14 subjects, 13 pairs and 1 group of 3 participants. We divided the subjects in two groups; each group is composed of 7 subjects. We estimated that 60 minutes was enough to conclude the experiment. This time is based on a previous experiment that one of the authors run. Therefore, we provide 100 minutes for the subjects to delimitate a deadline. The study was divided in two steps. The first step is summarized by background questionnaire and the second step is represented by experiment questionnaire. When the subjects start or finished one step they should put their spent time on questionnaire.

The background questionnaire is to analyze if their knowledge have some influence on the final result. We asked three main questions for each participant: (i) the work experience; (ii) the technology experience that participant was going to use; (iii) and the metrics experience. The experiment questionnaire was composed by the system description, God Class description, metrics definition and the following two questions:

(i) *What metrics do you think that are useful to identify a God Class? Choose threshold values that you think to be the most appropriate ones for the useful metrics. Briefly explain why you choose each threshold value. What detection strategy did you use?*

(ii) *Using the metrics you think are the most appropriate ones, identify what are the classes with the highest probability of having the God Class code smell;*

We analyze the coherence of the answers of the two questions to validate them. All answers were coherent and none of them were discarded. Each subject presented one different detection strategy. Hence, we decided did not report anyone. The versions that we analyze have 250 for AspectJ and 225 classes Java system. Therefore, we suppose that is too many classes for the subjects analyze and they could lose the interest and do not run one good experiment. For minimize this, we selected twenty percent of classes randomly by each system, totalizing 50 classes for AspectJ system and 45 classes for Java system. It is important to say that: (i) no IDE was used and (ii) subjects have no access to the system source code. Therefore, they analyzed mainly the metrics available.

3. Evaluation

The evaluation was made in five different main aspects: background of the subjects, God Classes, thresholds, time spent and metrics. Besides, the calculated percentages at recall and precision are used in order to quantify the results in a more comprehensible manner. Numbers for AOP and OOP are put side by side for comparison.

3.1. Background of Subjects

Subjects past experience on the area was collected through multiple choice questions. They were divided in (i) Experience, (ii) AspectJ/Java knowledge and (iii) Metrics experience.

Table 2 shows group one background and Table 3 shows group two background. Both tables are structured in the same way. Subjects were classified with some experience if they had worked one year or more, classified that knows about the technology if they had some knowledge about AspectJ for group one and Java for group two and we classified that they knows about metrics if they had any knowledge about software metrics.

Table 2. Background of participants in Group One

Experience	Participants
***(i)* Work**	
Less than 1 year	P8; P12
More than 1 year	P1; P2; P3; P4; P5; P6; P7; P9; P10; P11; P13; P14
***(ii)* AspectJ**	
Don't know about	P3; P4; P8; P11; P13; P14
Know something about	P1; P2; P5; P6; P7; P9; P10; P12
***(iii)* Metrics**	
Don't know about	P6; P8; P11; P13; P14
Know something about	P1; P2; P3; P4; P5; P7; P9; P10; P12

Table 3. Background of participants in Group Two

Experience	Participants
***(i)* Work**	
Less than 1 year	P15; P16; P21; P22; P24; P26; P27; P28; P29
More than 1 year	P17; P18; P19; P20; P23; P25;
***(ii)* Java**	
Don't know about	{}
Know something about	ALL
***(iii)* Metrics**	
Don't know about	P18
Know something about	ALL – P18

Comparison between two groups shows some differences that could be interesting to analyze for the ultimate results, but some care need to be taken on the data analysis. For example, group one has clearly greater proportion of work-experienced subjects than group two. On the other hand, group two has greater experience about the technology and metrics than group one.

In general, we have observed: (i) about 60% of the subjects have moderate work experience; (ii) 79% of the subjects have some technology experience (AspectJ or Java depends of the group); (iii) 80% of the subjects have some experience about metrics. Therefore, in general, all subjects have at least basic knowledge required to perform the experimental tasks.

We considered that if someone has some experience, this experience should be considered. For example, if someone of the pair knows about the technology, the pair knows about the technology. Analyzing each pair about the experience and spent time (Table 4) can be observed that only the subjects OOP1, OOP4, and OOP7 have never been worked; only the subjects AOP2 and AOP7 do not know about the technology; and only the subject AOP7 does not know about metrics.

