Identifying Architectural Problems through Prioritization of Code Smells

Santiago Vidal*, Everton Guimaraes[†], Willian Oizumi[¶], Alessandro Garcia[‡], Andrés Díaz Pace* and Claudia Marcos[§]

*ISISTAN-CONICET, Argentina. Email: {svidal, adiaz}@exa.unicen.edu.ar

[†]UNIFOR, Brazil. Email: eguimaraes@unifor.br

[¶]IFPR, Brazil. Email: willian.oizumi@ifpr.edu.br

[‡]PUC-Rio, Brazil. Email: afgarcia@inf.puc-rio.br

[§]ISISTAN-CIC, Argentina. Email: cmarcos@exa.unicen.edu.ar

Abstract—**Architectural problems constantly affect evolving software projects. When not properly addressed, those problems can hinder the longevity of a software system. Studies have revealed that a range of architectural problems are reflected in source code through two or more code smells. However, a software project often contains thousands of code smells and many of them have no relation to architectural problems. Thus, developers may feel discouraged to identify architectural problems if they are not equiped with means to focus their attention in a reduced set of locations in their system to start with. However, state-of-the-art techniques fall short in assisting developers in the prioritization of code smells that are likely to indicate architectural problems in a program. As a consequence, developers struggle to effectively focus on (groups of) smells that are architecturally relevant, i.e., smells that contribute to a critical design problem. This work presents and evaluates a suite of criteria for prioritizing groups of code smells as indicators of architectural problems in evolving systems. These criteria are supported by a tool called *JSpIRIT*. We have assessed the prioritization criteria in the context of more than 23 versions of 4 systems, analyzing their effectiveness for spoting locations of architectural problems in the source code. The results provide evidence that one of the proposed criteria helped to correctly prioritize more than 80 (locations of) architectural problems, alleviating tedious manual inspections of the source code vis-a-vis with the architecture. This prioritization criteria would have helped developers to discard at least 500 code smells having no relation to architectural problems in the analyzed systems.**

I. INTRODUCTION

Software systems usually suffer from architectural problems introduced either during development or along their evolution. Several systems have been restructured with high costs or even discontinued due to the constant occurrence of architectural problems [1], [2], [3]. Many architectural problems occur when one or more components of a system are violating design principles or rules [2]. These violations negatively affect the maintainability and other quality attributes of a system [2], [4]. Typical examples of architectural problems are Fat Interface and Unwanted Dependency between components [5], [1]. The former violates the principle of separation of concerns [2], while the latter violates a dependency rule in the system's architecture [1]. Both of them often negatively affect software maintainability and performance [2].

Unfortunately, the identification of locations in a system that likely indicate architectural problems is time-consuming and cumbersome for several reasons. This task generally requires the analysis of both architectural documentation and the realization of architecture decisions in source code. It is hard for a developer to effectively explore these two types of information altogether in order to uncover possible spots of architectural problems. Architecture documentation is often not well documented and scarce. Even when architecture information is available, it is often not detailed enough to help developers to reveal architectural problems[2], [4], [6]. Thus, developers need to resort to hints in the source code that indicate the presence of architectural problems. An ocurrence of certain types of well-known code anomalies (i.e. *code smells*) [7] may provide helpful, albeit partial, hints of the location of architecture problems in a system [4] (Section II). Classical examples of code smell types often related to architectural problems are Long Method, God Class and Feature Envy [7].

A wide range of architectural problems are often realized by a subgroup of code smells scattered in the source code, which makes their location even more challenging [4], [6]. Even for small systems, developers would need to analyze hundreds of smells and infer their likelihood of indicating the manifestation of an architectural problem. As those systems evolve, the number of smells tend to grow across system versions, thereby further obscuring the location of architectural problems in a program. In such situations, developers do not know where to focus on, i.e. which few groups of code smells can be used a starting point for locating architectural problems in the source code. We argue that a practical strategy here is to prioritize groups of code smells according to their criticality to the system architecture, that is, their ability to point out architectural problems. Unfortunately, existing work (Section II) does not support developers on automatically prioritizing code smells to reveal architectural problems, thereby discouraging them to locate such symptoms in their programs. In particular, they do not establish any criteria to guide developers on locating a first set of architectural problems in their program structure.

In this context, we propose and implement a suite of three scoring criteria for prioritizing code smells (Section IV). The goal is to constrain the search of developers by reducing the amount of candidates of program locations possibly containing architectural problems in their systems. Our criteria are centered on the notion of *agglomerations* of code smells,

a concept proposed and defined in our previous work [4], [6]. Agglomerations are groups of inter-related code smells (e.g., syntactically-related code smells within a component) that likely indicate together the presence of an architectural problem (Section III). However, our previous work [4], [6] did not address the key challenge of prioritizing smell agglomerations that indicate architectural problems. In fact, many smell agglomerations are not related to architectural problems [4], [6]. In order to rank smell agglomerations according to their impact on the architecture, the criteria proposed in this article consider both the implementation and architectural information that is available, including the version history of a system (Section IV). The goal is to assist developers in: (i) encountering agglomerations that more likely indicate architectural problems, and (ii) for each architectural problem, identifying its full extent in the source code by inspecting the group of smells comprising a prioritized agglomeration, while discarding irrelevant smells.

We assess how each of our three proposed criteria helps developers to locate symptoms of architectural problems in source code, while keeping aside irrelevant (agglomerated and non-agglomerated) code smells. Our study considered more than 23 versions of 4 Java systems (Section V). Furthermore, one of the systems had a very large size (1400 classes and 1800 code smells) so as to test the scalability of our approach. Our results (Section VI) suggest that the use of one of the criteria can consistently and accurately indicate several architectural problems. They helped to correctly locate more than 80 architectural problems in our top-7 rankings of the 4 systems, alleviating tedious manual inspections of the source code vis-a-vis with the architecture. Moreover, this prioritization criterion would have also helped developers to discard at least 500 code smells having no relation to architectural problems in the analyzed systems. At the end, we reflect upon these findings and present concluding remarks (Section VII).

