Modularity Anomalies in Software Reuse: An Empirical Study with Software Product Lines

Names omitted due to blind review

[1]Department of Computer Science, Federal University of Minas Gerais (UFMG)
Belo Horizonte – MG – Brazil

Email adresses omitted due to blind review

Abstract. *Software reuse requires a satisfactory code modularity to be effectively applied in software development. Therefore, modularity anomalies may difficult reuse. By detecting and solving an anomaly before reuse of a software component, we may increase the component quality and decrease time and efforts spent on maintenance, for instance. In this context, we need methods to support the detection of these anomalies. Different detection strategies support the identification of modularity anomalies. Although the known variety of anomalies that affect software reuse, and their respective detection strategies, we lack an investigation of the effectiveness of available detection strategies. In this paper, we investigate modularity anomalies in software reuse focused on Software Product Line (SPL). For this purpose, we provide (i) an overview of studies that relate software reuse to modularity anomalies and propose detection strategies for these anomalies, (ii) the proposal of detection strategies for two well-known modularity anomalies, God Class and God Method, and (iii) an empirical study of detection strategies for the cited anomalies using the MobileMedia SPL. As a result, our detection strategies present minimum recall (29% and 17% for God Class and God Method, respectively) but the highest precision results (100% and 50%) when compared to strategies from literature.*

1. Introduction

Software reuse is a development technique in which existing source code is used in the development of new software systems [Krueger 1992]. In order to be effectively applied in software development, reuse requires a satisfactory source code modularity [Johnson and Foote 1988]. Since previously implemented software components from a an existing system may contain problems that should not be propagated to other systems, then modularity anomalies may difficult reuse. Therefore, by detecting and solving an anomaly before reusing a software component, we may increase the component quality and decrease time and efforts spent on maintenance of the new system, for instance [Moha et al. 2010, Fontana et al. 2012].

In this context, we need effective methods to support the detection of modularity anomalies, and consequently, to support software reuse [Romero et al. 2004]. Different detection strategies have been proposed in literature to support the identification of modularity anomalies in software systems [Fernandes et al. 2016]. Although there is a large variety of anomalies that affect software reuse, and some strategies have been proposed to support their detection, we lack an investigation of the effectiveness of available detection

strategies. Research to explore this gap may be helpful in the selection of appropriate detection strategies for an specific software development domain, such as Software Product Lines (SPL).

In this paper, we investigate modularity anomalies in software reuse focused on SPL given constraints when considering all approaches that support reuse. For this purpose, we provide (i) an overview of studies that relate software reuse to modularity anomalies and propose detection strategies for these anomalies, (ii) an empirical study of detection strategies for well-known modularity anomalies using a largely investigated SPL, MobileMedia, and (iii) the proposal of new detection strategies to support the identification of two modularity anomalies: God Class and God Method.

As a results, with respect to (i) we discuss that there is a lack of studies to investigate the relation between modularity anomalies and software reuse. Regarding (ii), we identify two main different detection strategies for each of the studied modularity anomalies (namely, God Class and God Method). Finally, from (iii) we observe that our detection strategies present minimally relevant recall (29% and 17% for God Class and God Method, respectively) and the highest precison results (100% and 50%) when compared to strategies from literature.

The remainder of this paper is organized as follows. Section 2 provides background to support the comprehension of our study. Section 3 describes the study settings, including goals, research questions, and the study steps. Section 4 discuss the results we obtained through the empirical study conducted with the MobileMedia SPL [Figueiredo et al. 2008]. Section 5 presents some threats to the validity of our study with the respective treatments. Section 6 discusses related work. Finally, Section 7 concludes this paper with the study contributions and presents suggestions for future work.

2. Background

This section provides sufficient background to support the comprehension of this paper. Section 2.1 discusses modularity anomalies in source code. Section 2.2 discusses software reuse. Section 2.3 presents the SPL approach in the context of software reuse.

2.1. Modularity Anomalies

Modularity anomalies, also known as bad smells, are symptoms of problems in the modularity design of software systems [Fowler 1999]. There are several types of anomalies that may affect the modularity of a system [Fernandes et al. 2016]. For instance, God Class is a well-known anomaly that may be defined as a class that contains excessive knowledge of the system and, in addition, many responsibilities in terms of processing [Fowler 1999]. Other modularity-related anomaly is called God Method. Given a class, a method may be considered a God Method when it is large in terms of lines of code (LOC) and with many responsibilities, for instance [Lanza and Marinescu 2007].

