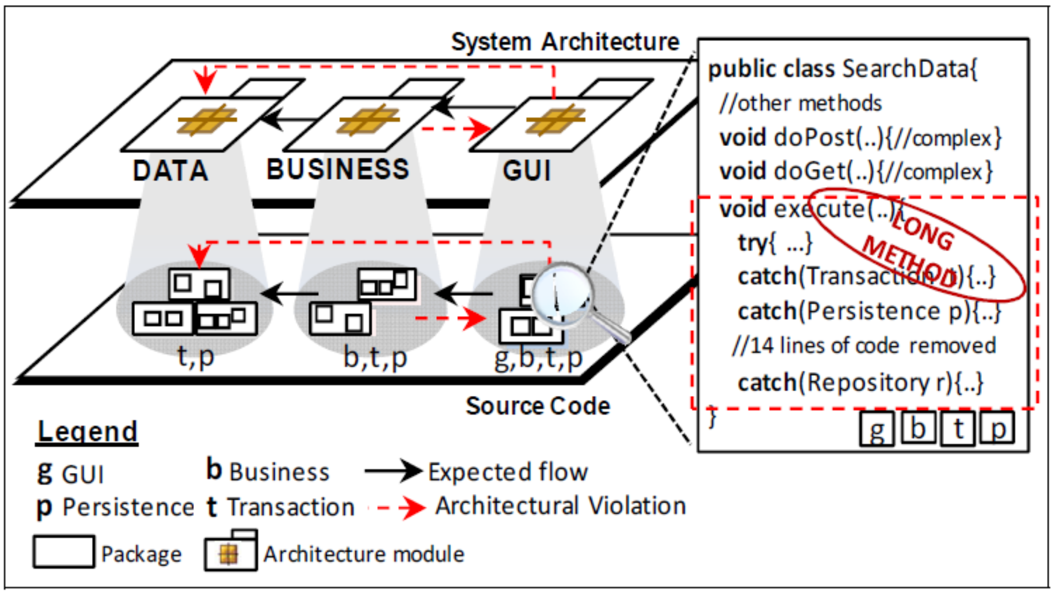
1
Introduction

Code anomalies, also referred to as code smells in the literature (FOWLER *et al.*, 1999), are symptoms in the source code of a deeper maintainability problem. Their occurrences often represent structural problems, making code less flexible, harder to read and to maintain. However, code anomalies are even more harmful when they negatively impact the modularity of the software architecture. According to Bass (2003) software architecture is the structure of a system, which comprises software modules, the interfaces of those modules, and the relationship among them. In many cases, instances of code anomalies can contribute directly to the decay of the software architecture. In fact, previous studies have confirmed that the progressive manifestation of code anomalies is a key symptom of architecture quality decline, or *architecture degradation* (HOCHSTEIN and LINDVALL, 2005; WONG *et al.*, 2010).

The term architecture degradation was introduced by Hochstein and Lindvall, when referring to the continuous quality decline of architecture designs in evolving software systems (HOCHSTEIN and LINDVALL, 2005). When architecture degrades, the maintainability of software projects can be compromised irreversibly (MACCORMACK *et al.*, 2006), and, eventually, a complete redesign is inevitable. An *architecturally relevant* code anomaly represents a symptom of an architecture problem in the implementation. Examples of architecture problems are Overused Interfaces (MARTIN *et al.*, 2002), Ambiguous Interfaces (GARCIA *et al.*, 2009) and cyclic dependencies between modules. Figure 1 illustrates an example of how architecture problems and code anomalies could be related.



```
public class SearchData{
  //other methods
  void doPost(..){//complex}
  void doGet(..){//complex}
  void execute(..){
    try{ ...}
    catch(Transaction t){..}
    catch(Persistence p){..}
    //14 lines of code removed
    catch(Repository r){..}
}
```

