On the Relevance of Code Anomalies for Identifying Architecture Degradation Symptoms

Isela Macia\(^1\), Roberta Arcoverde\(^1\), Alessandro Garcia\(^1\), Christina Chavez\(^2\), Arndt von Staa\(^1\)

\(^1\)Opus Group, LES, Informatics Department, PUC-Rio, RJ, Brazil
\(^2\)Computer Science Department, Federal University of Bahia, BA, Brazil

\{ibertran, racoverde, afgarcia, arndt\}@inf.puc-rio.br; flach@dcc.ufba.br

Abstract—The longevity of evolving software systems largely depends on their resilience to architectural design degradation. It is often assumed that code anomalies are always key indicators of architecture degradation symptoms. The problem is that there is still limited knowledge about the circumstances under which code anomalies represent architectural problems. Without this knowledge, developers are not able to implement architecturally-relevant strategies for code refactoring. This paper presents an empirical study about the influence of code anomalies on architecture degradation symptoms. To this end, we studied the relationship between code anomalies and architecture problems in 6 software systems, which were intended to adhere different architectural decompositions. A total of 40 versions and 2056 code anomalies were analyzed. Our study revealed that 78% of all architecture problems in the programs were related to code anomalies. In particular, more than 33% of all architecture problems were unexpectedly induced by anomalous code elements in the systems’ first version. We also found that the refactoring strategies, even when frequently applied in those systems, did not significantly contribute to remove architecturally-relevant code anomalies.

Keywords - code anomaly, refactoring, architectural violation, architectural anomaly

I. INTRODUCTION

The longevity of evolving software projects largely depends on their resilience to architecture degradation symptoms. It is often assumed that such degradation symptoms can be observed in the system implementation through the identification of code anomalies\(^1\)\([9,12]\). Code anomalies are implementation structures that possibly indicate a deeper design problem \([9]\). Then, given time constraints, developers often need to focus on removing code anomalies that are critical to architecture designs; we call them as architecturally-relevant code anomalies. The detection and removal of those architecturally-relevant anomalies are even more difficult when the architecture design is not explicitly documented or kept up to date. In fact, the lack of coherent architectural documentation hinders the application of current techniques for preventing architecture degradation \([7,24,25,35]\).

In these cases, developers can only resort to figuring out (or guessing) which anomalies in the code represent architectural problems. The challenge is that there is little evidence about the circumstances under which code anomalies actually represent architectural problems. Without this knowledge, developers cannot identify code refactoring strategies \([9]\) that fix architectural problems. In fact, it is questionable if all the types of code anomalies \([9]\) can incur in any damage to the architecture design \([20,21]\). Therefore, developers might waste time when removing anomalies that do not represent any threat to the architecture design.

Previous works have only studied the impact of code anomalies on changes and defects during systems’ evolution \([4,14,29]\). They found that, while the type of the code anomaly may be a good indicator of system’s changes, it is not useful for indicating system’s defects. However, these works did not try to investigate which types of code anomalies are likely to be related to architecture degradation symptoms. In fact, they did not analyze which characteristics of the code anomaly could be indicators of its adverse influence on architecture design. There is also no work that investigates to what extent the adopted refactoring strategies are able to remove architecturally-relevant code anomalies. This limitation is particularly relevant given the growing evidence that refactoring is increasingly a common practice to remove code anomalies \([15,28,37,38]\).

This paper presents a first study about the impact of code anomalies on architecture design degradation. A sample of 2056 code anomalies distributed in 40 versions of 6 software systems were analyzed. We analyzed in this sample if certain anomaly characteristics were objective indicators of architecture problems. We have also investigated to what extent refactoring were applied for removing architecturally-relevant code anomalies in these applications. The design decisions in such systems were intended to follow different architectural decompositions (e.g., Layers, Model-View-Controller, and Aspectual Design). We explicitly selected systems for which the architectural documentation was accessible so that we could correlate architecture problems with the presence of code anomalies. Their architects and developers were also consulted to confirm the relevance of code anomalies to architecture problems and validate other findings (Section IV).

Our results confirm that code anomalies often adversely impact architectural designs. More specifically:

- Approximately 65% of all code anomalies were related to 78% of all architecture problems. This result seems to confirm that detection of code anomalies is useful to locate potential sources of architecture degradation. In turn, this suggests that systematic code refactoring could contribute to address symptoms of architecture degradation (Section V.A).

---

\(^1\) We use the term code anomaly as a synonym of bad smell.
Certain types of code anomalies were consistently related to architecture problems (e.g., Long Method, God Class). However, none of them emerged as the best indicator of degradation symptoms across all the systems. Anomalous code elements often introduced architecture problems when implementing non-cohesive functionalities and accessing information from several modules; regardless of the type of the code anomaly (Section V.B.1).

More than 18% of all architecturally-relevant anomalies emerged from anomalous code elements in the system's first version. These early anomalies were responsible for inducing more than 33% of all architectural problems. This means that certain refactorings should be prioritized to remove code anomalies as early as possible (Section V.B.2).

