4
Evaluation

Chapter 3 detailed our heuristics for supporting the prioritization of code anomalies according to their relevance to a system architecture. Those heuristics were based on the assumption that by analyzing specific characteristics of a software project, it is possible to identify and rank architecturally relevant code anomalies. Such ranking could then help developers prioritize their refactorings while aiming to solve deeper maintainability problems, i.e. architecture design problems.

In order to verify whether those heuristics were indeed useful for ranking code anomalies, we conducted an empirical study, guided by the following questions:

1) Is it possible to accurately rank code anomalies based on their architecture relevance?
2) Which characteristics help accurately ranking code anomalies based on their architecture relevance?

Given that this is a first study in the field of ranking code anomalies, our evaluation is of exploratory nature (KITCHENHAM *et al.*, 2002) and focused on a detailed analysis of four software projects. In order to make the purpose of our evaluation clearer, we have defined a set of hypotheses (Section 4.2.1). However, it was not our goal to rely on a large data set and carry out statistical tests over this set. In other words, our aim here was to perform a first identification and evaluation of project factors that are likely to be useful to rank architecturally relevant code anomalies. Our evaluation is intended to derive some lessons learned related to the usefulness of these project factors. These lessons will provide insights for the conception of more specific hypotheses that need to be rigorously tested in further studies.

This chapter presents the evaluation of the prioritization heuristics. Section 4.1 describes the selection criteria and the characteristics of four target

applications we used for evaluating the prioritization heuristics. Section 4.2 describes our study setting, including our hypotheses, procedures and analysis methods. We discuss our results for each prioritization heuristic in Section 4.3. Finally, we present the threats to our study validity in Section 4.4.

4.1. Selection Criteria and Target Applications

In order to be feasible for the evaluation of the prioritization heuristics, the selected target applications had to adhere to a series of characteristics.

First, either the architecture specification or the original developers and architects had to be available. By analyzing such specification, or consulting the developers and architects, we are able to verify whether the ranked code anomalies are indeed architecturally relevant. It also helps us to better analyze the architecture roles played by each code element. Such information is essential to the application of the *architecture role heuristic* (Section 3.1.4), which directly depends on architecture information to compute the ranking of code anomalies.

Second, the source version control systems of the selected applications should be available, in order to enable the application of the *change-proneness heuristic* (Section 3.1.1). In fact, the selected systems should also ideally present a high number of changes (or revisions) through their evolutions.

Third, an available issue tracking system, although not mandatory, was highly recommendable for providing input to the *error-proneness heuristic*. As explained on Section 3.4.1.2, however, such heuristic might also use information from commit messages for inferring bug fixes revisions. However, in this case, the source version control system should not only be available, as required by our second criterion, but also contain templates for well-formatted commit messages. Such templates are essential for the retrieval of information regarding bug fixes and their corresponding code elements.

Finally, the applications should present different design and implementation structures. Such restriction helps us to better understand the impact of the proposed heuristics for diverse sets of code anomalies, emerging on different architecture designs. Moreover, it minimizes the possibility of bias for the results of a particular architecture design.

Based on these criteria, we selected the last versions of 4 software projects from different application domains: HealthWatcher (GREENWOOD *et al.*, 2007), MobileMedia (FIGUEIREDO *et al.*, 2008), MIDAS (MALEK *et al.*, 2007) and PDP – an acronym for an industry application, privately held by an entertainment company. The table below summarizes the main characteristics of the selected software projects.

Table 1 – Characteristics of target software projects

	MIDAS	**HW**	**MM**	**PDP**
Application Type	Middleware	Web application	Software Product Line	Web application
Programming Language	C++	Java/AspectJ	Java/AspectJ	C#
Architectural Design	Layers	MVC	Layers	PC & Layers
# of CE	21	137	82	97
# of AE	9	19	24	15
KLOC	72	46	51	22
# of CA	178	273	176	175
# of AP	29	112	90	28
# of Revisions	1	10	10	409

HW – HealthWatcher, MM – MobileMedia, CE – code elements, AE – architecture elements (modules and interfaces), CA – code anomalies, PC – Page Controller, AP – Architecture Problems

HealthWatcher (HW) is a web software system used for registering complaints about health issues in public institutions. Mobile Media (MM) is a software product line that manages different types of media on mobile devices. MIDAS is a lightweight middleware for distributed sensor applications (GARCIA *et al.*, 2009). PDP is a web application for managing scenographic sets in television productions. These projects were previously analyzed in studies of architectural degradation and refactoring (GARCIA *et al.*, 2009; DANTAS *et al.*, 2011; MACIA *et al.*, 2012b). Therefore, part of the needed information for performing our evaluation had already been previously collected, without introducing bias with respect to our specific research questions. More specifically, the code anomalies for all of these systems had already been collected in a previous study (MACIA *et al.*, 2012b), in the context of this dissertation. The

detected code anomalies were also classified regarding their architectural relevance in the aforementioned study.

It is important to mention the difficulty in finding software projects that adhered to all of the defined selection criteria. In fact, such difficulty led us to choose projects that only contained a subset of the characteristics we needed for evaluating all of the prioritization heuristics. For example, the MIDAS project was not suitable for the *change-proneness heuristic*, as only one revision of it was available. However, as we had already studied its architecture problems and detected its code anomalies for previous studies, it was reasonable to consider it when analyzing the *architecture role* (Section 3.1.4) and *anomalies density* (Section 3.1.3) heuristics.

4.2. Study Setting

Our study aims at analyzing whether the proposed set of heuristics might help developers to identify architecturally relevant code anomalies that should be prioritized. It is expected that the use of the proposed heuristics will help developers to early detect architecture problems reified in the implementation, so that architecture degradation can be avoided. Our analysis is carried out in terms of the *accuracy* (Section 4.2.1) of the prioritization heuristics towards ranking code anomalies correctly. Following the recommendation from Wohlin et al (2000), we defined our study and its goals using the GQM format, as follows:

> **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:** the researcher
>
> **In the context of**: 4 software systems from different domains and with different architectural and implementation detailed designs

The study was conducted in three phases: first, as a prioritization approach for ranking code anomalies, we needed to detect and classify those anomalies regarding their architecture relevance, in each of the target systems (Section 4.2.3.1). Second, we computed the scores for each detected code anomaly, according to the heuristic under analysis, producing a resulting ordered list (Section 4.2.3.4). In the third phase, we compared the heuristics results with rankings previously defined by developers or maintainers of each analyzed system, calculating their similarities (Section 4.2.4). Those rankings provided by developers represent the "ground truth" data in our analysis.