Figure 1 - Impact of code anomalies on architectural problems

The code example on the right hand side of Figure 1 depicts a code anomaly extracted from the HealthWatcher system (Section 4.1), detected by developers - namely, a Long Method (SearchData.execute). It was classified as a Long Method because (i) the method body has *high internal complexity*, and (ii) it implements several system functionalities, e.g. GUI, Transaction, Persistence and Business. High internal complexity in this case is given by the number of lines of code and cyclomatic complexity. Additionally, Figure 1 shows an example of architecture problem, associated with the realization of the exception handling policy. In this example, the SearchData class invokes different services from the Business module. Also, it ends up handling exceptions (e.g. Transaction) thrown by the Data module, including those that should be treated internally. That leads to additional code couplings between elements realizing the Data and GUI modules, resulting in architecture violations. These violations are represented by the dashed red arrows in Figure 1.

Ideally, the software project team wants to maximize the gains in maintainability, removing those anomalies that are critical to architecture designs. Refactoring (FOWLER *et al.*, 1999) is the most common approach for removing anomalies from code, whether they are relevant or not. Recent studies have shown refactoring has become a common practice (XING and STROULIA, 2006; MURPHY-HILL *et al.*, 2009; ARCOVERDE *et al.*, 2011), with well-known benefits (MURPHY-HILL et al., 2009). However, some categories or patterns of code anomalies seem to be, either deliberately or not, ignored. For many reasons, such as time constraints and their inability to identify relevant anomalies, developers often focus on removing only a subset of all code anomalies that infect their projects (KIM *et al.*, 2011; ARCOVERDE, *et al.*, 2011; MACIA *et al.*, 2012b). Moreover, most of the anomalies that are not removed, or prioritized, are architecturally relevant (MACIA *et al.*; 2012b).

The problem is that both detecting and removing architecturally relevant anomalies are not trivial tasks, especially when we consider the absence of proper and up-to-date architecture documentation on industry software systems. When relevant code anomalies are not distinguished from irrelevant ones, developers might waste time removing problems that do not harm the architecture design (MACIA *et al.*, 2012b).

1.1. Motivation and Problem

It is often the case that code reviews need to be performed in order to check how the implementation structure is adherent to the modularity of the software architecture. In the context of those code reviews, a code anomaly usually requires closer, more immediate attention when it is related to architectural problems. However, the removal of architecturally relevant code anomalies is a challenging task, especially when those anomalies are identified late. Furthermore, there is little knowledge on which factors or characteristics are good indicators of architecturally relevant anomalies.

This lack of knowledge implies that developers tend to not prioritize the code anomalies that are relevant to the architecture modularity adherence. In fact, there is growing empirical evidence that refactorings are more frequently performed on structures narrowly scoped than on those widely scoped (DIG *et al.*, 2006; MURPHY-HILL *et al.*, 2009; MACIA *et al.*, 2012b). Those *low-level* refactorings are often prioritized due to the localized nature of their changes (ARCOVERDE et al., 2011), as their impacts do not affect other modules and are not propagated through client modules. However, and for that same reason, low-level refactorings are unable to restructure architecture problems, as they do not change hierarchies, dependencies or public structures. In fact, in recent studies we found that around 70% and up to 95% of all refactorings performed through the evolution of a software project can be classified as low-level (MACIA *et al.*, 2012b), and are not handling architecturally relevant code anomalies. As a consequence, architecture degradation could not be avoided.

Furthermore, we performed an analysis on several software systems for understanding what proportion of their code anomalies were causing architecture problems. We found that around 40% of the entire set of code anomalies was not architecturally relevant (MACIA *et al.*, 2012b). This result, combined with the aforementioned study, shows us that developers are investing their refactoring efforts into solving code anomalies that do not harm architecture, while postponing those that do. Moreover, motivated by recent studies on the state of practice of refactoring, we wanted to understand (i) the reasons why some code anomalies lingered and (ii) what factors led developers to prioritize the removal of a particular code anomaly (ARCOVERDE *et al.*, 2011).

The aforementioned observations motivated us into studying three main subjects: (i) the factors that contribute to the longevity of code anomalies in software projects, (ii) which characteristics can be explored for distinguishing architecturally relevant code anomalies and (iii) how we can rank anomalies based on those characteristics, helping developers prioritize the removal of relevant code anomalies.

