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@exa.unicen.edu.ar
†UNIFOR, Brazil
Email: eguimaraes@unifor.br
‡PUC-Rio, Brazil
Email: woizumi@inf.puc-rio.br
§PUC-Rio, Brazil
Email: afgarcia@inf.puc-rio.br
¶ISISTAN-CONICET, Argentina
Email: adiaz@exa.unicen.edu.ar
‖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. Some studies revealed that a range of architectural problems are reflected in source code through code smells. However, a software project often contains thousands of code smells and many of them have no relation to architectural problems. Unfortunately, state-of-the-art techniques fail short in assisting developers on the prioritization of architectural problems realized in the source code. As a consequence, developers struggle to effectively determine which (groups of) smells are architecturally relevant, i.e., can expose critical problems. This work proposes 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 20 versions of 3 systems, analyzing their effectiveness for detecting symptoms of architectural problems. The results provide evidence that the proposed criteria helped to correctly prioritize more than 80 architectural problems in our top-7 rankings, alleviating tedious manual inspections of the source code vis-a-vis with the architecture. Our 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], [4]. 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], [5]. Typical examples of architectural problems are Fat Interface and Unwanted Dependency between components [6], [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 impair software performance and maintainability [2].

Unfortunately, the identification of architectural problems is time consuming and cumbersome for several reasons. The identification generally requires the analysis of both architectural information (e.g., documentation, specifications) and its realization in source code. It is hard for a developer to effectively explore these two types of information altogether in order to uncover architectural problems. Design documentation is often informal and scarce. Even when architecture information is available, it is often not detailed enough to help developers to detect architectural problems, such as fat interfaces and unwanted dependencies. Thus, developers need to find hints in the source code that indicate architectural problems. However, well-known anomalies in the source code – popularly known as *code smells* [7] – only provide partial hints of the location of architecture problems in a system [8], [5] (Section II). Classical examples of code smells are Long Method, God Class and Feature Envy.

An architectural problem is often realized by a subgroup of code smells scattered in the source code, which makes its detection even more challenging [8], [5], [9]. Even for small systems, developers would need to analyze hundreds of code smells and infer their likelihood of indicating 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 code smells may be indicators of architectural problems and thus be targets for refactorings. A more practical strategy 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. Even worse, they do not establish any criteria to guide developers on locating 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 help developers to expose the location of certain architectural problems in their systems. Our criteria are centered on the notion of *agglomerations* [9] of code smells. 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). The use of such agglomerations may assist developers in identifying the full extent of an architectural problem in source code, while discarding irrelevant code smells. The proposed criteria consider both the implementation and architectural information that is available, including the versioning history of a system (Section IV).

We assess how each of the three criteria helps developers to locate symptoms of architectural problems in source code, while keeping aside irrelevant code anomalies. Our study considered more than 20 versions of 3 systems (Section V). Our results (Section VI) suggest that the use of our three criteria can accurately indicate several architectural problems. They helped to correctly locate more than 80 architectural problems in our top-7 rankings, alleviating tedious manual inspections of the source code vis-a-vis with the architecture. Our prioritization criteria 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 studies (e.g. [2], [3], [4]). In particular, a catalog of architectural problems was recently documented [2]. There are also case studies reporting the severe impact of architectural problems, such as fat interfaces and unwanted dependencies, on the longevity of industrial software systems [3], [4]. In [10], five types of architectural smells (or issues) are presented and formalized. These smells can be detected in Design Structure Matrices (DSMs) by means of tool support. Two of the smells are based on the analysis of history information, while the other three smells come from the code structure. The authors show a correlation of such smells with high bug frequencies and maintenance efforts. . However, none of these works assists in the prioritization of code smells with the goal of identifying architectural problems. Moreover, the aforementioned studies only focus on analyzing a single system.

Other studies have investigated the impact of *single code smells* throughout system evolution [11], [12]. They analyzed whether the number of smells increased or decreased over time, and how often they resulted in code refactorings. In [8], [13], [5], 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 anomaly 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 (detached from other anomalies in the system) did not impact on the system design.

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 [9] to capture a group of inter-related code anomalies. 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 [14] the authors only documented a few relationships among code smells that may be related to four relevant design problems. However, to the best of our knowledge, no work has yet investigated strategies for prioritizing groups of code anomalies. 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 [8], [5], [14], [15], [16], [17]. 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.

