Architectural Bad Smells in Software Product Lines: An Exploratory Study

Hugo Sica de Andrade
Federal University of Bahia
Salvador, Brazil;
Mälardalen University
Västerås, Sweden
hugosica@dcc.ufba.br

Eduardo Almeida
Federal University of Bahia
Av. Adhemar de Barros
Salvador, Brazil
esa@dcc.ufba.br

Ivica Crnkovic
Mälardalen University
Högskoleplan 1, 721 23
Västerås, Sweden
ivica.crnkovic@mdh.se

ABSTRACT

The Software Product Lines (SPL) paradigm has arisen for taking advantage of existing common aspects between different products, while also considering product-specific features. The architecture of a SPL comprises a model that will result in product architectures, and may include solutions leading to bad (architectural) design. One way to assess such design decisions is through the identification of architectural bad smells, which are properties that prejudice the overall software quality, but are not necessarily faulty or errant. In this paper, we conduct an exploratory study that aims at characterizing bad smells in the context of product line architectures. We analyzed an open source SPL project and extracted its architecture to investigate the occurrence or absence of four smells initially studied in single systems. In addition, we propose a smell specific to the SPL context and discuss possible causes and implications of having those smells in the architecture of a product line. The results indicate that the granularity of the SPL features may influence on the occurrence of smells.

Categories and Subject Descriptors

D.2.7.g [**Software Engineering**]: Distribution, Maintenance and Enhancement - *Maintainability*; D.2.11 [**Software Engineering**]: Software Architectures

General Terms

Architecture, Evaluation

Keywords

software product lines, architectural bad smells, exploratory study

WICSA '14, April 07 - 11 2014, Sydney, NSW, Australia

http://dx.doi.org/10.1145/2578128.2578237

1. INTRODUCTION

It is known that the Software Product Lines (SPLs) paradigm provides a beneficial development style in which the commonalities and the differences in the applications of the SPL are designed in a common way [11]. The common and variable properties of the applications are managed into a rational, though often complex environment. The organization of such properties must enable the derivation of different product line members through a Product Line Architecture (PLA), which also supports SPL maintenance by changes in its design decisions.

In this work, rather than considering the product-specific architectures, we focus on the PLA, which should ideally be flexible enough to support changes in features and in how they are composed to develop products. Since PLAs have a longer life span and support the development of a range of products, it is important to carefully evaluate the design decisions prior to perform implementation and refactoring activities.

One way to assess design decisions is through the identification of *architectural smells*. The term refers to a set of architectural design decisions that negatively impacts system lifecycle properties, such as understandability, testability, extensibility and reusability. In other words, they indicate combinations of architectural constructs that induce reductions in system maintainability [6]. The referred smells are results of architectural decisions that despite causing a negative impact in quality, do not necessarily represent malfunctions or conformance violations (e.g. architectural drift [12] and erosion [14]).

The goal of this work is to characterize architectural bad smells in the context of PLAs through an exploratory study. For this, we investigate the occurrence of four representative architectural smells that were initially proposed for single systems. Further, we expand upon the original work by proposing a smell specific to PLAs. The investigation promotes a discussion about the relation between variability-based architectures and the identified architectural smells. The workflow of this work consisted in analyzing a sample SPL in the domain of text editors, extracting its conceptual architecture from the code and available documentation, and finally searching for the smells.

The remainder of this paper is organized as follows: Section 2 presents related work. Section 3 presents definitions and the representative smells studied in this work. Section 4 presents the exploratory study with a description of the Notepad SPL, its architecture recovery, and the search for

smells. Section 5 presents a discussion. In Section 6, we discuss the threats to validity of this work. Finally, Section 7 presents the conclusion and future work.

2. RELATED WORK

The issues involving both architectural refactoring and assessment techniques have been extensively discussed over the last years. In the context of single systems, methods such as ATAM [8] aim at assessing the consequences of architectural decisions in terms of quality requirements. The evaluation often uses metrics for measurements regarding the potential risk within a complex architecture, while the conformances to business drivers are also taken into consideration.

In this work, we are guided by the method presented by [6]. The authors discuss the identification of four representative architectural smells, i.e. design attributes that negatively impact the system's maintainability. Instead of addressing the refactoring of implementation artifacts, the study discusses architecture design decisions that are not necessarily faulty or errant, but still present a negative impact in software quality. The introduced smells were proposed based on the experience of two large industrial systems and case studies in the literature.

