Detecting Architecturally-Relevant Code Smells in Evolving Software Systems

Isela Macia Bertran
Informatics Department, Pontifical Catholic University of Rio de Janeiro, Brazil
ibertran@inf.puc-rio.br

ABSTRACT

Refactoring tends to avoid the early deviation of a program from its intended architecture design. However, there is little knowledge about whether the manifestation of code smells in evolving software is indicator of architectural deviations. A fundamental difficulty in this process is that developers are only equipped with static analysis techniques for the source code, which do not exploit traceable architectural information. This work addresses this problem by: (1) identifying a family of architecturally-relevant code smells; (2) providing empirical evidence about the correlation of code smell patterns and architectural degeneration; (3) proposing a set of metrics and detection strategies and that exploit traceable architectural information in smell detection; and (4) conceiving a technique to support the early identification of architecture degeneration symptoms by reasoning about code smell patterns.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics; K.6.3 [**Software Management**]: Software Maintenance;

General Terms

Measurement, Design, Experimentation.

Keywords

Architectural Degeneration, Design Rule, Code Smells.

1. INTRODUCTION

A key motivation to remove code smells [1] via refactoring is to avoid the early deviation of a program from its intended architecture design. The source code is the target of day-to-day changes, which often result, in turn, in the introduction of a multitude of additional code smells. However, refactorings are not applied if they are complex, error-prone or time-consuming or, more importantly, not (seen as) critical to maintain the long-term stability of its intended architecture [11]. Given time and resource constraints in software projects, developers often need to identify first which code smells are actually harmful to the sustenance of the architecture design. Otherwise, as the system evolves, its actual design can deviate substantially from the architecture as originally prescribed. If critical code smells are not revealed early in a project, architectural degeneration may eventually reach a level where a complete redesign of the software system is required [3].

However, it is far from trivial to perform an early diagnosis of *architecturally-relevant smells*, i.e. those that contribute to violations of prescribed rules or manifestations of architectural smells [2] in the source code. Recent empirical evidence seems to suggest that certain categories or smell patterns in evolving source code tend to be indicators of architectural degeneration [13][14]. For instance, Wong et al [14] observed that when code duplications change frequently together they may deviate from the prescribed decisions. Ratzinger et al. [13] previously observed how code smells may be related to change class couplings. These extra couplings might be the source of critical architectural smells, such as Extraneous Adjacent Connector defined in [2].

Therefore, there is a growing need to study architecturally-relevant smell occurrences in evolving software systems. There is little knowledge about the phenomena related to the manifestation and evolution of smell patterns that are early indicators of architectural degeneration A key obstacle for supporting this investigation is that current techniques for code smell detection solely focus on the analysis of source code structure [7][10]. They disregard architectural design information, which could be traced and blended with the code assets in order better indicate the adherence of the code to its architectural design. Consequently, they often cannot pinpoint architecturally-relevant code smells; instead, they tend to report a long list of smell occurrences, which are often not considered as threats [7][14]. Even worse, the false warnings often cover many parts of the system, making it hard to determine which part should be refactored first if at all.

State-of-the-art techniques for identifying code smells are based on metrics-based detection strategies [7][10] However, these metrics often lead to false positives or false negatives [14]. Figure 1 presents the God Class detection strategy [7] (details can be found in [7]) and a typical false negative. The example was extracted from a real web-based system which is decomposed in three main components (GUI, Business and Data) [4]. The class `InsertComplaint` defines several methods, is coupled to several classes and realizes different system's concerns (highlighted using different colors). However, it was not considered as a God Class due to different reasons. First, the cohesion metric did not report a low value when the class's methods were responsible for realizing several system's concerns. Second, the class methods did not

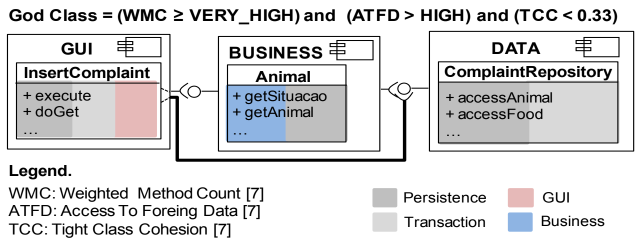

Figure 1. God Class smell: an example and detection strategy.

