# When Are Smells Indicators of Architectural Refactoring Opportunities? A Study of 50 Software Projects

Leonardo Sousa
ECE - Carnegie Mellon University
leo.sousa@sv.cmu.edu

Willian Oizumi
PUC-Rio, Brazil
woizumi@inf.puc-rio.br

Alessandro Garcia
PUC-Rio, Brazil
afgarcia@inf.puc-rio.br

Anderson Oliveira
PUC-Rio, Brazil
aoliveira@inf.puc-rio.br

Diego Cedrim
Amazon, Brazil
dccedrim@amazon.com

Carlos Lucena
PUC-Rio, Brazil
lucena@inf.puc-rio.br

## ABSTRACT

Refactoring is a widely adopted practice for improving code comprehension and for removing severe structural problems in a project. When refactorings affect the system architecture, they are called architectural refactorings. Unfortunately, developers usually do not know when and how they should apply refactorings to remove architectural problems. Nevertheless, they might be more susceptible to applying architectural refactoring if they rely on code smells and code refactoring – two concepts that they usually deal with through their routine programming activities. To investigate if smells can serve as indicators of architectural refactoring opportunities, we conducted a retrospective study over the commit history of 50 software projects. We analyzed 52,667 refactored elements to investigate if they had architectural problems that could have been indicated by automatically-detected smells. We considered purely structural refactorings to identify elements that were likely to have architectural problems. We found that the proportion of refactored elements without smells is much lower than those refactored with smells. By analyzing the latter, we concluded that smells can be used as indicators of architectural refactoring opportunities when the affected source code is deteriorated, *i.e.*, the code hosting two or more smells. For example, when *God Class* or *Complex Class* appear together with other smells, they are indicators of architectural refactoring opportunities. In general, smells that often co-occurred with other smells (67.53%) are indicators of architectural refactoring opportunities in most cases (88.53% of refactored elements). Our study also enables us to derive a catalog with patterns of smells that indicate refactoring opportunities to remove specific types of architectural problems. These patterns can guide developers and make them more susceptible to apply architectural refactorings.

## CCS CONCEPTS

• **Software and its engineering → Software design engineering**.


*ICPC '20, October 5–6, 2020, Seoul, Republic of Korea*

ACM ISBN 978-1-4503-7958-8/20/05…$15.00
https://doi.org/10.1145/3387904.3389276

**ACM Reference Format:**
Leonardo Sousa, Willian Oizumi, Alessandro Garcia, Anderson Oliveira, Diego Cedrim, and Carlos Lucena. 2020. When Are Smells Indicators of Architectural Refactoring Opportunities? A Study of 50 Software Projects. In *28th International Conference on Program Comprehension (ICPC '20), October 5–6, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3387904.3389276

## 1 INTRODUCTION

Software systems invariably undergo changes that can compromise their structural quality. If these changes are carried out recklessly, software systems may reach a degraded state that requires either significant maintenance effort or the complete redesign [12, 21, 43, 54]. Since these changes are driven by architectural decisions, the introduction of architectural problems is a reason why systems can degrade [21, 54]. An *architectural problem* is the result of one or more inappropriate architectural decisions that negatively impact non-functional requirements [6, 45, 50]. Examples include Fat Interface, Scattered Concern and Component Overload [10, 11, 26].

Empirical studies showed that architectural problems are the most important source of technical debt [7, 8, 44]. A study with 745 software projects showed that technical debt associated to architectural problems was directly related to a significant increase in software project costs [7]. Given the harmfulness of architectural problems, developers should remove them as early as possible. Refactoring, which consists of a program transformation used for improving the system structural quality, is the most common practice to remove architectural problems [9].

As architectural problems are the result of inappropriate architectural decisions, refactorings should affect the system architecture to remove them [17, 47]. Refactorings that also affect the system architecture are known as *architectural refactorings* [19, 39, 41, 47, 61]. An architectural refactoring comprises one or more code-level refactorings aimed to remove architectural problems [60, 61]. For example, a Fat Interface occurs when a developer aggregates non-cohesive functionalities into a single interface [26]. This problem can be removed through an architectural refactoring that combines *Extract Interface*, *Extract Superclass*, and *Move Method* refactorings [9, 26].

Despite the benefits of architectural refactoring, developers have not fully adopted it [8, 15, 61]. Some artifacts (*e.g.*, architectural documentation) that could guide developers to identify architectural refactoring opportunities are unavailable or outdated [14, 50, 61], which contributes to its low adoption. Furthermore, developers perceive architectural refactorings as costly and risky [8, 15] since

they are not confident in recognizing architectural refactoring opportunities, specially without automated support. Consequently, they do not know when and how they should apply architectural refactorings to remove architectural problems. In this scenario, we hypothesize that developers might be more susceptible in applying architectural refactoring if they rely on two familiar concepts that they have on their disposal: code smells and code refactoring [45, 46].

A code smell is a structure in the system implementation that represents a surface indication of architectural problems [9]. Developers have been using smells both as indicators of architectural problems (e.g., [24, 29, 31, 36, 42, 49]) as well as hints for code refactoring [9]. Therefore, developers could also use them to identify architectural refactoring opportunities. For example, classes that implement a Fat Interface usually have smells such as *Divergent Change*, and *Feature Envy*. These two smells are also hints for developers to apply *Extract Interface*, *Extract Superclass*, or *Move Method* refactorings [9], which are exactly the architectural refactorings used to remove the Fat Interface [9, 26]. If developers are aware of the smells that indicate architectural problems and the code refactorings to remove the smells, then they may feel more susceptible to apply architectural refactoring. In this scenario, they will know which smells indicate an architectural problem and which code refactorings should be applied to remove the smells. Consequently, they may remove the architectural problem by removing the smells.

Unfortunately, smells do not always indicate architectural problems [22]. Knowing when smells indicate architectural refactoring opportunities is essential to encourage developers in applying architectural refactoring. This knowledge can improve the state-of-the-art tools that support this activity. For example, a tool can help developers to remove architectural problems by showing in their code (i) the smells that indicate the problem, and, (ii) a ranked list of alternative sets of code refactorings to remove the smells, and, consequently, (iii) removing the architectural problem. Given this context, we investigate when smells can provide to developers opportunities to apply architectural refactorings that might remove architectural problems.

For this investigation, we identified architectural refactorings related to the removal of architectural problems. For this purpose, we located the code refactorings that occurred in the commit history (147,736 commits) of 50 software projects. Differently from other studies [2, 4, 5], we focused on analyzing purely structural code refactorings (also called *root-canal refactoring*). These code refactorings are those primarily targeted at improving the architecture structure [9, 30], thereby contributing to the partial or full removal of one or more architectural problems. After locating the refactored elements, we conducted a manual analysis to identify when developers applied code refactorings to remove architectural problems, *i.e.*, when they applied architectural refactoring. Then, we analyzed if the elements had smells that could indicate architectural problems. We used this analysis to find when smells indicate architectural refactoring opportunities.

We analyzed 52,667 refactored elements. We found that in most cases (59.48% of the refactored elements) smells are likely to indicate architectural refactoring opportunities. To confirm such result, we analyzed 1,168 root-canal refactorings. We were able to find examples of when smells indicate architectural refactoring opportunities that have not been reported in the literature. We were also able to complement and better explain the few examples from the literature. For instance, most related studies agree that *God Class* is an indicator of architectural refactoring for removing an architectural problem [1, 36, 58]. Our results showed that not only *God Class* is an indicator of architectural refactoring but also *Complex Class*. Moreover their likelihood of being indicators increases when they occur with other smells. We also found cases where smells cannot be used as indicators of architectural refactoring opportunities. Finally, we found patterns of smells that can often indicate architectural refactoring for a specific type of architectural problem. Most of these patterns have not been presented elsewhere.

## 2 BACKGROUND AND TERMINOLOGY

We discuss here basic concepts and terminology used in this study.

### 2.1 Smells Indicating Architectural Problems

Code smells are indicators of the occurrence of architectural problems in the source code [9]. For instance, the *Unused Abstraction* problem happens when the code element representing the abstraction is not directly used or is unreachable [3, 20]. Usually, this architectural problem manifests in the source code due to modifications that make code elements obsolete. This problem can be identified by the *Speculative Generality* – a code smell that indicates an element, usually a class, that was created to support anticipated future features that never have been implemented [9].

Other examples of smell types include *Long Method* and *God Class* [9, 18]. Studies based on smells use them as indicators of architectural problems since each smell may be fully or partially associated with an architectural problem [29, 31, 48, 55]. Let us consider the Figure 1 to illustrate how smells can be signs of architectural problems. This figure uses an UML-like notation to show a partial view of the Health Watcher system [13]: a web-based system for improving the quality of services that health vigilance institutions provide. The IFacade is affected by the Fat Interface problem, represented in the figure by the puzzle symbol. This interface declares methods to access three non-cohesive services, represented in the figure by shades of blue.