1.2.
Limitations of Related Work

A wide range of code analysis techniques and tools have been proposed for automatically detecting code anomalies (MARINESCU *et al.*, 2004; RATZINGER *et al.*, 2005; MUNRO *et al.*, 2005; LANZA *et al.*, 2006; WETTEL *et al.*, 2008). Those approaches are usually focused on the extraction and combination of static code measures. However, such state-of-art approaches are limited regarding the prioritization of architecturally relevant code anomalies. They are restricted to analyzing the source code structure only, leading to well-known false positives and false negatives (MACIA *et al.*, 2012a).

The aforementioned restrictions make it hard for developers to distinguish and prioritize architecturally relevant anomalous elements that demand immediate attention. Most of those techniques and tools also disregard software project factors, such as architecture designs, frequency of changes and number of errors. Additionally, developers cannot distinguish which anomalous code elements are architecturally harmful without knowing or considering the role that they play in the architectural design. To the best of our knowledge, there are not even empirical studies that investigate and compare the role of such project factors on the distinction and prioritization of architecturally relevant code anomalies.

Several tools have been proposed to enhance refactoring effectiveness. SafeRefactor (SOARES *et al.*, 2010) helps developers to verify whether their refactorings will change the system behaviour, avoiding incomplete or incorrect refactorings. BeneFactor (XI *et al.*, 2012) recommends refactorings based on the observation of manual refactoring steps. However, none of these tools help developers in prioritizing refactorings with respect to their contribution to sustaining the architecture design.

1.3. Goals and Research Questions

Based on the issues discussed in the previous sub-sections, we define our goals as follows:

- To identify the reasons why some kinds of code anomalies seem to linger on code
- To propose and study the usefulness of a set of heuristics for prioritizing code anomalies based on their architectural relevance
- To develop a system in order to support the automatic ranking of the code anomalies, according to the proposed prioritization heuristics.

We intend to achieve those goals by answering the following research questions:

1) Why some (patterns of) architecturally relevant anomalies are postponed or neglected? [RQ1]

2) Is it possible to rank code anomalies based on their architecture relevance with the proposed heuristics? [RQ2]

3) Which characteristics help in ranking code anomalies based on their architecture relevance? [RQ3]

1.4. Preliminary Studies and Tool Support

We started our study by analyzing the causes of the longevity of code anomalies and the reasons why state-of-practice refactoring is not effectively removing relevant code anomalies. For doing so, we carried out a number of different empirical analyses: three exploratory studies and one survey, involving over 50 versions of 6 different systems and more than 300 refactorings.

Furthermore, once we had addressed the problems related to our first research question (RQ1), we could contribute to the implementation of a relevant code anomalies detector, described on Section 1.4.2.

1.4.1. Empirical Studies

When trying to understand the causes why some anomalies seemed to linger on code for longer than others, we conducted a survey on refactoring habits. Our goal was to check to what extent programmers are able to identify, prioritize and remove architecturally relevant code anomalies. The study consisted of a survey which aimed at collecting information regarding which factors influence the persistence of code smells. This survey was applied with 33 developers from different companies. We found that developers often avoid refactoring structures with wide scope, such as public methods or classes, and prioritize less relevant anomalies, with respect to their impact on the system's architecture. This study addresses our first research question (RQ1):

- ARCOVERDE, R.; GARCIA, A.; FIGUEIREDO, E. *Understanding the Longevity of Code Smells - Preliminary Results of an Explanatory Survey*. 4th International Workshop on Refactoring Tools (WRT), held in conjunction with the 33rd International Conference on Software Engineering (ICSE), May 2011.