ICSE'11, May 21–28, 2011, Waikiki, Honolulu, HI, USA
ACM 978-1-4503-0445-0/11/05

present high complexity degree as demanded by the strategy. Instead of this, they presented a high tangling degree of architectural concerns, which is disregarded by the conventional techniques.

However, this class is actually considered a threat since it violates the prescribed design decisions of the well-known Layers architectural pattern. More specifically, it is responsible for introducing undesirable dependencies between GUI and Data components. We also can see how multiple components are responsible for realizing the same architectural concern and, additionally, some of them are responsible for orthogonal concerns. This situation may lead to critical architectural smells such as Scattered Parasitic Functionality [2]. In particular, such false negatives are very problematic; undetected architecturally-relevant smells, like the one above, may be perceived only when architectural degeneration is at an advanced stage. Therefore, detecting architectural degeneration when code smells are introduced allow developers to identify the most prominent problems that should be addressed soon.

This PhD research aims at investigating architecture degeneration symptoms by reasoning about code smell patterns. The first step towards this goal is to identify whether and how code smells and their (co-)occurrences in evolving systems are associated with architecturally-relevant flaws. It is important to point out that the architectural viewpoints to be analyzed in these studies will depend on the architectural decomposition of the target systems. This identification will be performed through exploratory studies gathering empirical evidence about the relationship between code smells and architectural degeneration. The second step is to investigate which kind of architectural information, when traced from architectural specification to source code, could be used in the detection and ranking of smells. To achieve this aim, it is required to identify how such information can be exploited in combination with classical static measures and strategies for smell detection. The third step encompasses the development of tool support for automatic detection of architecturally-relevant code smells in the source code. The fourth and last step is to perform a case study to evaluate our method in industrial software projects.

2. RESEARCH QUESTIONS AND HYPOTHESES

Our research goal is to investigate whether and how architecture degeneration can be detected through the progressive manifestation of smells in evolving systems. This goal leads to the following research questions.

- **RQ1:** Which categories and patterns of code smells are early indicators of architectural degeneration?
- **RQ2:** Which kinds of information traceable from architectural models can help source code analysts to detect architecturally-relevant smells?
- **RQ3:** Do techniques that exploit traceable architectural information in code smell detectors [10][12] present higher precision than conventional techniques?

Our research is based on a family of empirical studies to answer these questions and test the three hypotheses discussed in the following subsections. The gathered evidences will be also used in the construction of the technique for the automated identification of the most relevant smells.

2.1 H1: Smell Patterns as Indicators

Moha et al [10] categorized code smells in order to organize them in different levels of granularity (e.g. intra-class and inter-class smells). However, there is little knowledge about which of these categories, or even individual types of code smells, are usually correlated with architecturally-relevant flaws emerging in the current or later software releases. Understanding this correlation will help programmers to identify and rank smells according to their potential impact on further architectural degeneration. Therefore, developers may focus on the refactoring of the most relevant code smells in order to save resources in an otherwise time-consuming task. This leads to the first hypothesis which assumes that certain smell types and categories exert different impact on the architecture degeneration. To test this hypothesis, we will study if patterns of simultaneous occurrences or evolution of code smells (or their categories) are indicators of architectural degeneration. We will also analyze whether these phenomena can be used as predictors of architectural smells or violations before their actual manifestation.

