5
Conclusion

The presence of architecturally relevant code anomalies often leads to the decline of the software architecture quality. Unfortunately, the removal of those critical anomalies is not prioritized properly. This happens mainly because existing techniques and tools are not devised to support the prioritization of architecturally relevant code anomalies. Even worse, there is not much empirical knowledge towards the factors that could be used to facilitate the prioritization of code anomalies. In fact, we have observed that developers tend to restructure the source code prioritizing the anomalies that do not affect the architecture design.

In this context, our findings have shown that developers can be guided into prioritizing code anomalies according to architectural relevance. The anomaly prioritization may help developers to optimize the refactoring process through software evolution. In next subsections it is described how the contributions of this dissertation address the aforementioned problem.

5.1.
Dissertation Contributions

This study explores the problem of prioritizing code anomalies based on their architecture relevance. In this context, we outline our contributions below.

Prioritization heuristics. We proposed four prioritization heuristics for ranking code anomalies according to their architecture relevance. Those heuristics were based on four characteristics that might indicate symptoms of architecture problems. More specifically, they explore the change-proneness, error-proneness, anomaly density and architecture role for each infected code element in order to produce prioritization rankings of code anomalies. This contribution addresses our second research question (RQ2).

Evaluation of the proposed heuristics from the architects' point of view. We also evaluated the prioritization heuristics against rankings provided by architects, that represented the main maintainability issues for the systems we

analyzed. This evaluation comprised the comparison between the rankings produced by the heuristics and the ranking provided by the architects. In most cases, our heuristics were able to accurately detect a prioritization order that reflected the most relevant code anomalies, from the perception of the architects. Under this analysis, the prioritization heuristics were mostly useful when:

(i) there were architecture problems involving groups of classes that changed together;

(ii) there were architecture problems related to Façades or classes responsible for communicating with different modules;

(iii) changes were not predominantly perfective, i.e., the majority of the changes performed on the systems were not refactorings;

(iv) there were code elements infected by multiple code anomalies;

(v) the architecture roles are well defined and have distinct architecture relevancies.

Therefore, the main contributions of these results to the state-of-art are providing knowledge on (i) which factors and (ii) to what extent they could help developers prioritizing code anomalies according the architects' point of view.

Evaluation of the proposed heuristics with actual architecture problems. Besides evaluating the proposed heuristics from the point of view of the architects, we also analyzed whether they were correctly prioritizing code anomalies related to actual architecture problems. Overall, our results show that most of the elements ranked by our heuristics belong to the set of architecturally relevant code anomalies. In particular, the *error-proneness heuristic* consistently presented an accuracy of 80% in the worst case. Another successful case is related to the *change-proneness heuristic*. According to our analysis, between 70% and 100% of its top 10 elements were related to architecture problems. Therefore, these results provide evidences that the proposed heuristics could be used to guide developers towards the prioritization of code anomalies, when performing architecture revisions based on the source code

Tool support. Another contribution of this work was the implementation of a tool for applying the prioritization heuristics automatically. Such tool was developed as an extension for SCOOP (SCOOP, 2012), which is an ongoing implementation for detection of architecturally relevant code anomalies.

Finally, there is not much knowledge in the literature about the prioritization of code anomalies. Our exploratory study represents a first effort to address this gap. The dissertation presents in detail a systematic study, which evaluates the usefulness of the prioritization heuristics proposed. We accurately prioritized architecture relevant code anomalies in different levels, for 4 different software projects. Our evaluation has shown that this prioritization could help architects and developers to better invest their refactoring efforts, into removing architecturally relevant code anomalies.

5.2. Future Work

The results obtained and the aforementioned contributions represent an initial effort into investigating the prioritization of code anomalies according to their architecture relevance. We identify in this section our future plans towards improving and extending this study.

1. *Evaluate different combinations of the proposed heuristics*

Although the current implementation of our tool supports the combination of different heuristics (Section 3.2.2), we did not evaluate the benefits of such combinations on the prioritization results. In this context, we intend to investigate whether combining different heuristics improve the accuracy of the resulting rankings. We also plan to verify whether there are specific combinations that are always successful, regardless of the systems architecture designs. It would be interesting to identify which combinations of heuristics, as well as their respective weights, produce the best results. Finally, it would be also interesting to analyze to what extent those combinations would enable the prioritization of architecturally relevant code anomalies in the software project history. The separate use of certain heuristics, such as the change-proneness and error-proneness heuristics, might suffer from the problem of identifying critical anomalies too late in the project history.

2. *Discuss results with developers*

Our evaluation for the proposed prioritization heuristics used as input rankings of code anomalies provided by developers, which were compared to the

ones generated by our prioritization heuristics. As future work, we intend to discuss those comparisons and their results with the developers, providing and gathering feedback regarding how accurate they are. This feedback could help us to identify opportunities for improving our heuristics or even to propose new ones. More specifically, it would be interesting to analyze whether the generated rankings include architecturally-relevant code anomalies that developers did not anticipate.

3. *Evaluate the heuristics efficacy*

Our results show that the use of prioritization heuristics could help developers to identify better refactoring candidates, towards solving possible architecture problems. As future work, we intend to realize supervised studies with groups of developers, for analyzing whether those heuristics are indeed helping them to prioritize their refactorings. Our intention is to observe whether there was an increase in the proportion of refactorings aimed at removing architecturally relevant code anomalies. It would also be interesting to analyze whether the prioritization heuristics help increasing developers' productivity when identifying the main problems in their code bases. Furthermore, both the heuristics and their implementation could be evaluated against real development scenarios.

4. *Improve the implementation of the heuristics*

The main focus of this study was proposing and evaluating prioritization heuristics. Although tool support for applying them automatically was an important contribution, there are still many possible improvements to the current implementation. First, we intend to provide more flexibility to the definition of architecture roles and the code elements that implement them, as required by the *architecture role heuristic* (Section 3.1.4). Second, we intend to improve the graphic interface and usability of our tool. More specifically, we plan to improve the visualization mechanisms for presenting the prioritization results, allowing developers to easily switch between the heuristics results. Finally, we intend to improve the *error-proneness heuristic* mechanism, integrating it to issue tracking systems APIs.