Noname manuscript No.
(will be inserted by the editor)

Towards a Smell-Based Approach for Architectural Refactoring

Willian Oizumi · Leonardo Sousa · Anderson Oliveira · Alessandro Garcia · Diego Cedrim · Carlos Lucena

Received: date / Accepted: date

Abstract Design degradation occurs as a result of design decisions that negatively impact non-functional requirements, such as modifiability and extensibility. Architectural problems are specific forms of degradation that affect multiple code elements relevant to software architecture. Architectural problems must be identified and removed to avoid architecture decay. Otherwise, the software may suffer negative consequences, such as massive restructuring and increased maintenance costs. Code smells are symptoms of design degradation. However, little is known on when and what types of smells indicate an architectural problem that needs to be removed from the system. In this work, we conducted a multi-case study to investigate (1) when smells are indicators of architectural refactoring opportunities, and (2) what smell patterns indicate the existence of architectural problems. As a result, we identified smell patterns, which are different sets of one or multiple smell types that are associated with architectural problems. A pattern indicates not only the type of architectural problem but also the refactorings to remove it. Based on them, we developed an automated approach to find architectural refactoring opportunities. To evaluate its usefulness, we conducted a quasi-experiment with professional developers. We found that smell patterns have the potential to indicate refactoring opportunities for architectural problems such as *Concern Overload*, *Scattered Concern*, and *Fat Interface*. Our results also indicate that *God Class* and *Complex Class* are highly relevant smells for identifying and refactoring architectural problems. Finally, we identified factors that influence the developers' decision in classifying code elements as being affected by architectural problems.

Keywords Refactoring · Code Smell · Architectural Problem · Architectural Refactoring

Willian Oizumi
Pontifical Catholic University of Rio de Janeiro
E-mail: woizumi@inf.puc-rio.br

1 Introduction

The structural quality of a software system depends on how it meets non-functional requirements, such as modifiability, extensibility and readability [9, 59, 62]. There is a design degradation problem when a non-functional requirement is not met [59]. Degradation problems can impact isolated code elements (e.g., a method or a class) or multiple code elements and structures such as hierarchies and components [55]. The former is usually represented by code smells. A smell is a code-level structure that is a surface indication of a design degradation [13]. The latter is known as *Architectural Problem*. It is a degradation that affects how the system is decomposed into modules and how they communicate with each other [14, 15, 26]. An example of architectural problem is the *Fat Interface* [33].

There is empirical evidence that architectural problems are among the most important sources of technical debt [10, 12, 57]. For instance, a study with 745 software projects showed that architectural problems are directly related to a significant increase in software project costs [10]. Therefore, given the harmfulness of architectural problems, developers should remove them as early as possible.

As the architectural documentation is usually unavailable, developers may have to directly analyze the source code, using symptoms such as code smells and metrics, to identify architectural problems [59]. Refactoring is an activity that developers can use to remove architectural problems [13]. Despite the benefits, removing architectural problems through refactoring is not trivial [59]. Developers perceive architectural refactorings as costly and risky [12, 20] since they are not confident in recognizing architectural refactoring opportunities, specially without automated support. In fact, developers might not even know when a problem exists. Thus, developers need automated support to know when and how they should apply architectural refactorings.

We hypothesize that developers would benefit of an smell-based approach that indicates (i) the type of architectural problem and (ii) the refactorings to remove it. Additionally, we believe that a smell-based approach can help developers in adopting architectural refactoring. First, developers are familiar with code smells. According to the literature, not only they are familiar with smells [69], but they also use them to identify architectural problems [59, 60]. Second, they can rely on smells to identify low-level code refactorings [13]. Therefore, if they can use smells to identify architectural problems and apply code refactoring, they may benefit from using smells to also identify and apply architectural refactorings.

A smell-based approach relies on the identification of code smells. As not all smells are related to an architectural problem [6, 31, 43], it is important to find the smells that consistently indicate an architectural problem. There are multiple studies that investigate the relation between code smells and architectural problems [3, 34, 39, 40, 54]. Nevertheless, no previous work have defined specific patterns of code smell types that are consistent indicators of architectural problems. Additionally, the smells should also indicate the

architectural refactorings to remove the problem. Unfortunately, little is known on what types of refactoring are useful for removing architectural problems.

In order to develop a smell-based approach with these characteristics, we need to know when smells are indicators of architectural refactoring opportunities. Thus, in this work we conducted a multi-case study and a quasi-experiment to investigate when smells are reliable indicators of architectural refactoring opportunities. To achieve our goal, we focused on three research questions as described below:

RQ1. When are smells indicators of architectural refactoring opportunities for developers?

RQ1 allowed us to better understand the relation among refactoring, smells, and architectural problems - a triple relationship that has not been explored simultaneously in the literature. With this research question we investigated when smells are indicators of architectural refactoring opportunities. However, this research question is not enough to support developers in adopting architectural refactorings. Developers may need more concrete information that would help them to identify and remove the architectural problem. For this purpose, we want to find a subset of code smells that can help them in identifying and refactoring an instance of an architectural problem. To provide this subsets, we further examined the results of RQ1 and identified, what we call, **smell patterns**. A smell pattern is one or more types of smells that together are likely to indicate a particular architectural problem. This investigation is the focus of our second research question:

RQ2. What are the smell patterns that consistently indicate an architectural problem?

To answer RQ1 and RQ2, we conducted a multi-case study investigating architectural problems that underwent repairing actions by developers. For this purpose, we located the code refactorings that occurred in the commit history (147,736 commits) of 50 software projects. We searched for elements modified by root-canal refactorings, and we verified if they contained smells that could indicate architectural problems (RQ1). After that, we searched for recurrent cases in which the same set of code smells indicate the same type of architectural problem. We used the code smells within the set to extract the refactorings types required to remove the architectural problem (RQ2). We focus our analysis on the root-canal refactorings since these transformations are primarily targeted to improve the structural design [13, 38]. Thus, elements refactored during this tactic have a high chance of containing architectural problems.

These two research questions were the focus of our previous work [58]. In that work, we identified the smells patterns, which could be the core of a smell-based approach. Nevertheless, we did not have evidence whether developers would benefit from the patterns. Thus, in this paper we extended our previous

work by conducting a quasi-experiment with 13 professional software developers. In this quasi-experiment, we investigate whether the smell patterns help developers to identify architectural problems in practice. We also investigate if refactorings associated with the smells of each pattern are enough for removing architectural problems. This investigation is the focus of our third research question:

RQ3. To what extent do smell patterns help developers identify and refactor architectural problems?

To answer RQ3, we asked the quasi-experiment's participants to identify architectural problems in their software systems. They had to analyze different cases of possible architectural problems. In some cases, they had to use the patterns and not in other cases. This experiment allowed us to investigate to what extent smell patterns help developers to identify and refactor architectural problems.

Based on the multi-case study, 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 some examples from the literature. For instance, most related studies agree that *God Class* is an indicator of architectural refactoring for removing an architectural problem [1,46,69]. 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. The multi-case study also helped us to find smell patterns that can indicate architectural refactoring opportunities. Most of these patterns have not been presented elsewhere.

With our quasi-experiment, we were able to assess the usefulness of smell patterns based on the opinion of professional developers. As in the previous study, we found evidence that multi-smell patterns are the most relevant for detecting architectural problems. We observed that they may be strong indicators of architectural problems such as *Concern Overload*, *Scattered Concern*, and *Fat Interface*. However, it was not possible to observe a statistically significant correlation between patterns and architectural problems. Thus, we identified some factors that should be considered to improve the identification of architectural problems with smell patterns.