Hypothesis, H_1: Certain evolution patterns or simultaneous occurrences of code smells tend to be stronger indicators of architectural degeneration

2.2 H2: Smells and Architecture Information

Software architecture encompasses different kinds of design decisions to be adhered by the code through its evolution. Examples of this information are: (i) styles or patterns used in the design of the system's architecture, (ii) concern's scattered degree among different architectural elements, and (iii) decisions that govern the interaction between architecture elements. However, there is little knowledge about how code smell occurrences are related to architectural information. Studying this relationship is crucial because architectural decisions may be exploited by the code smell detectors helping developers to identify architecturally-relevant smells. For instance, the distribution of system's concerns among the architecture's components and their harmful interactions had been key aspects in the detection of the false negative presented in Figure 1. In addition, developers could detect early symptoms of architectural degeneration preventing that it reaches a level where a complete system's redesign would become necessary.

Hypothesis H_2: There is a significant relationship between certain kinds of architectural information and detection of architecturally-relevant code smell occurrences.

2.3 H3: Accuracy Detecting Relevant Smells

We suspect that conventional detection strategies [7][10] do not identify the majority of architecturally-relevant code smells. Our suspicion stems from the fact that they are based exclusively on information that emerges from the source code. However, there is no knowledge about the impact of extending conventional detection strategies to exploit traceable architectural information. That is, we expect that the most severe code smells can be detected with a high accuracy using the code structure in concert with its architectural traces. In this context, our third hypothesis assumes that the most severe code smells can be detected with high accuracy early in an evolving source code if traceable architecture information is exploited in metrics-based detection strategies.

Hypothesis H_3: The accuracy of detecting relevant code smells is higher when combining architectural information with static code analysis than conventional strategies.

3. RESEARCH METHOD AND PROGRESS

In this section, we further explain the proposed solution. To clarify the idea, we detail the notion of architectural degeneration used in our approach.

3.1 Smells and Architectural Degeneration

The first stage in our research aimed at investigating whether (or not) and how code smells are associated with architectural degeneration (H1). In this context, an exploratory study was conducted to analyze the impact of code smells in a total of 18 revisions of 3 evolving real-world systems [4][5][9]. These systems were chosen because they were implemented using hybrid architectural decompositions – i.e. based on multiple architectural styles, including Model-View-Controller, Layers and Aspectual design [6]. A large part of the system was implemented with Java and aspect-oriented programming (AOP). Therefore, some code smells were specific to object-oriented programming whereas others were related to AOP. The architectural analysis was performed by comparing the intended architecture's models (provided by system's architects) versus the actual architecture recovered from the source code. Some architectural viewpoints such as component-connector and modular were analyzed in this study. Architects and developers worked together in the confection of a reliable reference list of actual architectural smells and violations. This reference list was used in our correlation analysis. In order to assess the accuracy of the strategies in the detection of the most relevant smells we have used a reference list of actual smells given and double-checked by the application developers.

Table 1. Smells vs. architecture flaws: a first analysis

Code Smell	Architecturally-Relevant Flaw
God Aspect	Very Strong – 73%
Long Method	Very Strong – 71%
Composition Bloat	Very Strong – 68%
God Class	Strong – 39%
Duplicate Code	Strong – 27%
Shotgun Surgery	Medium – 23%
Forced Join Point	Medium – 21%
Lazy Aspect	Weak – 6%

Our study revealed a new set of code smells. The discovery of these smells was mainly driven by the correlation analysis of potentially-anomalous code structures and maintenance effort. For each code smell candidate, we analyzed if it was correlated with particular refactorings or symptoms of architecture degeneration. Our findings, showed in Table 1, revealed that *non-previously-documented* smells were more related to architectural degeneration than well-known ones, such as Fowler's smells [1] and AOP-specific smells. This is an interesting result as most of the empirical studies of smells and refactorings tend to focus on the narrow set of smells documented in Fowler's catalogue [1]. Furthermore, initial observations suggest that refactorings may also be related to architecture degeneration. We have observed that long-life smells [11] are often related to architecture degeneration (e.g. Composition Bloat [8]). Surprisingly, refactorings' ripple effects might also be a source of design flaws, specifically when they tend to remove short-life smells [12] For instance, ripple effects of God Class and Duplicate Code refactorings can lead to poorly designed inheritance hierarchies (e.g. many small classes or tight class coupling) causing architectural degeneration.

