Prioritization of Code Anomalies based on Architecture Sensitiveness

Roberta Arcoverde, Everton Guimarães, Isela Macía, Alessandro Garcia
Informatics Department, PUC-Rio
Rio de Janeiro, Brazil
{rarcoverde, eguimaraes, ibertran, afgarcia}@inf.puc-rio.br

Yuanfang Cai
Department of Computer Science
Drexel University
Philadelphia, USA
yfcai@cs.drexel.edu

***Abstract*— Code anomalies are symptoms of software maintainability problems, particularly harmful when contributing to architectural degradation. Despite the existence of many automated techniques for code anomaly detection, identifying the code anomalies that are more likely to cause architecture problems remains a challenging task. Even when there is tool support for detecting code anomalies, developers often invest a considerable amount of time refactoring those that are not related to architectural problems. In this paper we present and evaluate four different heuristics for helping developers to prioritize code anomalies, based on their potential contribution to the software architecture degradation. Those heuristics exploit different characteristics of a software project, such as change-density and error-density, for automatically ranking code elements that should be refactored more promptly according to their potential architectural relevance. Our evaluation revealed that software maintainers could benefit from the recommended rankings for identifying which code anomalies are harming architecture the most, helping them to invest their refactoring efforts into solving architecturally relevant problems.**

Keywords*— *Code anomalies, Architecture degradation and Refactoring;

I. INTRODUCTION

Code anomalies, commonly referred to as *code smells* [9], are symptoms in the source code that may indicate deeper maintainability problems. The presence of code anomalies often represents structural problems, which make code harder to read and maintain. Those anomalies can be even more harmful when they impact negatively on the software architecture design [4]. When that happens, we call those anomalies *architecturally relevant,* as they represent symptoms of architecture problems. Moreover, previous studies [14][35] have confirmed that the progressive manifestation of code anomalies is a key symptom of *architecture degradation* [14]. The term architecture degradation is used to refer to the continuous quality decay of architecture design when evolving software systems. Thus, as the software architecture *degrades*, the maintainability of software systems can be compromised irreversibly. As examples of architectural problems, we can mention Ambiguous Interface and Component Envy [11], as well as cyclic dependencies between software modules [27].

In order to prevent architecture degradation, software development teams should progressively improve the system maintainability by detecting and removing architecturally relevant code anomalies [13][36]. Such improvement is commonly achieved through *refactoring* [6][13] - a widely adopted practice [36] with well known benefits [29]. However, developers often focus on removing – or prioritizing - a limited subset of anomalies that affect their projects [1][16]. Furthermore, most of the remaining anomalies are architecturally relevant [20]. Thus, when it is not possible to distinguish which code anomalies are architecturally relevant, developers can spend more time fixing problems that are not harmful to the architecture design. This problem occurs even in situations where refactoring need to be applied in order to improve the adherence of the source code to the intended architecture [1][19][20].

Several code analysis techniques have been proposed for automatically detecting code anomalies [18][25][28][32]. However, none of them help developers to prioritize anomalies with respect to their architectural relevance, as they present the following limitations: first, most of these techniques focus on the extraction and combination of static code measures. The analysis of the source code structure alone is often not enough to reveal whether an anomalous code element is related to the architecture decomposition [19][20]. Second, they do not provide means to support the prioritization or ranking of code anomalies. Finally, most of them disregard: (i) the exploitation of software project factors (i.e. frequency of changes and number of errors) that may be an indicator of the architectural relevance of a module, and (ii) the role that code elements play in the architectural design.

In this context, this paper proposes four prioritization heuristics to help identifying and ranking architecturally relevant code anomalies. Moreover, we assessed the accuracy of the proposed heuristics when ranking code anomalies based on their architecture relevance. The assessment was carried out in the context of four target systems from heterogeneous domains, developed by different teams using different programming languages. Our results show that the proposed heuristics were able to accurately prioritize the most code relevant anomalies of the target systems mainly for scenarios where: (i) there were architecture problems involving groups of

DOI 10.1109/SBES.2013.14

classes that changed together; (ii) changes were not predominantly perfective; (iii) there were code elements infected by multiple anomalies; and (iv) the architecture roles are well-defined and have distinct architectural relevance.

The remainder of this paper is organized as follows. Section II introduces the concepts involved in this work, as well as the related work. Section III introduces the study settings. Section IV describes the prioritization heuristics proposed in this paper. Section V presents the evaluation of the proposed heuristics, and Section VI the evaluation results. Finally, Section VII presents the threats to validity, while Section VIII discusses the final remarks and future work.

II. Background and Related Work

This section introduces basic concepts related to software architecture degradation and code anomalies (Section II.A). It also discusses researches that investigate the interplay between code anomaly and architectural problems (Section II.B). Finally, the section introduces existing ranking systems for code anomalies (Section II.C).

A. Basic Concepts

One of the causes for *architecture degradation* [14] on software projects is the continuous occurrence of code anomalies. The term code anomaly or *code smell* [9] is used to define structures on the source code that usually indicate maintenance problems. As examples of code anomalies we can mention Long Methods and God Classes [9]. In this work, we consider a code anomaly as being *architecturally relevant* when it has a negative impact in the system architectural design. That is, the anomaly is considered relevant when it is harmful or related to problems in the architecture design. Therefore, the occurrence of an architecturally relevant code anomaly can be observed if the anomalous code structure is directly realizing an architectural element exhibiting an architecture problem [19-21].

Once a code anomaly is identified, the corresponding code may suffer some refactoring operations, so the code anomaly is correctly removed. When those code anomalies are not correctly detected, prioritized and removed in the early stage of software development, the ability of these systems to evolve can be compromised. In some cases, the architecture has to be completely restructured. For this reason, the effectiveness of automatically detected code anomalies using strategies has been studied under different perspectives [16][18][19][26][31]. However, most techniques and tools disregard software project factors that might indicate the relevance of an anomaly in terms of its architecture design, number of errors and frequency of changes. Moreover, those techniques do not help developers to distinguish which anomalous element are architecturally harmful without considering the architectural role that a given code element plays on the architectural design.