To the best of our knowledge, no studies have been undertaken from the viewpoint of architectural smells in SPLs. Instead, the published literature discusses software anti-patterns [3]. Despite the range of works discussing the impact of *code smells* [5] in software architecture [2, 10], they differentiate from architecture smells in the abstraction level. While architecture smells are related to design problems, code smells are obtained through an evaluation of source code. In this sense, different stakeholders are able to assess the architecture design of features and products as soon as in the design level, instead of having the source code artifact as the starting point.

3. ARCHITECTURAL SMELLS

The concept of code smells is often used to indicate properties that are not necessarily faulty or errant, but present negative effect in a system's maintainability [6]. For example, in an object-oriented system one entity can be designated as *God Class* if it assumes too much responsibility when compared to the remaining classes. These implementation decisions can make it difficult to maintain and evolve such class, due to its increased relationship level with the other entities. Other examples of code smells can be mentioned, such as *Feature Envy*, which means a case where one method it too interested in other classes.

Nevertheless, those smells are restricted to implementation level constructs, such as methods, classes, statements and parameters. When poor structuring of architecture-level constructs (i.e. components, connectors and interfaces) causes a reduction in the system maintainability, these properties are called architectural bad smells.

The term architectural bad smell was originally used by [9], for describing an indication of an underlying problem that occurs at a higher level of a system's abstraction than a code smell. Architectural smells are structural attributes that mainly affect lifecycle properties - such as understandability, testability and reusability - but can also affect quality properties, such as performance and reliability. According to the discussion in [6], such a phenomenon may be caused by (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.

The occurrence of smells in a design may represent a justification in different concerns. However, we argue that the trade-offs should be assessed to also allow means to adequately maintain the system at architecture level. Software architects should evaluate whether actions to change the identified properties will result in the actual benefits when accounting the change impact as a whole. In the context of SPLs, the importance of properly evaluating such impact is increased, due to the key role of the PLAs and the often-widespread effect of changes in design.

Architectural smells are remedied by changing the structure and the behaviors of the internal system elements without changing the external behavior of the system. That is, products within the scope of the SPL should not be affected in terms of functionality by architectural changes. Further, the PLA is still required to support the derivation of those products.

In this work, we do not address the issues related to differences between the intended and implemented architectures. Since the intended architecture is often outdated and/or poorly specified, we focus the analysis on the actual architecture. The disadvantage of such decision is that a recovery process is required prior to the identification of smells. On the other hand, the implemented architecture always exists, and represents actual constructs with respect to the system organization.

3.1 Representative Smells

This section describes the five architectural smells investigated in this work. In addition to the four representative smells formerly introduced in the literature [6], we propose a new type of smell that is specific to SPL projects. For the following definitions, we maintain the concepts of components, connectors and interfaces as further discussed in section 4.

Connector Envy: Components with Connector Envy cover too much functionality with regard to connections. Instead of having the interaction facilities delegated to a connector, the components encompass, to a great extent, one or more of the following types of interaction services: *communication*, *coordination*, *conversion* and *facilitation*.

Scattered Parasitic Functionality: 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.

Ambiguous Interfaces: When a component offers only one single and generic entry-point, such interface is referred to as ambiguous. Ambiguous interfaces reduce static analyzability and can occur independently of the implementation-level constructs that realize them. Despite the fact that the component may offer and process multiple services, an ambiguous interface will offer only one public service or method.

Extraneous Adjacent Connector: This smell is characterized by the use of two connectors of different types to link a pair of components. For example, the use of procedure calls makes the transfer of control explicit, thus understand-

ability is increased. On the other hand, event connectors can be easily replaced or updated because senders and receivers of events are usually unaware of each other. In this case, reusability and adaptability are increased. However, having both solutions simultaneously may cancel each other's benefits and negatively affect the architecture's understandability.

Feature Concentration: This smell is specific to the SPL context, and is characterized by the centralization of the SPL features in one architectural entity. The feature concentration smell opposes the properties in the scattered functionality smell by implementing different functionalities in a single design construct. Such decision might also be accompanied by the definition of only one generic entry point for a component, indicating the occurrence of the ambiguous interfaces smell. Concentrating all features in one architecture entity may represent high complexity in such construct, thus bringing negative effects to its understandability and changeability. We propose the feature concentration smell after observations from the exploratory study using a sample SPL, which is described in the next section.

4. THE EXPLORATORY STUDY

The present study aims at characterizing the problem through the main question: "**Do architectural bad smells occur in software product lines?**". We are particularly interested in investigating whether the same smells often found in single systems can also be present in PLAs. Further, we discuss the implications on maintainability when adopting design decisions that characterize smells in a SPL environment.

