Bad Smells in Software Product Lines: A Systematic Review

Gustavo Vale, Eduardo Figueiredo
Departamento de Ciência da Computação - Universidade Federal de Minas Gerais
{gustavovale, figueiredo}@dcc.ufmg.br

Ramon Abílio
Diretoria de Gestão da Tecnologia da Informação - Universidade Federal de Lavras
ramon.abilio@dgti.ufla.br

Heitor Costa
Departamento de Ciência da Computação - Universidade Federal de Lavras
heitor@dcc.ufla.br

***Abstract* — Software product line (SPL) is a set of software systems that share a common, managed set of features satisfying the specific needs of a particular market segment. Bad smells are symptoms that something may be wrong in system design. Bad smells in SPL are a relative new topic and need to be explored. This paper performed a Systematic Literature Review (SLR) to find and classify published work about bad smells in SPLs and their respective refactoring methods. Based on 18 relevant papers found in the SLR, we identified 70 bad smells and 95 refactoring methods related to them. The main contribution of this paper is a catalogue of bad smells and refactoring methods related to SPL.**

Keyword* — *Bad Smells; Software Product Lines; Refactoring

I. Introduction

Software Product Line (SPL) is a set of software systems that share a common, managed set of features satisfying the specific needs of a particular market segment [28]. The systematic and large-reuse adopted in SPL aims to reduce time-to-market and improve software quality [27]. The software products derived from an SPL share common features and differ themselves by their other features [27]. A feature represents an increment in functionality or a system property relevant to some stakeholders [17]. The possible combinations of features to build a product constitute the SPL variability [32] represented in a feature model [16]. A feature model is a formalism to capture and represent the commonalities and variabilities among the products in an SPL [7].

To develop an SPL, we can use different approaches, such as annotative and compositional [6]. For these approaches, we have several techniques, such as preprocessors, virtual separation of concerns, aspect-oriented programming [18], and feature-oriented programming [9]. Those approaches and techniques have been proposed to improve the separation of concerns (or features) and the software quality.

In spite of that, undesired properties may be present in the code or in all related artifacts, such as feature models [5]. A definition of 'variability smells' was proposed [5] by extending the definition of 'bad smells' [13] to address the SPL mechanisms and artifacts. Bad smells are metaphors to describe software patterns generally associated with bad design or bad programming practices [31]. We identified on literature two types of bad smells related on SPLs: (i) Architectural Bad Smells (or Architectural Smells) and (ii) Code Bad Smells (or Code Smells).

Architectural Smells describe an indication of an underlying problem that occurs at higher level of a system abstraction than a Code Smell [4]. Architectural Smells are structural attributes that mainly affect lifecycle properties, such as understandability, testability, and reusability, but they can also affect quality properties, such as performance and reliability [4]. Architectural Smells may be caused, for example, by [14] (i) applying a design solution in an inappropriate context, (ii) mixing combinations of design abstractions that have undesirable emergent behaviors or (iii) applying design abstractions at the wrong level of granularity.

Code Smells describe a situation where there are hints that suggest a flaw in the source code [13]. Code smells aim to diagnose symptoms that may be indicative of something wrong in the system code [13] or an undesired source code property [5]. Along the years, different Code Smells have been defined and catalogued [33].

In fact, software engineers have studied Code and Architectural Smells [33] in different systems developed with different technologies, such as Object-Oriented Programming (OOP) [13] and Aspect-Oriented Programming (AOP) [12]. Apel and his colleagues presented 14 variability smells and 7 refactoring methods [5]. Besides, they mentioned that variability smell is a relative young topic and a catalogue of variability smells and refactoring methods is necessary.

In this context, we performed a Systematic Literature Review (SLR) to find and classify definitions of bad smell in the SPL context and their respective refactoring methods. As a result, we analyzed 18 studies, classified 70 bad smells, and listed 95 refactoring methods. The main contribution of this work is a catalogue of bad smells and refactoring methods related to SPL.

The rest of this work is organized as follows. Section 2 reports the Systematic Literature Review. Section 3 highlights the overview on the selected studies, presents the data extracted and shows discussion of the results. Section 4 discusses limitations of this work. Section 5 concludes this study and presents suggestions of future work.

II. Research Method

We conducted an SLR which is a well-defined and rigorous method to identify, evaluate, and interpret all relevant studies regarding a particular research question, topic area, and phenomenon of interest [19]. Its goal is to give a fair, credible, and unbiased evaluation of a research topic using a trustworthy, rigorous, and auditable method. A common reason for undertaking an SLR is to summarize existing evidences concerning a technology [19]. Therefore, an SLR was an appropriate research method for our investigation that aimed to (i) find bad smell definitions in the SPL context (ii) identify and classify those bad smells, and (iii) identify and list refactoring methods. For our SLR, we followed the general guidelines to perform SLRs [19].

A. Development of Review Protocol

Prior to conducting our SLR, we developed a review protocol. A pre-defined protocol aims to reduce researcher bias and increases the rigor and repeatability of the review. An SLR protocol specifies review plans and procedures by describing the details of strategies to perform the SLR. In particular, it defines the research questions, search strategy to identify the relevant literature, the inclusion criteria for selecting relevant studies, and the methodology for extracting and synthesizing information in order to address the research questions. Figure 1 depicts the process used in the SLR.

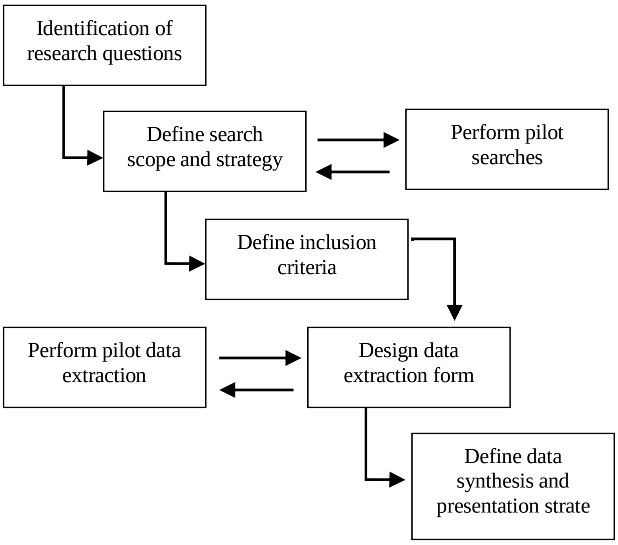

Figure 1. PROCESS TO PERFORM THE SLR ([1])

After identifying the research questions, we defined the search strategy and designed the search string used in eight electronic data sources. For publication space, we decided by eight electronic data sources. Table I presents *Name* (first column), *Link to Access* (second column), and *Export Method* (third column). *Export method* (forth column) refers to the way that the search results were downloaded from the electronic data sources. In summary, if an electronic data source allows exporting a set of references (papers) on Bibtex or EndNote format, it was considered an automatic export method. However, if each reference needs to be exported separated, it was considered a manual export method. IEEEXplore, Scopus, Elsevier Science Direct, El Compendex, and Web of Science provide automatic export methods. ACM Digital Library, DBLP Computer Science Bibliography, and Computer Science Bibliographies only provide manual export methods.