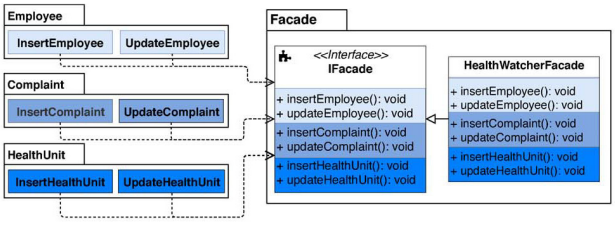


**Figure 1: Example of Architectural Problem**

Classes in Employee, Complaint and HealthUnit packages are clients of the IFacade interface and contain smells such as *Divergent Change*, *Feature Envy* and *Dispersed Coupling*. *Divergent Change* appears in the classes since they change whenever a change is made in one of the three services. *Dispersed Coupling* appears because the classes are all connected through the interface. Furthermore,

some of the methods in these classes may contain *Feature Envy* since they are more interested in other classes than the one related to its own service. Classes that implement IFacade also contain smells. As the interface declares more than one responsibility (*i.e.*, one service), it forces other classes to implement more than one as well. Consequently, classes such as HealthWatcherFacade contain *God Class* and *Feature Envy* smells. *God Class* emerges because the class implements more than one responsibility.

In this scenario, the combination of these smells indicates the architectural problem. Thus, a reasonable assumption is to expect that code refactorings to remove the smells will also remove the architectural problem. Therefore, developers could use the smells as hints to apply architectural refactoring. Unfortunately, as discussed before, some smells do not indicate architectural problems. Thus, developers cannot use them to apply architectural refactoring; after all, there is no architectural problem for developers to remove.

## 2.2 Refactorings for Architectural Problems

Refactoring is a popular technique to remove architectural problems from a system. Refactoring is defined as a program transformation intended at preserving the observable behavior and improving the program structure [9]. Although this definition entails an expectation that refactoring always preserve the system behavior, depending on the tactic applied, this is not always the case [30]. We describe next the tactics and other basic concepts.

**Refactoring Type** indicates the form of the transformation applied to attributes, methods, classes and interfaces. We considered the 13 refactoring types [9]: *Extract Interface*, *Extract Method*, *Extract Superclass*, *Inline Method*, *Move Class*, *Move Field*, *Move Method*, *Pull Up Field*, *Pull Up Method*, *Push Down Field*, *Push Down Method*, *Rename Class*, *Rename Method*. These types comprise the most popular refactoring types [30] and they are directly associated with the removal of architectural problems. For this study, we considered the architectural problems in Table 1. The first column presents the type of architectural problem, and the second one contains a brief definition about it. In the context of this study, we focused on a set of architectural problems that may be indicated by code smells and that may be removed through refactorings. In the example of Figure 1, refactorings such as *Extract Interface*, *Extract Class*, and *Move Method* could be applied to remove the Fat Interface.

**Refactored Elements** comprise all elements that refactorings directly affect. Each refactoring type affects directly different elements [9]. For example, the *Move Method* moves a method *m* from class *A* to *B*. Thus, the refactored elements are: *m*, *A* and *B*. Although other elements can be indirectly affected by the refactoring, we consider only these three elements as refactored ones.

**Refactoring Tactics** indicate the two main tactics that developers follow during refactoring [30]: *root-canal refactoring* and *floss refactoring*. Root-canal refactoring is applied to repair deteriorated code, and it involves a process of exclusively applying structural transformations. As the goal of this tactic is to repair deteriorated code, there is a high chance of this tactic to be applied to remove architectural problems. For instance, to get rid of Misplaced Concern, a developer can apply a *Move Method* and *Move Field* to move the misplaced functionality to the class to which it belongs. Conversely, floss refactoring is applied to achieve another objective that is different from structural improvements, such as adding features or fixing bugs. For example, a developer may need to apply the *Push Down Field* before fixing a bug related to hierarchy.

**Table 1: List of Architectural Problems**

| Architectural Problem | Definition |
|---|---|
| Ambiguous Interface | Architectural problem that happens when a component interface is ambiguous and provides non-cohesive services [10] |
| Cyclic Dependency | Architectural problem that happens when one or more elements depend on each other, creating a cycle [37] |
| Component Overload | Architectural problem that happens when a component is overloaded with responsibilities [23] |
| Concern Overload | Architectural problem that happens when a code element fulfills too many responsibilities [23] |
| Fat Interface | Architectural problem that happens when an interface exposes many funcionalities and many of those functionalities are not related to each other [26] |
| Incomplete Abstraction | Architectural problem that happens when an element does not support a responsibility completely in their enclosing component [35] |
| Misplaced Concern | Architectural problem that happens when an element implements a functionality, which is not the predominant one of their enclosing component [23] |
| Scattered Concern | Architectural problem that happens when elements are responsible for the same functionality, but some of them cross-cut the system [10] |
| Unused Abstraction | Architectural problem that happens when the code element representing the abstraction that is not directly used or is unreachable [3] |
| Unwanted Dependency | Architectural problem that happens when a dependency violates a rule defined on the system architecture [38] |

**Architectural Refactoring** occurs when developers apply refactorings that affect the system architecture. Examples include the introduction and removal of hierarchies, elimination of dependencies, retrofitting a design pattern, moving a large portion of the source code into smaller modules, and the like [19, 61]. For these scenarios, the developers have in mind a desired high-level architecture, in which they have to apply a series of low-level code refactorings to achieve the desired architectural impact [19].

# 3 STUDY DESIGN

Researchers have proposed some solutions to help developers in applying architectural refactoring to remove architectural problems [17, 19, 41, 61]. However, some of them fell short in helping developers identify and apply architectural refactoring. A reason for this is that developers usually do not have access to artifacts that could guide them in identifying architectural refactoring opportunities, such as the architectural documentation. In fact, identifying architectural refactoring opportunities in a system is far from trivial [45] as it requires a broad understanding of the system structure and its functionalities. To help developers with such an identification task, we investigated if code smells can be used to indicate architectural refactoring opportunities to remove architectural problems.

## 3.1 Research Question

As the architectural documentation is usually unavailable, developers have to directly analyze the source code to identify and remove architectural problems through architectural refactoring [45]. In this scenario, we hypothesize that developers can increase their chances of adopting architectural refactoring if they rely on activities that they are familiar with. The first activity is the detection of code smells. According to the literature, not only developers are familiar with code smells [58], but they also use them to identify

architectural problems [45, 46]. Refactoring is the second activity that developers are familiar with. Usually, they rely on the presence of smells to identify low-level code refactorings [9]. Thus, if they already use smells to identify architectural problems and to apply code refactoring, they should be able to benefit from using smells to identify and apply architectural refactoring. As not all smells are related to an architectural problem, we need to investigate when smells are indicators of architectural refactoring opportunities. This investigation is the focus of our research question:

**RQ.** When are smells indicators of architectural refactoring opportunities for developers?

To answer our RQ, we investigate the occurrence of architectural problems that underwent repairing actions by developers. We searched for elements that had been modified by root-canal refactorings, and we verified if they contained smells that could indicate architectural problems. We can focus our analysis on the root-canal refactorings since those are those primarily targeted at improving the system internal structure [9, 30]. Thus, elements refactored during this tactic have a high chance to contain architectural problems. Furthermore, we will be able to better understand the relation among code refactoring, smells, and architectural problems – triple relationship that has not been explored simultaneously in the literature.. As result, we expect to find recurrent cases of when code smells are indicator of architectural refactoring opportunities.

## 3.2 Investigation of Refactored Elements

We rely on actual code refactorings to find the refactored elements that had architectural problems. However, we cannot assume that all refactored elements have architectural problems. Hence, we focused on analyzing root-canal refactorings. Root-canal refactoring is a tactic primarily applied to repair deteriorated code [9, 30], which can be caused by the presence of architectural problems. Thus, elements refactored under this tactic have a high chance to contain architectural problems. After finding these refactored elements, we categorized them according to the number of smells:

- **Smell-free category**: This category encompasses refactored elements that are NOT affected by smells.
- **Single smell category**: This category comprises refactored elements affected by only ONE smell.
- **Multiple smells category**: This category includes refactored elements affected by more than one smell.

We highlight that we are interested in architectural problems that may be so harmful to the point of forcing developers to apply architectural refactoring. This is the reason why we analyzed elements that developers focused their effort during root-canal refactoring. They are the ones that may contain architectural problems that can lead to the system redesign [12, 21, 43, 54].