Two approaches are used to detect modularity anomalies: manual and automated detection [Moha et al. 2010]. An example of manual detection is the use of predefined detection strategies [Lanza and Marinescu 2007]. In general, these strategies are a composition of metric-based rules that defines when a specific software component (class, method, or package, for instance) is prone to present a modularity anomaly [Lanza and Marinescu 2007]. In turn, the automated detection is supported by

tools that essentially apply some type of detection strategy to detect modularity anomaly occurrences in an automated fashion [Fernandes et al. 2016].

2.2. Software Reuse

Software systems have become more complex and difficult to manage given clients' needs [Prado et al. 2014]. Reasons for this increasing complexity are the technological evolution, the software quality required by clients, and constrains regarding the time to delivery of software products. In this context, software reuse may be adopted to support the development of systems [Jacobson et al. 1997]. Software reuse is a development technique in which existing source code is used in the development of new software systems [Krueger 1992].

However, an inappropriate application of software reuse may lead to more development costs and efforts than software development without reuse [Pohl et al. 2005]. In order to be effectively applied in software development, an important issue is to assure satisfactory source code modularity [Johnson and Foote 1988]. Since previously implemented software components from a an existing system may contain problems that should not be propagated to other systems, then modularity anomalies may difficult reuse. To support reuse, many approaches have been proposed in literature, including Software Product Lines (SPL) [Pohl et al. 2005].

2.3. Software Product Lines

A SPL is a set of software systems that share features that are designed to a specific software domain [Pohl et al. 2005]. The feature model is an approach to model these features of a SPL [Kang et al. 1990]. There are four types of features in product line: (i) mandatory, features that have to be including in every product from the SPL, (ii) optional, features that may be included in the product according to the client's needs, (iii) OR, features that may be co-included in a product, and (iv) XOR, features that are mutually exclusive when included in the product [Weiss 1999]. Each product from a SPL is composed by general features that define the SPL (mandatory features) and specific features that differ a product from another (optional, OR, or XOR features) [Pohl et al. 2005].

Artifacts of a SPL may contain modularity anomalies as in other types of software systems. However, there are few studies to investigate anomalies in this specific context [Apel et al. 2013]. A recent systematic literature review (SLR) [Vale et al. 2014] discusses that new anomalies may be proposed, considering the inherent complexity of SPL design. Furthermore, anomalies proposed in other software settings may be applied in the context of SLP. Finally, new detection strategies may be proposed to improve the effectiveness of detecting anomalies.

3. Study Settings

This section describes the study design. Section 3.1 provides goal and research questions we defined to guide this study. Section 3.2 describe an *ad hod* literature review of modularity anomalies and software reuse. Section 3.3 presents the design of an empirical study to compare different detection strategies for detection of modularity anomalies.

3.1. Goal and Research Questions

This study aims to investigate the impact of modularity anomalies on software reuse with focus on SPL. We are specifically concerned about the detection strategies that have been proposed to identify modularity anomalies that may hinder reuse in the SPL context. For this purpose, we designed a study to assess (i) the current state of research on the relation between modularity anomalies and software reuse in SPL settings, (ii) the need of proposing new detection strategies, and (iii) a comparative study of detection strategies from literature and, eventually, proposed by the authors of this work.

Basically, we are interested in the effectiveness of available detection strategies to support the identification of modularity anomalies in SPL. Since, these anomalies may affect the quality of reuse activities, then we require effective and useful strategies. The scope of our study, based on the Goal-Question-Metric (GQM) method [Wohlin et al. 2012], is: *Analyze detection strategies for modularity anomalies, from the purpose of identifying appropriate strategies to support software development, from the point of view the authors based on statistical analysis, in the context of Software Product Lines (SPL).*

To guide our study, we designed the following research questions.

RQ1. Are there studies to related explicitly modularity anomalies to software reuse? – Through RQ1, we expect to comprehend the current state of research on the relation between modularity anomalies that may affect software reuse.

RQ2. Are the available detection strategies for modularity anomalies effective? – With RQ2, we aim to investigate whether the detection strategies for modularity anomalies proposed in literature are sufficiently able to support software reuse.

Figure 1 presents the steps that compose our study. We designed the nine study steps presented in this figure based on our main study goal and research questions. To ease the comprehension of our study design, we grouped Step 1 to 9 in study phases. We describe each step and respective rationale, as well as the groups of steps, as follows.