4.2.1. Hypotheses

In order to evaluate the accuracy of the proposed heuristics for ranking code anomalies based on their architecture relevance, we first established thresholds of acceptable accuracy. Such thresholds, in this study, were defined in three different levels:

0%-40% - low accuracy

40%-80% - acceptable accuracy

80%-100% - high accuracy

We chose to analyze those three levels of ranking accuracy for analyzing to what extent the prioritization heuristics might be helpful – qualifying the results in three possible ranges. For example, an accuracy level of 50% means the rankings produced by the proposed heuristics should be able to identify at least half of the most architecturally relevant code anomalies outlined by the developers, in the right order of priority. We performed our analysis over top ten rankings, as explained on Section 4.2.4.

In this context, we defined the following null hypotheses:

$H1_0$. The change-proneness heuristic cannot accurately identify architecturally relevant code anomalies ranked as top ten

$H2_0$. The error-proneness heuristic cannot accurately identify architecturally relevant code anomalies ranked as top ten

$H3_0$. The anomaly density heuristic cannot accurately identify architecturally relevant code anomalies ranked as top ten

$H4_0$. The architecture role heuristic cannot accurately identify architecturally relevant code anomalies ranked as top ten

We also defined the following alternative hypotheses:

$H1_A$. The change-proneness heuristic can accurately identify architecturally relevant code anomalies ranked as top ten

$H2_A$. The error-proneness heuristic can accurately identify architecturally relevant code anomalies ranked as top ten

$H3_A$. The anomaly density heuristic can accurately identify architecturally relevant code anomalies ranked as top ten

$H4_A$. The architecture role heuristic can accurately identify architecturally relevant code anomalies ranked as top ten

In the context of the hypotheses defined above, *accurately* means that the heuristic was able to identify architecturally relevant code anomalies with, at least, acceptable accuracy. Therefore, the accuracy level should reach 40% or more.

4.2.2. Variable Selection

In order to test our hypotheses, we have defined the following dependent and independent variables.

Independent variables. There are five independent variables in this study: the rankings produced by each of the prioritization heuristics and the ranking that represents the ground truth. Those rankings are lists of code anomalies, ordered by their respective scores. The ranking representing the ground truth was produced by the original architects of each system.

Dependent variables. There are as many dependent variables as there are similarity measures between the rankings produced by the proposed prioritization heuristics and the rankings representing the ground truth. We selected three different similarity measures: number of overlaps, Spearman's Footrule (DIACONIS *et al.*, 1977) and Fagin's extension to Speaman's Footrule (FAGIN *et al.*, 2003). Those measures are defined in Section 4.2.4.

4.2.3. Data Collection

The data collection process of our study involved different activities, including: detecting code anomalies, identifying the rankings representing the ground truth and collecting the scores for each anomaly under the perspectives of each prioritization heuristic. These activities are described next.

4.2.3.1. Detecting Code Anomalies

As a first step, code anomalies were automatically identified using well known detection strategies and thresholds (KHOM *et al.*, 2009; OLBRICH *et al.*, 2010). These strategies and thresholds were also similarly applied in other studies based on the same systems we analyzed (FIGUEIREDO *et al.*, 2008; MACIA *et al.*, 2012a, 2012b). The analysis and validation of such detection strategies are out of the scope of this dissertation; nevertheless, they are detailed and discussed by previous works (MACIA *et al.*, 2012a). The metrics required by detection strategies were mostly collected with current tools such as: Together (TOGETHER, 2012), NDepend (NDEPEND, 2012) and Understand (UNDERSTAND, 2012). These tools are complementary: Together analyzes Java programs, while NDepend and Understand analyze C++ and C# programs.

As a second step, the list of code anomaly suspects was checked and refined by the developers and architects. This validation was an important step and motivated by the fact that strategies presented low accuracy rates when detecting architecturally relevant code anomalies (MACIA *et al.*, 2012a). By mixing automatic with manual detection, we aimed at analyzing a more reliable set of code anomalies.

4.2.3.2. Rankings Representing the Ground Truth

The *ground truth ranking,* in the context of this dissertation, is a list of anomalous elements, ordered by their architecture relevance. For this analysis, we chose to consider only classes as the analyzed elements. Such ranking was defined by the architects of the target systems. We analyzed the accuracy of our

prioritization heuristics by comparing the similarity between the rankings each heuristic produced with the ground truth ranking.

For collecting the rankings representing the ground truth, we contacted the architects of the analyzed systems. They were asked to provide an ordered list of the top 10 classes they believed better represented the main sources of maintainability problems from the perspective of the architecture of those systems. We did not provide any specific criteria for guiding them to choose the most critical classes. In particular, we did not mention any of the factors explored by the prioritization heuristics (Section 3.1), such as number of changes or faults per module. Our intention was not influence the results at all towards a better performance of any of the heuristics.

In some cases, like for the HealthWatcher system, architects claimed it was not possible to fit only 10 classes into the ranking, as many of them were considered equally problematic. For this particular case, the ranking representing the ground truth had 14 elements, and our analyses were performed on 14-sized rankings.

Besides informing the lists of high priority elements, we also asked developers and architects to provide information regarding the architecture design of the analyzed systems. More specifically, we asked for a list of architecture roles that were present in each analyzed system, as well as their order of relevance from the architecture perspective. They also provided traces between architecture elements and code elements, which were essential for computing the *architecture role heuristic*. It is worth mentioning that this data was only requested after the architects had already provided their ground truth rankings, in order to avoid the influence of the leveraged architecture information against the rankings.

4.2.3.3. Calculating Scores

The second phase of our study involved the calculation of scores for each code anomaly, in the context of the proposed heuristics. As the mechanics of each heuristic are fairly different (Section 3.2.1), we detail below the steps followed for each of them separately.

