On the Impact of Aspect-Oriented Code Smells on Architecture Modularity:
An Exploratory Study

Isela Macia[1], Alessandro Garcia[1], Arndt von Staa[1], Joshua Garcia[2], Nenad Medvidovic[2]
[1]Opus Group, LES, Informatics Department, PUC-Rio, RJ, Brazil
[2]University of Southern California, Los Angeles, CA, USA
{ibertran, afgarcia, arndt}@inf.puc-rio.br, {joshuaga, neno}@usc.edu

***Abstract*—** Aspect-oriented programming (AOP) aims to improve software modularity, although developers can unwittingly introduce code smells in their programs. A code smell is any symptom in the source code that possibly indicates a deeper modularity problem. Several works have been concerned about code smell occurrences in aspect-oriented systems. However, there is little knowledge about their actual damage to the modularity of architectural designs. This gap makes it difficult for developers understand and manage the harmful architecture-level impact of recurring code smells. This paper presents an exploratory analysis that investigates the influence of aspect-oriented code smells on evolving architectural designs. We analyzed code smell occurrences in 14 versions of 2 applications designed using different architectural styles. The outcome of our evaluation suggests that code smell occurrences often entail architecture modularity problems in these systems. Even worse, our analysis revealed that certain architecturally-relevant code smells were not targeted by the refactoring strategies in place.

Keywords- aspect-oriented programming, code smells, architectural smells

I. INTRODUCTION

New modularity techniques [14][23][24], such as aspect-oriented programming (AOP) [14] and its dialects [8][16] have been constantly emerging over the last years. Their powerful programming mechanisms are claimed to support program decompositions with superior modularity. Therefore, it is expected these decompositions are also closer to the intended architecture modularity [15]. For instance, AOP aims at achieving this promise by consistently preserving modularity of crosscutting concerns [14] from architecture to code. The use of AOP allows making architecture decisions governing crosscutting concerns more explicit in the implementation. Then, someone could expect that modularity of aspect-oriented architectures is more resilient when the source code is produced and evolved.

On the other hand, the expressive power of AOP mechanisms might facilitate the introduction of certain *code smells* [18][22][28]. Code smells are implementation structures that possibly indicate modularity problems [5]. Code smells are particularly severe when introduce modularity problems in architecture designs, i.e. the so-called *architectural smells* [6]. Architectural smells are compositions of architecture-level elements that often plaster the architecture modularity [6][10] and, hence, hinder software maintainability [6] . As a consequence, as new code smells continuously lead to architectural smells and remain undetected, it becomes increasingly more difficult to modify and include new features in aspect-oriented systems.

The challenge is that there is no empirical knowledge about the relationship between aspect-oriented code smells and architectural smells. In fact, many of these code smells might not incur in any damage to the modularity of architecture designs. Previous work only focused on the characterization of code smells [18][22][28] that occur in aspect-oriented systems. Some of them empirically studied the code smell behavior in evolving software systems [18][28]. However, there was no attempt in understanding the cause-effect relationship between code smells and architectural smells in aspect-oriented systems. As a result, software developers have no knowledge on how to master and rank code smell occurrences according to their influence on architectural design degradation [10]. The likelihood of an aspect-oriented code smell affecting the architecture modularity is even more critical in realistic cases: architectural designs are not explicitly documented as the system evolves. Then, developers also need this knowledge to determine, for instance, which part of the code should be refactored first given their potential negative impact on the 'implicit' architecture in the code.

Therefore, there are many open questions regarding the interplay of architectural problems and the emergence of smells in aspect-oriented code. To what extent are existing catalogs of aspect-oriented code smells [18][22][28] helpful for supporting the identification or prediction of architectural problems? What is the frequency of aspect-oriented code smells that induce architectural smells? How can patterns of aspect-oriented code smell occurrences be used to prioritize architecturally-relevant refactorings? Current empirical studies tend to focus solely on the factual analysis of code smells [18][22][28], disregarding their impact on architecture modularity.

This paper presents an exploratory analysis about the influence of aspect-oriented code smells in two evolving system's architectures [11][19]. Our investigation focused on 14 versions of those product lines (Section III.C). These systems were chosen because more than 20 developers with different levels of experience in aspect-orientation were involved in the projects. In addition, the original architects were available to support us in reliably identifying architectural smells in those systems. More importantly, we needed

978-0-7695-4626-1/11 $26.00 © 2011 IEEE
DOI 10.1109/SBCARS.2011.18

to rely on systems where architectural information was available, so that we could correlate the architectural smells with the presence of aspect-oriented code smells. In particular, our analysis (Section IV) revealed that aspect-oriented code smells can be used as effective indicators of architectural problems.

We have also performed some statistical analyses in our study, which enabled us to derive a number of interesting observations (Section V), such as:

- The amount of architectural smells related to aspect-oriented code smells was much higher than those that present no relation. This means that the observation of smells in aspect-oriented code would be very useful to indicate or predict the presence of architectural problems in these systems.
- Some categories of aspect-oriented code smells tended to be more related to architectural smells than others. In particular, code smells associated with code damages (e.g. faults and complex refactorings) presented the highest coefficients of correlation with architectural smells. For instance, major architectural problems were observed in the presence of complex or redundant declarations and harmful aspect interdependencies; they were also responsible for faults and undesirable changes in the code. This means that architecturally-relevant code smells were, in fact, the most severe ones as they tend to negatively affect a wide range of quality attributes.
- Some architectural smells tended to co-occur with other architectural ones. That is, some architectural smells often caused the occurrence of other architectural smells. For example, we observed that architectural interfaces with generic and complex parameters [6] often might lead to inappropriate separation of architectural concerns [6]. This finding can warn designers that the occurrence of a smell might be a sign that other smells are also affecting the architecture modularity.

Therefore, our exploratory study provides new insights for software engineers and tool developers. They also assist researchers, for instance, to identify more specific hypotheses that should be tested in controlled experiments in the future. The paper also describes our strategies to mitigate some limitations or imperfections in our study (Section VI). We also emphasize the originality of our study with respect to related work (Section VII). Finally, we provide some concluding remarks and directions for future work (Section VIII).