2 Background and Terminology

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

2.1 Design Degradation Problems

Design degradation problems occur when non-functional requirements are negatively impacted by design decisions [59,62]. In this work, we consider two categories of design degradation: architectural problems and implementation problems. These two categories provides different perspectives on what developers may consider as degradation. Each one of them covers a different scope of degradation in the source code.

Architectural problem is a degradation that negatively impacts high level structures, such as components, abstractions, hierarchies of code elements, and other structures that are relevant to the software architecture [4]. Hence, these problems affect, but are not limited to, how the system is organized into subsystems and components, how and which code elements encapsulate process and data to address each functionality, and how the elements interact with each other and their execution environment [14,15,26]. These types of architectural problems are often harmful in software systems [16,18,27,53]. An example of architectural problem is *Fat Interface*, which occurs when an interface provides multiple unrelated services [33]. A *Fat Interface* usually harm quality attributes such as modifiability and extensibility. Since this degradation problem affects major architectural structures, developers may need more support to identify and remove them.

An implementation problem is a degradation that negatively impacts isolated code elements such as a method or a class [55]. A recurrent example of implementation problem is when a method is too long and complex to understand. Such a problem negatively affects the readability of the source code. Since they are more localized, they might have less impact on the architecture.

2.2 Smells Indicating Design Degradation Problems

Code smells are indicators of the occurrence of design degradation problems in the source code [13,23]. Examples of smells include *Long Method*, *God Class*, *Long Parameter List*, and *Speculative Generality* [13,23]. Code smells are often associated with implementation problems. For instance, a *Long Method* smell is an indicator that the method may be too long and complex to understand.

There is also evidence that smells may be fully or partially associated with architectural problems [36,40,61,66]. For instance, the *Unused Abstraction* problem happens when the code element representing the abstraction is not directly used or is unreachable [7,26]. 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 [13].

In this work we are investigating the use of smells to identify architectural problems. Thus, let us consider the Figure 1 to better 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 [17]: 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.

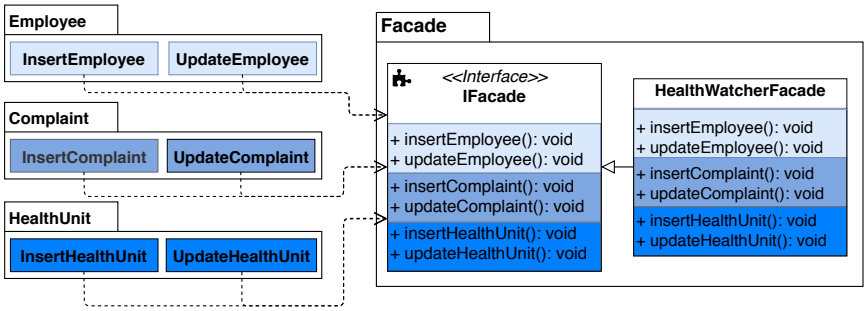

Fig. 1 Example of Architectural Problem

Classes in Employee, Complaint and HealthUnit packages are clients of the IFacade interface and contain smells such as *Feature Envy* and *Dispersed Coupling*. *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.

2.3 Refactorings for Removing Architectural Problems

Refactoring is a popular technique to remove degradation problems from a system. Refactoring is defined as a program transformation intended at preserving the observable behavior and improving the program structure [13]. Although this definition entails an expectation that refactoring always preserve the system behavior, depending on the tactic applied, this is not always

the case [38]. We describe next the tactics and other basic concepts related to architectural refactoring.

Refactoring Type indicates the form of the transformation applied to attributes, methods, classes and interfaces. We considered the 13 refactoring types [13]: *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 [38] 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*.

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 [14]
Cyclic Dependency	Architectural problem that happens when one or more elements depend on each other, creating a cycle [47]
Component Overload	Architectural problem that happens when a component is overloaded with responsibilities [30]
Concern Overload	Architectural problem that happens when a code element fulfills too many responsibilities [30]
Fat Interface	Architectural problem that happens when an interface exposes many functionalities and many of those functionalities are not related to each other [33]
Incomplete Abstraction	Architectural problem that happens when an element does not support a responsibility completely in their enclosing component [45]
Misplaced Concern	Architectural problem that happens when an element implements a functionality, which is not the predominant one of their enclosing component [30]
Scattered Concern	Architectural problem that happens when elements are responsible for the same functionality, but some of them cross-cut the system [14]
Unused Abstraction	Architectural problem that happens when the code element representing the abstraction that is not directly used or is unreachable [7]
Unwanted Dependency	Architectural problem that happens when a dependency violates a rule defined on the system architecture [48]

Refactored Elements comprise all elements that refactorings directly affect. Each refactoring type affects directly different elements [13]. 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 [38]: *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.

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 [72, 25]. 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 [25].

3 Identification of Smell Patterns: Study Design

As discussed before, we investigate architectural problems that underwent repairing actions to answer RQ1 and RQ2. First, we searched for elements modified by root-canal refactorings - these are the repairing actions. Second, we verified if the elements had smells that could indicate architectural problems. We focus our analysis on the root-canal refactorings since it is a tactic primarily applied to repair deteriorated code [13, 38], which can be caused by the presence of architectural problems. Thus, elements refactored under this tactic have a high chance of containing 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 [16, 18, 27, 53].

Subsections bellow present the fours phases to answer RQ1 and RQ2[1].

3.0.1 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 [5,8]. Furthermore, we also selected projects in Java due to the availability of tools to identify refactorings [63] and code smells [42]. Thus, we selected Java projects with a diversity of structure, size and popularity. Table 2 lists the projects used in the experiment.

3.1 Phase 2: Code Refactoring Detection

The second phase consisted of detecting code refactorings for all selected projects. We chose Refactoring Miner [63,64] (version 0.2.0) as the tool to detect code refactorings. This tool implements a lightweight version of UMLDiff algorithm [68] for differentiating 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% [64,50] 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 2.3).

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*

[1] We provide a replication package with the entire dataset, including the refactorings and smells collected for the 50 GitHub projects used in our study. It is available here [44]

Table 2 Open Source Projects Used in this Study

Domain	Project	LOC	N°of Classes	Commits	Stars
Android	JARA	188,003	69	109	0
	Facebook Fresco	50,779	860	744	14,679
	OkHttp	49,739	642	2,645	27,421
	Google I/O Sched App	40,015	754	129	15,686
	Mayhem and Hell	25,043	304	148	1
	PhilJay MPAndroidChart	23,060	268	1,737	23,036
	WhatsUp (MarvinBellmann)	10,453	40	108	1
	Dagger	8,889	441	696	11,097
	Android Bootstrap	4,180	123	230	4,298
	LeakCanary	3,738	127	265	19,847
	Orhanobut Logger	887	11	68	9,423
Application	Containing	4,022,774	136	818	1
	Bublag Confetti	1,481,974	417	210	0
	Google J2ObjC	385,012	4,866	2,823	5,172
	ArgoUML	177,467	2,597	17,654	5
	Apache Ant	137,314	1,784	13,331	205
	Achilles	83,124	653	1,188	207
	Passsafe	12,203	196	150	0
	Market-monitor	3,763	44	125	0
Database	Apache Derby	1,760,766	3,741	8,135	140
	Presto DB	350,976	4,146	8,056	7,740
	Realm Java	50,521	1,018	5,916	9,682
Framework	Spring Framework	555,727	12,715	12,974	22,052
	Ikasan	537,283	2,515	2,465	15
	Apache Dubbo	104,267	1,690	1,836	19,934
	Tap4j	34,026	123	146	16
	Alfred MPI	5,545	54	145	2
	JUnit4	2,113	1,251	309	6,935
Library	Elasticsearch	578,561	8,845	23,597	32,200
	PhiCode Philib	238,086	163	892	1
	Spring Boot	178,752	5,178	8,529	26,294
	JBoss Xerces	140,908	1,136	5,456	4
	Facebook SDK for Android	42,801	836	601	4,534
	Netflix Hystrix	42,399	1,569	1,847	14,172
	JBoss Ballroom	20,695	215	635	0
	Retrofit	12,723	554	1,349	26,557
	Drugis Common	12,195	254	240	6
	IRC Bot (c2nes/ircbot)	8,159	80	107	0
	Whydah - UserAdminService	7,454	60	249	0
	Pusher Java Client	7,029	74	352	174
	Lyra	6,603	95	192	245
	Dynamic Collections	5,955	215	180	6
	Elasticsearch Transport Thrift	4,450	48	113	80
Pluggin	Sen Word-Builder	736,148	65	120	0
	TUBAME Migration Tool	378,855	552	315	9
	GitHub Pull Request Builder	8,094	66	589	0
Web Application	Apache Tomcat	668,720	2,275	18,068	2,406
	Media Magpie	62,938	470	336	1
	Netflix SimianArmy	16,577	244	710	6,618
	OpenConext-crunche	4,574	31	108	0

