Do Critical Components Smell Bad? An Empirical Study with Component-based Software Product Lines

Anderson Uchôa
Informatics Department, PUC-Rio
Rio de Janeiro, RJ, Brazil
auchoa@inf.puc-rio.br

Wesley K. G. Assunção
Informatics Department, PUC-Rio
Rio de Janeiro, RJ, Brazil
wassuncao@inf.puc-rio.br

Alessandro Garcia
Informatics Department, PUC-Rio
Rio de Janeiro, RJ, Brazil
afgarcia@inf.puc-rio.br

ABSTRACT

Component-based software product line (SPL) consists of a set of software products that share common components. For a proper SPL product composition, each component has to follow three principles: encapsulating a single feature, restricting data access, and be replaceable. However, it is known that developers usually introduce anomalous structures, i.e., code smells, along the implementation of components. These code smells might violate one or more component principles and hinder the SPL product composition. Thus, developers should identify code smells in component-based SPLs, especially those affecting highly interconnected components, which are called critical components. Nevertheless, there is limited evidence of how smelly these critical components tend to be in component-based SPLs. To address this limitation, this paper presents a survey with developers of three SPLs. We inquire these developers about their perceptions of a critical component. Then, we characterize critical components per SPL, and identify nine recurring types of code smells. Finally, we quantitatively assess the smelliness of the critical components. Our results suggest that: (i) critical components are ten times more prone to have code smells than non-critical ones; (ii) the most frequent code smell types affecting critical components violate several component principles together; and (iii) these smell types affect multiple SPL components.

CCS CONCEPTS

• **General and reference** → **Empirical studies**; • **Software and its engineering** → **Abstraction, modeling and modularity**; **Software product lines**; **Software evolution**.

KEYWORDS

Component-based software product line, smell, empirical study

ACM Reference Format:
Anderson Uchôa, Wesley K. G. Assunção, and Alessandro Garcia. 2021. Do Critical Components Smell Bad? An Empirical Study with Component-based Software Product Lines. In *15th Brazilian Symposium on Software Components, Architectures, and Reuse (SBCARS '21), September 27-October 1, 2021, Joinville, Brazil.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3483899.3483907

SBCARS '21, September 27-October 1, 2021, Joinville, Brazil

1 INTRODUCTION

Component-based software product line (SPL) consists of a set of software products that share common components [4]. The SPL product composition requires that each component follows three principles [10, 21, 26] summarized as follows. First, the feature decomposition into components should be such that each component encapsulates only a single feature [26]. Second, each component should restrict data access with respect to other components through an explicit interface [21]. Third, the component has to be as replaceable as possible, allowing developers to easily replace an existing component with another that realizes the same feature [10].

Despite the developers' efforts for following the three component principles, they eventually introduce anomalous structures, i.e., code smells, in the SPL components. Consequently, code smells [20] usually violate those component principles [27, 30, 34]. For instance, *Feature Envy* [20] occurs when part of the component's code is more interested in data of other components. *Feature Envy* induces inter-dependencies that violate both encapsulation and data access principles. There is evidence that software systems might have hundreds of code smells [30] that violate various design principles [34]. A similar reasoning applies to SPL [1, 48].

However, the negative effect of code smells in SPL is even more important than it is in a single system [2]. Such effect can also include the propagation of maintenance problems to multiple SPL products. Thus, the SPL developers need to prioritize the identification and elimination of code smells [1]. In component-based SPLs, such a need is essential when handling with highly interconnected components, which the literature refers to as critical components [4, 24]. That is the case of *Feature Envy*, illustrated above, that affects the SPL product compositions by forcing the inclusion of unnecessary components in a SPL product just because of the "envied" responsibilities with respect to other components.

Critical components often implement essential SPL features, which makes them part of several products [4]. The literature assumes that critical components are recurrently maintained in a system [4, 24]. However, multiple code smells might affect the SPL critical components. Thus, eliminating code smells affecting critical components is of paramount importance. In order to support the identification and elimination of code smells in SPLs, developers need a clear understanding of the characteristics of such smells in critical components. However, there is limited evidence on how smelly theses critical components tend to be in component-based SPLs. This makes the proposal of approaches and development of tools hard to deal with code smells, towards improving the component-based SPL maintainability.

This paper addresses the aforementioned limitations through an empirical study. First, we collect three Java component-based SPLs

from GitHub. Second, we survey the SPL developers about their perceptions of a critical component to build a reliable reference list of critical components per SPL. Third, based on the SPL developers' perception, we select software metrics to support the detection of critical components via a lightweight metric-based strategy. After, we compute nine recurring types of code smells and map on the component principles that they violate. It is important to highlight that it is *not* the goal of this study to focus on variability-related smells; this topic is covered by other studies reported in the literature [2, 14, 16]. We summarize our study findings as follows.

- Critical components are at least ten times more prone to have code smells than non-critical components. It suggests the need for prioritizing code smells affecting critical components rather than non-critical components.
- The most frequent code smell types affecting the critical SPL components violate at least two out of the three component principles. For instance, *Feature Envy*, which violates both encapsulation and data access principles, was the most frequent code smell type.
- The majority of code smells affecting critical components also affects multiple SPL components together. It suggests a broad effect of code smells on the SPL product composition.

All the aforementioned findings serve as recommendations to SPL developers aimed at prioritizing code smells for elimination. Moreover, our results, shed light for both SPL researchers and developers to begin more aware of the smelliness of critical components.

2 BACKGROUND AND RELATED WORK

This section provides the background information of the paper and an overview of the related works.

2.1 Component-based SPL

Major companies such as Boing [39] and Bosch [43] have adopted component-based SPL to boost product composition. By following the three component principles – feature encapsulation [26], data access restriction [21], and replaceability [10] – component-based SPL makes easier to compose SPL products. There are several composition models aimed at supporting the implementation of component-based SPLs [13, 40]. These models guide the feature decomposition into components to address all three component principles. However, it might cause certain differences between the SPL implementation and the SPL feature model. We further discussed these differences in Section 7.

2.2 Critical Components and Software Metrics

Along feature decomposition into components, SPL developers eventually find certain components that are inherently more essential than others [24]. Consequently, these components are essential to the SPL product composition. Existing studies [4, 24] report that critical components are recurrently maintained in a software system. However, those studies mostly characterized critical components in a subjective manner rather than based on empirical evidence. In general, they assumed that an SPL component is critical when it is highly interconnected with other ones in the SPL. However, this assumption lacked empirical evidence with respect to the SPL developers' perception of critical component.

Software metrics have been shown useful to characterize different software properties [12, 24, 25, 28]. For instance, some studies [24, 25] apply software metrics to assess the component reusability of either single software systems or SPLs. Another study [12] uses metrics to guide the management of component-based software systems. However, none of the aforementioned studies aimed at characterizing critical components in component-based SPLs. To support such characterization, we rely on a lightweight metric-based strategy to characterize critical components that we have proposed [45]. This strategy relies on traditional coupling metrics, such as Coupling between Objects (CBO) [8]. Additionally, our strategy has pointed out around 86% of accuracy in the detection of critical components in component-based SPLs.