Change-proneness heuristic. As explained in Section 3.2.1, the change-proneness heuristic computes its scores based on the number of changes made to a given code element. In the context of this study, we computed file changes only, i.e, different code elements modified within a single file are not distinguished.

In order to correctly compute the scores for this heuristic, we first had to extract the change log from the version control systems for each of the target applications. We then processed the resulting log files, counting the number of times each resource was changed. This extraction was first performed in the root directory for each system, retrieving all the file changes in a single command. Once the number of changes was computed, we ordered the list of resources and their respective number of changes, thus producing our final ranking.

Error-proneness heuristic. For computing the error-proneness heuristic scores, we used two different techniques: the first one is based on change log inspection, looking for common terms like *bug* or *fix.* Once those terms are found on commit messages, we increment the scores for the classes involved in that change. Such technique has been recently applied on many relevant studies (FISCHER *et al.*, 2003; KIM *et al.*, 2011). However, as we could not rely on it for all of the analyzed systems – as only the PDP system had high quality and well-formatted commit messages – we also used a bug detection tool on MobileMedia and HealthWatcher. As both systems were implemented in Java, we chose the *findBugs* tool (AYEWAH *et al.*, 2008) for automatically detecting blocks of code that could be related to bugs. The *fingBugs* tool uses static analysis to detect potential bugs, such as security violations (SQL injection), runtime errors (dereferencing a null pointer) and logical inconsistencies (a conditional test that can't possibly be true). Once those possible bugs are identified, we collect the code elements causing them and increment their scores.

Anomaly density heuristic. Computing the scores for this heuristic was rather straightforward. In this phase, we verified which code elements concentrated the highest number of code anomalies. Those anomalies had already been detected in the first phase of our study, as described on Section 4.2.3.1.

Architecture role heuristic. The architecture role heuristic depends on two kinds of information, regarding the system's design: first, which roles each class plays in the architecture and second, how relevant those roles are towards architecture maintainability. For example, consider an application following the

MVC pattern (BUSCHMANN *et al.*, 2007). Code anomalies found on classes that implement Views might not be considered as relevant as those infecting classes that implement the Model, as Views are not supposed to implement complex behaviors or business rules.

For this study setting, when analyzing this heuristic, we first had to leverage architecture design information in order to map code elements to their architecture roles, as explained in Section 4.2.3.3. Part of this information extraction had already been performed in our previous studies (MACIA *et al.*, 2012a, MACIA *et al.*, 2012b). Then, we asked the architects to assign different levels of importance to those roles, according to the architecture patterns implemented (e.g., MVC, 3-layers, Page Controller).

Finally, 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 – namely, 10. 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 of existing roles and the classification provided by the architects. For example, in PDP, which implemented the Page Controller pattern, the architects identified 4 different architecture roles. Hence, the scores for each architecture role could assume the values 0, 2.5, 5, 7.5 or 10 (as 10/4 is 2.5, which we used as the default interval). We assigned a 10-value score to anomalies found on communication classes or interfaces; business classes received a 7.5 scores; and finally, anomalies found on data classes were given a 2.5 score.

4.2.4. Analysis Method for Comparing Rankings

In the third and last phase of our study, we compared the rankings produced by each of the heuristics with those representing the ground truth – provided by the developers or maintainers of the analyzed systems. We decided to analyze only the top ten code anomalies ranked, for three main reasons: first, asking developers to rank an extensive list of anomalies would be unviable and counterproductive. Second, we wanted to evaluate our prioritization heuristics

mainly for their abilities to improve refactoring effectiveness – that is, improving the chances developers will prioritize the removal of relevant code anomalies. In this context, the top ten code anomalies represent a significant sample of anomalies that could possibly cause architecture problems. Third, we focused on analyzing the top 10 anomalies for assessing whether they already represent a useful subset of architecturally relevant anomalies. Otherwise, the need to look beyond those top 10 anomalies to find something useful could discourage developers when using our prioritization heuristics.

For comparing such rankings, we considered three different measures: the size of the overlap, Spearman's footrule (DIACONIS *et al.*, 1977) and Fagin's extension to the Spearman's footrule for disjoint lists (FAGIN *et al.*, 2003). The first and simplest one was the size of the overlap between the two top ten lists. We chose this measure mainly because (i) it is fairly simple and (ii) it tells us whether the prioritization heuristics are accurately distinguishing the top *k* items from the others. This measure has some disadvantages, as it does not consider permutations between two lists. For example, lists containing the same elements in different orders (or with different ranks) would have the same similarity measure, becoming indistinguishable from lists that present a perfect match.

The second measure we considered was Spearman's footrule (DIACONIS *et al.*, 1977), a well-known metric for permutations – or full rankings. It measures the distance between two ranked lists by computing the differences in the rankings of each item. That is, given two lists *a* and *b*, Spearman's footrule is defined as $F(a, b) = \Sigma\ i \in U\ |a(i) - b(i)|$. Therefore, when the lists are identical, Spearman's result is 0. The maximum value of $F(a,b)$ is $n^2/2$ when n is even and $(n + 1)\ (n - 1) / 2$ when n is odd – where n represents the size of the lists. We chose this measure mainly because it is a classic, well-known metric on permutations (or full rankings), along with Kendall tau (KENDALL and GIBBONS, 1990) – a metric based on the number of permutations needed to turn one list into the other. We did not measure the Kendall tau distance for our study, as it belongs to the same equivalence class as Spearman's footrule (FAGIN *et al.*, 2003).

A clear limitation of Spearman's footrule is the fact that it cannot be applied to disjoint rankings; however, in our context, the top 10 code anomalies identified by developers and the results of our heuristics are not necessarily identical. Thus, in order to use Spearman's footrule for measuring similarity, we needed to

transform the obtained lists, eliminating items that did not overlap and re-ranking the remaining items. This approach has been previously documented by (BAR-ILLAN, 2006) when comparing search engine results.

Finally, the third measure used in this work was proposed by (FAGIN, 2003) as an extension to Spearman's footrule for top *k* lists. A top *k* list is a list *r* with $|r| = k$, and all other items $i \notin r$ are assumed to be ranked below every item in *r*. Although the rankings produced by our prioritization heuristics are *complete rankings* – in the sense that all elements are associated with a score – we are only comparing the top ten items. Therefore, the use of measures for top *k* lists is appropriate, especially considering that such lists might contain different elements.