# 4 DATA COLLECTION AND ANALYSIS

This section presents the phases followed to answer our RQ. The replication package [34] contains the algorithm used (i) to identify the refactored elements, (ii) to collect code smells, and (iii) to categorize the refactored elements according to the number of smells. It also offers the entire dataset with the refactorings and smells collected for the 50 GitHub projects.

## 4.1 Phase 1: Selection of Software Projects

The first phase consisted of choosing open source projects. We focused on open source projects to allow the study replication. As GitHub is the world's largest open source community, we established it to be the source of software projects, in which we selected projects that matched the following criteria:

- Projects that have been evaluated with different levels of popularity. As GitHub star is a metric to keep track of how popular a project is among GitHub users, we used it to select projects with different popularity levels;
- Projects with an active issue tracking system.
- Projects with at least 90% of code written in Java language.

These criteria allowed us to select 50 software projects that are active and has been used in diverse contexts by the software community. We focused on Java projects because Java is a very popular programming language, and it was also targeted by related studies [2, 4]. Furthermore, we also selected projects in Java due to the availability of tools to identify refactorings [51] and code smells [32]. Thus, we selected Java projects with a diversity of structure, size and popularity. The replication package [34] contains their detailed information.

## 4.2 Phase 2: Code Refactoring Detection

The second phase consisted of detecting code refactorings for all selected projects. We chose Refactoring Miner [51, 52] (version 0.2.0) as the tool to detect code refactorings. This tool implements a lightweight version of UMLDiff algorithm [57] for differencing object-oriented models. When the tool is applied between two versions, it returns the elements that changed from one version to the other. It also returns the refactoring type associated to the change. The reported precision of 98% [40, 52] led to a very low rate of false positives, as confirmed in our validation phase. The tool detects the 13 refactoring types used in our study (Section 2.2).

After the code refactoring detection, we divided the refactorings according to the applied tactic. We computed the number of refactorings belonged to the floss or root-canal refactoring based on the output of the Refactoring Miner and the eGit plugin (http://www.eclipse.org/egit/). We ran the eGit plugin to provide us the changes within files between commits. Then, we used the UNIX diff tool to analyze all the changes in the classes modified by the refactorings. A code refactoring is considered root-canal if (and only if) no other non-refactoring operation occurs in the same change (*i.e.*, same commit). For instance, we do not consider as root-canal refactoring a *Move Method* detected in a commit together with one or more other file modifications that were not detected as code refactorings. We performed this detection by crossing the list of changes of each file in each commit with the changes identified as code refactorings by the Refactoring Miner tool. From the total of 51,227 refactorings, 76.56% were classified as floss refactoring and 23.44% were classified as root-canal. This distribution is similar to the one reported in a previous study [30].

## 4.3 Phase 3: Code Smell Detection

For this study, we decided to detect code smells with metrics-based strategies [18, 25]. Thus, we can compare our results with related

studies that used the same detection strategies [2, 4]. These strategies are based on a set of detection rules [2, 18] that compare metric values with predefined thresholds according to logical operators. Since we did not find any publicly available tool that implements these detection rules, we developed the Organic tool [32]. Our tool detects the following smells: *Brain Class*, *Brain Method*, *Class Data Should Be Private*, *Complex Class*, *Data Class*, *Dispersed Coupling*, *Feature Envy*, *God Class*, *Intensive Coupling*, *Lazy Class*, *Long Method*, *Long Parameter List*, *Message Chain*, *Refused Bequest*, *Shotgun Surgery*, *Spaghetti Code*, *Speculative Generality*. Metrics, rules and thresholds for the 17 code smells implemented in our tool are the ones used in the literature [2, 18] and are all available in our replication package [34]. These smells are closely related to architectural problems [9, 22, 24, 28, 29, 31, 55].

### 4.4 Phase 4: Manual Validation of Refactorings

The last phase comprises the validation of the code refactorings. Even though Refactoring Miner achieved a precision of 98% [52], we were not sure if it would achieve the same precision in our set of software systems. For the validation, we conducted two inspections.

The first inspection was to validate each one of the 13 refactoring types (Section 2.2). For this inspection, we randomly sampled refactorings of each type since the precision of the Refactoring Miner could vary from one type to another. Such variation is due to the rules implemented in the tool to detect each refactoring type. To deliver an acceptable confidence level to the results, we calculated the sample size of each refactoring type based on a confidence level of 95% and a confidence interval of 5 points. For this inspection, we recruited ten students to validate the samples manually. The manual inspection started by presenting to the students a pair of versions of elements marked as refactored by Refactoring Miner. For each pair of elements, the student had to mark it as a valid refactoring or not. We highlight that our goal was to ensure the trustability of the tool for our set of software systems. This is the reason why we relied on the students' inspection, who were familiar with refactoring. In general, we observed a high precision for each refactoring type, with a median of 88.36%. The precision found in all refactoring types is within one standard deviation (7.73). Applying the Grubb outlier test (alpha=0.05), we could not find any outlier, indicating that no refactoring type is strongly influencing the median precision found. Thus, the results found in the representative sample represents a key factor to provide trustability to our results.

The second manual inspection was to validate the classification of the refactorings into tactics. We conducted three steps. First, we used Eclipse and the eGit plugin to classify a refactoring as root-canal refactoring or floss refactoring. Second, we used the UNIX diff tool to analyze all the changes in the classes modified by the refactoring operations. Third, we analyzed the tool output, searching for a behavioral change. When finding one, we filled a form explaining it, and we classified the change as floss refactoring. When we did not find a behavioral change, we classified it as root canal refactoring. This second validation was conducted by three researchers from our group given their expertise in refactoring. Each code refactoring was manually classified by two researchers. In a case of a disagreement, the third researcher stepped in. As result, we classified 4,991 refactorings into root-canal and floss refactoring. We found that developers apply root-canal refactoring in 31.5% of the cases. The confidence level for this number is 95% with a confidence interval of 5%. For this inspection, we did not use students; instead, we (paper's authors) relied on our expertise to conduct the inspection. We decided to conduct this inspection among us for a couple of reasons. First, we needed people who had experience with refactoring before. To determine that there was a behavior change due to a refactoring, one needs to have knowledge that students may not have. Second, this second inspection requires more effort and time than to validate the refactoring types.

## 5 RESULTS

We discuss here the results to our RQ. First, we analyzed whether the refactored elements are associated with the absence or presence of smells (Section 5.1). Second, we investigate when smells can indicate architectural refactoring opportunities in general (Section 5.2). Then, we further discuss different examples of when smells are (or are not) indicators of architectural refactoring opportunities (Section 5.3). Finally, we used these examples to find recurring types of smells that may indicate architectural refactoring opportunities for specific types of architectural problems (Section 5.4).

### 5.1 Categorization of Refactored Elements

Table 2 provides a summary of the relationships between the refatorings, and the absence or presence of smells. The table presents the total number of refactorings ($1^{st} column$), while the next columns show how often the refactored elements contain none smells, ($2^{nd} column$), only a single smell ($3^{rd} column$) or multiple smells ($4^{th} column$).

**Table 2: Categorization of Refactored Elements**

| Refactorings | Smell-free Category | Single Smell Category | Multiple Smells Category |
|---|---|---|---|
| 51,227 | 10,512 (20.52%) | 16,443 (32.10%) | 24,272 (47.38%) |

**Most refactored elements are smelly.** If we have found that most refactorings belong to the *smell-free* category, we could conclude that smells are often no indicators of architectural refactoring. However, we found exactly the opposite result. Only 20.52% of the code refactorings (10,512) were applied to elements without smells. From the 51,227 code refactorings, 40,715 (79.48%) were applied to elements with at least one smell. From this result, 47.38% were applied to 24,953 elements with more than one smell, and 32.10% were applied to 16,906 elements with one smell.

One could argue that most code refactorings were applied to smelly elements because most elements contain smells. In other words, the refactored elements contain smells because their software system has a high rate of smells, thereby increasing the likelihood that code refactorings are applied to smelly elements. To verify if most elements contain smells, we computed the probability of randomly choosing a smelly element in our dataset ($|smelly\ elements|/|all\ elements|$), which is 0.3%. This result shows that, in our dataset, code refactorings are not applied to smelly elements by coincidence because only 0.3% of the elements contain smells.

## 5.2 Analysis of Root-canal Refactoring

Table 3 presents a summary of all collected code refactorings. It is structured in terms of two major columns. They indicate the two samples we analyzed. The first one provides information about all collected code refactorings. The second column contains details about the code refactorings manually classified as root-canal. The rows present the refactoring types ($3^{rd}$ *subcolumn*), which are divided according to their category ($1^{st}$ *subcolumn*) and the kind of activity that they represent to the program ($2^{nd}$ *subcolumn*). The category ($1^{st}$ *subcolumn*) indicates if a refactoring affects the architectural structure of: (i) elements within a class hierarchy (*Within Hierarchy*), or (ii) elements across multiple hierarchies (*Across Hierarchies*). The activities ($2^{nd}$ *subcolumn*) indicate the mechanism behind each refactoring type: (i) extraction of statements of a method, (i) moving up or down methods and attribute within a hierarchy, (iii) moving members, or (iv) restructuring modules across hierarchies.