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 [38].

3.2 Phase 3: Code Smell Detection

For this study, we decided to detect code smells with metrics-based strategies [32, 23]. Thus, we can compare our results with related studies that used the same detection strategies [5, 8]. These strategies are based on a set of detection rules [5, 23] that compare metric values with predefined thresholds according to logical operators.

In this study, we decided to use the Organic tool [42], which is able to detect 17 code smells. We selected Organic and its smells due to the following reasons. First, metrics, rules and thresholds implemented in Organic are the ones commonly used in the literature for code smells [23, 5]. Moreover, there is evidence that those smells may be closely related to architectural problems [13, 36, 40, 66, 31, 28, 37]. Next we describe our rationally on how each of the 17 smells may be related to architectural problems:

- ***Brain Class*, *God Class*, *Brain Method*, and *Long Method*.** Smells that are closed related and usually indicate classes and methods that concentrates many responsibilities and are too large. Such smells may indicate low cohesion and lack of modularization.
- ***Class Data Should Be Private*.** A smell that indicates a lack of encapsulation in the affected class, which is related to the architecture.
- ***Data Class* and *Lazy Class*.** Two smells that occur in classes that only hold data or that are too small to exist. They may be indicators of inadequate separation of concerns and violation of object-oriented principles.
- ***Complex Class*.** This smell indicates a class that has high cyclomatic complexity. Although a high cyclomatic complexity does not directly affect the architecture, it can indicate a lack of modularization.
- ***Dispersed Coupling* and *Intensive Coupling*.** Both smells are related to coupling, which is an important architectural concept.
- ***Feature Envy* and *Shotgun Surgery*.** Both smells may indicate the existence of a functionality that is scattered in two or more code elements.
- ***Long Parameter List* and *Message Chain*.** Both smells are not directly related to the architecture. However, they may be indirect symptoms of poorly designed APIs or interfaces, which may negatively impact the architecture.
- ***Spaghetti Code*.** A smell type that indicates the possible violation of object-oriented principles.

– ***Speculative Generality* and *Refused Bequest.*** Two smell types that occur in the context of class hierarchies. Problems in hierarchical structures are often relevant to architecture, since changing inheritance relationships is often costly and tends to result in multiple side effects.

3.3 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% [64], 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 2.3). 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.

3.4 Algorithm for Categorization

Algorithm 1 presents our method to categorize the refactored elements according to the number of smells. The algorithm receives as input a set of projects, chosen in Phase 1 (Line 1). Then, it iterates over each project (Line 2), in which it retrieves two versions of the project: the i version and the subsequent one (Lines 3 and 4). The algorithm calls the Refactoring Miner (Line 5) to collect the refactoring operations, which comprises the first phase of the data collection process. Similarly, it calls our code smell detection tool to collect all the code smells for all the elements in the i version (Line 6). This routine comprises the third phase of the data collection process. The algorithm iterates over each detected refactoring to categorize the refactored elements (Line 8) into one of the categories defined in Section 3: *smell-free*, *single smell* or *multiple smells*. The categorized element is associated to the refactoring (Line 9); thus, for each refactoring, the algorithm associates all the refactored elements to one category. Then, the algorithm saves the results in the database (Line 11). We used the results of this algorithm to answer the research question RQ1.

Algorithm 1 Categorization of Refactored Elements

Require: Set of projects P, versions V, elements E, refactoring operations R
Ensure: The categorization of the refactored elements

```
 1: P ← {p_1, p_2, ··· , p_n} {Phase 1}
 2: for all p ∈ P do
 3:     for all v_i ∈ V(p) do
 4:         if v_{i+1} ∈ V(p) then
 5:             Collect all refactoring operations R(v_i, v_{i+1}) {Phase 2}
 6:             Collect smells for all e ∈ E(v_i) {Phase 3}
 7:             for all r ∈ R(v_i, v_{i+1}) do
 8:                 smellCategory = categorizeAccordingToSmellPresence(r.e)
 9:                 r.targets += (smellCategory)
10:             end for
11:             saveResultDB(R(v_i, v_{i+1}))
12:         end if
13:     end for
14: end for
```

3.5 Identification of Smell Patterns

Several studies have investigated the relation between code smells and architectural problems [11,62,27,52,36,6,29,31,69,46,65,40]. The use of code smells to indicate architectural problems is indeed reasonable as each smell may be fully or partially associated with an architectural problem [62,36,31,28,61,66, 40]. For instance, `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 [13]. This code smell can be associated with *Unused Abstraction* [7,26]. Moreover, smells can be identified directly in source code, which is often the only tangible artifact available for developers. Hence, it is no surprise that the use of code smells is often associated with architectural problem identification in the literature [13,37,36,31,28,66,40].

To answer RQ2, we first searched on the literature studies that investigated when one or more types of smells are likely to indicate a particular architectural problem. Based on them, we can associate some types of code smells with some types of architectural problems, as we did when we associated `Speculative Generality` with *Unused Abstraction*. Such an association is possible due to smell patterns reported in studies that investigated the relation between smells and architectural problems. After identifying the patterns defined in the literature, we searched in the database for the 50 software projects if these patterns appeared in the refactored elements. Then we conducted a manual validation to verify if the smells within the pattern are associated with an architectural problem.

4 Towards Smell Patterns

We discuss here the results for RQ1 and RQ2. First, we analyzed whether the refactored elements are associated with the absence or presence of smells (Section 4.1). Second, we investigate when smells can indicate architectural refactoring opportunities in general (Section 4.2). Then, we further discuss different examples of when smells are (or are not) indicators of architectural refactoring opportunities (Section 4.3). Finally, we used these examples to find recurring types of smells that may indicate architectural refactoring opportunities for specific types of architectural problems (Section 4.4).

4.1 Categorization of Refactored Elements

To answer our first research question ("When are smells indicators of architectural refactoring opportunities for developers?"), we searched for elements modified by root-canal refactorings (refactored elements), and we verified if they contained smells that could indicate architectural problems. To answer this question, we first categorized the refactored elements according to the number of smells found in the element (Section 3). The result for this categorization, showed in Table 3, considers both refactoring tactics (root-canal

and floss refactoring). 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 3 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.

4.2 Analysis of Root-canal Refactoring

Table 4 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. This

organization allows us to explore when each code refactoring type is an architectural refactoring to certain architectural problems. For instance, refactoring types classified as *Within Hierarchy* are usually applied to remove those architectural problems related to design decisions that affect the system hierarchy, *e.g.*, *Ambiguous Interface* and *Fat Interface*.