2.3 Code Smells

Code smells are indicators of problems in software systems [20]. These problems affect any software system [30, 34], including SPLs [16, 32]. In component-based SPLs, code smells can harm the SPL product composition by violating the three component principles which are feature encapsulation [26], data access restriction [21], and replaceability [10]. In addition to *Feature Envy*, illustrated in Section 1, other code smells violate these principles. For instance, *Divergent Change* [20] occurs when a component has to change whenever others change due to inter-dependencies. Thus, *Divergent Change* violates both feature encapsulation and replaceability. Similar violations are found in *Shotgun Surgery* [20], which occurs when changing a component implies changing many others.

Previous studies [30, 32, 34] provide evidence that code smells are sufficient indicators of code parts affected by poor feature decomposition, in the case of single software systems. We extend this assumption to component-based SPLs, by relying on studies that investigate the smelliness of SPLs implemented using other composition mechanisms than components [14, 16]. In fact, as aforementioned, *Feature Envy*, *Divergent Change*, and *Shotgun Surgery* are types that somehow violate the component principles, which affect the SPL product composition. Section 4.2 further details how each selected code smell type harms the SPL product composition.

2.4 Related Work

We found studies [2, 3, 16] that investigate the effects of code smells on SPL feature decomposition, but none of them assesses component-based SPL. Apel et al. [3] present a set of 14 code smells (most related to variability smells) that may occur in different phases of the SPL engineering. Andrade et al. [2] assesses the frequency of four code smell types affecting the architecture of a single SPL. The study results suggest that the granularity of feature decomposition into components might influence the frequency of code smells. Finally, Fernandes et al. [16] assesses to what extent inter-related code smells affecting four SPLs might indicate architectural problems. The authors observed that some types of inter-related code smells could help anticipating the number of changes required by certain SPL architectural components.

Differently from those existing studies, we investigate the smelliness of component-based SPLs. Due to our concern about critical components, we had to properly characterize these components. However, the limited empirical evidence available to support such

characterization led us to apply a survey with SPL developers. We aimed at understanding their perception of a critical component. After, similarly to the first study [2], we assess both frequency and granularity of code smell types affecting the selected SPLs. Besides that, we assess the density and proportion of code smells in both critical and non-critical components. We relied on the other study [16] to select the code smell types.

3 MOTIVATING EXAMPLE

In this section, we use the NotePad SPL to motivate our study on code smells of critical components in component-based SPLs. Our choice relies on the availability of design documentation that allowed us to derive the feature model and other useful information for this SPL. NotePad is designed to create text editors composed of basic features, such as text edition, file management, and user interface. Figure 1 presents the NotePad's partial feature model, which includes only the main SPL features. Our goal is to illustrate the negative effects of code smells on the MainWindow component, which the NotePad developers reported as a critical component. This component realizes the text editor's interface.

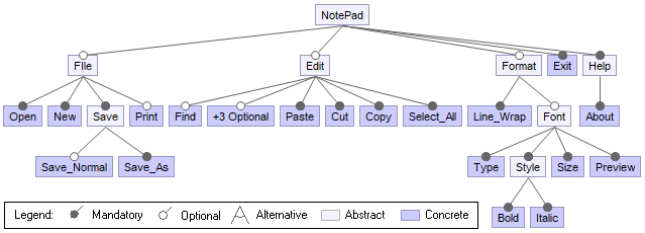

Figure 1: Partial Feature Model of the NotePad SPL

Figure 2 presents the partial design of NotePad [41]. Dashed boxes represent packages, while continuous boxes represent the SPL components. Each component is annotated with the number of code smell instances affecting it and whether the component is critical from NotePad SPL developers' perception. This information was obtained from the results of our study (see Section 6), as NotePad is one of the subject systems, which is used here for illustration.

Let us consider the `MainWindow`[1] component, from the View package, to exemplify how code smells harm the SPL product composition. `MainWindow` is responsible for implementing the main screen of the NotePad user interface. Thus, `MainWindow` is essential to the SPL product composition and should be included in all products. Moreover, this component has several dependencies with other components, such as `MainPanel` and `FileController` from View and Controller packages, respectively. Such dependencies are required to realize the main screen mostly implemented by `MainWindow`. Consequently, including `MainWindow` in any product implies including all dependent components.

As we can observe in Figure 2, `MainWindow` is affected by three code smell instances, which are either *Feature Envy* or *Large Class*. *Feature Envy* suggests a violation of the feature encapsulation principle with respect to `MainWindow`. That is, some parts of the component's source code are scattered over the other components that it depends on. This implies several unnecessary dependencies among `MainWindow` and other components. As a result, any SPL product composition that includes the smelly component would also have to include the other components, which could be avoided by eliminating the code smell. *Feature Envy* also suggests a violation of the data access restriction principle in the dependent components.

Conversely, *Large Class* suggests that `MainWindow` implements too many responsibilities and possibly violates the feature encapsulation principle. That is, `MainWindow` should be split into multiple architectural components in such a way that each component implements a single feature. Otherwise, certain SPL product compositions that require only a part of those features would have to include the smelly component, which includes unnecessary features. In case of eliminating the code smells, the same SPL product composition would include only the components that implement required features. In summary, we hypothesize that code smells actually harm the SPL product composition. Therefore, we investigate to what extent code smells affect different component-based SPLs, by violating the main component principles that are essential to assuring a proper SPL product composition.

[1]Originally called `JanelaPrincipal` in the Portuguese language.

4 STUDY SETTINGS

The artifacts and data used in our study are available on our companion website [46].

4.1 Goal and Research Questions

During the development of component-based SPLs, developers should follow all three component principles whenever possible [22]. To address these principles is essential identifying and eliminating code smells that harm the SPL product composition. Due to limited empirical knowledge on this matter, this paper assesses *the smelliness of components in component-based SPLs*. We describe our study goal based on well-known guidelines [49] as follows: *analyze* SPL components; *for the purpose of* assessing how smelly are (non-)critical components; *with respect to* code smell proportion, types per number of violated principles, and granularity; *from the viewpoint of* developers; *in the context of* three component-based SPLs. We detail our three research questions (RQ$_s$) as follows.

RQ$_1$: *Are critical SPL components more smell-prone than non-critical SPL components?* This RQ assesses the smell-proneness of critical components when compared with non-critical components. Whether code smells affect more often critical components, it is worthwhile to investigate their effect on SPL product composition. We measure smell-proneness through two metrics: density, i.e., how many code smells affect the critical components per SPL, and proportion, i.e., the probability of critical components being affected by code smells against non-critical components. Density aims to capture the smell distribution among SPL critical components. Proportion aims at revealing whether critical components are more smelly-prone than non-critical components.

