Automatically Detecting Architecturally-Relevant Code Anomalies

Roberta Arcoverde, Isela Macia, Alessandro Garcia, Arndt von Staa
OPUS Research Group, Informatics Department, PUC-Rio, Rio de Janeiro, RJ, Brazil
{rarcoverde, ibertran, afgarcia, arndt}@inf.puc-rio.br

***Abstract*—Software architecture degradation is a long-standing problem in software engineering. Previous studies have shown that certain code anomalies - or patterns of code anomalies - are likely to be harmful to architecture design, although their identification is far from trivial. This study presents a system for not only detecting architecturally-relevant code anomalies, but also helping developers to prioritize their removal by ranking them. We detect code anomaly patterns based on static analysis that also exploit architecture information.**

***Keywords*-code anomaly, architectural problem, refactoring**

I. ARCHITECTURALLY RELEVANT CODE ANOMALIES

One of the key symptoms of system's degradation is the progressive manifestation of code anomalies [3] - structures that possibly indicate a deeper maintainability problem. In particular, their impact on system maintainability is critical when they introduce problems on the architecture design [2]. The detection and removal of architecturally-relevant code anomalies are important tasks to prolong the system's longevity. When those problems are not solved soon, as the system evolves, its architecture can degenerate, and, eventually, a complete redesign is inevitable [2].

However, detecting architecture problems is particularly hard, since they often arise from recurring inter-related anomalies scattered through different modules - called as *anomaly patterns* in this work. For instance, co-occurrences of *Large Class* [1] and *Shotgun Surgery* [1] tend to be responsible for introducing undesirable dependencies between architecture modules [3]. This problem is illustrated in Figure 1; the class *HWFacade* was classified as a *Large Class* since it defines many methods and realizes several architecture concerns. This class was also classified as a *Shotgun Surgery*, as its services are invoked by many architecture modules (*Symptom*, *Complaint* and *Employee*). This pattern of co-occurring anomalies was related to architectural problems, since *HWFacade* raises exceptions that should be treated internally, increasing the complexity of its many client modules and forcing them to deal with information that they are not interested in.

However, current tools are unable to detect anomaly patterns, as they focus on: (i) analyzing the source code structure solely, while disregarding how it relates to the architecture decomposition, and (ii) solely identifying *individual* code anomalies. Furthermore, these tools can report hundreds of suspects even for medium-size systems, without any indications whatsoever on which are potentially the most harmful ones from an architecture point of view. Being unable to distinguish architecturally-relevant anomalies may guide developers in wrong directions when addressing them [3].

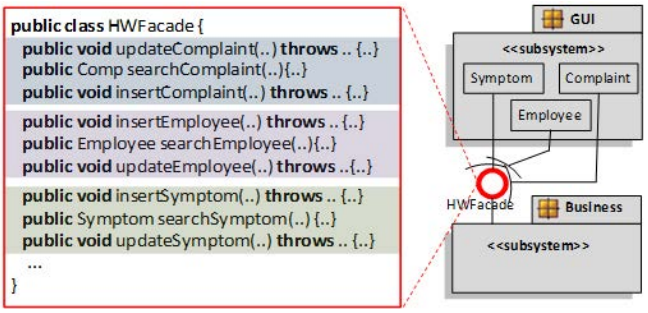

Figure 1 - Architecturally-relevant code anomaly pattern

II. A SYSTEM FOR PRIORITIZING CODE ANOMALIES

This paper presents a system to recommend and rank architecturally-relevant code anomalies. The system is rooted on the idea of detecting code anomaly patterns. Its objective is to help developers finding code structures that cause architecture problems more accurately, and, therefore, helping their prioritization.

In order to correctly identify architecturally-relevant code anomalies automatically, we had to decide which kinds of architectural information would be taken into consideration for classifying anomalies as relevant or not. As previous studies have shown [3][4], these kinds of information could vary from which classes implement which concerns to static structure of the source code, like dependencies between classes. The main modules of the system that collect and analyze this information are discussed below.

A. Enriching the Detection of Individual Code Anomalies

Our recommender system relied on detection strategies [6] to identify code anomalies, similarly to state-of-art code analysis tools, such as PMD and NDepend. However, based on empirical evidence [3][4], we use architecture information to improve the effectiveness and expressiveness of detection strategies. For instance, in order to detect if a code element deals with multiple responsibilities, we rely on the scattering degree of architectural concerns. These concerns could be automatically retrieved from concern mining techniques or provided by developers manually. We also exploit mappings between code and architecture elements to detect if a code element introduces a tight coupling between different architecture modules, like *HWFacade* in Figure 1. In this example, the *HWFacade* class was mapped to the Business module, while the *Symptom* module had classes like *UpdateSymptom* and *InsertSymptom* mapped to it. These mappings could be produced previously, by using

architecture modeling tools or retrieved using architecture recovery techniques.

