No Code Anomaly is an Island

Anomaly Agglomeration as Sign of Product Line Instabilities

Eduardo Fernandes[1,2], Gustavo Vale[3], Leonardo Sousa[2], Eduardo Figueiredo[1], Alessandro Garcia[2], Jaejoon Lee[4]

[1]Department of Computer Science, Federal University of Minas Gerais, Brazil
[2]Informatics Department, Pontifical Catholic University of Rio de Janeiro, Brazil
[3]Department of Informatics and Mathematics, University of Passau, Germany
[4]School of Computing and Communications, Lancaster University, United Kingdom

{eduardofernandes, figueiredo}@dcc.ufmg.br,
vale@fim.uni-passau.de, {lsousa, afgarcia}@inf.puc-rio.br,
j.lee3@lancaster.ac.uk

Abstract. A software product line (SPL) is a set of systems that share common and varying features. To provide large-scale reuse, the components of a SPL should be easy to maintain. Therefore, developers have to identify anomalous code structures – i.e., code anomalies – that are detrimental to the SPL maintainability. Otherwise, SPL changes can eventually propagate to seemly-unrelated features and affect various SPL products. Previous work often assume that each code anomaly alone suffices to characterize SPL maintenance problems, though each single anomaly may represent only a partial, insignificant, or even inexistent view of the problem. As a result, previous studies have difficulties in characterizing anomalous structures that indicate SPL maintenance problems. In this paper, we study the surrounding context of each anomaly and observe that certain anomalies may be interconnected, thereby forming so-called anomaly agglomerations. We characterize three types of agglomerations in SPL: feature, feature hierarchy, and component agglomeration. Two or more anomalies form an agglomeration when they affect the same SPL structural element, i.e. a feature, a feature hierarchy, or a component. We then investigate to what extent non-agglomerated and agglomerated anomalies represent sources of a specific SPL maintenance problem: instability. We analyze various releases of four feature-oriented SPLs. Our findings suggest that a specific type of agglomeration indicates up to 89% of sources of instability, unlike non-agglomerated anomalies.

Keywords: Code Anomaly Agglomeration; Software Product Line; Instability.

1 Introduction

A software product line (SPL) is a set of systems that share common and varying features [22]. Each feature is an increment in functionality of the product-line systems [2]. The combination of features generates different products [4]. Thus, the main goal of

SPL is to provide large-scale reuse with a decrease in the maintenance effort [22]. The implementation of a feature can be distributed into one or more source files, called components. To support large-scale reuse, the components and features of a SPL should be easy to maintain. Therefore, developers should identify anomalous code structures that are detrimental to the SPL maintainability. Otherwise, changes can eventually be propagated to seemly-unrelated features and affect various SPL products.

Code anomalies are anomalous code structures that represent symptoms of problems in a system [12]. They can harm the maintainability of systems in several levels by affecting classes and methods, for instance [12, 16]. Code anomalies affect any system, including SPL [7]. Previous work states that SPL-specific anomalies can be easier to introduce, harder to fix, and more critical than others, due to the inherent SPL complexity [18]. An example of code anomaly is *Long Refinement Chain* [7], related to the feature hierarchy (see Section 3.1). This anomalous code structure may hinder developers in understanding and performing proper changes. Eventually, these changes might affect several SPL products in the whole product-line. Thus, understanding the negative impact of anomalies in the SPL maintainability is even more important than in stand-alone systems, as their side effects may affect multiple products. Still, there is little understanding about the impact of such anomalies on the SPL maintainability.

Some studies assume that each anomaly alone suffices to characterize SPL maintenance problems [7, 23]. However, each single anomaly may represent only a partial view of the problem. This limited view is because, in several occasions, a maintenance problem is scattered into different parts of the code [20]. For instance, *Long Method* is a method with too many responsibilities that, if isolated, represents a punctual, simple problem [12]. In turn, *Long Refinement Chain* is a method with too many refinements in different features [7] that, in isolation, does not indicate a critical problem depending on the refined method. However, if we observe both anomalies in the same method, we may assume an increasing potential of the anomalies in hindering the SPL maintainability, since an anomalous method is excessively refined and causes a wider problem. As a result, previous studies have limitations to characterize anomalous structures that indicate SPL maintenance problems. On the other hand, previous work has observed that certain anomalies may be interconnected, forming so-called anomaly agglomerations. They investigate to what extent these anomaly agglomerations support the characterization of maintenance problems of a single system [21]. The authors define a *code anomaly agglomeration* as a group of two or more anomalous code elements directly or indirectly related through the program structure of a system [21]. However, they do not characterize and study specific types of anomaly agglomerations in SPLs.

In this paper, we first characterize common types of anomaly agglomerations in SPLs. Then, we investigate how often non-agglomerated versus agglomerated anomalies occur in SPLs and if they indicate sources of instability, a specific SPL maintenance problem. In fact, our findings suggest that “no code anomaly is an island”, i.e., code anomalies often interconnect to other anomalies in critical elements of a SPL, such as a feature, a feature hierarchy, or a component. We also confirm that non-agglomerated anomalies do not support the identification of structures that often harm the SPL maintainability. We then investigate to what extent certain types of agglomerations represent sources of instabilities. We propose three types of agglomeration based on the key SPL

decomposition characteristics, i.e. features, refinement chains, and components. Two or more anomalies form an agglomeration when they affect together a feature, a feature hierarchy, or a component. We then analyze the relationship between agglomerations and instabilities. Our analysis relies on different releases of four feature-oriented SPLs.