B. Harmful Impact and Detection of Code Anomalies

The negative impact of code anomalies on the system architecture has been investigated by many studies in the state-of-art. For instance, the study developed in [23] reported that the Mozilla's browser code was overly complex and tightly coupled therefore hindering its maintainability and ability to evolve. This problem was the main cause of its complete reengineering, and developers' took about five years to rewrite over 7 thousand source files and 2 million lines of code [12]. Another study [7] showed how the architecture design of a large telecommunication system degraded in 7 years. Particularly, the relationship between the system modules increased over the time. This was the main cause why the system modules were not independent anymore, and as consequence, further changes were not possible. Finally, a study performed in [14] investigated the main causes for architecture degradation. As a result, the study indicated that refactoring specific code anomalies could help to avoid it. Another study [35] has identified that duplicated code was related to design violations.

In this sense, several detection strategies have been proposed in order provide means for the automatic detection of code anomalies [18][25][28]. However, most of them is based on source code information and relies on a combination of static code metrics and thresholds into logical expressions. This is the main limitation of those detection strategies, since they disregard architecture information that could be exploited to reveal architecturally relevant code anomalies. In addition, current detection strategies only consider individual occurrences of code anomalies, instead of analyzing the relationships between them. Such limitations are the main reasons why the current detection strategies are not able to support the detection of code anomalies responsible for inserting architectural problems [19]. Finally, a recent study [19] investigated to what extent the architecture sensitive detection strategies can better identify code anomalies related to architectural problems [22].

C. Ranking Systems for Code Anomalies

As previously mentioned, many tools and techniques provide support for automatically detecting code anomalies. However, the number of anomalies tends to increase as the system grows and, in some cases, the high number of anomalies can be unmanageable. Moreover, maintainers are expected to choose which code anomalies should be prioritized. Some of the reasons why this is necessary are (i) time constraints and (ii) attempts to find the correct solution when restricting a large system in order to perform refactoring operations to solve those code anomalies. The problem is that the existing detection strategies do not focus on ranking or prioritizing code anomalies. Nevertheless, there are two tools that provide ranking capabilities for different development platforms: Code Metrics and InFusion.

The first tool is a .NET based add-in for the Visual Studio development environment and it is able to calculate a limited set of metrics. Once the metrics are calculated, the tool assigns a "maintainability index" score to each of the analyzed code elements. This score is based on the combination of the metrics for each code element. The second tool is the InFusion, which can be used for analyzing Java, C and C++ systems. Moreover, it allows calculating more than 60 metrics. Besides the statistical analysis for calculating code metrics, it also provides numerical scores in order to detect the code

anomalies. Those scores provide means to measure the negative impact of code anomalies in the software system. When combining the scores, a deficit index is calculated for the entire system. The index takes into consideration size, encapsulation, complexity, coupling and cohesion metrics.

However, the main concern of using these tools is that the techniques they implement have some limitations: (i) usually it only considers the source code structure as input for detecting code anomalies; (ii) the ranking system disregards the architecture role of the code elements; and (iii) the user cannot define or customize their own criteria for prioritizing code anomalies. In this sense, our study proposes prioritization heuristics for ranking and prioritizing code anomalies. Moreover, our heuristics are not only based on source code information for detecting code anomalies. It also considers information about architecture relevance of the detected code anomalies. For this, we analyze different properties of the source code they affect, such as information about changes on software modules, bugs observed during the system evolution and the responsibility of module in the system architecture.

III. STUDY SETTINGS

This section describes our study hypotheses and variables selection, as well as the target systems used to evaluate the accuracy of the proposed heuristics. The main goal of this study is to evaluate whether the proposed heuristics for prioritization of architecturally relevant code anomalies can help developers on the ranking and prioritization process. It is important to emphasize that the analysis of the proposed heuristics is carried out in terms of accuracy. Also, Table I defines our study using the GQM format [34].

TABLE I. STUDY DEFINITION USING GQM FORMAT

GQM (Goal, Question, Metric)	
Analyze:	The proposed set of prioritization heuristics.
For the purpose of:	Understanding their accuracy for ranking code anomalies based on their architecture relevance.
With respect to:	Rankings previously defined by developers or maintainers of each analyzed system.
From the viewpoint of:	Researchers, developers and architects
In the context of:	Four software systems from different domains with different architectural designs.

Our study was basically performed in three phases: (i) as we have proposed prioritization heuristics for ranking code anomalies, in the first phase we performed the detection and classification of code anomalies according to their architecture relevance for each of the target systems. For such detection, we used a semi-automatic process based on strategies and thresholds, which has been broadly used on previous studies [2][11][16][20]; (ii) in the second phase, we applied the proposed heuristics and computed their scores for each detected code anomaly. The output of this phase is an ordered list with the high-priority anomalies; finally, (iii) in the third phase, we compared the heuristics results with rankings previously defined by developers or maintainers of each target system. The ranking list provided by developers represents the "ground truth" data in our analysis, and were produced manually.

A. Hypotheses

In this section, we describe the study hypothesis in order to test the accuracy of the proposed heuristics for ranking code anomalies based on their relevance. First, we have defined some thresholds of what we consider as an acceptable accuracy: (i) low accuracy, 0-40%; (ii) acceptable accuracy, 40-80%; and (iii) high accuracy, 80-100%. These thresholds are based on the ranges defined in [37], where the values are applied in statistical tests (e.g. Pearson's correlation). We adapted these values in order to better evaluate our heuristics, since we are only interested in values that indicate a high correlation.

Moreover, we analyzed the three levels of accuracy in order to investigate to what extent the prioritization heuristics would be helpful. For instance, a heuristic with an accuracy level of 50% means the ranking produced by the heuristic should be able to identify at least half of the architecturally relevant code anomalies. In order to test the accuracy of the prioritization heuristics, we have defined 4 hypotheses (see Table II).

TABLE II. STUDY HYPOTHESES

Hypothesis		**Description**
H1	$H_{1.0}$	The change-density heuristic cannot accurately identify architecturally relevant code anomalies ranked as top ten.
	$H_{1.1}$	The change-density heuristic can accurately identify architecturally relevant code anomalies ranked as top ten.
H2	$H_{2.0}$	The error-density heuristic cannot accurately identify architecturally relevant code anomalies ranked as top ten.
	$H_{2.1}$	The error-density heuristic can accurately identify architecturally relevant code anomalies ranked as top ten.
H3	$H_{3.0}$	The anomaly density heuristic cannot accurately identify architecturally relevant code anomalies ranked as top ten
	$H_{3.1}$	The anomaly density heuristic can accurately identify architecturally relevant code anomalies ranked as top ten
H4	$H_{4.0}$	The architecture role heuristic cannot accurately identify architecturally relevant code anomalies ranked as top ten.
	$H_{4.1}$	The architecture role heuristic can accurately identify architecturally relevant code anomalies ranked as top ten

B. Target Systems

In order to test the study hypotheses, we selected 4 target systems from different domains: (i) MIDAS [24], a lightweight middleware for distributed sensors application; (ii) Health Watcher [13], a web system used for registering complaints about health issues in public institutions; (iii) PDP, a web application for managing scenographic sets in television productions; and (iv) Mobile Media [6], a software product line that manages different types of media in mobile devices. All the selected target systems have been previously analyzed in other studies that address problems such as architectural degradation and refactoring [11][20].