More specifically, we have observed that architecturally-relevant flaws are likely to be provoked by certain patterns of (co-) occurrences or evolutive behaviors of code smells (H1). For instance, in aspect-oriented programs, undesirable relationships between multiple features within an aspect can increase its internal complexity, an alternative condition for the Composition Bloat smell (Figure 2). Improperly feature modularization may lead to Scattered Parasitic Functionality occurrences or violations of the prescribed design rules.

```
public aspect PhotoAndMusicAndVideo{
 ...
 pointcut startApp(MainUIMidlet mdlt):
   execution(public void MainUIMidlet.startpp())
   && this(midlet);
 after(MainUIMidlet mdlt): startApp(mdlt){
  BaseController imgCtr = mdlt.imageRootController;
  BaseController mCtr= mdlt.musicRootController;
  BaseController vCtr= mdlt.videoRootController;
  ... // 11 lines of code removed
  Selectcontroller.setAlgumController(imgCtr);
  selectcontroller.setMusicController(mCtr);
  selectcontroller.setVideoController(vCtr);
  mainscreen.append("Photos");
  mainscreen.append("Music");
  mainscreen.append("Videos");
  ...
 }
}
```

Legend:
▩ Photo Feature ▨ Music Feature □ Video feature

Figure 2. Example of Composition Bloat.

Furthermore, initial results revealed that the fact of conventional detection strategies disregard the architectural information may cause imperfections in their accuracy [8] (H3). This is particularly interesting because, even though our analyses were limited to eighteen releases, our finding may question about the universality of such strategies. For instance, they only identified about the 40% of the code smell occurrences related to architectural degeneration. This rate may be considered low when compared with other empirical studies [7][10]. It is important to point out that in this study we have used the conventional strategies that have presented most effective rates in the literature [7]. Even worse, the majority of the most relevant smells were included in 60% of the instances not detected by the conventional detection strategies (Figure 1). Our next step is to replicate this study in the context of other multi-release systems aiming at collecting additional empirical evidence. We are interested in evaluating systems with: (i) different architectural decompositions and with the actual architecture reached a more degenerated stage, and (ii) a large number of code smell occurrences.

3.2 Blending Architecture Rules with Code

The second step in our research involved two main stages. The first one focused on the investigation of the kinds of architectural information that can help us detecting architecturally-relevant code smells. The second stage was concerned with how to combine architectural design decisions with the static analysis of the source code.

Investigating the Relevant Architectural Information. Specific kinds of architectural information when exploited in the code analysis may help developers to detect symptoms of architectural degeneration (H2). For instance, it would be very useful to trace: (i) how classes and packages are *organized* to represent the hierarchical architectural structure, and (ii) which architectural decisions are governing the *interaction* relationships between the source code elements. These kinds of information when enriching conventional static analysis may detect symptoms of architecture degeneration caused by code smell occurrences (Figure 1). We assume that this approach will address the problem that architecturally-relevant code smells are not frequently detected by conventional techniques. For instance, as we have seen in Figure 1, considering only the information emerged from the source code may lead to false negatives. This smell occurrence had been detected considering, for example, the number of system's concerns being realized by a class (Figure 2).

Exploiting the Architectural Information. We are currently working on the definition of a metric suite and detection strategies to exploit architectural information in code smell detection. Table 2 describes some defined metrics, which will be used in the definition of detection strategies for architecturally-relevant smells. These metrics are based on source code structure enriched with traced architectural information.

Table 2. Software architecture metrics.

Metric	Description
UCC	Number of Undesirable Client Components
UCD	Number of Undesirable Class Dependencies
SCC	Number of System Concerns per Class
SCCo	Number of System Concerns per Component
NCC	Number of Connectors per Component
NDCC	Number of Different Connectors per Component
DDC	Dependency Dispersion between Components

God Class = (WMC > MEDIUM) and (SCCo > LOW) and (UCC > LOW or ATFD > HIGH)

The above equation represents an enriched detection strategy for God Classes, which exploits the architectural information. In order to detect God Class's occurrences that are closely related to architectural degeneration, we are looking for classes with the following characteristics. First, they do not necessarily have a high complexity (in terms of size of code elements) like in a conventional strategy [7]. But, in return, they realize different system's concerns. Second, they introduce undesirable relationships between classes or use a lot of information of other classes. The metrics used for detecting architecturally-relevant God Classes are taken from the conventional detection strategy, i.e. ATFD and WMC, and from others defined in Table 2.