II. RELATED WORK

As far as we are aware of, our investigation represents the first effort in supporting the prioritization of architecturally-relevant code smells. Architectural problems have been the focus of various recent studies (e.g. [2], [8], [3]), including a catalog documenting them [2]. However, existing work rarely provides support for locating and prioritizing such problems in the source code. In addition, they tend to assume that a detailed architecture documentation is always available, which is rarely the case. Case studies also mostly focus on reporting the severe impact of architectural problems, such as Fat Interfaces and Unwanted Dependencies, on the longevity of industrial software systems (e.g. [3]). In [9], five types of architectural problems are presented and formalized. These problems can be detected in Design Structure Matrices (DSMs). Two of such problems are based on the analysis of history information, while the other three problems are inferred from the source code structure. The authors show a correlation of these problems with both high frequencies of bugs and major maintenance effort. However, none of these works assists developers in the prioritization of code smells with the goal of identifying and prioritizing architectural problems. We have observed in our study that the use of project history information alone, albeit useful, is of limited applicability (Section VI-B).

Other studies have investigated the impact of *single code smells* throughout system evolution [10]. They analyzed whether the number of smells increased or decreased over time, and how often they resulted in code refactorings intended to improve the system design. In other recent studies [11], [12], the authors report on the relevance of code smells for the identification of architectural problems. As main findings, they observed that tracking of individual code smells without regarding the occurrence of other smells do not suffice to assist developers in revealing architectural problems. Each smell only provides a partial realization of an architectural problem. Each architectural problem tended to be realized by seven or more code smells. In addition, a very high proportion of individual smells (i.e. those smells detached from other smells in the system) did not impact on the architecture.

Only a few studies have gone beyond and looked at groups of code smells as indicators of architectural problems. The concept of agglomerations was presented in our recent work [4] to capture a group of inter-related code smells. The work confirmed that agglomerations are much better indicators of architectural problems than non-agglomerated code smells. However, the results also showed that several agglomerations are not related to any architectural problem. As a result, there is a pressing need for supporting the prioritization of agglomerations that are related to architectural problems. In [13] we briefly discuss two criteria to prioritize agglomerations driven by concerns and the version history. However, we do not explain all the criteria proposed here and how these criteria could be computed. In that short paper, we did not conduct a proper case study either. In [14] the authors only documented a few relationships among code smells that may be related to four architecture problems. However, to the best of our knowledge, no work has yet investigated strategies for prioritizing groups of code smells. In this way, this is the first work that proposes, implements, and evaluates a criteria for prioritizing agglomerations aimed at locating architectural problems. Existing tools do not provide any support for these tasks [14], [15], [16], [12]. They normally only consider the source code structure, disregarding architectural information. Even worse, the users also cannot use, define or customize their own criteria for prioritizing code-smell agglomerations. We address all these gaps in the study reported in this article.

III. AGGLOMERATIONS AS POINTERS TO ARCHITECTURAL PROBLEMS

In the context of our work, we focus on architectural problems [2] that represent violations of design principles or rules [2], [17]. We mainly target architectural problems affecting the modular decomposition of a system into components and their interfaces, i.e., maintainability-related problems [2]. Based on previous empirical findings [6], [4], the premise is that groups of code smells, so-called *agglomerations*, are normally associated with several architectural problems.

A. Illustrative Example

In order to illustrate the relation between architectural problems and agglomerations, let us consider the example of Fig. 1. The left side of the figure shows a fragment of the component structure of the MobileMedia architecture – a system for managing photos, music and videos in mobile
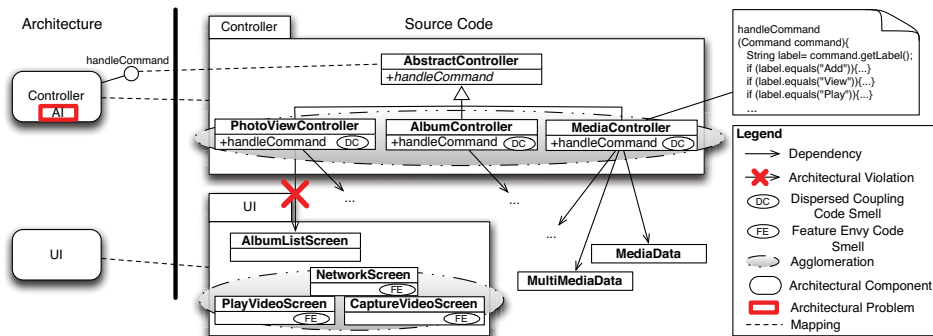
Fig. 1: Example of a code-smell agglomeration related to architectural problems

devices. Components *Controller* and *UI* are mapped to separate Java packages in the implementation, each one containing several classes (right side). If a static analysis tool is run over the implementation of these components, the developer will receive a list of more than one hundred code smells. Then, it may not be clear which code smells should be the focus of her attention as candidates for revealing architectural problems in those components, which prevents her from performing effective maintenance or refactoring activities.

Architectural Problems. This example reveals three architectural problems. First, the *Controller* component is mainly realized by the class hierarchy rooted at class *AbstractController*, which is responsible for handling different commands through the *handleCommand* method. After a broad look at all the anomalous implementations of method *handleCommand* and their callers, the developer realizes, based on her experience, that there is an overload of responsibilities, which leads to two architectural problems, called Fat Interface and Ambiguous Interface [2]. These problems mean that *Controller*, as an architectural component, provides several non-cohesive services (Fat Interface) that are not properly exposed in its interface *handleCommand* (Ambiguous Interface). Note that the problem is not the controller itself or its object-oriented materialization in terms of an abstract class with many concrete classes, but rather the decision of having only one single controller (at the architecture level), which can generate ripple effects to other components and their implementations if the controller logic has to be changed. Second, the call from class *PhotoViewController* to class *AlbumListScreen* leads to a usage dependency between packages *Controller* and UI, which is not allowed by the component architecture. This violation is indicated in the architecture (left side) by the absence of arrows between the two components.

In these examples, a possible way for a developer to identify the architectural problems is by reasoning about the architecture documentation and checking candidate problems in the source code. Unfortunately, developers are usually overwhelmed by these tasks because, even with tool support, it is hard to effectively explore all the available information and all code smells in order to uncover architectural problems. In these cases, the developer needs to turn her attention to the (partial or full) realization of architectural problems in the source code. Along this line, she might discover that the implementations of *handleCommand* in the subclasses of

AbstractController are simultaneously affected by the code smell called Dispersed Coupling (DC), which is a method that calls various methods of several classes. That is to say that the subclasses of *AbstractController* generate dependencies on many other classes. Since there are several DC smells within the *Controller* package, this group of smells is considered as an agglomeration. Therefore, this package-level agglomeration is a sign of (potential) architecture decay [2], which in this case affects the *Controller* component.