**Root-canal refactoring is most applied to smelly elements.** As not all refactored elements may contain architectural problems, we analyzed elements refactored through root-canal refactoring [30]. The $7^{th}$ *subcolumn* of Table III shows the 1,168 refactorings manually validated as root canal (Section 4.4). We classified these root-canal refactorings according to our three categories: *smell-free*, *single smell* and *multiple smells* (*subcolumns 8-10*). This classification shows that most code refactorings (88.53%) were applied to smelly elements, either with a single smell (21.23%) or with multiple smells (67.30%). As likelihood of these elements to contain architectural problems is high, this result indicates that developers tend to prioritize architectural problems that manifest as multiple smells in the source code rather than isolated smells. We only found a low percentage (11.47%) of root-canal refactorings (8th sub-column) affecting smell-free program elements (*smell-free category*). We noticed that these code refactorings are those often involving: (i) very simple classes being moved (but without internal smells on them) across packages, (ii) one or more renames, or (iii) method inlining.

**Smells as Indicators of architectural refactoring opportunities.** When we analyzed only the smelly elements, most code refactorings were applied to elements with multiple smells (67.30%). This number is higher than the overall code refactorings applied to the *multiple smells category* (47.38%) shown in Table 2. In summary, when developers focus on repairing deteriorated code, most code refactorings are applied to elements that contain multiple smells. As these code refactorings occurred in the context of root-canal refactoring, these elements are very likely to contain architectural problems. Thus, we can consider that smells are likely to indicate architectural refactoring to remove architectural problems. This result is summarized in our first finding:

> **Finding 1**: In general, code smells can be used to indicate architectural refactoring to remove architectural problems when developers intend to repair their deteriorated code.

## 5.3 Relating Code Refactoring, Smells and Architectural Problems

When we analyze each refactoring type in Table 3, we noticed that most types are applied to elements with multiple smells. Only five refactoring types did not follow this distribution (gray rows). *Extract Interface*, *Extract Superclass*, and *Rename Method* were frequently applied to single smell elements in 51.85%, 62.16% and 55.13% of the cases, respectively. On the other hand, *Move Class* was frequently applied to elements without smells (51.66%). These four types provide us with interesting discussions. The *Rename Class* does not provide discussions due to its low number of operations.

**Across hierarchy refactorings are often applied to single smells.** *Extract Interface*, *Extract Superclass* and *Move Class* belong to the *Across Hierarchy* category. These refactoring types comprise the activity of moving (part of) modules across different hierarchies. *Extract Interface* and *Extract Superclass* are usually architectural refactoring. For instance, they can be used to solve architectural problems such as Ambiguous Interface, Cyclic Dependency, Incomplete Abstraction, and Unused Abstraction. These problems are related to an inappropriate architectural decision that a stakeholder made when designing the abstraction for code elements. Some of these architectural problems can be identified with only a single smell. Thus, it is not surprising that most of *Extract Interface* and *Extract Superclass* were applied to the *single smell* category.

**Extract Interface refactorings applied to Lazy Classes.** After analyzing the smells touched by *Extract Interface*, we found that most them (46%) were *Lazy Class*. At first, there seems to be no logical relationship of *Extract Interface* with *Lazy Class*. However, some *Lazy Class* occurred in interfaces and in abstract classes. In these cases, *Extract Interface* is considered as an architectural refactoring. In fact, we found that this architectural refactoring was applied to meet the Interface Segregation Principle (ISP) [26]. This principle is often followed to remove Ambiguous Interface and Fat Interface. Nevertheless, *Lazy Class* alone does not suffice to indicate architectural refactoring opportunities. The reason is that even well designed abstractions may be affected by this smell type.

**Extract Superclass applied in complex implementations.** Analyzing the smells touched by *Extract Superclass*, we observed that 37% were either *Complex Class* or *God Class*. *Extract Superclass* applied to classes with these two smells were often related to the implementation of architectural patterns such as Strategy and Command. Such refactorings are directly related to removing architectural problems, which make them architectural refactorings. For example, we observed *Extract Superclass* being applied to create super classes in hierarchies related to parsing (Elasticsearch) and command/request handling (Apache Coyote and Spring Framework). *Complex Class* and *God Class* are often associated with different types of architectural problem [1, 36, 58]. However, we found that the presence of any of them alone is usually not sufficient to indicate a architectural problem. We will discuss how such smells may be indicators of architectural refactoring opportunities when they are combined with other smells (Section 5.4).

**Extract Superclass applied in lazy implementations.** We found a high proportion (35%) of *Lazy Class* and *Data Class*. These smells were mostly observed when *Extract Superclass* aimed at improving polymorphism. For instance, we observed an *Extract Superclass* refactoring in the ArgoUML system, which involved 53 *Lazy Classes* that represent elements of the UML metamodel. Such code refactoring introduced a new abstraction that helped to create more reusable implementations. Even though the refactorings improved the reusability, most classes had no architectural problems. The examples of *Extract Superclass* led us to our second finding:

**Table 3: Categorization of Refactorings According to Root-canal Refactoring**

| All Collected Refactorings | | | | | | Validated Root-canal Refactorings | | | |
|---|---|---|---|---|---|---|---|---|---|
| **Category** | **Activity** | **Refactorting Type** | **Total** | **Floss Refactoring** | **Root-canal Refactoring** | **Operations** | **Smell-free** | **Single Smell** | **Multiple Smells** |
| Within Hierarchy | Extraction | Extract Method | 15,629 | 12,084 (77.32%) | 3,545 (22.68%) | 345 | 3.77% | 15.94% | 80.29% |
| | | Inline Method | 4,979 | 4,589 (92.17%) | 390 (7.83%) | 38 | 15.59% | 13.16% | 71.05% |
| | Up/Down Moves | Pull Up Method | 4,610 | 2,585 (56.07%) | 2,025 (43.93%) | 197 | 3.05% | 3.05% | 93.91% |
| | | Pull Up Attribute | 3,495 | 2,097 (60.00%) | 1,398 (40.00%) | 136 | 2.94% | 25.00% | 72.06% |
| | | Pull Down Method | 575 | 482 (83.83%) | 93 (16.17%) | 9 | 0% | 44.44% | 55.56% |
| | | Pull Down Attribute | 246 | 133 (54.07%) | 113 (45.93%) | 11 | 9.09% | 36.36% | 54.55% |
| Across Hierarchies | Member Moves | Move Method | 5,000 | 3,900 (78.00%) | 1,100 (22.00%) | 107 | 2.80% | 21.50% | 75.70% |
| | | Move Attribute | 5,134 | 4,826 (94.00%) | 308 (6.00%) | 30 | 10.00% | 33.33% | 56.67% |
| | Module Restructuring | Extract Inteface | 614 | 336 (54.72%) | 278 (45.28%) | 27 | 7.41% | 51.85% | 40.74% |
| | | Extract Superclass | 2,540 | 2,159 (85.00%) | 381 (15.00%) | 37 | 8.11% | 62.16% | 29.73% |
| | | Move Class | 2,042 | 490 (24.00%) | 1,552 (76.00%) | 151 | 51.66% | 17.22% | 31.13% |
| Renaming | | Rename Class | 1,350 | 1,329 (98.44%) | 21 (1.56%) | 2 | 0% | 50.00% | 50.00% |
| | | Rename Method | 5,013 | 4,211 (84.00%) | 802 (16.00%) | 78 | 19.23% | 55.13% | 25.64% |
| **Total** | | | **51,227** | **39,221 (76.56%)** | **12,006 (23.44%)** | **1,168** | **11.47%** | **21.23%** | **67.30%** |

**Finding 2**: *Complex Class* and *God Class* usually indicate architectural refactoring opportunities when they occur together with other smells while *Lazy Class* and *Data Class* are not.

**Code smells are not able to indicate the need for Move Class.** Interesting enough, *Move Class* was frequently applied to the *smell-free* category. However, *Move Class* is a refactoring that is closely related to architectural problems. This architectural refactoring can be applied to remove architectural problems such as Scattered Concern and Component Overload. As we will discuss in Section 5.4, *Scattered Concerns* are often indicated by smells such as *Dispersed Coupling*, *Divergent Change*, *Feature Envy*, *God Class*, *Intensive Coupling*, and *Shotgun Surgery*. However, by analyzing the *Move Class* refactorings, we never observed such smells in the moved classes. The reason is that a *Move Class* refactoring is unable to remove architectural problems related to most of these smells. *Move Class* is adequate to tackle architectural problems related to high coupling and low cohesion at the component level. Smells such as *God Class* and *Intensive Coupling* are best suited to indicate these problems at the code element level, which are often removed by other refactoring types, such as *Extract Class* and *Move Method*.