In order to address the aforementioned issues, we selected a sample SPL that has been redefined from an open source project in the text editor domain. The redefinition process is further explained in the following subsections. Since there was no architecture artifacts related to this project, we undertook a recovery process prior to identifying smells. In summary, the study workflow consisted in (1) analyzing a sample SPL in the domain of text editors, (2) extracting its conceptual architecture from the code, and (3) searching for smells in the recovered architecture. Due to space constraints, in this paper we focus on the description of Notepad SPL, present its conceptual architecture, and discuss the search for smells.

4.1 Notepad SPL

Notepad is a Java implementation resulting from a feature-oriented design course in the University of Texas at Austin. It consists of several different product lines that were also used in an empirical study within the FeatureVisu project, which is a structure analysis and measurement tool for SPLs [1]. The class assignment was to individually develop a SPL in the text editor domain using a common base and applying feature orientation development concepts.

The idea of Notepad SPL is to allow the selection of a number of features to compose text editor functions. With this implementation, the user is able to choose different features that will be composed into the desired product, thus making an impact on both the GUI and the functional capabilities of the product to be derived.

The Notepad release contains a set of 7 different SPL versions, and can be obtained in the FeatureVisu project website[1]. Each of the SPLs implements a random set of features related to text editing, such as Copy, Paste, and Find. The products contain from 1397 to 1716 lines of code, and also vary in the number of features: 4 to 10.

We intended to consider every feature implemented across the different SPL versions. Since the source code of each SPL was conceived separately, the first step of our work consisted in analyzing the code and designing a Feature Model [7] using the FeatureIDE tool [13] and providing each feature the proper level of abstraction. A few graduate student members of RiSE research group[2] were involved in the task of modeling the features and turning the separated SPLs into one single SPL (which we call Notepad SPL in the context of this work). Notepad SPL contains 18 features, from which 5 are mandatory to all products, and the remaining 13 are either optional or required according to the selection of features and the rules specified in the feature model.

4.1.1 Variability Management

When implementing our version of the Notepad SPL, we used the Colored Integrated Development Environment (CIDE) tool [4] to orthogonally manage the SPL variability. The tool consists of an Eclipse IDE[3] plug-in that features manual code annotations and results in conditional compilation using preprocessors. The concerns (SPL features) are separated visually through background coloring. Moreover, different views can be used to identify scattered functionalities that may not be modularly considered during the design stage.

The CIDE tool allows programmers to annotate blocks of code concerning each of the previously defined features. Once the feature-related code is annotated and the tool is executed, the tool provides a feature selection window through which the product configuration is possible. Every feature contains a corresponding checkbox, and the mandatory ones are set as true by default. When a constraint-related feature is selected, the software automatically provides checks to solve conflicts. A new product is generated once all the desired features are selected and the code corresponding to that given set of features is compiled.

Notepad SPL was developed without proper documentation to support its decisions. Besides the feature model, the only artifact that could be used for analysis and possible improvements was the source code with very few comments. Thus, we extracted its architecture through an analysis based on both automated (using the Structure101 tool[4]) and manual analysis (considering the source code and documentation).

4.1.2 Notepad SPL Conceptual Architecture

The extracted Notepad SPL architecture is the composition of 6 components, as presented in Figure 1. It respects the rules that are determined by the components and interfaces as well as corresponds to the features and the variability management mechanism implemented in the source code. Component interactions are defined by the interfaces depicted in the figure, which determine the type of communications that are entitled.

[1]http://www.fosd.de/FeatureVisu
[2]http://wordpress.dcc.ufba.br/riselabs
[3]http://www.eclipse.org
[4]http://structure101.com

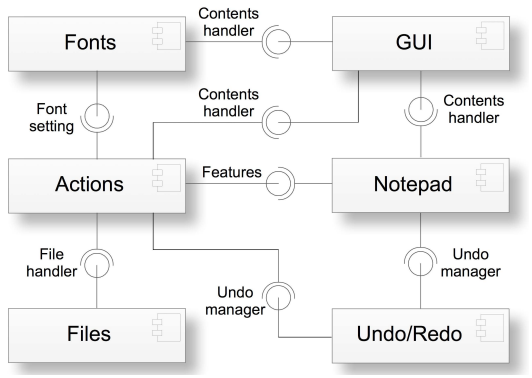

Figure 1: Notepad SPL Component Model

In order to perform the recovery of the PLA, we considered the available artifacts: source code, design artifacts and domain knowledge, which were manually and automatically analyzed. The resulting component model was validated by the programmers involved in the assembly of Notepad SPL.