The target systems were selected based on 4 criteria: (i) the availability of either architecture specification or original developers. The architectural information is essential to the application of the *architecture role* heuristic, which directly depends on architecture information to compute the ranks of code anomalies; (ii) availability of the source version control systems of the selected applications; the information for the version control system provides input for the change-*density*

heuristic; (iii) availability of an issue tracking system. Although this is not a mandatory criterion, it is highly recommended for providing input for the *error-density* heuristic; and (iv) the applications should present different design and architectural structures. This restriction allows us to better understand the impact of the proposed heuristics for a diverse set of code anomalies, emerging from different architectural designs.

IV. Prioritization of Code Anomalies

In this section, we describe 4 prioritization heuristics proposed in this work. These heuristics are intentionally simple, in order to be feasible on most software projects. Their main goal is to help developers on identifying and ranking architecturally relevant code anomalies.

A. Change Density Heuristic

This heuristic is based on the idea that anomalies infecting unstable code elements are more likely to be architecturally relevant. An unstable element can be defined as a code element that suffers from multiple changes during the system evolution [15]. In some cases, for instance, changes occur in cascade and affect the same code elements. Those cases are a sign that such changes are violating the "open-closed principle", which according to [27] is the main principle for the preservation of architecture throughout the system evolution. In this sense, the *change-density* heuristic calculates the ranking results based on the number of changes performed on the anomalous code element. The *change-density* heuristics is defined as follows: given a code element *c*, the heuristic will look for every revision in the software evolution path where the element *c* has been modified. That is, the number of different revisions represents the number of changes performed in the element. Thus, the higher the number of changes, the higher is the element priority.

The only input required for this heuristic is the change sets that occurred during the system evolution. The change sets is composed by the list of existing revisions and the code elements that were modified on each revision. For this heuristic, we are only able to calculate the changes performed to an entire file. For this scoring mechanism, all code anomalies presented in the same file will receive the same score. We adopted this strategy in our heuristics because none of the studied code anomalies emerged as the best indicator of architectural problems across the systems [20]. However, it is possible to differentiate between two classes by ranking those that have changed most as high-priority. In order to calculate the score for each code anomaly, the heuristic assign the number of changes that were performed in the infected class. Once the number of changes was computed, we ordered the list of resources and their respective number of changes, thus producing our final ranking. This information was to extract the change log from the version control systems for each of the target applications.

B. Error-Density Heuristic

This heuristic is based on the idea that code elements that have a high number of errors observed during the system evolution might be considered high-priority. The *error-density* heuristic is defined as follows: given a resolved bug *b*, the heuristic will look for code elements *c* that was modified in order to solve *b*. Thus, the higher the number of errors solved as a consequence of changes applied to *c*, the higher is the position in the prioritization ranking.

This heuristic requires two different inputs: (i) change log inspection – our first analysis was based on change log inspection, looking for common terms like *bug* or *fix*. Once those terms are found on commit messages, we incremented the scores for the classes involved in a given change. This technique has been successfully applied in other relevant studies [17]; and (ii) bug detection tool – as we could not rely on the change log inspection for all system, we have decided to use a bug detection tool, namely *findBugs*, for automatically detecting blocks of code that could be related to bugs. Once possible bugs are identified, we collect the code elements causing them and increment their scores. Basically, the heuristic works as follows: (i) firstly, the information of bugs that were fixed is retrieved from the revisions; (i) after that, the heuristic algorithm iterates over all classes changes on those revisions and the score is incremented for each anomaly that infect the classes. In summary, when a given class is related to several bug fixes, the code anomaly will have a high score.

C. Anomaly Density Heuristic

This heuristic is based on the idea that each code element can be affected by many anomalies. Moreover, a high number of anomalous elements concentrated in a single component indicate a deeper maintainability problem. In this sense, the classes internal to a component with a high number of anomalies should be prioritized. Furthermore, it is known that developers seem to care less about classes that present too many code anomalies [27], when they need to modify them. Thus, anomalous classes tend to remain anomalous or get worse as the systems evolve. Thus, prioritizing classes with many anomalies should avoid the propagation of problems. This heuristic might also be worthy when classes have become brittle and hard to maintain due to the number of anomalies infecting them.

Computing the scores for this heuristic was rather straightforward. Basically, it calculates the number of anomalies found per code element. Thus, we consider that elements with a high number of anomalies are high-priority targets for refactoring. The *anomaly density* heuristic is defined as follows: given a code element *c*, the heuristic will look to the number of code anomalies that *c* contains. Thus, the higher the number of anomalies found in *c*, the higher would be the ranking in the prioritization heuristic result. This heuristic requires only one input: the set of detected code anomalies for each code element in the system. Moreover, the heuristic can be customized to compute only architecture relevant anomalies, instead of computing the set of all the anomalies infecting the system. In order to define whether an anomaly is relevant or not, our work relies on the detection mechanisms provided by SCOOP tool [21].

D. Architecture Role Heuristic

Finally, this heuristic proposes a ranking mechanism based on the architectural role a given class plays in the system. The fact is that, when the architecture information is available, the architectural role influences the priority level. The *architecture role* heuristic is defined as follows: given a code element *c*, this heuristic will examine the architectural role *r* performed by *c*. The relevance of the architectural role in the system represents the rank of *c*. In other words, if *r* is defined as a relevant architecture role and it is performed by *c*, the code element *c* will be ranked as high priority.