We used JabRef[1] software to manage the downloaded references. The search results from automatic download method were imported directly to JabRef and the references from manual download method were inserted each one manually in the JabRef. Hence, to reduce the work, we included only the important information to identify the paper. This way, to remove duplicates, we used the following criteria: if a reference is repeated in an automatic and manual export method then the reference from automatic source was kept. If a reference is repeated from automatic download methods then the most complete was kept.

TABLE I. ELECTRONIC DATA SOURCES

Name	Link to Access	Export Method
IEEEXplore	ieeexplore.ieee.org/	Automatic
Scopus	www.scopus.com	Automatic
Elsevier Science Direct	www.sciencedirect.com	Automatic
El Compendex	www.engineeringvillage.com	Automatic
Web of Science	www.isiknowledge.com	Automatic
ACM Digital Library	www.acm.org/	Manual
DBLP Computer Science Bibliography	www.informatik.unitrier.de/~ley/db/	Manual
Computer Science Bibliography	www.ira.uka.de/bibliography/index.html	Manual

We conducted pilot searches to test the quality of the search string. After the definitions of the search scope and strategy, we defined the inclusion criteria and decided on the data to be extracted. Such data should provide us with information to answer the research questions listed in the next subsection. We designed a preliminary data extraction form and tested it in a pilot involving five studies that we identified in the data extraction phase. As a final step in designing the protocol, we decided how to synthesize the extracted data and how to present the results.

B. Research Questions

Bad smells are well known in single and traditional software systems [33]. However, in the context of SPL, the research is in early phases and it is necessary to know the state of the art. Therefore, we elaborated the following research questions concerning bad smells in the context of SPL:

RQ1: What were the SPLs used in studies of bad smells?

RQ2: What are the bad smells already defined in the SPL context?

RQ3: Is the definition of a bad smell the same in the context of SPL and single software systems?

RQ4: What are the refactoring methods applied to the SPL context?

C. Search Strategy

For an SLR, a well-planned search strategy is important so that every relevant work can be expected to appear in the search results. For the search strategy two variables should be chosen: publication period and publication space. We decided to not limit the study by publication period (time). Therefore, all studies were included regardless their publication period.

[1] http://jabref.sourceforge.net/

The search string used was:

("bad smell" OR "code smell" OR "code anomaly" OR "variability anomaly" OR "variability smell") AND ("software product line" OR "software product-line" OR "software product family" OR "software product-family" OR "software family based" OR "software family-based" OR "software variability" OR "software mass customization" OR "software mass customization production lines" OR "software-intensive system")

This search string was constructed after performing a pilot searches. The electronic data sources provided different features, such as different field codes and syntax of search strings. Therefore, we constructed a semantically equivalent search string for each electronic data source. After perform the search, we obtained 165 references (Figure 2): 13 references (refs) from ACM Digital Library, 55 from Scopus, 19 from Science Directed, 31 from Computer Science Bibliographies, 3 from El Compendex, 18 from Web of Science, and 26 studies from IEEEXplore. DBLP did not return any result and was not represented in Figure 2.

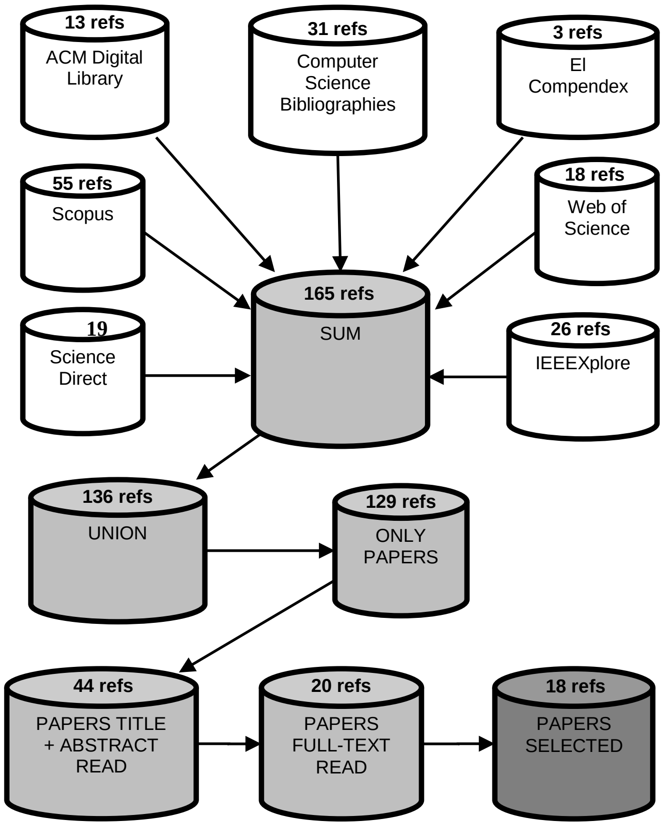

Figure 2. STEPS TO SELECT PAPERS

We cloned the *Sum* database to *Union* database and removed all duplicated references (136 refs remained). After this, the *Union* database was cloned to *Only Papers* database and the references to no papers were removed (129 refs remained). We read the title and the abstract of the papers (44 studies remained - *Paper Title + Abstract Read* database) and only papers that agreed with the inclusion criteria were selected to full-text read. The inclusion criteria are: (i) it must be on computer science area; (ii) it must be written in English; (iii) it must be completely in electronic form; and (iv) it must treat SPLs and bad smells. After full-text read (20 studies remained - *Paper Full-Text Read* database), data were extracted and the relevant papers were obtained (18 studies - *Papers Selected* database). The difference of *Papers full-text read* database and *Papers Selected* database is that the last has only papers that some data were extracted from them. In the Appendix, we provide the complete list of reviewed studies [A-R].

D. Data Extraction, Synthesis, and Aggregation

The selected primary studies were read in depth in order to extract data needed to answer the research questions. Two researchers read the selected papers in parallel. Data were extracted based on a detailed set of questions. Some of the fields of our data extraction form included: study ID, SPL used, SPL domain, study aim, studied characteristics, bad smells used, study findings, refactoring method, type/size of the system, technology and languages used, research method used, and type of subjects. We recorded the extracted information in a spreadsheet for subsequent analysis. We noted the lines and paragraphs of the paper where the information were obtained, for the case that when a researcher disagree with other and shows why the paper was chosen (resolve disagreements). This helped to increase our confidence that the extraction process was consistent and minimally biased.