Steps 1 and 2 compose a group we called Literature Review (more details in Section 3.2). In **Step 1**, we conduct an *ad hoc* literature review of studies that relate modularity anomalies to software reuse, not necessarily in the context of SPL. We aim to identify common anomalies in the context of reuse and detection strategies proposed for them, for instance. In **Step 2**, we focus our *ad hoc* review on modularity anomalies that may occur in product lines.

The remaining steps, Steps 3 to 9, compose other group called Comparative Study. In **Step 3**, we select a SPL to assess the effectiveness of modularity anomalies detection using different detection strategies, through an empirical study. **Step 4** consists of the selection of modularity anomalies for investigation trough an em, based on anomalies that may be detected in the system chosen at Step 3. **Step 5** defines the selection of detection strategies from literature to be compared in this study. **Step 6** is dedicated to the proposal of new detection strategies for the modularity anomalies chosen in Step 3 to be compared with the strategies from literature provided by Step 5.

In **Step 7**, we select a method for threshold derivation to support the manual detection of modularity anomalies using detection strategies. Thresholds are numerical values

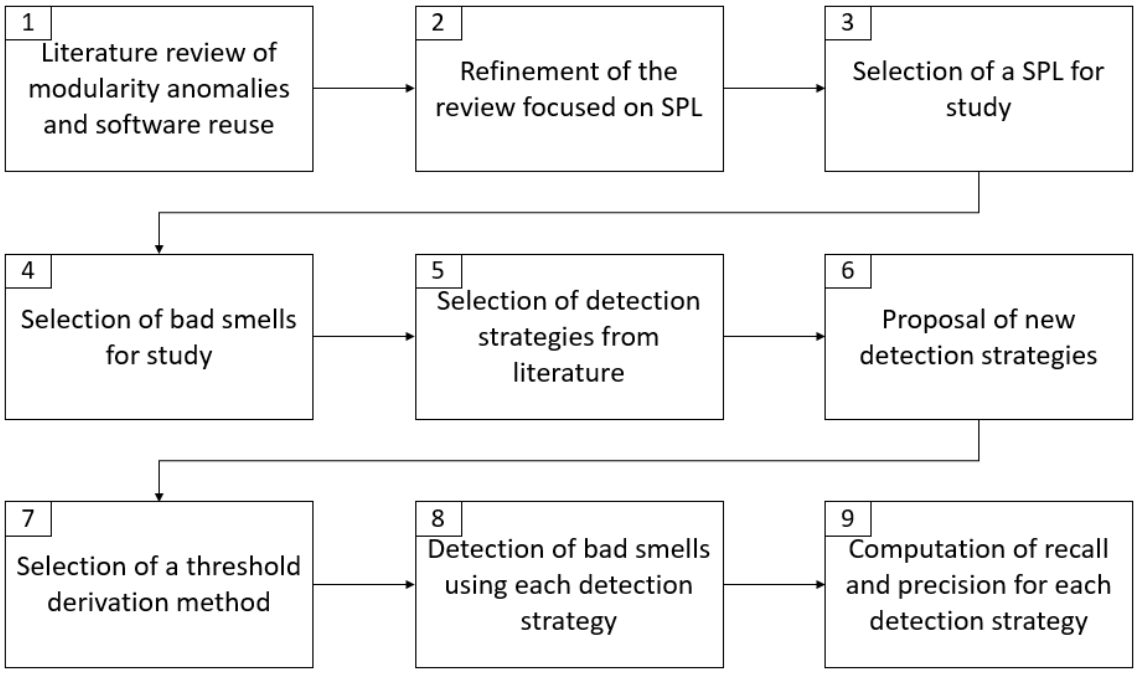

Figure 1. Steps of the study

that defines ranges for a measure. These ranges, also called labels, may be used to classify a measure in a given context. As example, a threshold 100 may be proposed to define that a value for LOC $>$ 100 is a high value. **Step 8** is the detection of selected bad smells from Step 4, using the strategies selected and proposed in Steps 5 to 6, and with support of the method selected in Step 7, in the SPL selected in Step 3. Finally, **Step 9** is the computation of recall and precision for comparative analysis of detection strategies.

3.2. A Literature Review

With respect to **Steps 1-2**, we conducted a literature review using the Google Scholar[1] platform. In a first moment, we decided not to use specific electronic data sources (such as ACM Digital Library[2] and IEEE Xplore[3]) because we expected the *ad hoc* review would lead us to a systematic literature review (SLR) [Kitchenham and Charters 2007] in the future. Therefore, our goal in this preliminary review is to provide an overall of studies that relate modularity anomalies and software reuse.

