6 Final Remarks

In this chapter, we discuss the main threats to validity that permeates the empirical studies performed in this thesis (Section 6.1). After that, we discuss the conclusions about the results achieved in this thesis (Section 6.2), as well as we summarize the main contributions and points out the activities to be performed as future works (Section 6.3).

6.1. General Threats to Validity

This section discusses the main threats to validity according to the guidelines defined in (Wohlin *et al.*, 2000). These threats are categorized in three categories, addressing internal, external and construct validity.

Internal Validity. Our first internal validity threat is related to the quality of the architecture blueprints used in the thesis. Three properties have been presented (Section 2.4) in order to make it clear how we selected architecture blueprints that reach a minimum quality, so that it can be used as artifact on the prioritization process. However, as the system evolves it is hard to synchronize changes in the system architecture, and the code elements in the system implementation. Thus, changes in the architecture blueprints and in the source code might lead to different results when prioritizing and ranking critical code anomalies. Our second internal threat is related to the mapping between architecture blueprints and source code elements. The mapping process is out of scope of our research. However, in order to mitigate this threat, we have validated the mappings with the system architects and developers. Even though there might be some imperfections on the mapping, they provide the information required to execute the heuristics proposed in our study.

External Validity. Our first external threat is related to possible errors on the detection of anomalies. As the architecture sensitive heuristics consist of prioritizing previously detected code anomalies, the method for detecting those code anomalies must be trustworthy. To avoid the risk of imprecision on the detection process: (i) the original developers and architects have been involved in this process; and (ii) well-known metrics and thresholds were used by the detection strategies. The second external threat is related to the use of the ground truth. Although each system experts have used their own strategies to identify the most critical code anomalies, around 75% of code anomalies could be equally identified. Therefore, the ground truth of code anomalies had to be produced as a joint decision between all the system experts involved in the study.

Construct Validity. Our construct validity threat is related to the choice of the target applications. The problem is that the results found in our study are limited to 3 target applications. In order to minimize this threat, we selected systems developed by different programmers, with different domains, programming languages and architectural styles. However, in order to make the results more generalizable further empirical investigation is required. For instance, other researchers could replicate our study using systems from different domains. We have tried to make our best to describe carefully our solution so that others can replicate it using, for example, other systems.

6.2. Revisiting the Thesis Contributions

In this thesis we discussed how architecture blueprints, representing some common types of architectural information, could be used as means to support the prioritization and ranking of critical code anomalies. In this direction, we performed empirical studies in order to reveal how inconsistencies in the system's descriptive architecture would be associated with the presence of critical code anomalies in the system actual implementation. In addition, we documented scenarios where critical code anomalies can be associated with architecture degradation symptoms, which in turn, can be visualized through the analysis of architecture information represented in blueprints. We proposed and evaluated heuristics aiming at providing semi-automated support for prioritizing and ranking critical code anomalies as early as possible in the system development. When critical code anomalies associated with drift problems are correctly addressed earlier in the system development, architecture degradation symptoms can be avoided.

Therefore, we now revisit the main contributions of this thesis and lessons learned observed from each empirical study. The goal of this retrospective analysis is to reflect upon the research questions initially defined in this thesis. In this sense, we summarize the contributions for each empirical study and discuss how how their findings contribute to our final conclusions. These conclusions regard the different aspects on the use of architecture blueprints for prioritizing and ranking critical code anomalies. It also important to mention that all the results presented in Chapters 3, 4 and 5 has been either published or submitted to international conferences in the field of software engineering.

On the Relation of Architecture Blueprints and Code Anomalies: A Study of Evolving Software Systems. We have initially studied how AO blueprints, representing the system's descriptive architecture, would help to reveal inconsistencies in architecture design decisions in relation to the actual implementation. In this sense, our analysis relied on evaluating how inconsistencies in the system's descriptive architecture would be related to the presence of anomalies when observing the actual implementation of two target applications. Thus, we investigated how certain modularity properties, represented in the architecture blueprints, could help to void inconsistencies in the architecture decomposition. We also investigated the correlation between the presence of anomalies in the source code and inconsistencies observed in the descriptive architecture. From this empirical investigation we observed that, since the AO architecture blueprints have an improved modularization, we could better localize inconsistencies in the descriptive architecture in relation to the system's actual implementation.