II. ARCHITECTURAL SMELLS AND CODE SMELLS

This section characterizes architectural problems (Section A) and code smells (Section B) investigated in our paper. It also illustrates how code smell occurrences could be related to architectural smells (Section C) using a running example.

A. Architectural Smells

Modularity problems can be introduced in architectural decompositions through applying inappropriate design decisions. Architectural smells are recognized as a manifestation of architectural modularity problems since they comprise decisions that may negatively impact software design quality [8]. In particular, architectural smells are compositions of architecture-level elements that decrease the architecture modularity and, hence, may hinder software maintainability [6]. Architectural violations are also a manifestation of modularity problems in architectural designs [10]. At this point, we did not consider the violations because, differently from smells, they depend on the existence of the intended architectural information, which we might not have.

In order to select a reliable set of architectural smells, we identified how the observed architecture modularity problems in the target systems (Section III.C) were related to the smells documented in the published catalogs (Table I). This set of five architectural smells was selected because it comprises all smell instances identified by architects in the target systems. A complete definition of each architectural smell can be found in [6].

TABLE I. ARCHITECTURAL SMELLS

Architectural Smell	Definition
Ambiguous Interface (AI)	Interfaces that offer only a single, general entry-point into a component
Extraneous Adjacent Connector (EAC)	Connectors of different types are used to link a pair of components
Connector Envy (CE)	Components which encompass functionality that should be delegated to a connector
Scattered Parasitic Functionality (SPF)	Multiple components are responsible for realizing the same high-level concern and orthogonal ones
Component Concern Overload (CCO)	Components responsible for realizing several architectural concerns

Figure 1 presents a design slice of the HealthWatcher [11] architecture model. This system is one of the target systems used in our analysis (Section III.C). In this figure we omitted the inner members of each component and certain components such as *Transaction_Control*. Therefore, in this figure we are only representing a few architecture-level elements of the system's architecture. As an illustration, Figure 1 depicts an example of the Scattered Parasitic Functionality smell in the context of the HealthWatcher system used in our study [11]. As we can see, the architectural components *Concurrency_Control*, *Distribution_Manager* and *Persistence_Manager* are responsible for realizing the same high-level concern called Persistence. On the other hand, *Concurrency_Control* and *Distribution_Manager* are responsible for realizing orthogonal high-level concerns. These three components cannot be combined without creating a component that deals with more than one clearly-defined concern. *Concurrency_Control* and *Distribution_Manager* violate the principle of separation of concerns at the architectural design, as they are both responsible for realizing multiple orthogonal concerns. Therefore, these components are suffering for the Scattered Parasitic Functionality smell.

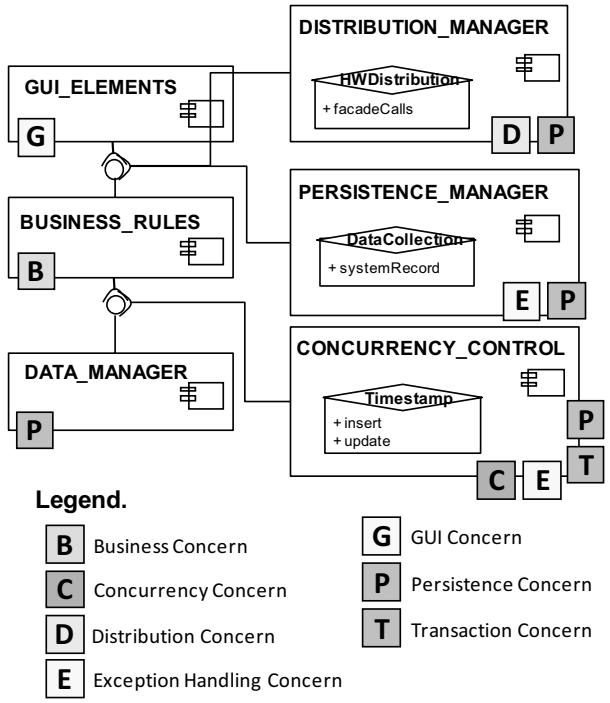

Figure 1. A design slice of HealthWatcher architecture model

B. *Aspect-Oriented Code Smells*

Aspect-oriented code smells [17] can affect different code units of an aspect-oriented program, such pointcuts, advice and aspects [15]. For example, God Aspect [17] is a code smell in which an aspect (i) is extensive and complex, (ii) has a high degree of coupling with other modules, and (iii) its pointcut expressions and other inner members are not cohesive. If the aspect is heterogeneous and too large, it may indicate that it would be better to decompose this aspect into other more modular aspects. For instance, the aspect *Timestamp* in Figure 1 was classified as God Aspect since it contains many lines of code, its inner members are non-cohesive and, additionally, it realizes several high-level concerns.

Table II summarizes the set of analyzed aspect-oriented code smells that occurred in this study. The complete list of analyzed code smells is available in [18]. These smells were selected because they represent all the aspect-oriented code smells identified by developers in the systems we analyzed. In this table the column "*Maintainability Impact*" indicates the smell's severity degree with maintainability problems in terms of faults (F) and refactorings (R). This impact was observed in a previous analysis of several evolving systems [3][18]. In the previous study, however, we have not analyzed how such code smells were related to architecturally-relevant problems (Section II.A). In the column "*Maintainability Impact*", we have marked with '-' those code smells that did not bring harm to the code in terms of corrective or perfective changes in later versions. The details of the fault and refactoring analysis process can be found in [3], while the details of the smells' definitions as well as their detection strategies [17] can be found in [18]. We elide these details here in the interesting of brevity.