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. For example, when comparing lists of size *k*, this placement can be *k+1* for all missing items in each list. This extension rationale is that items appearing only in one of the lists must have *k+1* ranks, or higher. Fagin's extended metric is defined below:

$$F(a, b)^k = 2(k - |Z|)(k + 1) + \Sigma_{i \epsilon z} \mid a(\mathrm{i}) - b(\mathrm{i}) \mid - \Sigma_{i \epsilon s}\, a(\mathrm{i}) - \Sigma_{i \epsilon t}\, b(\mathrm{i})$$

In this equation, *Z* is the set of overlapping items, *S* is the set of items that belong to *a* but not to *b* and *T* is the set of items that belong to *b* but not to *a*.

In order to compare the results obtained from Spearman's footrule and Fagin's equation, we conveniently used normalized versions of both measures, by dividing each measure by its maximum value. The normalized versions lie in the interval [0,1] – where 0 means the rankings were identical and 1 means they were reversed. For Fagin's extension, considering $k = 10$ (as we are comparing *top 10* rankings), the normalization factor is the maximum value $F(a,b)^{10}$ can assume. This situation happens when the overlap between the lists is 0 (i.e., when the lists have no elements in common). Thus, *Z* is an empty set, *S* is equivalent to *a* and *T* is equivalent to *b*. Therefore:

$$F(S,T)^{10} = 2\,(10 - |Z|)(10 + 1) + 0 - \Sigma_{\mathrm{i}\,=\,1,\,10}\,\mathrm{i} - \Sigma_{\mathrm{i}\,=\,1,\,10}\,\mathrm{i}$$

$$F(S,T)^{10} = 2\,(10 - 0)(11) - 55 - 55$$

$$F(S,T)^{10} = 220 - 110$$

$F(S,T)^{10} = 110$

Using 110 as the normalization factor for $k = 10$, we can calculate the normalized value for Fagin's extension as:

$Fn(a,b) = F(a,b) / 110$

Such normalizations were discussed and proven equivalent on (FAGIN, 2003).

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. Clearly, a high number of overlaps for top 10 items in a list of 1000 items is much harder to obtain than in a list of 100 items. Therefore, 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 two remaining measures' purpose is to analyze the similarity between two rankings. Therefore, 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 - respecting the order among the overlapping elements. 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. We obtained the accuracy percentage for each measure as described below:

Table 2 – Calculating similarity accuracy level

	Overlap	**NSF**	**NF**
Accuracy %	k * 10	(1 – n) * 100	(1 – n) * 100

NSF – normalized Spearman's footrule measure, NF – normalized Fagin's extension to Spearman's footrule measure

Assuming that there are always at most 10 overlaps, the number of overlaps measure will present 100% of accuracy when k = 10. In this context, *k* represents the number of overlaps between the rankings.

For the NSF and NF measures, the rankings are perfect matches when they evaluate to 0; therefore, (1 – 0) * 100 = 100%. In this context, *n* represents the value obtained for NSF and NF, as previously defined by their equations. Furthermore, a high number of overlaps increases the precision of both Spearman's footrule and Fagin's measures. When that number is smaller than 5, for example, Spearman's footrule measure can only assume at most 5 different values – e.g., for 4 overlaps, 0, 2,4,6 and 8 (i.e., 0, 25%, 50%, 75% or 100% of accuracy). A small number of overlaps also affects Fagin's measure – although not as severely: as Fagin's measure assigns a default value (*k+1*) to all missing elements, the number of possible results is not affected. However, missing elements belonging to top positions will have a greater impact in the final measure – as the distance between their original ranks and the default value of (*k+1*) will be large.

4.2.5. Code Anomaly Rankings and Actual Architecture Problems

We used the ground truth ranking provided by architects to analyze the accuracy of our prioritization heuristics. However, that analysis only takes into consideration the point of view of the architects, regarding the most architecturally relevant anomalies. Although that point of view provides interesting insights on how architecture problems are perceived, we also wanted to investigate the actual architecture problems that occurred through the systems' history. For doing so, we compared the results of the prioritization heuristics rankings to previously identified architecture problems. In this context, we investigated the proportion of architecturally relevant code anomalies among the top ten rankings produced by each heuristic.

This analysis was motivated by our previous studies (MACIA *et al.*, 2012b), where we investigated the correlations between code anomalies and architecture problems. In this context, we wanted to understand whether – and to what extent - our prioritization heuristics' top results comprehend actual architecturally relevant

code anomalies. This analysis may provide insights into which characteristics, or combined characteristics, explored by our heuristics are more helpful towards finding architecture problems. Moreover, it adds another source of information for evaluating our results, in addition to the rankings provided by developers.

For performing this investigation, we relied on the lists of architecture problems detected in the aforementioned study. For each problem reported in those lists, we identified the code anomalies that caused it, if pertinent, and the code elements related to it. Next, we were able to look for those elements in the rankings produced by our prioritization heuristics. If a ranked code element was related to at least one architecture problem, we considered it architecturally relevant. Table 3 summarizes the information regarding architecture problems and architecturally relevant elements for each analyzed system.

Table 3 – Architecturally relevant code elements

	# of CE	# of AP	# of architecturally relevant CE
HW	137	112	39 (28%)
MM	82	90	30 (36%)
PDP	97	28	37 (38%)
MIDAS	21	29	6 (28%)

CE – code elements, AP – architecture problems

We can observe from Table 3 that part of the existing code elements are responsible for every architecture problem found. Thus, in all systems, there are code elements that were not related to any architecture problem. Our goal is to analyze whether the prioritization heuristics are able to outline code elements that are in fact related to at least one architecture problem.

4.3. Heuristics Evaluation

This section presents the results of our heuristics evaluation. We evaluated each heuristic separately, as they exploit different project characteristics in order to determine the architecture relevance of anomalous elements in the implementation. Such evaluation was performed in two separate phases: first, we conducted a quantitative analysis on the similarity results; then, we perform a qualitative evaluation of the results, regarding their relations to actual architecture problems (Section 4.2.5), discussing them in detail.