Table 4. Experience of subjects

Subjects	Experience		
	Work	Technology	Metric
P1 and P2 = AOP1	Yes	Yes	Yes
P3 and P4 = AOP2	Yes	No	Yes
P5 and P6 = AOP3	Yes	Yes	Yes
P7 and P8 = AOP4	Yes	Yes	Yes
P9 and P10 = AOP5	Yes	Yes	Yes
P11 and P12 = AOP6	Yes	Yes	Yes
P13 and P14 = AOP7	Yes	No	No
P15 and P16 = OOP1	No	Yes	Yes
P17 and P18 = OOP2	Yes	Yes	Yes
P19 and P20 = OOP3	Yes	Yes	Yes
P21 and P22 = OOP4	No	Yes	Yes
P23 and P24 = OOP5	Yes	Yes	Yes
P25 and P26 = OOP6	Yes	Yes	Yes
P27, P28 and P29 = OOP7	No	Yes	Yes

3.2. Reference List

To evaluate the results, we had created a Reference List, once there is no other study that specifies what classes are God Classes in iBATIS. To create this list we analyze the source code and the measures. We supposed that five and seven classes are God Classes in Aspect and Object systems, respectively. After, we used the information fulfilled in our experiment, considering that a class has the code smell God Class if more than 50% of the pairs identified the class as a God Class who has this code smell. The classes that more than 50% of subjects considered God Class was exactly the God Class that we identified. Tables 5 and 6 present all classes chosen by the subjects.

Table 5. Result of God Class for the aspect group

Classes	Subjects	%
BaseStatement	AOP1, AOP2, AOP6, AOP7	57.14
BasicParameterMap	ALL – AO6	85.71
JavaBeanProve	AOP6	14.29
SqlMapClientImp	ALL – AOP3	85.71
Stopwatch	AOP1, AOP2, AOP5, AOP6	57.14
XmlDaoMangerbuilder	ALL	100.00

Table 6. Result of God Class for the object group

Classes	Subjects	%
AutoResultMap	OOP1, OOP3, OOP4	42.86
CacheModel	OOP1,OOP2, OOP3, OOP4, OOP5	71.43
ClassInfo	OOP1, OOP3, OOP4, OOP5, OOP6	71.43
JavaBeanProbe	OOP1, OOP3, OOP4, OOP7	57.14
PaginatedArrayList	OOP1, OOP2, OOP3, OOP4, OOP5, OOP6	85.71
ScriptRunner	ALL	100
SqlExecuter	ALL	100
SqlTag	OOP1, OOP3, OOP4, OOP5	57.14

The group one has 5 classes marked as God Classes the follows: BaseStatement, BasicParameterMap, SqlMapClientImp Stopwatch and XmlDaoMangerBuilder (Table V). The group two has 7 classes marked as God Classes the follows: CacheModel, ClassInfo, JavaBeanProbe, PaginatedArrayList, ScriptRunner and SqlTag (Table 6).

3.3. Recall and Precision

Data here was derived from both direct and indirect measures. Number of subjects that picked some class is an example of the former, while Recall and Precision illustrate the latter. The OOP version of the system was used as baseline to AOP version.

We rely on two metrics, namely True Positive (TP) and False Positive (FP), collected based on Reference List. True Positive and False Positive quantify the number of correctly and wrongly identified code smells by a subject. Based on these metrics, we quantify recall and precision, presented below, to support our analysis. Recall measures the fraction of relevant classes listed by a subject. Relevant classes are classes that appear in Reference List. Precision measures the ratio of correctly detected code smells by the total classes a subject listed (TP + FP).

$$\text{Recall (R)} = \frac{TP}{TP + FN} \qquad \text{Precision (P)} = \frac{TP}{TP + FP}$$

We focus our discussion mainly on precision because it is a measure of completeness [Padilha et al., 2014]. That is, high recall means that the subject was able to identify most code smells in the system. High precision, on the other hand, means that a subject indicated more relevant (TP) than irrelevant (FP) code smells. For code smell detection, a large number of false positives are preferred over a large number of false negatives, because manual inspection, which is inevitable, tends to uncover false positives.