The architecture role heuristic depends on two kinds of information, regarding the system's design: (i) which roles each class plays in the architecture; and (ii) how relevant those roles are towards architecture maintainability. For this study setting, we first had to leverage architecture design information in order to map code elements to their architecture roles. Part of this information extraction had already been performed on our previous studies [19][20]. Then, we asked the original architects to assign different levels of importance to those roles, according to the architecture patterns implemented. Moreover, we defined score levels to each architecture role. For doing so, we considered the number of roles identified by the architects, and distributed them according to a fixed interval from 0 to 10. Code anomalies that infected elements playing critical architecture roles were assigned to the highest score. On the other hand, when the code anomaly affected elements related to less critical architecture roles, they would be assigned to lower scores, according to the number architecture roles provided by the original architects.

V. EVALUATION

This section describes the main steps for evaluating the proposed heuristics, as well as testing the study hypotheses. The evaluation is organized into three main activities: (i) detect of code anomalies; (ii) identify of the rankings representing the ground truth; and (iii) collect scores for each anomaly under the perspective of the prioritization heuristics.

A. Detecting Code Anomalies

The first step was the automatic identification of code anomalies for each of the 4 target systems by using well-known detection strategies and thresholds [16][31]. These detection strategies and thresholds used in our study have been used previously in other studies [6][19][20]. The metrics required by the detection strategies are mostly collected with current tools [30][33]. After that, the list of code anomalies is checked and refined by original developers and architects of each target system. Through this validation we can make sure that results produced by the detection tools do not include false positives [19].

We have also a defined *ground truth ranking* in order to compare the results of the analysis provided by the software architects and the resulting ranking provided by each of the proposed heuristics. The *ground truth ranking* is a list of anomalous elements in the source code ordered by their architecture relevance, defined by the original architects of each target application. Basically, the architects were asked to provide an ordered list of the top ten classes that, in their beliefs, represented the main sources of maintainability problems of those systems. Besides providing a list of the high priority code elements, the architects were also asked to provide information regarding the architectural design of each target system. That is, they should provide a list of architectural roles presented in each target system ordered by their relevance from the architecture perspective.

B. Analysis Method

After applying the heuristics, we compared the rankings produced by each of them with the *ground truth ranking*. We decided to analyze only the top ten code elements ranked, for three main reasons: (i) it would be unviable if we have asked developers to rank an extensive list of elements; (ii) we wanted to evaluate our prioritization heuristics mainly for their abilities to improve refactoring effectiveness. Thus, the top ten anomalous code elements represent a significant sample of elements that could possibly cause architecture problems; and (iii) we focused on analyzing the top 10 code elements for assessing whether they represent a useful subset of sources of architecturally relevant anomalies.

In order to analyze the rankings provided by the heuristics, we have considered three measures: (i) Size of overlap – measures the number of elements that appear both in the ground truth ranking and in the heuristic ranking. It is fairly simple to calculate and tells us whether the prioritization heuristics are accurately distinguishing the top *k* items from the others; (ii) Spearman's footrule [5] – it is a well-known metric for permutations. It measures the distance between two ranked lists by computing the differences in the rankings of each item; and (iii) Fagin's extension to the Spearman's footrule [8] – it is an extension to Spearman's footrule for top *k* lists. Fagin extended Spearman's footrule by assigning an arbitrary placement to elements that belong to one of the lists but not to the other. Such placement represents the position in the resulting ranking for all of the items that do not overlap when comparing both lists.

It is important to notice the main differences between the three measures: the number of overlaps indicates how effectively our prioritization heuristics are capable of identifying a set of *k* relevant code elements, disregarding the differences between them. This measure becomes more important as the number of elements under analysis grows. Thus, the number of overlaps might give us a good hint on the heuristics capability for identifying good refactoring candidates, disregarding the differences between them. The purpose of the other two measures is to analyze the similarity between two rankings. Unlike the number of overlaps, they take into consideration the positions each item has in the compared rankings. It is important to mention the main differences between those two measures: when calculating Spearman's footrule, we are only considering the overlapping items. When the lists are disjoint, the original ranks are lost, and a new ranking is produced. On the other hand, Fagin's measure takes into consideration the positions of the overlapping elements in the original lists. Finally, we used the measures results to calculate the similarity accuracy – as defined in our hypotheses.

VI. EVALUATING THE PROPOSED PRIORITIZATION HEURISTICS

The evaluation of the proposed heuristics involved two separated activities: (i) quantitative analysis on the similarity results; and (ii) quantitative evaluation of the results regarding their relations to actual architecture problems.

A. Change-Density Heuristic

Evaluation. This heuristic was applied in 3 out of the 4 target applications selected in our study. Our analysis was based on different versions of Health Watcher (10 versions), Mobile Media (8 versions) and PDP (409 versions). Our goal was to check whether the prioritization heuristics performed well or not on systems with shorter and longer longevity. Additionally, it was not a requirement to only embrace projects with long histories, once we wanted also to evaluate whether the heuristics would be more effective in preliminary versions of a software system. Table III shows the evolution characteristics analyzed for each system.

TABLE III. CHANGE CHARACTERISTICS FOR EACH TARGET APPLICATION

Name	CE	N-Revisions	M-Revisions	AVG
Health Watcher	137	10	9	1.5
Mobile Media	82	9	8	2.6
PDP	97	409	74	8.8

As we can observe, Mobile Media and Health Watcher presented similar evolution behaviors. As the maximum number of revisions (M-Revisions) was limited to the total number of revisions for a system (AVG), neither Health Watcher nor Mobile Media could have 10 or more versions of a code element (CE). We can observe that Health Watcher had more revisions than Mobile Media. However, those changes were scattered between more files. Due to the reduced number of revisions available for both systems, we have established a criterion for selecting items when there were ties in the top 10 rankings. For instance, we can use alphabetical order when the elements in the *ground truth* are ranked equally harmful.

TABLE IV. RESULTS FOR THE CHANGE-DENSITY HEURISTIC

Name	Overlap		NSF		NF	
	Value	Accuracy	Value	Accuracy	Value	Accuracy
HW	8	57%	0.62	38%	0.87	13%
MM	5	50%	1	0%	0.89	11%
PDP	6	60%	0.44	56%	0.54	46%

Table IV show the results observed when analyzing the *change-density* heuristic. As we can observe, the highest absolute overlap value was obtained for Health Watcher. It can be explained by the fact that the Health Watcher system has many files with the same number of changes. In this sense, when computing the scores we did not consider only the 10 most changed files, as that approach would discard files with as many changes as the ones selected. So, we decided to select 14 files, where the last 5 presented exactly the same number of changes. Moreover, the Health Watcher presented the highest number of code elements, having a total of 137 items (see Table III) that could appear on the ranking produced by applying the heuristic. Another interesting finding was observed in the Mobile Media system. Although the change-density heuristic detected 5 overlaps, all of them were shifted exactly two positions, thus resulting in the 1 value for the NSF measure. On the other hand, when we considered the non-overlaps, the position for one item matched. Moreover, the results show us that the NSF measure is not adequate when the number of overlaps is small.