Table 4 Categorization of Refactorings According to Root-canal Refactoring

All Collected Refactorings						Validated Root-canal Refactorings			
Category	**Activity**	**Refactoring 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 Interface	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%**

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 [38]. The 7^{th}*subcolumn* of Table III shows the 1,168 refactorings manually validated as root canal (Section 3.3). 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 Section 3. In summary, when developers focus on repairing deteriorated code, most code refactorings are applied to elements that contain mul-

tiple 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.

4.3 Relating Code Refactoring, Smells and Architectural Problems

When we analyze each refactoring type in Table 4, 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) [33]. 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, 46, 69]. 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 4.4).

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:

> **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* do 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 4.4, *Scattered Concerns* are often indicated by smells such as *Dispersed Coupling*, *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 indicators 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.

4.4 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 [28,31,37,36,40,66,13], we searched for patterns of smells and architectural problems (Section 3.5). 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 5. 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 6) to get rid of the smell, and, therefore, remove the architectural problem.

Table 5 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, Feature Envy, God Class/Complex Class, Intensive Coupling, and Long Method)
Concern Overload	(Complex Class, Feature Envy, God Class/Complex Class, Intensive Coupling, Long Method, and Shotgun Surgery)
Fat Interface	(Shotgun Surgery in the interface) or (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, 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 6 Recommended Refactoring for the Code Smell

Code Smell	Common Refactorings
Complex Class	Extract Method, Move Method, Extract Class [5]
Dispersed Coupling	Extract Method
Feature Envy	Move Method, Move Field, Extract Field [13]
God Class	Extract Class, Move Method, Move Field [5]
Invensive Coupling	Move Method, Extract Method
Lazy Class	Inline Class, Collapse Hierarchy [13]
Long Method	Extract Method [13]
Shotgun Surgery	Move Method, Move Field, Inline Class [13]
Speculative Generality	Collapse Hierarchy, Inline Class, Remove Parameter, Rename Method [13]

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 7 shows the number of architectural problems that we found in our database according to these patterns.

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

Table 7 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		

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*, *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 (*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 [1,30,40,66,70]. 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%.

4.5 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 [5, 23, 40].

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 [21], *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 system-

atic process to select projects. As a result, we obtained relevant Java projects with a diversity of structure and size metrics.

5 Evaluating the Smell Patterns: Experiment Design

In the previous study (Section 4.4), we identified smell patterns – a pattern is a recurrent case in which the same set of code smells indicate the same type of architectural problem. Additionally, the smells within a pattern also indicate the refactoring types that the developers may apply to remove the architectural problem.

This second study investigates if developers would benefit from the smell patterns to identify and refactor architectural problems. We conducted a quasi-experiment with professional software developers to evaluate the extent to which smell patterns help developers identify architectural problems (RQ3). First, we recruited software developers to participant in the quasi-experiment. Then, we asked them to identify architectural problems in their software systems. They had to analyze different cases of possible architectural problems. In some cases, they had to use the patterns and not in other cases. The following subsections present the study settings.

5.1 Recruiting Participants

We recruited the participants of this quasi-experiment through our network of contacts in the industry and in other research groups. We also looked for potential participants in our professional social media (Twitter and LinkedIn). We defined the following criteria to select the final list of participants:

1. Intermediary knowledge about the Java programming language.
2. Intermediary knowledge about software architecture.
3. Basic knowledge about code smells and refactoring.
4. More than one year of experience with software development.

Such characteristics were informed by the participants themselves through a characterization form. We did not carry out any tests to verify the experience and level of knowledge on each topic.

Table 8 shows a list of participants selected based on our criteria. We identify each participant by an identification number (ID) in the first column[2]. The second column shows the experience (in years) of each participant with software development. The last four columns show each participant's knowledge about the Java programming language, software architecture, code smells, and refactoring, respectively. We provide the other characteristics of the participants in our replication package [44].

We asked the participants to analyze a project they are familiar with as part of the study (Section 5.2). Thus, we also collected information about the

[2] We will use the ID to reference particular participants.

Table 8 List of the selected participants for this study.

ID	Programming Experience	Java Knowledge	Architecture Knowledge	Code Smells Knowledge	Refactoring Knowledge
1	6 to 10 years	Specialist	Experienced	Experienced	Experienced
2	1 to 5 years	Intermediate	Intermediate	Experienced	Experienced
3	6 to 10 years	Intermediate	Intermediate	Experienced	Specialist
4	6 to 10 years	Intermediate	Intermediate	Intermediate	Intermediate
5	6 to 10 years	Intermediate	Intermediate	Basic	Basic
6	6 to 10 years	Experienced	Intermediate	Intermediate	Intermediate
7	6 to 10 years	Intermediate	Intermediate	Basic	Intermediate
8	1 to 5 years	Intermediate	Intermediate	Experienced	Intermediate
9	6 to 10 years	Experienced	Experienced	Experienced	Experienced
10	1 to 5 years	Intermediate	Intermediate	Experienced	Experienced
11	11 to 15 years	Specialist	Experienced	Experienced	Experienced
12	1 to 5 years	Experienced	Specialist	Experienced	Experienced
13	6 to 10 years	Experienced	Intermediate	Experienced	Experienced

Table 9 List of the software projects analyzed by participants in this study.

Name	License	Domain	Size	Participants
OPLA-Tool	OS	Academic/Tool/Model Optimization	Large	1
Fresco	OS	Mobile/Library/Media Management	Large	3
Fastjson	OS	Library/Parser	Large	6
Soot	OS	Academic/Tool/Compiler Optimization	Large	12
Couchbase Java Client	OS	Database driver	Medium	2, 10
JDeodorant	OS	Tool/Plugin/Source Code Analysis	Small	8
REST System	CS	REST API	Small	4
School Mgt System	CS	Web system/School Management	Small	5
Image Composition System	CS	Library/Media Manipulation	Small	11
Grace Language Compiler	OS	Academic/Compiler	Small	7
SportsTracker	OS	Desktop/Personal App	Small	9
Glide	OS	Library/Android/Caching/Media Loading	Medium	13

software projects selected by participants; this information is important to analyze the experiment's results. Table 9 presents a summary of the software projects. The first column shows the name of each project. In the second column we show each project's type of license - we use OS for denoting Open Source licences and CS for the Closed Source ones. The third column shows information about the domain of each project. In the fourth column, we classified the projects according to their size. We defined the following criteria for this classification: Small - less than 100K Source Lines of Code (SLOC), Medium - between 100K and 499K SLOC, and Large - more than 499k SLOC. Finally, in the last column, we show the ID of participants that analyzed each project.

5.2 Study Procedures

The experiment to answer our third research questions has four main steps. All the participants had to follow them.

Step 1: Characterization. Each participant provided information that helped us to understand their characteristics and how such characteristics impact on our results and conclusions. We selected the following characteristics

as we believe that they may influence the results: location (City/Country), gender, age, current role, software development experience (in years), Java knowledge, software architecture knowledge, refactoring knowledge, and code smells knowledge.

Step 2: Training. Participants had to go through a basic training one the study-related concepts - *i.e.*, software structural quality, design degradation problems, source code metrics, code smells, and refactoring. The goal of this training was to make sure all the participants had an understanding of the terms used during the quasi-experiment. To avoid bias, we restricted ourselves in providing only the main definition for each concept, and we also explained the tasks that the participants were required to perform. Since we adopted a remote approach, the training was provided through recorded videos and textual documents.

Step 3: Environment Setup. As part of the study, we asked the participants to analyze software projects they are familiar with (next step). To support this analyze, we had to collect information about the projects' internal structure (metrics, smells, and smell patterns). Thus, for each participant, we conducted the configuration of the environment and the collection of the required information as follows.