RQ$_2$: *Do the smell types affecting critical SPL components violate multiple component principles?* This RQ assesses if the most frequent code smells affecting critical components are the ones that violate various component principles together. This knowledge is essential to prioritize code smells for elimination. That is because critical components are often the target of maintenance tasks. Thus, there is a need for improving the maintainability of

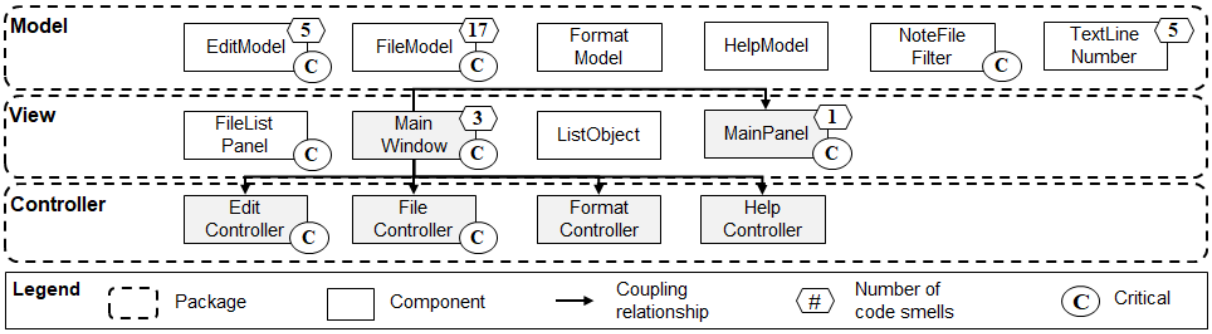

Figure 2: Partial Design of the NotePad SPL

these components whenever possible. We answer **RQ**$_2$ as follows. First, we compute the most frequent code smell types affecting critical components per SPL. We then assess what component principles the most frequent code smell types violate. Additionally, we compute the ranks of most frequent code smell types considering different criticality levels (discussed in Section 4.2).

RQ$_3$: ***Do the smell types affecting critical SPL components also affect multiple SPL components together?*** This RQ assesses the smell granularity of the most frequent code smell types affecting critical components. Similar to previous work [19, 36], we consider granularity as the effect range of a code smell, e.g., a method or a component. Depending on the smell granularity, the effects on SPL product composition might change. To answer **RQ**$_3$, we classified the most frequent code smell types affecting critical components per granularity. Then, we assess the possible negative effects of each code smell granularity by relying on well-known code smell catalogs [31, 35, 50]. These catalogs classify smell types by their scattering over components.

4.2 Study Steps

To conduct our study we followed nine steps, as follows.

Step 1: Select target component-based SPLs. We selected component-based SPLs developed in Java given the high language popularity.[2] For this purpose, we conducted an ad-hoc literature review to identify Java component-based SPLs. We prioritized SPLs used in previous studies, and that has the source code available for analysis. As a result, we have found four SPLs: BET-SPL [11], Mercurius [44], MobileMedia [44], and NotePad [41]. We manually confirmed that each selected SPL uses the component-based SPL guidelines [4]. We decided to discard Mercurius due to constraints in computing its code smells and metrics.

The selected SPLs are proposed by different development teams and belong to different domains. Table 1 presents general data per SPL. The 1st column names the SPL. The remainder columns present: number of lines of code (LOC); number of methods (Methods); and number of SPL components (Components). This study considers a class as a component, due to the limited size of the SPLs available for study. We collected all data via Understand tool [38], which includes inner classes and methods in the final count.

Step 2: Select developers and apply the survey. We surveyed the developers that contributed to the implementation of selected SPLs. Our goal was to assess the developers' perceptions about critical and non-critical components of the SPLs. Another objective was reaching a certain reliability to the reference lists of critical components derived per SPL. We designed two surveys: (i) *Participant Characterization Form* aimed at collecting background data of each SPL developer. For this purpose, we asked developers about their highest education level and skills in component-based SPL development. (ii) *Critical Component Elicitation Form* aimed at collecting the SPL developers' perceptions of a component in general and a critical component. Additionally, we asked for examples of critical components in the SPL that developers have contributed with. We hosted our surveys at LimeSurvey [23].

Table 1: General Data of the Target Component-based SPLs

SPL	LOC	Methods	Components
BET-SPL	13,318	1,576	177
MobileMedia	14,250	1,333	332
NotePad	2,626	128	15
Average	**10,064**	**1,0123**	**174,66**
All	**30,194**	**3,037**	**524**

Step 3: Define metrics and critical components based on developers' perception. To analyze the SPL developers' perceptions, we transcribed all survey answers into a plain text. We then read and summarized answers by discarding text parts that are irrelevant to our analysis, such as out-of-context discussions about SPL and components. After, we highlighted in the text all parts, including nouns and expressions, that make explicit their perception of either a component or a critical component. Finally, we grouped the survey answers (with eventual overlaps) that share similar perceptions of either a component or a critical component, strictly based on the highlighted parts. We proceeded with this computation in pair and discussed divergences on the relevance of answers and text parts to our study.

Based on the analysis of the survey, we selected five coupling metrics from well-known metric catalogs [8, 29, 33] that map the concepts reported by the developers. Table 2 describes the five selected metrics, namely *Coupling between Objects* (CBO) [8], *Fan-in* (FANIN) [8, 29], *Fan-out* (FANOUT) [8, 29], *Depth in Inheritance Tree* (DIT) [27, 29], and *Number of Children* (NOC) [27, 29].

We also built a reference list of critical components per SPL, based on definitions of a critical component provided by the developers of each collected SPL, via online report, and their experience with component-based SPL. This reference list was built in pair

[2] https://www.tiobe.com/tiobe-index/

Table 2: Metrics to Characterize Critical Components

Metric	Description
CBO	Number of components coupled to one specific component
FANIN	Number of dependencies from various components to one
FANOUT	Number of dependencies from one components to others
DIT	Maximum depth of a component in the inheritance tree
NOC	Number of children components of a component

to avoid biases. Thus, two authors individually built the reference list for each SPL. After, we compared the results and discussed any disagreements until reaching a consensus. The references list per SPL is available on our companion website [46].

Step 4: Compute metrics and thresholds per SPL. To computed software metrics and support critical component detection, we used the Understand tool. We also analyzed the distribution of each metrics to select a threshold derivation method that best fits our metric distributions. Figure 3 shows that the five metrics follow a heavy-tailed distribution [17], in which the higher the metric value, the more critical is the metric.

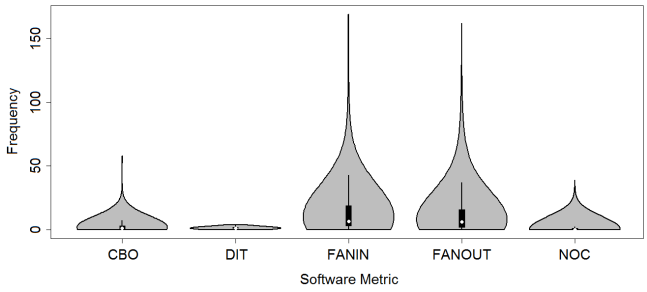

Figure 3: Distribution of Frequency per Software Metric

To derive thresholds that characterize critical metric values, we used a quantile-based method from the literature [47]. It defines five value intervals, including high (values > 90%) and very high (values > 95%). We call *Critical* the interval [90%, 95%], which has the critical SPL component, and *Strongly Critical* the interval [95%, 100%], which has the most critical components.