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], [18]. We mainly target problems affecting the modular decomposition of a system into components and their interfaces, i.e., modifiability-related problems. Based on previous empirical findings (Section II), the premise is that groups of code smells, so-called *agglomerations*, are normally associated with several architectural problems. In order to illustrate the link 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 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 anomalies. Then, it may not be clear which code smells should be the focus of her attention as candidates for revealing architectural problems in those

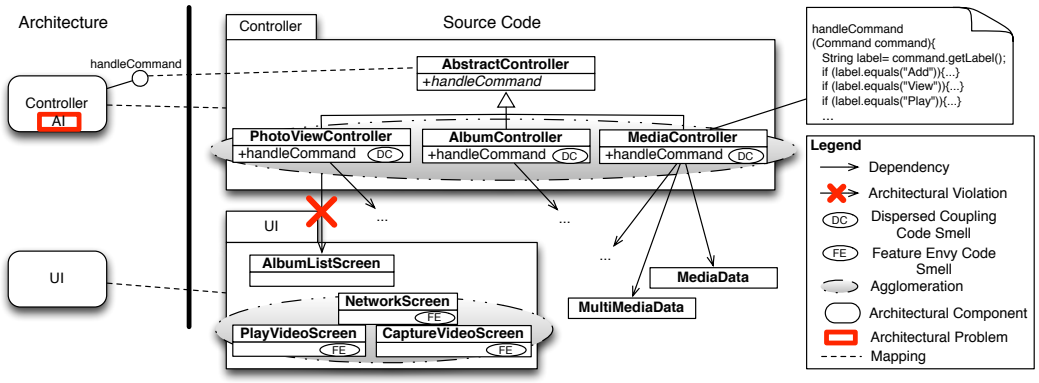


Fig. 1: Example of a code-smell agglomeration related to architectural problems

components, which prevents her from performing effective maintenance or refactoring activities.

Architectural Problems. This example reveals three architectural problems. First, the design of class *AbstractController* 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 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/classes if the controller´s 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 that the developer reasons about the architecture documentation and checks 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 anomalies within the *Controller* package, this group of anomalies 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 sure about the architectural problem exposed by the DC agglomeration, 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 for a class being more interested in accessing data from other classes (instead of using its own data), which often indicates a poor division 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 prioritization criteria for agglomerations.

A. Detecting Code Smells

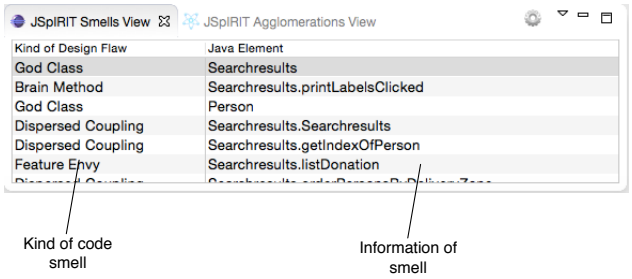
Existing catalogs of code smells define guidelines to identify single smells and how to provide tool support for their detection [7], [19]. In this work, we use the *JSpIRIT*[1] tool for that purpose. *JSpIRIT* is an Eclipse plugin for detecting code smells of a (Java-based) system and ranking them according to different criteria [17]. 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] [17] following the detection strategies presented in the catalog of Lanza and Marinescu [19].

B. Identifying Agglomerations

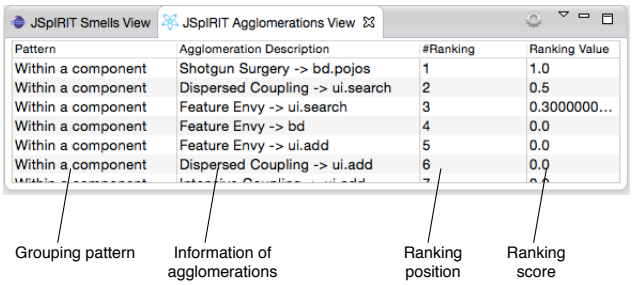
The original version of *JSpIRIT* was extended to support the detection of agglomerations. Fig. 2b shows a list of agglomerations detected on the basis of the smells of Fig. 2a. Since our work focuses on architectural information regarding static code structures, we worked with agglomerations within

[1]https://sites.google.com/site/santiagoavidal/projects/jspirit

[2]A complete list of the supported code smells can be found at https://db.tt/T6uWtdM0



(a) Code anomalies



(b) Agglomerations

Fig. 2: *JSpIRIT* outputs

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 for grouping code anomalies, which are briefly described next.