In addition, we performed the analysis of two modularity properties associated to systems implement using the aspect-oriented software development paradigm, *obliviousness* and *quantification*. Firstly, we found that components in the architecture blueprints with higher obliviousness tended to present lower number o inconsistencies. Secondly, we also observed that aspectual components with higher quantification are often related to inconsistencies with the system's actual implementation. Thus, inconsistencies in the AO architecture blueprints are usually also be related to the presence of anomalies in the source-code. Moreover, most part of those anomalies is related to the misuse of AO mechanisms. In this sense, when using aspect-oriented software development paradigm, developers should avoid cases where aspectual components have a high quantification and obliviousness is low. In addition, some inconsistencies might be observed in the AO architectural design, mainly when architectural components are implemented by anomalous code elements. Thus, developers must prevent the overuse of aspects with high quantification, particularly those pointcuts that are associated with the pointcut-related code anomalies.

Controlled Experiment for Prioritizing and Ranking Critical Code Anomalies vs. Usefulness of Architecture Blueprints. Based on the finding that architecture blueprints could be used as means to reveal how inconsistencies associated to the presence of anomalies in the source-code, we decided to on a more in-depth analysis. Thus, we investigated how the use of architecture information could be properly use to prioritize and rank the critical code anomalies. Therefore, we focused on figuring out what architectural information could be more effectively used in the prioritization and ranking process. The prioritization and ranking of code anomalies associated with problems in the system's descriptive architecture, as early as possible in the development of software systems, might assist architects and developers on preventing architecture degradation symptoms. Thus, we performed controlled experiments, with participants from different universities and with different technical knowledge, in order to evaluate how the use of architecture blueprints would improve the prioritization and ranking of code anomalies. For doing so, we used three different measures, namely Precision, Recall and Time Spent. As lessons learned from the controlled experiment, we observed that: (i) architecture blueprints could somehow improve **Precision** measures on the prioritization and ranking process, although the statistical tests indicated the results as being marginally statistical significant; (ii) a positive impact in terms of **Recall** when architecture blueprints on the process of prioritizing and ranking critical code anomalies; and (iii) besides improving the **Precision** and **Recall**, the use of architecture blueprints did not bring any additional effort in terms of time spent. Therefore, unlike we initially assumed, when participants are provided with additional artifacts for prioritizing and ranking code anomalies, it does not necessarily mean they will spent more time for analyzing those additional artifacts. Besides the experimental tasks, participants should also provide feedback about the usefulness of the architecture information provided in the blueprints when performing the experimental tasks. Around 71.4% of participants claimed the architecture blueprints as being useful when prioritization and ranking process code anomalies, while 28.6% said they relied only in the use of source code metrics when performing the experimental tasks. On the other hand, the latter suggested other architecture information that would be helpful when prioritizing and ranking critical code anomalies.

Semi-Automated Heuristics for Prioritizing Critical Code Anomalies. Based on our previous research findings, we observed the need for automating the process of prioritizing and ranking critical instances of code anomalies. According to the feedback provided in our controlled experiments, we figured out that not always developers would be able to optimize the prioritization and ranking of code anomalies. The reason is that even though the use of architectural information can improve the prioritization and ranking critical code anomalies, it is not possible to know which specific architectural information should be used when performing those activities. Secondly, we observed that manual prioritization and ranking of code anomalies is far from being trivial, although the controlled experiment revealed that no additional time was required to perform both activities. However, the Precision and Recall measures were somehow compromised by the misinterpretation of existing artifacts, including the architectural information represented in the blueprints. In this sense, we provided a semi-automated way for prioritizing and ranking critical code anomalies based on their architectural relevance. The architectural relevance is assessed in the context of different architectural problems to which an anomalous code element might be associated. Besides the existing drift problems documented in the literature (Garcia *et al.*, 2009b), we studied drift problems in terms of violation of design principles. The drift problems documented can co-exist with other definitions. For instance, *External Addictor Component* is often associated with *Overused Interface*. These architectural problems are also related, for instance, to the violation of two different design principles, namely *Single Responsibility Principle* and *Interface Segregation Principle* (Martin, 2003). Finally, we observed that each different architectural drift problems required different architectural information when compared to the other architectural problems. In this sense, we defined different information to be explored by the architecture sensitive heuristics. Moreover, we also used architecture sensitive metrics in order to break ties and make the process of prioritizing and ranking code anomalies more accurate when compared to the *ground truth* provided by the system experts.