4.3.1. Evaluation of the Change-Proneness Heuristic

The Change-Proneness Heuristic was applied to three out of the four target systems. We did not analyze MIDAS under this heuristic as only one version of this application was available. Our evaluation was based on the analysis of 10 different versions of HealthWatcher, 8 versions of MobileMedia and 409 versions of PDP. We have selected software projects with different history sizes on purpose. We wanted to check whether the heuristic performed well or not on systems with shorter and longer longevity. In addition, it was not a requirement to only embrace projects with long histories, as we wanted to better analyze whether the heuristics would be effective in preliminary versions of a software system, when there is more opportunity for architecture-wide refactorings.

The evolution characteristics of the analyzed systems are summarized on Table 4. Those characteristics comprehend the total number of code elements per system, the maximum number of revisions per file and the average number of revisions per file. As PDP is the system with the highest number of revisions, its files were also changed the most, with up to 74 changes in a single file – namely, `Inicial.aspx.cs`. The second most changed file was also part of PDP, having 40 different versions. The reason for such discrepancy relies on the fact that the most changed file acted as a controller Façade, concentrating all the request handling methods.

Table 4 – Change characteristics for each system

	# of revisions	# of CE	max revisions	avg revisions
HW	10	137	9	1,5
MM	9	82	8	2,6
PDP	409	97	74	8,8

CE – code elements, avg – average

As we can see, HealthWatcher (HW) and MobileMedia (MM) had similar evolution behaviors. As the maximum number of revisions for a single file is limited to the total number of revisions for a system, neither HealthWatcher nor MobileMedia could have 10 or more versions of a code element. Although HealthWatcher had more revisions than MobileMedia, those changes were

scattered between more files. Therefore, changes to MobileMedia were concentrated in a smaller number of code elements, resulting in a higher average number of changes per file.

Because of the reduced number of revisions available for HealthWatcher and MobileMedia, we had to establish a criterion for selecting items when there were ties in our top 10 rankings. For example, Table 5 illustrates the change-proneness ranking for MobileMedia. In this ranking, we used alphabetical order for breaking ties. That approach was only possible because there were 9 elements in the ground truth ranked as equally harmful. Therefore, we sorted the elements alphabetically to avoid inconsistencies when calculating the scores, as we also sorted alphabetically the elements on the rankings produced by our heuristics. Another approach would be to rank files with the same number of changes so that they co-occupied the same ranking position.

Table 5 – Top 10 change-proneness ranking for MM

Rank	File	# of changes
1	ubc.midp.mobilephoto.core.ui.datamodel.AlbumData	8
2	ubc.midp.mobilephoto.core.ui.MainUIMidlet	8
3	ubc.midp.mobilephoto.core.ui.screens.AlbumListScreen	8
4	ubc.midp.mobilephoto.core.ui.controller.BaseController	7
5	ubc.midp.mobilephoto.core.ui.screens.PhotoViewScreen	6
6	ubc.midp.mobilephoto.core.ui.controller.PhotoViewController	5
7	ubc.midp.mobilephoto.core.ui.datamodel.ImageAccessor	5
8	ubc.midp.mobilephoto.core.util.Constants	5
9	ubc.midp.mobilephoto.core.util.ImageUtil	5
10	ubc.midp.mobilephoto.sms.SmsSenderThread	4

Finally, Table 6 shows the results for each measure when analyzing the change-proneness heuristic.

Table 6 – Results for the change-proneness heuristic

	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%

NSF – normalized Spearman's footrule measure, NF – normalized Fagin's extension to Spearman's footrule measure

We can see that for the change-proneness heuristic, the highest absolute overlap was obtained for HealthWatcher – with 8 overlapping items. The main reason why this happened is that HealthWatcher had many files with the same number of changes. Therefore, for computing the overlap measure, we did not consider only the 10 most changed files, as that approach would discard files with as many changes as the ones selected. Instead, we selected 14 files, where the last 5 had exactly the same number of changes. It is also important to notice that HealthWatcher was the system with the highest number of code elements (classes, interfaces or abstract classes) – having a total of 137 items that could appear on our rankings.

However, although the number of overlapping items for HealthWatcher is high, the similarity measures for this system show their rankings were quite mismatched. The NSF measure of 0,75 was obtained because, from the 7 overlapping items, only 2 were ranked in the same position (#2 and #5). The NF measure was also high due to the amount of non-overlapping items (7) and the fact that their ranks were relatively high (i.e., were positioned in the top of the ranking - for example, items #2, #4 and #5).

Another interesting finding was regarding the MobileMedia system: although this heuristic identified 5 overlaps between the generated ranking and the ground truth, all of them were shifted by exactly two positions – resulting in the 1 value for the NSF measure. However, when we considered the non-overlaps, the position for one item matched - resulting in a slightly smaller distance measure for NF. Moreover, this result shows us that the NSF measure is not adequate when the number of overlaps is small: there were similarities between the compared rankings that could be inferred from the resulting value. For example, the ranking order for elements #1, #2 and #3 was preserved – as they appeared in the #3, #4 and #5 positions, respectively.

When comparing the results of MobileMedia and HealthWatcher to those obtained by PDP, we realize there is a significant difference between them: all of PDP measures performed above our acceptable similarity thresholds (> 45% similarity). In this case, we found that the similarity was related to a set of classes that were deeply coupled: an interface acting as a Façade (PDPServices) and three realizations of this interface, implementing a client module, a Proxy and the server module. As changes in the interface triggered changes in those three classes, they

suffered many modifications through the system's evolution. Moreover, such design was considered a serious architecture problem in this system.

Furthermore, the nature of the changes that those systems underwent is fairly different: most changes on HealthWatcher were perfective, intending to improve the overall system quality, without adding new features. Therefore, classes that were top ranked by the change-proneness heuristic were probably refactored many times throughout the system evolution. As those classes were repeatedly refactored, they no longer represent threats to the systems' architecture, which explains why they did not appear on the ground truth ranking. Most changes performed on MobileMedia, on the other hand, were related to the addition of new functionalities, which was also the case for PDP. However, MobileMedia also had low accuracy rates. Therefore, the different results for this heuristic might also be associated with the differences between the evolution histories of the analyzed systems: while MobileMedia and HealthWatcher had around 10 analyzed revisions, PDP had 409.