First, we defined the following search string to run in Scholar: *(“software reuse” OR “software reutilization”) AND (“modularity smell*” OR “modularity anomal*” OR “architecture smell*” OR “architecture anomal*” OR “design smell*” OR “design anomal*”)*. As a pilot-search, we pairwise combined terms from the first and second clauses of string to run in the search platform. However, we obtained few results and, then, we decided to discard this string and conduct a more specific *ad hoc* search.

Second, we decided to focus on specific approaches to support software reuse. The search string we designed to run in the electronic source is: *(“modularity anomalies” OR*

[1]https://scholar.google.com.br/
[2]http://dl.acm.org/
[3]http://ieeexplore.ieee.org/Xplore/home.jsp

"bad smell") AND ("component-based" OR "service-oriented" OR "software product line"). In turn, this string provided some interesting results discusses in Section 4.1. We observed, then, an opportunity to focus on the SPL context because of the number of retrieved studies by through this search.

3.3. A Comparative Study

Regarding **Steps 3-9**, we conducted an empirical study to compare different detection strategies for modularity anomalies. Our goals is to assess the effectiveness of each strategy when compared to others. In Step **Step 3**, we chose the MobileMedia SPL for analysis, extracted from a benchmark of SPLs [Vale and Figueiredo 2015]. We took this decision based on the availability of source code, pre-calculated software metrics, and the existence of a reference list for two types of modularity anomalies in the system: God Class and God Method. MobileMedia is implemented using feature-oriented programming (FOP) in AHEAD programming language. We used the version 7 of the system with 2691 LOC. Table 1 presents the list of software metrics that are pre-calculated for MobileMedia.

Class-level Metrics
Coupling between Objects (CBO): the number of dependencies among a given class and other classes [Chidamber and Kemerer 1994].
Lines of Code (LOC): it counts the number of lines of code from a given class [Lorenz and Kidd 1994].
Number of Constant Refinements (NCR): it counts the number of refinements for a given constant [Abilio et al. 2015].
Number of Attributes (NOA) a.k.a. Number of Fields (NOF): the number of attributes of the class [Lorenz and Kidd 1994].
Number of Methods (NOM): it counts the number of methods of the class [Lorenz and Kidd 1994].
Weighted Methods per Class (WMC): it computes a weight for the class based on the sum of weights for each method. This weight can be defined using LOC, for instance [Chidamber and Kemerer 1994].
Method-level Metrics
McCabe's Cyclomatic Complexity (Cyclo): it counts the number of possible deviations in the method execution, based on the number of branches such as `if`, `while`, and `for` [McCabe 1976].
Method Lines of Code (MLOC): it counts the number of code lines of the method [Lorenz and Kidd 1994].
Number of Operations Overrides (NOOr): it counts the number of overrides of a method [Miller et al. 1999].
Number of Method Refinements (NMR): it counts the number of refinements of the methods [Abilio et al. 2016].
Number of Parameters (NP): it counts the number of parameters of the method [Lorenz and Kidd 1994].

Table 1. Software metrics for MobileMedia

The benchmark that provides MobileMedia also provides reference lists for modularity anomalies in the system. These lists were composed by experienced developers

from the MobileMedia development team. In this study, we investigate the occurrence of two modularity anomalies: God Class and God Method. Therefore, the reference lists provide us data to compute recall and precision of the detection strategies we compare. Table 2 presents the reference lists for the two investigated anomalies.

God Classes: 7	**God Methods: 6**
AlbumController	AlbumController.handleCommand()
MediaController	MediaController.handleCommand()
PhotoViewController	MediaListController.showMediaList()
MediaAccessor	PhotoViewController.handleCommand()
CaptureVideoScreen	PlayVideoController.handleCommand()
MediaUtil	MainUIMidlet.startApp()
SmsMessaging	-

Table 2. Reference lists for God Class and God Method in MobileMedia

In **Step 4**, we decided to evaluate God Class and God Method because we have reference lists of these modularity anomalies for MobileMedia. In **Step 5**, we selected detection strategies for each anomaly using the following search strings: *("detection strategy" AND ("god class" OR "large class" OR "brain class"))* for God Class and *("detection strategy" AND ("god method" OR "long method" OR "brain method"))* for God Method. We based our string on a systematic view of tools for anomaly detection [Fernandes et al. 2016]. For each anomaly, we obtained two detection strategies for God Class, and other two for God Method, that we are able to use with the metrics provided for MobileMedia. Table 3 present these strategies adapted according to the available metrics (we only discarded terms with metrics that our data source does not provide).