About 66% of all refactorings did not contribute to fix architecturally-relevant code anomalies. These refactorings were usually confined to the private members of classes. However, as relevant anomalies often infected public members of interfaces and superclasses, they could only be removed by applying API-level refactoring (Section V.C).

Finally, Section VI discusses how we circumvented threats to validity and Section VII provides the concluding remarks.

II. RELATED WORK

Empirical Studies on Code Anomaly Effects. Recent empirical studies have investigated the effects of code anomalies on software maintenance. Olbrich et al. [29] and Khomh et al. [14] investigated the impact of code anomalies throughout the system's evolution. Specifically, the authors analyzed whether the number of code anomalies tended to increase over time, and how often they resulted in code-level changes. D'Ambros et al. [4] studied the impact of code anomalies on software defects. They found that, while some types of code anomalies are more frequent, none of them can be considered more harmful with respect to software defects. In the context of aspect-oriented systems, Macia et al. [22] analyzed the influence of code anomalies on corrective changes. They also analyzed their influence on perfective changes (i.e., refactoring effort).

However, none of the aforementioned studies have investigated whether and to what extent code anomalies could be used as indicators of architecture degradation symptoms in the source code (Section V.A). Understanding the effects of code anomalies on architecture designs is even more relevant. They have a broader impact on maintenance of a software system, making the system harder to evolve and compromising its longevity.

Preventing Architecture Degradation in Source Code. A number of techniques address the problem of enforcing design rules in the source code; their goal is to support the prevention of architecture degradation symptoms. Aldrich [1] proposed ArchJava, a technique for statically enforcing the structural conformance of Java code with respect to the architectural specification. Eichberg et al. [7] proposed Vespucci, an approach to support design rule checking. Marwan and Aldrich [24] developed SCHOLIA, a technique for documenting system architecture in the source code and checking its conformance with the intended architecture. Morgan et al. [25] defined a domain-specific language to specify and check design rules in object-oriented systems. Ubayashi et al. [35] presented Archface, a programming-level interface to represent the intended architectural design and check its conformance with the source code.

All these techniques rely on the explicit documentation of the architecture design, which is often absent or obsolete in real-life software projects. In addition, they only support the detection of a particular form of degradation symptoms, the so-called architecture erosion (Section III.B). They also often demand a complete specification of the projections of architecture elements in the code, which is a daunting task. In contrast, we are interested in investigating whether the systematic detection and removal of code anomalies could contribute to detect both forms of architecture degradation symptoms in the source code (Section V.B); regardless of the existence of architectural documentation.

Empirical Studies on Refactorings. Refactoring [9] is the primary mechanism to remove code anomalies from the source code. Several studies have investigated the impact of refactorings on software quality. Murphy-Hill et al. [28] observed several factors associated with the characteristics and frequency of refactorings. Xing and Stroulia [38] studied the impact of refactoring on structural modifications over time. Dig et al. [5] studied the role of refactorings in API evolution. Weißgerber and Diehl [37] found that refactorings may introduce a high number of bugs. Kim et al. [15] found that, while the number of bugs increases after API-level refactorings, the time taken to fix bugs decreases. In contrast to these studies, we aim at investigating to what extent the refactorings applied by developers are effective to address architecturally-relevant code anomalies (Section V.C).

III. CODE ANOMALIES AND ARCHITECTURAL PROBLEMS

This section introduces concepts related to code anomaly (Section III.A) and classify the problems that might occur in an evolving system's architecture (Section III.B). It also illustrates how these architecture problems can be related (Section III.C) using a running example.

A. Code Anomalies

Code anomalies are symptoms in the source code that possibly indicate a deeper design problem [9]. They affect different code units, such as classes and methods, having a negative impact on system's maintainability. For example, Long Method [9] is a code anomaly in which a method (i) implements various functionalities (i.e., concerns) – i.e. it performs too much work on its own, and (ii) presents a high complexity. As an illustration, the right hand side of Figure 1 depicts a Long Method extracted from HealthWatcher (Section IV.A), detected by developers - namely, SearchData.execute. It was classified as a Long Method because the method body has a high internal complexity - as it contains many lines of code and has high cyclomatic
exception handling policy. For instance, the SearchData class invokes different services from the Business module. SearchData ends up handling exceptions (e.g. Transaction) signaled by the Data module, including those that should be treated internally. That leads to additional code couplings, between elements realizing the Data and GUI modules, resulting in architecture violations. These violations are represented by the dashed red arrows in Figure 1.

**Architectural Drift** is the introduction of design decisions into a system’s architecture that were not included in the intended architecture, albeit they do not violate any of the previously-made design decisions [30]. Symptoms of architectural drift are caused by applying design decisions that impair system’s maintenance. Each of these symptoms of architectural drift is often referred to as architectural anomaly [10][17]. These anomalies may negatively impact architectural modularity principles, such as narrow module interfaces and modules realizing a single concern [17][30].

In order to select a set of architectural anomalies to be analyzed in our study, we considered catalogs of architectural anomalies explicitly documented in the literature [10][17]. Our final subset of analyzed anomalies encompassed those that were identified by architects in the target systems of our study (Section IV.B). The types of architectural anomalies we analyzed are summarized in Table I. Note that each architectural anomaly hinges different modularity principles; for instance, while Module Concern Overload anomaly does not adhere to the single concern principle, the Ambiguous Interface hinders the satisfaction of the simple interface principle.