Architecture Sensitive Heuristics vs. Architectural Drift Symptoms. In order to help developers on semi-automating the process of prioritizing and ranking critical code anomalies according to their architectural relevance, we proposed 4 different prioritization heuristics. The heuristics are based on different architectural problems related either with problems in the inter-component communication or with the implementation of concerns in a software system. In addition, we evaluated the proposed heuristics with 3 medium-size applications, as well as discussed how the proposed heuristics would assist developers when correctly prioritizing and raking the critical code anomalies in the target applications under analysis. When evaluating the proposed heuristics we observed that several architecture drift problems involved architecture components infected by multiple anomalies. Moreover, we also observed that even in systems where the concerns are well modularized, the prioritization heuristics were efficient to pinpoint problems with the implementation of those concerns. Finally, when considering an overall analysis of the architecture sensitive heuristics, we were able to effectively prioritize and rank critical code anomalies according to different architectural drift problems.

6.3. Future Works

In addition to the contributions of this thesis described in Section 6.2, we have identified needs of future work. Basically, five main topics can be derived and they are described as follows.

***Further Evaluations*:** The proposed architecture-sensitive heuristics were evaluated in the context of three representative Java systems (Chapters 5). The performed studies provided evidence with respect to the benefits of exploiting architecture information related to the system's descriptive architecture. In addition, we explored the mapping between the descriptive architecture and the code elements in the actual implementation of different software systems. However, further empirical investigations are still required. Firstly, our evaluation focused on more stable versions of three different medium-size systems with different architectural drift symptoms. Therefore, it could be interesting to perform further studies considering the further evolution of those software systems. Moreover, we could also consider larger systems were architecture problems can be more severe if critical code anomalies are not properly addressed. The goal of this kind of study is to gather findings about how early the critical code anomalies are formed, as well as understanding to what extent developers can properly prioritize and rank those critical code anomalies in more complex situations. We could also assess how software developers can avoid more severe architectural degradation symptoms and save effort in later maintenance tasks. Finally, when evaluating the proposed heuristics, we relied on the study of object-oriented systems. Therefore, it is necessary to replicate the studies in systems developed with other programming techniques (e.g. aspect-oriented programming, procedural programming, and others).

Evaluate different combinations of the proposed heuristics. Although the proposed heuristics provide semi-automated support for prioritizing and ranking critical code anomalies, we did not evaluate the benefits of combining them on the prioritization results. Therefore, we intend to investigate whether combining different heuristics would improve the accuracy of the results when prioritizing and ranking code anomalies. Moreover, it would be also interesting to analyze to what extent those combinations would enable the prioritization and ranking of code anomalies in different versions of software systems. In this sense, we could also consider combining the proposed heuristics with those documented by Arcoverde *et al.*, (2012), which explore the code evolution history of software projects.

Evaluate the heuristics efficacy. Our results show that the use of proposed heuristics could assist developers when prioritizing and ranking potential refactoring candidates, according to their architectural relevance. As future work, we intend to realize controlled experiments with groups of developers, for analyzing whether those heuristics are indeed helping them prioritize their refactorings. Our intention is to observe whether there was an increase in the proportion of refactorings aimed at removing critical code anomalies. It would also be interesting to analyze whether the architecture sensitive heuristics would help increasing developers' productivity when prioritizing and ranking the most critical anomalous code elements.

Implementation of the heuristics. One of the main focus of this thesis was to propose and evaluate heuristics for prioritizing and ranking critical code anomalies.

Although the actual solution provides semi-automated ways for executing the prioritization heuristics, we consider providing developers with an Eclipse-based integrated solution – i.e. enabling the use of the prioritization heuristics while programmers progressively edit their code. In addition, we intend to provide more flexibility in the definition of the mappings between the architecture and source code elements, since in the current solution, we assume developers would perform the mappings between architectural elements and concerns to code elements using the *ConcernMapper* tool (Robillard and Warr, 2005). There are any concern mining techniques in the literature, which can automate the mapping process, but we know they are not perfectly accurate.