We also investigated to what extent code anomalies are good indicators of architecture problems, and, more specifically, how often relevant anomalies are refactored. This investigation was important for identifying the level of effectiveness of state-of-practice refactorings, when addressing architecturally relevant anomalies (RQ2). For doing so, we analyzed 40 versions of 6 different real-world systems. We found that 78% of all architecture problems were related to code anomalies, and also that they were not often refactored – only 30% of all the performed refactorings removed relevant anomalies. This study resulted in the publication of a full paper at the European Conference on Software Maintenance and Reengineering; our major contribution to this study was the analysis of how often refactorings remove architecturally relevant anomalies in real software projects:

- MACIA, I.; ARCOVERDE, R.; GARCIA, A.; CHAVEZ, C.; STAA, A. *On the Relevance of Code Anomalies for Identifying Architecture Degradation Symptoms*. Proceedings of the 16th European Conference on Software Maintenance and Reengineering (CSMR), Szeged, Hungary, March 2012.

Once we had identified reasons why some code anomalies are neglected, and how often they affect the preservation of architecture design, we started exploring some ideas for detecting and ranking them. According to the previous study (MACIA *et al.*, 2012b), one of the main reasons why architecturally relevant anomalies are not removed is the inaccuracy of current techniques for detecting code anomalies. More specifically, such techniques are based solely on the analysis of the source code structure. Thus, we started an exploratory study towards finding which characteristics in an evolving software project could be exploited to guide the ranking of code anomalies – such as change-proneness and error-proneness. We wrote a first draft of our proposed heuristics in a short paper, partially addressing our second and third research questions (RQ2, RQ3):

- ARCOVERDE, R.; MACIA, I.; GARCIA, A.; STAA, A. Automatically Detecting Architecturally-Relevant Code Anomalies. 3rd International Workshop on Recommendation Systems for Software Engineering, held in conjunction with the 34th International Conference on Software Engineering (ICSE 2012). Zurich, Switzerland, June 2012.

Finally, for detecting architecturally relevant code anomalies, we designed a recommendation tool we called SCOOP. SCOOP's detection mechanisms are based on (i) using architecture information, instead of static analysis only and (ii) analyzing groups of anomalies for identifying patterns of code anomaly occurrences. Examples of architecture information explored by SCOOP are mappings between architectural elements and code elements. SCOOP is able to automatically detect and rank code anomalies, based on the prioritization heuristics we proposed, partially addressing our third research question (RQ3). Furthermore, our implementation was accepted for the Tool Demo Track of the International Conference on Software Maintenance:

- MACIA, I.; ARCOVERDE, R.; CIRILO, E.; GARCIA, A.; STAA, A. Supporting the Identification of Architecturally-Relevant Code Anomalies. 28th IEEE International Conference on Software Maintenance (ICSM 2012). Riva Del Garda, Italy, September 2012

1.4.2. SCOOP

In order to detect relevant code anomalies, we contributed to the implementation of SCOOP (SCOOP, 2012), a tool aimed at automatically detecting architecturally relevant code anomalies. This tool was strongly based on the notion that recurring occurrences of code anomalies – or *code anomaly patterns* – and the relationships between them can be better indicators of architecture problems than individual anomaly occurrences. Besides identifying some of those relevant patterns of code anomalies (MACIA *et al.*, 2012a), this tool is also able to detect individual anomalies from metrics-based detection strategies (LANZA and MARINESCU, 2006). We described the main features of SCOOP, as well as our initial ideas for prioritization heuristics, in the aforementioned paper “Automatically Detecting Architecturally relevant Code Anomalies”. It is also important mentioning that, although we have implemented some of its components, SCOOP’s detection mechanism is part of an ongoing doctorate work; our study focuses specifically on prioritizing the anomalies detected, as described in Chapter 3.

1.5. Dissertation Structure

The next chapters have the following purposes:

Chapter 2- *Background and Related Work*: presents general background and outlines related work on code anomalies detection and ranking.

Chapter 3- *Prioritization of Code Anomalies*: describes our prioritization heuristics for ranking code anomalies based on four different characteristics and presents their implementation details.

Chapter 4- *Evaluation*: presents and discusses results for the evaluation of our prioritization heuristics.

Chapter 5- *Conclusion*: discusses the conclusions and the contributions for this dissertation, and describes planned future work.