To verify our reasoning above, two researchers manually investigated the *Move Class* refactorings applied to smelly elements. We found that most of the smells touched by *Move Class* refactorings were either *Data Class* (21%) or *Lazy Class* (46%). Classes affected by those smells were mostly Data Transfer Objects (DTOs) and persistence entities. We also found many cases of smells such as *Complex Class* (12%), *Speculative Generality* (6%), and *Spaghetti Code* (4%) touched by *Move Classes*. However, those smells were never removed by the *Move Class* refactorings. Thus, we conjecture that many of these code refactorings were applied to organize classes into sub-packages that best reflect the system domain. Nevertheless, this intention has nothing to do with the detected smells. The analysis of the *Move Class* led us to our third finding:

**Finding 3**: Smells are not indicator of the *Move Class* architectural refactoring.

**Rename Method as a complementary refactoring.** Both rename refactoring types did not follow the distribution of most refactorings types. In the case of *Rename Class*, we cannot say much since there were only two refactorings. On the other hand, we can discuss the *Rename Method*, which it was most applied to elements with only one smell. This refactoring type is not exactly one directly related to an architectural problem. It is most likely that is applied as part of a series of code refactorings. Thus, the *Rename Method* can be applied in methods that contain a smell that points out that the method name no longer makes sense. For instance, 61% of smells touched by *Rename Method* were *Feature Envy*. This smell indicates that part of the method is more interested in another class; thus, the method may be implementing two functionalities. A developer can apply refactoring to move parts of the method to another class. If s/he does it, the method may have to change the name to be consistent with the functionality left behind. Then, s/he applies the *Rename Method* before moving part of the functionality.

## 5.4 Analysis of Smell Patterns

Our previous analysis of refactored elements led to various discussions on when smells are indicators of architectural refactoring opportunities. We manually investigated 189 instances to support these discussions. We used them to find the refactorings to remove each type of architectural problem. Based on this analysis and relying on the literature [9, 22, 24, 28, 29, 31, 55], we searched for patterns of smells and architectural problems. As result, we associated some types of smells with some types of architectural problems. For instance, we noticed that usually when *Dispersed Coupling*, *Feature Envy*, and *Long Method* appeared together in an element, there was a high likelihood that the element had the Misplaced Concern problem. Based on our analysis, we defined a set of smell patterns, which are shown in Table 4. A **smell pattern** represents one or more types of smells ($2^{nd}$ *column*) that are likely to indicate an architectural problem ($1^{st}$ *column*) if they appear in the code elements. Consequently, if these smells appear together, then they indicate an architectural refactoring opportunity. Thus, the developer should use the recommended refactoring (Table 5) to get rid of the smell, and, therefore, remove the architectural problem.

**Table 4: Smell Patterns to Indicate Architectural Problems**

| Architectural Problem | Code Smells |
|---|---|
| Ambiguous Interface | (Long Method and Feature Envy in the interface and Dispersed Coupling in elements that are clients or implement the interface) |
| Cyclic Dependency | (Intensive Coupling and Shotgun Surgery) |
| Component Overload | (Shotgun Surgery, Divergent Change, Feature Envy, God Class/Complex Class, Intensive Coupling, and Long Method) |
| Concern Overload | (Complex Class, Divergent Change, Feature Envy, God Class/Complex Class, Intensive Coupling, Long Method, and Shotgun Surgery) |
| Fat Interface | (Shotgun Surgery in the interface) or (Divergent Change, Dispersed Coupling, and Feature Envy in elements that are clients or implement the interface |
| Incomplete Abstraction | (Lazy Class) |
| Misplaced Concern | (God Class/Complex Class or Dispersed Coupling, Feature Envy, and Long Method) |
| Scattered Concern | (Dispersed Coupling, Divergent Change, Feature Envy, God Class/Complex Class, Intensive Coupling, and Shotgun Surgery) |
| Unused Abstraction | (Speculative Generality) |
| Unwanted Dependency | (Feature Envy, Long Method, and Shotgun Surgery) |

**Table 5: Recommended Refactoring for the Code Smell**

| Code Smell | Common Refactorings |
|---|---|
| Complex Class | Extract Method, Move Method, Extract Class [2] |
| Dispersed Coupling | Extract Method |
| Divergent Change | Extract Class [9] |
| Feature Envy | Move Method, Move Field, Extract Field [9] |
| God Class | Extract Class, Move Method, Move Field [2] |
| Invensive Coupling | Move Method, Extract Method |
| Lazy Class | Inline Class, Collapse Hierarchy [9] |
| Long Method | Extract Method [9] |
| Shotgun Surgery | Move Method, Move Field, Inline Class [9] |
| Speculative Generality | Collapse Hierarchy, Inline Class, Remove Parameter, Rename Method [9] |

**Patterns with one and multiple smells.** According to these patterns, some architectural problems that can be identified by only a single smell, such as Incomplete Abstraction and Unused Abstraction, while there are other architectural problems that are most suitable to be identified by multiple code smells, such as Ambiguous Interface, Cyclic Dependency, Scattered Concern and Unwanted Dependency. Finally, there are architectural problems that can be identified by a single or multiple smells, such as Concern Overload and Fat Interface. Table 6 shows the number of architectural problems that we found in our database according to these patterns.

**Table 6: Architectural Problems in Refactored Elements**

| Architectural Problems | Single Smell | Multiple Code Smells |
|---|---|---|
| Ambiguous Interface | N/A | 0 |
| Ciclic Dependency | N/A | 31 |
| Concern Overload | 490 | 514 |
| Fat Interface | 0 | 2 |
| Incomplete Abstraction | 12,503 | N/A |
| Scattered Concern | N/A | 26 |
| Unused Abstraction | 2,317 | N/A |
| Unwanted Dependency | N/A | 11 |
| N/A = Not Applicable | | |

**Smells that indicate architectural refactoring for Concern Overload.** The two architectural problems in gray rows provide us with interesting discussions. We found 1,004 classes that could have Concern Overload architectural problem according to the smell patterns. This problem can be identified with the pattern *Complex Class/God Class* (490 instances) or with multiple smells: *Complex Class*, *Divergent Change*, *Feature Envy*, *God Class*, *Intensive Coupling*, *Long Method*, and *Shotgun Surgery* (514 instances). We randomly sampled 100 classes: half with the *Complex Class/God Class* pattern and the other half with the pattern with multiple smells. Our manual validation shows that the pattern with multiple smells is most likely to indicate architectural refactoring to remove the problem. From the classes that had only *Complex Class* or *God Class*, 26% of them had the Concern Overload (13 classes). From the classes that had the pattern with multiple smells, 64% had the architectural problem (32 classes).

We also investigated the relation of the Concern Overload with *Move Method* and *Move Attribute*. We selected these refactoring types because they are often architectural refactorings applied for removing occurrences of Concern Overload. We found that 69% of *Move Methods* and 30% of *Move Attributes* occurred in elements affected by *Complex Class*, *God Class* or by their combination. The high percentage of *Move Methods* and *Move Attributes* that touch the *Complex Class/God Class* patterns is another evidence that such patterns are indicators of architectural refactoring opportunities.

**Smells that indicate architectural refactoring for Fat Interface.** *Fat Interface* is another architectural problem that can be identified through one or multiple smells. However, we did not find any interface in the systems that had the pattern with one smell (*Shotgun Surgery*). On the other hand, we found 2 instances of *Fat Interface* when we searched for elements that had the pattern with multiple smells (*Divergent Change*, *Dispersed Coupling*, and *Feature Envy*). Even though there were only two instances of *Fat Interface*, these are two more cases which show that the pattern with multiple smells is more likely to indicate an architectural refactoring for architectural problem than the pattern with only a single smell.

**Architectural Refactoring of latent architectural problems.** According to the smell patterns, to identify Ambiguous Interface architectural problem, a developer needs to find code elements that use the interface and contain *Long Method*, *Feature Envy*, and *Dispersed Coupling*. We did not find elements that are connected to the interface and contain this smell pattern. However, when we considered only *Long Method* and *Feature Envy*, then the number increases to 130 possible instances of architectural problems. This result is interesting to show that these elements were presenting the first signs of an architectural problem. Even if the developers' intention was not to remove an architectural problem, this result indicates that they refactored elements that had a sign of a latent architectural problem, and these elements had code smells that could indicate a (potential) architectural problem.