However, the developer cannot be fully sure about the architectural problem exposed by the agglomeration of DC smells, as it could be a false positive. Other agglomerations can be present nearby, as it is the case of a group of instances of the smell called Feature Envy (FE) in package *UI*, which corresponds to the *UI* component. FE is a smell representing a class that is more interested in accessing data from other classes (instead of using its own data), which often indicates a poor assignment of responsibilities. Things get further complicated for the developer because agglomerations normally vary from a system version to another. These two factors (false positives and variations over time) motivate our interest in the definition of criteria for prioritizing agglomerations.

B. Detecting Individual Code Smells

Existing catalogs of code smells define guidelines to identify single smells and how to provide tool support for their detection [7], [18]. In this work, we use the *JSpIRIT*[1] tool for that purpose. *JSpIRIT* is an Eclipse plugin for detecting code smells of a (Java-based) system and ranking them according to different criteria [12]. Fig. 2a shows the view of *JSpIRIT* that lists the code smells for a system. Currently, *JSpIRIT* supports the identification of 10 single smells (such as Feature Envy, Brain Method, and Disperse Coupling)[2] [12] following the detection strategies presented in Lanza's catalog [18].

C. Types of Agglomerations

The original version of *JSpIRIT* was extended to support the detection of agglomerations. Fig. 2b shows a list of agglomerations detected on the basis of the smells of Fig. 2a. Since our work focuses on architectural information regarding

---

[1]https://sites.google.com/site/santiagoavidal/projects/jspirit
[2]A complete list of the supported code smells can be found at http://www.les.inf.puc-rio.br/opus/priorizatingAgglomerations/

(a) Code smells

(b) Agglomerations

Fig. 2: *JSpIRIT* outputs

static code structures, we worked with agglomerations within the scope of architectural components. For our case-studies (Sections V and VI), we assumed a relation (or mapping) between an architectural component and its realization as a Java package in the code. However, developers can flexibly establish other kinds of mappings between components and packages/classes in a program. We are mainly interested in two particular patterns of agglomerations:

- **Smells within a component.** This grouping pattern identifies code smells that are implemented by the same architectural component. Specifically, we look for one single component with: (i) code smells that are syntactically related, or (ii) code elements infected by the same type of code smell. Two classes are syntactically related if at least one of them references the other one. Fig. 1 showed an example of this kind of agglomeration where different classes in package *UI* are affected by the Feature Envy (FE) smell.

- **Smells in a hierarchy.** This grouping pattern identifies code smells that occur across the same inheritance tree involving one or more components. We only consider hierarchies exhibiting the same type of code smell. The rationale is that a recurring introduction of the same smell in different code elements might represent a bigger problem in the hierarchy. An example of this agglomeration is the *AbstractController* hierarchy in Fig. 1 whose subclasses are affected by Dispersed Coupling (DC) smells.

A more complete description of the above agglomerations can be found in [6]. Certainly, other types of agglomerations are possible, as reported in [4], but they are related to other architectural problems not addressed in this article.

IV. PRIORITIZATION APPROACH

In this section, we present three criteria to prioritize agglomerations by means of scoring criteria. The proposed criteria were implemented in *JSpIRIT*. Our hypothesis is that these criteria are useful for prioritizing those agglomerations with high chances of spotting architectural problems. In this way, a criterion can be seen as a function:

$$criterion_A(agglomeration_B) = score_{A,B}$$

where the score for an agglomeration B given by a criterion A is a value between 0 and 1. The score value indicates how critical the agglomeration is for the system architecture (0=not
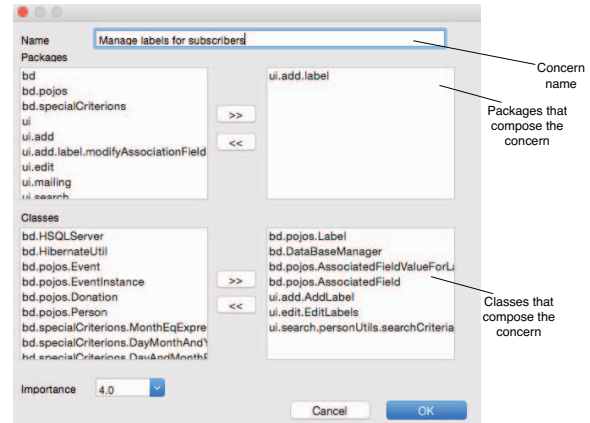

Fig. 3: Wizard to provide concern mappings in *JSpIRIT*

critical, 1=very critical). In particular, we began working with a criterion based solely on architectural information (namely, architectural concerns). Later, we developed two additional criteria based on the combination of the code versions and architectural information (namely, history of changes and *agglomeration cancer*). In our context, architectural information refers both to the component structure (as used for the detection of the agglomerations) and the architectural concerns.

A. Architectural Concerns

This criterion analyzes the relationship between an agglomeration and an architectural concern. An architectural concern is some important part of the problem (or domain) that developers aim at treating in a modular way [8], such as graphical user interface (GUI), exception handling, or persistence. For example, in Fig. 1, the subclasses of *AbstractController* (along with other system classes) all address a concern called *PhotoLabelManagement*. This criterion was adapted from [11] where it is used to rank single code smells. The rationale behind this criterion is that an agglomeration that realizes several concerns could be an indicator of an architectural problem.

The *JSpIRIT* tool offers a simple interface to specify concerns (Fig. 3). Specifically, the developer must provide a concern name and select the system packages and classes the concern maps to. To compute the ranking score of a given agglomeration, we count the number of concerns involved in that agglomeration. A concern is involved in an agglomeration if the agglomeration is located in a class mapped by the

concern. At last, we normalize the values to obtain scores between [0..1]. To do so, the highest number of concerns affecting a single agglomeration is used. For example, given four agglomerations A1, A2, A3, and A4 that involve 0, 1, 2 and 3 concerns respectively, the highest number of concerns per agglomeration is 3. Then, the agglomeration scores will be $0/3 = 0.0$, $1/3 = 0.33$, $2/3 = 0.66$ and $3/3 = 1.0$.

Certainly, the specific mappings of concerns to program elements affect the results of this criterion. Furthermore, as the implementation evolves, the mappings might need to be adjusted. Existing feature-location tools, such as Mallet [19] and XScan [20], can be used here to derive concern mappings automatically and with high accuracy according our recent experience on using them [4].

B. History of Changes

This criterion analyzes the stability of the classes in which the code smells (of an agglomeration) are located. By looking at the "stability" of the main classes of these smells, we want to check whether the agglomeration is in a component or class hierarchy that is usually modified. Our assumption is that agglomerations appearing in classes that changed often should have a higher score. Note that this notion of stability relies not only on the actual architectural information (e.g. the agglomeration affecting a particular component), but also on information from the history of class changes.