First, we run an adapted version of the Organic tool [42] (a metric-based smell detector) in the participants' systems [3]. This version of Organic analyzes the source of code of a software project to collect metrics, detect code smells, and find smell patterns. After that, it selects the code elements that the participants should analyze. The criteria for selecting code elements are the number of commits and the number of smells, in this priority order. To better illustrate these selection criteria, consider the following example.

A code element *CE1* is involved in ten commits and is affected by four code smells. Another code element *CE2* is involved in nine commits and is affected by five code smells. In such a scenario, *CE1* would be prioritized above *CE2*. We adopted such selection criteria because there is evidence that they are relevant for software maintainability [41].

Step 4. Experimental Task. After collecting all the required information, our tool provides six cases to be analyzed. A case is composed by one or multiple code elements that are affected by code smells. The six cases are divided into three groups:

- Multiple Smells Patterns: Two cases affected by the smell patterns composed by two or more smell types.
- Single Smell Patterns: Two cases affected by the smell patterns composed by a single smell type.
- Other (Combination of) Smells: Two cases affected by combinations of smells that are different from all smell patterns from the previous groups.

As each participant selected a different project to analyze, there were cases in which the tool was unable to detect all six cases involving the smell patterns

[3] The source code is available at: https://github.com/wnoizumi/degradation-experiment

and other combinations of smells. Therefore, in some cases the participant evaluated less cases (e.g., only four cases) or evaluated more than two cases involving other combinations of smells. This happened only with participants that analyzed small projects.

Our tool provides the appropriate information for each case according to the groups above. For example, suppose a participant is analyzing its first case using the single or multiple smells patterns. Then, our tool would provide information about the code metrics (*e.g.*, coupling, complexity, cohesion, etc.), code smell(s), and the pattern affecting the elements analyzed in this first case. The tool also provided a description (see Table 1) of the possible architectural problem occurring in the case and the suggested refactorings to remove it. Supposing the participant is analyzing a second case using other combinations of smells rather than a smell pattern. In this scenario, the tool provides almost the same information except for the architectural problem description. As we do not map the types of architectural problems indicated by other combinations of smells, in these cases, we provide only a description of a generic problem, indicating a possible impact on maintainability.

Case Analysis. For the case analysis task, our tool randomly presented six cases. Participants had to analyze each one according to the the information provided by the tool; then they had to fill out a form. In the form, they should inform whether they believed there was a structural degradation problem or not. In the cases that participants considered there is NO degradation problems, they had to provide a justification to their conclusion. They also had to rate the usefulness of the information provided by the tool to support their conclusion.

On the other hand, in the cases in which participants considered that there were degradation problems, we asked the following information:

- Description and justification for the degradation problems.
- Whether they considered that the degradation problems were at the architectural level or implementation level.
- Their perceived severity of the degradation problems.
- Their perceived usefulness of the information provided by the tool to reach such a conclusion.
- A description of each information (provided by the tool or not) that they used to identify the degradation problem.
- Whether the suggested refactorings would be enough for removing the degradation problem.

Post-study Interview. At the end of the study, we asked participants to answer a post-study interview. We asked them about external factors that may have affected the experiment – for example, – interruptions, problems with the tool, time and effort needed to perform the tasks etc. We also asked them the desirable characteristics that a degradation detection tool should have.

5.3 Data Analysis Procedures

We conducted quantitative and qualitative analyses to answer our third research question (RQ3). Each analysis allowed us to examine the data from complementary perspectives. The purpose of quantitative analyzes was to provide an objective answer to RQ3. After that, we applied qualitative analysis to understand and explain the quantitative results. Below we present details about the procedures to conduct our analysis.

Quantitative Analysis. For the quantitative analysis, we considered the responses according to the three groups of cases, i.e., multiple smells patterns, single smell patterns, and other combinations of smells. We compared the different groups of responses to evaluate whether smell patterns:

1. Are better than other combinations of smells to indicate the presence of degradation problems.
2. Are better than other combinations of smells to indicate the presence of architectural problems.
3. Indicate the presence of architectural problems more often than the presence of other degradation problems.

For the first two evaluations, we compared the precision of participants in identifying degradation problems and architectural problems, respectively. In this study, *Precision* is the number of cases classified by participants as representing degradation/architectural problems divided by the number of analyzed cases. To assess the statistical significance, we applied the One-Way Repeated Measures ANOVA.

For the third evaluation, we compared the proportion of smell patterns cases indicating architectural problems to the proportion of smell patterns cases indicating implementation problems. We also compared such proportions between the different groups (Multiple Smells, Single Smell, Other Combinations) using the Chi-Square Test. Finally, we analyzed the precision of each smell pattern type for indicating architectural problems.

Qualitative Analysis. As described in the design of this study, in addition to providing objective responses related to the occurrence of architectural problems, the participants also provided open responses to justify and explain each analyzed case. Therefore, we conducted a systematic analysis of open responses. In this analysis, two co-authors worked together to categorize the responses and extract recurring codes that help to explain the results. After this procedure, all authors discussed and improved the categorization of responses and the extraction of codes. Such analysis, was inspired on procedures commonly used in qualitative research methods such as Grounded Theory and Content Analysis [24].

Grouping the responses into categories. For the categorization of responses, we defined five categories that reflect relevant scenarios for the evaluation of the use of smell patterns. In the first (C1) and second (C2) categories, we grouped the responses in which participants confirmed the existence

of an architectural problem. However, C1 includes the cases in which the participants agreed that the suggested refactorings could remove the problem. C2, on the other hand, is the case that the suggested refactorings would not completely remove the architectural problem. These two groups help us understand which characteristics are decisive for a set of smells to be considered indicators of architectural problems. Also, we identified the reasons that led to the acceptance or rejection of the suggested refactorings through these two categories.

In the third (C3) and fourth (C4) categories, we included the participants' responses that agreed or not with the suggested refactorings, respectively. However, differently from previous categories, they considered that exists (a certain level of) degradation but not an architectural problem. These two categories allow us to understand when participants consider that a degradation problem is not an architectural one. We also identified the reasons that led to the acceptance or rejection of the suggested refactorings in such cases.

Finally, in the last category (C5), we grouped the responses in which participants considered that there was no occurrence of degradation problems. This category aims to identify which factors lead participants to conclude that the analyzed smells do not represent degradation problems. We also identified what information has been taken into account to rule out the occurrence of degradation problems.

We summarize each category bellow:

- **C1:** There is an architectural problem and the suggested refactorings are accepted.
- **C2:** There is an architectural problem but the suggested refactorings are not accepted.
- **C3:** There is a degradation problem and the suggested refactorings are accepted.
- **C4:** There is a degradation problem but the suggested refactorings are not accepted.
- **C5:** There is no degradation.

Such a categorization helped us to better understand and describe the quantitative results. Therefore, in the next section we will present each qualitative result together with its related quantitative results. The codes and categories created during this analysis will not be directly presented. Nevertheless, they are available in our replication package [44].

6 Evaluation Results: Applying Smell Patterns in Practice

In this section, we present the results of our second study, which is focused on evaluating the use of smell patterns for finding architectural refactoring opportunities. As described in Section 5, we performed a quasi-experiment with 13 professional software developers. We analyzed the results using quantitative and qualitative methods to answer our third research question (RQ3): "To

what extent do smell patterns help developers identify and refactor architectural problems?" We present and discuss the results of RQ3 below.

6.1 Are Smell Patterns Better Indicators of Degradation?

In this study, we asked participants to analyze and classify six cases as being affected by degradation or not – either at the implementation or the architecture level. We discuss in this section to what extent the patterns help developers to identify and remove design degradation.