For each proposed type of agglomeration, we compute the strength of the relationship between agglomerations and instability in SPLs. We also compute the accuracy of agglomerations in indicating sources of instability. Our data suggest that feature hierarchy agglomerations and instability are strongly related and, therefore, this type of agglomeration is a good indicator of instabilities. The high precision of 89% suggests feature hierarchy can support developers in anticipating SPL maintenance problems. These findings are quite interesting because SPLs implemented with feature-oriented programming (FOP) are rooted strongly on the notion of feature hierarchies. It indicates that developers of FOP-based SPLs should design carefully the feature hierarchies, since they might generate hierarchical structures that hamper the SPL maintainability.

The remainder of this paper is organized as follows. Section 2 provides background information. Section 3 proposes and characterizes three types of anomaly agglomerations in SPL. Section 4 describes the study settings. Section 5 presents the study results. Section 6 discusses related work. Section 7 presents threats to the study validity with respective treatments. Section 8 concludes the paper and suggests future work.

2 Background

This section provides background information to support the paper comprehension. Section 2.1 presents feature-oriented SPLs. Section 2.2 discusses instability in SPL.

2.1 Feature-Oriented Software Product Lines

In this paper, we analyze SPLs developed with feature-oriented programming (FOP) [4]. FOP is a compositional technique in which physically separated code units are composed to generate different product-line systems. We analyze SPLs implemented using the AHEAD [4] specific-language technique and the FeatureHouse [3] multi-language technique. We chose these technologies because they compose features in separated code units and are well-known in FOP community. Both technologies implement SPLs through successive refinements, in which complex systems are developed from an original set of mandatory features by incrementally adding optional features, called SPL variability [4]. A feature is composed by one or more component (*constants* or *refinements*) that represents a code unit. A constant is the basic implementation of functionality and a refinement adds or changes the functionality of a constant [4].

To illustrate the FOP main concepts, Fig. 1 presents the partial design view of MobileMedia [10], a SPL for management of media resources. In Fig. 1, there are 3 features and 13 components. Lines connecting components indicate a *refinement chain* with a constant in the topmost feature and refinements in the features below. When generating a SPL product, only the bottom-most refinement of the chain is instantiated, because it implements all the capabilities assigned to the respective chain [4]. In this

study, we also refer to refinement chains as feature hierarchies, due to the order of components stablished by a refinement. As an example, the feature *SMSTransfer* has four constants and one refinement (`MediaController`). This refinement is part of a feature hierarchy that cuts across the three features presented in Fig. 1.

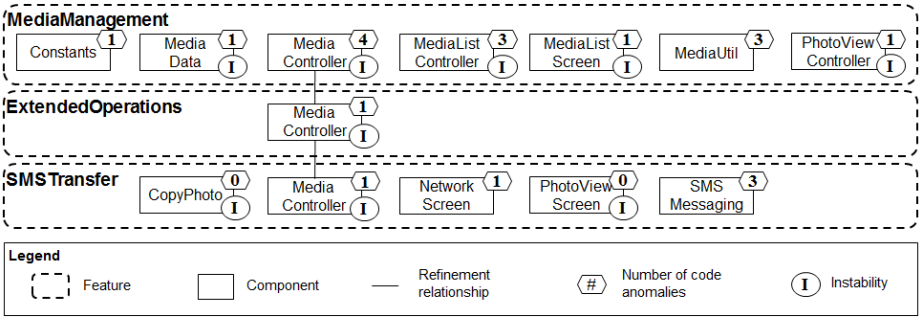

Fig. 1. Partial Design View of the MobileMedia SPL

2.2 Sources of Instability in SPL

Instability is the probability of a system to change, due to changes performed in different parts of the source code [1]. A previous work states that instability relates mostly to the maintenance of a system and, therefore, instability harms the SPL maintainability [25]. Moreover, a previous work has found evidence that code anomalies can induce to instability in systems [14]. Stability is even more important for SPLs than single systems, since changes in one feature can propagate to other features and affect seemly-unrelated configurations of a SPL [10]. In this study, we assess to what extent anomaly agglomerations support the identification of sources of instability in SPLs. We are concerned about the relationship between agglomerated anomalies in indicating parts of the code that change frequently and represent an instability.

In this study, we consider a component as instable if it has changed in at least two SPL releases. We made this decision because there are few available releases per analyzed SPL, seven at most. As a SPL evolves, components may change and be improved. However, after a manual inspection of the target SPLs, we observed that most of the changes reflected poor SPL design decisions. Thus, we considered instability as harmful to the SPL maintainability. Fig. 1 presents instable components in MobileMedia by assigning "I" to each instable component of the SPL. For instance, all presented components of feature *MediaManagement* are instable, except `Constants` and `MediaUtil`. We do not consider comment-related changes in the count of instability.

3 Code Anomalies and Agglomerations in SPL

Section 3.1 discusses code anomalies and agglomerations. Sections 3.2, 3.3, and 3.4 characterize feature, feature hierarchy, and component agglomerations, respectively.

3.1 Agglomerating Code Anomalies

Code anomalies are symptoms of deeper problems in a software system [12]. They make a specific source code element difficult to understand and maintain. Any software system is prone to have anomalies and the SPL variability can introduce anomalies, e.g., because of feature interactions [7]. As an example, a *Long Refinement Chain* occurs when a method has too many successive refinements in different features. It harms the SPL maintainability because it makes harder to understand the side effects caused by changing a feature or selecting a different set of SPL features [7]. The following sections present the definition of three types of agglomerations that take into account the main characteristics of SPLs. We based our definitions on a previous work [21] that investigates agglomerations as indicators of maintenance problems in single systems.