Step 5: Detect critical components per SPL. We applied our metric-based strategy [45] to detect critical components. This strategy combines selected metrics of Table 2 and detects two categories of components: *critical component*, for which the metric values are higher than at least 90% of all metric values in the analyzed SPLs; and *strongly critical*, for which the metric values are higher than 95% of all metric values in the analyzed SPLs. We present the metric-based strategy as follows: (i) for critical components (90%): *(FANIN > 30) OR [(FANOUT > 34) AND (DIT > 3 OR NOC > 1)] OR (CBO > 6)*; and (ii) strongly critical components (95%): *(FANIN > 40) OR [(FANOUT > 46.85) AND (DIT > 3 OR NOC > 1)] OR (CBO > 8)*.

We assume that both FANIN and CBO explicitly capture coupling between critical components. Based on the literature of component-based SPL, a critical component is the one that has too many dependencies with other components, which we can measure with CBO > Threshold and FANIN > Threshold. However, from our viewpoint, FANOUT > Threshold may not suffice to detect critical components because, considering the SPL architecture, each component tends to use features provided by several other components. However, when a component is deeply located in the inheritance tree and has too many children components, such external dependencies indicated by FANOUT > Threshold may negatively impact on the SPL product configuration, which makes the critical components. Thus, FANOUT has to be combined with DIT > Threshold and NOC > Threshold for capturing critical components.

Step 6: Detect code smells and map them to component principles. The code smells were detected in the target SPLs using two tools [15], namely JDeodorant and JSpIRIT. Table 3 presents the nine code smell types detected. The first and second columns name and describe each type with definitions adapted from Fowler's book [20] to components. We selected code smells rather than the architectural smells investigated by previous work [30] because we assess whether the negative effects on SPL product composition go beyond architectural decisions, i.e., source code-level decisions.

To map code smell types to principles they violate, we analyzed the definitions of code smell types based on two well-known catalogs [20, 27]. After, we discuss different scenarios of how each code smell type violates the three component principles. Finally, we mapped the code smell types per violated principle. We conducted this mapping in pair and discussed any divergences until reaching a consensus. The third column of Table 3 lists which component principles each smell type violates. This column relies on classification made in studies about smell type definitions [20, 27, 30, 50].

Step 7: Compute the smell-proneness of critical components. To measure the smell-proneness of critical components, for answering RQ_1, we computed the strength of the relationship between (non-)critical components and code smells via Fisher's exact test [18]. This test provided two values: a p-value that indicates the statistical relevance of our results, and the Odds Ratio [9], a coefficient that indicates the possibility of the presence or absence of a property (critical component) to be associated with the presence or absence of other property (code smell).

Step 8: Compute code smell types per violated principle and granularity. To compute the number of code smell types per violated principles (RQ_2), we designed a twofold procedure. First, we computed the number of instances of each code smell type that affects critical and strongly critical components per SPL. Second, we analyzed the code smell types per violated principles (see Table 3). Also, we compute the ranking of the more recurrent code smell types, and analyzed the correlation according to the criticality of components via Spearman's rank correlation coefficient [42]. With respect to code smell granularity, we computed the frequency of code smells that affect critical components per granularity, for answering RQ_3, following well-known classifications [31, 35, 50].

Step 9: Analyze the collected data. We first assess the SPL developers' perceptions about critical and non-critical components (Section 5) and then address our RQs (Section 6).

5 VALIDATING DEVELOPERS' PERCEPTIONS

To investigate the smell-proneness of critical components, we assess the SPL developers' perceptions about critical and non-critical components. We aim to understand whether developers' perception matches the literature assumptions. For that, we build reliable reference lists of critical components for analysis (Step 3 in Section 4.2). We conduct this validation to support our decision to characterize

Table 3: Code Smells Analyzed in Our Study and the Component Principles they Violate

Code Smell Type	Description	Violated Principles*
Data Class [20]	A component that encapsulates data without complex responsibilities. It mostly has attributes, and `get()` and `set()` methods	I
Dispersed Coupling [27]	A method that depends just a little on several methods provided by components other than the component it belongs to	I
Feature Envy [20]	Part of the code in a component uses more resources from another component rather than from the host component	I, II
Intensive Coupling [27]	A component that depends a lot on a few methods of other components	I
Large Class [20]	A component that implements too many functionalities. It tend to be very large and hard to read and understand	I, III
Long Method [20]	A method that implements several operations of the component. In general, it is has too many lines of code	I, III
Refused Bequest [20]	A child component, in the component inheritance tree, that generally does not use the resources provided by its parent component	I
Shotgun Surgery [20]	A component that, when changes, requires changes in many other components	I, III
Type Checking [20]	A switch structure with too many cases (a.k.a Switch Statement)	III

*Component Principles: (I) feature encapsulation, (II) data access restriction, and (III) replaceability

critical components via a coupling metric-based strategy. We did not find previous work aimed at understanding the developers' perception of a critical component in the context of component-based SPL. Previous studies [4, 24, 45] limitedly provide informal definitions without empirical evidence that confirms their alignments with the SPL developer perception, except for a previous [45].

Participant distribution and background. Table 4 present the response rate for the survey per SPL. The first column lists the SPLs. The second column shows the number of developers per SPL. The third and fourth columns show, respectively, the absolute number and the rate of responses per SPL. In total, 5 out of 10 developers fully completed the survey (response rate equals 50%).

Table 4: Response Rate for the Survey per SPL

SPL	Developers	Responses	Response Rate
BET-SPL	2	1	50%
NotePad	3	2	66.6%
MobileMedia	5	2	40%
All	**10**	**5**	**50%**

We discuss the background level of the surveyed SPL developers as follows. The majority (60%) of the survey participants have a medium skill level in component-based SPL development and a high experience with component-based SPL. Therefore, the SPL developers' perceptions of a component and a critical component are relevant to our study. Moreover, most SPL developers have a high education level. 83.3% hold a PhD degree and 16.66% hold a MSc degree, and their experience in component-based SPL development ranges from 3 to 30 years (average equals 10.8 years). About the task that developers have engaged during component-based SPL development, we have observed the following all developers (100%) have participated in architectural design tasks. Also, the majority of developers (80%) have engaged in SPL implementation from scratch and in extending an existing SPL (60%). Finally, almost a half of developers (40%) have maintained an SPL.

Perceptions of a software component. Next, we present quotes of responses provided by the surveyed developers about how they perceive a component.