When we compare the results of Mobile Media and Health Watcher to those obtained by PDP, we observed a significant difference. All PDP measures performed above our acceptable similarity threshold, which means a similarity value higher than 45%. For this case, we observed that the similarity was related to a set of classes that were deeply coupled: an interface acting as Facade and three realizations of this interface, implementing a server module, a client module and a proxy. When performing changes on the interface, many other changes were triggered in those three classes. For this reason, they have suffered many other modifications during the system evolution. Moreover, the nature of changes that the target applications underwent is different. For instance, on Health Watcher most part of changes was perfective (changes made aiming to improve the overall structure of the application). On the other hand, on Mobile Media, most part of the changes was related to the addition of new functionalities, which was also the case for PDP. However, we observed that Mobile Media had also low accuracy rates.

In summary, the results on applying the *change-density* heuristic showed us that it could be useful for detecting and prioritizing architecturally relevant anomalies in the following scenarios: (i) there are architecture problems involving groups of classes changing together; (ii) there are problems in the architecture related to Facade or communication classes; and (iii) changes were predominantly perfective. In this sense, from the results observed in the analysis, we can reject the null hypothesis H1. The fact was that the *change-density* heuristic was able to produce rankings for PDP with at least acceptable accuracy in all the analyzed measures.

Correlation with Architectural Problems. Based on the results produced by the *change-density* heuristic, we also needed to evaluate whether there is a correlation between the rankings with architectural problems. In this sense, we performed the analysis by observing which ranked elements are related to actual architectural problems (see Table V). We can observe that elements containing architecturally relevant anomalies (Arch-Relevant) were likely to be change-prone. For PDP system, all of the top 10 most changed elements were related to architectural problems. Also, if we consider that PDP has 97 code elements, and 37 of them are related to architectural problems, the results give us a hint that change-density is a good heuristic for detecting them.

TABLE V. RESULTS FOR THE CHANGE-DENSITY HEURISTIC VS. ARCHITECTURAL PROBLEMS

Name	N-ranked CE	Arch-Relevant	% of Arch-Relevant
HW	14	10	71%
MM	10	7	70%
PDP	10	10	100%

B. Error-Density Heuristic

Evaluation. This heuristic is based on the assessment of bugs that are introduced by a code element. So, the higher the number of bugs observed in a code element, the higher is its priority. Thus, in order to correctly evaluate the results produced by the *error-density* heuristics, a reliable set of detected bugs should be available for each target system. This was the case for the PDP system, where the set of bugs was well documented. On the other hand, for Mobile Media and Health Watcher, where the documentation of bugs was not available, we relied on the analysis of bug detection tools. The results of applying the *error-density* heuristics are showed in Table VI. It is important to highlight that for Health Watcher there were 14 ranked items, due to ties between some of them. Nevertheless, Health Watcher presented the highest overlap measures. That happens because the detected bugs were related to the behavior observed in every class implementing the *Command* pattern. Furthermore, each of the classes implementing this pattern was listed a high-priority in the *ground-truth ranking*.

TABLE VI. RESULTS FOR THE ERROR-DENSITY HEURISTIC

Name	Overlap		NSF		NF	
	Value	Accuracy	Value	Accuracy	Value	Accuracy
HW	10	71%	0	100%	0.74	26%
MM	3	30%	0	100%	0.76	24%
PDP	5	30%	0.83	17%	0.74	26%

Another interesting finding we observed was that the priority order for overlapping elements was exactly the same as the one pointed out in the *ground truth*. However, the 4 remaining non-overlapping elements were the same 4 elements in the *ground truth ranking*. The fact that top 4 elements are not listed in the ranking list produced by the heuristic resulted in a low accuracy for NF measure. For the Mobile Media, we have applied the same strategy, but all the measures also presented low accuracies. Due to the small number of overlaps, the results for NSF may not confidently represent the heuristics' accuracy. Finally, for the PDP the results were evaluated in a different perspective once we considered the manually detected bugs. That is, the bugs were collected through its issue tracking system, instead of using automatic bug detection tools. However, even when we performed the analysis using a reliable set of bugs, the overall results presented low accuracy. That is, from the 5 non-overlapping items, 4 of them were related to bugs on utility classes. Once those classes were neither related to any particular architectural role, nor implementing an architecture component, they were not considered architecturally relevant.

Correlation with Architectural Problems. Based on the results produced by the error-density heuristic, we could investigate the correlation between the rankings with actual architectural problems. That is, we could analyze whether the error-density heuristics presented better results towards detecting architecturally relevant anomalies. Table VII presents the results from applying this heuristic. As we can see, at least 80% of the ranked elements were related to architecture problems for all the analyzed systems. Moreover, Health Watcher system reached the most significant results with 85% of the ranked elements related to architectural problems. When we take into consideration that the ranking for Health Watcher was composed of 14 code elements (instead of 10), this result is even more significant. As mentioned before, the rankings for Health Watcher and Mobile Media were built over automatically detected bugs. It means that even when formal bug reports are not available, the use of static analysis tool [3] for predicting possible bugs might be useful.

TABLE VII. RESULTS FOR THE ERROR-DENSITY HEURISTIC VS. ACTUAL ARCHITETURAL PROBLEMS

Name	N-ranked CE	Arch-Relevant	% of Arch-Relevant
HW	14	10	85%
MM	10	8	80%
PDP	10	8	80%

On the other hand, for the PDP system where we considered actual bug reports, the results were also promising. From the top 10 ranked elements, 8 were related to architecture problems. When we consider that PDP system has 97 code elements, with 37 of them related to architecture problems, it means that the remaining 29 were distributed among the 87 bottom ranked elements. Moreover, if we extend the analysis over the top 20 elements, we observe a better correlation factor. That is, in this case the correlation showed us that around 85% of the top 20 most error-prone elements were related to architecture problems.