Figure 1 shows an example of the Scattered Parasitic Functionality anomaly. The three modules are responsible for partially implementing widely-scope and independent system’s concerns, called Transaction and Persistence. This situation hinders the architecture modularity in two ways. First, the scattered functionality leads to additional dependencies between modules [10]. Second, at least one module addresses multiple independent concerns. Note that even though this situation does not necessarily incur in violation of intended design decisions (i.e. symptoms of architectural erosion), the scattered functionality favors tight module coupling as the system evolves. Additional examples and a discussion of each architectural anomaly can be found in [10].

### Code Anomalies as Indicators of Architectural Degradation Symptoms

Our study (Section IV) aims at analyzing to what extent code anomalies introduce architectural problems. In this work we consider that a code anomaly $C$ is correlated to an architecture problem $P$ when: (i) the code elements (e.g., methods or classes) affected by $C$ implement architecture elements (e.g., modules and interfaces), and (ii) these architecture elements are affected by $P$. We only considered those correlations where the cause-effect relationship was confirmed by architects and developers (Section IV.B).
IV. Study Settings

Our study aims to answer three research questions: (a) are anomalous code elements related to architecture problems?, (b) if so, which characteristics of the code anomaly are relevant for the architectural design?, and (c) to what extent the applied refactorings actually addressed architecturally-relevant code anomalies?

For the second question, we focus on analyzing two characteristics of code anomalies: their type (Section III.A), and the earliness of their occurrence in a software project history. A code anomaly is considered to be early if it was introduced in the first version of a system. The motivation for the former analysis is that certain types of code anomalies tend to occur more often than others [4][14]. However, there is no knowledge about how often frequent anomaly types are sources of architecture problems. The motivation for the later analysis is to understand if code anomalies introduced early in a software project are harmful (or not) to architecture design; if so, this implies that developers should watch out for harmful early anomalies and anticipate their removal through early refactorings. The target systems and data collection methodology are presented in Section IV.A and Section IV.B, respectively.

A. Selection Criteria and Target Systems

The first major decision that had to be made in our study was the formulation of the criteria for selecting the target applications. First, it is crucial to select systems developed with a modular architecture design upfront. Otherwise, the lack of a good design or guidelines could introduce "noise" in our results. Second, we need to rely on the architecture design documentation in order to better analyze the relationships between code anomalies and architectural problems. Third, the original architects and developers of the selected systems should be available to validate the impact of code anomalies on architecture designs. Similar systems with different designs and implementations should be selected to observe whether particular characteristics of code anomalies were recurrently harmful regardless the design and implementation decisions. It is also important to select systems that underwent changes and suffered from a high number and variety of code anomalies. These project properties allow us to analyze the harmfulness of code anomalies on architecture design that evolved in different ways. A complete list of the criteria is provided in [3].

Based on these criteria, we chose 6 systems, totaling 40 versions. Table II summarizes the general characteristics of each target system. The first system is a lightweight middleware platform, called MIDAS, for distributed, event-based sensor applications [22]. The two selected versions are the before and after versions of a major restructuring with the widest impact in MIDAS' history. A high number of architectural elements suffered from changes in this transition. Two of these systems, MobileMedia (MM) and AspectualMedia (AM), are software product lines for deriving applications that manipulate photos, videos and music on mobile devices [8]. We separated the data of the object-oriented versions (Java) and its aspect-oriented counterpart (AspectJ) using a slash ("/”). The fourth and fifth systems, HealthWatcher (HW) and AspectualWatcher (AW), are real web framework systems, which allow citizens to register complaints about health issues in public institutions [11]. The sixth system is a real web-based system that allows scenographers to plan and manage scenic sets in television productions. To preserve copyright constraints, the fictitious name of PDP is used in this paper to refer to it. Similarly to MIDAS, both selected versions of PDP are the before and after versions of representing the major changes in that system history.

B. Procedures for Data Collection and Analysis

The data collection process involved several activities, including: recovering the actual architecture of each analyzed system version, identifying architectural problems, detecting code anomalies and analyzing their impact on architecture design. These activities are described next.

Recovering the Actual Architecture. This activity was based on a semi-automatic process. We used Sonar [33] and Understand [36] to support the recovery of the actual
architecture from the source code. They support architecture and code analysis to help developers measuring modularity in both levels. Furthermore, architects and developers mapped the code elements to the architecture elements in the module view. These mappings allowed us to trace the influence of a code anomaly on architecture design. These mappings also allowed us to identify how the modularization of concerns in the code was related to architecture problems. An example of this mapping is shown in Figure 1. We can see that the GUI module has to deal with several concerns as a consequence of the SearchData class, which realizes GUI, Persistence, Transaction and Business concerns.