- P1 (BET-SPL): "A set of *encapsulated elements* that one can access via an *interface*, preferably compatible in isolation [...]. It also should completely *realize a single feature*."
- P2 (NotePad): "Component is a *composition unit* with a contractually specified *interface* and explicit *dependencies*. A component can be deployed independently and is subject to *composition* by third-parties."
- P3 (MobileMedia): "It has three fundamental properties: (i) it *encapsulates a functionality* of the system and exposes it through services in an explicitly-defined *interface*, (ii) it explicitly defines its dependencies to other components via a required *interface*, and (iii) given the two previous properties, it can be compiled independently from the rest of the system."

These responses suggest that the surveyed SPL developers have a common perception of a component. They agreed that component is a composition unit (as stated by P1, P2, and P3) that realizes a specific feature of the software system (P1 and P3) and provides features to others via a well-defined interface (P1, P2, and P3). We then conclude that their perception of a component matches the definition of component reinforced by the literature [4, 10, 21, 24, 26]. Their perception also supports our decision for considering a class as a SPL component in this study (Step 1 in Section 4.2).

Perceptions of a critical component. We sample the SPL developers' responses regarding their perceptions as follows.

- P1 (BET-SPL): "This term might have multiple meanings. For instance, it might relate to features that are critically relevant to a *specific SPL product composition*, i.e., a component that has a special feature in a specific product composition."
- P2 (NotePad): "Thinking of linked entities in a whole system, a critical component is likely to be that one which *might affect the other components* the most, as *it holds important core elements* from which the *other components depend upon*."
- P3: (MobileMedia): "This is a component that *implements a feature that is often used* (via its provided interface) by other components in the system. In the context of a SPL, a critical component is also the one that takes part in most (or all) *possible SPL compositions*."

Based on the aforementioned responses, we can observe that the SPL developer perceptions of a critical component converge to some extent. Critical components implement key features of the SPL (as mentioned by P2 and P3), are essential to the SPL product composition (P1 and P3), and are highly coupled to other SPL components due to dependencies (P2 and P3). With respect to coupling and its relationship with critical components, their responses support our decision to use the selected strategy to characterize critical components per SPL, which relies on traditional coupling metrics (as detailed in Step 3 of Section 4.2)

6 RESULTS AND DISCUSSION

Section 6.1 overviews the data distribution regarding the smelliness of the three component-based SPLs analyzed. Section 6.2 discusses

the smell-proneness of critical components **(RQ_1)**. Section 6.3 discusses as code smells that affect critical components, also violate one or more component principles **(RQ_2)**. Finally, Section 6.4 characterizes the type and granularity of code smells that often affect critical components in the component-based SPLs **(RQ_3)**.

6.1 Data Distribution

Table 5 presents the density of code smells that affect each SPL. The first column lists the nine types of code smells. We group the types by granularity (also discussed in Section 6.4): the first four types are intra-component smells; the last five ones are inter-component smells. The second, third, and fourth columns of Table 5 present the density of each smell type per SPL. The fifth column represents density. We observe that, although BET-SPL and MobileMedia have quite similar LOC and number of methods (see Table 1), the first SPL has around two times more code smells.

Table 5: Number of Code Smells per SPL

Code Smell	BET-SPL	MobileMedia	NotePad	All
Data Class	4	0	0	**4**
Large Class	15	17	4	**36**
Long Method	44	30	8	**82**
Type Checking	0	1	0	**1**
Dispersed Coupling	23	14	6	**43**
Feature Envy	73	51	11	**135**
Intensive Coupling	3	0	3	**6**
Refused Parent Bequest	34	2	0	**36**
Shotgun Surgery	18	6	1	**25**
All	**214**	**121**	**33**	**368**

Results of Table 5 reveal some interesting findings. The top-five code smell types with highest density are, in decreasing order, *Feature Envy*, *Long Method*, and *Dispersed Coupling*, *Large Class*, and *Refused Parent Bequest*. Except for *Dispersed Coupling* and *Refused Parent Bequest*, these smell types violate at least two component principles. For instance, *Large Class* and *Long Method* violate both feature encapsulation and component replaceability. Conversely, the remainder types mostly violate a single principle. That is the case of *Type Checking*, which violates replaceability, once several branches make the component very complex.

Figure 4 summarizes Table 5 in terms of the density of each type of code smell throughout the component-based SPLs. In general, we observe that five out of the nine types of code smell affect at least one component of each SPL. These smells are *Dispersed Coupling*, *Feature Envy*, *Large Class*, *Long Method*, and *Shotgun Surgery*. It suggests that, depending on the decomposition of SPL features into components adopted by the component-based SPL, different types of code smell tend to affect the SPL components.

6.2 Smell-Proneness of Critical Components

Table 6 presents the contingency table used to compute the **smell-proneness** of critical components via Fisher's test. The first column lists the component type: critical or non-critical. The other columns present the number of smelly and non-smelly components considering *critical* and *strongly critical* components. This table allows us to draw some interesting conclusions. First, by comparing the second and third columns, we observe that critical components tend to be more smelly than non-critical components, by taking into consideration *critical components*. That is, when considering our less strict category of critical components, code smells affect the critical components rather than the non-critical ones. A similar observation applies to the fourth and fifth columns, regarding *strongly critical components*. However, to support these observations, we have computed the Fisher's test, by considering confidence internal of 99% (p-value < 0.01), as follows.

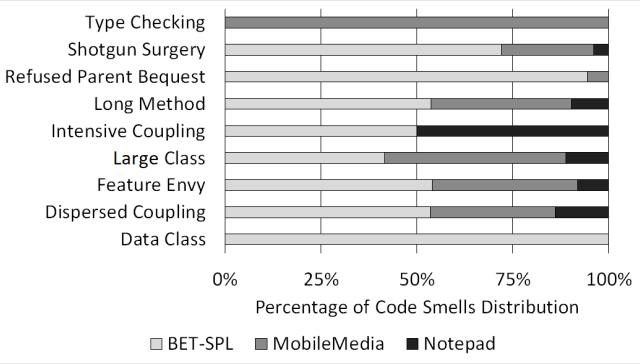

Figure 4: Code Smell Type Distribution throughout the SPLs

Table 6: Contingency Table for Fisher's Test

Criticality vs. Smelliness	Critical (90%)		Strongly Critical (95%)	
	Smelly	Non-Smelly	Smelly	Non-Smelly
Critical	64	16	33	11
Non-critical	80	364	110	370

Table 7 presents the results for the Fisher's test. The first column lists each interval of critical component, i.e., *critical components* and *strongly critical components*. The last two columns present the p-value computed via Fisher's test and the results for Odds Ratio.

Table 7: Fisher's Test Results

Fisher's Test Results	p-value	Odds Ratio
Critical (90%)	<0.001	18.05665
Strongly Critical (95%)	<0.001	10.03461

Note that, for the two intervals, we obtained statistical significance (p-value < 0.001). Regarding Odds Ratio, we have a value around 18 for *critical* and 10 for *strongly critical components*. That means that a strongly critical component is at least ten times more prone to be smelly than a non-critical component. A similar observation is valid for critical components, which are 18 times more prone to be smelly than non-critical ones.