3.2 Feature Agglomeration

We define *feature agglomerations* as follows. Let *f* be a feature and *c* be an anomalous component. Let *c* -> *f* when an anomalous component *c* contributes to implement the feature *f*. A *feature agglomeration* of a feature *f* is a set of anomalous components *C* in which there exists a relation *c* -> *f* for all $c \in C$ and $|C| \geq 2$. There is a simple reason for considering a feature as a natural grouping of code anomalies, i.e., FOP expects that developers implement all components related to a specific functionality of the SPL into the same feature [2, 4]. Although there might be no explicit, syntactic relationship among components of the same feature, they are typically located in the same folder at the SPL source code. Thus, grouping components by feature reflects the semantical relationship among components. With this type of agglomeration, we hypothesize that the occurrence of different anomalies in components of the same feature are indicators of instabilities in SPL. In other words, we analyze anomalies from different components as a single anomalous structure at the feature-level. We expect that this wider view of anomalies may better indicate instabilities in the SPL.

Fig. 1 presents the feature *MediaManagement* with seven constants. For each component, we have the respective number of code anomalies represented by "#" on the upon-right side of the component. All these components are anomalous and, therefore, this set of components corresponds to a feature agglomeration. By analyzing in details each anomalous component separately, we observe that most of them have only one anomaly. For instance, `Constants` and `PhotoViewController` contain only *Long Parameter List*. Although this anomaly is a symptom of maintenance problems, it provides a limited view of the maintenance problem that affects the SPL.

In turn, by analyzing the entire anomaly agglomeration, we may observe wider issues. As an example, the components `MediaController` and `MediaUtil` have both, *God Class*, *Long Method*, and *Long Parameter List*. In general, these anomalies relate to high difficulty to maintain the affected code elements (classes or methods, in this case). Since components of the same feature implement the same functionality, we expect that they access and use to one another. Thus, these anomaly occurrences in the same feature may lead to major maintenance problems in the feature as a whole. More-

over, attempts to treat these problems can lead to the overall feature instability. Therefore, feature agglomeration may help us to understand problems that affect multiple source files in the same feature that implement together the same SPL functionality.

3.3 Feature Hierarchy Agglomeration

We define a *feature hierarchy agglomeration* as follows. Let r be a refinement chain and c be an anomalous component. Let c -> r when an anomalous component c belongs to the refinement chain r. A *feature hierarchy agglomeration* of a refinement chain r is a set of anomalous components C in which there exists a relation c -> r for all $c \in C$ and $|C| \geq 2$. A refinement is an inter-component relation explicitly declared in the refinement's code that indicates the refined constant. For instance, the components `MediaController` of the three features in Fig. 1 compose a refinement chain. Since all these components are anomalous, they form a feature hierarchy agglomeration. We observe that two of the components individually have only one anomaly; `MediaController` of both *MediaManagement* and *SMSTransfer* have only *Long Parameter List*. This anomaly provides a limited view of maintenance problems (Section 3.2).

However, by analyzing in detail the feature hierarchy of `MediaController`, we can reason about major maintenance problems that encompass the entire refinement chain. The component `MediaController` of feature *MediaManagement* is a constant and, therefore, the components below it in the feature hierarchy are refinements. This constant has four code anomalies: *God Class*, *Long Method*, *Long Parameter List*, and *Long Refinement Chain*. The high number of anomalies that affect locally `MediaController` suggests this component has one or more problems. Besides that, there are two other components refining the constant. Because of the *Long Parameter List*, that may indicate an overload of responsibilities in the method, it is even more critical the fact that we have too many refinements of the constant, i.e. the *Long Refinement Chain* is potentially critical. Therefore, the impact of these anomalies is wider than an analysis of individual components may cover. Feature hierarchy agglomeration aims to indicate problems that affect a scattered concern associated with multiple features.

3.4 Component Agglomeration

We define a *component agglomeration* as follows. Let c be a component and e be a code element. Let e -> c when a code element e belongs to the component c. A *component agglomeration* of a component c is a set of anomalous code elements E when there exists a relation e -> c for all $e \in E$ and $|E| \geq 2$. In Fig. 1, the component that contains the highest amount of anomalies is `MediaController` of feature *MediaManagement*. Four anomalies with potential to harm the SPL maintainability occur in this component: *God Class*, *Long Method*, *Long Parameter List*, and *Long Refinement Chain*. By analyzing each anomaly separately, we limit our observations to the possible problems that the respective anomaly may cause. In turn, by agglomerating anomalies that affect the same component may lead to observations that are more conclusive. For instance, if we consider *God Class* and *Long Method* separately, we may overlook two important issues regarding `MediaController`. First, this component is a constant and many other

components refine its implementation. Second, this component has a *Long Refinement Chain* that makes code harder to understand and evolve. This anomaly, summed to the occurrences of *Large Class* and *Long Method*, tend to harm the SPL maintainability even more. Thus, component agglomeration may support the identification of major SPL maintenance problems in a component caused by inter-related anomalies.

4 Study Settings

Section 4.1 describes the study goal and research questions. Section 4.2 presents the target SPLs used in our analysis. Section 4.3 describes the study protocols.

4.1 Goal and Research Questions

We aim to investigate whether non-agglomerated and agglomerated anomalies indicate sources of instability in SPL. Our research questions (RQs) as discussed below.