Identifying Architecture Problems. In order to identify symptoms of architectural erosion we used Software Reflexion Model [27]. The comparison of the actual, extracted architecture (EA), and the intended architecture (IA) was supported by two groups of architects: (i) those that defined the original intended architecture, and (ii) independent reviewers of the system architecture. We measured the architecture conformance in terms of convergence (a module or relationship that is in both EA and IE), divergence (a module or relationship that is in EA but not in IA), and absence relationships (a module or relationship that is in IA but not EA). For instance, all absence classifications were considered violations. Although divergence classifications are natural suspects of possible violations, they might be related to intended architecture decisions. Therefore, architects and developers needed to validate their actual impact on architecture designs.

Furthermore, architectural anomalies (Section III.B) were detected by architects based mainly on: (i) a visual inspection of the EA, and (ii) a careful analysis of the mappings between code elements and architecture elements, due to the lack of tools for doing so automatically. We also asked the original architects to indicate other anomalies observed in the architecture design beyond those presented in Table I such as cyclic dependencies. This helped us to better judge whether and which code anomalies are good indicators of architecture problems.

Detecting Code Anomalies. As a first step, code anomalies were automatically identified using detection strategies [16]—similarly to other studies [14][21][29]. The metrices required by detection strategies were mostly collected with current tools such as: Together [34], MuLATo [26] and Understand [36]. These tools are complementary: Together analyzes Java programs, Understand analyzes C++ and C# programs, and MuLATo is a static analyzer for AspectJ programs. As a second step, the list of suspects provided by detection strategies was validated by the developers. This validation was motivated by the fact that strategies presented low rates of accuracy when detecting architecturally-relevant code anomalies [19]. By mixing automatic with manual detection, we aimed at relying on a reliable set of code anomalies.

Analyzing the Impact of Code Anomalies. In order to analyze the relationships between code anomalies and architecture problems (i.e. correlation and cause-effect), we started from reports provided by the architects. These reports included fine-grained and accurate details about identified architecture problems, facilitating our correlation analysis. They describe, for instance, the problem’s type, its location in the architecture design and the code elements related to it. We have also systematically applied the following heuristics to infer each cause-effect relationship. First, we observed whether the same structural modifications caused a code anomaly and an architecture problem. Second, we checked whether the definition of an anomalous code element introduced an architectural problem. Finally, we examined whether changes performed considering the evolution of an anomalous code element caused an architecture problem.

We also relied on a set of criteria to validate the cause-effect relationship between code anomalies and architecture problems. First, the cause-effect relationship is recurrently inferred in almost all systems’ versions and for many of the involved code anomaly occurrences and architecture problems. Second, the cause-effect relationship is observed in different modules of the same system and, additionally, these modules involved the contribution of different developers. Lastly, all the inferred cause-effect relationships are confirmed by architects and developers.

V. FINDINGS ON THE IMPACT OF CODE ANOMALIES

The following sections present and discuss the main findings associated with each of the aforementioned research questions (Section IV). Section V.A discusses whether anomalous code elements impact on architecture designs. Section V.B reports our observations about the architecture degradation impact of two characteristics of code anomalies, i.e. their type and earliness in a software project (Section IV). Finally, Section V.C discusses whether architecturally-relevant code anomalies were often removed via refactoring realized by programmers in later versions.

A. Are Anomalous Code Elements Architecturally Relevant?

In order to investigate whether anomalous code elements are more related to architecture problems than anomalous-free code elements, we first applied the Fisher’s exact test [32]. It checks whether the proportion of architecture problems varies across code elements with or without code anomalies. We have also calculated the odds ratio (ORs) [32] to check whether anomalous code elements have the same probability to be related to architecture problems that anomaly-free elements.

Results of Fisher test for both architectural violations and architectural anomalies are presented in Tables III and IV, respectively. In these tables, the versions of MIDAS and PDP named as “BEF” and “AFT” correspond to the version before and after applying major changes, respectively. Also, lower p-values indicate that elements with code anomalies adversely impact architecture design more than anomalous-free elements. Data for MIDAS and PDP are not presented in Table III as no violation occurred. This finding was expected as the development process in these projects strictly enforced architecture conformance.
TABLE III.  FISHER TEST FOR VIOLATIONS

<table>
<thead>
<tr>
<th>Releases</th>
<th>p-values</th>
<th>ORs</th>
<th>Releases</th>
<th>p-values</th>
<th>ORs</th>
</tr>
</thead>
<tbody>
<tr>
<td>AspectualWatcher</td>
<td>HealthWatcher</td>
<td>AspectualMedia</td>
<td>MobileMedia</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.0</td>
<td>&lt;0.05</td>
<td>0.8</td>
<td>1.0</td>
<td>&lt;0.05</td>
<td>4.2</td>
</tr>
<tr>
<td>4.0</td>
<td>&lt;0.05</td>
<td>2.4</td>
<td>4.0</td>
<td>&lt;0.05</td>
<td>6.0</td>
</tr>
<tr>
<td>7.0</td>
<td>&lt;0.05</td>
<td>3.3</td>
<td>7.0</td>
<td>&lt;0.05</td>
<td>5.1</td>
</tr>
<tr>
<td>10.0</td>
<td>&lt;0.05</td>
<td>3.9</td>
<td>10.0</td>
<td>&lt;0.05</td>
<td>2.8</td>
</tr>
</tbody>
</table>