TABLE II. ASPECT-ORIENTED CODE SMELLS

Code Smell	Description	Maintainability Impact
Anomalous Pointcut Definitions		
Anonymous Pointcut (AP)	The pointcut is directly defined in the advice signature.	12F* and 7R*
Duplicate Pointcut (DP)	Pointcuts that collect the same set of entities in the base code.	21F* and 17R*
God Pointcut (GP)	The pointcut has either a complex expression or picks out many entities in the source code.	23F* and 11R*
Idle Pointcut (IdP)	The pointcut do not match any entity in the base code.	19F* and 3R*
Redundant Pointcut (RP)	The pointcut has partial expressions equivalent to others that have already been defined.	18F* and 17R*
Aspect Definition		
Lazy Aspect (LA)	The aspect has either none or fragmented responsibility.	3R*
Undesirable Interdependencies		
Composition Bloat (CB)	Complex base computation that is advised by multiple aspects and leads to complex implementations in one or more aspects.	14F* and 6R*
Forced Join Point (FJP)	Entities in the base code that are only exposed to be used by aspects.	-
God Aspect (GA)	The aspect is realizing more than one concern.	31F* and 24R*

*#F and #R indicate the number of faults and refactorings associated with a code smell, respectively.

C. *Relating Architectural Smells and Code Smells*

We consider that a code smell C is correlated with an architectural code smell A when the implementation elements affected by C are the same implementation elements that realize an architectural element affected by A. A similar strategy was followed in other studies [29][30] to correlate code-level problems with architecture-level problems.

Figure 1 depicts an example of this correlation. The *Timestamp* aspect, classified as God Aspect, realizes several concerns (e.g. Concurrency, Persistence, Transaction and Exception Handling), condition which contributes to increase its internal complexity. Specifically, some of these concerns are shared by other components. At the same time, *Timestamp* is in charge of implementing *Concurrency_Control*, component which suffers from Scattered Parasitic Functionality. Additionally, this component was also classified as Component Concern Overload as its members (e.g. *Timestamp*) are realizing many concerns. This particular case also helps to illustrate the question of whether the same code smell occurrence (e.g. God Aspect instance) can be correlated with different architectural smells (e.g. Scattered Parasitic Functionality and Component Concern Overload).

III. STUDY SETTINGS

Our study aims to better understand the severity of aspect-oriented code smells from an architecture design perspective. Therefore, we performed an exploratory investigation to reveal whether and how aspect-oriented code smells (Section II.B) are related to problems that occur in systems' architectures, such as architectural smells (Section II.A). This Section describes a number of procedures that were followed to analyze the impact of aspect-oriented code smells on two evolving system's architectures.

A. Research Questions and Hypotheses

In the context of this study we have two research questions. The first research question investigates the relationship between aspect-oriented code smells and architectural smells. It is not obvious that aspects with smells are more prone to exhibit architectural smells than other aspects. The reason is that many other aspects are modified in the implementation of each software version. After we have addressed the first question, we can also check to what extent each category of code smell is related to architecture smells. In case we confirm this relationship through software evolution histories, it is likely that code smell C is a major contributor to architectural problems. Therefore, software developers should devote more attention to those code smells usually harm architecture modularity. In order to answer these questions, we defined two hypotheses as presented below.

$H1_0$: *Aspects with code smells are not significantly more related to architectural smells than smell-free aspects.*
$H1_A$: *Aspects with code smells are significantly more related to architectural smells than smell-free aspects.*

$H2_0$: *Certain code smells do not tend to be significantly more related to architectural smells than others.*
$H2_A$: *Certain code smells tend to be significantly more related to architectural smells than others*.

B. Variable Selection

In order to test our hypotheses, we have defined the following independent and dependent variables.

Independent Variables. For both hypotheses we have as many independent variables as kinds of code smells (see Table II). Each variable $C_{k,i,j}$ indicates the number of times an entity k has a code smell i in a version v_j. In particular, for H1 we have aggregated these variables into a Boolean variable $C_{k,j}$ indicating whether an entity k has at least one code smell of any kind. It is important to point out that all code smell occurrences used in testing these hypotheses were confirmed by developers.

Dependent Variables. Similar to the independent variables, we have as many dependent variables as kinds of architectural smells (Table I). Each variable $A_{k,i,j}$ indicates the number of times an entity k has the architectural smell i in a version v_j. For H1 we have aggregated these variables into a Boolean variable $A_{k,i}$ indicating whether an entity k has at least one architectural smell of any kind.

C. System Characteristics

The first major decision that had to be made in our study was the selection of the target systems. We chose 14 versions of 2 medium-size systems. The first one, called Health Watcher is a real Web-based software that allows citizens to register complaints about health issues in public institutions [7][11][26]. It was first versioned in 2000 and 10 other versions are also available at [11]. We selected all the 10 versions of HealthWatcher in our study. The second system called MobileMedia is a software product line for deriving applications that manipulate photos, videos, music and messages on mobile devices, such as mobile phones and PDAs. It was first developed in 2004 and 8 versions are available at the project's SourceForge.net repository [19]. We have concentrated on a subset of four MobileMedia versions as they were the only ones that realized widely-scoped changes [4]. We considered that a change was widely scoped when either architectural and code elements underwent many changes. Furthermore, the lists of faults and modifications for these four versions were previously documented in an independent fashion by others [4]. These reports helped us to conduct a reliable, in-depth analysis of the underlying causes for these observed problems. Table III summarizes characteristics of each target application, while Table IV lists the analyzed concerns for each target system.

TABLE III. CHARACTERISTICS OF TARGET SYSTEMS

	HealthWatcher	**MobileMedia**
Application Type	Web-based system	Product Line
# of Versions	10	8
Selected Versions	10	4
KLOC	60	32
# of Components	52	27
# of Aspects	195	59

These systems were chosen because they met a number of relevant criteria in addition to the factors mentioned above. First, the actual architectural design was recovered by developers involved in the maintenance of those systems. For instance, these systems followed different architectural decompositions (e.g., Layers, Model-View-Controller, and Aspectual Design). In addition, a detailed list of components, connectors, interfaces and their services were provided (and/or recovered) by architects. Second, the original architects and developers were available to validate the actual influence of the identified code smells. Only through developers opinion we are able to confirm whether the detected suspect is really a smell occurrence and, additionally, confirm whether it is influencing or not the architectural design. Third, these systems encompass a rich set of 'architecturally-relevant' aspects, such as implementations of design patterns (e.g. Adapter, Strategy, and Observer) and widely-scoped crosscutting concerns, including Concurren-

cy and Persistence. Fourth, they are non-trivial systems and their sizes (varying from 30 to 60 KLOC) are manageable for an in-depth analysis of code smells, as required in our study. The aspect-oriented implementation is of relevant complexity for this kind of study. They were implemented by more than 30 programmers with different levels of aspect-orientation skills. Finally, they have now a significant lifetime, comprising of several releases developed over the last ten years.