Answering RQ_1: ***Are critical SPL components more smell-prone than non-critical SPL components?*** Critical components tend to be more smelly than non-critical components in component-based SPLs. *Strongly critical components* are ten times more prone to be smelly than a non-critical component. *Critical components* are 18 times more smelly-prone. Besides that, our data suggest that the developers need more attention when maintaining and evolving the SPL components. This problem is even more critical since the smelliness of SPL components can eventually propagate to several SPL products.

6.3 Violations of Component Principles

Table 8 presents the density and percentage of types of code smells that affect the *critical components* (90%) and *strongly critical components* (95%) per SPL. Our goal is understanding if the most frequent code smell types are the ones that **violate various component principles**. The first column lists each type of code smell and the other columns the density of code smells in each SPL. Overall, we observe that the density of code smells that affect *Strongly Critical* components is a half of the density for *Critical* components. However, this observation is not valid for NotePad, in which the density is equal for both intervals.

Table 8: Number of Code Smells in Critical Components

Code Smell	BET-SPL		MobileMedia		NotePad		All	
	90%	95%	90%	95%	90%	95%	90%	95%
Data Class	4	4	0	0	0	0	4	4
Large Class	8	3	12	4	4	4	29	11
Long Method	16	7	19	13	8	8	43	28
Type Checking	0	0	1	1	0	0	1	1
Dispersed Coupling	13	4	8	4	6	6	27	14
Feature Envy	26	10	33	22	9	9	68	41
Intensive Coupling	2	2	0	0	3	3	5	5
Refused Parent Bequest	9	1	2	1	0	0	11	2
Shotgun Surgery	14	12	1	0	1	1	16	13
All	**92**	**43**	**76**	**45**	**31**	**31**	**199**	**119**

The ranking of most frequent code smell types for critical component (90%), from the most frequent to the less frequent, is: *Feature Envy*, *Long Method*, *Dispersed Coupling*, *Large Class*, *Shotgun Surgery*, *Refused Bequest*, *Intensive Coupling*, *Data Class*, and *Type Checking*. The ranking of most frequent code smell types for strongly critical component (95%) is: *Feature Envy*, *Long Method*, *Dispersed Coupling*, *Shotgun Surgery*, *Large Class*, *Intensive Coupling*, *Data Class*, *Refused Bequest*, and *Type Checking*.

These results suggest that, regardless the criticality, four out of the top-five most frequent code smells types violate at least two component principles. It suggests that, even when developers are concerned on decomposing features into architectural components, code smells still manifest as recurring symptoms of poor feature decomposition. Thus, developers have to carefully design each critical component, since it is essential to assure a proper SPL product composition. Additionally, we compared both rankings through the Spearman's correlation coefficient [42]. The results equals 93.3% (p-value = 0.05) that implies a very strong correlation according to well-known guidelines [5, 37]. This result suggests that there is no statistically significant difference between the most frequent code smell types per criticality level.

Answering RQ$_2$: *Do the smell types affecting critical SPL components violate multiple component principles?* The most frequent types of code smells affecting critical components are *Feature Envy*, *Shotgun Surgery*, *Long Method*, *Dispersed Coupling*, and *Large Class*. They mostly violate two or more component principles, which implies a need for identifying and eliminating them whenever possible.

6.4 Smell Granularity in Critical Components

Inspired by the literature of code smells [16, 31, 34, 35, 50], we analyzed the **granularity of the most frequent types of code smells** that affect the component-based SPL. We considered two levels of granularity: (i) *Intra-component code smell*: a code smell that affects a single SPL component. This granularity encompasses the code smells with an inadequate relationship with the decomposition of SPL features into components, but with a poor design that locally affects a SPL component. In the context of our study, such code smells are *Data Class*, *Large Class*, *Long Method*, and *Type Checking*. (ii) *Inter-component code smell*: a code smell that affects multiple SPL components. This granularity encompasses the code smells closely related with the decomposition of SPL features into components, which clearly indicate poor component design. In our study, they are *Feature Envy*, *Dispersed Coupling*, *Intensive Coupling*, *Shotgun Surgery*, and *Refused Parent Bequest*.

Based on results of Table 8 and the granularity levels, we can see that most types of code smells (5 out of 9) that affect component-based SPLs are *inter-component code smells*. Thus, they affect multiple SPL components together. Also, the most frequent types of code smells (3 out of 5) are *inter-component code smells*, which suggests that critical components are often affected by poor feature decomposition. Still, some *intra-component code smells* often affect component-based SPLs, like *Long Method* and *Large Class*. Although they are not explicitly related to poor decomposition, they are symptoms of problems that SPL developers should be aware of. For instance, all methods in NotePad affected by *Long Method* are public methods. Thus, they may compose the component interface, which implies a poor SPL feature decomposition.

Answering RQ$_3$: *Do the smell types affecting critical SPL components also affect multiple SPL components together?* Despite the effort of developers in properly decomposing the SPL features into components, we can observe that poor decomposition often affects negatively the SPL product composition. Most code smells that affect component-based SPLs affect multiple SPL components together (*inter-component code smells*).

7 STUDY IMPLICATIONS

This section discusses how developers could improve the maintainability of critical components in component-based SPLs.

Identifying critical components is a "real deal". In this paper, we assess the *the smelliness of critical components* in component-based SPLs. The results of our study suggest that critical components are more smell-prone than non-critical components. We also found that code smells affecting critical components violate at least two out of the three component principles. Finally, our results point out that the most frequent code smells affect multiple SPL components together. Overall, our study sheds light on the topic of critical components. In fact, the results suggest that detecting critical components might aggregate to the existing techniques for prioritizing code smells for elimination. Thus, we made both SPL researchers and developers aware of the smelliness of critical components, as well as the need of improving their maintainability.

Elimination of code smells that affect multiple critical components. In this paper, we compute critical components based

on two categories: critical components (90%) and strongly critical components (95%). The results suggest that, regarding the category, the most frequent code smell types are the ones that violate various component principles and affect multiple SPL components together. We expect a certain generalization of our findings to code smell types that we did not investigate but share a set of violated component principles and granularity. That might be the case of the Long Refinement Chain [14] in the context of feature-oriented SPLs [3]. This code smell can be considered an inter-class code smell because it affects multiple components together. Thus, the same reasoning applied to inter-class smells might apply to this smell type.

Refactoring critical components for better SPL modularity. Refactoring critical components aiming at eliminating code smells might not reduce their criticality. That is because detecting critical components usually relies on traditional coupling metrics, which might not be sensitive to the changes caused by each refactoring operation on the code structure [6, 7]. Moreover, refactoring code might generate more critical components, since the essential features realized by a single critical component are distributed to multiple components, which become critical because of that. However, refactoring might have a positive effect on the maintainability of critical components. We illustrate this scenario as follows.