During an SLR, extracted data should be synthesized in a manner suitable for answering the research questions [19]. For the reported SLR, we decided to perform descriptive synthesis of the extracted data and to present the results in tabular form. The analysis of the data revealed that each study has some contribution in the context of bad smells in SPLs and more than one piece of information can be obtained by one study: 12 studies showed previous bad smells; 5 proposed bad smells; in 3 studies, refactoring methods are presented; in 15 studies, SPLs are presented; and, finally, the definitions of bad, architectural and code smells were obtained in the analyses of the 18 reviewed studies to improve the knowledge in this topic.

III. RESULTS

This section presents an overview of the reviewed studies: analysis of the data extracted from the reviewed papers to answer the research questions. We anchored our presentation on the characteristics studied in the selected papers.

A. Overview of the Reviewed Studies

This section presents information related to method, type and setting of the 18 selected papers. Year-wise distribution of the papers revealed that over the past completed years (2007-2013). Only one paper was found on years 2007, 2008, and 2009. Two papers were found on years 2010 and 2011. Four and six papers were found on 2012 and 2013, respectively. One paper was found on 2014, but the search was until April 2014. Therefore, we can conclude that the interest of the community in investigating bad smells in the context of SPLs is an increasing (Figure 3).

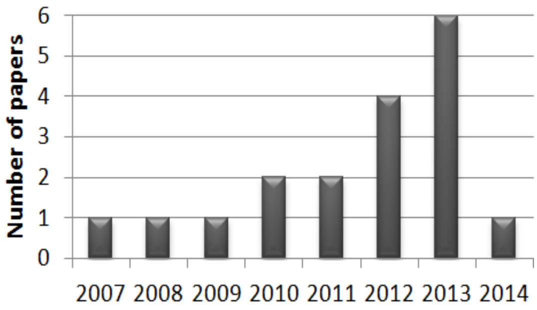

Figure 3. YEAR-WISE DISTRIBUTION OF THE PAPERS

The selected papers were published in software engineering conferences and journals (Figure 4): 4 papers in the International Conference on Aspect-Oriented Software Development (AOSD); 1 paper in ACM/IEEE International Conference on Software Engineering (ICSE); 1 paper in ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA); 2 papers in European Conference on Software Maintenance and Reengineering (CSMR); 2 papers in International Conference on Generative Programming and Component Engineering (GPCE) - together International Workshop on Feature-Oriented Software Development (FOSD); 1 paper in Information and Software Technology Journal; 1 paper in Software Product Line Conference (SPLC); 2 papers in Science of Computer Programming Journal; 1 paper in Working IEEE/IFIP Conference on Software Architecture (WICSA); 2 papers in Brazilian Symposium on Software Engineering (SBES); and, 1 paper in the Brazilian Symposium on Software Components, Architectures and Reuse (SBCARS). Those conferences and journals are known for publishing high quality software engineering papers.

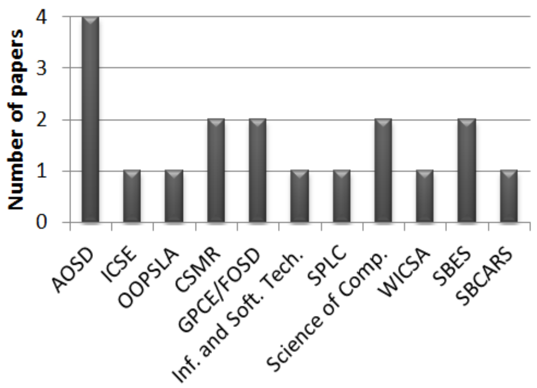

Figure 4. CONFERENCES AND JOURNALS OF THE SELECTED PAPERS

B. Reported Research Methods and Study Settings

A suitably designed and rigorously conducted study follows a well-defined research methodology to ensure the reliability and validity of the findings. It is expected that a study explicitly reports and justifies the used research methodology and its related logistics. Table II provides information on the type of research methods reported by the authors in the reviewed papers. We can note that 'Exploratory Study' research method is the dominant approaches used by researchers to explore the question topic. Out of 18 selected papers, two studies reported the use of 'Systematic Mapping or Systematic Review' and 'Case Study' methods, while only one study used 'Empirical Study' method and other used 'Controlled Experiment' method. However, five studies did not state their research method.

TABLE II. RESEARCH METHODS REPORTED

Method	Studies	Number
Exploratory Study	[A][B][C][F][H][K][P]	7
Systematic Mapping or Systematic Review	[D][R]	2
Case Study	[E][I]	2
Empirical Study	[L]	1
Controlled Experiment	[M]	1
Not mentioned	[G][J][N][O][Q]	5

An overview of the contexts and settings in which empirical evaluations are performed can reveal the level of empirical research practice in a discipline. However, it is difficult to delineate what constitutes the context or settings of an empirical study. As we observed, studies provided limited information regarding their experimental setup and, in most of the cases, they were not explicitly reported in the reviewed studies for this systematic review. Although, we encountered studies conducted in different settings, only one study was performed in an Industrial environment (Table III).

TABLE III. STUDY SETTINGS

Settings	Studies	Number
Industrial	[I]	1
Academic	Remaining reviewed studies	17

C. Research Questions

Analyzing the reviewed papers, we were able to answer the research questions:

RQ1: What were the SPLs used in studies of bad smells?

Using the data extraction form, we identified 16 SPLs (Table IV). We looked at the SPLs used in the reviewed studies. We extracted the name, size (Lines of Code - LOC), domain, language, and system type. However, these data were difficult to obtain, because some papers do not mentioned these information. To minimize, we used extra studies, i.e., looked for papers at internet to find identified SPLs to get the missing information, when possible. Therefore, items with "**" means that this information was not obtained in the reviewed studies and items with "*" means that data was not explicit in the paper. For example, it is not explicit in the reviewed study [E] that the SPL was developed in AspectJ. We inferred this information by the paper context.

Table IV shows that AspectualMedia (AM) SPL has ~4,000 LOC, belongs to Manipulate media on mobile devices domain, has versions developed in Java and AspectJ; is considered a toy system, and the [B] reviewed paper cited this SPL. The biggest SPL found was Danfoss Power Electronics (DPE) with ~2,000,000 LOC. We found systems developed with 7 different languages. Two different types of systems (Real and Toy). The same SPL could be found in a unique paper or in more than one, for instance the MobileMedia (MM) SPL was found in 8 different papers.