In conclusion, the results for this heuristic show us that it would be probably useful for detecting and ranking architecturally relevant anomalies when: (i) there are architecture problems involving groups of classes that change together; (ii) there are architecture problems related to Façades or communication classes, as changes to those classes might trigger changes in client components; (iii) changes were not predominantly perfective, i.e., the majority of the changes performed on the system were not refactorings.

The results we obtained in this analysis also helped us to reject the null hypothesis $H1_0$ – as the Change-Proneness Heuristic was able to produce rankings for PDP with at least acceptable accuracy in all of the analyzed measures (60%, 56% and 46%).

Correlation with Actual Architecture Problems

When analyzing the correlation of the *change-proneness heuristic* rankings with actual architecture problems for each system, we also had interesting results, confirming the usefulness of the prioritization. Such analysis was performed by observing which of the ranked elements were related to actual architecture problems. Table 7 summarizes our results.

Table 7 – Change-proneness and actual architecture problems

	# of ranked CE	architecturally relevant	% of architecturally relevant
HW	14	10	71%
MM	10	7	70%
PDP	10	10	100%

CE – code elements

It can be observed on Table 7 that elements containing architecturally relevant anomalies were very likely to be change-prone. In fact, for the PDP system, all of the top 10 most changed elements were related to architecture problems. Considering that PDP has 97 code elements, and 37 of them are related to architecture problems, this can give us a hint that change-proneness is a good heuristic for identifying them.

4.3.2. Evaluation of the Error-Proneness Heuristic

The *error-proneness heuristic* is based on the assessment of bugs that were introduced by a given code element. The higher the number of bugs found on that element, the higher its priority. Therefore, in order to correctly evaluate the results for this heuristic, a reliable set of detected bugs should be available. This was the case for one of the analyzed systems – namely, the PDP system. However, for the remaining target systems, there was no such documentation of detected bugs and the code elements that caused them. As explained in Section 4.2.3, for such case, we relied on the analysis of bug detection tools, which indicate code elements that could possibly introduce bugs.

The results for the *error-proneness heuristic* are summarized on Table 8.

Table 8 – Results for the Error-Proneness Heuristic

	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	50%	0,83	17%	0,74	26%

It is important to mention that for the HealthWatcher system, exceptionally, there were 14 ranked items, instead of 10, due to ties between some of them.

Nonetheless, HealthWatcher presented the highest overlap – achieving an accuracy of 71%. The reason why that happened is that the detected bugs were related to a behavior found on every class implementing the Command role. Furthermore, every Command class was listed by the developers as high-priority in the ground truth ranking – meaning they represented a top rated architecture problem. Therefore, the overlaps between the rankings were all related to classes that implemented commands on HealthWatcher.

Besides the high number of overlaps, we also observed that the priority order for the overlapping elements was exactly the same as the one pointed out in the ground truth – hence, the accuracy obtained by NSF was 100%. However, the 4 remaining non-overlapping elements were exactly the top 4 elements in the ground truth ranking. The fact that the top 4 elements did not appear in the ranking produced by the *error-proneness heuristic* resulted in a low accuracy for the NF measure – 26%.

We applied the same strategy for bugs mining in MobileMedia. However, for this system, all of the measures presented low accuracies. In fact, because of the small number of overlaps, the results for NSF may not confidently represent the heuristics' accuracy, as explained in Section 4.2.4.

For PDP, the results may be evaluated from a different perspective, as we considered manually detected bugs, reported on its issue tracking system, instead of automatically detected ones. However, even considering a reliable set of bugs for performing our analysis, the overall results presented low accuracy. From the 5 non-overlapping items, 4 of them were related to bugs on utilitarian classes – mainly related to data conversion and validation methods. As those classes were neither related to any particular architectural role, nor implementing an architecture component, they were not considered architecturally relevant. The other non-overlapping element was a presentation class, which presented a bug related to the graphical user interface.

Those results indicate that the *error-proneness heuristic* could benefit from architecture information for discarding elements that are not related to any architecture role. By itself, this heuristic might also not be able to identify the most relevant code anomalies, as bugs can be found throughout the systems code elements, regardless of their architecture relevance. Therefore, by combining it

with other heuristics, such as the *architecture role heuristic*, we can possibly improve its results.

Correlation with Actual Architecture Problems

The analysis of correlation with actual architecture problems for the *error-proneness heuristic* presented better results towards detecting relevant anomalies. As we can observe on Table 9, at least 80% of the ranked elements were related to architecture problems, for all of the systems we analyzed.

Table 9 – Error-proneness and actual architecture problems

	# of ranked CE	**architecturally relevant**	**% of architecturally relevant**
HW	14	12	85%
MM	10	8	80%
PDP	10	8	80%

CE – code elements

The most significant results were obtained for the HealthWatcher system, with 85% of the ranked elements related to architecture problems. That number is even more significant when we consider that the ranking for HealthWatcher was composed of 14 elements, instead of only 10. Furthermore, it is important to mention that the rankings for HealthWatcher and MobileMedia were built over automatically detected bugs. This shows us that even when formal bug reports are not available, the use of static analysis tool for predicting possible bugs might be useful.

Regarding the PDP system, which was the only one where actual bug reports were considered, the results were also promising: from the top 10 ranked elements, 8 were related to architecture problems. Considering that PDP had 97 code elements, with 37 of them related to architecture problems, that means the remaining 29 were distributed among the 87 bottom ranked elements. When we extended the analysis over the top 20 elements, we found an even better correlation factor: 17, or 85% of the top 20 most error-prone elements were related to architecture problems.