Table 10 presents the results for all participants. First column presents the identification number (ID) of each participant. Second, third, and fourth columns show the number of Multiple Smells Patterns (MPS) cases, Single Smell Patterns (SSP) cases, and cases with Other Smell Combinations (Others), respectively. The number of cases classified by participants as being affected by degradation problems are presented for each group in the MPS-DP (5^{th}), SSP-DP (6^{th}), and Others-DP (7^{th}) columns. These three columns also show inside the parentheses the precision of participants in each group. The last table row summarizes the number of cases and precision of each group. The remainder columns will be described and explored in the next section.

Table 10 Number of cases and precision of each participant using MPS, SSP and Others.

Id	# of MSP	# of SSP	# of Others	MSP-DP	SSP-DP	Others-DP	MSP-AP	SSP-AP	Others-AP
1	2	2	2	2 (1.00)	2 (1.00)	2 (1.00)	0 (0.00)	0 (0.00)	1 (0.50)
2	2	2	2	2 (1.00)	2 (1.00)	0 (0.00)	2 (1.00)	1 (0.50)	0 (0.00)
3	2	2	2	2 (1.00)	2 (1.00)	2 (1.00)	1 (0.50)	0 (0.00)	2 (1.00)
4	0	1	3	0 (-)	0 (0.00)	1 (0.33)	0 (-)	0 (0.00)	0 (0.00)
5	2	1	3	2 (1.00)	0 (0.00)	2 (0.67)	2 (1.00)	0 (0.00)	2 (0.67)
6	2	2	2	2 (1.00)	2 (1.00)	1 (0.50)	1 (0.50)	0 (0.00)	0 (0.00)
7	2	2	2	0 (0.00)	2 (1.00)	2 (1.00)	0 (0.00)	2 (1.00)	0 (0.00)
8	2	2	2	2 (1.00)	2 (1.00)	1 (0.50)	2 (1.00)	2 (1.00)	0 (0.00)
9	2	2	2	2 (1.00)	0 (0.00)	2 (1.00)	0 (0.00)	0 (0.00)	0 (0.00)
10	2	2	2	1 (0.50)	0 (0.00)	2 (1.00)	0 (0.00)	0 (0.00)	1 (0.50)
11	0	2	4	0 (-)	1 (0.50)	2 (0.50)	0 (-)	0 (0.00)	0 (0.00)
12	2	2	2	1 (0.50)	2 (1.00)	1 (0.50)	1 (0.50)	2 (1.00)	1 (0.50)
13	2	2	2	1 (0.50)	1 (0.50)	0 (0.00)	0 (0.00)	0 (0.00)	0 (0.00)
All	**22**	**24**	**30**	**17 (0.77)**	**16 (0.66)**	**18 (0.60)**	**9 (0.40)**	**7 (0.29)**	**7 (0.23)**

Almost all participants analyzed cases in all groups. Only participants 4 and 11 did not analyze cases in all groups. The reason is because these participants analyzed cases in small software projects; thus, our tool did not find any MSP case.

Precision with MSP is slightly higher. When analyzing the precision of participants in the different groups, it is possible to observe that the use of MSP tends to result in a higher precision when compared to SSP and Others. Considering all cases and all participants, we observed an overall precision of 0.77 for MSP, 0.66 for SSP, and 0.60 for Others.

This result is an initial evidence that MSP can contribute to the identification of degradation problems with higher precision than other combinations

of smells. Similarly to what we observed in the multi-case study (Section 4), we also found evidence in this study that patterns with the *God Class* and *Complex Class* smells are often reliable indicators of degradation problems. Moreover, in this specific quasi-experiment, we observed that the combination of *Shotgun Surgery*, *Feature Envy*, and *Intensive Coupling* in a hierarchy of classes is a reliable indication of degradation. We will further discuss such smell types latter in this section.

There is no statistically significant difference between the groups' precision. We applied the One-Way Repeated Measures ANOVA Test to compare the precision of participants in the different groups. This statistical test is useful for comparing the use of different treatments by the same group of participants. In this evaluation, the treatments are the use of MSP, SSP, or other smell combinations for identifying any form of degradation. For applying the test, we had to remove the participants 4 and 11 as they did not use all treatments. After applying the statistical test with an alpha level of *.05*, we observed a F-ratio of *0.24232* and a p-value of *.787072*. The p-value is higher than our alpha level (.05), indicating that our results are not statistically significant. The F-ratio is smaller than 1, which indicates that the use (or not) of smell patterns alone is not enough for explaining the variance of precision in the identification of degradation problems.

Upon data analysis, we found multiple factors that influence on the precision of a smell-based approach. Next, we describe the most recurrent factors.

Context sensitive degradation detection. A recurrent observed factor is that certain degradation problems were considered inevitable in certain contexts. For example, participant 13 informed that the degradation indicated by the *Message Chain* and *Long Parameter List* smells is common and acceptable in Android applications. Another example is the occurrence of the *Data Class* smell – a single smell pattern used to indicate an *Incomplete Abstraction*, which was considered acceptable by the participants in object-relational mapping classes (a.k.a. entity classes).

Use of customized strategies and thresholds. For the detection of code smells, we adopted default detection strategies and thresholds from the literature [23]. However, such strategies and thresholds were not effective in multiple cases. For instance, we observed multiple cases of *Long Parameter List* for which the participants did not agree with the number of parameters threshold. There were also cases of *Speculative Generality* for which the participants argued that the abstract methods of a abstract class were used in a sufficient number of sub-classes. In fact, this observation is corroborated by other studies (e.g., [19]) that indicate the need for customized detection strategies for code smells.

There should be more support for refactoring complex problems. Finally, in some cases participants agreed with the existence of degradation problems, but argued that refactoring such problems would require an effort that is not worth. Thus, they decided to classify such cases as non-degraded. Based on this result, we hypothesize that a better tool support could encourage developers to conduct complex refactorings for removing the problems. In this

study, the tool only provided a list of recommended refactorings for each case. A better tool support would help developers to perform complex architectural refactorings automatically.

In summary, we conclude that smell patterns have the potential to help developers in finding degradation problems, specially the MPS. However, other factors must be considered to achieve a higher precision. Next, we present the results on the use of patterns for finding only architectural problems.

6.2 When are Smell Patterns Indicators of Architectural Problems?

In the previous section, we present the results on the use of smell patterns for finding degradation problems. However, our goal here is to verify if the smell patterns can help developers identify architectural degradation problems. Thus, in this section, we present an assessment about when smell patterns are indicators of architectural problems.

The numbers of cases classified by participants as being affected by architectural problems are presented for each group in the MPS-AP (8^{th}), SSP-AP (9^{th}), and Others-AP (10^{th}) columns of Table 10. Such columns also show, inside parentheses, the precision of participants in each group for identifying architectural problems. In the last line of the table, we present the total number of cases and the overall precision for each group.

The group with the highest precision was MSP (0.40), followed by SSP (0.29) and Others (0.23). Thus, the participants' precision in finding architectural problems was much smaller than their precision in finding any types of degradation problems, regardless of the use of patterns. This happened because in multiple cases the participants considered that there was a degradation problem; however, this problem was not relevant to the architecture of the software. According to them, they considered that exists (a certain level of) degradation but not an architectural problem.

In order to search for a possible association of the occurrence of patterns with the existence of architectural problems, we applied the Chi-Square Test. Table 11 presents the continence table for this test. Second column presents, for each group (MSP, SSP, and Others), the number of cases that were classified as having implementation problems – i.e., degradation problems that are not related to the architecture. Third column shows the number of cases classified as having architectural problems. Forth column presents the number of cases classified as not having any kind of degradation. In the last column we show the total number of cases in each group. Last row shows the totals of each column.