To calculate the score of an agglomeration we use the LENOM metric [10]. We previously used this metric to rank single code smells [21]. This metric identifies the classes that experienced most changes in the last versions of the system. In LENOM, the classes that most frequently changed are identified by weighting the delta in the method count (NOM) of a class between two adjacent versions. More formally:

$$LENOM_{j..k}(C) = \sum_{i=j+1}^{k} | NOM_i(C) - NOM_{i-1}(C) | * 2^{i-k}$$

where $1 \leq j < k \leq n$ being j the first version of the system analyzed, k the last version analyzed and n the total number of versions of the system.

Once the LENOM values for each main class of the code smells are obtained, the criterion computes the score of the containing agglomeration by averaging the LENOM values. For example, given an agglomeration A1 that is composed by three Brain Method (BM) smells: Foo.a(), Foo.b(), and Foo2.c(), and knowing that $LENOM(Foo) = 0.8$ and $LENOM(Foo2) = 0.5$, the score of A1 will be $\frac{0.8+0.8+0.5}{3} = 0.7$. A score close to 1.0 means that the classes composing the agglomeration change often during the history of the system. In contrast, a score of 0.0 means that the classes composing the agglomeration did not change since their initial implementation.

C. Agglomeration Cancer

This criterion makes an analogy of the agglomeration with a disease in the system. Our assumption is that a disease that is growing is more critical than a disease that is stable (i.e. it does not change) or that is on a remission state (i.e. it is shrinking).

Along this line, we analyze the behavior of the agglomerations across system versions and compute a variation rate in terms of the number of code smells that compose the agglomeration. We can think of this criterion as a variation of the previous one, which concentrates on the "volume" of smells over time. This criterion combines history-based and architectural information.

To calculate the score of a given agglomeration, we consider pairs of adjacent versions and determine the percentage of variation in the number of code smells that constitute the agglomeration. This percentage will be positive or negative, depending on whether the smells increased or decreased. For example, given agglomeration A1 with 3 code smells in version v1, 5 smells in v2, and 4 smells in v3, the corresponding variation rates are $\frac{5*100}{3} - 100 = 66.6\%$ from v1 to v2, and $\frac{4*100}{5} - 100 = -20\%$ from v2 to v3 (by definition, agglomerations always have at least two smells). Then, all the percentages of variation (for the same agglomeration) are averaged. In our example, this value becomes $\frac{66.6-20}{2} = 23.3\%$. Once averages for all the agglomeration are obtained, we normalize these values to produce scores in the range [0..1].

V. Study Settings

This section describes the research question and hypothesis of our study. We also describe the target applications used in our empirical evaluation, as well as the procedures for data collection and analysis. More details about the evaluation can be found at http://www.les.inf.puc-rio.br/opus/priorizatingAgglomerations/.

A. Research Question and Hypothesis

To investigate the effectiveness of the scoring criteria on the prioritization of architectural problems, we defined the following research question (**RQ**): *Does the use of a scoring criterion assist developers to prioritize smell agglomerations that indicate architectural problems?*

We derived the corresponding hypothesis for this research question: ($H1_0$) *the use of a scoring criterion do not assists developers to prioritize critical agglomerations*. We consider that a scoring criterion is effective to assist developers if at least half of the prioritized agglomerations are related to architectural problems. The reasoning is that developers would give up in inspecting the agglomerations if more than 50% of them are not related to architectural problems. If the criterion ranks correctly most agglomerations, we can conclude that the criterion enables developers to analyze the most critical agglomerations.

B. Target Applications

The systems chosen for our study had exhibited several symptoms of architecture degradation, so that we could properly evaluate the effectiveness of the prioritization criteria. The selection of the target systems was performed in two stages. In the first stage, we selected 3 Java applications of reasonable size (from 10 to 54 KLOC). They were also chosen because the original developers were available to validate the architectural problems and code smells being inspected in our analyses. Due to their availability, we produced both reliable ground truths and findings. The first application is Mobile Media (MM) [22], a software product line that provides

TABLE I: Characteristics of the Target Applications

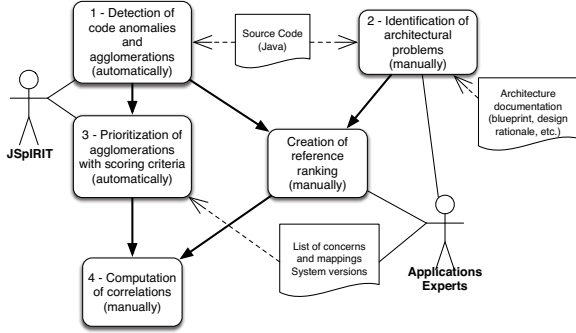| Target Application	MM	HW	S_{DB}	OODT
System Type	Software Product Line	Web	Web	Middleware
Programming Language	Java	Java	Java	Java
Architecture Design	MVC	Layers	MVC	Layers
Selected Version	5	8	2.4	10
KLOC	54	49	10	182
#Classes	77	125	151	1424
#Code Smells	260	497	82	1816

Fig. 4: Procedures for data collection

support for manipulation of media on mobile devices. The second application is Health Watcher (HW) [23], a Web-based application that allows citizens to register complaints about health issues in public institutions. Our third application is SubscriberDB (S_{DB}) [12], a subsystem of a publishing house that manages data related to the subscribers of its publications, and also supports different queries on the data.

In the second stage, we selected a large system, consisting of 182 KLOC. The goal was to check whether our most effective scoring criteria would also be effective to prioritize architectural problems in more complex projects. Given this goal, we selected Apache OODT (OODT) [24], a middleware framework aimed at supporting the management and storage of scientific data. A summary of the application characteristics is given in Table I.

C. Data Collection and Analysis

This section describes each of the main activities of the study, which are graphically summarized in Fig. 4.

1-Detection of code smells and agglomerations. We used *JSpIRIT* to detect both code smells and agglomerations automatically. After detecting all instances of code smells, *JSpIRIT* proceeds to identify the agglomerations based on the grouping patterns described in Section III-C. *JSpIRIT* presents two different outputs: (i) a list of smell instances, and (ii) groups of inter-related code smells, i.e., the agglomerations presented in Section IV, along with their score for a given criterion (Fig. 2). For the criterion of architectural concerns, we relied on a list of concerns provided by the original architects of each system. For each concern, they provided a list of packages/classes realizing the concerns, i.e., the concern mappings.