- **Anomalies 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 anomaly. 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) anomaly.
- **Anomalies 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 (DE) smells.

A more complete description of the above agglomerations can be found in [20]. Certainly, other types of agglomerations of smells are possible, as reported in [9].

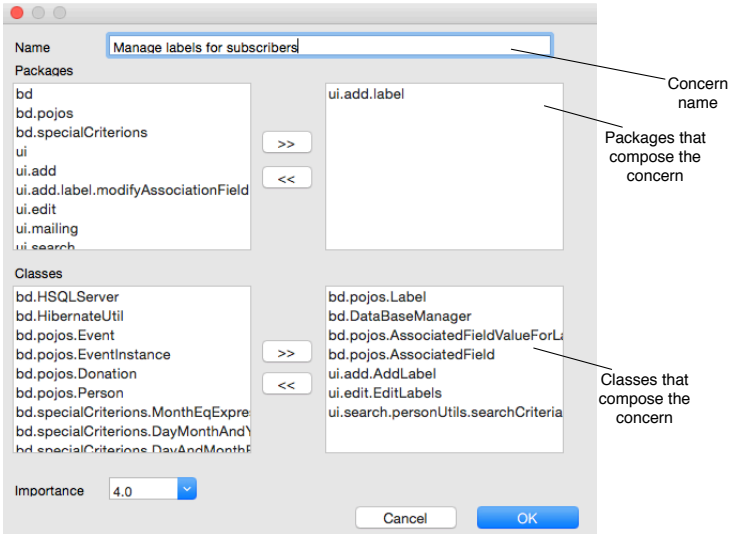


Fig. 3: Wizard to provide concern mappings in *JSpIRIT*

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[3]. The score value indicates how critical the agglomeration is for the system architecture (0=no 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 [21], such as graphical user interface (GUI), exception handling, or persistence. For example, in Fig. 1, the subclasses of *AbstractControlle*r (along with other system classes) do all address a concern called *PhotoLabelManagement*. This criterion was adapted from [13] where it is used to rank single code smells. The rationale behind this criterion is that an agglomeration that realizes several concerns (of the architecture) could be an indicator of an architectural problem.

[3]There can be ties in the scores assigned to different agglomerations.

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. 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 scores for the agglomerationswill be $0/3 = 0.0$, $1/3 = 0.33$, $2/3 = 0.66$ and $3/3 = 1.0$.

Certainly, the specific concern mappings to code affects the results of this criterion. Furthermore, as the implementation evolves, the mappings might need to be adjusted. Existing feature-location tools, such as XScan [22], can be used here to find concern mappings to program elements automatically and with high accuracy.

B. History of Changes

This criterion analyzes the stability of the classes in which the code smells (of an agglomeration) are implemented. By looking at the stability of the main classes of these code smells, we want to check whether the agglomeration is in a part of the system 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 (from the agglomeration), but also on evolving information (from the history of class changes).

To calculate the score of an agglomeration we use the LENOM metric [11]. 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 number of methods (NOM) of a given class between two adjacent versions. More formally:

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

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, we compute 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) code 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 "volume" of code anomalies 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 anomalies in v2, and 4 anomalies 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 score values in the range [0..1].

V. Study Settings

This section describes the research questions and hypotheses of our study. We also describe the target applications used in our empirical evaluation, as well as the procedures for data collection and analysis.

A. Research Question and Hypothesis

To investigate the effectiveness of our scoring criteria on the prioritization of architectural problems, we defined two research questions to be addressed in this study:

- **RQ#1** - Does the sole use of each scoring criterion assists developers to prioritize agglomerations that indicate architectural problems?
- **RQ#2** - Does the joint use of the scoring criteria improve the prioritization of the agglomerations with respect to architectural problems?

We derived a hypothesis for each research question: ($H1_0$) the use of a single scoring criterion suffices on assisting developers to prioritize critical agglomerations; and ($H2_0$) the joint use of all the scoring criteria improves the prioritization of critical agglomerations. If the prioritization criteria prove to be effective, they can enable developers to focus on analyzing and fixing the most critical agglomerations. In addition, the prioritization criteria might help developers to discard code smells that do not contribute to any architectural problem. Therefore, we are interested in understanding first how code-smell agglomerations alone might suffice to point out architectural problems.