4.3.3. Evaluation of the Anomaly Density Heuristic

The Anomaly Density heuristic was applied to all of our target systems. We analyzed 178 code anomalies from MIDAS, 273 from HealthWatcher, 176 from MobileMedia and 175 from PDP – totaling 802 code anomalies. The results for this heuristic are shown on Table 10:

Table 10 – Results for the Anomaly Density Heuristic

	Overlap		NSF		NF	
	Value	**accuracy**	**Value**	**Accuracy**	**Value**	**accuracy**
HW	5	50%	0,66	34%	0,54	46%
MM	7	70%	0,41	59%	0,70	30%
PDP	8	80%	0,37	63%	0,36	64%
MIDAS	9	90%	0,4	60%	0,2	80%

As we can see, this heuristic had many good results in terms of accurately ranking architecturally relevant anomalous elements. In fact, good results were obtained not only when correctly selecting the top 10 (as evidenced by the number of overlaps) but also when defining their ranking positions: only 2 out of 8 measures had low accuracies, according to our accuracy level thresholds (Section 4.2.1).

The number of overlaps in this heuristic was considered highly accurate in 3 out of the 4 analyzed systems. This is an indication that code elements infected by multiple code anomalies are often perceived as high priority by maintainers. The only system where this did not occur was HealthWatcher, which had only 5 overlaps. When analyzing the number of anomalies for each element on the ranking representing the ground truth, we found that many of them had exactly the same number of code anomalies, namely 8. However, around 40 classes in the entire system (or 30%) had at least 8 anomalies, which is why those elements did not appear in the top 10 ranking. In fact, the 10^{th} element in the ranking was infected by 13 anomalies. It is important to mention that for this heuristic, in contrast to the *change-proneness* and *error-proneness* heuristics, we only considered the top 10 elements for the HealthWatcher system, as there were not ties to be taken into consideration.

MIDAS had a significant number of overlaps – 9 out of 10 elements appeared in both rankings. However, this was highly expected, as this system was

composed of 21 code elements only. Nonetheless, the NSF and NF measures also presented a high accuracy – meaning that the rankings were similarly ordered. The NF measure had a better result, influenced by the fact that the only mismatched element was ranked as #10 – a low position. As explained on Section 4.2.4, missing items appearing in low positions have a positive effect on the NF measure – i.e., they decrease the distance between the rankings, and increase the accuracy.

MobileMedia had the most discrepant results regarding the two ranking measures: 59% accuracy for the NSF measure, and 30% for the NF measure. Such difference was also related to the position of the non-overlapping elements in the ranking generated by the prioritization heuristic: the elements ranked as #1, #3 and #4 were all missing from the ranking provided by developers. Therefore, their ranks were assigned to *k+1* in the developers list, which resulted in a huge distance from their original positions. Those elements comprehended a data model class (core.ui.datamodel.ImageMediaAccessor), a utilitarian class (core.util.MediaUtil) and a base class for controllers (core.ui.controller.BaseController). Intuitively, utilitarian classes are not cohesive and often contain multiple methods – characteristics that favor the presence of code anomalies. The ImageMediaAccessor class was infected with 23 code anomalies – mostly long methods – and was also involved with architecture problems. However, it was not mentioned by developers in their rankings. Finally, the BaseController class, although missing from the original ranking, is the base class for all the other controllers in the MobileMedia architecture, which correspond to 9 out of the top 10 elements ranked by developers.

By analyzing the results for this heuristic, we observed that code elements infected by multiple code anomalies are often perceived as high priority. This could be an indication that automatic detection of code anomaly patterns is desirable (ARCOVERDE *et al.*, 2012; MACIA *et al.*, 2012c), as those groups of code anomalies could be identified together. We also identified that many false positives (i.e., classes that are infected by multiple code anomalies, but are not related to architecture problems) could arise from utilitarian classes, as those classes are often large and not cohesive. By combining this heuristic with the *architecture role heuristic*, we could help discarding such results.

Finally, the results obtained in this analysis also helped us rejecting the null hypothesis $H3_0$ – 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 project in 2 out of 3 measures (90% for the overlaps and 80% for NF).

Correlation with Actual Architecture Problems

When analyzing the rankings produced by the *anomaly density heuristic*, comparing them to the actual architecture problems, the results were not consistent through all the analyzed systems. For the HealthWatcher system, only 5 out of the 10 top ranked elements were related to architecture problems, as shown on Table 11. Those results are similar to the ones found when comparing the ranking to the ground truth provided by the architects. In fact, we the 5 elements related to actual architecture problems are exactly the 5 overlapping items between the compared rankings (Table 10). The reason why that happened for HealthWatcher is related to the high number of anomalies, concentrated in a small number of elements, that were not architecturally relevant. In this context, all of the 5 elements not related to architecture problems were data access classes, responsible for communicating with the database.

Table 11 – Anomaly density and actual architecture problems

	# of ranked CE	**architecturally relevant**	**% of architecturally relevant**
HW	10	5	50%
MM	10	9	90%
PDP	10	8	80%
MIDAS	10	6*	60%*

CE – code elements

Another interesting result for this analysis emerged from the MIDAS system. From the top 10 elements with the higher number of anomalies, 6 were architecturally relevant - or 60% of them. However, the MIDAS system has exactly 6 elements that contribute to the occurrence of architecture problems. Thus, the *anomaly density heuristic* correctly outlined all of them in the top 10 ranking. For that reason, those results are highlighted on Table 11.

4.3.4. Evaluation of the Architecture Role Heuristic

For evaluating the architecture role heuristic, we analyzed three systems: HealthWatcher, MobileMedia and PDP. We consulted the original architects of those systems for gathering information on the architecture roles. Table 12 depicts our results.

Table 12 – Results for the Architecture Role Heuristic

	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%

PDP held the most consistent results across the three similarity measures, having around 60% of accuracy when comparing the similarity between the rankings. It was also the only system where we could divide the classes and interfaces in more than three levels when analyzing their architecture roles. In this context, we divided classes and interfaces from PDP in four different roles, as detailed below, in Table 13:

Table 13 – Architecture roles for PDP

Architecture roles	Priority Score	# of CE
Utilitarian and internal classes	1	23
Presentation and data access classes	2	28
Domain model, business classes	4	24
Public interfaces, communication classes, façades	8	6

CE – code elements (classes and interfaces)

From this classification, we were able to draw the architecture role heuristic ranking for PDP, containing all of the 6 elements from the highest category (public interfaces, communication classes and façades) and 4 elements from the domain model and business classes. In this case, we ordered the elements alphabetically for breaking ties. Therefore, although 24 classes had the same score, we only compared the first 4 of them. However, some of the elements

ranked by developers belonged to that group of discarded elements. Had we chosen a different approach, such as considering all the ties as one item, we would turn our top 10 ranking into a list of 30 items and have a 100% overlap rate – as all of the non-overlapping items belonged to the domain model category.