For the assumption of architectural components being represented by Java packages, we looked at the DSMs of each system using Lattix[3]. We applied the component partitioning

algorithms of Lattix, and analyzed the differences between the components suggested by Lattix and the package structure. We found that it was fair to assume a component as a Java package. For example, in the case of HW, Lattix identfied 17 component from 20 packages.

2-Identification of architectural problems. For HW, MM and S_{DB}, the application architects identified and reported to us the architectural problems they faced along their projects. Based on a catalog of architectural problems [2], architects reported the existence of 7 types of architectural problems, namely: Ambiguous Interface, Concern Overload, Connector Envy, Cyclic Dependency, Scattered Functionality, Unused Interface, and Architectural Violations (unwanted dependencies among components). To confirm the presence of architectural problems, the architects first manually inspected the source code and the architecture blueprint of each system. Based on their experience along the project, they produced a list of the most critical architectural problems for each version of the target applications. As a result, using the list of architectural problems, we produced a reference ranking of the agglomerations detected by *JSpIRIT* that contribute to the most critical architectural problems for each target application.

To determine if an agglomeration X was linked to an architectural problem Y, we check whether problem Y mapped to (some of) the main classes hosting the smells of agglomeration X. That is, we looked at intersections between the program elements realizing the architectural problem and those related to the agglomeration. Coming back to Fig. 1, we can see an example of this intersection for the Ambiguous Interface problem, which is realized by the *Controller* package and some of its classes take also part in an agglomeration. The reference ranking of agglomerations was built in such a way it has in the first positions the agglomerations being related to the highest number of architectural problems. That is to say, the score of an agglomeration is the number of related architectural problems. The agglomerations along with their related architectural problems for each case-study constituted our ground truth. For OODT, we did not produce a ground truth, due to the size and complexity of this application. Our goal with OODT was to evaluate the criteria in a system larger than HWS, MM or S_{DB} and assess the architectural problems for the best-ranked agglomerations. Along this line, we performed a manual analysis considering the top-12 agglomerations as ranked by the cancer criterion (Section IV-C).

3-Prioritization of agglomerations with scoring criteria (JSpIRIT). We simply executed *JSpIRIT* to apply automatically the scoring strategies from Section IV, one by one, on the agglomerations detected in the previous activity. As a result, the agglomerations were ranked according to their scoring value in a decreasing order. We focused on analyzing the top-7 rankings for each system, as those high-priority agglomerations would represent the initial focus of the developer's attention.

As an example, Fig. 5 shows the ranking of agglomerations for MM as produced by *JSpIRIT* with the cancer criterion (rows), and the associated architectural problems (columns) determined from the ground truth. Note that cell 2f (intra-component agglomeration based on DC for *Controller* intersecting with Ambiguous Interface in *Controller*) corresponds to the situation of Fig. 1. Also note that the intra-component in SmsMessaging (row 7) has no association to architectural

Ranking using the cancer criterion	a	b	c	d	e	f	g	h	i	j	k
1 - IntraComponent: FE datamodel	X	X	X	X				X			X
2 - IntraComponent: DC controller						X			X	X	
3 - IntraComponent: FE screens							X				
4 - Hierarchical: BM AbstractController						X			X	X	
5 - IntraComponent: BM controller						X			X	X	
6 - Hierarchical: DC AbstractController						X			X	X	
7 - IntraComponent: SmsMessanging											
8 - Hierarchical: FE MediaAccesor		X									X
9 - IntraComponent: MediaController						X			X	X	
10 - IntraComponent: MediaAccessor											X
11- Hierarchical: FE AlbumData	X	X				X					X
12 - IntraComponent: DC sms						X					
13 - IntraComponent: AlbumController						X					
14 - IntraComponent: SS controller						X			X	X	
15 - IntraComponent: AbstractController											
16 - Hierarchical: SS AbstractController						X			X	X	

a: Concern Overload - ImageAlbumData
b: Concern Overload - MusicAlbumData
c: Concern Overload - MusicMediaAccessor
d: Concern Overload - VideoAlbumData
e: Concern Overload - MusicMediaUtil
f: Ambiguous Interface - Controller
g: Ambiguous Interface - PlayMediaScreen
h: Redundant Interface - Datamodel
i: Cyclic dependency - Controller
j: Architectural violation - Controller
k: Architectural violation - Datamodel

Fig. 5: Matrix of ranked agglomerations for MM versus related architectural problems ('a' to 'k').

problems, even when it is relatively high in the ranking. This is a case of a false positive. which can be due to variations in the number of smells of the agglomerations across system versions, as detected by the cancer criterion. In other cases, like the hierarchical agglomeration based on FE for *AlbumData* (row 11), the agglomeration is ranked low in spite of being related to four architectural problems. This situation can be explained by the fact that the smells of the agglomeration remained almost constant over time.

4-Computation of correlations: For HW, MM and S_{DB}, once a given scoring strategy was applied on the agglomerations, we measured the correlation between the ranking generated by *JSpIRIT* and the reference ranking (from the ground truth). To do so, we applied the Spearman's correlation coefficient for rankings with ties (p) [25]. This coefficient measures the strength of the association between two rankings. The coefficient can take values between 1 and -1. If $p=1$, it indicates a perfect association between both rankings. If $p=0$, it indicates no correlation between the rankings. If $p=-1$, it indicates a negative association between the rankings. Finally, values between 0.5 and 0.7 are regarded as a good correlation, while values higher than 0.7 are regarded as a strong correlation. As we did not have a ground truth for OODT, we did not use the same correlation strategy. To evaluate OODT, we looked instead at the precision of the criteria for the top-12 ranked agglomerations. With the help of an OODT architect, we analyzed whether each agglomeration (in the top-12 ranking) was related to architectural problems. Table III shows the correlation results computed on the three case-studies, plus the precision value for OODT (7 true positives over 12 cases).

VI. EMPIRICAL EVALUATION

In this section, we first report on the results of applying each of the 3 scoring criteria to the 3 target applications. Then, as previously explained (Section V-B), we also discuss the results of applying the most effective criterion to the fourth software project, OODT, which is the largest one. Table

TABLE II: Architectural problems and code-smell agglomerations for the 4 applications

	HW	MM	S_{DB}	OODT
#Architectural problems	61	41	60	n/a
#Agglomerations	11	16	22	431

TABLE III: Correlation results (note that the value for OODT is a precision and not a correlation value)

Applications	Architectural concerns	History of changes	Agglomeration cancer
Health Watcher (HW)	0.01	0.57	0.62
Mobile Media (MM)	0.53	0.34	0.77
SubscriberDB (S_{DB})	0.38	0.71	0.14
SubscriberDB ($S_{DB_{v2}}$)	0.1	0.51	0.6
Apache OODT	n/a	n/a	0.58

II shows the number of architectural problems (reported by architects) and agglomerations (identified by JSpIRIT) in each system. On one hand, as suggested in recent studies [6], [4], we confirmed that the use of the agglomerations helped to discard hundreds of (non-agglomerated) smells that had no relationship to architectural problems. On the other hand, up to 60% of the agglomerations had no relationship to architectural problems, thereby confirming the need for defining and assessing the effectiveness of alternative prioritization criteria. Therefore, in the following subsections, we carefully analyze the correlation results for each scoring criterion and discuss their effectiveness to indicate locations of architectural problems. We also derive additional insights after inspecting all the prioritized agglomerations.