Table 7. Results for Recall and Precision for group one

Pair	AOP1	AOP2	AOP3	AOP4	AOP5	AOP6	AOP7
R(%)	100	100	40	60	80	80	80
P(%)	100	100	100	100	100	80	100
Time(min)	34	27	30	20	45	50	31

Table 8. Results for Recall and Precision for group two

Pair	OOP1	OOP2	OOP3	OOP4	OOP5	OOP6	OOP7
R(%)	100	57.14	100	71.43	85.71	85.71	42.86
P(%)	87.5	100	87.5	83.33	100	100	100
Time(min)	51	50	45	41	45	31	25

Tables 7 and Table 8 show the recall, precision, and spent time for all pairs of group one and group two. The group one is composed by the pairs AOP1, AOP2, AOP3, AOP4, AOP5, AOP6, and AOP7, The group two is composed by OOP1, OOP2, OOP3, OOP4, OOP5, OOP6, and OOP7. For example, the pair AOP1 had 100% of recall, 100% of precision and spent 34 minutes to conclude the experimental tasks.

We can observe that the times for both groups are sparse. For example the faster subject on group one spent 20 minutes and the slower spent 50 minutes the difference is bigger than the time of faster subject. In spite of this, the faster subject has high precision than the slower subject. On the second group, the same behavior can be seen. Therefore, it is difficult to get some conclusion about time and efficiency.

The three pairs with low work experience (OOP1, OOP4, OOP7) had different and sparse values of recall and precision. Hence. it is difficult to make conclusion. The two subjects (AOP2 and AOP7) that reported that they do not have technology experience had high values of recall and precision. The subject AOP2 had 100% of recall and precision and AOP7 does not have experience with metrics too. However, AOP7 had high values of recall (80%) and precision (100%). Another interesting point is that AOP2 and AOP7 spent less time than the average of the subjects of group one. Therefore, it is hard to make some conclusions about time, experience and efficiency in our experiment.

3.4. Hard to Define Thresholds

We collected the thresholds for each metric (Tables 9 and 10). However, we do not aim to define any heuristic in our experiment. Each subject used a different approach to identify if a class has God Class code smell or not. This made every pair to choose a different metric to evaluate the code smell. They had to choose a threshold depending of the combination of metrics that they had chosen.

Table 9. Results of thresholds per metric from group one

Pair	AOP1	AOP2	AOP3	AOP4	AOP5	AOP6	AOP7
LOC	80	81	100	100	100	150	-
WOM	10	14	-	15	12	10	10
CMC	4	-	-	-	10	10	-
CFA	-	-	-	-	-	2	-
CBM	5	-	5	4	-	10	5
RFM	15	-	-	-	-	-	25
LCO	50	80	30	50	80	10	80

Table 10. Results of thresholds per metric from group two

Pair	OOP1	OOP2	OOP3	OOP4	OOP5	OOP6	OOP7
LOC	99	190	120	195	75	54.75	200
WOM	8	20	10	14	8	6.31	15
CMC	2	6	-	-	-	-	-
CFA	-	-	-	-	-	-	-
CBM	-	6	-	6	-	1.02	-
RFM	12	20	10	-	-	-	20
LCO	5	110	100	-	10	19.22	-

We choose nine metrics to compose the experiment, but only seven were chosen by the subjects (Table 11). The metric CFA was chosen by only one subject of our experiment. The metrics LOC, WOM and LCO were chosen by almost all subjects in group one and the metrics LOC and WOM were chosen by all subjects in group two

(Table 11). We can observed that the three metrics (LOC, WOM and LCO), in general, were the most useful. The metric CBM was chosen by more than 50% of the subjects.

Table 11. Percentage of subjects who used each metric

Metrics	AOP(%)	OOP(%)	Δ(%)	Total Average (%)
LOC	85.71	100	16.77	92.86
WOM	85.71	100	16.77	92.86
CMC	42.86	28.57	-33.33	35.71
CFA	14.29	0.00	-100.00	7.14
CBM	71.43	42.86	-40.00	57.14
RFM	28.57	57.14	100.00	42.86
LCO	100	71.43	-28.57	85.71

Therefore, the subjects reduced our set of 9 metrics to 7 useful metrics. As CFA was chosen by only one subject we can considered 6 metrics useful: LOC, WOM, CMC, CBM, RFM and LCO. Therefore, we are agreeing with the majority of subjects 4 metrics (LOC, WOM, CBM and LCO) are able to identify God Classes with high recall and precision in object and aspect oriented software systems. One important point to highlight is that this set of 4 metrics includes the three attributes that explicitly stated in the code smell definition, namely coupling (CBO), cohesion (LCO), and size (LOC and WOM).

3.5. Join data Analysis

Table 12 presents the average recall and precision of both systems (AOP and OOP). The recall values are almost the same. They have one small difference of 0.53%. The precision are better than recall and the difference is approximately 3% from AOP and OOP systems. The subjects whose run the experiment with AOP systems had more classes to analyze (50 – AOP and 45 – OOP). However, they spent less time than OOP. We believe that the work experience can have affected this result.