RQ1. *Can non-agglomerated code anomalies indicate instability in SPL?*
RQ2. *Can agglomerated code anomalies indicate instability in SPL?*
RQ2.1. *How strong is the relationship between agglomerations and instability?*
RQ2.2. *How accurate is the relationship between agglomerations and instability?*

To the best of our knowledge, we did not find studies that investigate non-agglomerated anomalies as indicators of instability in SPL. Therefore, we assess if non-agglomerated anomalies can provide instability hints in SPL (RQ1). RQ2 focuses on the investigation of whether agglomerations can be indicators of instability. We address this question according to two perspectives. First, we compute the strength of the relationship between each type of agglomeration and instability (RQ2.1). That is, we assess the potential of agglomerated anomalies in indicating instabilities. We say a relationship is strong if agglomerated anomalies are able to identify at least 100% more instabilities than non-agglomerated anomalies. We chose this rounded threshold based on the guidelines of Lanza and Marinescu [16]. Second, we then compute the accuracy of agglomerations in identifying instability (RQ2.2), in terms of precision and recall. In other words, we assess if agglomerated anomalies can identify instabilities correctly.

4.2 Target SPLs

We selected four SPLs implemented in AHEAD or FeatureHouse: MobileMedia [10], Notepad [15], TankWar [23], and WebStore [13]. We selected these SPLs for some reasons. First, these SPLs are part of a SPL repository proposed in a previous work [24]. Second, they have been published and investigated in the literature [9, 23]. Third, there are different releases per SPL and, therefore, we could compute instability for the SPLs throughout consecutive releases. MobileMedia provides products for media management in mobile devices, and it has seven releases [9, 24]. Notepad aims to generate text editors and it has two releases [24]. TankWar is a war game for personal computers

and mobile devices and it has seven releases [23]. Finally, WebStore derives Web applications with product management, and it has six releases [9, 24]. Fourth, developers of these SPLs were available for consultation, except in the case of Notepad.

According to the developers of the four SPLs, each of them evolved to address different issues. MobileMedia initially supported photo management only, but evolved to manage other media types, such as video and music. This evolution required a revision of the SPL assets [9]. Notepad was completely redesigned in the two available releases [15]. Developers added new functions and created new ones to ease the introduction of functions and to improve the feature modularization. TankWar evolved only to refactor the SPL without changing any functions but to improve its maintainability. Finally, WebStore initially supported a few payment types and data management options. As WebStore evolved, it has changed to cover these and other new functionalities. Although this is a similar scenario to MobileMedia, the initial development of WebStore took into account future planned evolutions, making this SPL more stable [9].

4.3 Data Collection and Analysis Protocols

Our data collection and analysis comprised three activities presented as follows. The artifacts produced during this process are available in the research website [8].

Identifying Sources of Instabilities. We first computed instability per SPL. We manually computed the number of changes per component between releases. Then, we identified the main sources of instability per SPL, based on the changed components. We used the instability computation for MobileMedia and WebStore provided by a previous work [9]. To increase the data reliability and to compute instability for TankWar and Notepad, we a tool for source code file comparison called WinMerge[1]. We count an instability index if the file changes between consecutive releases. As stated in Section 2.2, we considered as instable a component with two or more changes, due to the few available releases per SPL. Regarding the sources of instabilities, we analyzed the reasons that lead to instability per component to identify groups of components with similar instability sources, e.g. because a new feature was added, and represent a major source of instability. Whenever was possible, we validated the detected instability with developers of the target SPL by showing them the numbers obtained per component.

Table 1 presents the sources of instabilities identified in the four analyzed SPLs. The first column indicates the category and the sum of affected components per source. The second column presents the description of each source of instability. The last line (i.e., Others) represents the sources of instability that we were not able to categorize. As an example, we named "Add crosscutting feature" when a new feature is added to the SPL and it affects the implementation of existing features. This particular instability is interesting in the SPL context because, according to the open/closed principle, software entities should be open for extension, but closed for modification [19].

[1] http://winmerge.org/

Table 1. Sources of Instabilities in SPL

Source	Description
Add crosscutting feature (122)	When we add a new feature to the SPL and, consequently, the new functionalities are of interest of components from several existing features. Many components from different features change
Distribute code among features (39)	When we extract code parts of a component from an existing feature and, then, distributed these code parts to components from existing features
Change from mandatory to optional (19)	When we distribute the implementation of an existing feature to: (i) a new, basic mandatory feature, and (ii) a new optional features, with specific functionalities
Pull up common feature code (63)	When we extract code parts that are common into child features to a parent feature above in the feature hierarchy
Others (195)	General sources unrelated explicitly to SPL maintenance, e.g. attribute renaming

Identifying Code Anomalies and Agglomerations. Our process of identifying code anomalies consists in three steps: (i) to define the anomalies for study, (ii) to define the metric-based detection strategies to identify each anomaly, and (iii) to apply the defined detection strategies to each SPL. We investigate eight anomalies defined in our website [8], namely: *Data Class*, *Divergent Change*, *God Class*, *Lazy Class*, *Long Method*, *Long Parameter List*, *Shotgun Surgery* [12, 16], and *Long Refinement Chain* [7]. Our analysis relies mostly on such general-purpose anomalies, except for *Long Refinement Chain* [7], but all of them relate somehow to the SPL composition. These anomalies affect the source code of SPLs in different levels, including feature hierarchies.