God Class
GC1: Detection Strategy 1 [Vale and Figueiredo 2015]
$[(LOC > High)$ AND $(WMC > High)$ AND $(CBO > Low)]$ OR $(NCR > High)$
GC2: Detection Strategy 2 [Filó et al. 2015]
$(WMC > 34)$ AND $(NOM > 14)$ AND $(NOA > 8)$
God Method
GM1: Detection Strategy 1 [Fard and Mesbah 2013]
$(MLOC > 50)$
GM2: Detection Strategy 2 [Lanza and Marinescu 2007]
$(MLOC > High)$ AND $(Cyclo \geq High)$

Table 3. Detection strategies from literature

In **Step 6**, considering the few detection strategies we were able to collect, we proposed a new strategy for each anomaly. Table 4 presents the new detection strategies. The rationale for each strategy is describe as follows.

We designed C3 based on: (i) LOC, because a high number of code lines may indicate excessive processing and responsibilities of the class, (ii) NOA and NOM, because a high number of attributes (NOA) and (NOM) points excessive knowledge of the class (attributes) and responsibilities (methods), and (iii) WMC, because a high weight of the class

indicates that the class is doing more than it should do. In turn, we designed GM3 based on: (i) MLOC, because a high number of code lines is a symptom of complex method, (ii) NP, because a large list of parameters may point that the method requires excessive knowledge of the current or external classes, and (iii) Cyclo, because high complexity is an indication of many responsibilities of the method.

God Class
GC3: Detection Strategy 3
$(LOC > High)$ AND $(NOA > High)$ AND $(NOM > High)$ AND $(WMC > High)$
God Method
GM3: Detection Strategy 3
$(MLOC > High)$ AND $(NP > High)$ AND $(Cyclo > High)$

Table 4. New detection strategies we propose

In **Step 7**, we selected a method for threshold derivation. We chose Vale's method [Vale and Figueiredo 2015] because it supports five different labels and uses is benchmark-based. The labels, calculated in terms of percentiles considering entities from the an entire benchmark, are: Very Low ($> 0\%$ of the values), Low ($> 3\%$), Moderate ($> 15\%$), High ($> 90\%$), and Very High ($> 95\%$). Therefore, this method provides us the sufficient flexibility to define detection strategies, and also the computation of thresholds based on a set of software systems from the same domain. To support the computation of labels, we used the R[4] statistical environment. Moreover, to apply Vale's method and derive thresholds for the metrics described in Table 1, we used the entire SPL-benchmark from which we obtained MobileMedia. This benchmark has 35 product lines from 17 LOC to 42 KLOC. Table 5 presents the derived thresholds.

Class-level Metrics					**Method-level Metrics**				
Metrics	**Thresholds**				**Metrics**	**Thresholds**			
	3%	**15%**	**90%**	**95%**		**3%**	**15%**	**90%**	**95%**
LOC	2	4	77	133	Cyclo	0	1	3	5
NCR	0	0	1	2	MLOC	0	2	13	22
NOA	0	0	4	8	NOOr	0	0	0	0
NOM	0	1	10	16	NMR	0	0	1	1
WMC	0	1	17	31	NP	0	0	2	3

Table 5. Derived thresholds for metrics from Table 1

In **Step 8**, we combined the detection strategies from Tables 3 and 4 to the thresholds from Table 5 that we derived to detect God Class and God Method. This procedure was conducted with support of spreadsheets with the computed software metrics. The results are presented and discussed in Section 4. Finally, in **Step 9** we computed recall and precision for each detection strategy. The results of this analysis are also presented and discussed in Section 4.

[4]https://www.r-project.org/

4. Results and Discussion

This section presents and discusses the results of our study. Section 4.1 presents the results obtained trough the literature review of modularity anomalies and software reuse (described in Section 3.2). Section 4.2 discusses the results we obtained with the empirical comparative study of detection strategies (as presented in Section 3.3).

4.1. Results from Literature Review

In this section, we discuss the results from our *ad hoc* literature review of modularity anomalies and software reuse. Therefore, we aim to answer RQ1.

RQ1. Are there studies to related explicitly modularity anomalies to software reuse?

Through our first search in Scholar, we did not find studies that explicitly relate modularity anomalies to software reuse. However, in our first attempt to search for studies in the electronic source, using terms for specific software reuse approach, we were able to extract some interesting studies. For instance, we found studies that cover Component-Based Software Engineering (CBSE) [von Detten and Becker 2011, Platenius et al. 2012] and Service-Oriented Architecture (SOA) [Palma 2012, Moha et al. 2012].