For HealthWatcher and MobileMedia, we followed a different scoring approach, by consulting the original architects for each system directly. They informed us the existing architecture roles, and their respective relevance on the systems' architectures. This procedure is described on Section 4.2.3.3. Once we identified which classes were directly implementing which roles, we were able to produce the rankings for this heuristic.

HealthWatcher presented the worst results among the analyzed systems. However, when producing the ranking for the architecture role heuristic, almost 20 elements were tied with the same scores. We first only selected the top 10 elements, and broke the ties according the alphabetic order of the ranked elements. That led us to an unreal low number of overlaps, as some of the discarded items were present in the ground truth ranking. In fact, because of the low number of overlaps, it would not be fair to evaluate the NSF measure as well – since it is calculated over the overlapping elements only. We then 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 such analysis, the number of overlaps went up to 6, but the accuracy for the NSF measure decreased to 17% – indicating a larger distance between the compared rankings. This shows us that the 50% accuracy for NSF obtained in the first comparison round was misleading, as expected, due to the low number of overlaps. The NF measure, on the other hand, considers missing elements as part of the distance measure. In fact, it computes the non-overlapping items as a mismatch between the rankings – thus, resulting in a more reliable similarity index. For HealthWatcher, that index was .72 – or 28% of accuracy. This number was, nonetheless, highly influenced by the low number of overlapping items – as that indicates a big mismatch between the compared rankings.

As for MobileMedia, high accuracy rates were found for both the NSF and NF measures. We observed when analyzing its architecture documentation that many elements reported as high priority by developers were implementing architecture components. More specifically, there were 8 architecture components

described in the document directly related to 9 out of the top 10 high priority classes, according to the developers' ranking. Furthermore, we identified one architecture component (SMSController) that was implemented by two classes (SMSReceiverController and SMSSenderController). As they were related to the same architecture role, they were both present in the ranking produced by the heuristic. However, only one of them appeared in the top 10 ranking representing the ground truth.

The results for this heuristic are highly dependent on the quality of the architecture roles defined. More specifically, we observed that the best results were obtained for the PDP system, with had multiple architecture roles defined, and with different levels of relevance. Furthermore, this heuristic is probably better when combined to other heuristics, in order to discard elements that do not represent an important architecture role from the final results.

Finally, the results obtained in this analysis also helped us to reject the null hypothesis $H4_0$ – as the *architecture role heuristic* was able to produce rankings with at least acceptable accuracy in all of the systems we analyzed for at least one measure.

Correlation with Actual Architecture Problems

The analysis of correlation between the *architecture role heuristic* rankings and actual architecture problems for each system is summarized on Table 14.

Table 14 – Architecture role and actual architecture problems

	# of ranked CE	architecturally relevant	% of architecturally relevant
HW	10	4	40%
MM	10	9	90%
PDP	10	10	100%

CE – code elements

Again, the results are quite discrepant between HealthWatcher and the two remaining systems. However, for this analysis, the problem resided on the analyzed data. We identified two different groups of architecture roles among the top 10 elements for HealthWatcher, ranked as equally relevant: 6 of the related elements were playing the role of repository interfaces; the remaining 4 elements were Façades (GAMMA *et al.*, 1994) 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 less relevant. That refinement led to a completely different ranking, which went up from 4 to 7 elements related to architecture problems.

The results obtained for HealthWatcher show us the importance of correctly identifying the architecture roles and their relevancies for improving the accuracy of this heuristic. However, this also constitutes a drawback, as we cannot evaluate the heuristic without some input from the original architects, who might not be available or give imprecise information. 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 utilitarian classes. This indicates the need to further analyze different combinations of prioritization heuristics.

4.4. Threats to Validity

This section describes the main threats to validity of our studies and the mitigations we considered.

Validity of detected code anomalies. A threat to construct validity is related to possible errors in the detection of code anomalies in each selected system. As our approach consists of ranking previously detected code anomalies, the method for detecting them must be trustworthy. There are several kinds of detection strategies documented on literature; however, many have been proved inefficient for detecting relevant anomalies in previous studies (MACIA *et al.*, 2012a). We mitigated the risk of imprecision when detecting code anomalies by (i) involving the original developers and architects in this process and (ii) using well-known metrics and thresholds, previously and independently evaluated elsewhere, for constructing our detection strategies (KHOM *et al.*, 2009; OLBRICH *et al.*, 2010).

Identification of errors. Another threat to construct validity is related to how we identified errors for applying the *error-proneness* heuristic. We first relied on commit messages for identifying classes related to bug fixes, which

means some errors might be missing. We tried to mitigate such threat by also investigating issue tracking systems, looking for error reports and traces between those errors and the code that was changed to fix them. Moreover, we also investigated test reports, when available, in order to identify the causes for eventual broken tests. When that information was unavailable, we relied on the use of static analysis methods for identifying bugs (AYEWAH *et al.*, 2008).

Identification of architecture roles. The architecture role heuristic is based on identifying the relevance of a given code element regarding the system's architecture design. Therefore, in order to compute its scores, we needed to assess the roles each code element plays on the overall architecture. This information was extracted differently depending on the project under analysis, which we considered a threat to construct validity. For example, for HealthWatcher and MobileMedia, we studied the architecture documentation, looking for classes that implemented described interfaces or components. For PDP, on the other hand, as the architecture documentation was absent, we interviewed the original architects for identifying those classes. Nonetheless, we understand that the absence of architecture documentation reflects a common situation, and may be inevitable when analyzing real world systems.

Choice of the target applications. The choice of the target applications is related to a threat to external validity. As in every empirical study, our results are limited to the scope of these applications. However, we tried to minimize such threat by selecting systems developed by different programmers, with different domains, programming languages, environment (i.e., academy and industry) and architecture styles. Nonetheless, in order to better generalize the obtained results, our study should be replicated with other applications, from different domains – as long as the selection criteria described in Section 4.1 are respected.