In the cells of second, third, and forth columns of Table 11, we also show, inside parenthesis, the expected distributions according to the totals in each row and column. Values in square brackets represent how much each cell contributed to the Chi-Square statistic. The higher the difference between the expected value and the observed value, the higher the contribution to the Chi-Square statistic. For instance, the expected number of cases with architectural

problems for the MSP group is *6.66*, but the actual number of cases were *9*, resulting in a contribution of *0.82* to the Chi-Square statistic.

Table 11 Results of the Chi-Square Test for the association of groups with architectural problems, implementation problems and no problems.

	# Architectural Problems	# Implementation Problems	# of Non-degradations	Row Total
Multiple Smells Patterns	9 (6.66) [0.82]	8 (8.11) [0.00]	5 (7.24) [0.69]	22
Single Smell Patterns	7 (7.26) [0.01]	9 (8.84) [0.00]	8 (7.89) [0.00]	24
Other Combinations	7 (9.08) [0.48]	11 (11.05) [0.00]	12 (9.87) [0.46]	30
Column Total	23	28	25	76

Correlation between patterns and architectural problems. The highest differences were observed in the second (cases with architectural problems) and fourth (cases without degradation) columns of Table 11 for the MSP and Others groups. The number of MSP cases with architectural problems is higher than the expected, while the number of Others cases with architectural problems is lower than the expected. However, by applying the Chi-Square test with an alpha level of *.50*, we obtained a Chi-Square statistic of *2.4671* and a p-value of *.65053*. Thus, we cannot reject the hypothesis that there is no correlation between the evaluated patterns and the existence of architectural or implementation problems. Given such lack of correlation, in the following sub-sections, we present qualitative analysis that help us to understand and explain the results.

6.2.1 What is Really an Architectural Problem?

We observed contradictory cases that were classified as not being affected by architectural problems, but presented justifications describing exactly architectural problems. For example, when analyzing the Couchbase Java Client project, participant 10 classified one case involving a MSP as not having any kind of degradation. The justification for such classification was the following:

> *The class does have methods that use cross-cutting functionalities, but this is necessary due to the nature of the software. It is a class that represents a Bucket (abstraction that would be equivalent to a database in other non-relational databases), it is justifiable to provide multiple functionalities through this class, which serves as an interface between the user source code and the database.*

In the aforementioned case, the MPS was intended to indicate a *Scattered Concern* problem involving the *CouchbaseAsyncBucket* class through the following combination of smells: *Dispersed Coupling*, *Feature Envy*, *God Class*, *Complex Class*, and *Intensive Coupling*. According to the open response presented by the participant, such a class would in fact be affected by a functionality (concern) that is scattered across different classes.

Since this system is a popular and well-structured open source project, we can conjecture that the decision to keep the functionality scattered may have been conscious indeed. However, this does not mean that there is no architectural problem, since a scattered functionality may affect the extensibility and modifiability of the software project. Creating abstractions to isolate a feature can make the software design more complex. In this sense, we believe that, in this case, developers may have taken the decision to favor code design simplicity over other quality attributes.

We also observed cases involving patterns that were classified as implementation problems but their description indicated architectural problems. For instance, when relying on a MSP to analyze the Facebook Fresco library, participant 3 indicated the existence of an implementation problem in the *FrescoController2* Java interface and its implementations. This participant provided the following description for this problem:

> *The classes are very dependent and coupled with the FrescoDrawback2 class. All of these classes and methods need to call various methods of that class to meet their responsibilities.*

In this case, the MPS was intended to indicate the *Fat Interface* problem, which occurs when an interface exposes many functionalities and many of those functionalities are not related to each other. The fact that one class is tightly coupled to another does not necessarily mean that there is a *Fat Interface* problem. However, high coupling is a negative characteristic that directly affects the architecture and may be indirectly related to the existence of a *Fat Interface*.

We believe that participant 3 may have initially conducted a shallow analysis of the case, without considering a possible impact on the architecture. Nevertheless, we believe that in this case, information about metrics and smells assisted the participant to unconsciously identify an architectural problem. Even though the participant thought that there was no architectural problem, a degradation problem related to high coupling was detected. The participant identified the problem through the combination of the smells and the presented system metrics.

On our qualitative analysis, we identified how participant 3 proceeded with the analysis. In this case, the participant identified that (i) there was feature envies in some methods, (ii) the class had a high complexity (being a *Complex Class* smell), and, through the metrics, (iii) the high coupling of that class. This information helped in the identification of a degradation problem, even if the participant does not have full knowledge about the architectural impact of such a problem.

Given the aforementioned analysis, we can consider that the use of smells and metrics can help less experienced developers in relation to architectural problems. In other words, this can be a starting point for identifying that there is a degradation problem, and in further analysis, identify and remove the architectural problem. Therefore, to assist in the complete identification

and removal of the architectural problem, the information provided by the patterns should be complemented by other information, such as the concerns that are implemented in each class. In this specific case of participant 3, the information about concerns would be useful for identifying the presence of multiple unrelated services.

Unfortunately, reliably identifying concerns in source code is not an easy task. Performing a manual mapping of concerns is costly and error prone. In fact, there are automated techniques to perform such a mapping [40]. However, there are limitations with such techniques. First, they are not able to provide a description for each concern that is easily understood by humans. In addition, the mapping would need to be updated whenever new changes were made to the source code.

Other cases similar to those described above happened with different participants. This indicates that the developers' understanding of what an architectural problem may not be entirely correct and accurate. We conjecture that many developers only consider architectural problems to be those that were not intentionally introduced and those that explicitly affect many components of the architecture. This shows that, although smell patterns are able to indicate the existence of architectural problems in relevant scenarios, they still do not provide enough information to make evident the causes and the possible impact caused by the detected architectural problems. Thus, in future studies we intend to investigate effective ways to combine patterns with complementary information, such as the mapping of concerns. These results leads to our fifth finding:

Finding 5: Developers perceive architectural problems as those which they did not intentionally introduce and that explicitly affect many components of the architecture

6.2.2 Evaluating Specific Smell Patterns

To better understand when smell patterns are indicators of architectural problems, we conducted an analysis of individual types of patterns. Table 12 presents the results for the assessed pattern types. Second column shows the number of evaluated cases of each pattern type. In the third and fourth columns, we show the number of cases with any kind of degradation problem and with only architectural problems, respectively. In both columns, we show inside parenthesis the precision of each pattern type. Finally, fifth column presents the mean severity – in a scale from 1 to 5, according to the classifications provided by participants.

Unused Abstraction pattern as an indicator of implementation problems. The pattern type for *Unused Abstraction* presented the highest number of evaluated cases (17) and a precision of *0.76* for finding degradation problems. However, its precision for finding architectural problems was of only *0.29*. This result shows that the Unused Abstraction pattern was consistently

Table 12 Number of cases, precision and mean severity of each pattern type.

Pattern Type	# of Cases	# Degradation	# Arch. Problems	Mean Severity
Multi-Smell Patterns				
Concern Overload	3	2 (0.66)	1 (0.33)	4.50
Fat Interface	6	6 (1.00)	3 (0.50)	3.50
Scattered Concern	6	5 (0.83)	2 (0.33)	4.60
Unwanted Dependency	7	4 (0.57)	3 (0.42)	3.25
Single-Smell Patterns				
Incomplete Abstraction	6	3 (0.50)	2 (0.33)	3.00
Unused Abstraction	17	13 (0.76)	5 (0.29)	3.38

considered relevant to indicate the presence of degradation problems. However, on the other hand, the participants considered that such a problem is not relevant to the software architecture. Thus, we have evidence that *Unused Abstraction* is often considered by developers as an implementation problem. This observation becomes more evident by the fact that the mean perceived severity for *Unused Abstraction* was only *3.38*.