TABLE IV. ANALYZED CONCERNS IN THE TARGET SYSTEMS

Target System	Concerns
HealthWatcher (HW)	Concurrency, Exception Handling, Distribution, Persistence, Complaint, Disease, GUI, and 5 Design Patterns (Abstract Factory, Adapter, Command, Observer, State)
MobileMedia (MM)	12 Product-Line Features: Capture, Controller, Copy, Favorites, Labeling, Media, Music, Persistence, Photo, SMS, Sorting, and Video

D. Procedures for Data Collection

The assessment procedures of our study were based on the analysis of the recovery system's architecture and the system's implementation. In order to investigate the influence of aspect-oriented code smells on architecture designs, we count on the help of original developers and architects. The main phases of our study are described next.

Recovering the Actual Architecture. This phase was based on a semi-automatic process. We have used the tools Sonar [27] and Understand [31] to recover the actual architecture. These tools support architectural and code analyses in order to help developers to analyze and measure the quality of the system's implementation. Original architects and developers have worked together to fix possible defects of the architecture models generated by the tools and, therefore, make the analysis of architectural smells possible. For instance, they enriched the generated models with the high-level concerns that components are responsible for realizing. Figure 1 shows an abstract model fragment of some recovered architecture-level elements enriched with the relationship between components and high-level concerns. Specifically, the *Concurrency_Control* component is realizing Concurrency, Exception Handling, Persistence and Transaction concerns.

Mapping Architecture to Code Entities. This phase is concerned in explicitly tracing architecture-level elements (e.g. component, connectors or interfaces) to code-level elements (e.g. aspects or classes). Architects and developers mapped code-level elements to architecture-level elements. These mappings allowed us to trace the influence of a code smell on the introduction of problems in a system's architecture. In all the cases, the possible correlations of architectural smells and code smells had to be confirmed by the original architects and developers.

Identifying Architectural Smells. Due to the lack of tools, architectural smells were detected by architects based on a visual inspection of extracted architecture and a careful analysis of the code-level elements mapped to architecture-level elements. We also asked the original architects to indicate other architectural smells observed in the architecture design beyond those presented in Table I. This helped us to better judge whether and which code smells are good indicators of architectural problems. Finally, we performed complementary analyses in certain cases using the enriched models. For example, we investigated whether an architectural problem was introduced as a consequence of code smell occurrence or when a modification took place in the code element affected by the smell.

Identifying Occurrences of Aspect-Oriented Code Smells. Aspect-oriented code smells were identified using two complementary techniques: detection strategies [17] and visual inspection. In particular, the detection of relevant code information for detecting code smells was supported by tool called MuLATo [21]. The threshold values used in these strategies were derived from previous empirical studies reported in literature [17][22][28]. The complete list of the detection strategies and their corresponding thresholds are available in a supplementary web site [2]. In order to decide whether a code smell is actually an anomaly and, thus, to perform a reliable test of our hypotheses, we count on the help of at least three of the original developers. They were chosen based on their involvement in the systems' implementation process. We only consider as code smells all candidates that were confirmed by all the original developers.

E. Analysis Method

In order to reject H1, we test whether the proportion between aspects with code smells exhibiting at least one architectural smell and those that are not exhibiting any architectural smell significantly varies. We used Fisher's exact test [25], which checks whether a proportion vary between two samples. We also compute the odds ratio, OR [25], which indicates the likelihood of an event's to occurrence. The odds ratio is defined as the ratio of the odds p of aspects with code smells related to architectural smells (experimental group), to the odds q of aspects with no code smell related to architectural smells (control group). OR = 1 indicates that the likelihood of the event is equal in the two samples (entities with and without code smells); OR > 1 indicates that the event is more likely in the first sample (aspects with smells), while OR < 1 indicates that it is more likely in the second sample (aspects without smells).

In order to reject H2, we verified whether our samples were normally distributed through the Kolmogorov-Smirnov test [25]. Once we verified that the distribution is normal, we obtained evidence of the co-occurrence by applying a parametric Pearson's correlation coefficient [25] with a level of significance $\alpha = 0.05$, which means the level of confidence is 95%. It is obtained by dividing the covariance of the two variables by the product of their standard deviations. The Pearson correlation is +1 in the case of a

perfect increasing linear relationship, zero in the case there is no relationship, −1 in the case of a perfect decreasing linear relationship, and some value between −1 and 1 indicating the degree of linear dependence between the variables. Using Pearson's correlation coefficient, each of the smells was individually correlated to the others. It is important to highlight that all the relationships between code smells and architectural problems considered in this analysis were confirmed by developers and architects.

IV. Study Results

Tables V and VI present the number of architectural smells in the HealthWatcher and MobileMedia versions, respectively. In these tables, the "Other" column is associated with the total number of architectural smell occurrences, which were not related to code smell occurrences. The "Total" column presents the total number of architectural smell occurrences identified for each version.

TABLE V. Architectural Analysis in HealthWatcher

Versions	Architectural Smells					Other	Total
	AI	CCO	CEn	EC	SPF		
HW 1.0	1	-	0	1	1	5	**8**
HW 4.0	6	-	5	2	2	9	**24**
HW 7.0	8	2	8	2	4	11	**35**
HW 10.0	8	2	8	3	5	14	**40**
Total	**23**	**4**	**21**	**8**	**12**	**39**	**107**

TABLE VI. Architectural Analysis in MobileMedia

Versions	Architectural Smells					Other	Total
	AI	CCO	CEn	EC	SPF		
MM 4.0	7	2	1	2	1	6	**19**
MM 5.0	10	2	2	2	4	11	**31**
MM 6.0	12	2	2	2	6	14	**38**
MM 7.0	12	3	5	4	9	16	**49**
Total	**41**	**9**	**10**	**10**	**20**	**47**	**137**

Architectural smell occurrences behaved similarly to code smell occurrences [18] along the HealthWatcher evolution. That is, architectural and code smell occurrences varied only slightly from the previous to next version. On the other hand, we can see how the number of architectural smell occurrences increased from previous to next versions in MobileMedia. In addition, MobileMedia presented a higher number of architectural smell occurrences than HealthWatcher. The increase in the number of architectural smells is related to the scalability problems of aspect-oriented design in the presence of domain-specific non-crosscutting features [4]. As a consequence, the design started to become more fragile when the number of variabilities increased over time.