Table 12. Average values of God Class

	AOP	OOP	Δ(%)
Recall	77.14%	77.55%	0.53
Precision	97.14%	94.05%	-3.19
Time(min)	33.86	41.14	21.52

Table 13 presents the average of thresholds values for each metric. It is difficult to have some conclusion because in some cases the values are similar, such as WOM. On the other hand, in more often cases, the threshold values are distant such as LOC and CMC. Another point that should be considered is that subjects with less experience can provide unreal threshold values and distort the average.

4. Related Work

Several metrics have been proposed and combined to identify code smells in software systems. However, it is difficult to count metrics without any tool that automates the measurement. Identified this point, one alternative way is the use of tools that compute metrics at the module level.

Table 13. Average thresholds values per metric

	AOP	OOP	Δ(%)
LOC	101.83	133.39	30.99
WOM	11.83	11.62	-1.84
CMC	8.00	4.00	-50.00
CBM	5.80	4.34	-25.17
RFM	20.00	15.50	-22.50
LCO	54.29	48.84	-10.02

Padilha and her colleagues (2014) performed an empirical study involving 54 subjects to analyze the better set of metrics to compute three code smells (Divergent Change, Shotgun Surgery, and God Class). They analyzed three set of metrics: one with CK metrics, other involving concern metrics, and other with both CK and concern metrics. As results, concern metrics are clearly useful to detect Divergent Change and God Class and the effectiveness of each metric suite is largely dependent on the adequacy of each metric to quantify a property explicitly mentioned in the code smell definition.

In another work, Padilha et al. (2013) focus on detecting God Method using concern metrics. This empirical study involves 47 subjects. Three set of metrics were used, and the focus is to assess the concern metrics. As results, the concern metrics were efficient to detect God Method code smell.

The main difference of the two related works for our study is that we are analyzing a set of traditional metrics in AOP and we do not have a previous reference list of code smells. In addition, we consider the threshold values of different technologies, but this correlation was not possible. Another difference is that we do not evaluate metrics, we assume that our set of metrics are useful.

5. Threats to the Study Validity

This section discusses possible threats to the validity of our study. First, the experiment was made with students in a class and the subjects may not represent correctly a company work force or all the IT professionals. We randomly distributed graduate and undergraduate students in different groups. However, even with this action, we cannot guarantee that our experiment subjects are heterogeneous enough to generalize our results. The same can be occurred as the selected classes. We selected 20% of the systems classes randomly, and so we cannot guarantee that our selection cut was the best to be analyzed.

We trust that AOPMetrics collect correct code metrics values. Therefore, we did not check the correctness of AOPMetrics. We try to measure the subjects' experience, but we cannot obtain successful. We made the experiment with only one system and, so, we cannot guarantee that the same results hold for different systems. Additionally, some subjects may have tried to guess the hypotheses that we were investigating and, therefore, they may have based their behavior on their guesses. Finally, as some students did not know the motivation of the experiment, they could have to think that we were evaluating them. Therefore, their response can be biased trying to getting better grades instead of answer what they truly believe.

6. Conclusions and Future Work

Software metrics and code smells are convenient ways of finding critical problems in a software system. This paper showed how these two artifices are assimilated aspect-oriented programming (AOP). The experiment conducted a series of questions on which metrics and strategies should be used to detect God Classes in two different groups: one working with AspectJ (AOP), and the other with Java (OOP). The OOP version of the system was used as baseline to evaluate the AOP version.

As results, we had higher average Recall and Precision values for both groups in our experiment, an indicator of greater concordance within it. The set of nine metrics was reduced to 6 metrics (LOC, WOM, CMC, CBM, RFM and LCO) that 5 or more pairs chosen (5 of 14). If we consider more than 50% of metrics that the subjects have chosen, we have a set of 4 metrics (LOC, WOM, LCO). With this sets the attributes: size, coupling and cohesion can be quantified.

Few could be said about the thresholds values obtained. Their sparse behavior did not lead to any significant conclusion. With respect to the time measurement, the AOP group had average smaller values. Nonetheless, we admit the shorter number of subjects in this group could have biased the high values and we could not make strong conclusions about the relation of time, experience and efficiency.