TABLE IV.  FISHER TEST FOR ANOMALIES

<table>
<thead>
<tr>
<th>Releases</th>
<th>p-values</th>
<th>ORs</th>
<th>Releases</th>
<th>p-values</th>
<th>ORs</th>
</tr>
</thead>
<tbody>
<tr>
<td>AspectualWatcher</td>
<td>HealthWatcher</td>
<td>AspectualMedia</td>
<td>MobileMedia</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MIDAS</td>
<td>PDP</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BEF</td>
<td>0.07</td>
<td>1.9</td>
<td>BEF</td>
<td>&lt;0.05</td>
<td>3.8</td>
</tr>
<tr>
<td>AFT</td>
<td>&lt;0.05</td>
<td>2.1</td>
<td>AFT</td>
<td>&lt;0.05</td>
<td>3.2</td>
</tr>
</tbody>
</table>

Architectural Significance of Code Anomalies. Our analysis revealed a statistically-significant relationship between anomalous code elements and architecture problems in 77.5% of the analyzed versions according to the level of confidence (i.e. 0.05). Also, the odds ratio revealed that the chances for anomalous code elements to be related to architecture problems were twice or more higher than for anomalous-free elements. However, the significance of this relationship was not statistically confirmed in the first version of certain systems. In these first versions anomalous elements were scattered over all architecture modules, while architecture problems were confined to certain modules. On the other hand, architecture problems emerged in multiple modules over time as a consequence of changes performed on anomalous code elements. Thus, the relationship between anomalous code elements and architecture problems was statistically confirmed in later versions. This finding reveals that developers should be more concerned with refactoring anomalous elements in the first version. Certain early anomalies require special attention even when they do not represent a threat to the architecture design (Section V.B.2).

Upstream and Downstream Analyses. In order to complement the Fisher's analysis, we analyzed the upstream and downstream relationships between code anomalies and architecture problems. The upstream relationship refers to what extent code anomalies caused architecture problems, while the downstream corresponds to what extent architecture problems were caused by code anomalies. These analyses are useful because they show: (i) what proportion of code anomalies is critical for systems' architecture, and (ii) what proportion of architecture problems could be fixed by refactoring code anomalies. The upstream analysis showed that up to 81% of code anomalies was correlated to architecture problems in HW, 72% in MM, 68% in AW, 65% in PDP, 63% in AM and 51% in MIDAS. As we can notice, a considerable proportion of code anomalies did not impact the architecture design in the analyzed systems. This finding highlights the necessity about understanding whether certain characteristics of the code anomaly were the source of architecture problems (Section V.B).

On the other hand, a downstream analysis revealed that up to 86% all architecture problems were caused by code anomalies in HW, 83% in AM, 80% in MM, 75% in AW, 71% in PDP, and 70% in MIDAS. The observed results indicated the vast majority of architecture problems was caused by anomalous code elements. The high cause-effect was particularly surprising in certain systems or sub-systems with stringent architecture enforcement. We found high cause-effect rates through the evolution of modules where conformance of architectural rules was strictly enforced in the source code. The MIDAS project is the best example: although design rules were not violated, many occurrences of architectural anomalies emerged over time. They were found in code elements with low cohesion and high coupling. Even though these code elements mainly implemented a single architecture module, they realized multiple scattered concerns. This phenomenon was often caused by broadly-scoped scattered concerns, such as service discovery, fault tolerance policies, and dynamic adaptation.

The results of these multi-dimensional analyses seem to confirm that detection of code anomalies is useful to locate potential sources of architecture erosion and drift (Section III). This suggests in turn that application of systematic code refactoring could effectively contribute to combat early symptoms of architecture degradation. This observation is even more relevant for projects where there is neither effort on proactive maintenance of architecture documentation nor investment on using heavyweight architectural conformance tools (Section II). We also found that a considerable proportion of all code anomalies, about 40%, were not indicators of relevant design problems. This means that refactorings cannot be chosen in an arbitrary manner when architecture revisions of the source code are being carried out. Developers should be equipped with guidance and tool support to identify and rank code anomalies according to their relevance to architecture degradation. In this context, the next section discusses whether certain characteristics of code anomalies were better indicators of architecture problems in the analyzed systems.

B. Are Particular Characteristics of Code Anomalies Indicators of Architectural Problems?

Once confirmed that anomalous elements were often related to architecture problems, our study investigated the role played by the type and earliness of code anomalies in this context. Additionally, the results of the previous section also reinforce the importance of studying the earliness of code anomalies.

1) Types of Code Anomalies

A logistic regression model [13] was used to investigate to what extent particular types of code anomalies introduced architecture problems. The method predicts whether code
elements infected by a particular type of anomaly are likely to introduce architecture problems. The closer the value of the regression model is to 1, the higher is the likelihood that the code anomaly introduces an architecture problem. Differently from the Fisher test, in this method we only considered those cases where the cause-effect relationship between code anomalies and architecture problems were confirmed by architects and developers (Section IV.B). Details about the regression model can be found in [3].