TABLE IV. SPLs USED IN THE REVIEWED STUDIES

Name	Size (~LOC)	Domain	Language	System type	Cited
Aspectual Media (AM)	4K**	Manipulate media on mobile devices	Java/AspectJ	Toy	[B]
ATM	Unmentioned	Bank	DeltaJ	Toy	[P]
Bali Product Line (BPL)	16K	Grammar tool	Java/AHEAD	Real	[Q]
Danfoss Power Electronics (DPE)	2M	Electronic power conversation	C/C++	Real	[I]
Expresion Product Line (EPL)	98**	Expression evaluation	AHEAD	Toy	[N]
Genuine Soccer SPL	Unmentioned	Mobile Game of soccer	AspectJ*	Toy	[E]
Graph Product Line (GPL)	1.8K	Graph and algorithm library	Java/AHEAD	Toy	[N][Q]
Graphical Calculator	Unmentioned	Mathematical calculations	rbFeature	Toy	[N]
JPL	48K	Unmentioned	Java/AHEAD	Unmentioned	[Q]
Media Shop	Unmentioned	E-commerce	AspectJ*	Toy	[E]
Mobile Media (MM)	4K**	Manipulate media on mobile devices and controls	Java/AspectJ	Toy	[A][B][F][G][H][K][L][M]
Notepad SPL	1.7K	Text editor	Java	Real	[C]
Prevaler Product Line (PPL)	2K	Database	Java/AHEAD	Real	[Q]
TankWar SPL	5K**	PC and Mobile Game	AHEAD	Toy	[J]
Undefined Name (unmentioned) - (UN)	1M	Control system firmware of electric drives	C/C++	Real	[I]
µCOS	15K	Operational System	C/C++	Real	[I]

RQ2: What are the bad smells already defined in the SPL context?

We identified 70 bad smells in the reviewed studies. This identification was a hard task because some studies used the same definition of a smell for both single systems and SPLs. We classified those bad smells in code smells (Table V), architectural smells (Table VI), and hybrid smells (Table VII).

Table V presents 49 code smells related to SPLs that is more common and more dependent of technology than architectural smells. In addition to the code smells name, Table V presents technology, language, SPL used in the study, what reviewed studies cited the code smell, and who proposed the code smell. For example, the *Proliferation of Variant Elements* code smell was proposed by [20] and cited by [I]. It was identified in three SPLs (*Danfoss Power Electronics*, *Undefined Name*, and *µCOS*) developed using OOP and C/C++. We noted that *Feature Envy* code smell was studied concerning OOP [13], as a method that seems more interested in a class other than the one it actually is in, and AOP [26], pointcuts could be defined in aspects and also in classes. If a single aspect uses a class-defined pointcut, it is interesting to move it from the class to the aspect that uses it.

TABLE V. CODE SMELLS IN CONTEXT OF SOFTWARE PRODUCT LINES

Code Smell	Technology	Language	SPL	Cited	Proposed
Proliferation of Variant Elements	OOP	C/C++	DPE; UN; µCOS	[I]	[20]
Variation Combinatorics	OOP	C/C++	DPE; UN; µCOS	[I]	[21]
Unintended Interdependencies of Variant Elements	OOP	C/C++	DPE; UN; µCOS	[I]	[8]
Hidden Interdependencies of Variant Elements	OOP	C/C++	DPE; UN; µCOS	[I]	[20]
Unnecessary Configuration Overhead	OOP	C/C++	DPE; UN; µCOS	[I]	[22]
Inexplicit Variant Types	OOP	C/C++	DPE; UN; µCOS	-	[I]
Inexplicit Variant Elements	OOP	C/C++	DPE; UN; µCOS	[I]	[8]
Resource Overhead	OOP	C/C++	DPE; UN; µCOS	[I]	[8]
Divergence of Variant Elements	OOP	C/C++	DPE; UN; µCOS	[I]	[20]
Explicit Product References	OOP	C/C++	DPE; UN; µCOS	[I]	[21]
Insufficient Variant Traceability	OOP	C/C++	DPE; UN; µCOS	-	[I]
Feature Envy	OOP/AOP	Java/AspectJ	MM/AM	[B]	[13]
Data Class	OOP/AOP	Java/AspectJ	MM/AM	[B]	[13]
Divergent Change	OOP/AOP	Java/AspectJ	MM/AM	[B] [H] [L] [M]	[13]
God Class	OOP/AOP	Java/AspectJ	MM/AM	[B] [H] [L] [M]	[13]
Large Class	OOP/AOP	Java/AspectJ	MM/AM	[B]	[13]
Long Method	OOP/AOP	Java/AspectJ	MM/AM	[B] [L]	[13]
Long Parameter	OOP/AOP	Java/AspectJ	MM/AM	[B] [L]	[13]
Misplaced Class	OOP/AOP	Java/AspectJ	MM/AM	[B] [L]	[13]
Shotgun Surgery	OOP/AOP	Java/AspectJ	MM/AM	[B] [L] [M]	[13]
Inappropriate Intimacy	OOP/AOP	Java/AspectJ	MM/AM	[L]	[13]
Small Class	OOP/AOP	Java/AspectJ	MM/AM	[B] [L]	[13]
Cloned Code / Duplicated Code	OOP/AOP	Java/AspectJ	MM/AM	[O]	[13]
Duplicate Pointcut	AOP	AspectJ	MM; AM	[A] [B] [F] [K]	[31]
Anonymous Pointcut	AOP	AspectJ	MM; AM	[A] [F] [K]	[26]

TABLE V. CODE SMELLS IN CONTEXT OF SOFTWARE PRODUCT LINES (CONT.)

Code Smell	Technology	Language	SPL	Cited	Proposed
Junk Material	AOP	AspectJ	MM; AM	[A] [F]	[31]
Borrowed Pointcut	AOP	AspectJ	MM; AM	[A] [F]	[31]
Lazy Aspect	AOP	AspectJ	MM; AM	[A] [F] [K]	[26]
Various Concerns	AOP	AspectJ	MM; AM	[A] [F]	[31]
Abstract Method Introduction	AOP	AspectJ	MM; AM	[A] [F]	[26]
Feature Envy AO	AOP	AspectJ	MM; AM	[A] [F] [L]	[26]
God Pointcut	AOP	AspectJ	MM; AM	[B] [F] [H] [K]	[A]
Idle Pointcut	AOP	AspectJ	MM; AM	[B] [F] [K]	[A]
Redundant Pointcut	AOP	AspectJ	MM; AM	[B] [F] [K]	[A]
Forced Join Point	AOP	AspectJ	MM; AM	[B] [F] [K]	[A]
God Aspect	AOP	AspectJ	MM; AM	[B] [F] [K]	[A]
Composition Bloat	AOP	AspectJ	MM; AM	[B] [F] [K]	[A]
Large Aspect	AOP	AspectJ	MM	[F]	[26]
Identical Role	AOP	AspectJ	MM	[F]	[31]
Duplicated Delta Action	DOP	DeltaJ	ATM	-	[P]
Dead Delta Action	DOP	DeltaJ	ATM	-	[P]
Dead Delta Module	DOP	DeltaJ	ATM	-	[P]
Empty Delta Module	DOP	DeltaJ	ATM	-	[P]
Unused Feature	DOP	DeltaJ	ATM	-	[P]
Empty Feature	DOP	DeltaJ	ATM	-	[P]
Duplicated Features	DOP	DeltaJ	ATM	-	[P]
Joined Features	DOP	DeltaJ	ATM	-	[P]
Complex Feature Configurations	DOP	DeltaJ	ATM	-	[P]
Complex Application Conditions	DOP	DeltaJ	ATM	-	[P]

Table VI presents 14 architectural smells. This type of smell occurs at a higher level of a system's abstraction than code smells. According to the authors listed on Cited and Proposed columns, architectural smells are independent of technology. However, as the authors performed only an experiment with a specific SPL developed with the language indicated in the column *Language*, we preferred to explicit this information. In addition, when one smell is specific to SPLs, we highlighted this information in column *Context*. In this table, the *Ambiguous Interface* architectural smell [14] can be identified independent of technology and was cited by four reviewed papers ([B], [G], [K], and [L]). This architectural smell was identified in two SPLs (Aspectual Media and Mobile Media) developed in AspectJ and Java/AspectJ, respectively.