A. Do Architectural Concerns Help?

The architectural concerns were provided by the system architects. Our goal was to check whether the scoring criterion (Section IV-A) would work with a minimal amount of architectural information, which is usually part of either the project documentation or the mindset of architects. The architects spontaneously defined: (i) nine architectural concerns for HW – their mappings encompass around 100 classes in the program, which cover 74% of the total number of classes), (ii) seven architectural concerns for MM – their mappings include 65 classes (84% coverage), and (iii) five concerns for S_{DB} – their mappings subsume 45 classes (30% coverage).

After applying this criterion, *JSpIRIT* ranked the agglomerations according to their number of concerns. All the agglomerations were related to at least 3 architectural concerns. As shown in Table III, only MM had a moderate correlation with this criterion (correlation of 0.53 with p-value = 0,03471); the correlations for HW and S_{DB} turned out low. The correlations were low because the agglomerations with the largest number of architectural problems were not ranked first. The reason for these low correlations is that, albeit agglomerations were often related to architectural problems, this criterion gave the highest scores to agglomerations that were not related to architectural problems. Developers would need to inspect more agglomerations in the ranking in order to find the architectural problems. The use of this criterion would require more effort (than the other criteria) to find the first spots of architectural problems in the source code.

This criterion had the worst results in the HW system. In this system, the first two agglomerations ranked were related to 13 problems. However, the agglomerations ranked third, fourth and fifth were not related to architectural concerns. In the case of the HW system, 11 agglomerations were found, but only 5 of such agglomerations were actually related to problems. Therefore, this result does not necessarily represent a negative result because, on average, the developer would need to approximately inspect two agglomerations for finding at least one agglomeration related to architectural problems. Moreover, we found that certain agglomerations in the ranking tend to concentrate most of the architectural problems. For example, in the case of the HW system: two agglomerations were actually related to 14 problems, two with 13, and one agglomeration with 7 problems.

Concluding, we observed that the successful use of architectural concerns (as a criterion) depends on the completeness and coverage of the list of architectural concerns provided by the developers. In fact, MM was the system that had the mappings with the highest coverage (84%) and the highest correlation (0.53). In addition, we observed that this criterion worked well in systems (e.g. MM) where most problems were caused by the poor modularization of architectural concerns.

B. Does Change History Help?

In order to compute the rankings using the history criterion (Section IV-B), we loaded in *JSpIRIT* previous versions of the analyzed systems. In particular, we analyzed all the versions available for each application, namely: 10 versions of HW, 8 versions of MM and 15 versions of S_{DB}. In this case, we were able to find a moderate correlation for HW (0.57 with p-value = 0.06713) and a strong correlation for S_{DB} (0.71 with p-value = 0.00021). Also, we obtained a positive correlation for MM (Table III). These results mean that the agglomerations located in the classes that changed often during the history represent sources of architectural problems in the implementation. We observed that the classes realizing agglomerations related to architectural problems experienced more changes during their history than the agglomeration classes that were not affected by those problems. For instance, in the case of HW, after applying this criterion, the agglomeration ranked first by *JSpIRIT* is related to 7 architectural problems, while the agglomerations ranked second and third are related to 14 problems. Regarding the agglomerations related to 13 problems, they were tied in the sixth position. Therefore, we observed that the consideration of history information improves the prioritization of architectural problems as compared to the use of architectural concerns, presented in the previous sub-section.

However, the use of the change history criterion was also not effective to prioritize architectural problems in all software projects. Recent studies [3] have suggested that anomalous source code, whenever it is frequently changed, indicates the presence of major design problems. We found this might be true in certain systems and, therefore, this factor helps to identify architecturally-relevant code smells. However, this is not always the case, as captured by the low correlation (0.34) in MM. In this system, several architecturally-harmful agglomerations were not often touched by changes. Moreover, the success of the history criterion depends on having several versions to be processed.

C. Does Agglomeration Cancer Help?

Overall, the use of agglomeration cancer was the best-performing criterion in the context of our dataset. As shown in Table III, we obtained strong correlations for HW and MM. However, in principle, we obtained a low correlation in S_{DB}. However, while understanding the reasons for this low correlation, we realized there was an issue to be addressed in the S_{DB} artifacts. When examining the architectural blueprints provided by the system architects, we found out that the blueprint of S_{DB} was inconsistent with the source code [11]. By inconsistent, we mean that the blueprint was an "ideal" design model of the application, but it was not faithfully implemented in the source code. In fact, we computed a consistency metric [11] for HW, MM and S_{DB}. We found that the HW blueprint had a 89.6% of consistency, the MM blueprint had a 67.9%, and the S_{DB} blueprint had just a 54.5%.

For this reason, with the help of an S_{DB} architect, a new, more realistic blueprint called S_{DB_v2} was created, which had a consistency of 77.3%. In this case, the architect found 11 critical architectural problems. Then, we re-computed the reference ranking of this application and ran again the scoring criteria using *JSpIRIT*. Then, we observed a significant improvement in the correlation for the cancer criterion (0.6 with p-value=0.00315), that is, a moderate correlation. As shown in Table III, the correlation values for the remaining criteria decreased in S_{DB_v2}. In the case of the change history criterion, the correlation was still acceptable with the adjusted blueprint. Nonetheless, this was not the case for the criterion of architectural concerns that dropped to 0.1. These results indicate that the correlation values are sensitive to the way in which the blueprints are defined. Therefore, in order to get the best results of this criterion, developers also need to rely on blueprints of the implemented architecture rather than on blueprints of the planned architecture.