We use regression models as an alternative to the Analysis Of Variance (ANOVA) for dichotomous variables. Then, for each code anomaly and for the 40 analyzed versions, we count the number of times that the p-values obtained by the regression model were significant. In this sense, as the regression model demands, we discarded variables that were highly correlated to others. Therefore, the model contains a non-redundant set of code anomalies.

Tables V and VI show the results of the logistic regression model for the contribution of each type of code anomaly on architecture problems. In particular, these tables summarize the total number of analyzed releases for which each type of anomaly was statistically-significant in the regression model. For instance, Table VI presents that Long Method was statistically-significant for causing architectural anomalies in 8 of out 10 versions of HW. We highlighted in boldface those anomalies that present a significant p-value for at least 70% of the releases where they occurred. This threshold was previously documented in the literature and used in other studies with similar analytical purposes [14].

A first analysis of Tables V and VI suggests that none of the types of code anomalies (Section II) stand out as the best indicator of architecture problems. None of them were clearly the best indicator across all the systems. However, certain types of anomalies were significantly related to architecture problems. In certain cases, the high cause-effect relation was consistently observed regardless of the analyzed system. For instance, 77% of God Classes, 71% of Long Methods, and 64% of Inappropriate Intimacy instances introduced architecture problems.

There were also situations where the high cause-effect relationship was observed in the majority of, albeit not all, the systems. This is the case of code anomalies such as Long Parameter List and Small Class, which presented interesting and distinct effects. For instance, about 63% and 22% of Long Parameter List introduced architectural violations in later versions of aspect-oriented and object-oriented systems, respectively. Artificial parameters had to be created through aspect-oriented system’s evolution. The goal was to allow aspects to access restricted contextual information, thereby breaking the intended encapsulation. This finding suggests that certain types of code anomalies could have been a typical source of architecture problems depending on the type of the adopted programming language and architecture decomposition; but it should be tested in further studies.

Our results reveal that no type stands out as the best indicator of the code anomaly’s harmfulness on architecture designs. Other characteristics associated with the code anomaly evolution, may be indicators of architecture problems. For instance, the incremental increase of the number of changes that a method suffers over time can indicate the presence of Divergent Change. This means that the harmful nature of certain types of code structures might be only confirmed over time. In this context, the next section analyzes whether the anomaly earliness was or not a better indicator of its impact on architecture designs.

2) Earliness of Code Anomalies

Interesting results emerged when analyzing the influence of early code anomalies on architecture degradation. We defined early code anomalies as those anomalies that appeared in the first version of each system. In order to analyze the impact of early code anomalies over system's evolution we relied on the use of history-sensitive tools for anomaly detection [23][31]. They allowed us to analyze to what extent changes performed on code elements, infected by early anomalies, resulted in architecture problems in later versions.

Harmfulness of Early Anomalies. Our analysis revealed that more than 18% of all architecturally-relevant anomalies emerged from anomalous code elements in the systems' first version. Although someone could consider that 18% is not a high percentage, these early code anomalies were critical to the architecture sustainability. They were related to more than 37% and 31% of all violations and architectural anomalies, respectively. In particular, early code anomalies induced architectural anomalies, which in turn led to violations. Both proportions point out the importance of understanding the impact of certain categories of early code anomalies. Thereby, developers should be able to identify at early stages occurrences of architecturally-relevant code anomalies to remove them by means of refactoring.

### TABLE V. SIGNIFICANT P-VALUES FOR VIOLATIONS

<table>
<thead>
<tr>
<th>Kinds of Code Anomalies</th>
<th>AW</th>
<th>HW</th>
<th>AM</th>
<th>MM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Divergent Change</td>
<td>5</td>
<td>7</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>Feature Envy</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>God Class</td>
<td>6</td>
<td>5</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>Inappropriate Intimacy</td>
<td>6</td>
<td>9</td>
<td>5</td>
<td>8</td>
</tr>
<tr>
<td>Long Method</td>
<td>7</td>
<td>8</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>Long Parameter List</td>
<td>7</td>
<td>2</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>Misplaced Class</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Shotgun Surgery</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>Small Class</td>
<td>4</td>
<td>7</td>
<td>3</td>
<td>6</td>
</tr>
</tbody>
</table>

### TABLE VI. SIGNIFICANT P-VALUES FOR ARCHITECTURAL ANOMALIES

<table>
<thead>
<tr>
<th>Kinds of Code Anomalies</th>
<th>AW</th>
<th>HW</th>
<th>AM</th>
<th>MM</th>
<th>MIDAS</th>
<th>PDP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Divergent Change</td>
<td>4</td>
<td>6</td>
<td>2</td>
<td>5</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Feature Envy</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>God Class</td>
<td>4</td>
<td>7</td>
<td>4</td>
<td>5</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Inappropriate Intimacy</td>
<td>4</td>
<td>7</td>
<td>5</td>
<td>5</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Long Method</td>
<td>5</td>
<td>8</td>
<td>4</td>
<td>7</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Long Parameter List</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Misplaced Class</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Shotgun Surgery</td>
<td>2</td>
<td>7</td>
<td>3</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Small Class</td>
<td>4</td>
<td>7</td>
<td>3</td>
<td>6</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>
Figure 2 depicts an example of the evolution of the method `MediaController.handleCmd`, an early **Long Method** extracted from the MM (Section IV.A). This figure shows the major changes performed on this method in terms of the evolution of: (i) coupling strength, and (ii) its contribution to implement system's concerns. The width of each arrow is linearly proportional to the cardinality of low-level dependencies abstracted in the corresponding edge [19]. Low-level dependencies, in this context, refer to method invocations, type references and attributes accesses that a given method executes from other classes. Figure 2 also relies on the *distribution map* [6], a technique for visualizing system's properties throughout evolution, where colors represent the dominant concern in each class.