Figure 5 presents the MainWindow component of NotePad that realizes multiple SPL features (see Figure 1). This component is affected by a Large Class, which implies that the component is too large and complex. In order to eliminate this code smell, developers could apply multiple Extract Subclasses. As a result, a new subclass is created to realize each feature in isolation. Without applying such refactoring, adding a new SPL feature would require increasing both the length and complexity of MainWindow. It implies worsening the component maintainability. On the other hand, by refactoring this component, developers adding a new feature would just have to implement a new subclass that realizes the added feature without harming the maintainability of any SPL component.

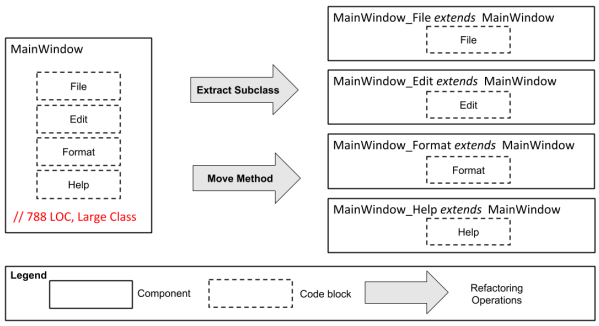

Figure 5: Improving the MainWindow Structural

8 THREATS TO VALIDITY

Construct and Internal Validity. This study analyzes a limited set of code smell types [20]. Thus, our findings might be biased by these types. To minimize possible threats to validity, we selected code smell types that: are commonly investigated by previous work [15, 34, 35, 50]; violate at least one out of the three component principles [10, 21, 26]; and have different levels of granularity. To classify code smell types by the violated component principles, we minimize subjectivity by conducting it in pairs and discussing divergences based on the related literature [20, 27]. Finally, regarding the detection of critical components and code smells, we conducted all detection in pairs and compared the results with the literature. This comparison aimed at avoiding problems with missing data. Each detection was supported by either well-known detection tools or accurate metric-based strategies. Finally, concerning the possible bias between the metrics used to detect critical components and the existence of code smells, we have selected the metrics based on the SPL developers' perceptions collected via survey.

Conclusion and External Validity. Regarding the study about the SPL developers' perception of critical and non-critical components, we followed well-defined procedures. In fact, we carefully transcribed and summarized all developers' responses. We mitigated possible biases in the analysis by conducting it in pairs. To compute the statistical significance and strength of the relationship between critical components and code smells, we relied on two reliable techniques: the Fisher's exact test [18] and the Odds Ratio [9]. Both techniques have been successfully used by previous work [16, 34]. With respect to the discussion about the most frequent code smell types for critical and strongly critical components, we have applied a well-known rank correlation coefficient called Spearman's correlation [42]. Finally, regarding the generalization of findings, we analyzed only three component-based SPLs. We mitigate possible threats to validity by selecting SPLs with varied sizes, from different domains, and implemented in a popular language – Java in this case. We expect that our results are extensible to other SPL product composition techniques than component-based SPL, but further investigation is required.

9 CONCLUSION AND FUTURE WORK

This paper presents a study about the smelliness of critical components in component-based SPLs. We rely on data from three Java component-based SPLs. We also rely on a reliable reference list of critical components built from the SPL developers' perceptions of a critical component. For the analysis, we detected critical components through a light-weighted metric-based strategy that we have previously proposed [45]. After, we computed nine code smell types that, by definition, violate one or more basic component principles: feature encapsulation [26], data access restriction [21], and replaceability [10]. Finally, we conduct quantitative analysis to understand how smelly are the SPL critical components based on smell-proneness, code smell type, and code smell granularity.

Our findings suggest that critical components are at least 10 times more smell-prone than non-critical components in component-based SPLs. Moreover, our findings point out that the most frequent code smell types affecting the critical components violate at least two out of the three component principles. This result suggests a significant negative effect of those code smells on the SPL product composition. Finally, we have found that the most frequent code smell types affecting critical components also affect multiple SPL components together. Overall, our study provides useful hints for techniques that prioritize code smells for elimination in component-based SPLs. As future work, we aim to: assess other techniques

for SPL product composition; and assess the relationship between component variability and criticality.

ACKNOWLEDGMENTS

This work was partially funded by CNPq (434969/2018-4, 408356/2018-9, 141285/2019-2), CAPES/Procad (175956), and FAPERJ (200773/2019, 010002285/2019, 202073/2020).