C. Anomaly Density Heuristic

Evaluation. The *anomaly density* heuristic was applied to the 4 target systems selected in our study. We have observed good results in terms of accuracy on ranking the architecturally relevant anomalies. As we can see in Table VIII, good results were obtained not only on ranking the top 10 anomalies, but also on defining its positions. We observed that only 2 of 8 measures had low accuracy when compared to the thresholds defined in our work. Furthermore, the number of overlaps achieved by this heuristic can be considered highly accurate in 3 of the 4 target systems. This indicates that code elements affected by multiple code anomalies are often perceived as high priority. It did not occur only in the case of Health Watcher, where we observed only 5 overlaps. When analyzing the number of anomalies for each element on the ranking representing *ground truth*, we could observe that many of them had exactly the same number of code anomalies, namely 8. Also, it is important to mention that for this heuristic, in contrast to the *change-density* and *error-density* heuristics, we only considered the top 10 elements for the Health Watcher system - once there were not ties to be taken into consideration.

When analyzing the MIDAS system, we could not observe a significant number of overlaps, once 9 out of 10 elements appeared in both rankings. However, this fact was expected as the system is composed by only 21 code elements. Nevertheless, we observed that both NSF and NF presented a high accuracy, which means that the rankings were similarly ordered. Moreover, the NF measure presented a better result, which was influenced by the fact that the only mismatched element was ranked in the 10th position. On the other hand, when analyzing the Mobile Media we observed discrepant

results regarding two ranking measures. We found 59% of accuracy for the NSF measure, and 30% for the NF measure. This difference is also related to the position the non-overlapping elements in the ranking generated by the heuristic. Therefore, the ranks for those elements were assigned to *k+1* in the developers' list, which resulted in a huge distance from their original positions. It is also important to mention that those elements comprehended a data model class, a utility class and a base class for controllers.

TABLE VIII. RESULTS FOR THE ANOMALY DENSITY HEURISTIC

Name	Overlap		NSF		NF	
	Value	Accuracy	Value	Accuracy	Value	Accuracy
HW	5	50%	0.66	34%	0.54	46%
MM	7	70%	0.41	59%	0.7	30%
PDP	8	80%	0.37	63%	0.36	64%
MIDAS	9	90%	0.4	60%	0.20	80%

By analyzing the results for this heuristic, we observed that code elements infected by multiple code anomalies are often perceived as high priority. We also identified that many false positives could arise from utility classes, as those classes are often large and not cohesive. Finally, the results obtained in this analysis also helped us rejecting the null hypothesis H3 – as the *anomaly density heuristic* was able to produce rankings with at least acceptable accuracy in all of the systems we analyzed for at least one measure. Furthermore, we obtained a high accuracy rate for the MIDAS system in 2 out of 3 measures, which means 90% for the overlaps and 80% for NF.

Correlation with Architectural Problems. We also performed an analysis in order to evaluate whether the rankings produced by the *anomaly density* heuristic. However, when evaluating the results produced by this heuristic, we observed that they were not consistent if compared them with architecturally relevant anomalies. This is valid conclusion for all target systems.

TABLE IX. RESULTS FOR THE ANOMALY DENSITY HEURISTIC VS. ACTUAL ARCHITETURAL PROBLEMS

Name	N-ranked CE	Arch-Relevant	% of Arch-Relevant
HW	10	5	50%
MM	10	9	90%
PDP	10	8	80%
MIDAS	10	6*	60%*

For instance, Table IX shows that for the Health Watcher system only 5 out of the top 10 ranked elements were related to architectural problems. The 5 code elements related to architectural problems are exactly the same overlapping items between the compared ranks. It happens due the high number of anomalies, which are concentrated in a small numbers of elements that are not architecturally relevant. Moreover, all the 5 non-architecturally relevant elements were data access classes responsible for communicating to the database. For the MIDAS system, we observed that from the top 10 code elements with the higher number of anomalies, 6 were architecturally relevant. In addition, the MIDAS system has exactly 6 elements that contribute to the occurrence of architecture problems. So, we can say that the *anomaly density heuristic* correctly outlined all of them in the top 10 ranking.

D. Architecture Role Heuristic

Evaluation We analyzed 3 of the 4 systems in order to evaluate the *architecture role* heuristic. As we can observe (see Table X), PDP achieved the most consistent results regarding the three similarity measures. The heuristic achieved around 60% of accuracy when comparing the similarity between the rankings. Also, the PDP is the only system where it was possible to divide classes and interfaces in more than three levels when analyzing the architectural roles. For instance, Table XI illustrates the four different architectural roles defined on the PDP system.

TABLE X. RESULTS FOR THE ARCHITECTURE ROLE HEURISTIC

Name	Overlap		NSF		NF	
	Value	Accuracy	Value	Accuracy	Value	Accuracy
HW	4	40%	0.5	50%	0.72	28%
MM	6	60%	0.22	78%	0.41	59%
PDP	6	60%	0.33	67%	0.41	59%

TABLE XI. ARCHITECTURE ROLES IN PDP

Architecture Roles	Score	# of CE
Utility and Internal Classes	1	23
Presentation and Data access classes	2	28
Domain Model, Business classes	4	24
Public Interfaces, Communication classes, Facades	8	6

Based on the classification provided in Table XI, we can draw the architecture role heuristic ranking for PDP. As we can see, the ranking contains all of the 6 code elements (# of CE) from the highest category and 4 elements from the domain model and business classes. We ordered the elements alphabetically for breaking ties. Therefore, although 23 classes obtained the same score, we are only comparing 4 of them. However, it is important to mention that some of the elements ranked by the original architects belonged to the group of discarded elements. Once we have chosen a different approach, such as considering all the ties as one item, we would turn our top ten ranking into a list of 30 items and have a 100% overlap rate. On the other hand, we decided to follow a different score approach for Mobile Media and Health Watcher, by consulting original architects for each of the target applications. The architects provided us the architecture roles and their relevance on the system architecture. Once we identified which classes were directly implementing which roles, we were able to produce the rankings for this heuristic.