As an example, *Divergent Change* is a class that changes due to divergent reasons [16]. If these reasons relate to different features, this anomaly may harm the SPL modularization. *Long Method* is a method with too many responsibilities [12]. This anomaly is harmful in SPLs if the responsibilities of the method relate to different features, for instance. Finally, *Long Refinement Chain* [7] is a method with excessive number of successive refinements. This SPL-specific anomaly is harmful since it hampers the understanding of side effects of changes in the generation of SPL products. To detect each anomaly, we adapted detection strategies from the literature [16] whenever possible. We extracted the metric values per SPL via the VSD tool [24]. Once detected the anomalies, we computed manually the three types of agglomerations per SPL (see Section 3). Two authors contributed to double-check the results in order to prevent errors.

Correlating Agglomerations and Instabilities. To answer our research questions, we defined a criterion for correlating agglomerations and instabilities. Consider a general agglomeration that can be either a feature, a feature hierarchy, or even a component agglomeration. We say that such agglomeration indicates an instability when there exists an instable code element in the feature, feature hierarchy, or component that have the agglomeration. Even though agglomerations and instabilities may be located in more than two anomalous elements, our criterion considers sufficient if the agglomeration is affected by at least one problem. Thus, an agglomeration fails to indicate instability when none of its components relates to an instability. With respect to the number of agglomerations that indicate instability in the analyzed SPLs, we observed that an average of 94%, 78%, and 32% of the agglomerations indicate 2 or more instable components for feature, feature hierarchy, and component agglomeration respectively.

5 Results and Analysis

Section 5.1 presents the results for non-agglomerated anomalies. Section 5.2 discusses the results for the three proposed types of anomaly agglomeration in SPLs.

5.1 Non-Agglomerated Code Anomalies

First, we investigate whether non-agglomerated code anomalies are sufficient indicators of instabilities in SPL. Therefore, we aim to answer RQ1.

RQ1. *Can non-agglomerated code anomalies indicate instabilities in SPL?*

We computed the strength of the relation between non-agglomerated anomalies and instabilities via Fisher's test [11]. We also used the Odds Ratio [5] to compute the possibility of the presence or absence of a property (i.e., the non-agglomeration) to be associated with the presence or absence of other property (i.e. instability). We computed both statistics via the R tool[2]. Table 2 presents the results for non-agglomerated anomalies. The first column lists each SPL. The second column present the number of non-agglomerated anomalies that indicate instabilities. The third column presents the number of agglomerated anomalies that do not indicate instabilities, i.e. they indicate stability. The fourth column presents the total number of anomalies per SPL.

Table 2. Analysis Results for Non-Agglomerated Anomalies

SPL	Non-Agglomerated and Instability	Agglomerated and Stability	Total Number of Anomalies
MobileMedia	1	11	87
Notepad	0	1	24
TankWar	0	2	106
WebStore	0	4	29

By comparing the second and third columns, we observe that for the 4 SPLs the number of non-agglomerated anomalies that indicate instability is very low. In general, this number is even lower than the number of agglomerated anomalies that indicate stability. Since each SPL has several anomalies (fourth column), we may assume that agglomerations are potentially useful to identify instabilities in SPL. In addition, considering all the four analyzed SPLs, we have a p-value of 0.1488 and Odds Ratio equals 0.0816. Thus, our results suggest that the possibility of a non-agglomerated anomaly to indicate instabilities is close to 0 when compared with an agglomerated anomaly.

Summary for RQ1. Our data suggest that non-agglomerated anomalies may not suffice to indicate instabilities in SPL. The low number of non-agglomerated anomalies that indicate instabilities supports this finding. On the other hand, there is a potential for agglomerations in indicating instabilities.

[2] https://cran.r-project.org/

5.2 Agglomerated Code Anomalies

In this section, we analyze the relationship between agglomerations and instabilities. We aim to answer RQ2 decomposed into RQ2.1 and RQ2.2 discussed as follows.

RQ2.1. *How strong is the relationship between agglomerations and instability?*

Table 3 presents the results per type of agglomeration. The first column lists each type of agglomeration. The second column presents the number of agglomerations that indicate correctly an instability for the four analyzed SPLs. The third column presents the number of non-agglomerations that does not indicate instability. The last two columns present the p-value computed via Fisher's test and the results for Odds Ratio.

Table 3. Analysis Results for Agglomerated Anomalies

Type of Agglomeration	Agglomeration and Instability	Non-Agglomeration and Stability	p-value	Odds Ratio
Feature	31	6	1	1.1598
Feature Hierarchy	28	13	0.0478	3.8492
Component	28	124	0.8761	0.9290

Note that, for all types of agglomerations, we obtained similar numbers of agglomerations that indicate instability, but the values of non-agglomerations that indicate stability vary according to the type of agglomeration. Regarding p-value, we assume a confidence level higher than 95%. Only feature hierarchy agglomerations presented p-value lower than 0.05 and, therefore, it is the only type of agglomeration with statistical significance with respect to the correlation between agglomerations and instabilities. Regarding Odds Ratio, we have a value significantly greater than 1 only for feature hierarchy agglomerations, around 3.8. That means that the possibility of a feature hierarchy agglomeration to relate with instabilities is almost 4 times higher than a non-agglomerated code anomaly. For the other two types of agglomerations, we have values close to 1 and, therefore, we may not affirm that such types of agglomeration have more possibilities to "host" instabilities when compared to non-agglomerated anomalies.