The HealthWatcher variabilities were naturally crosscutting, which were a good fit for the aspect-oriented design. However, some architectural smells were introduced in the context of feature compositions. Aspects had to share and duplicate information related to different features when they are not interested in that information in some cases. For instance, *DataCollection* and *Timestamp* share and duplicate information related to Persistence and Exception Handling concerns (Figure 1).

A. Code Smells and Architectural Smells

The results of the Fisher significance test on HealthWatcher and MobileMedia versions are reported in Tables VII and VIII, respectively. The first five columns of these tables present the number of code-level elements distributed into the following categories: (i) with architectural and code smells (A-C); (ii) with architectural smells but without code smells (A-NC); (iii) without architectural smells but with code smells (NA-C); and, (iv) without neither architectural or code smells (NA-NC). In the sixth column, we are comparing the obtained results with 0.05, which is the "critical" probability because there is no "tabulated" probability for this test. Finally, the last column reports the result of ORs when testing H1.

TABLE VII. Contingency Table and Fisher Results for HW

Versions	A-C	A-NC	NA-C	NA-NC	p-values	OR
HW 1.0	8	3	5	2	0.4	1.07
HW 4.0	12	2	4	6	**< 0.05**	7.5
HW 7.0	12	3	5	7	**< 0.05**	5.6
HW 10.0	13	4	2	6	**< 0.05**	9.6

TABLE VIII. Contingency Table and Fisher Results for MM

Versions	A-C	A-NC	NA-C	NA-NC	p-values	OR
MM 4.0	7	4	2	4	0.2	3.5
MM 5.0	7	3	1	6	**< 0.05**	14
MM 6.0	9	3	2	7	**< 0.05**	10.5
MM 7.0	16	5	8	10	**< 0.05**	4

Results of Table VII for HealthWatcher show that the proportions are significantly different, thus allowing us to reject H1. The odds for aspects with architectural and code smells are very high, that is twice times higher or more than for free-smell aspects. The H1 rejection and the ORs provide a posteriori concrete evidence of the negative impact of smells propagation to system's architecture. Developers should be worried of aspects with code smells as they are likely to be the source of architectural smells which considerably increase the cost of software project. In addition, the number of entities related to architectural and code smell occurrences varied slightly from the previous to next versions. This happened because the majority of the new code smells were introduced in the same entities which already contained code smells from previous versions. In some cases, existing smells were replaced by other smell instances as the software evolved. As the Fisher's test just considered the number of code-level elements, the number of smell occurrences will be considered when we discuss hypotheses H2 below.

Results of Table VIII for MobileMedia show that in the version 4.0 the proportions are not significantly different. The high p-value in this version can be explained by its size (i.e. it is the smallest version). On the other hand, the re-

maining results indicate that the architectural smell occurrences are significantly related to code smell occurrences (i.e. p-values lower than 0.05). In most cases the odds for architectural smell occurrences is at least two times higher than for code smell occurrences. We conclude that, in general, code smells are higher for code-level elements with architectural smells. Therefore, using the p-values, we can reject H1. We also could draw interesting observations from the modularity impact analysis, which are presented in the following section.

B. Correlation between Code and Architectural Smells

Tables IX and X show the results of linear Pearson's correlation analysis for one version of the two analyzed systems. Hopkins classifies a correlation value lower than 0.1 as trivial, 0.1–0.3 as minor, 0.3–0.5 as moderate, 0.5–0.7 as large, 0.7–0.9 as very large, and 0.9–1.0 as almost perfect [20]. We have highlighted in bold font the significant positive and negative correlations at the 0.05 level, respectively. For samples size of 4 and 10 and with $\alpha = 0.05$, the tabular Pearson's coefficient t are 2.92 and 1.86, respectively [25]. Because the computed coefficients are higher than the tabular t, the null hypothesis H2, is rejected. Hence, we can conclude that there is a significant correlation among certain architectural and code smells in the target systems. It is important to say that this correlation was also observed in the other versions.

TABLE IX. CORRELATION BETWEEN SMELLS IN HW VERSION 10

	AP	CB	DP	FJP	GA	GP	IdP	LA	RP
AI	**0.85**	0.78	0.53	0.17	**0.91**	**0.97**	0.48	-0.67	0.73
CCO	0.72	**0.82**	0.44	0.26	**0.87**	**0.94**	0.29	-0.8	0.54
CEn	0.59	0.34	0.58	0.49	0.62	**0.9**	-0.46	0.24	0.16
EAC	0.26	0.12	0.15	-0.4	0.1	0.5	-0.21	0.13	0.31
SPF	0.44	**0.89**	0.8	0.55	**0.94**	**0.95**	0.27	0.36	**0.98**

TABLE X. CORRELATION BETWEEN SMELLS IN MM VERSION 6

	CB	DP	FJP	GA	GP	IdP	LA	RP
AI	0.71	0.76	0.18	0.67	**0.98**	-0.21	-0.51	**0.88**
CCO	**0.9**	**0.87**	0.21	**0.89**	**0.92**	0.38	-0.6	0.42
CEn	0.41	0.53	0.04	0.62	**0.89**	0.16	0.02	-0.26
EAC	0.17	0.17	0.43	0.5	0.64	-0.24	-0.27	-0.38
SPF	**0.88**	**0.91**	-0.42	**0.95**	**0.93**	-0.31	0.62	-0.4

An observation of Tables IX and X reveals that certain code smell occurrences are strongly related to architectural ones. Therefore, we can suspect that these relationships are likely not to be coincident in these systems. The negative correlation coefficients identified for Redundant Pointcut in MobileMedia were caused by the fact that its occurrences were removed in version 6 (i.e. it did not accompany the increment of the architectural smells after version 5). As we can see in those tables, there are different types of code smells correlated to the same architectural smell. For instance, God Pointcut and Composition Bloat presented significant correlation with Scattered Parasitic Functionality in both systems. This happened because in some cases these code smells tend to occur simultaneously [18]. It is important to highlight that God Pointcut and Composition Bloat did not always occur together and their individual occurrences were also related to the Scattered Parasitic Functionality. We can also draw other interesting observations from the co-occurrence analysis, which are discussed in the next section.