As we can see, the number of concerns realized by modules increases over time. In some cases, the high number of concerns realized by the same module was introduced by certain code elements. For instance, all if statements defined in `MediaController.handleCmd` realize multiple concerns in version 8.0; which are represented by vertical bars of different colors. Hence, the method was responsible for `Controller` suffering from **Module Concern Overload** and **Scattered Parasitic Functionality** anomalies.

Additionally, `MediaController.handleCmd` handles more types of commands than it will actually process by accepting the parameter `c` of generic type `Cmd`. This generic parameter makes the method harder to understand because it does not reveal which commands the method is interested in. This situation worsened over time due to the incremental addition of commands. Therefore, the Ambiguous Interface was progressively introduced by the changes made in the method body.

**Characteristics of Harmful Early Anomalies.** Early code anomalies were recurrently the source of later relevant problems when they infected the API of critical points of the system's architecture. Examples of these critical points are modules' interfaces and super-classes. When code anomalies infect modules' interfaces, they might be propagated to the: (i) internal code of the module, and (ii) coupled modules over system's evolution. This situation occurred with 49% of all relevant code anomalies in HW, 42% in PDP, 38% in MM, 31% in AW and MIDAS and 24% in AM. These proportions were directly related to the number of code anomalies infecting critical points in each system.

Moreover, when anomalies infect super-classes they might be propagated from parents to children in inheritance trees. This situation occurred with 53% of all relevant code anomalies in MM, 50% in AM, 45% in HW, 36% in MIDAS, 33% in AW and 24% in PDP. The infected super-classes had a considerable negative effect as the architectural violations and architectural anomalies were propagated down through the class hierarchies. For instance, the class `MediaController` inherits the Ambiguous Interface occurrence from its parent; the class `Controller` (Figure 2). As `MediaController` overrides the `handleCmd` method, developers were forced to use if statements to handle the commands in which the method is interested.

Finally, it was observed that early code anomalies emerged in code elements that accessed information from classes defined in multiple architecture modules. These code elements also implemented several system's concerns. The method `MediaController.handle` is an example of a code element that suffers from this co-occurrence in MM. About 85% of these code elements introduced architecture problems in the analyzed systems. The issue is that these code elements were the subject of changes associated with each of the system’s concerns that they dealt with. Also, the increasing coupling strength was a reason why various changes were applied to this method during its evolution. The diverse nature of the changes confirmed the harmful impact of this co-occurrence on architecture design.

**C. Are Relevant Code Anomalies often Refactored?**

The aforementioned findings motivated us to investigate whether performed refactorings were targeted at resolving architecturally-relevant anomalies in the analyzed systems. To this end we analyzed two groups of refactorings: (i) **high-level** refactorings [28] that involve API-level changes – that is, structural modifications that affected interfaces and
services, and (ii) low-level refactorings [28] that imply narrowly-scoped changes, which do not affect the clients of the module being modified—for example, renaming local variables or extracting private methods fall in this category.

We focused on the analysis of both categories of refactoring. Even though there are attempts to automate their detection [5][15], such tools are not available yet. Therefore, we needed to partially rely on the manual inspection of the source code for identifying their occurrences. We analyzed the commit messages that indicated the occurrence of a refactoring in a given revision. This analysis was possible because the majority of the commit messages followed a template, which allows developers to recover the purpose of the commit. Also, we relied on using diff tools to analyze refactorings when commit messages were absent. By mixing both strategies we aimed at improving the reliability of our analysis. Finally, we analyzed a total of 217 refactorings in HW, 160 in MM, 112 in PDP, 97 in AW and 72 in AM.

Even though refactorings were often applied in the target systems, they rarely served to remove architecturally-relevant code anomalies. Refactorings were responsible for removing only up to 37% of all architecturally-relevant code anomalies. This means that the majority of architecturally-relevant code anomalies were left in the code as the system evolved. Our investigation revealed that this may be caused by three main reasons: (i) a low frequency of high-level refactorings, (ii) a high frequency of low-level refactorings, and (iii) the inability of current mechanisms detecting architecturally-relevant code anomalies. Even though these were not the unique reasons on why architecture problems were left in the code, they were the most frequent ones.