Thus, regarding RQ2.1, we conclude that the relationship between agglomerations and instabilities is strong for *feature hierarchy agglomeration*. We then answer RQ2 partially. This observation is quite interesting, since in FOP the features encapsulate the implementation of SPL functionalities. Besides that, our data suggest the refinement relationship may hinder this encapsulation by causing instability into multiple features. This problem is even more critical since the instabilities caused by a feature hierarchy agglomeration can eventually propagate to several seemly-unrelated SPL products.

We also investigate the accuracy of code anomaly agglomerations to indicate instabilities in SPLs, per type of agglomeration. We answer RQ2.2 as follows.

RQ2.2. *How accurate is the relationship between agglomerations and instability?*

To assess accuracy of each type of agglomeration, we compute precision and recall in terms of true positives (TP), false positives (FP), true negatives (TN), and false negatives (FN) [6]. TP is the number of agglomerations that indicate correctly instabilities. FP is the number of agglomerations that indicate incorrectly instabilities, i.e. indicate stability. TN is the number of non-agglomerations that does not indicate instability.

Finally, FN is the number of non-agglomerations that indicate instability. The formula for precision and recall are $P = TP / (TP + FP)$ and $R = TP / (TP + FN)$ [6].

Since even small-sized systems have several anomalies [17], developers should focus their maintenance effort on anomalies that represent the most critical maintenance problems. Thus, agglomerating anomalies can reduce the search space for finding those problems. We focus our analysis on accuracy computed in terms of precision and recall. In this study, we compute precision and recall per type of agglomeration considering all the instable components, regardless the sources of instability of each component. We made this decision because some instable components have multiple sources that relate to different types of agglomeration. For instance, the component `MediaController` of feature *MediaManagement* has changed because of an "Add crosscutting feature" and a "Distribute code among features" in MobileMedia, Release 4.

Table 4 presents precision (P), recall (R), and the number of instable components indicated per type of agglomeration (#IC). This table also presents median, mean, and standard deviation for the results obtained for the four SPLs under analysis. We provide a discussion of our results per type of agglomeration as follows.

Table 4. Precision and Recall per Type of Agglomeration

Agglomeration	**Feature**			**Feature Hierarchy**			**Component**		
SPL	**P**	**R**	**#IC**	**P**	**R**	**#IC**	**P**	**R**	**#IC**
MobileMedia	76%	72%	65	100%	59%	30	50%	10%	8
Notepad	50%	20%	4	75%	50%	8	50%	25%	3
TankWar	92%	61%	37	82%	82%	66	65%	23%	17
WebStore	75%	60%	26	100%	24%	10	0%	0%	0
Median	**76%**	**61%**	**32**	**91%**	**54%**	**20**	**50%**	**16%**	**6**
Mean	**73%**	**53%**	**33**	**89%**	**54%**	**29**	**41%**	**14%**	**7**
Std. Dev.	**15%**	**20%**	**22**	**11%**	**21%**	**23**	**25%**	**10%**	**6**

Feature Agglomeration. The first three columns in Table 4 correspond to the results for feature agglomeration. We observed a precision with median of 76% and mean of 73%. We then observe that each 3 out of 4 feature agglomerations indicate instabilities. These results are expressive if we consider that agglomerations aim to provide a precise indication of instability, based on high frequencies of code anomalies that may occur in any system. To illustrate the effectiveness of a feature agglomeration in indicating instability, let us consider again the example of MobileMedia from Section 3.2. In fact, the feature agglomeration formed by components from feature *MediaManagement* indicated relevant instabilities generated by a source of instability categorized as "Distribute code among features". In this case, the implementation of the component `BaseController` from feature Base, the most important controller of the SPL, was distributed to several features including *MediaManagement*. This distribution of source code to other features made the components of the feature agglomeration instable.

Regarding recall, we obtained a mean of 53%, with median of 61%, for the SPLs under analysis. We observed a percentage of recall equals or higher than 60% in 3 out of 4 SPLs. Indeed, low percentages of recall are expected in this study, since not all instabilities in SPL are related to anomalous code structures. Through a manual analysis of the four SPLs, we identified various sources of instability that do not relate with code anomalies. For instance, in MobileMedia some components have changed from one

release to another because of the inclusion of new functionalities by means of features (e.g., in Releases 1 to 2). In TankWar, some components have changed due to the inclusion of FOP-specific mechanisms (e.g., in Releases 2 to 3).

Note that, for Notepad, the low rates of both precision and recall may be justified by the small percentage for both instable and anomalous components. As an example, Notepad has only 37.5% of instable components, against 58.9%, 79.3%, and 44.2% for MobileMedia, TankWar, and WebStore respectively. Despite of that, in general our results suggest that there is a high rate of feature agglomerations that, possibly, may indicate instabilities in the SPLs. However, since we did not observe statistical significance for this type of agglomeration (see Section 5.2), we may not affirm that feature agglomerations are indicators of instability in SPLs.

Feature Hierarchy Agglomeration. The three next columns in Table 4 present precision, recall, and #IC for the analysis of feature hierarchy agglomeration. We obtained values similar to the first analysis, with respect to the feature analysis. First, regarding precision, we have a mean value of 89%, the highest value among types of agglomeration. This data suggests the only a few feature hierarchy agglomerations – that is, related to a refinement chain formed by components and its refinements – are not related to instabilities. We additionally obtained a mean recall of 54% for the target SPLs, that is, the best value among agglomeration types. This result indicates that a significant number of feature hierarchy agglomerations are candidates to indicate instabilities. We conclude that the feature hierarchy agglomeration is an indicator of instabilities in SPL.