**Using multiple smells.** The results about *Concern Overload*, *Fat Interface*, and *Ambiguous Interface* are useful to discuss to what extent smells help developers to find architectural refactoring opportunities. According to the validation of these 102 instances of architectural problems, relying on a single smell may not be enough to identify architectural refactoring for some architectural problems. Our data suggest that if developers rely on smell patterns with

multiple smells, they have a higher chance of identifying architectural problems, which is consistent with results from other studies [1, 23, 31, 55, 59]. This observation leads to our fourth finding:

> **Finding 4**: Patterns with multiple smells are more likely to indicate architectural refactoring opportunities for Concern Overload, Fat Interface, and Ambiguous Interface than patterns with only one smell.

There are some pros and cons in using these smell patterns. Analysis of various smells in some of these patterns may increase developers' confidence on applying architectural refactorings. On the other hand, to reason about multiple smells simultaneously may be a hard task. Additionally, we highlight that developers cannot identify an architectural refactoring opportunity if they expect to find all instances of the smells within a pattern. For example, a developer would not identify any architectural refactoring to remove *Ambiguous Interface* in our dataset if s/he was expecting to find instances of *Long Method*, *Feature Envy*, and *Dispersed Coupling*. As mentioned, we did not find elements connected to the interface with all the smells in this pattern. Thus, developers cannot expect to find all the smells of a pattern to identify an architectural refactoring opportunity, especially if the affected elements present the first signs of a problem. When we analyzed the relation of Ambiguous Interface with *Extract Method*, for example, we observed 79% of architectural refactorings occurring in elements affected either by *Long Method* or *Feature Envy*. However, if we consider only when both smells appear together, this percentage drops to 22%.

## 6 THREATS TO VALIDITY

We focused on refactored elements to identify elements with architectural problems. Regarding this threat, we are aware that smells and architectural problems can appear in non-refactored elements; thus, we are missing these elements. Consequently, we could (erroneously) conclude that smells are not indicators of architectural refactoring opportunities. Nevertheless, we found that code refactorings are not applied to smelly elements by coincidence. The chance of randomly choosing a smelly element in our dataset is only 0.3%. Thus, refactoring operations indeed tend to concentrate on smelly elements. Therefore, the use of smells to identify architectural refactoring was appropriate. Additionally, we highlight that we are interested in refactored elements because they are elements which developers focused their effort on. Hence, these elements may contain architectural problems that were important enough to be architecturally refactored. Additionally, we focused our analysis on root-canal refactoring since elements refactored during this tactic have the highest chance to contain architectural problems.

We had to manually validate a set of refactoring types. We relied on the students to perform this validation, which represents a threat. In order to mitigate this threat, each refactoring instance was validated by at least two students. In case there was a divergence, one of the authors worked as third reviewer. In the second inspection, we carefully characterized the refactoring as root-canal or floss refactoring, which is another threat to internal validity. Notice that such analysis is limited to two versions of the source code directly impacted by the refactoring, *i.e.*, not considering all versions in the repository. Moreover, the manual analysis only considers the constraint of behavior preservation in the elements that were actually affected by the refactoring transformations.

The smell detection rules and the thresholds can be a threat. Thus, the results are sensitive to code smell detection rules. As mentioned before, such rules are based on thresholds. The risk is that different thresholds can lead to completely distinct results. We decided to not validate the detected smells because our investigation was focused on automatically detected smells. Thus, a manual validation of smells would not contribute to our goal. To mitigate this threat, we used rules and thresholds previously validated by other researchers [2, 18, 31].

We assume that the elements that had a root-canal refactoring have a high chance of containing architectural problems. This may pose a threat since developers may apply root-canal refactoring and even so, they do not remove architectural problems. However, we focused our analysis on root-canal refactoring because this is the tactic that developers intentionally focus their effort in repairing the structural quality. Indeed, a study with 328 engineers, where 83% of them were developers, shown that developers tend to create branches to apply refactorings exclusively [16], i.e., these branches mainly contain root-canal refactoring. Therefore, the chance of developers target architectural problem during root-canal is higher than when developers apply floss refactoring. To support this assumption, we manually validated all the examples discussed throughout the paper.

We selected a set of 50 software projects to analyze. Thus, the representativeness of these projects is a threat. We mitigate it by establishing a systematic process to select projects. As a result, we obtained relevant Java projects with a diversity of structure and size metrics.

## 7 RELATED WORK

We discuss some studies that are closely related to our study.

### 7.1 Code Smells and Architectural Problems

Code smells have been a well-researched topic over the last decade [22, 24, 28, 29, 31, 33, 53, 55]. For example, Tufano *et al.* [53] investigated when developers introduce code smells in their software projects, and under what circumstances the introduction occurs. First, they mined over 0.5M commits, and then they manually analyzed 9,164 commits to identify when the smells were introduced. Among the results, they found that refactorings can also introduce smells. In fact, they found that smells are introduced in the system as consequence of maintenance and evolution activities.

Some studies investigated the developers' perception of code smells [36, 53, 58]. Yamashita and Moonen conducted an exploratory survey with developers about their knowledge and concern with code smells [58]. The authors investigated through developers' perspective if code smells should be considered meaningful conceptualization of architectural problems. They applied a survey with 73 software developers. Based on the survey answers, they were able to identify the smells that developers perceived as critical, why they are critical, and what features a smell detection tool should have. The results indicated that only 18% of respondents (13 developers) had a good understanding of code smells. However, the

majority of the developers (19 out of 50 developers who finished the survey) are concerned about smells in their source code, while 14% (7 developers) were extremely concerned. *Duplicated Code*, *God Class*, and *Long Methods* were the smells perceived as critical.

Palomba *et al.* analyzed if smells are perceived as problems [36]. They validated 12 different smells in three open source projects. Next, they showed developers code snippets affected and not affected by these smells. Then, developers answered if they considered the code snippets as actual problems. If so, they asked developers to explain what type of problems they perceived. They reported that most code smells are, in general, not perceived by developers as actual problems. However, there are some code smells (*Complex Class*, *God Class*, *Long Method*, and *Spaghetti Code*) that developers immediately perceived as problems for the source code structure.

Even though, these studies provided few examples of smells as indicators of architectural problems. Our results are consistent with these studies. We showed that God Class and Complex Class can indicate architectural problems, especially if they occur with other smell types. However, we further explain when these smells are related to the architectural problems. Different from these studies, our goal is to investigate if code smells can be used as indicators of architectural refactoring opportunities. Yamashita and Moonen focused on investigating to what extent developers had a theoretical knowledge of code smells; while Palomba *et al.* investigated if developers perceive smells as actual problems.

## 7.2 Identification of Architectural Problems

Studies have investigated approaches to identify architectural problems [27, 56]. Mo *et al.* [27] proposed a suite of hotspot patterns: recurring architectural problems that lead to high maintenance cost. They showed that these patterns might be the causes of bug-proneness and change-proneness. Xiao *et al.* [56] introduced an approach to identify and quantify architectural problems. The approach uses four patterns to show the correlation between architectural problems and the decrease of software quality.

Another study focused on investigating the identification of architectural problems from the perspective of developers [45]. Sousa *et al.* [45] proposed a theory to describe how architectural problem identification happens in practice. Their theory explains factors that influence developers during the identification of architectural problems. These studies focused on observing developers identifying architectural problems, showing that developers rely on smells and other indicators to identify these problems. However, they did not investigate when smells can be used to indicate architectural refactoring opportunities. In fact, our results indicate that there are some cases that smells cannot indicate any architectural refactoring to remove architectural problems.

## 7.3 Investigation of Architectural Refactoring

Few studies have investigated architectural refactorings [17, 19, 39, 41, 61]. For example, Kumar and Kumar [17] reported an architectural refactoring of a payment integration platform of a corporate banking organization. In this industrial experience report, the authors presented the key drives that motivated the refactoring, including architectural problems such as Concern Overload. As a result, the refactorings led to significant improvement in application stability and throughput. Lin *et al.* [19] proposed an approach to guide developers in applying architectural refactorings. In their approach, the developers indicate the target architectural design, and then the approach suggests stepwise code refactorings that will change the source code to meet the target architecture.

Zimmermann [61] positioned architectural refactoring as a task-centric technique for restructuring an existing architecture, allowing him to focus on design rationale and related tasks instead. He then "introduced a quality story template that identifies potential architectural smells and an architectural refactoring template that lists the architectural decisions to be revisited as well as the design and development tasks to be conducted when an architectural refactoring is applied." Rizzi, Fontana and Roveda [41] proposed a tool prototype to remove the Cyclic Dependency problem. Their tool suggests the refactorings steps that the developer should follow to remove the Cilic Dependency. Recently, Rachow [39] proposed a research idea to develop a framework that (i) detects an architectural problem, (ii) selects and prioritizes code refactorings, and (iii) shows to developers the impact on the architecture. Although his framework has not been implemented, the author indicates seven architectural problems that the framework will provide support.