3.3 Initial Proposed Approach

We are currently working on a recommendation system aiming at detecting symptoms of architecture degeneration via code smell occurrences. Our technique is governed by the use of a DSL and 3 key steps. First of all (Step 1), we will exploit Model-Driven Development techniques in order to trace the architectural information and associate it with elements of the source code. Then, the traced information is represented using a DSL. Our intention is that metrics defined in Table 2 consider the architectural information described in the DSL (Step 2). We have decided to represent the relevant architectural information using a DSL because it allows solutions to be expressed at the level of abstraction of the problem domain. The definition of a new DSL is needed because we are searching for a language that provides meaningful architectural abstractions. As a last step, the anomalies detected will be presented in a report ranked according their harmfulness degree. The goal is to make it easier to determine which part of the code should be refactored first.

4. CONTRIBUTIONS AND EVALUATION

The expected contributions of this research are: (1) a family of architecturally-relevant code smells, including the identification of not-previously documented smells; (2) further empirical evidence on whether symptoms of architecture degeneration can be identified via code smells occurrences; (3) lessons learned on how traceable architectural information may help developers in the identification of architecturally-relevant code smells, (4) a set of heuristics and metrics that exploit the architectural information in the detection of code smells, and (5) a technique to support the early identification of architecture degeneration symptoms by reasoning about code smell patterns.

As far as evaluation is concerned, case studies involving various industrial systems will be performed to more precisely test the three hypotheses defined previously. We plan to have available, for each system, their relevant architectural information provided by developers. Code smells will be detected using: (i) conventional tools [10][12] which automate the application of classical heuristics, and (ii) the proposed tool which exploit architectural information to test our hypothesis. We also plan to include developers and maintainers in order to validate the identification of code smells occurrences using both approaches and to discuss the cases that will need more clarification. Finally, our results will be published in order to stimulate further replications of our studies.

5. REFERENCES

[1] Fowler, M. et al. Refactoring: Improving the design of existing code. Addison-Wesley, 1999.

[2] Garcia, D. et al Identifying architectural bad smells. In Proc. of 13th CSMR, 2009.

[3] Hochtein L., Lindvall, M. Combating architectural degeneration: A survey. Info. & Soft. Technology July, 2005.

[4] HW http://www.comp.lancs.ac.uk/~greenwop/tao

[5] iBATIS: Data Mapper - http://ibatis.apache.org/ .

[6] Kiczales, G.,et al. Aspect-oriented programming. In Proc. of 11th ECOOP, 1997.

[7] Lanza, M.; Marinescu, R. Object-oriented metrics in practice. Springer-Verlag Berlin, Germany, 2006.

[8] Macia, I. et al. An Exploratory Study of Code Smells in Evolving Aspect-Oriented Systems. AOSD'11 (to appear).

[9] MM: http://sourceforge.net/projects/mobilemedia/

[10] Moha, N. et al "DECOR: A Method for the Specification and Detection of Code and Design Smells" IEEE Trans 2010

[11] Murphy-Hill, E.,et al A. How we refactor, and how we know it. In Proc. of 31st ICSE, USA, 2009.

[12] Murphy-Hill., E. Scalable, expressive, and context-sensitive code smell display. In Proc of 23rd OOPSLA, 2008.

[13] Ratzinger, J. et al Improving evolvability through refactoring. In Proc. of 5th MSR, May 2005.

[14] Wong S. et al. Detecting Design Defects Caused by Design Rule Violations. In Proc of 18th ESEC/FSE, 2010.