To illustrate a feature hierarchy agglomeration that indicated instability, let us consider the example of MobileMedia from Section 3.3. The feature hierarchy agglomeration formed by components of the refinement chain of `MediaController` indicated several relevant sources of instability. For instance, this agglomeration captured the instability caused by a source categorized as "Pull up common feature code". In this case, due to the addition of new types of media in MobileMedia, it was reorganized the implementation of feature *CopyPhoto* into two features: *CopyPhoto* and *CopyMedia*. This change affected all components from the agglomeration in terms of instability.

Component Agglomerations. The three last columns in Table 4 present precision, recall, and #IC for the analysis of component agglomeration. In this case, we obtained values significantly different when compared to the feature agglomeration analysis. With respect to the four SPLs, we obtained a mean precision of 41%. This result points that less than a half of the observed component agglomerations relate, in fact, to instabilities. Based on this data, we may not affirm that this type of agglomerations is effective in indicating instabilities. Moreover, we obtained a mean recall of 14% for the SPLs. This result is very low when considering that systems tend to present several instable components and code anomalies. Therefore, our data suggests that the component agglomeration is not an indicator of instabilities in SPL.

Although precision and recall are, in general, low, we observed interesting cases of component agglomerations that indicate instabilities. Consider the example presented in Section 3.4. Code elements from the component `MediaController`, of the feature *MediaManagement*, indicated correctly different sources of instability. These sources

include (i) “Distribute code among features” regarding the implementation of component `BaseController` from feature *Base* and (ii) “Pull up common feature code” regarding the reorganization of feature *CopyPhoto*. We discuss both sources previously in this section, for feature agglomeration and feature hierarchy agglomeration.

Summary for RQ2. Our data suggest that feature hierarchy is the most effective type of agglomeration for identification of sources of instability in SPLs, due to the p-value lower than 0.05 (given a 95% confidence interval) and the highest Odds Ratio close to 3.8. When compared to non-agglomerated anomalies (RQ1), with Odds Ratio equals 0.08, we observe that feature hierarchy agglomeration is 3.8 times more effective in identifying instabilities. The high precision of 89% for this type reinforces our findings.

6 Related Work

Previous works propose or investigate anomalies that indicate potentially SPL maintenance problems [2, 7]. Apel et al. [2] introduce the term “variability smell”, i.e. anomalies that capture the notion of SPL variability, and present a set of 14 anomalies that may occur in different phases of the SPL engineering. In turn, Fenske and Schulze [7] provide a complementary set of variability-aware anomalies, besides of an empirical study to assess the occurrence of these anomalies in real SPLs. However, none of these studies neither has used anomaly agglomeration nor has analyzed instability.

In particular, Oizumi et al. [21] investigate the use of inter-related anomalies, i.e. anomaly agglomerations, to identify design problems in general source code. They define strategies to group different anomaly occurrences in source code elements. The authors discuss that the defined agglomerations are better indicators of design problems than non-agglomerated anomalies. The results suggest that some types of agglomeration can indicate sufficiently problems with accuracy higher than 80%. However, the authors do not explore neither instability as a design problem nor the relationship between agglomerations and instability in SPL. In turn, this paper focus on the analysis agglomerations as indicators of instability in the context of feature-oriented SPL.

7 Threats to Validity

We discuss threats to the study validity, with respective treatments, as follows.

Construct and Internal Validity. We carefully designed our study for replication. However, a major threat to our study is the set of metrics used in the detection strategy composition. This set is restricted to the metrics provided by the SPL repository [24] adopted in our study. To minimize this issue, we selected some well-known and largely studied metrics, such as *McCabe’s Cyclomatic Complexity (Cyclo)*. The list of detection strategies used in this study is available in the research website [8]. Regarding the small length of the analyzed SPLs, we highlight the limited number of SPLs available for research, as the limited number of releases for the available SPLs. The low number of available releases has lead us to consider a component as instable if it has changed in two or more releases. To minimize this issue, we analyzed the SPLs in all available

releases. Finally, we conducted the data collection carefully. To minimize errors, two authors checked all the collected data and re-collected the data in case of divergence.

Conclusion and External Validity. We designed a data analysis protocol carefully. To compute the statistical significance and strength of the relationship between agglomerations and instabilities, we computed the Fisher's test [11] and Odds Ratio [5], two well-known and reliable techniques. We also computed precision and recall for the accuracy analysis of agglomerations, based on previous work [21]. These procedures aim to minimize issues regarding the conclusions we draw. Two authors checked the analysis to avoid missing data and re-conducted the analysis to prevent biases. Regarding the generalization of findings, we expect that our results are extensible to other SPL development contexts than FOP. However, further investigation is required.

8 Conclusion and Future Work

Some studies assume that each code anomaly alone suffices to characterize SPL maintenance problems [7, 23]. Nevertheless, each single anomaly may represent only a partial view of a problem. To address this issue, a previous work investigates to what extent agglomerating code anomalies may support the characterization of maintenance problems in single systems [21]. However, we lack studies to investigate and compare the use of anomalies and their agglomerations as indicators of problems that harm the SPL maintainability. In this paper, we focus on a specific maintenance problem in SPLs: instability. We first investigate if non-agglomerated anomalies may indicate instability in SPL. Our findings suggest that non-agglomerated anomalies do not support the identification of anomalous code structures that cause instability. We then investigate to what extent anomaly agglomerations represent sources of instability in SPL. Our study relies on the analysis of different releases of four feature-oriented SPLs.