The *Component (Module) Concern Overload* architectural smell was cited in four different reviewed papers which informed that this smell was proposed by [14], but we did not find this smell there. The [G] reviewed paper presents a table with 9 architectural smells, but did not mentioned who proposed those smells. Five of these nine smells were cited in others studies and we found who proposed them. However, we have four smells which authors were Unmentioned (Proposed Column). When a smell was proposed but was not cited in another review paper, the Cited column has a hyphen.

TABLE VI. ARCHITECTURAL SMELLS IN CONTEXT OF SOFTWARE PRODUCT LINES

Architectural Smells	Context	Language	SPL	Cited	Proposed
Ambiguous Interface	independent	Java/AspectJ	MM, AM	[B][G][K][L]	[14]
Extraneous (Adjacent) Connector	independent	Java/AspectJ	AM	[B][G][K][L]	[14]
Connector Envy	independent	Java/AspectJ	AM	[B][G][K][L]	[14]
Scattered Parasitic Functionality	independent	Java/AspectJ	AM	[B][G][K][L]	[14]
Component (Module) Concern Overload	independent	Java/AspectJ	MM, AM	[B][G][K][L]	Not found
Cyclic Dependency	independent	Java/AspectJ	MM	[G]	Unmentioned
Overused Interface	independent	Java/AspectJ	MM	[G]	Unmentioned
Redundant Interface	independent	Java/AspectJ	MM	[G]	Unmentioned
Unwanted Dependencies	independent	Java/AspectJ	MM	[G]	Unmentioned
Connector Envy SPL	SPL specific	Java	Notepad SPL	-	[C]
Scattered Parasitic Functionality SPL	SPL specific	Java	Notepad SPL	-	[C]
Ambiguous Interface SPL	SPL specific	Java	Notepad SPL	-	[C]
Extraneous Adjacent Connector SPL	SPL specific	Java	Notepad SPL	-	[C]
Feature Concentration	SPL specific	Java	Notepad SPL	-	[C]

The four architectural smells: Connector Envy, Scattered Parasitic Functionality, Ambiguous Interface, Extraneous Adjacent Connector have the same name in generic (independent) context and SPL context. Therefore, in this paper we put the word SPL to differentiate them. The definitions are different, but similar, for example, *Scattered Parasitic Functionality* [14]: describes a system where multiple components are responsible for realizing the same high-level concern and, additionally, some of those components are responsible for orthogonal concerns. This smell violates the principle of separation of concerns in two ways. First, this smell scatters a single concern across multiple components.

Secondly, at least one component addresses multiple orthogonal concerns. In other words, the scattered concern infects a component with another orthogonal concern, akin to a parasite. *Scattered Parasitic Functionality SPL* [C]: this smell is characterized by the existence of a high-level concern that is realized across multiple components. That is, at least one component addresses multiple concerns, which makes the smell a bottleneck for modifiability. Components realizing scattered concerns are dependent from each other, thus have their reusability and modularity reduced.

We prefer to present these four architectural smells separately to highlight the context. Therefore, the unique architectural smell defined specific to SPL context that any similar concept had been published before is *Feature Concentration*.

Table VII presents 7 hybrid smells. This type of smell is similar to code smells, but has particularities in their definition. Considering *Feature Envy* smell, called here *Feature Envy Hybrid*, authors know the existence of this smell proposed by [13] and showed explicitly the difference. *Feature Envy* smell refers to methods that seem to be more interested in a class other than the one it actually is in. The conventional strategy to detect this smell only focuses on measuring the number of attributes that a given method accesses. The architecture-sensitive strategy is a step beyond because it considers different types of accessed code [24]. In [24], each hybrid smell is described, but they are called architecture-sensitive strategies, as these anomalies cannot be classified as code or architectural smells, we understand that a new type of smell is necessary (defined on the next section).

TABLE VII. HYBRID SMELLS IN CONTEXT OF SPL

Hybrid Smells	Technology	Language	SPL	Cited	Proposed
Shotgun Surgery Hybrid	AOP	AspectJ	MM	-	[G]
Feature Envy Hybrid	AOP	AspectJ	MM	-	[G]
Long Method Hybrid	AOP	AspectJ	MM	-	[G]
God Class Hybrid	AOP	AspectJ	MM	-	[G]
Misplaced Class Hybrid	AOP	AspectJ	MM	-	[G]
Intensive Coupling	AOP	AspectJ	MM	-	[G]
Disperse Coupling	AOP	AspectJ	MM	-	[G]

RQ3: Is the definition of a bad smell the same in the context of SPL and single software systems?

Bad smell is an extensive term and, by definition, it is associated with bad design (design anomalies) or bad programming (code anomalies). Design and code anomalies can be identified in all software systems. Thus, the definition is not different for SPLs. In the context of SPLs, the terms architectural anomalies or architectural smells are used instead of design anomalies or design smells.

RQ4: What are the refactoring methods applied to the SPL context?

Using the data extraction form, we identified 95 refactoring methods (Table VIII). Bad smells are really close and related to refactoring methods. Therefore, we decided to list the refactoring methods used in SPLs and explore them in future works. Table VIII presents 95 refactoring methods by showing name, cited, and proposed. For example, *Rename Pointcut* refactoring method was cited in [A], proposed by [25], and consists in renaming the pointcuts when the current name do not express the real function of the pointcut.