The worst results were observed in the Health Watcher system, where almost 20 elements were ties with the same scores. So, we first selected the top 10 elements, and broke the ties according to the alphabetic order. This led us to an unreal low number of overlaps, as some of the discarded items were present in the ground truth ranking. In fact, due to low number of overlaps, it would not be fair to evaluate the NSF measure as well. Thus, we performed a second analysis, considering the top 20 items instead of the top 10, for analyzing the whole set of elements that had the same score. In this second analysis, we observed the number of overlaps went up to 6, but the accuracy for the NSF measure decreased to 17% - which indicates a larger distance between the compared rankings. In addition, this also shows us that the 50% accuracy for NSF obtained in the first comparison round was

misleading, as expected, due the low number of overlaps. For the Mobile Media system, we observed high accuracy rates for both NSF and NF measures. Furthermore, we observed that several elements of the Mobile Media were documented as being of high priority on the implementation of architectural components. More specifically, there were 8 architecture components described in that document directly related to 9 out of the top 10 high priority classes.

It is important to notice that the results for this heuristic are dependent on the quality of the architecture roles defined by the software architect. Moreover, we observed that PDP system achieved the best results, even with multiple architecture roles defined, as well as different levels of relevance. Finally, we conclude that the results of applying the *architecture role* heuristic helped to reject the null hypothesis H4. In other words, the heuristic was able to produce rankings with at least acceptable accuracy in all of the target applications.

Correlation with Architectural Problems. Similarly to the other heuristics, we have also evaluated whether the rankings produced by the *architecture role* heuristic are related to actual architectural problems for each of the target applications (see Table XII). As we can observe, the results are discrepant between the Health Watcher and the other three systems. However the problem in this analysis is related to the analyzed data. We identified two different groups of architecture roles among the top 10 elements for Health Watcher, ranked as equally relevant. That is, 6 of the related elements were playing the role of repository interfaces. The 4 remaining elements were Facades [10], or elements responsible for communicating different architecture components. We then asked the original architects to elaborate on the relevance of those roles, as we suspected they were unequal. They decided to differentiate the relevance between them, and considered the repository role as less relevant. This refinement led to a completely different ranking, which went up from 4 to 7 elements related to architecture problems.

TABLE XII. ARCHITECTURE ROLE HEURISTIC AND ACTUAL ARCHITECTURAL PROBLEMS

Name	# of ranked CE	Arch-Relevant	% of Arch-Relevant
HW	10	4	40%
MM	10	9	90%
PDP	10	10	100%

The results obtained for Health Watcher show us the importance of correctly identifying the architecture roles and their relevancies for improving the accuracy of this heuristic. When that information is accurate, the results for this heuristic are highly positive. Furthermore, the other proposed prioritization heuristics could benefit from information regarding architecture roles in order to minimize the number of false positives, like utility classes. This indicates the need to further analyze different combinations of prioritization heuristics.

VII. THREATS TO VALIDITY

This section describes some threats to validity observed in our study. The first threat is related to possible errors on the anomalies detection in each of the selected target systems. As the proposed heuristics consist of ranking previously ranked code anomalies, the method for detecting these anomalies must be trustworthy. Although there are several kinds of detection strategies in the state-of-art, many studies have proven that they are inefficient for detecting architecturally relevant code anomalies [19]. In order to reduce the risk of imprecision when detecting code anomalies: (i) the original developers and architects were involved in this process; and (ii) we used well-known metrics and thresholds for constructing our detection strategies [16][31]. The second threat is related to how we identified errors in software systems in order to apply the error-density heuristic. Firstly, we relied on commit messages for identifying classes related to bug fixes. So, it implies that some errors might be missed. In order to mitigate this threat, we also investigated issue-tracking systems. Basically, we looked for error reports and traces between these errors and the code changed to fix them. Furthermore, we investigated test reports in order to identify the causes for eventual broken tests. Finally, for some cases where the information is not available, we relied on the use of static analysis methods for identifying bugs [3].

The third threat is related to the identification of the architectural roles for each of the target systems. The architecture role heuristic is based on identifying the relevance of code elements regarding the system architectural design. Thus, in order to compute the scores for this heuristic, we needed to assess the roles that each code element plays in the system architecture. In this sense, we considered the identification of architectural roles as being a threat to construct validity because the information regarding the architectural roles was extracted differently depending on the target system. Furthermore, we understand that the absence of architecture documentation reflect a common situation that might be inevitable when analyzing real world systems. Finally, the fourth threat to validity is an external threat and it is related to the choice of the target systems. The problem here is that our results are limited to the scope of the 4 target systems. But in order to minimize this threat, we selected systems developed by different programmers, with different domains, programming languages, environment and architectural styles. In order to generalize our results, further empirical investigation is still required. In this sense, our study should be replicated with other applications, from different domains.

VIII. FINAL REMARKS AND FUTURE WORK

The presence of architecturally relevant code anomalies often leads to the decline of the software architecture quality. Furthermore, the removal of those critical anomalies is not properly prioritized, mainly due to the inability of current tools to identify and rank architecturally relevant code anomalies. Moreover, there is no sufficient empirical knowledge towards factors that could make it easier the prioritization process. In this sense, our work has shown that developers can be guided through the prioritization of code anomalies according to architectural relevance. The main contributions of this work are: (i) four prioritization heuristics based on the architecture relevance and (ii) the evaluation of the proposed heuristics on four different software systems.

In addition, during the evaluation of the proposed heuristics, we found that they were mostly useful in scenarios where: (i) there are architectural problems involving groups of classes that change together; (ii) there are architecture problems related to Facades or classes responsible for communicating different modules; (iii) changes are not predominantly perfective; (iv) there are architecture roles infected by multiple anomalies; and (v) the architecture roles are well defined in the software system and have distinct architecture relevance. Finally, in this work we evaluated the proposed heuristics individually. Thus, we have not evaluated how their combinations could benefit the prioritization results. In that sense, as a future work, we aim to investigate whether the combination of two or more heuristics would improve the efficiency of the ranking results. We also intend to apply different weights when combining the heuristics, enriching the possible results and looking for an optimal combination.

REFERENCES

[1] R. Arcoverde, A. Garcia and E. Figueiredo, "Understanding the Longevity of Code Smells – Preliminary Results of an Survey," in Proc. of 4th Int'l Workshop on Refactoring Tools, May 2011

[2] R. Arcoverde *et al*., "Automatically Detecting Architecturally-Relevant Code Anomalies," 3rd Int'l Workshop on Recommendation Systems for Soft. Eng., June 2012.

[3] N. Ayewah *et al*., "Using Static Analysis to Find Bugs," IEEE Software, Vol. 25, Issue 5, pp. 22-29, September 2008.