TABLE I: Characteristics of the Target Applications

Target Application	MM	HW	S_{DB}
System Type	Software Product Line	Web	Web
Programming Language	Java	Java	Java
Architecture Design	MVC	Layers	MVC
Selected Version	5	8	2.4
KLOC	54	49	10

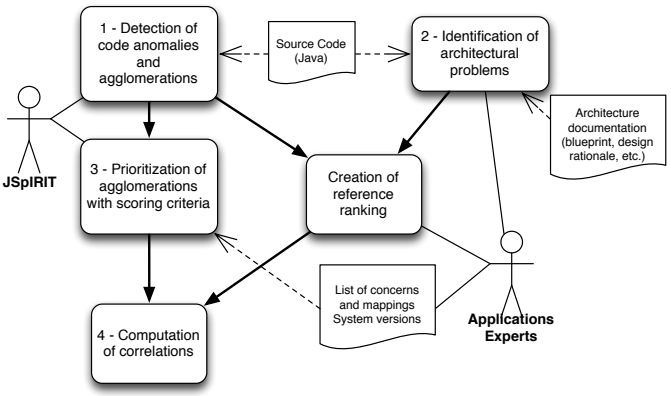


Fig. 4: Procedures for data collection

Once we understand how agglomerations might be associated with architectural problems, we can apply the scoring criteria (Section IV) to prioritize those agglomerations that have the highest chances of indicating an architectural problem. The next sections describe the 3 applications selected for our study and the evaluation activities.

B. Target Applications

We selected 3 Java applications for this study. The first application is Mobile Media (MM) [23], a software product line that provides support for manipulation of media on mobile devices. The second application is Health Watcher (HW) [24], a Web-based application that allows citizens to register complaints about health issues in public institutions. Our third application is SubscriberDB (SDB) [17], a subsystem of a publishing house that manages data related to the subscribers of its publications, and also upports different queries on the data. A summary of the main characteristics of the applications can be found in Table I. These applications were chosen because they met a number of relevant criteria for our study, namely: (i) they are non-trivial systems and their sizes (varying from 10 to 54 KLOC) are manageable for an in-depth analysis of code smells; (ii) the applications have been evaluated in other studies [13], [21], [24], [17], [23], [9] regarding design problems, and (iii) the original developers were available to validate the detection strategies for the code smells as well as the architectural problems observed for each version.

C. Data Collection and Analysis

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

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

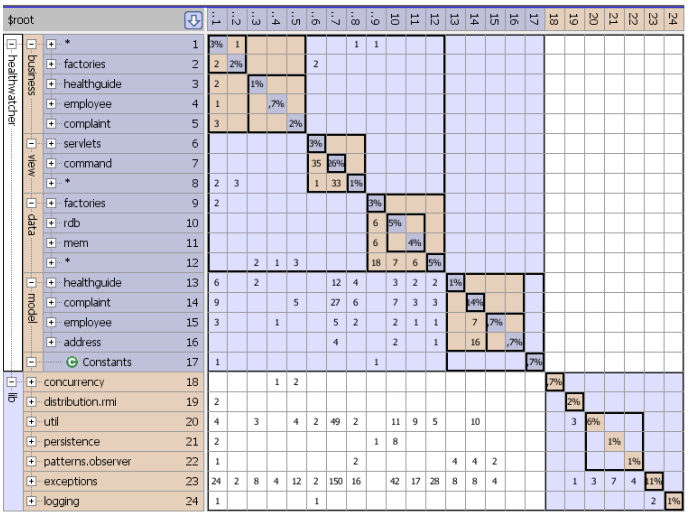


Fig. 5: DSM of HW after applying the standard partitioning algorithm of Lattix.

patterns described in Section III-B. *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 scoring criterion (Fig. 2). For the criterion of architectural concerns, we relied on a list of concerns provided by the original developers of each system. For each concern, they provided a list of classes/methods realizing the concerns, i.e., the concern mappings. These mappings were further validated by us with the help of Mallet [9], [25]. For the assumption of architectural components being represented by Java packages, we looked at the DSMs of each system using Lattix[4]. We applied the component partitioning algorithms of Lattix, and analyzed the differences between the components suggested by Lattix and the package structure. For instance, Fig 5 shows a DSM for HW with 17 components (at the lowest abstraction level) and 20 packages. In all case-studies, the differences in the number of components/packages were small.