TABLE VIII. REFACTORING METHODS IN CONTEXT OF SOFTWARE PRODUCT LINES

Refactoring Method	Cited	Proposed	Refactoring Method	Cited	Proposed
Rename Pointcut	[A]	[25]	Replace nested Conditional with Guard Clauses	[D]	[30]
Introduce Aspect	[A]	[25]	Create Template Method	[D]	[30]
Rename Aspect	[A]	[25]	Consolidate Duplicate Conditional Fragments	[D]	[30]
Extract Aspect	[A]	[25]	Collapse Hierarchy	[D]	[30]
Decompose Pointcut	[A]	[25]	Pull up Constructor Body	[D]	[30]
Combine Pointcut	[A]	[F]	Pull up Field	[D]	[30]
Extract Method/Resource to Aspect	[D]	[2]	Pull up Method	[D]	[30]
Extract Context	[D]	[2]	Remove Middleman	[D]	[30]
Extract Before/After Block	[D]	[2]	Remove Setting Method	[D]	[30]
Move Field to Aspect	[D]	[2]	Extract Class	[D]	[30]
Move Import Declaration to Aspect	[D]	[2]	Extract Subclass	[D]	[30]
Move Interface Declaration to Aspect	[D]	[2]	Extract Superclass	[D]	[30]
Move Method to Aspect	[D]	[2]	Renaming of Files and Functions	[D]	[30]
Move Extends Declaration to Aspect	[D]	[17]	Splitting of Long Files	[D]	[30]
Extract Introduction	[D]	[17]	Moving of Functions From One Module to Another	[D]	[30]
Extract Advice	[D]	[17]	Conversion of Macros to Inline Functions	[D]	[30]
Extract Beginning	[D]	[17]	Changing of Data Type	[D]	[30]
Extract End	[D]	[17]	Removal of internal and external code clones Merging of different implementations and realization using conditional compilation	[D]	[30]
Extract Before/After Call	[D]	[17]	Reduction of the scale and complexity of functions	[D]	[30]
Addition at the beginning	[D]	[15]	Remove optional component	[D]	[11]
Addition at the end of the method	[D]	[15]	Make optional component a core component	[D]	[11]
Addition anywhere with a hook method	[D]	[15]	Make the one variant a core component and remove other variants	[D]	[11]
Overwrite method	[D]	[15]	Remove the unused variants	[D]	[11]

TABLE VIII. REFACTORING METHODS IN CONTEXT OF SOFTWARE PRODUCT LINES (CONT.)

Refactoring Method	Cited	Proposed	Refactoring Method	Cited	Proposed
Move entire method	[D]	[15]	Split off an optional component	[D]	[11]
Move field	[D]	[15]	Rename Feature	-	[P]
Remove field modifiers declarations	[D]	[15]	Rename Delta Module	-	[P]
Move entire class	[D]	[15]	Rename Product Line	-	[P]
Convert Alternative to Or	[D]	[3]	Extract Delta Action	-	[P]
Collapse Optional and Alternative to Or	[D]	[3]	Extract Connecting Actions	-	[P]
Add New Alternative	[D]	[3]	Resolve Modification Action	-	[P]
Turn variable features into mandatory features	[D]	[23]	Resolve Removal Action	-	[P]
Remove variable features	[D]	[23]	Merge Delta Modules with Equivalent Conditions	-	[P]
Turn variable into alternative features	[D]	[23]	Merge Delta Modules with Equivalent Content	-	[P]
Combine variable features	[D]	[23]	Merge Delta Modules with In-verse	-	[P]
Make Mandatory	[D]	[29]	Merge Configurations into Conditions	-	[P]
Make Alternative	[D]	[29]	Extract Configurations from Conditions	-	[P]
Delete Feature	[D]	[29]	Resolve Duplicated Actions	-	[P]
Copy Feature	[D]	[29]	Remove Dead Delta Action	-	[P]
Reduce Group Cardinality	[D]	[29]	Remove Dead Delta Module	-	[P]
Inline Method	[D]	[30]	Remove Empty Delta Module	-	[P]
Inline Class	[D]	[30]	Merge Compatible Partition Parts	-	[P]
Remove Middleman	[D]	[30]	Remove Empty Feature	-	[P]
Remove Setting Method	[D]	[30]	Remove Unused Feature	-	[P]
Replace Delegation with Inheritance	[D]	[30]	Merge Duplicated Features	-	[P]
Replace Temp with Query	[D]	[30]	Merge Joined Features	-	[P]
Inline Temp	[D]	[30]	Simplify Application Conditions	-	[P]
Extract Method	[D]	[30]	Simplify Feature Configurations	-	[P]
Replace Conditional with Polymorphism	[D]	[30]			

IV. DISCUSSION

We identified smells that apparently are code smells, but some information to identify these smells were obtained on architecture level. This was called architecture-sensitive strategies [4], but we are classifying as a new type of smell and calling it as hybrid smell. We considered as new type of smell because these smell's definitions are different to code and architectural smells presents on literature. However, some smells have the same name, such as the `Shotgun Surgery` code smell. Hybrid smells can be defined as:

Hybrid smells are one type of bad smells that can be identified combining the idea of one or more architectural smells with one or more code smells.

Code anomalies are even more critical to a system design when they are related to architectural problems [24]. A code anomaly, `C`, is considered as hybrid smell when: i) the code elements (e.g., methods or classes) affected by `C` are in charge of implementing architectural elements (e.g., components and interfaces); and ii) these architectural elements are affected by an architectural problem, `P` [24].

Variability smell is a relative new bad smell. This smell was mentioned in one paper of our review ([I]) and defined in a book entitled Feature-Oriented Software Product Lines [5] as a perceivable property of a product line that is an indicator of an undesired code property. It may be related to all kinds of artifacts in a product line, including feature models, domain artifacts, feature selections, and derived products.

Bad smells in the context of SPL is a young topic. This topic needs to be explored, because the use of SPLs has been grown in industry and academia [5]. When a new technology or language is created, probably new different bad smells can be identified.

Figure 5 depicts a view on bad smells. Bad smells is the biggest concept. Code and architectural smells are divisions of bad smells (types). Hybrid smells combine architectural and code smells. Variability smells are bad smells specific to SPLs and can be divided in parts, such as architectural and code smells. Considering that architectural smells can be identified in feature models, we have some information that is abstract (e.g. abstract features) that do not appear in SPL source code. Therefore, we propose an adaptation of Apel's variability smells definition [5]:

A variability smell is a perceivable property of an SPL that is an indicator of an undesired property related to all kinds of artifacts in an SPL, including feature models, domain artifacts, feature selections, source code and derived products.

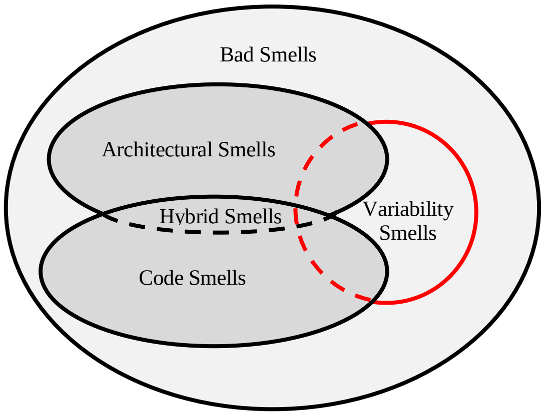

Figure 5. VIEW ABOUT DIFFERENT TYPES OF SMELLS

V. THREATS TO THE STUDY VALIDITY

The findings of this systematic review may have been affected by limitations such as bias in selection of the reviewed studies, inaccuracy in data extraction, and inaccuracy in classifying the reported evaluation approaches. It is possible that we may have not found those papers whose authors might use other terms for bad smells. Another threat to the study validity is the type of studies; we included only journals and conferences in English. We could miss important concepts in books, thesis or papers in other language than English.