[4] L. Bass, P. Clements and R. Kazman, " Software Architecture in Practice", Second Edition, Addison-Wesley Professional, 2003.

[5] P. Diaconis and R. Graham, "Spearman's Footrule as a Measure of Disarray", in Journal of the Royal Statistic Society, Series B, Vol. 39, pp. 262-268, 1977.

[6] E. Figueiredo *et al.*, "Evolving Software Product Lines with Aspects: An Empirical Study on Design Stability," in Proc. of 30th Int'l Conf. on Software Engineering, New York, USA 2008.

[7] S. Eick, T. Graves and A. Karr, "Does Code Decay? Assessing the Evidence from Change Management Data", IEEE Transactions on Soft. Eng., Vol. 27, Issue 1, pp. 1-12, 2001

[8] R. Faing, R. Kumar and D. Sivakumar, "Comparing Top K Lists", in Proc. of 14th Annual ACM-SIAM Symposium on Discrete Algorithms, Society for Industrial and Applied Mathematics, pp. 28-36,, USA 2003.

[9] M. Fowler, "Refactoring: Improving the Design of Existing Code," Addison-Wesley, 99.

[10] E. Gamma *et al.*, "Design Patterns: Elements of Reusable Object-Oriented Software", Addison-Wesley, Boston, USA, 1995.

[11] J. Garcia, D. Popescu, G. Edwards and N. Medvidovic, "Idenfitying Architectural Bad Smells," in Proc. of CSMR, Washington, USA 2009.

[12] M. Godfrey and E. Lee, "Secrets from the Monster: Extracting Mozilla's Software Architecture", in Proc. of 2nd Int'l Symp. On Constructing Software Engineering Tools, 2000.

[13] P. Greenwood *et al.*, "On the Impact of Aspectual Decompositions on Design Stability: An Empirical Study," in Proc. of 21st Conf. of Object-Oriented Programming, Springer, pp. 176-200, 2007.

[14] L. Hochestein and M. Lindvall, "Combating Architectural Degeneration: A Survey," Information of Software Technology, Vol. 47, July 2005.

[15] D. Kelly, "A Study of Design Characteristics in Evolving Software Using Stability as a Criterion," IEEE Transactions on Software Engineering, Vol. 32, Issue 5, pp. 315-329, 2006.

[16] F. Khom, M. Penta and Y. Guéhenéuc, "An Exploratory Study of the Impact of Code Smells on Software Change-Proneness," in Proc. of 16th Working Conf. on Reverse Eng., pp. 75-84, 2009

[17] M. Kim, D. Cai and S. Kim, "An Empirical Investigation into the Role of API-Level Refactorings during Software Evolution," in Proc. of 33rd Int'l Conf. on Software Engineering, USA 2011.

[18] M. Lanza and R. Marinescu, "Object-Oriented Metrics in Practice," Springer-Verlag, New York, USA 2006

[19] I. Macia *et al*., "Are Automatically-Detected Code Anomalies Relevant to Architectural Modularity? An Exploratory Analysis of Evolving Systems," in Proc. of 11th AOSD, pp. 167-178, Germany, 2012.

[20] I. Macia *et al*., "On the Relevance of Code Anomalies for Identifying Architecture Degradation Symptoms", in Proc. of 16th CSMR, Hungary, March 2012.

[21] I. Macia *et al*., "Supporting the Identification of Architecturally-Relevant Code Anomalies", in Proc. of 28th IEEE Int'l Conf on Soft. Maint., Italy, 2012.

[22] I. Macia *et al*., "Enhancing the Detection of Code Anomalies with Architecture-Sensitive Strategies", in Proc. of the 17th CSMR, Italy, March 2013.

[23] A. MacCormack, J. Rusnak and C. Baldwin, "Exploring the Structure of Complex Software Design: An Empirical Study of Open Source and Proprietary Code", in Management Science, Vol. 52, Issue 7, pp. 1015-1030, 2006.

[24] S. Malek *et al*., "Reconceptualizing a Family of Heterogeneous Embedded Systems via Explicit Architectural Support", in Proc. of the 29th Int'l Conf on Soft. Eng., IEEE Computing Society, USA 2007.

[25] M. Mantyla and C. Lassensius, "Subjective Evaluation of Software Evolvability using Code Smells: An Empirical Study, Vol. 11, pp. 395-431, 2006

[26] R. Marinescu, "Detection Strategies: Metrics-Based Rules for Detecting Design Flaws," in Proc. Int'l Conf. on Soft. Maint., pp. 350-359, 2004.

[27] R. Martin, "Agile,Software Software Development, Principles, Patterns ans Practices. Prentice Hall, 2002.

[28] M. J. Munro, "Product Metrics for Automatic Identification of Bad Smells Design problems in Java Source-Code", In Proc. of 11th Int'l Sympositum on Soft. Metrics, pp. 15, September 2005.

[29] E. Murphy-hill, C. Parnin and A. Black, " How We Refactor and How We Know it," in Proc. of 31st Int'l Conf. on Software Engineering, 2009.

[30] NDepend. Available at http://www.ndepend.com. 2013.

[31] S. Olbrich, D. Cruzes and D. Sjoberg, "Are Code Smells Harmful? A Study of God Class and Brain Class in the Evolution of Three Open Source Systems," in Proc. of 26th Int'l Conf. on Soft. Maint., 2010.

[32] J. Ratzinger, M. Fischer and H. Gall, "Improving Evolvability through Refactoring," in Proc. of 2nd Int'l Workshop on Mining Soft. Repositories, ACM Press, pp. 69-73, New York, 2005.

[33] Understand, 2013. Available at: http://www.scitools.com/

[34] C. Wohlin, *et al*., "Experimentation in Software Engineering – An Introduction", Kluwer Academic Publisher, 2000.

[35] S. Wong, Y. Cai and M. Dalton, "Detecting Design Defects Caused by Design Rule Violations," in Proc. of 18th ESEC/ Foundations on Software Engineering, 2010.

[36] Z. Xing and E. Stroulia, "Refactoring Practice: How it is and How it should be Supported: An Eclipse Study," in Proc. of 22nd IEEE Int'l Conf. on Software Maintenance, pp. 458-468, 2000.

[37] D. Sheskin, "Handbook of Parametric and Nonparametric Statistical Procedures", Chapman & All, 4th Edition, 2007.