On the Criteria for Prioritizing Code Anomalies to Identify Architectural Problems

Santiago Vidal  
ISISTAN-CONICET, Argentina  
svidal@exa.unicen.edu.ar

Everton Guimaraes  
UNIFOR, Brazil  
eguimaraes@unifor.br

Willian Oizumi  
PUC-Rio, Brazil  
woizumi@inf.puc-rio.br

Alessandro Garcia  
PUC-Rio, Brazil  
afgarca@inf.puc-rio.br

Andrés Díaz Pace  
ISISTAN-CONICET, Argentina  
adiaz@exa.unicen.edu.ar

Claudia Marcos  
ISISTAN-CIC, Argentina  
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 wide range of architectural problems are reflected in source code through code anomalies. However, a software project often contains thousands of code anomalies and many of them have no relation to architectural problems. As a consequence, developers struggle to effectively determine which (groups of) anomalies are architecturally relevant. This work proposes criteria for prioritizing groups of code anomalies as indicators of architectural problems in evolving systems.

CCS Concepts
• Software and its engineering → Software maintenance tools; Software architectures; Software design;

Keywords
Code anomalies, architectural problems, software maintenance

1. INTRODUCTION
Software systems usually suffer from architectural problems that negatively affect the maintainability and other quality attributes of a system [3, 6]. Unfortunately, the identification of architectural problems is time consuming and cumbersome for several reasons. Even when architecture information is available, it is often not detailed enough to help developers to detect architectural problems. Thus, developers need to find hints in the source code that indicate architectural problems. However, well-known anomalies in the source code (also known as code smells [2]) only provide partial hints of the location of architecture problems in a system [1, 6] (Section 2).

Under these circumstances, developers do not know where to focus on, i.e. which code anomalies may be indicators of architectural problems. A more practical strategy is to prioritize groups of code anomalies according to their criticality to the system architecture, that is, their ability to point out architectural problems. Unfortunately, existing work (Section 2) does not support developers on automatically prioritizing code anomalies to reveal architectural problems.

In this context, we propose two scoring criteria for prioritizing code anomalies (Section 4). The goal is to help developers to reveal the location of certain architectural problems in their systems. Our criteria is centered on the notion of agglomerations [7, 8] of code anomalies. Agglomerations are groups of inter-related code anomalies that likely indicate together the presence of an architectural problem (Section 3). The use of such agglomerations may assist developers in identifying the full extent of an architectural problem in source code, while discarding irrelevant code anomalies.

We assess how criteria help developers to locate symptoms of architectural problems in the source code of 2 systems, and not be falsely alarmed by irrelevant code anomalies. Our results (Section 5), while preliminary, suggest that the use of our two criteria can accurately indicate several architectural problems. At the end, we reflect upon all these findings and present concluding remarks (Section 6).

2. RELATED WORK
As far as we are aware of, our work represents the first effort on supporting the prioritization of architecturally-relevant anomalies. In [1, 4, 6], the authors report on the relevance of code anomalies for the identification of architectural problems. As main findings, they observed that single anomalies alone do not suffice to assist developers in revealing architectural problems. In addition, a very high proportion of individual anomalies (detached from other anomalies in the system) did not impact the system’s architectural design.

Only a few studies have gone beyond and looked at groups of code anomalies as indicators of architectural problems. The
3. AGGLOMERATIONS AS POINTERS TO ARCHITECTURAL PROBLEMS

In this work, we focus on architectural problems [3] that represent violations of design principles or rules [3]. Based on previous empirical findings (Section 2), the premise is that groups of code anomalies, so-called agglomerations, are normally associated with several architectural problems.

Detecting Code Anomalies We use the JSpIRIT¹ tool to identify single anomalies. JSpIRIT is an Eclipse plugin for detecting code anomalies of a system and ranking them according to different criteria [9]. Currently, JSpIRIT supports the identification of 10 single anomalies (such as Feature Envy, Brain Method, and Disperse Coupling)² [9] following the detection strategies presented in the catalog of Lanza and Marinescu [5].

Identifying Agglomerations The original version of JSpIRIT was extended to support the detection of agglomerations. Since our work focuses on architectural information, we worked with agglomerations within the scope of architectural components. We assume here an implicit relation (or mapping) between an architectural component and its realization as a Java package in the code. However, developers can flexibly establish mappings between components and classes in a program. We are mainly interested in two particular patterns³ for grouping code anomalies, which are briefly described next. A more complete description of these agglomerations can be found in [7].

Anomalies within a component. This grouping pattern identifies code anomalies that are implemented by the same architectural component. Specifically, we look for one single component with: (i) code anomalies 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.

Anomalies in a hierarchy. This grouping pattern identifies code anomalies that occur across the same inheritance tree involving one or more components. We only consider hierarchies exhibiting the same type of code anomaly. The rationale is that a recurring introduction of the same anomaly in different code elements might represent a bigger problem in the hierarchy.