V. DISCUSSIONS

Our previous analysis (Section IV) brings initial evidence that specific code smells are better indicators of architectural smells than others. This section discusses how certain code smells may lead to architectural smell occurrences.

A. Co-occurrences between Code and Architectural Smells

As Tables IX and X show, there are certain code smells that seem to be good indicators of architectural smells. This subsection discusses and illustrates some co-occurrences using examples extracted from the target systems.

Ambiguous Interface and Complex Advice. We have observed that there are code smells that seem to be good indicators of architectural smells. For instance, correlations between Ambiguous Interface and God Pointcut were observed since the first systems' versions, when Ambiguous Interfaces in the architectural design were traduced to God Pointcuts in the implementation phase. On the other hand, several occurrences of God Pointcut also entailed the emergence of the Ambiguous Interfaces in both systems. These observations were also confirmed using correlation coefficient (see Tables IX and X). These entities affected the aspect's interface by having a complex definition with a generic type as a parameter, which is used to perform a dynamic dispatch. Thereby, their implementation became more complex since it was necessary to filter the dynamic type of the parameter and then, used it to decide whether or not to dispatch to other methods.

```
public aspect DataCollection{
  ...
 Object around: call (SysRecord+.new()){
  class type = thisJoinPoint.getDeclaringType();
  ...
  if(type.equals(CompRecord.class)) {...}
  else if(type.equals(HealthUnit.class)) {...}
  else if(type.equals(MedicReport.class)) {...}
      <<other similar else-ifs were omitted>>
  ...
  return null;
 }
}
```

Figure 2. Example of Ambiguous Interface in the aspect code.

Figure 2 presents an example of the aforementioned coexistence. The pointcut defined as part of the implementation of the aspect *DataCollection* picks out all the calls of the constructors of *SysRecord* descendents. However, the pointcut is really interested in a subset of *SysRecord* descendents and some filters may be executed in order to perform the expected behavior, increasing the complexity of the advice implementation. This observation may indicate

that the pointcut analyzability and understandability might be reduced because it does not reveal which *SysRecord* descendents it is interested in. Even worse, its advice implementation behavior is related to multiple architecturally-relevant concerns leading to other architectural smells, which are discussed later. In some cases, the number of Ambiguous Interface instances increased due to simultaneous occurrences between God Pointcut and Redundant Pointcut.

Composition Bloat and Scattered Parasitic Functionality. Another interesting observation is the relationship between Composition Bloat and Scattered Parasitic Functionality, which was confirmed for both systems in Tables IX and X. These correlations emerged in later systems' versions due to the incremental addition of features. Composition Bloat instances expose information to be shared and advised by different entities in the code. This smell may force entities to live with information related to other concerns that they are not interested in, akin to a parasite. Even worse, there are other entities responsible for the modularization of the "parasite" concern information. Therefore, such cases are considered as instances of Scattered Parasitic Functionality. For instance, Figure 3 shows how the aspect *PhotoAndMusicAspect* has to deal with information related to Photo and Music features when in the ideal (refactored) solution this information would be moved out to aspects responsible for their modularization (e.g. *PhotoAspect* and *MusicAspect*). This situation is further aggravated by the relationship between Composition Bloat and code duplication (i.e. clones) where it is expected that this situation occurs simultaneously with other aspects.

```
public aspect PhotoAndMusicAspect{
  ...
  pointcut startApp(MainUIMidlet mid):
   execution( public void MainUIMidlet.startApp())
   && this(mid);

  after(MainUIMidlet mid): startApp(mid) {
   BaseController imgCtr = mid.imageController;
   AlbumData imgModel = mdlt.imageModel;
   BaseController mCtr= mdlt.musicRootController;
   AlbumData musicModel = mdlt.musicModel;
          <<11 lines of code removed>>
   selectcontroller.setNextController(imgCtr);
   selectcontroller.setImageAlbumData(imageModel);
   selectcontroller.setImageController(imgtCtr);
   selectcontroller.setMusicAlbumData(musicModel);
   selectcontroller.setMusicController(mCtr);
   mainscreen.append("Photos");
   mainscreen.append("Music");
  }
}
Legend:
 Photo Feature  Music Feature
```

Figure 3. Example of Scattered Parasitic Functionality.

Connector Envy and God Pointcut. We have also observed that God Pointcut was usually an indicator of Connector Envy instances in HealthWatcher and MobileMedia. As we have mentioned in Section II the pointcut-advice complexity is one of the alternative conditions for the manifestation of a God Pointcut. In some occurrences this complexity increased in successive versions. The advice increasingly performed communications with other modules as part of their implementation. It is important to point out that Connector Envy is not only related to God Pointcut instances. We also observed occurrences where the entity (class or aspect) encompasses functionality that should be delegated to a connector. In those cases, the Connector Envy occurrence was not motivated by any code smell occurrence.

God Aspect and Component Concern Overload. We have observed that another noticeable characteristic of God Aspects is the total number of concerns they are dealing with. When the aspect's inner members are not very cohesive (Section II), they may deal with different kind of information (high-level concerns). Additionally, in later systems' versions it was observed that the aspect's internal complexity increased due to it was realizing different high-level concerns. This situation led to God Aspects introduced different concerns in the architecture-level elements that they were responsible for implementing. Consequently, the architectural component suffered from Component Concern Overload smell. However, this kind of observation was not only associated with single instances of God Aspects. An interesting finding emerges from analyzing groups of God Aspect instances. For instance, when a group of God Aspects is implementing the same architectural component A, it may indicate that A suffers from Component Concern Overload. Of course, this occurs when the group of God Aspects is realizing, in total, many high-level concerns.