Accuracy and consistency during the review process are based on a common understanding among the reviewers. Misunderstandings can result in biased results. One of the main limitations of the review can be the possibility of bias in the selection of studies. To help ensuring that the selection process was as unbiased as possible, we developed detailed guidelines in the review protocol prior to the start of the review. During the paper screening phase, we documented the reasons for its inclusion. We considered that all papers were excluded and, when one paper was accorded with all inclusion criteria it should be included. Then, we also rechecked the papers based on the inclusion criteria.

We also found that many papers lacked sufficient details of designing and executing, because sometimes we had to infer required information. This information was obtained in other studies and we reported as unmentioned or unfound when we did not find information. Additionally, we held frequent discussions among the researchers involved in this review in order to clarify any ambiguity during the review process. This practice served as a way to recheck our results, ensure that there was consistency among individual researchers, and help resolving any disagreements. We selectively ran cross-checks during the different phases of this study.

The process of classifying the evaluation approaches used (such as exploratory study and case study) involve subjective decision. To minimize it, we decided putting only when the information was explicit. The bad smells were classified as code smells, architectural smells and hybrid smells, but we did not delimitate in only one, because types of smells could be created. We could miss some information by generalizing some terms.

VI. CONCLUSION

In this paper, we presented the results of an SLR on bad smells in SPLs context. We obtained 165 references from searching the literature, from which 20 papers were selected in primary selection phase. After full-text reading, two papers were excluded, totalizing 18 selected papers.

We believed that the results provide insights into the current status of bad smell research in SPLs context. It could be seen that bad smells and code smells are a consolidated terms. Architectural smells were formally defined [4] and variability smells is a young topic that has been reported less than three years ([I] and [5]). The SLR shows that the works in Software Engineering still have problems to explicit the research method report. We can note that bad smells in SPLs context is relative new topic of study, starting in 2007. In addition, could be seen that this topic is increasing.

The main contributions of this paper are: the methodological details and results of our SLR reporting 70 bad smells classified as 49 code smells, 14 architectural smells, and 7 hybrid smells, providing a catalogue of bad smells; furthermore, we listed 16 SPLs and 95 refactoring methods; the new type of smell defined as hybrid smells, the view about smells involving bad, code, architectural, hybrid and variability smells; and; the adaptation in the Apel's definition on variability smells.

As future work, we suggest exploring and classifying the refactoring methods listed, identifying what refactoring methods can be applied to minimize or solve some bad smells, and exploring variability smells to detect gaps in literature aims to propose new smells.

ACKNOWLEDGEMENTS

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

APPENDIX

[A] I. Macia, A. Garcia and A. Von Staa. "An Exploratory Study of Code Smells in Evolving Aspect-Oriented Systems". In: 10th International Conference on Aspect-Oriented Software Development (AOSD), pp. 203-214, 2011.

[B] I. Macia, J. Garcia, D. Popescu, A. Garcia, N. Medvidovic and A.Von Staa. "Are automatically-detected code anomalies relevant to architectural modularity? An exploratory analysis of evolving systems". In: 11th Annual International Conference on Aspect Oriented Software Development (AOSD), pp. 167-178, 2012.

[C] H. S. Andrade, E. Almeida, and I. Crnkovic. "Architectural Bad Smells in Software Product Lines: An Exploratory Study". In: Working IEEE/IFIP Conference on Software Architecture (WICSA), Article No. 12, 2014.

[D] M. A. Laguna and Y. Crespo. "A Systematic Mapping Study on Software Product Line Evolution: From Legacy System Reengineering to Product Line Refactoring". Science of Computer Programming , pp. 1010 - 1034, 2013.

[E] N. Niu and S. Easterbrook. "Concept Analysis for Product Line Requirements". In: 8th ACM International Conference on Aspect-oriented Software Development (AOSD), pp. 137-148, 2009.

[F] I. Macia, A. Garcia and A. Von Staa. "Defining and Applying Detection Strategies for Aspect-Oriented Code Smells". In: 24th Brazilian Symposium on Software Engineering (SBES), pp. 60-69, 2010.

[G] I. Macia, A. Garcia, C. Chavez and A. von Staa. "Enhancing the Detection of Code Anomalies with Architecture-Sensitive Strategies". In: 17th European Conference on Software Maintenance and Reengineering (CSMR), pp. 177-186, 2013.

[H] G. F. Carneiro, M. Silva, L. Mara, E. Figueiredo, C. Sant'Anna, A. Garcia and M. Mendonca. "Identifying Code Smells with Multiple Concern Views". In: 24th Brazilian Symposium on Software Engineering (SBES), pp. 128-137, 2010.

[I] T. Patzke, M. Becker, M. Steffens, K. Sierszecki, J. Savolainenand T. Fogdal. "Identifying Improvement Potential in Evolving Product Line Infrastructures: 3 case studies". In: ACM International Conference Proceeding Series, 1, pp. 239-248, 2012.

[J] S. Schulze, M. Lochau, S. Brunswig. "Implementing Refactorings for FOP: Lessons Learned and Challenges Ahead". In: 5th International Workshop on Feature-Oriented Software Development (FOSD), pp. 33-40, 2013.

[K] I. Macia, A. Garcia, A. Von Staa, J. Garcia and N. Medvidovic. "On the Impact of Aspect-Oriented Code Smells on Architecture Modularity: An Exploratory Study". In: 5th Brazilian Symposium on Software Components, Architectures and Reuse (SBCARS), pp. 41-50, 2011.

[L] I. Macia, R. Arcoverde, A. Garcia, C. Chavez and A. Von Staa. "On the Relevance of Code Anomalies for Identifying Architecture Degradation Symptoms". In: 16th European Conference on Software Maintenance and Reengineering (CSMR), pp. 277-286, 2012.

[M] E. Guimaraes, A. Garcia, E. Figueiredo and Y. Cai. "Prioritizing Software Anomalies with Software Metrics and Architecture Blueprints". In: 35th Workshop on Software Engineering for Adaptive and Self-Managing Systems (ICSE), pp. 82-88, 2013.

[N] S. Günther and S. Sunkle. "rbFeatures: Feature-Oriented Programming with Ruby". Science of Computer Programming, pp. 152 - 173, 2012.

[O] M. Ribeiro and P. Borba. "Recommending Refactorings when Restructuring Variabilities in Software Product Lines". In: 2nd Workshop on Refactoring Tools, (WRT), in conjunction with the Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA, 2008.

[P] S. Schulze, O. Richers and I. Schaefer. "Refactoring Delta-Oriented Software Product Lines". In: 12th Aspect-Oriented Software Development (AOSD), pp. 73-84, 2013.

[Q] S. Thaker, D. Batory, D. Kitchin and W. Cook. "Safe Composition of Product Lines". In: 6th International Conference on Generative Programming and Component Engineering (GPCE), pp. 95-104, 2007.