As agglomeration cancer was consistently the best-performing criterion in all the three systems, we also applied it to OODT. Our goal was to check whether this criterion would scale well to very large systems, such as OODT. Following the procedures described in Section V-C, we analyzed the top-12 ranked agglomerations in OODT (Fig. 6). In this analysis, we observed a moderate precision for architectural problems: 7 out of 12 agglomerations were true indicators of architectural problems. In addition, three of the top-4 agglomerations were related to one or more architectural problems. Therefore, the results for OODT suggest that the proposed criterion can work well for large software projects in terms of "circumscribing" the search for architectural problems. As an example of this, consider the OODT's *crawl* component. In OODT, *crawl* is responsible for listing the contents of a staging area and submitting products to be persisted by the *file management* component. However, as presented in Figure 6, there is an agglomeration indicating the presence of an architectural problem named Scattered Functionality. This architectural problem takes place when an architectural concern is scattered across multiple components and some of those components are responsible for other independent concerns. This problem occurs in *crawl* because functionality related with the extraction of metadata from products, which should be done by the *metadata* component, is mixed with the crawling functionality. The Scattered Functionality prob-

Ranking using the cancer criterion	a	b	c	d	e	f	g	h
1 - IntraComponent: FE action								
2 - IntraComponent: FE crawl	X							
3 - IntraComponent: FE page.metadata		X						
4 - Hierarchical: DC system			X	X				
5 - IntraComponent: FE mux								
6 - IntraClass: filemgr.util.XmlRpcStructFactory					X			
7 - IntraClass: workflow.util.XmlRpcStructFactory						X		
8 - IntraClass: resource.util.XmlRpcStructFactory								
9 - IntraComponent: FE jobqueue						X		
10 - IntraComponent: MediaAccessor							X	
11- Hierarchical: FE AlbumData								
12 - IntraComponent: DC sms								

a: Scattered Functionality - craw
b: Scattered Functionality - metadata
c: Connector Envy - system
d: Cyclic Dependency - system
e: Ambiguous Interface - XmlRpcStructFactory
f: Connector Envy - XmlRpcStructFactory
g: Cyclic Dependency - jobqueue
h: Connector Envy – query

Fig. 6: Matrix of (top-12) ranked agglomerations for OODT versus related architectural problems ('a' to 'h')

lem may affect aspects like maintainability and reusability. This happens because, when the shared concern needs to be changed, all the components that realize it should be updated and tested. Therefore, the metadata concern should be better modularized by the *metadata* component.

The only top-ranked agglomeration unrelated to any architectural problem is an intra-component agglomeration, found in the *action* component. In this agglomeration, the number of Feature Envy (FE) smells grew over time. However, this agglomeration turned out to be a false positive when judged by the OODT architect. The rationale is that the *action* component was responsible for parsing command line options, and thus it was expected to depend on several other classes for this task. The ranking also exposed several FE-related agglomerations, because this smell was more prevalent in the OODT versions than other types of smells. Thus, we consider that this result does not undertake our conclusions.

Overall Conclusion

After analyzing the results of the 3 scoring criteria: (i) we cannot reject $H1_0$ for the first two criteria since none of them sufficed to rank correctly at least half of the agglomerations across all the systems, and (ii) we can reject $H1_0$ for the third criterion, the agglomeration cancer criterion, since it ranked correctly more than half of the agglomerations in all the systems, including OODT. The use of the latter would help developers to find most architectural problems in all the systems with less effort. Developers would still need to inspect each ranked agglomeration and discard the irrelevant ones. However, we found they could discard more than 500 code smells in their analysis. If developers analyze all these individual code smells, they would need to carefully inspect dozens or hundreds of smells in order to eventually find a partial source of a single architectural problem.

Furthermore, even when the detection might lead to some false positives, the automation of the criteria with *JSpIRIT* contributes to significantly reducing mistakes and manual effort of developers. With existing solutions/tools, developers would have to investigate all the architectural information, code smells, and all their relationships, in order to luckily find key architectural problems.

When analyzing why the agglomeration cancer was consistently the best criterion, we observed an interesting phenomenon affecting most of the architectural problems: groups of smells flocking together tend to better indicate the presence of architectural problems, and these groups tend to be increasingly connected with additional new smells when changes are made in the source code over time. This phenomenon is often caused by an ill architectural decision in early system versions. This finding can be illustrated by the misuse of *Controllers* in the MVC architecture of MobileMedia. In principle, the architects decided to rely on the use of a single *Controller* instead of multiple *Controllers* (Section III-A). There were only three smells as part of the agglomeration affecting Controllers in the first version. These smells were located in the BaseController class. In subsequent versions, this agglomeration was being "expanded" to several code elements, including those located in new BaseController subclasses and new BaseController clients. The newly-introduced smells in the existing agglomeration were all directly caused by the harmful constraint of having only a single Controller. Therefore, inter-related smells in the code of evolving software systems (i.e. cancer criterion) tend to be good indicators of architectural problems.

D. Threats to Validity

In this section, we present potential threats to the validity of our study and how we tried to mitigate them.

Internal and External Validity. An internal threat is related to the quality of the mappings between architectural problems and code elements. We used a consistency metric [11] to make sure that the architectural specification reached a minimum quality. In addition, for each target application, we validated with system experts all the responsibilities and architectural components realized by the code elements in the different system versions. A threat related to the criteria was about the mapping of concerns to code and the selection of the system versions. Also, the usage of LENOM as the main metric for the history criterion can introduce a bias, because some kinds of changes are insensitive to LENOM. The main threat to external validity is that the applications analyzed were relatively small with few instances of code smells and agglomerations. We mitigated this threat by analyzing the cancer criterion in the context of Apache OODT. Unfortunately, performing a complete analysis in larger applications (like OODT) is not always viable because an expert must manually analyze the source code and the blueprints to find the architectural problems.

Construct and Conclusion Validity. As construct threats, we can mention possible errors introduced in the generation of the reference ranking. We partially mitigated this imprecision by involving the original architects and developers in the inspection process. For all target applications, architects with previous experience on the detection of architectural problems and code smells, validated and refined the list of problems. The main threat to conclusion validity refers to the number of target applications. We are aware that a higher number of applications is needed for generalizing our findings. However, the information required to conduct this kind of studies can be difficult to obtain. For instance, the activities of identifying and validating architectural problems is highly dependent on having the original personnel available. In order to account for this threat, we selected applications with different sizes,

purposes and domains. Moreover, the applications had different architectural styles and involved a different set of architectural problems (with a minor overlapping).

VII. CONCLUDING REMARKS

As far as we are aware of, no previous work supports the prioritization of smell agglomerations in order to assist developers to focus on a limited set of potential sources of architectural problems. The prioritization is based on three scoring criteria that have the goal of ranking first the agglomerations that likely indicate locations of architectural problems. In order to rank agglomerations, the criteria explored different types of information, which are typically available in software projects, including (partial) lists of architectural concerns, (approximate) component structure, and change history. As a proof-of-concept, the scoring criteria were implemented in the *JSpIRIT* tool.