The features implemented in the Actions component are manually annotated at development time. A number of features require core handlers that are provided by pre-set Java libraries, such as the ones related to the Files and Fonts components. Derived products are instances of Notepad, which implements listeners for events provided through the GUI interfaces. Also, a dedicated component is designated to implement Undo/Redo text functionalities. Annotated code inputs the CIDE plugin, which in turn provides a valid configuration in a XML file according to previous feature modeling.

4.2 Search for Smells

From the recovered Notepad SPL architectural model, we searched for the aforementioned smells and discussed their implications in the SPL context. We maintain the definitions of components, connectors and interfaces as previously discussed. Next, we discuss the existence of the smells and highlight their impacts on quality attributes.

4.2.1 Connector Envy

As previously mentioned, the Actions component contains the implementation of the functionalities, as the Notepad component triggers what is displayed in the text editor window after CIDE's validation. We identified the connector envy smell in the Notepad component, due to its extensive coordination and communication links to other components. Notepad contains an operation that transfers control set by the provided events to the Actions component through several conditional statements. The direct method invocation represents a negative effect to the component's reusability in that the dependability of these components is increased.

Despite the fact that ideally each feature should be considered a component, the granularity of these functionalities' specifications is set by the simplicity of the operations. By grouping fine-grained features into one architecture entity, different functionalities may demand diverse interfaces to realize their requirements. In the case of Notepad component, it requires both Undo/Redo functionalities and feature implementations in Actions, which can be translated to intense communication requirements between these components. The Notepad component imports several libraries that are used to implement the graphical interface of generated products, and also implements a listener of events, which are sent through GUI interactions.

Combining product construction capabilities and connector responsibilities represents a reduction in testability because application functionalities and interaction functionalities cannot be separately tested. In a SPL, it should be possible to perform tests both in product-specific architectures (e.g. through scenarios) and in the PLA for assessing the variability mechanism and derivation procedures with respect to quality attributes.

Understandability is also compromised due to the unclear separation of roles and concerns. On the other hand, providing an interaction mechanism would imply in the deployment of new processes, as well as increment the overall complexity of the system.

4.2.2 Scattered Parasitic Functionality

We were not able to find scattered functionality evidence in Notepad SPL. Even though the GUI concern implements the triggered GUI attributes in Actions and aids in constructing the visual components orchestrated by Notepad, it refers to the Java native library to handle interfaces. Since it does not represent a high-level concern, changes in such related components would not represent a widespread impact.

The main class responsible for specifying the product, Notepad, inherits the properties of JFrame class. Its constructor sets the default window settings that will be composed with the valid configuration received from CIDE at development time. The Actions component is notified about the user graphics interface events and realizes the required functionalities via a direct procedure call from Notepad. In this sample case, refactoring these two components would not be complicated. However, the cost requirements to change aspects in a shared concern of bigger projects might prevent such changes to be made.

In SPLs, scattered functionalities can represent aspects that are difficult to realize, given the widespread effect among products. In a shared concern environment, the understandability is compromised due to orthogonal contexts being realized in a single component. Feature granularities should be adjusted at specification time in order to take advantage of each component's reusability levels. That is, having coarse-grained features can elevate the chances of specifying a single component with multiple concerns. On the other hand, fine-grained features can induce architects to realize errors during specification, for instance, having similar functionalities under different components.

4.2.3 Ambiguous Interfaces

Ambiguous Interfaces offer a single generic entry-point. This smell results in reduced understandability because an ambiguous interface does not reveal which services a component is offering. In other words, a user of this component needs to inspect the implementation of the internals before using its services. In the case of Notepad SPL, this smell is characterized in the Actions-Notepad components interface. All features are implemented in the Actions component and accessed through one generic action listener.

The event-based parameterization present a dynamic dispatch despite the conditional compilation imposed by CIDE.

Such scenario is usually motivated by the implementation of generic GUI listeners. In the case of Notepad, the triggers are executed through the user relation with the buttons and text pane area. This type of solution can be implemented in small SPLs, but might cause complications when understanding and analyzing the feature entity. On the other hand, with such centralized solution, new features can be more easily added and specified respecting the limits of one component and obeying a single connecting point.