We were able to found a more significant number of studies in the context of SPL [Niu and Easterbrook 2009, Andrade et al. 2014, Vale et al. 2014]. In general, these studies investigate the occurrence of different types of modularity anomalies that may affect not only SPLs, but other software systems. They also discuss the use of detection strategies that cover traditional and SPL-specific software metrics. However, we did not find a study that compares different detection strategies. That is one of the reasons we propose a comparative study of strategies for modularity anomalies.

4.2. Results from Comparative Study

In this section, we discuss the results from our comparative study of detection strategies for modularity anomalies. Then, we discuss RQ2.

RQ2. Are the available detection strategies for modularity anomalies effective?

Table 6 presents recall and precision that we computed in the comparative study, considering both God Class and God Method. Note that: (i) TP is the number of true positives (correct answers in the identification of real anomalies), (ii) FP is the number of false positives (incorrect answers for identification of real anomalies), (iii) TN is the number of true negatives (correct answers for non-identification of anomalies), and (iv) FN is the number of false negatives (incorrect answers for non-identification of anomalies) [Fawcett 2006].

With respect to God Class detection strategies, we observe that the best precision is provided by our strategy GC3 (100%), with a significant different of 67% when compared to the second best precision by GC1. In turn, regarding recall, GC1 provides the best results (71%), although our strategy GC3 has a significant recall of 29%, more than the results from GC2 (0%).

With respect to God Method, we observe that the best precision is provided by our GM3 (50%). This results is 44% better than the second best precision (GM2). Finally, GM2 has the best recall (33%), but with a minimum different of 16% when compared to

God Class				God Method			
Value	Strategy			Value	Strategy		
	GC1	GC2	GC3		GM1	GM2	GM3
TP	5	0	2	TP	0	2	1
FP	10	0	0	FP	1	31	1
TN	126	7	136	TN	366	336	366
FN	2	136	50	FN	6	4	5
Recall	71%	0%	29%	Recall	0%	33%	17%
Precision	33%	0%	100%	Precision	0%	6%	50%

Table 6. Recall and precision for each detection strategy

our strategy GM3. Therefore, we conclude that our proposed strategies are an improvement in terms of precision for both God Class and God Method, but with respect to recall, our strategies present an average but significant performance.

5. Threats to Validity

We designed and conducted carefully an *ad hoc* literature review, as presented in Section 3.2, and an experiment as described in Section 3.3. For instance, we delimited our experiment scope prior to the execution of the experiment, defined our research questions, and how to assess them, after study and based on previous studies. However, there are some threats to the validity of our findings. We discuss, as follows, each of the four threats, with respective treatments, as presented by Wolin et al. [Wohlin et al. 2012]: internal, conclusion, construct, and external validity.

Construct Validity. We designed our experiment to be replicated with different detection strategies, methods for threshold derivation, and so on. For this purpose, we conducted various searches on electronic bases to find similar studies to base our experiment design. In addition, we conducted many searches to retrieve studies relate modularity anomalies to software reuse, and also to identify proposed detection strategies. These treatments aim to minimize threats related to the coverage of our study. Therefore, we expect that our results are enough meaningful and consistent in the context of reuse with focus on SPL.

Internal Validity. We conducted a careful data collection to minimize problem with respect to missing data, incorrect selection of metrics, and inappropriate use of threshold derivation methods, for instance. Therefore, we expect that our data collection is reliable to provide our the correct data analysis. Since we provide the data used for study and the experiment design (protocols, steps, etc.), we allow other researches to assure the correctness in our data management process.

Conclusion Validity. We carefully performed the data analysis to draw conclusions regarding effectiveness of evaluated detection strategies. This procedure minimize problems related to data interpretation. We based the choice of mathematical computation (recall and precision) on previous studies to provide the appropriateness of the techniques we adopted in the data analysis presented in Section 4. Moreover, we used a reliable and largely used supporting tool for data analysis.

External Validity. Some factor may prevent the generalization of our research findings. For instance, we conducted an *ad hoc* literature review of studies that relate anomalies

to reuse, and also regarding detection strategies for modularity anomalies in SPL. This procedure may not cover all existing studies in the target topic. However, we defined search string composed by diverse popular and related terms to minimize this problem. With respect to the experiment execution, we proposed detection strategies based on the subjective authors' knowledge of metrics and anomalies. To minimize this problem, we carefully analyzed the available metrics in the context of our study, and we based our definitions of each covered anomaly in the literature.