Identification of architectural problems. For all target applications, the application developers identified and reported to us the architectural problems they faced along their projects. Specifically, developers 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) [2], [5]. To confirm the presence of architectural problems, developers first manually inspected the source code and the architecture blueprint of each system. This task was aided with a questionnaire that we provided to them. 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

[4]http://lattix.com/

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 agglomeration was built in such a way it has in the first positions the agglomerations being related to the highest number of critical architectural problems. That is to say, the score of the agglomeration is the number of related architectural problems. The agglomerations along with their related architectural problems for each case-study constituted our ground truth.

Prioritization of agglomerations with scoring criteria (JSpIRIT). We simply asked JSpIRIT to apply 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 focus of the developer's attention.Fig. 6 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 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.

Computation of correlations: 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) [26]. This coefficient measures the strength of 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. Table III shows the correlation results computed on our case-studies.

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. 6: Matrix of ranked agglomerations for MM versus related architectural problems.

TABLE II: Architectural problems and code-smell agglomerations for the 3 applications

	HW	MM	S_{DB}
#Architectural problems	61	41	60
#Agglomerations	11	16	22

VI. Empirical Evaluation

In this section, we report on the results of applying our 3 scoring criteria to the 3 target applications. Table II shows the number of architectural problems and agglomerations in each application. One one hand,we observed that the use of the agglomerations helped to discard many (individual) smells that have 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 assessing the effectiveness of our scoring criteria. Therefore, in the following sections, we analyze the correlation results for each scoring criterion and discuss whether it suffices (or not) indicate architectural problems.

A. Do Architectural Concerns Suffice?

The architectural concerns were provided by the system developers. Our goal was to check whether the scoring cri-

TABLE III: Correlation results

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 ()	0.38	0.71	0.14
SubscriberDB ($S_{DB_{v2}}$)	0.1	0.51	0.6

terion (Section IV-A) would work with minimal amount of architectural information, which is usually part of either the project documentation or the mindset of architects. Therefore, the developers defined 9 concerns for HW (involving around 100 classes, 74% of the total number of classes), 7 for MM (around 65 classes, 84%), and 5 for S_{DB} (around 45 classes, 30%)[5].

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 reason for these low correlations is that this criterion gave the highest scores to agglomerations that were not related with architectural problems. For example, in the case of HW, 11 agglomerations were found, but only 5 of such agglomerations were actually related with problems. However, this does not represent a negative result because, on average, the developer would need to inspect two agglomerations for finding at least one architectural problem. Actually, we found in HW that agglomerations were related with 14 problems, 2 ones with 13 and, 1 with 7.

If developers analyze individual code smells, they would need to exhaustively inspect dozens or hundreds of smells in order to eventually find a partial source of a single architectural problem. After applying this criterion, *JSpIRIT* ranked the agglomerations according to their number of concerns. All the agglomerations were related with at least 3 architectural concerns. In HW, the first 2 agglomerations ranked were related with 13 problems. However, the agglomerations ranked third, fourth and fifth were not related with architectural concerns. We observed that the successful use of architectural concerns (as a criterion) depends on the list and coverage of the architectural concerns provided by the developers. .

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 implemented in the classes that changed often during the history are related to architectural problems. In other words, the classes of agglomerations related with problems experienced more changes during their history than the classes of agglomeration that were not affected by 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 addition of history information improves the results in the prioritization of architectural problems as compared to the strict use of architectural concerns, presented in the previous sub-section.

[5]A complete list of the concerns can be found at https://db.tt/T6uWtdM0

C. Does Agglomeration Cancer Help?

To apply the cancer criterion, we used the last available version of each system, an intermediate one, and the first available version of each system. For example, from the 10 versions available for HW, we analyzed versions 1, 5 and 10 in order to compute this criterion. Overall, this 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, we obtained a low correlation in S_{DB}. To understand the reasons for this low correlation, we examined the architectural blueprints provided by the system experts, and found out that the blueprint of S_{DB} was inconsistent with the source code [13]. 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 [13] for the 3 applications. 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} expert, a new, more realistic blueprint called S_{DB_v2} was created, which had a consistency of 77.3%. In this case, the expert found 11 critical architectural problems. Then, we re-computed the reference ranking of this application and ran again the scoring strategies using *JSpIRIT*. We obtained an 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 need to rely on blueprints of the actual implemented architectures rather than blueprints of the planned (albeit not implemented) architecture.