[R] D. Rattan, R. Bhatia and M. Singh. "Software clone detection: A systematic review". Information and Software Technology , 55, pp. 1165-1199, 2013.

REFERENCES

[1] M. Ali, M. A. Babar, L. Chen and K. J. Stol. "A Systematic Review of Comparative Evidence of Aspect-Oriented Programming". In: Journal Information and Software Technology, v.52, n.9, pp.871-887, 2010.

[2] V. Alves, P. M. Junior, L. Cole, P. Borba and G. Ramalho. "Extracting and Evolving Mobile Games Product Lines". In: 5th Software Product Lines Conference (SPLC), pp. 70-81, 2005.

[3] V. Alves, R. Gheyi, T. Massoni, U. Kulesza, P. Borba, and C. Lucena. "Refactoring Product Lines". In: 5th International Conference on Generative Programming and Component Engineering (GPCE), pp. 201-210, 2006.

[4] H. S. Andrade, E. Almeida and.I. Crnkovic. "Architectural Bad Smells in Software Product Lines: An Exploratory Study". In: Working IEEE/IFIP Conference on Software Architecture (WICSA), Article No. 12, 2014.

[5] S. Apel, D. Batory, C. Kastner and G. Saake. "Feature-Oriented Software Product Lines: Concepts and Implementation". Springer, p 315, 2013.

[6] S. Apel and C. Kastner. "An Overview of Feature-Oriented Software Development". Journal of Object Technology, v.8, n.5, p. 49-84, 2009.

[7] T. Asikainen, T. Mannisto, and T. Soininen. "A Unified Conceptual Foundation for Feature Modelling". In: 10th International Software Product Line Conference (SPLC), pp. 31-40, 2006.

[8] P. G. Bassett. "Framing Software Reuse". Yourdon Press, 1997.

[9] D. Batory, J. Sarvela and A. Rauschmayer. "Scaling Step-Wise Refinement". In: 25th International Conference on Software Engineering, pp.187-197, 2003.

[10] Q. Boucher; A. Classen; P. Faber; P. Heymans, "Introducing TVL, a Text-Based Feature Modelling Language". In: 4th International Workshop on Variability Modelling of Software-intensive Systems, pp. 159-162, 2010.

[11] M. Critchlow, K. Dodd, J. Chou, and V. Hoek. "Refactoring Product Line Architectures". In: Achievements, Challenges, and Effects, pp. 23-26, 2003.

[12] E. Figueiredo, C. Sant'Anna, A. Garcia, and C. Lucena. Applying and Evaluating Concern-Sensitive Design Heuristics. In Journal of Systems and Software (JSS), 85(2), pp. 227-243, 2012

[13] M. Fowler. "Refactoring: Improving the Design of Existing Code". Addison Wesley, 1999.

[14] J. Garcia, D. Popescu, G. Edwards and N. Medvidovic. "Identifying Architectural Bad Smells". In: 13th Software Maintenance and Reengineering (CSMR), pp 255-258, 2009.

[15] R. E. L. Herrejon, L.M. Mendizabal and Egyed. "From Requirements to Features: An Exploratory Study of Feature-Oriented Refactoring". In: 15th International Software Product Line Conference (SPLC), pp. 181-190, 2011.

[16] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, A. S. Peterson. "Feature-Oriented Domain Analysis (FODA) - Feasibility Study". In: SEE Technical Report CMU/SEI-90-TR-021. 1990.

[17] C. Kastner, S. Apel and D. Batory. "A Case Study Implementing Features Using AspectJ". In: 11th International Software Product Line Conference (SPLC) pp. 223-232, 2007.

[18] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J. M. Loingtier and J. Irwin. "Aspect-oriented programming". In: 11th European Conference on Object-Oriented Programming, pp.220-242, 1997.

[19] B. Kitchenham and S. Charters. "Guidelines for Performing Systematic Literature Reviews in Software Engineering". Software Engineering Group, School of Computer Science and Mathematics, Keele University, EBSE Technical Report Version 2.3, 2007.

[20] R. Kolb, D. Muthig, T. Patzke, and K. Yamauchi. "Refactoring a Legacy Component for Reuse in a Software Product Line: A Case Study". Journal of Software Maintenance and Evolution: Research and Practice 18(2): pp. 109-132, 2006.

[21] C. W. Krueger. "New Methods behind a New Generation of Software Product Line Successes". In Kang, K.C., Sugumaran, V., and Park, S. (Eds.): Applied Software Product Line Engineering. Auerbach Publications, pp. 39-60, 2010.

[22] C. W. Krueger. "The 3-Tiered Methodology". In: Software Product Line Conference (SPLC) pp. 97-106, 2007.

[23] F. Loesch and E. Ploedereder. "Restructuring Variability in Software Product Lines using Concept Analysis of Product Configurations". In: 11th European Conference on Software Maintenance and Reengineering (CSMR), pp. 159-170, 2007.

[24] I. Macia, A. Garcia, C. Chavez, and A. von Staa. "Enhancing the Detection of Code Anomalies with Architecture-Sensitive Strategies". In: Software Maintenance and Reengineering (CSMR), 2013 17th European Conference on, pp. 177-186, 2013.

[25] M. P. Monteiro and J. M. Fernandes. "Towards a Catalog of Aspect-Oriented Refactorings". In: 4th International Conference on Aspect-Oriented Software Development (AOSD) pp. 111-122, 2005.

[26] E. K. Piveta, M. Hecht, M. S. Pimenta and R. T. Price. "Detecting Bad Smells in AspectJ". Journal of Universal Computer Science, vol. 12, 2006.

[27] K. Pohl, G. Bockle, F. J. V. Linden. "Software Product Line Engineering: Foundations, Principles, and Techniques". Berlin: Springer, p. 490, 2005.

[28] K. Pohl and A. Metzger. "Software Product Line testing". In: Communication of the ACM, pp78-81, 2006.

[29] I. Savga and F. Heidenreich. "Refactoring in Feature-Oriented Programming Open Issues". In: Workshop on Modularization, Composition, and Generative Techniques for Product Line Engineering, pp. 41-46, 2008.

[30] N. Siegmund, M. Kuhlemann, M. Pukall, S. Apel. "Optimizing Non-functional Properties of Software Product Lines by means of Refactorings". In: Fourth International Workshop on Variability Modelling of Software-intensive Systems (VaMoS), pp. 115-122, 2010.

[31] K. Srivisut, and P. Muenchaisri. "Bad-Smell Metrics for Aspect-Oriented Software". In: 6th IEEE/ACIS International Conference on Computer and Information Science (ICIS), pp. 1060-1065, 2007.

[32] D. M. Weiss and C. T. R. Lai. "Software Product-Line Engineering: A Family-Based Software Development Process". Addison-Wesley, 1999.

[33] M. Zhang, T. Hall and N. Baddoo, "Code Bad Smells: A Review of Current Knowledge". In: Journal of Software Maintenance and Evolution: Research and Pratice, pp. 179-202, 2011.