6. Related Work

To the best of our knowledge, we were not able to find studies that are similar to ours. One of the closely related studies is proposed by Fernandes et al. (2016). The authors conduct a systematic literature review of bad smell detection tools. They were able to find 84 different supporting tools, 29 of them available online for download. In addition, the authors conduct a comparative study of four tools to compute recall, precision, and agreement of the tools. They observe that most of the proposed tools are metric-based, therefore they apply detection strategies based on software metrics. Moreover, in general, the authors observe that tools are redundant in terms of detected anomalies and, then, new detection strategies may be explored in order to improve their effectiveness.

In this study, we aim to contribute by the filling of the gap observed by Fernandes et al. (2016), although we focus on software reuse and SPL specifically. For this purpose, we investigate the current state of proposed detection strategies to observe whether there is a sufficient variety of strategies to support the implementation of new detection tools, for instance. Moreover, we compare detection strategies to identify their effectiveness. At last, we propose new detection strategies to improve the state of the art on detection of modularity anomalies.

7. Conclusion and Future Work

This paper investigates modularity anomalies that affect software reuse. We focus on the Software Product Line (SPL) reuse approach. For this purpose, we provide an overview of studies that relate software reuse to modularity anomalies, specially in the SPL context. Furthermore, we propose new detection strategies for two well-known modularity anomalies: God Class and God Method. Moreover, we compare our new strategies to others proposed in literature, to assess their effectiveness it terms of recall and precision. We use the MobileMedia SPL, with reference lists of God Class and God Method anomalies, for assessment of effectiveness of the compared strategies.

As a result, we observe that there is a lack of studies to investigate modularity anomalies to software reuse. However, considering the context of SPL, some studies have been investigating anomalies and detection strategies that affect reuse. Moreover, we observe that our proposed detection strategies provide minimum recall (29% and 17% for God Class and God Method, respectively) and the highest precision results (100% and 50%) when compared to strategies from literature. Data from this work are available online in the website of this study[5], for assessment and and replication.

As future work, we suggest the investigation of other modularity anomalies such as Lazy Class, Shotgun Surgery, and Divergent Change that may affect software reuse. We

[5]Website omitted due to blind review

also suggest the proposal of new detection strategies for these anomalies for evaluation and comparison, given the lack of proposed strategies for modularity anomalies in general. Finally, we suggest the replication of our study in other software development contexts that apply reuse of source code using different approaches than SPL.

References

Abilio, R., Padilha, J., Figueiredo, E., and Costa, H. (2015). Detecting Code Smells in Software Product Lines: An Exploratory Study. In *Proceedings of the 12th International Conference on Information Technology: New Generations (ITNG)*, pages 433–438.

Abilio, R., Vale, G., Figueiredo, E., and Costa, H. (2016). Metrics for Feature-Oriented Programming. In *Proceedings of the 7th International Workshop on Emerging Trends in Software Metrics (WETSoM)*, pages 36–42.

Andrade, H., Almeida, E., and Crnkovic, I. (2014). Architectural Bad Smells in Software Product Lines: An Exploratory Study. In *Proceedings of the 11th Working Conference on Software Architecture (WICSA)*, page 12.

Apel, S., Batory, D., Kästner, C., and Saake, G. (2013). *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer Science & Business Media.

Chidamber, S. and Kemerer, C. (1994). A Metrics Suite for Object Oriented Design. *Transactions on Software Engineering (TSE)*, 20(6):476–493.

Fard, A. and Mesbah, A. (2013). JSNOSE: Detecting JavaScript code smells. In *Proceedings of the 13th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 116–125.

Fawcett, T. (2006). An Introduction to ROC Analysis. *Pattern Recognition Letters*, 27(8):861–874.

Fernandes, E., Oliveira, J., Vale, G., Paiva, T., and Figueiredo, E. (2016). A Review-based Comparative Study of Bad Smell Detection Tools. *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering (EASE)*.

Figueiredo, E., Cacho, N., Sant'Anna, C., Monteiro, M., Kulesza, U., Garcia, A., Soares, S., Ferrari, F., Khan, S., Castor Filho, F., and Dantas, F. (2008). Evolving Software Product Lines with Aspects. *Proceedings of the 30th International Conference on Software Engineering (ICSE)*, pages 261–270.