a) Overall Conclusion: After analyzing the results of the 3 scoring criteria, we can reject $H1_0$ since each criterion only sufficed to rank first those agglomerations related with architectural problems in different systems. On the contrary, we can accept $H2_0$ as the joint use of the scoring criteria would help developers to find architectural problems in all the systems. They 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. Furthermore, even when the detection might lead to some false positives, the automation of the criteria with JSpIRIT contributes to significantly reducing mistakes and the manual effort of developers. With existing solutions, developers would have to investigate all the architectural information, code smells, and all their relationships, in order to luckily find key 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 [13] to make sure that the architectural specification reached a minimum quality. In addition, for each target application under evaluation, 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. For the first criterion, we believe that the usage of Mallet mitigated the threat. 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. Unfortunately, performing the same analysis in larger applications 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 anomalies, 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 accout 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

In this article, we proposed a novel approach to prioritize groups of code smells (or agglomerations) that are critical for the architecture of a system. As far as we are aware of, no previous work supports the prioritization of code smells and their agglomerations in order to better assisting developers to find 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 certain types of architectural problems. Our goal is not helping developers to find all sorts of architectural problems; instead, we aim at helping them to detect architectural problems that manifest as anomalous structures in the source code, while discarding anomalous structures (both agglomerated and non-agglomerated anomalies) that have no relationship to major design 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. The scoring criteria are implemented in the *JSpIRIT* tool.

In order to assess the advantages of our approach, we conducted a study based on the analysis of several versions of 3 applications. In our study, we found out that: (i) overall, the effectiveness of a certain criterion depended on the characteristics of each project, and (ii) the combined use of all three criteria helped developers to reveal different architectural problems. For instance, in two projects, the use of architectural information alone did not suffice to distinguish agglomerations related with architectural problems from the rest. We observed that the criterion based on architectural concerns only suffices 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 case-study, 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.

Therefore, we believe that these findings have practical implications. For instance, the choice between architecture-based and history-based criteria has tradeoffs [17]. The usage of the criterion of architectural concerns criterion 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. Our results show evidence that architectural information should be complemented with history information in order to get good indicators of architectural problems. Certainly, this potential improvement depends on a judicious selection of the system versions.

As future work, we will evaluate our criteria with more (and larger-size) applications in order to get additional insights. For instance, we also plan to study particular combinations of our scoring criteria (and other criteria) in order to understand which combination tends to be more productive across projects. Furthermore, we would like to experiment with other types of agglomerations (as reported in [9]) and also with architectural smells reported by other authors, such as [10]. As regards history, we envision the usage of metrics other than LENOM as proxy for maintenance efforts.

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] R. Schwanke, L. Xiao, and Y. Cai, "Measuring architecture quality by structure plus history analysis," in ICSE, 2013.

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

[5] I. Macia, F. Dantas, A. Garcia, and A. von Staa, "Are code anomaly patterns relevant to architectural problems?" in IEEE Trans. on Soft. Eng., 2014.

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

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

[8] R. Arcoverde, E. Guimaraes, I. Macia, A. Garcia, and Y. Cai, "Prioritization of code anomalies based on architecture sensitiveness," in 27th SBES, 2013.

[9] 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 Proceedings of the 38th International Conference on Software Engineering (ICSE'16) – to appear, May 2016 2015.

[10] R. Mo, Y. Cai, R. Kazman, and L. Xiao, "Hotspot patterns: The formal definition and automatic detection of architecture smells," in Software Architecture (WICSA), 2015 12th Working IEEE/IFIP Conference on, May 2015, pp. 51–60.

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

[12] A. Lozano and M. Wermelinger, "Assessing the effect of clones on changeability," in ICSM, 2008.

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

[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." [Online]. Available: http://www.intooitus.com/products/infusion.html

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

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

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

[20] W. Oizumi, A. Garcia, M. Ferreira, A. von Staa, and T. E. Colanzi, "When code-anomaly agglomerations represent architectural problems?" in SBES, 2014.

[21] 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.

[22] T. T. Nguyen, H. V. Nguyen, H. A. Nguyen, and T. N. Nguyen, "Aspect recommendation for evolving software," in Proceedings of the 33rd International Conference on Software Engineering, ser. ICSE '11. New York, NY, USA: ACM, 2011, pp. 361–370. [Online]. Available: http://doi.acm.org/10.1145/1985793.1985843

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

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

[25] A. K. McCallum, "MALLET: A Machine Learning for Language Toolkit," 2002, http://mallet.cs.umass.edu. [Online]. Available: http://mallet.cs.umass.edu

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