Architectural Smells, Faults and Instabilities. Surprisingly, the code smells with high correlation coefficients are those that presented the highest rates of faults and instabilities in a previous study (see Table II). For instance, code smells related to complex program elements (e.g. God Aspect and God Pointcut) seemed to be good predictors of architectural smells. They were also related to several architectural smells, such as Ambiguous Interface and Scattered Parasitic Functionality. This is a relevant observation because it shows that existing catalogs of architectural smells may be used to manage and early predict the impact degree of harmful code smells on software maintenance. In addition we have observed that existing smells did not relate highly to faults and instabilities, but they presented high correlation coefficients with architectural smells. This was the case for the Composition Bloat smell, which was seldom refactored in later versions. On the other hand, other code smells had a big impact on further faults and instabilities study, but they presented a low correlation coefficient with architectural smells. For instance, Idle Pointcut was responsible for 9 out of 25 faults in HealthWatcher. However, it presented a low correlation coefficient with architectural smells in all the HealthWatcher and MobileMedia versions. These observations are interesting because they likely allow to improve

the ranking of the most severe code smells and, therefore, contributing to the derivation of more effective refactoring strategies.

B. Simultaneous Occurrences of Architectural Smells

We also observed that certain architectural smells occurred simultaneously, i.e. a pair of architectural smells was affecting the same architectural element. This information can alert code and architecture reviewers that the occurrence of a smell might be a sign that other smells are also affecting the code and architecture. This is particularly handy when one of the co-occurring smells is not easy to identify. For instance, in both systems some Scattered Parasitic Functionality instances were related to Ambiguous Interface instances. This happened because some methods have a complex definition with a generic type as parameter, condition for Ambiguous Interface instances. At the same time, their implementation becomes more complex since it is necessary to filter the dynamic type of the parameter according to the different high-level concerns. By dealing with these different high-level concerns, the architecture-level element was also considered a Scattered Parasitic Functionality instance. Figure 2 also presents an example of this coexistence where the piece of code is aware of many different business classes. Consequently, it is realizing a high tangling of architecturally-relevant concerns (e.g. Complaint and Disease).

Similar situations were observed with respect to Connector Envy and Ambiguous Interfaces in both systems. These observations indicate that generic and complex entry points often might lead to other architectural smells. Finally, it was also observed that Component Concern Overload instances might be associated with Scattered Parasitic Functionalities. This situation occurs when an architectural component A is realizing many high-level concerns and, additionally, some of these concerns are just realized by A, while others are shared by multiple components. In order to confirm or reject the correlation between different kinds of architectural smells a further exploratory study should be performed

We classified the analyzed code smells as indicators of architecture problems according to their impact on the architecture designs (see Table XI). We classified the code smell C with a percentage of occurrences related to architectural smells lower than 0.3 as *Weak*, 0.3-0.4 as *Medium*, 0.4-0.5 as *Strong* and higher than 0.6 *Very Strong*. The goal of this classification is just to characterize code smells in different bands. Other thresholds could eventually be used. However, at this point the most important is to verify which code smells presented higher correlation.

TABLE XI. CODE SMELL CLASSIFICATION

Code Smell	Architectural Impact
God Aspect	Very Strong
God Pointcut	Very Strong
Composition Bloat	Strong
Duplicate Pointcut	Medium
Redundant Pointcut	Medium
Idle Pointcut	Weak
Forced Join Point	Weak
Lazy Aspect	Weak
Anonymous Pointcut	Weak

VI. STUDY LIMITATIONS

Some limitations or imperfections of our study can be identified and are discussed in the following.

Construct Validity. Threats to construct validity are mainly related to possible errors introduced in the identification of code and architectural smell instances. We are aware that detection strategies, manual inspection and other mechanisms to identify code and architectural smells can introduce imprecision. However, we ameliorated this threat by using and double-checking results collected from other studies [18] and involving the original developers. In addition the architectural smells were systematically identified by original architects, who had previous experience on the detection of architectural smells in other systems. The correlation analysis between code smells and architectural problems were also validated with the original architects and developers.

Conclusion Validity. We have two issues that threaten the conclusion validity of our study: the number of evaluated systems and the evaluated smells. Four versions of MobileMedia and ten versions of HealthWatcher were used for the purposes of this study, totaling 14 analyzed versions. Of course, a higher number of systems is always desired. However, the analysis of a bigger sample in this kind of study could be impracticable. The relationship between code smells and architectural problems need to be confirmed using manual inspection in order to analyze only those co-occurrences that seem to be not merely coincident. We need to study the causality between smells and architectural problems. Also, it is scarce the number of systems with all the required information and stakeholders available to perform this study. Then, our sample can be seen as appropriate for a first exploratory investigation [13] for raising hypotheses that can be further tested in more controlled replications. In addition, our evaluation was also enough to provide initial statistically-relevant evidence for the two hypotheses. Related to the second issue (completeness of architectural and code smells), our analysis was concerned with at least all the document aspect-oriented smells. There are documented smells, which were not selected due to their limitations and imperfections (e.g. Junk Material [28]) whereas all the architectural smells reported in the literature were considered in our study.

Internal and External Validity. The main threats to internal and external validity are the following. First, the level of experience of the developers in the systems' implementations could be an issue. In order to limit such a threat we used systems that were developed by more than 20 programmers with different levels of AspectJ skills. The main threat to external validity is related to the nature of the evaluated systems. In order to minimize this threat we have tried to use applications with different sizes and that were implemented using different architectural styles and environments.

However, we are aware that more studies involving a higher number of systems should be performed in the future.

VII. Related Work

Some studies proposed refactoring techniques to improve modularity of aspect-oriented systems. Iwamoto and Zhao [12] investigate the impact of existing OO refactorings [5] on aspect-oriented programs. Furthermore, Monteiro and Fernandez [20] propose a collection of 28 aspect-oriented refactorings, covering both the extraction of aspects from OO legacy code and the subsequent tidying up of the resulting aspects.