For flexibility purposes, allowing developers to set their own thresholds and detection strategies, we defined a domain-specific language embedded in our system. It allows developers to implement different strategies for distinct projects, and to calibrate their sensitivity accordingly. For example, when analyzing legacy code bases, developers might set less strict thresholds, in order to focus on more critical violations. An example of such language is shown below, implementing the *God Class* detection strategy:

```
codeanomaly<class> God Class : (LOC > 150) or (NOM > 15)
                               or (NCC > 3)
LOC: Lines of Code
NOM: Number Of Methods
NCC: Number of Concerns per Class
```

In this example, in order to detect architecturally-relevant *God Classes*, we are not only looking for complex classes (in terms of size of code elements), like in a conventional strategy, but for those that realize different system's concerns (using the NCC clause). Using this detection strategy, every class with more than 150 lines of code, 15 methods or implementing more than 3 architectural concerns is detected as a *God Class*. Currently, the code metrics are loaded from a CSV file, received as input.

B. Detecting Code Anomaly Patterns

In order to find relevant problems, our system must identify code anomaly patterns by relating architecture information with simple code anomalies. Two of the patterns are explained below. A complete list can be found at [5].

Similar Anomaly Occurrences. In this pattern, the same anomaly infects different classes from the same architectural module. For example, when the systems interfaces are not well modularized, the implementation of layered architectures might lead to a number of *God Classes*, responsible for redirecting calls to appropriate inner layers. This example was found on a number of real world projects [3][4], and was a major cause of architecture problems. For detecting this pattern, architecture information is highly desirable: using the previous example, such detection was only possible when the information regarding which classes belonged to which layers was available.

Anomalous God Parent Class. This pattern also involves *God Class* occurrences. It occurs when a *God Class* anomaly infects a parent class of an inheritance tree and also some of its children. This pattern was a good indicator of architectural problems in previous studies [3]. We observed that several parent classes infected by this pattern: (i) introduced undesirable relationships via their methods' parameter types, and (ii) forced their children to deal with information that they were not interested in. For detecting this pattern, we run the simple anomalies detector and verify, for each infected parent class, whether their children also contain the same anomaly. It is important to note that the effects of other code anomalies can be also propagated in the inheritance tree. Several patterns involving anomalous parent classes are presented at [5].

C. Refactoring Prioritization

The separation between detected code anomaly patterns and individual code anomalies might already help developers prioritize their refactorings. However, the anomalies detection could bring false positives. For example, classes that implement the Façade pattern might be classified as *God Classes*, even though they are not architecturally relevant - as they could be naturally related to several concerns. To better indicate relevant problems, we describe in this section a ranking system for helping on refactoring prioritization.

Many factors could influence how harmful an anomaly really is; as an exploratory study towards finding the best heuristics for refactoring prioritization, we chose to analyze different characteristics. For example, change-proneness and fault-proneness, retrieved from version control and bug tracking systems, could be considered when analyzing which classes should be prioritized. We are focusing on analyzing how effective those different heuristics are, by comparing their results to prioritization ranks provided by developers.

From the architecture perspective, there are two heuristics we consider relevant: the number of anomalies found per module, and the component's architecture role. Regarding the first one, classes with more anomalies are considered high-priority targets for refactoring. As for the second one, when architecture information is available, the role a component plays on the overall architecture model influences its priority level. For example, classes that implement public interfaces, like Façades, are more critical than those with internal responsibilities only.

All of the proposed heuristics are complementary; depending on the developers need, different heuristics may be more suitable for refactoring prioritization than others. A list of the proposed heuristics is present at [5].

III. Current Status and Future Work

Our current implementation is able to detect all the code anomaly patterns identified in previous studies [3,4]. Additionally, many simple code anomalies can be identified, using rules expressed in the embedded DSL. As future contributions, we plan to (i) evaluate the tool on real world projects where the architecture information may be absent and (ii) implement macro-refactorings recommendations, aiming at correctly removing those code anomalies. We also intend to empirically evaluate the effectiveness of the prioritization heuristics described in section II.C.

References

[1] Fowler, M. Refactoring: Improving the Design of Existing Code. Addison-Wesley, 1999.

[2] Hochstein, L. and Lindvall, M. Combating Architectural Degeneration: A Survey. Info. And Soft. Technology. July, 2005.

[3] Macia, I. et al. Are Automatically-Detected Code Anomalies Relevant to Architectural Modularity? In Proc. of 11th AOSD 2012.

[4] Macia, I. et al. On the Relevance of Code Anomalies for Identifying Architecture Degradation Symptoms. In Proc. of 16th CSMR 2012.

[5] SCOOP: http://www.inf.puc-rio.br/~ibertran/rsse12.

[6] Marinescu, R. Detection Strategies: Metrics-based rules for detecting design flaws. In Proc. Of 20th ICSM, 2004.