We argue that this smell is common in SPL engineering due to its intrinsic characteristics. The common implementation and composition of features is given by one single repository of features that obey a rule at the time of composition and product derivation. This solution would probably carry a number of down points in the case of a large SPL supporting hundreds of features, for example. Such a generic entry-point could be overloaded by functionalities that differ too much, thus require very different handlers for effective management.

4.2.4 Extraneous Adjacent Connector

If two connectors of different types are used to link a pair of components, it is possible that the beneficial effects of each individual connector will cancel each other out. Although this smell also considers other types of connectors, we focus on the combination of procedure calls and event connectors.

Having two components with two connector types may affect understandability, because it becomes difficult to determine whether and under which circumstances additional communication occurs between the affected components. The side effects of using both connector types should be assessed, since the beneficial effects of each individual connector might cancel each other out. Although we focused on the combination of procedure calls and event connectors, this smell also considers other types of connectors.

In our sample SPL, the Actions component provides functionality services to construct a Notepad instance (product). A procedure call would represent the occurrence of this smell due to the use of event listeners to trigger the execution of features. However, the only interaction between Notepad and Action components is through events provided by the GUI. Thus, we were not able to identify the extraneous adjacent connector smell in Notepad SPL.

The scenario of extraneous adjacent connector could be found in a SPL implementation that breaks the unique connecting restriction set by a given PLA. For instance, if an event bus is designed to handle communications between an instantiation entity and a feature repository, there should be no direct procedure call between these two components. Such restriction can be assured by setting protective rules to access the repository, either through parameterization or strict procedure call policies.

4.2.5 Feature Concentration

Design decisions are usually evaluated driven by metrics, such as coupling and cohesion between different entities. For example, in a store management system, the employee registration and payment modules should be closely related in the architectural design because both of them represent human resources concerns. At the same time, the inventory should be handled by different instances since it is not closely related to these modules.

Such design solution might not be obvious in a SPL environment when the feature granularity is too fine grained, because it would require effort to accommodate different concerns in different design entities. A generic solution would be to group all features into one entity and implement a configuration management mechanism that will derive products according to previously defined rules. The decision to ignore different concerns at design time represents easier development, but also negatively affects the system understandability and modularity attributes.

As previously mentioned, the Actions component concentrates the features in one component through a single generic entry-point, which characterizes both feature concentration and ambiguous interfaces smells. In the case of Notepad SPL, such decisions are supported by the fact that it is not a complex implementation, so understandability is not a major concern in the project. However, we argue that making the same design decisions in a SPL with e.g. several hundreds of features could probably represent negative impacts on this PLA's understandability and maintainability capabilities throughout its lifecycle.

5. DISCUSSION

The identification of smells is relevant because points of improvement can be identified and refactored in the architecture level regardless of changes in the SPL scope. This level of abstraction addresses structural properties that have direct influence in the overall system lifecycle properties. Despite the generic decisions that can be observed in any software architecture, some SPL-specific properties can indicate the occurrence of smells, such as the concentration of features in a single design entity. However, we point that the identification of smells is dependent on the architecture design specification, which may in turn carry subjective interpretations (e.g. the definition of a concern). Recovering design decisions represent a difficult task that may influence in the detection of properties that can be remedied.

This exploratory study provides insights on the causes and the issues involving the identification of architectural smells in SPLs. For instance, the existence of connector envy smell in Notepad SPL suggests that PLAs are likely to be designed with an entity containing a central mechanism that deals with both functionality and connection capabilities. Such design decision increases the implementation complexity and impacts in the PLA evolution capabilities.

Further, we argue that some types of design decision could be related to the granularity of features. For instance, when having fine-grained features, developers are tempted to group all functionalities into one design entity, as a feature repository. Such decision may cause negative effects to the PLA's maintainability, which can be refactored through the identification of properties described as ambiguous interfaces and feature concentration smells.

The SPL considered in this study is part of two academic projects, and a set of products was generated to perform the architecture recovery process. Even though we focused the analysis on the PLA in the search for points of improvement, all products derived from it would also be affected by the identified smells. Regardless of the variability present in the derivation process and different features included in different products, the architectural entities affected by the smells are mandatory to all products. That is, all product specific architectures carry the aforementioned trade-offs and could

be individually refactored for better maintainability outside the SPL environment.

6. THREATS TO VALIDITY

Most threats to the validity of this work are related to subjective interpretations. For the architecture recovery process, we designed the PLA based on the analysis of the source code and adjacent artifacts. Recovering architectures from code represents a difficult task through which important aspects may not be properly captured accurately, such as coupling and cohesion. We minimized the threat of misconceptions in defining the architecture by carrying out discussions with developers involved in the implementation our Notepad SPL version.