Filó, T., Bigonha, M., and Ferreira, K. (2015). A Catalogue of Thresholds for Object-Oriented Software Metrics. *Proceedings of the 1st International Conference on Advances and Trends in Software Engineering (SOFTENG)*.

Fontana, F., Braione, P., and Zanoni, M. (2012). Automatic Detection of Bad Smells in Code: An Experimental Assessment. *Journal of Object Technology (JOT)*, 11(2):5–1.

Fowler, M. (1999). *Refactoring: Improving the Design of Existing Programs*. Addison-Wesley Publishing.

Jacobson, I., Griss, M., and Jonsson, P. (1997). *Software Reuse: Architecture, Process and Organization for Business Success*. Addison-Wesley Publishing.

Johnson, R. and Foote, B. (1988). Designing Reusable Classes. *Journal of Object-Oriented Programming (JOOP)*, 1(2):22–35.

Kang, K., Cohen, S., Hess, J., Novak, W., and Peterson, A. (1990). Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical report, DTIC Document.

Kitchenham, B. and Charters, S. (2007). Guidelines for Performing Systematic Literature Reviews in Software Engineering. In *Technical report, Ver. 2.3 EBSE Technical Report. EBSE*.

Krueger, C. (1992). Software Reuse. *Computing Surveys (CSUR)*, 24(2):131–183.

Lanza, M. and Marinescu, R. (2007). *Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*. Springer Science & Business Media.

Lorenz, M. and Kidd, J. (1994). *Object-Oriented Software Metrics: A Practical Guide*. Prentice-Hall.

McCabe, T. (1976). A Complexity Measure. *Transactions on Software Engineering (TSE)*, (4):308–320.

Miller, B., Hsia, P., and Kung, C. (1999). Object-Oriented Architecture Measures. In *Proceedings of the 32nd Annual Hawaii International Conference on Systems Sciences (HICSS)*.

Moha, N., Gueheneuc, Y.-G., Duchien, L., and Le Meur, A.-F. (2010). DECOR: A Method for the Specification and Detection of Code and Design Smells. *Transactions on Software Engineering (TSE)*, 36(1):20–36.

Moha, N., Palma, F., Nayrolles, M., Conseil, B. J., Guéhéneuc, Y.-G., Baudry, B., and Jézéquel, J.-M. (2012). Specification and Detection of SOA Antipatterns. In *Proceedings of the 10th International Conference on Service Oriented Computing (ICSOC)*, pages 1–16.

Niu, N. and Easterbrook, S. (2009). Concept Analysis for Product Line Requirements. In *Proceedings of the 8th ACM International Conference on Aspect-Oriented Software Development (AOSD)*, pages 137–148.

Palma, F. (2012). Detection of SOA Antipatterns. In *Proceedings of the 10th International Conference on Service-Oriented Computing (ICSOC) Workshops*, pages 412–418.

Platenius, M., Von Detten, M., and Becker, S. (2012). Archimetrix: Improved Software Architecture Recovery in the Presence of Design Deficiencies. In *Proceedings of the 16th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 255–264.

Pohl, K., Böckle, G., and van Der Linden, F. (2005). *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer Science & Business Media.

Prado, E., Ornellas, R., and Araújo, L. (2014). *Fundamentos de Sistemas de Informação*. Elsevier Brasil.

Romero, S., de Almeida, E., Alexandre, A., Lucredio, D., and García, V. (2004). RiSE Project: Towards Robust Framework for Software Reuse. In *Proceedings of the 5th International Conference on Information Reuse and Integration (IRI)*, pages 48–53.

Vale, G. and Figueiredo, E. (2015). A Method to Derive Metric Thresholds for Software Product Lines. In *Proceedings of the 29th Brazilian Symposium on Software Engineering (SBES)*, pages 110–119.

Vale, G., Figueiredo, E., Abilio, R., and Costa, H. (2014). Bad Smells in Software Product Lines: A Systematic Review. In *Proceedings of the 8th Brazilian Symposium on Software Components, Architectures and Reuse (SBCARS)*.

von Detten, M. and Becker, S. (2011). Combining Clustering and Pattern Detection for the Reengineering of Component-based Software Systems. In *Proceedings of the International Symposium on Architecting Critical Systems (ISARCS) and International Conference on Quality of software Architectures (QoSA)*, pages 23–32.

Weiss, D. (1999). Software Product-Line Engineering: A Family-based Software Development Process.

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M., Regnell, B., and Wesslén, A. (2012). *Experimentation in Software Engineering*. Springer Science & Business Media.