4. PRIORITIZATION APPROACH

In this section, we present 2 criteria to prioritize agglomerations by means of scoring criteria. The proposed criteria were implemented in JSpIRIT. Our hypothesis is that these criteria are useful to prioritize, i.e., to segregate, those agglomerations with high chances of spotting architectural problems. In this way, a criterion can be seen as a function:

\[ \text{criterion}_A(\text{agglomeration}_B) = \text{score}_{A,B} \]

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=no critical, 1=very critical). In particular, we began working with a criterion based solely on architectural information (namely, architectural concerns). Later, we developed a novel criteria based on the combination of the code versions and architectural information (namely, agglomeration cancer).

4.1 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 we developers aim at treating in a modular way [4], such as graphical user interface (GUI), exception handling, or persistence. This criterion was adapted from [4] where it is used to rank single code anomalies. 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 load concerns. 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]. For example, given 4 agglomerations A1, A2, A3, and A4 that involve 0, 1, 2 and 3 concerns respectively, their scores will be 0, 0.33, 0.66 and 1.

4.2 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 anomalies that compose the agglomeration.

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 anomalies that compose the agglomeration. This percentage will be positive or negative, depending on whether the anomalies increased or decreased.

For example, given agglomeration A1 with 3 code anomalies in version v1, 5 anomalies in v2, and 4 anomalies in v3, the corresponding variation rates are \( \frac{5 - 3}{3} = 66.6\% \) (v1/v2) and \( \frac{4 - 5}{5} = -20\% \) (v2/v3). 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,

¹https://sites.google.com/site/santiagoavidal/projects/jspirit
²A complete list of the supported anomalies can be found at https://db.tt/T6uWtdM0
³Certainly, other grouping patterns for the code anomalies are possible.
Table 1: Architectural problems and agglomerations

<table>
<thead>
<tr>
<th>#Architectural problems</th>
<th>HW</th>
<th>MM</th>
</tr>
</thead>
<tbody>
<tr>
<td>#Agglomerations</td>
<td>61</td>
<td>41</td>
</tr>
</tbody>
</table>

Table 2: Correlation results

<table>
<thead>
<tr>
<th>Applications</th>
<th>Architectural concerns</th>
<th>Agglomeration cancer</th>
</tr>
</thead>
<tbody>
<tr>
<td>HW</td>
<td>0.01</td>
<td>0.62</td>
</tr>
<tr>
<td>MM</td>
<td>0.53</td>
<td>0.77</td>
</tr>
</tbody>
</table>

we normalize these values to produce score values in the range [0, 1].

5. CASE STUDY

To investigate the effectiveness of our scoring criteria on the prioritization of architectural problems, we analyzed 2 Java applications: i) Mobile Media (MM)\(^4\), a software product line that provides support for manipulation of media on mobile devices, and ii) Health Watcher (HW)\(^4\), a Web-based application that allows citizens to register complaints about health issues in public institutions. The goal of the evaluation was to analyze if the sole use of each scoring criterion assist developers to prioritize agglomerations that indicate architectural problems. If the prioritization criteria prove to be effective, they can enable developers to focus on analyzing and fixing the most critical agglomerations.

We used JSpIRIT to detect both code anomalies and agglomerations automatically. For all target applications, expert developers identified and reported to us the architectural problems they faced along their projects. Table 1 shows the number of architectural problems and agglomerations in each application. 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. This reference ranking is built in such a way that it has in the first positions the agglomerations being related to the highest number of critical architectural problems. Once a given scoring strategy was applied on the agglomerations, we measured the Spearman’s correlation between the ranking generated by JSpIRIT and a reference ranking.

Do Architectural Concerns Sufficient? The architectural concerns were provided by the system developers. Developers defined 9 concerns for HW (involving around 100 classes, 74% of the total number of classes) and 7 for MM (around 65 classes, 84%). As it is shown in Table 2, while MM had a moderate correlation with this criterion; the correlations for HW turned out low. The reason for this low correlations is that this criterion gave the highest scores to agglomerations that were not related with architectural problems. In the case of HW, 11 agglomerations were found, but only 5 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 that 2 agglomerations were related with 14 problems, 2 ones with 13 and, 1 with 7.

Does Agglomeration Cancer Help? In order to apply the agglomeration 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. As it is shown in Table 2, we obtained strong correlations for HW and MM.

After analyzing the results of both scoring criteria, we can say that our preliminary results are encouraging and that they have demonstrated the potentialities of the approach. However, we think that further experiments with other applications are necessary in order to generalize our results.

6. CONCLUSIONS

In this article, we proposed a novel approach to prioritize groups of code anomalies (or agglomerations) that are critical for the architecture of a system. The prioritization is based on two scoring criteria that have the goal of ranking first the agglomerations that likely indicate certain types of architectural problems. To assess the advantages of our approach, we conducted a study based on the analysis of several versions of 2 applications. While preliminary, the results indicate the viability of the approach. As future work, we will evaluate our criteria with more applications in order to get additional insights.

7. ACKNOWLEDGMENTS

This work was partially supported by MIntCyT-CAPES through bilateral Project BR/13/09.

8. REFERENCES


\(^4\)http://ptolemy.cs.iastate.edu/design-study/