High-level vs. Low-Level Refactorings. Only 73 of applied refactorings were of high-level nature in HW, 41 in MM, 12 in AM, 9 in PDP and 7 in AW. That is, less than 34% of refactorings were of high-level nature, which implies that the impact of a few refactorings was of wide scope. The most high-level refactorings were: move public member (16%) and extract class or extract superclass (12%). As it can be noticed, they represent a minority of the total number of applied refactorings. These refactorings are stronger candidates to the removal of architecturally-relevant code anomalies as they modify the code element’s API. However, their application was often confined to later versions where instabilities clearly achieved critical stages. We considered that instabilities achieved critical stages when changes need to be performed across many code elements belonging to multiple modules in order to add the new features.

This observation suggests that developers chose to focus their effort on architecturally-relevant refactorings on specific versions. This strategy prevailed in all systems over the option of distributing the effort through consecutive versions. For instance, the highest number of high-level refactorings in MM was applied in version 7.0 in order to support the inclusion of different requirements. Otherwise, changes associated with such requirements would be scattered and duplicated in many elements belonging to the Controller module. Thus, several classes and super-classes were extracted and many public methods changed their APIs.

A possible reason for the rare implementation of high-level refactorings is that developers are not equipped with proper tools for automating this task. High-level refactorings are associated with changes on code element’s interfaces, which, in some cases, belong to different architecture modules. The application of these refactorings, without proper tooling support, may imply in higher risks such as: unexpected breaks on the client code or undesirable semantic changes. This could be one of the reasons why developers need to rely on tools for applying this kind of refactoring rather than apply it manually. However, usability problems is one of the reasons pointed out by recent studies about why developers seldom use refactoring tools [2][28]. Specifically their inability to properly visualize the effects of an applied refactoring was pointed out as a major problem [2][28].

On the other hand, more than 60% of refactorings were of low-level nature. This implies that the impact of most refactorings was of narrow scope. The most applied low-level refactorings were: rename private members (32%), extract local variable (16%), and move private members (12%). As we can notice, they represent a high proportion of the total number of refactorings applied. However, these refactorings did not contribute significantly to remove architecturally-relevant code anomalies. The reason is that modifications were often confined to the internal code of the class, whereas the removal of architecturally-relevant code anomalies requires modifications in several classes.

VI. THREATS TO VALIDITY

Threats to construct validity. A first construct validity threat concerns the way we associate code anomalies with architectural problems. We are aware that code anomalies might be accidentally related to architecture problems. However, we limited such a threat by considering only the code anomalies and architecture problems whose cause-effect relationship was identified and confirmed by developers and architects. Another threat concerns the set of analyzed code anomalies and architecture problems. We have tried to mitigate this threat by using systems that suffer from the same set of anomalies that systems used in other studies [4][14]. Lastly, construct validity concerns how refactorings were identified. As we have relied on commit heuristics and diff tools, some refactorings might be missing.

Threats to conclusion and external validity. The former concerns the relationship between the treatment and the outcome. In our study all the Fisher test and logistic regression model results were statistically-significant at the 95% level. On the other hand, threats to external validity concern the generalization of the findings. We focused on the analysis of medium-size systems. However, the development processes of large-scale systems might differ and lead to different results. We have tried to use systems of different types, implemented using different programming languages, environments (i.e. academy and industry) and with different architecture decompositions. Our target systems also present a similar density of code anomalies to large systems that were used in previous studies [4][14][29].
Another external validity threat concerns to obtain better generalization of the results. We plan to apply this kind of study to large open-source and industry systems.

VII. CONCLUDING REMARKS

Our results showed that the majority of the architecture problems in the source code emerged from anomalous code elements. These results suggest that systematic removal of code anomalies can be used to effectively combat symptoms of architecture degradation in the code. Our study also revealed how certain kinds of early code anomalies cause adverse impact on the architecture design as the systems evolve. This means that developers should promptly identify and address them upfront; otherwise, those code anomalies are likely to contribute to the anomaly in coupled modules, thereby accelerating the architecture degradation processes.

The aforementioned finding raises the concern about the effectiveness of state-of-the-art history-sensitive tools for code anomaly detection [23][31]. These tools rely on change analysis across several system versions to detect anomalies with acceptable accuracy [23][31]. Then, they cannot reveal harmful anomalies and their detection might occur too late, when anomaly removal might become imperative. This problem is exacerbated by the fact that a recent study [20] has showed that state-of-art mechanisms are not accurate to detect architecturally-relevant code anomalies. More than 60% of the detected anomalies seldom introduce any architecture problem. Also, over 50% of code anomalies not detected by these mechanisms (false negatives) were found to be the main sources of architecture problems.

This inefficiency of existing tools is probably one of the reasons that developers were barely able to remove architecturally-relevant anomalies via refactorings. Even during architecture-driven code reviews, developers are wasting their time by checking and restructuring code that do not represent threats to the system’s architecture. Even worse, developers might neglect code anomalies that are critical to the architecture sustainability. Therefore, our findings may indicate the need for better tools to: (i) support developers in locating architecturally-relevant code anomalies, and (ii) apply API refactorings that effectively contribute to address architecture degradation symptoms.

REFERENCES

[33] Sonar: http://docs.codehaus.org/display/SONAR/
[38] Xing, Z. and Stroulia, E. Refactoring practice: How it is and how it should be supported. In Proc of the 22sd ICSM 2006