Few research works are focused on the characterization of code smells in aspect-oriented systems [18][22][28]. Piveta [22] defined 5 smells occurring in AO systems. Srivisut [28] extended Piveta's work in order to define metrics that help to identify the code smells proposed by him. However, these works only focus on the definition of code smells, without empirically studying their behavior along software evolution. Macia et al. [18] defined a new suite of six code smells, which complement previously-documented ones. The study revealed that: (i) the number of faults related to code smells is higher than the number of faults that are not related and, (ii) certain code smells require more sophisticated refactoring strategies, though not frequently performed by programmers. This work complements the previous ones because we analyzed whether and how the code smell occurrences can impact on the architectural design. Specifically, we studied the ripple effect of code smells making system's architectures more complex, difficult to change and rigid.

VIII. Concluding Remarks

In this study we found that certain categories of aspect-oriented code smells were good indicators of architectural problems in the analyzed systems. In addition, our study also revealed that code anomalies affecting the composition of modules (e.g. pointcuts and inter-type declarations) in an aspect-oriented program are usually the source of architecturally-relevant smells. For instance, anomalies in pointcuts picking out methods scattered among classes, realizing different architectural components, usually impact the modularity of architecture decompositions. The effects of such architecturally-relevant code smells are further aggravated because they might be observed in later stages of software development; in these circumstances, they are more difficult and expensive to fix.

Furthermore, our study provided some findings that can help developers to build more effective tools for identifying more severe code smells. For instance, some architecturally-relevant code smell occurrences cannot be detected and prioritized if architectural decisions are not somehow traced and mapped to the source code, and used by code-level smell detection tools. Current tools cannot decide whether (or not) relationships between code-level elements are associated with architectural smells. They cannot decide either whether an aspect is modularizing concerns scattered and shared among different architecture-level elements.

Finally, it is important to highlight that we have analyzed only 14 releases of two systems. Then, the cause-effect relationships of aspect-oriented code smells and architectural smells should be tested in broader contexts in the future.

References

[1] Bergmans, L. et al Composition Filters: Extended Expressiveness for OOPLs. In Proc of the 7th OOPSLA, 1992.

[2] Code smells in AOP: evaluation, April/2010. http://www.inf.puc-rio.br/~ibertran/codesmells.

[3] Fabiano, F. et al. An exploratory study of error-proneness in evolving Aspect-Oriented Programs. In: Proc. of 25th OOPSLA, USA, 2009.

[4] Figueiredo, E et al. Evolving software product lines with aspects: An empirical study on design stability. In Proc. of the 30th ICSE, 2008.

[5] Fowler, M. et al. Refactoring: Improving the design of existing code. Addison-Wesley, 1999.

[6] Garcia, J.; et al. Identifying Architectural Bad Smells. In Proc. of 13th CSMR. 2009.

[7] Greenwood, P. et al. On the impact of aspectual decompositions on design stability: An empirical study. In Proc. 21st ECOOP, 2007.

[8] Griswold, W. et al. Modular Software Design with Crosscutting Interfaces. IEEE Software, Special Issue on AOP, 2006

[9] Hohenstein, U. Improving the performance of database applications with Aspect-Oriented Programming. In Proc.of the 5th AOSD, 2006.

[10] Hochstein, L. and Lindvall, M. Combating architectural degeneration: A survey. Info. & Soft. Technology July, 2005.

[11] HW http://www.comp.lancs.ac.uk/~greenwop/tao

[12] Iwamoto, M. and Zhao, J. Refactoring aspect-oriented programs. In Proc. of 4th AOSD Modeling with UML Workshop, 2003.

[13] Kitchenham, B et al. Evaluating guidelines for empirical software engineering studies. ISESE pp 38-47, 2006

[14] Kiczales, G.,et al. Aspect-oriented programming. In Proc. of 11th ECOOP, 1997.

[15] Kiczales, G. and Mezini, M. Aspect-oriented programming and modular reasoning. In Proc. of the 27th ICSE, 2005.

[16] Kulesza, U. et al. Implementing Framework Crosscutting Extensions with EJPs and AspectJ. Technical Report, PUC-Rio, Brazil, 2006

[17] Lanza, M and Marinescu, R. Object-Oriented Metrics in Practice. Springer, 2006.

[18] Macia, I. et al. An Exploratory Study of Code Smells in Evolving Aspect-Oriented Systems. In Proc. of the 10th AOSD, 2011.

[19] MM: http://sourceforge.net/projects/mobilemedia/

[20] Monteiro, M.P. and Fernandez, J.M. Towards a catalog of aspect-oriented refactorings. In Proc.of the 4th AOSD, USA, 2005.

[21] MuLATo tool, http://sourceforge.net/projects/mulato/ (3/08/2009)

[22] Piveta, E.K. et al. Detecting bad smells in AspectJ. Journal of Universal Computer Science, vol. 12, 2006.

[23] Prehofer, C. Feature-oriented programming: A fresh look at objects. In Proc. of the 11th ECOOP, 1997

[24] Schaefer,I. et al. Delta-oriented Programming of Software Product Lines. In Proc of the 14th SPLC, 2010..

[25] Sheskin, D. Handbook of Parametric and Nonparametric Statistical Procedures. Chapman & All, 2007.

[26] Soares, S. et al. Implementing distribution and persistence aspects with AspectJ. In Proc. of 17th OOPSLA. 2002.

[27] Sonar: http://docs.codehaus.org/display/SONAR/

[28] Srivisut, K. and Muenchaisri, P. Bad-smell Metrics for Aspect-Oriented Software. In Proc. of the 6th ICIS, Canada, 2007.

[29] Wong, S. et al. Detecting Design Defects Caused by Design Rule Violations. In Proc of the 18th ESEC/FSE, 2010.

[30] Wong, S. et al. Detecting Software Modularity Violations. In Proc of the 33rd ICSE, 2011.

[31] Understand tool, http://www.scitools.com/.