Our data suggest that feature hierarchy agglomeration, one of the three types of agglomeration proposed in this study, is up to 3.8 times more effective than non-agglomerated anomalies in identifying sources of instability in SPL. The high precision of 89% for this type of agglomeration reinforces that it can support developers in anticipating critical instabilities that harm the SPL maintainability. These findings have clear implications in the FOP development. Since feature hierarchies are a basis for FOP, developers of feature-oriented SPLs should design carefully feature hierarchies to prevent the implementation of hierarchical structures that hamper the SPL maintainability. As future work, we intend to investigate alternative types of agglomeration for other SPL maintainability problems, as the impact of different anomalies on instability.

Acknowledgments. This work was partially supported by CAPES/Procad, CNPq (grants 424340/2016-0 and 290136/2015-6), and FAPEMIG (grant PPM-00382-14).

References

1. Ampatzoglou, A., Chatzigeorgiou, A., Charalampidou, S., Avgeriou, P.: The Effect of GoF Design Patterns on Stability. IEEE Trans. Softw. Eng. 41, 8, 781–802 (2015)

2. Apel, S., Batory, D., Kästner, C., Saake, G.: Feature-Oriented Software Product Lines. Springer (2013)
3. Apel, S., Kätner, C., Lengauer, C.: FeatureHouse. In: 31st ICSE, pp. 221–231 (2009)
4. Batory, D., Sarvela, J., Rauschmayer, A.: Scaling Step-Wise Refinement. In: 25th International Conference on Software Engineering (ICSE), pp. 187–197 (2003)
5. Cornfield, J.: A Method of Estimating Comparative Rates from Clinical Data. Journal of the National Cancer Institute 11, 6, 1269–1275 (1951)
6. Fawcett, T.: An Introduction to ROC Analysis. Pattern Recogn. Lett. 27, 8, 861–874 (2006)
7. Fenske, W., Schulze, S.: Code Smells Revisited. In: 9th VaMoS, pp. 3–10 (2015)
8. Fernandes, E., Vale, G., Sousa, Figueiredo, E., L., Garcia, A., Lee, J.: No Code Anomaly is an Island: Anomaly Agglomeration as Sign of Product Line Instabilities – Data of the Study. http://labsoft.dcc.ufmg.br/doku.php?id=about:no_code_anomaly_is_an_island
9. Ferreira, G., Gaia, F., Figueiredo, E., Maia, M.: On the Use of Feature-Oriented Programming for Evolving Software Product Lines. Sci. Comput. Program. 93, 65–85 (2014)
10. Figueiredo, E., Cacho, N., Sant'Anna, C., Monteiro, M., Kulesza, U., Garcia, A., Soares, S., Ferrari, F., Khan, S., Castor Filho, F., Dantas, F.: Evolving Software Product Lines with Aspects. In: 30th Int'l Conf. on Softw. Eng. (ICSE), pp. 261–270 (2008)
11. Fisher, R.: On the Interpretation of x^2 from Contingency Tables, and the Calculation of P. Journal of the Royal Statistical Society 85, 1, 87–94 (1922)
12. Fowler, M.: Refactoring. Object Technology Series. Addison-Wesley (1999)
13. Gaia, F., Ferreira, G., Figueiredo, E., Maia, M.: A Quantitative and Qualitative Assessment of Aspectual Feature Modules for Evolving Software Product Lines. Science of Computer Programming 96, 2, 230–253 (2014)
14. Khomh, F., Di Penta, M., Gueheneuc, Y.: An Exploratory Study of the Impact of Code Smells on Software Change-Proneness. In: 16th WCRE, pp. 75–84 (2009)
15. Kim, C., Bodden, E., Batory, D., Khurshid, S.: Reducing Configurations to Monitor in a Software Product Line. In: 1st Int'l Conf. on Runt. Verif. (RV), pp. 285–299 (2010)
16. Lanza, M., Marinescu, R.: Object-Oriented Metrics in Practice. Springer (2006)
17. Macia, I., Garcia, J., Popescu, D., Garcia, A., Medvidovic, N., von Staa, A.: Are Automatically-Detected Code Anomalies Relevant to Architectural Modularity? In: 11th Int'l Conference on Aspect-Oriented Software Development (AOSD), pp. 167–178 (2012)
18. Medeiros, F., Kästner, C., Ribeiro, M., Nadi, S., Gheyi, R.: The Love/Hate Relationship with the C Prepocessor. In: 29th ECOOP, pp. 495–518 (2015)
19. Meyer, B.: Object-Oriented Software Construction. Prentice Hall (1988)
20. Moha, N., Gueheneuc, Y., Duchien, L., Le Meur, A.: DECOR. IEEE Transactions on Software Engineering 36, 1, 20–36 (2010)
21. Oizumi, W., Garcia, A., Sousa, L., Cafeo, B., Zhao, Y.: Code Anomalies Flock Together. In: 38th International Conference on Software Engineering (ICSE), pp. 440–451 (2016)
22. Pohl, K., Böckle, G., van der Linden, F.: Software Product Line Engineering. Springer Science & Business Media (2005)
23. Schulze, S., Apel, S., Kästner, C.: Code Clones in Feature-Oriented Software Product Lines. In: 4th GPCE, pp. 103–112 (2010)
24. Vale, G., Albuquerque, D., Figueiredo, E., Garcia, A.: Defining Metric Thresholds for Software Product Lines. In: 19th SPLC, pp. 176–185 (2015)
25. Yau, S., Collofello, J.: Design Stability Measures for Software Maintenance. IEEE Transactions on Software Engineering 11, 9, 849–856 (1985)