In order to assess and compare the effectiveness of the prioritization criteria, we conducted a study based on the analysis of four systems, one of them with a very large size (OODT). In our study, we found out that, although the effectiveness of most criteria depended on the characteristics of each project, the use of the agglomeration cancer criterion tended to be consistently effective across all the projects, including OODT. The other criteria did not present a good correlation in certain projects. For instance, in two projects, the use of architectural concerns alone did not suffice to pinpoint agglomerations related to architectural problems. We observed that the criterion based on architectural concerns is effective only in projects were developers are able to provide a complete coverage of the architectural concerns. In our dataset, this was the case of the MM, where the specification of the architectural concerns covered 84% of the classes in the source code. However, even for this project, it would be useful to also rely on the use of the cancer criterion as: (i) it had a strong correlation with architectural problems in this system, and (ii) it helped to spot a different list of architectural problems, not detected with the criterion of architectural concerns.

We also believe that these findings have practical implications. For instance, the choice amongst concern-based, history-based and cancer-based criteria has tradeoffs [12]. The usage of the criterion of architectural concerns may be preferred in several cases, even at the cost of being an inferior indicator of architectural problems, because problems can be spotted since the first versions when they are usually easier to be dealt with. Further studies could investigate how the use of architectural concerns and agglomeration cancer could be combined in order to get better indicators of architectural problems. In fact, in our study, using them in conjunction would lead to the identification of almost 90% of all architectural problems affecting the three first systems: HW, MM and S_{DB}. Futher work can investiage particular combinations of scoring criteria and understand which combination tends to be most effective across a larger sample of projects. As regards history-based information, we envision the usage of metrics other than LENOM as an alternative proxy for maintenance effort.

ACKNOWLEDGMENT

REFERENCES

[1] P. Clements and R. Kazman, *Software Architecture in Practice*. Addison-Wesley, 2003.

[2] J. Garcia, D. Popescu, G. Edwards, and N. Medvidovic, "Identifying architectural bad smells." in *CSMR*. IEEE, 2009, pp. 255–258.

[3] S. Wong, Y. Cai, M. Kim, and M. Dalton, "Detecting software modularity violations." in *ICSE*, 2011.

[4] W. Oizumi, A. Garcia, L. da Silva Sousa, B. Cafeo, and Y. Zhao, "Code anomalies flock together: Exploring code anomaly agglomerations for locating design problems," in *ICSE'16 – to appear*, May 2016 2015.

[5] J. Garcia, D. Popescu, G. Edwards, and N. Medvidovic, "Toward a catalogue of architectural bad smells," in *QoSA*, 2009.

[6] W. Oizumi, A. Garcia, T. Colanzi, M. Ferreira, and A. von Staa, "On the relationship of code-anomaly agglomerations and architectural problems," *Journal Soft. Eng. Research and Dev.*, vol. 3, no. 11, pp. 1–22, 2015.

[7] M. Fowler, *Refactoring: improving the design of existing code*. Addison-Wesley, 1999.

[8] C. Sant'Anna, E. Figueiredo, A. Garcia, and C. Lucena, "On the modularity assessment of software architectures: Do my architectural concerns count," in *AOSD*, 2007.

[9] R. Mo, Y. Cai, R. Kazman, and L. Xiao, "Hotspot patterns: The formal definition and automatic detection of architecture smells," in *WICSA'15*, May 2015, pp. 51–60.

[10] T. Gîrba, S. Ducasse, and M. Lanza, "Yesterday's weather: Guiding early reverse engineering efforts by summarizing the evolution of changes." in *ICSM*, 2004.

[11] E. Guimaraes, A. Garcia, and Y. Cai, "Exploring blueprints on the prioritization of architecturally relevant code anomalies," in *COMPSAC*, 2014.

[12] S. Vidal, C. Marcos, and J. A. Díaz Pace, "An approach to prioritize code smells for refactoring," *Automated Soft. Eng.*, pp. 1–32, 2014.

[13] S. Vidal, E. Guimaraes, W. Oizumi, A. Garcia, A. Díaz Pace, and C. Marcos, "On the criteria for prioritizing code anomalies to identify architectural problems," in *31th Annual ACM Symposium on Applied Computing*, ser. SAC '16. New York, NY, USA: ACM, 2016.

[14] N. Moha, Y.-G. Guéhéneuc, L. Duchien, and A.-F. L. Meur, "Decor: A method for the specification and detection of code and design smells." *IEEE Trans. on Soft. Eng.*, vol. 36, pp. 20–36, 2010.

[15] D. Sjoberg, A. Yamashita, B. Anda, A. Mockus, and T. Dyba, "Quantifying the effect of code smells on maintenance effort," *IEEE Trans. on Soft. Eng.*, vol. 39, pp. 1144–1156, 2013.

[16] InFusion, "www.intooitus.com/products/infusion.html."

[17] D. E. Perry and A. L. Wolf, "Foundations for the study of software architecture," *SIGSOFT Softw. Eng. Notes*, vol. 17, pp. 40–52, 1992.

[18] M. Lanza and R. Marinescu, *Object-Oriented Metrics in Practices*. Springer, 2006.

[19] A. K. McCallum, "Mallet: A machine learning for language toolkit," 2002, http://mallet.cs.umass.edu.

[20] T. T. Nguyen, H. V. Nguyen, H. A. Nguyen, and T. N. Nguyen, "Aspect recommendation for evolving software," in *ICSE'11*. New York, NY, USA: ACM, 2011, pp. 361–370.

[21] S. A. Vidal, C. Marcos, and J. A. Díaz Pace, "Analyzing the history of software systems to predict class changes," in *IEEE Biennial Congress of Argentina (ARGENCON)*, 2014.

[22] T. J. Young, "Using aspectj to build a software product line for mobile devices," Ph.D. dissertation, The University of British Columbia, 2005.

[23] S. Soares, E. Laureano, and P. Borba, "Implementing distribution and persistence aspects with aspectj," in *ACM Sigplan Notices*, 2002.

[24] C. A. Mattmann, D. J. Crichton, N. Medvidovic, and S. Hughes, "A software architecture-based framework for highly distributed and data intensive scientific applications," in *ICSE'06*, 2006.

[25] F. Ricci, L. Rokach, B. Shapira, and P. Kantor, Eds., *Recommender Systems*. Springer, 2011.