We understand that many software projects do not include architecture documentation, which makes it difficult and costly for one to investigate the occurrence of architectural smells. Further, despite the fact that several studies had been undertaken in the area of code smells, the concept of architectural smells is fairly recent. We limited the discussion on a set of five smells, although additional ones may occur.

7. CONCLUSIONS AND FUTURE WORK

In this work, we presented a case study that aimed at characterizing the phenomenon of architectural bad smells in the context of SPLs. The study consisted in (i) analyzing a sample SPL in the domain of text editors; (ii) extracting its conceptual architecture from the code; and (iii) searching for bad smells in the recovered architecture. We focused on the analysis and in the search for smells due to space constraints.

We have discovered that not all architectural smells previously proposed occurred in the sample text editor SPL. We argue that the choice of using Java libraries may have influenced the process of identifying smells due to its intrinsic architectural implications. In addition, we proposed a SPL specific smell that indicates the concentration of features in a single design entity.

As future work, we plan to investigate the occurrence of smells - those discussed here and possibly new ones - in a SPL of a different domain, in order to compare results and draw further conclusions. It would be valuable to also investigate the occurrence of smells in industrial projects through a survey with SPL experts, considering actual architectural problems they encounter. Another interesting discussion would be about how to effectively remedy the occurrence of smells, either through specific manual operations or using tools to support corrections.

Acknowledgments: This work was partially supported by the National Institute of Science and Technology for Software Engineering (INES[5]), funded by CNPq and FACEPE, grants 573964/2008-4 and APQ-1037-1.03/08 and CNPq grants 305968/2010-6, 559997/2010-8, 474766/2010-1 and FAPESB.

8. REFERENCES

[1] S. Apel and D. Beyer. Feature cohesion in software product lines: an exploratory study. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 421–430, New York, NY, USA, 2011. ACM.

[2] R. Arcoverde, I. Macia, A. Garcia, and A. von Staa. Automatically detecting architecturally-relevant code anomalies. In *Recommendation Systems for Software Engineering (RSSE), 2012 Third International Workshop on*, pages 90 –91, june 2012.

[3] W. J. Brown, R. C. Malveau, H. W. McCormick, III, and T. J. Mowbray. *AntiPatterns: refactoring software, architectures, and projects in crisis*. John Wiley & Sons, Inc., New York, NY, USA, 1998.

[4] J. Feigenspan, C. Kästner, M. Frisch, R. Dachselt, and S. Apel. Visual support for understanding product lines. In *The 18th IEEE International Conference on Program Comprehension, ICPC 2010, Braga, Minho, Portugal, June 30-July 2, 2010*, pages 34–35, 2010.

[5] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA, 1999.

[6] J. Garcia, D. Popescu, G. Edwards, and N. Medvidovic. Toward a catalogue of architectural bad smells. In *Proceedings of the 5th International Conference on the Quality of Software Architectures: Architectures for Adaptive Software Systems*, QoSA '09, pages 146–162, Berlin, Heidelberg, 2009. Springer-Verlag.

[7] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical report, Carnegie-Mellon University Software Engineering Institute, November 1990.

[8] R. Kazman, M. Klein, and P. Clements. Atam: Method for architecture evaluation. Technical Report CMU/SEI-2000-TR-004, Carnegie Mellon Uiversity, Software Engineering Institute, 2000.

[9] M. Lippert and S. Roock. *Refactoring in Large Software Projects: Performing Complex Restructurings Successfully*. Wiley, 1 edition, May 2006.

[10] 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 *Proceedings of the 11th annual international conference on Aspect-oriented Software Development*, AOSD '12, pages 167–178, New York, NY, USA, 2012. ACM.

[11] K. Pohl, G. Böckle, and F. J. v. d. Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[12] J. Rosik, A. Le Gear, J. Buckley, M. A. Babar, and D. Connolly. Assessing architectural drift in commercial software development: a case study. *Softw. Pract. Exper.*, 41(1):63–86, Jan. 2011.

[13] T. Thüm, C. Kästner, F. Benduhn, J. Meinicke, G. Saake, and T. Leich. Featureide: An extensible framework for feature-oriented software development. *Science of Computer Programming*, 2012. to appear; accepted 7 Jun 2012.

[14] J. van Gurp and J. Bosch. Design erosion: problems and causes. *J. Syst. Softw.*, 61(2):105–119, Mar. 2002.

[5]http://www.ines.org.br