REFERENCES

[1] Ramon Abílio, Juliana Padilha, Eduardo Figueiredo, and Heitor Costa. 2015. Detecting Code Smells in Software Product Lines – An Exploratory Study. In *12th Int. Conference on Information Technology - New Generations (ITNG)*. 433–438.
[2] Hugo Andrade, Eduardo Almeida, and Ivica Crnkovic. 2014. Architectural bad smells in software product lines. In *11th IEEE/IFIP Conference on Software Architecture (WICSA)*. 12:1–12:6.
[3] Sven Apel, Don Batory, Christian Kastner, and Gunter Saake. 2013. *Feature-oriented software product lines*. Springer.
[4] Colin Atkinson, Joachim Bayer, and Dirk Muthig. 2000. Component-based product line development. In *1st Int. Software Product Line Conference (SPLC)*. 289–309.
[5] Thorsten Berger and Jianmei Guo. 2014. Towards system analysis with variability model metrics. In *8th Int. Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, 23.
[6] Diego Cedrim, Alessandro Garcia, Melina Mongiovi, Rohit Gheyi, Leonardo Sousa, Rafael de Mello, Baldoino Fonseca, Márcio Ribeiro, and Alexander Chávez. 2017. Understanding the impact of refactoring on smells: a longitudinal study of 23 software projects. In *11th Joint Meeting on Foundations of Software Engineering (FSE)*. 465–475.
[7] Alexander Chávez, Isabella Ferreira, Eduardo Fernandes, Diego Cedrim, and Alessandro Garcia. 2017. How does refactoring affect internal quality attributes?: A multi-project study. In *31st Brazilian Symposium on Software Engineering (SBES)*. 74–83.
[8] Shyam Chidamber and Chris Kemerer. 1994. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng. (TSE)* 20, 6 (1994), 476–493.
[9] Jerome Cornfield. 1951. A method of estimating comparative rates from clinical data. *J. Natl. Cancer Inst.* 11, 6 (1951), 1269–1275.
[10] Ivica Crnkovic and Magnus Larsson. 2002. *Building reliable component-based software systems*. Artech House.
[11] Paula Donegan and Paulo Masiero. 2007. Design issues in a component-based software product line. In *1st Brazilian Symposium on Software Components, Architectures and Reuse (SBCARS)*. 3–16.
[12] Reiner Dumke and Achim Winkler. 1997. Managing the component-based software engineering with metrics. In *5th Int. Symposium on Assessment of Software Tools and Technologies (SAST)*. 104–110.
[13] Wolfgang Emmerich and Nima Kaveh. 2001. Component technologies: Java beans, COM, CORBA, RMI, EJB and the CORBA component model. In *ACM SIGSOFT Softw. Eng. N.*, Vol. 26. ACM, 311–312.
[14] Wolfram Fenske and Sandro Schulze. 2015. Code smells revisited: A variability perspective. In *9th Int. Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*. 3.
[15] Eduardo Fernandes, Johnatan Oliveira, Gustavo Vale, Thanis Paiva, and Eduardo Figueiredo. 2016. A review-based comparative study of bad smell detection tools. In *20th Int. Conference on Evaluation and Assessment in Software Engineering (EASE)*. 18:1–18:12.
[16] Eduardo Fernandes, Gustavo Vale, Leonardo Sousa, Eduardo Figueiredo, Alessandro Garcia, and Jaejoon Lee. 2017. No code anomaly is an island. In *16th Int. Conference on Software Reuse (ICSR)*. 48–64.
[17] Tarcısio Filó, Mariza Bigonha, and K Ferreira. 2015. A catalogue of thresholds for object-oriented software metrics. *1st International Conference on Advances and Trends in Software Engineering (SOFTENG)* (2015), 48–55.
[18] Ronald Fisher. 1922. On the interpretation of χ 2 from contingency tables, and the calculation of P. *J. R. Stat. Soc.* 85, 1 (1922), 87–94.
[19] Francesca Arcelli Fontana, Pietro Braione, and Marco Zanoni. 2012. Automatic detection of bad smells in code: An experimental assessment. *J. Obj. Tech. (JOT)* 11, 2 (2012), 5–1.
[20] Martin Fowler. 1999. *Refactoring*. Addison-Wesley Professional.
[21] William Frakes and Kyo Kang. 2005. Software reuse research. *IEEE Trans. Softw. Eng. (TSE)* 31, 7 (2005), 529–536.
[22] Nasib Gill. 2006. Importance of software component characterization for better software reusability. *ACM SIGSOFT Softw. Eng. N.* 31, 1 (2006), 1–3.
[23] LimeSurvey GmbH. 2021. *LimeSurvey*. Available at: https://www.limesurvey.org/.
[24] Jin Her, Ji Kim, Sang Oh, Sung Rhew, and Soo Kim. 2007. A framework for evaluating reusability of core asset in product line engineering. *Info. Softw. Tech. (IST)* 49, 7 (2007), 740 – 760.
[25] Marcus Kessel and Colin Atkinson. 2015. Ranking software components for pragmatic reuse. In *6th Int. Workshop on Emerging Trends in Software Metrics (WETSoM)*. 63–66.
[26] Charles Krueger. 2006. New methods in software product line practice. *Comm. ACM* 49, 12 (2006), 37–40.
[27] Michele Lanza and Radu Marinescu. 2006. *Object-oriented metrics in practice*. Springer Science & Business Media.
[28] Luan Lima, Anderson Uchôa, Carla Bezerra, Emanuel Coutinho, and Lincoln Rocha. 2020. Visualizing the Maintainability of Feature Models in SPLs. In *Anais do VIII Workshop de Visualização, Evolução e Manutenção de Software*. 1–8.
[29] Mark Lorenz and Jeff Kidd. 1994. *Object-oriented software metrics*. Prentice-Hall.
[30] Isela Macia, Joshua Garcia, Daniel Popescu, Alessandro Garcia, Nenad Medvidovic, and Arndt von Staa. 2012. Are automatically-detected code anomalies relevant to architectural modularity?. In *11th Annual Int. Conference on Aspect-oriented Software Development (AOSD)*. 167–178.
[31] Mika Mäntylä and Casper Lassenius. 2006. Subjective evaluation of software evolvability using code smells. *Emp. Softw. Eng. (ESE)* 11, 3 (2006), 395–431.
[32] Júlio Martins, Carla Ilane Moreira Bezerra, and Anderson Uchôa. 2019. Analyzing the Impact of Inter-smell Relations on Software Maintainability: An Empirical Study with Software Product Lines. In *XV Brazilian Symposium on Information Systems (SBSI)*. 1–8.
[33] Thomas McCabe. 1976. A complexity measure. *IEEE Trans. Softw. Eng. (TSE)* 4 (1976), 308–320.
[34] Willian Oizumi, Alessandro Garcia, Leonardo Sousa, Bruno Cafeo, and Yixue Zhao. 2016. Code anomalies flock together. In *38th Int. Conference on Software Engineering (ICSE)*. 440–451.
[35] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Andrea De Lucia. 2014. Do they really smell bad?. In *30th Int. Conference on Software Maintenance and Evolution (ICSME)*. 101–110.
[36] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. 2013. Detecting bad smells in source code using change history information. In *28th Int. Conference on Automated Software Engineering (ASE)*. 268–278.
[37] Neil J Salkind and Terese Rainwater. 2003. *Exploring research*. Prentice Hall Upper Saddle River, NJ.
[38] Inc Scientific Toolworks. 2021. *Understand Tool*. Available at: https://scitools.com/.
[39] David Sharp. 1998. Reducing avionics software cost through component based product line development. In *17th Digital Avionics Systems Conference (DASC)*. G32–1.
[40] M Silva, P Guerra, and Cecília Rubira. 2003. A java component model for evolving software systems. In *18th Int. Conference on Automated Software Engineering (ASE)*. 327–330.
[41] Larissa Soares, Ivan Machado, and Eduardo Almeida. 2015. Non-functional properties in software product lines. In *9th Int. Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*. 67.
[42] Charles Spearman. 1904. The proof and measurement of association between two things. *American Journal of Psychology* 15, 1 (1904), 72–101.
[43] Christian Tischer, Andreas Muller, Markus Ketterer, and Lars Geyer. 2007. Why does it take that long?. In *11th Int. Software Product Line Conference (SPLC)*. 269–274.
[44] Leonardo Tizzei, Marcelo Dias, Cecília Rubira, Alessandro Garcia, and Jaejoon Lee. 2011. Components meet aspects. *Info. Softw. Tech. (IST)* 53, 2 (2011), 121–136.
[45] Anderson Uchôa, Eduardo Fernandes, Ana Carla Bibiano, and Alessandro Garcia. 2017. Do Coupling Metrics Help Characterize Critical Components in Component-based SPL? An Empirical Study. In *5th Workshop on Software Visualization, Evolution and Maintenance (VEM)*. 36–43.
[46] Anderson Uchôa, Wesley K. G. Assunção, and Alessandro Garcia. 2021. Research companion website. https://anderson-uchoa.github.io/SBCARS2021/
[47] Gustavo Vale and Eduardo Figueiredo. 2015. A method to derive metric thresholds for software product lines. In *29th Brazilian Symposium on Software Engineering (SBES)*. 110–119.
[48] Gustavo Vale, Eduardo Figueiredo, Ramon Abílio, and Heitor Costa. 2014. Bad Smells in Software Product Lines: A Systematic Review. In *8th Brazilian Symposium on Software Components, Architectures and Reuse (SBCARS)*. 84–94.
[49] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. 2012. *Experimentation in software engineering*. Springer Science & Business Media.
[50] Aiko Yamashita and Leon Moonen. 2013. Do developers care about code smells?. In *20th Working Conference on Reverse Engineering (WCRE)*. 242–251.