# 8 CONCLUSION

We conducted a study with 50 software projects to investigate when smells are indicators of architectural refactoring opportunities. We found that most refactored elements have at least one smell that can indicate an architectural refactoring opportunity. From 52,667 code refactorings, 79.48% were applied to elements with at least one smell while 47.38% were applied to elements with multiple smells.

Analyzing the root-canal refactorings, we found that developers have a higher chance to identify an architectural refactoring to remove an architectural problem when it is related to multiple smells. As a result of our analyses, we came out with a set of smell patterns that are very likely to indicate architectural refactoring opportunities. We are the first one to catalog these patterns.

We identified important implications for researchers and practitioners. For example, we found that *God Class*, *Complex Class*, *Lazy Class* and *Data Class* appear in classes that developers applied *Extract Superclass*. However, only *God Class* and *Complex Class* are indicators of architectural refactoring opportunities. Tool builders can take advantage of this information to prioritize the recommendation of architectural refactorings according to smell types. The patterns with multiple smells can be used by developers to identify and refactor architectural problems such as Concern Overload.

As future work, we intend to investigate the smell patterns. Our goal is to explore them to help developers in automatically identifying architectural refactoring opportunities and to guide them to apply code refactorings to remove an architectural problem.


# ACKNOWLEDGMENT

We want to thank the reviewers for their valuable suggestions. This work is funded by CNPq (grants 434969/2018-4, 312149/2016-6), CAPES (grant 175956), and FAPERJ (grant 22520-7/2016).

# REFERENCES


[1] M Abbes, F Khomh, Y Gueheneuc, and G Antoniol. 2011. An Empirical Study of the Impact of Two Antipatterns, Blob and Spaghetti Code, on Program Comprehension. In *Proceedings of the 15th European Software Engineering Conference; Oldenburg, Germany*. 181–190.

[2] Gabriele Bavota, Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, and Fabio Palomba. 2015. An experimental investigation on the innate relationship between quality and refactoring. *Journal of Systems and Software* 107 (2015), 1 – 14. https://doi.org/10.1016/j.jss.2015.05.024

[3] Timothy A. Budd. 2001. *An Introduction to Object-Oriented Programming* (3rd ed.). Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[4] Diego Cedrim, Alessandro Garcia, Melina Mongiovi, Rohit Gheyi, Leonardo Sousa, Rafael de Mello, Baldoino Fonseca, Márcio Ribeiro, and Alexander Chávez. 2017. Understanding the Impact of Refactoring on Smells: A Longitudinal Study of 23 Software Projects. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*. ACM, New York, NY, USA, 465–475. https://doi.org/10.1145/3106237.3106259

[5] Alexander Chávez, Isabella Ferreira, Eduardo Fernandes, Diego Cedrim, and Alessandro Garcia. 2017. How Does Refactoring Affect Internal Quality Attributes?: A Multi-project Study. In *Proceedings of the 31st Brazilian Symposium on Software Engineering (SBES'17)*. ACM, New York, NY, USA, 74–83. https://doi.org/10.1145/3131151.3131171

[6] O. Ciupke. 1999. Automatic detection of design problems in object-oriented reengineering. In *Proceedings of Technology of Object-Oriented Languages and Systems - TOOLS 30 (Cat. No.PR00278)*. 18–32.

[7] Bill Curtis, Jay Sappidi, and Alexandra Szynkarski. 2012. Estimating the Size, Cost, and Types of Technical Debt. In *Proceedings of the Third International Workshop on Managing Technical Debt (MTD '12)*. IEEE Press, Piscataway, NJ, USA, 49–53. http://dl.acm.org/citation.cfm?id=2666036.2666045

[8] Neil A. Ernst, Stephany Bellomo, Ipek Ozkaya, Robert L. Nord, and Ian Gorton. 2015. Measure It? Manage It? Ignore It? Software Practitioners and Technical Debt. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*. Association for Computing Machinery, New York, NY, USA, 50âĂŞ60. https://doi.org/10.1145/2786805.2786848

[9] M Fowler. 1999. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, Boston.

[10] J Garcia, D Popescu, G Edwards, and N Medvidovic. 2009. Identifying Architectural Bad Smells. In *CSMR09; Kaiserslautern, Germany*. IEEE.

[11] Joshua Garcia, Daniel Popescu, George Edwards, and Nenad Medvidovic. 2009. Toward a Catalogue of Architectural Bad Smells. In *Architectures for Adaptive Software Systems*, Raffaela Mirandola, Ian Gorton, and Christine Hofmeister (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 146–162.

[12] M Godfrey and E Lee. 2000. Secrets from the Monster: Extracting Mozilla's Software Architecture. In *CoSET-00; Limerick, Ireland*. 15–23.

[13] Phil Greenwood, Thiago Bartolomei, Eduardo Figueiredo, Marcos Dosea, Alessandro Garcia, Nelio Cacho, Cláudio Sant'Anna, Sergio Soares, Paulo Borba, Uirá Kulesza, and Awais Rashid. 2007. On the Impact of Aspectual Decompositions on Design Stability: An Empirical Study. In *Proceedings of the 21st European Conference on Object-Oriented Programming (ECOOP'07)*. Springer-Verlag, Berlin, Heidelberg, 176–200. http://dl.acm.org/citation.cfm?id=2394758.2394771

[14] P. Kaminski. 2007. Reforming Software Design Documentation. In *14th Working Conference on Reverse Engineering (WCRE 2007)*. 277–280.

[15] Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan. 2012. A Field Study of Refactoring Challenges and Benefits. In *Proceedings of the 20th International Symposium on the Foundations of Software Engineering (FSE '12)*. ACM, New York, NY, USA, Article 50, 11 pages. https://doi.org/10.1145/2393596.2393655

[16] Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan. 2014. An empirical study of refactoringchallenges and benefits at microsoft. *IEEE Transactions on Software Engineering* 40, 7 (2014), 633–649.

[17] M. Raveendra Kumar and R. Hari Kumar. 2011. Architectural Refactoring of a Mission Critical Integration Application: A Case Study. In *Proceedings of the 4th India Software Engineering Conference (ISEC âĂŹ11)*. Association for Computing Machinery, New York, NY, USA, 77âĂŞ83. https://doi.org/10.1145/1953355.1953365

[18] M Lanza and R Marinescu. 2006. *Object-Oriented Metrics in Practice*. Springer, Heidelberg.

[19] Yun Lin, Xin Peng, Yuanfang Cai, Danny Dig, Diwen Zheng, and Wenyun Zhao. 2016. Interactive and Guided Architectural Refactoring with Search-Based Recommendation. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016)*. Association for Computing Machinery, New York, NY, USA, 535âĂŞ546. https://doi.org/10.1145/2950290.2950317

[20] M. Lippert and S. Roock. 2006. *Refactoring in Large Software Projects: Performing Complex Restructurings Successfully*. Wiley. https://books.google.com.br/books?id=bCEYuB83ROcC

[21] A MacCormack, J Rusnak, and C Baldwin. 2006. Exploring the Structure of Complex Software Designs: An Empirical Study of Open Source and Proprietary Code. *Manage. Sci.* 52, 7 (2006), 1015–1030.

[22] I Macia. 2013. *On the Detection of Architecturally-Relevant Code Anomalies in Software Systems*. Ph.D. Dissertation. Pontifical Catholic University of Rio de Janeiro, Informatics Department.

[23] I. Macia, R. Arcoverde, A. Garcia, C. Chavez, and A. von Staa. 2012. On the Relevance of Code Anomalies for Identifying Architecture Degradation Symptoms. In *CSMR12*. 277–286.

[24] Isela Macia, Joshua Garcia, Daniel Popescu, Alessandro Garcia, Nenad Medvidovic, and Arndt von Staa. 2012. Are Automatically-detected Code Anomalies Relevant to Architectural Modularity?: An Exploratory Analysis of Evolving Systems. In *AOSD '12*. ACM, New York, NY, USA, 167–178.

[25] Marinescu. 2004. Detection strategies: metrics-based rules for detecting design flaws. In *Proceedings of 20th IEEE International Conference on Software Maintenance (ICSM); Chicago, USA*. 350–359.

[26] Robert C. Martin and Micah Martin. 2006. *Agile Principles, Patterns, and Practices in C# (Robert C. Martin)*. Prentice Hall PTR, Upper Saddle River, NJ, USA.

[27] Ran Mo, Yuanfang Cai, R. Kazman, and Lu Xiao. 2015. Hotspot Patterns: The Formal Definition and Automatic Detection of Architecture Smells. In *Software Architecture (WICSA), 2015 12th Working IEEE/IFIP Conference on*. 51–60.