The conclusion obtained here are restricted to the involved metrics, code smell (God Class), and target system. These limitations are typical of studies like ours. Although we acknowledge these limitations, we note that our study fills a gap in the literature by reporting original analyses on the benefits of using one defined set metrics for detecting God Classes in software systems implemented in AOP. Another limitation is that our experiment is based only in measures that AOPMetric can collect.

For future work, different code smells could be tested, and even compared with the results obtained for God Class. Also, experiments with larger groups would surely bring sharper or different results. Identify subjects' detection strategies more efficient to AO systems using the metrics set presented in this work. Finally, a more straightforward question about the thresholds could solve the problem of sparse numbers given as response.

Acknowledgements

This work was partially supported by Capes, CNPq (grant Universal 485907/2013-5) and FAPEMIG (grants APQ-02532-12 and PPM-00382-14).

References

Carneiro, G. F. et al. (2010) "Identifying Code Smells with Multiple Concern Views". In: Proceedings of the Brazilian Symposium on Software Engineering (SBES), pp. 128-137.

Ceccato, M. and Tonella, P. (2004) "Measuring the Effects of Software Aspectization". In: 1st workshop on Aspect Reverse Engineering.

Chidamber, S. R. and Kemerer, C. F. (1994) "A Metrics Suite for Object Oriented Design". In: Transactions on Software Engineering.

Fenton, N. E. and Pfleeger, S. L. (1998) “Software Metrics: A Rigorous and Practical Approach”. 2nd, Publishing Co. Boston, p. 656.

Ferrari, F., Burrows, R., Lemos, O., Garcia, A., Figueiredo, E., Cacho, N., Lopes, F., Temudo, N., Silva, L., Soares, S., Rashid, A., Masiero, P., Batista, T. and Maldonado, J. (2010) “An Exploratory Study of Fault-Proneness in Evolving Aspect-Oriented Programs”. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE), pp. 65-74.

Figueiredo, E., Sant’Anna C., Gracia, A., Bartolomei, T., Cazzola, W. and Marchetto, A.(2008) “On the Maintainability of Aspect-Oriented Software: A Concern-Oriented Measurement Framework”. In: Proceedings of 12th European Conference on Software Maintenance and Reengineering (CSMR), pp. 183-192.

Fowler, M., Beck, K., Brant, J., Opdyke W. and Roberts D. (1999) “Refactoring: Improving the Design of Existing Code”. Addison Wesley.

iBATIS - http://ibatis.apache.org/ - 07/04/2014.

Lanza, M. and Marinescu, R. (2006) “Object-Oriented Metrics in Practice”. Springer Verlag.

Marinescu, R. (2004) “Detection Strategies: Metrics-Based Rules for Detecting Design Flaws”. In: Proceedings of 20th IEEE International Conference on Software Manutenance (ICSM), pp. 350-359.

Nguyen, T. T., Nguyen, H. V., Nguyen, H. A. and Nguyen, T. N. (2011) “Aspect recommendation for evolving software”. In: Proceedings of the 33rd International Conference on Software Engineering (ICSE), pp. 361-370.

Padilha, J., Figueiredo, E., Santana, C., Garcia, A. (2013) “Detecting God Methods with Concern Metrics: An Exploratory Study”. In: Proceedings of 7th Latin American Workshop on Aspect-Oriented Software Development (LA-WASP), Brasilia.

Padilha, J., Pereira, J., Figueiredo, E., Almeida, J., Garcia, A., Sant’anna, C. (2014) “On the Effectiveness of Concern Metrics to Detect Code Smells: An Empirical Study”. In: Proceedings of the 26th International Conference on Advanced Information Systems Engineering (CAiSE), Thessaloniki, Greece.

Riel, A.J. (1996) “Object-Oriented Design Heuristics”. Addison-Wesley Professional, p. 400.

Robillard, M. and Murphy, G. (2007) “Representing Concerns in Source Code”. In: Transactions on Software Engineering and Methodology.

Subramanyan, R. and Krishnan, M.S. (2003) “Empirical Analysis of CK Metrics for Object-Oriented Design Complexity: Implications for Software Defects”. In: IEEE Transactions on Software Engineering, pp. 297–310.

Wroclaw University of Technology (2005), e-Informatyka and Tigris developers: aopmetrics project – http://aopmetrics.tigris.org/.

Zhang, M., Hall, T., Baddoo, N. (2011) “Code Bad Smells: a review of current knowledge”. In: Journal of Software Maintenance and Evolution: Research and Pratice, pp. 179-202.