[28] N. Moha, Y. g. Gueheneuc, and P. Leduc. 2006. Automatic Generation of Detection Algorithms for Design Defects. In *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*. 297–300. https://doi.org/10.1109/ASE.2006.22

[29] N Moha, Y Gueheneuc, L Duchien, and A Le Meur. 2010. DECOR: A Method for the Specification and Detection of Code and Design Smells. *IEEE Transaction on Software Engineering* 36 (2010), 20–36.

[30] Emerson Murphy-Hill, Chris Parnin, and Andrew P. Black. 2009. How We Refactor, and How We Know It. In *Proceedings of the 31st International Conference on Software Engineering (ICSE '09)*. IEEE Computer Society, Washington, DC, USA, 287–297. https://doi.org/10.1109/ICSE.2009.5070529

[31] W Oizumi, A Garcia, L Sousa, B Cafeo, and Y Zhao. 2016. Code Anomalies Flock Together: Exploring Code Anomaly Agglomerations for Locating Design Problems. In *The 38th International Conference on Software Engineering; USA*.

[32] Willian Nalepa Oizumi, Leonardo da Silva Sousa, Anderson Oliveira, Alessandro Garcia, O. I. Anne Benedicte Agbachi, Roberto Felicio Oliveira, and Carlos Lucena. 2018. On the identification of design problems in stinky code: experiences and tool support. *J. Braz. Comp. Soc.* 24, 1 (2018), 13:1–13:30. https://doi.org/10.1186/s13173-018-0078-y

[33] Anderson Oliveira, Leonardo Sousa, Willian Oizumi, and Alessandro Garcia. 2019. On the Prioritization of Design-Relevant Smelly Elements: A Mixed-Method, Multi-Project Study. In *Proceedings of the XIII Brazilian Symposium on Software Components, Architectures, and Reuse (SBCARS '19)*. ACM, New York, NY, USA, 83–92. https://doi.org/10.1145/3357141.3357142

[34] 2020 Replication Package. 2020. https://figshare.com/s/3ac0da284f700d186dfa. (2020).

[35] Meilir Page-Jones. 2000. *Fundamentals of Object-oriented Design in UML*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[36] F. Palomba, G. Bavota, M. D. Penta, R. Oliveto, and A. D. Lucia. 2014. Do They Really Smell Bad? A Study on Developers' Perception of Bad Code Smells. In *2014 IEEE International Conference on Software Maintenance and Evolution*. 101–110. https://doi.org/10.1109/ICSME.2014.32

[37] David L. Parnas. 1978. Designing Software for Ease of Extension and Contraction. In *Proceedings of the 3rd International Conference on Software Engineering (ICSE '78)*. IEEE Press, Piscataway, NJ, USA, 264–277.

[38] Dewayne E. Perry and Alexander L. Wolf. 1992. Foundations for the Study of Software Architecture. *SIGSOFT Softw. Eng. Notes* 17, 4 (Oct. 1992), 40–52. https://doi.org/10.1145/141874.141884

[39] P. Rachow. 2019. Refactoring Decision Support for Developers and Architects Based on Architectural Impact. In *2019 IEEE International Conference on Software Architecture Companion (ICSA-C)*. 262–266. https://doi.org/10.1109/ICSA-C.2019.00054

[40] Refactoring Oracle [n. d.]. Refactoring Oracle. ([n. d.]). http://refactoring.encs.concordia.ca/oracle/

[41] Luca Rizzi, Francesca Arcelli Fontana, and Riccardo Roveda. 2018. Support for Architectural Smell Refactoring. In *Proceedings of the 2nd International Workshop on Refactoring (IWoR 2018)*. Association for Computing Machinery, New York, NY, USA, 7âĂŞ10. https://doi.org/10.1145/3242163.3242165

[42] Santonu Sarkar, Shubha Ramachandran, G. Sathish Kumar, Madhu K. Iyengar, K. Rangarajan, and Saravanan Sivagnanam. 2009. Modularization of a Large-Scale Business Application: A Case Study. *IEEE Softw.* 26, 2 (March 2009), 28–35. https://doi.org/10.1109/MS.2009.42

[43] S Schach, B Jin, D Wright, G Heller, and A Offutt. 2002. Maintainability of the Linux kernel. *Software, IEE Proceedings* - 149, 1 (2002), 18–23.

[44] Marcelino Campos Oliveira Silva, Marco Tulio Valente, and Ricardo Terra. 2016. Does Technical Debt Lead to the Rejection of Pull Requests?. In *Proceedings of the 12th Brazilian Symposium on Information Systems (SBSI '16)*. 248–254.

[45] Leonardo Sousa, Anderson Oliveira, Willian Oizumi, Simone Barbosa, Alessandro Garcia, Jaejoon Lee, Marcos Kalinowski, Rafael de Mello, Baldoino Fonseca,

Roberto Oliveira, Carlos Lucena, and Rodrigo Paes. 2018. Identifying Design Problems in the Source Code: A Grounded Theory. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. ACM, New York, NY, USA, 921–931. https://doi.org/10.1145/3180155.3180239

[46] Leonardo Sousa, Roberto Oliveira, Alessandro Garcia, Jaejoon Lee, Tayana Conte, Willian Oizumi, Rafael de Mello, Adriana Lopes, Natasha Valentim, Edson Oliveira, and Carlos Lucena. 2017. How Do Software Developers Identify Design Problems?: A Qualitative Analysis. In *Proceedings of 31st Brazilian Symposium on Software Engineering (SBES'17)*. 12.

[47] Michael Stal. 2014. Chapter 3 - Refactoring Software Architectures. In *Agile Software Architecture*, Muhammad Ali Babar, Alan W. Brown, and Ivan Mistrik (Eds.). Morgan Kaufmann, Boston, 63 – 82. https://doi.org/10.1016/B978-0-12-407772-0.00003-4

[48] Girish Suryanarayana, Ganesh Samarthyam, and Tushar Sharma. 2014. *Refactoring for Software Design Smells: Managing Technical Debt* (1st ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[49] A. Trifu and R. Marinescu. 2005. Diagnosing design problems in object oriented systems. In *WCRE'05*. 10 pp.

[50] Adrian Trifu and Urs Reupke. 2007. Towards Automated Restructuring of Object Oriented Systems. In *CSMR '07*. IEEE, Washington, DC, USA, 39–48.

[51] Nikolaos Tsantalis, Victor Guana, Eleni Stroulia, and Abram Hindle. 2013. A multidimensional empirical study on refactoring activity. In *Proceedings of the 2013 Conference of the Center for Advanced Studies on Collaborative Research*. IBM Corp., 132–146.

[52] Nikolaos Tsantalis, Matin Mansouri, Laleh M. Eshkevari, Davood Mazinanian, and Danny Dig. 2018. Accurate and Efficient Refactoring Detection in Commit History. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. ACM, New York, NY, USA, 483–494. https://doi.org/10.1145/3180155.3180206

[53] Michele Tufano, Fabio Palomba, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Andrea De Lucia, and Denys Poshyvanyk. 2015. When and Why Your Code Starts to Smell Bad. In *Proceedings of the 37th International Conference on Software Engineering (ICSE '15)*. ACM, New York, NY, USA, 13.

[54] J van Gurp and J Bosch. 2002. Design erosion: problems and causes. *Journal of Systems and Software* 61, 2 (2002), 105 – 119.

[55] S. Vidal, E. Guimaraes, W. Oizumi, A. Garcia, A. D. Pace, and C. Marcos. 2016. Identifying Architectural Problems through Prioritization of Code Smells. In *SBCARS16*. 41–50.

[56] Lu Xiao, Yuanfang Cai, Rick Kazman, Ran Mo, and Qiong Feng. 2016. Identifying and Quantifying Architectural Debt. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, New York, NY, USA, 488–498. https://doi.org/10.1145/2884781.2884822

[57] Zhenchang Xing and Eleni Stroulia. 2005. UMLDiff: An Algorithm for Object-oriented Design Differencing. In *Proc. of ASE '05*. 54–65. https://doi.org/10.1145/1101908.1101919

[58] A. Yamashita and L. Moonen. 2013. Do developers care about code smells? An exploratory survey. In *2013 20th Working Conference on Reverse Engineering (WCRE)*. 242–251. https://doi.org/10.1109/WCRE.2013.6671299

[59] A Yamashita and L Moonen. 2013. Exploring the impact of inter-smell relations on software maintainability: an empirical study. In *Proceedings of the 35th International Conference on Software Engineering; San Francisco, USA*. 682–691.

[60] O. Zimmermann. 2015. Architectural Refactoring: A Task-Centric View on Software Evolution. *IEEE Software* 32, 02 (mar 2015), 26–29. https://doi.org/10.1109/MS.2015.37

[61] Olaf Zimmermann. 2017. Architectural refactoring for the cloud: a decision-centric view on cloud migration. *Computing* 99, 2 (01 Feb 2017), 129–145. https://doi.org/10.1007/s00607-016-0520-y