Indirect detection of Unwanted Dependencies. An *Unwanted Dependency* occurs when a dependency violates a rule defined on the system architecture [48]. The precision of our *Unwanted Dependency* pattern was *0.57* for degradation problems and *0.42* for architectural problems. Given the lack of information about architectural rules, we believe that such precision was higher than expected. Unfortunately, architectural rules are not always defined and documented. In such cases, the pattern would be useful for the indirect detection of *Unwanted Dependencies*.

Incomplete Abstraction is highly sensitive to contextual factors. The *Incomplete Abstraction* pattern presented the least satisfactory results in terms of precision and severity. In cases involving such a pattern, we observed that participants frequently mentioned contextual factors to justify the fact that this pattern does not represent any kind of design degradation. The *Incomplete Abstraction* pattern is composed by the *Lazy Class* smell, which is intended to indicate a class that is too small to exist. Indeed, there is a fine line that separates what is a well modularized class and what is a too small class. Therefore, we believe that this pattern could present better results if the detection of *Lazy Class* were customized according to the developer and the project being analyzed [19]. These results lead to our next finding:

> **Finding 6**: The usefulness of the Unused Abstraction, Unwanted Dependencies, and Incomplete Abstraction patterns is limited in the context of architectural refactoring opportunities. Nevertheless, such patterns can be useful in specific contexts.

High Precision for the Fat Interface pattern. The pattern type with highest precision was the one for *Fat Interface*, with *1.00* for degradation problems and *0.50* for architectural problems. As described in Section 4.4, our pattern for *Fat Interface* is composed by a *Shotgun Surgery* in the interface or *Dispersed Coupling* and *Feature Envy* in elements that are clients or implement the interface. We believe that one of the reasons for the success of this pattern

is the fact that it makes clear the impact on hierarchical structures composed of interfaces and classes. In addition, smells like *Dispersed Coupling* and *Feature Envy* are strongly associated with the architectural concepts of high coupling and low cohesion. Although other patterns are also composed of smells that can reveal a relevant impact for the architecture, the pattern for *Fat Interface* makes it clear that multiple interrelated code elements are affected.

Results for the *Fat Interface* pattern provide evidence that identifying architectural problems requires more detailed information in addition to the description of metrics and smells. In fact, participant 12 informed us in the post-experiment interview that, besides information about smells, the tool should provide explicit information about the dependencies that involve the affected code elements.

High severity for Concern Overload and Scattered Concern patterns. Participants classified cases involving the *Concern Overload* and *Scattered Concern* patterns with the highest severity. The mean perceived severity was *4.50* for *Concern Overload* and *4.60* for *Scattered Concern*. This indicates that, even though they were not classified as architectural problems in many cases, these patterns were considered to be highly severe. Both patterns involve the *God Class* and *Complex Class* smells. In our case study (Section 4), we observed such smells to be recurrent indicators of architectural refactoring opportunities. Thus, the high severity observed in this experiment is consistent with the results observed in the case study. As we discussed earlier, the precision of these patterns could be higher if we had provided additional information about the dependencies and concerns of each code element. The results for these patterns lead us to our seventh finding:

> **Finding 7**: The Fat Interface, Concern Overload and Scattered Concern patterns are the most promising for identifying architectural refactoring opportunities. They may be even more useful and precise if complemented with information about dependencies and concerns.

6.3 Code smells as Indicators of Refactoring Opportunities

Besides presenting information on metrics and code smells, we also suggested refactorings for each smell (Table 6). Thus, we asked the participants if the suggested refactorings were sufficient to remove the identified degradation problems. With their answer, we aimed at understanding the scenarios where developers accepted, partially accepted, or rejected the suggested refactorings. Table 13 summarizes their answers.

For Single Smell Patterns (SSP), in most of the cases, participants either accepted or partially accepted the refactorings. The main reasons for the partially accepted refactorings were (i) the effort needed to conduct the suggested refactorings, and (ii) not fully understanding the design of the software project. In those cases, the architectural problem was *Unused Abstraction*. For

such a problem, the participant would have to understand where the abstraction could be used. Therefore, a deeper knowledge about the system design would be necessary. To illustrate this recurrent observation, we may take as an example the justification of participant 2 for one of the cases:

> *The refactoring suggestion, as useful as it may be, may not be just what is necessary to solve the problem. It potentially needs a reevaluation of the design itself to discover the real use of that abstract class, and because no other class uses it.*

We observed that even if the developer understands the design of the project, the refactorings can be difficult. As suggested by one of the participants, the refactorings indeed would reduce the complexity of a class, however with a high effort cost. For that case, even though he recognized that there is an architectural problem, the participant thought that the trade-off between the class complexity and the effort was not worthwhile.

Besides analyzing the reject and partially accepted refactorings, we also analyzed the cases when the participants accepted the suggested refactorings. In such cases, they usually mentioned that the smells and metrics were useful for their analysis. They even mentioned that the metrics gave them a better overview of the analyzed code elements. Finally, we noticed that the refactorings for simpler smells were more widely accepted. By simpler, we mean those smells that are easier to understand or at least, those that participates are familiar. Examples of such smells are *Long Method* and *Long Parameter List*.

Table 13 Number of refactoring suggestions that were accepted, partially accepted, or rejected.

	SSP	**MSP**	**Others**
Accepted	9	6	9
Partially Accepted	6	11	6
Rejected	1	0	2

We also analyzed the cases involving Multiple Smell Pattern (MSP). As presented in Table 13, the participants partially accepted most of the suggested refactorings for MSP cases. Among the causes for them partially accepting the refactorings, we observed (i) the complexity of the degradation problem, (ii) the use of design patterns, and (iii) the context of the software project.

Participants mentioned, for example, that multiple code elements with smells were presented, and that made the analysis difficult. One of the obstacles was how to relate the smells and metrics of different code elements to identify the main problem. For that matter, they accepted only some refactorings since they were not fully aware of the whole architectural problem affecting these classes. Another cause for partially accepting the refactorings was related to the possible modification related to a higher level abstraction. In these cases, the participants mentioned that instead of only applying the

refactorings they would also make more massive changes, such as introducing design patterns.

One of the reasons mentioned by the participants was the context of the projects. Even though there was an architectural problem, they mentioned that this was needed due to the APIs used on the system, for example. Hence, they accepted only some refactorings, since other cases of the smell were required for API usage. Following, we present an example of this scenario involving participant 10:

> *This is a library that serves as an API for communication with a database and this is a central abstraction of the database. Thus, it is justified that this class adds a high amount of functionality so that (i) this abstraction is not spread over many classes, and (ii) API usability is also facilitated.*

When the participants accepted all suggested refactorings, we observed that they used our tool to identify problem that at a glance could be considered as complex. For instance, participant 1 mentioned that some problems were hard to identify due to the complexity of the project. However, using the provided source code, smells and metrics, the participant managed to identify and fully understand the problem.

In addition, in the acceptance cases, some participants also mentioned the time constraints for the project development. Due to such constraints, they recognized the introduction of implementation and architectural problems during the development and evolution of the source code. For example, participant 5 provided the following justification in one of the cases:

> *Due to development time, I simplified this implementation, adopting conditionals, instead of using polymorphism.*

As reported by the participant, this simplification led the class to become a *Complex Class* and have methods with *Message Chains*, *Long Methods*, and *Feature Envies*. Combined with other smells, they were indicators of the *Unwanted Dependency* problem.

Now let us consider the cases involving other combinations of smells (Others). In this case, we noticed that the participants do not accepted the refactorings especially due to design decisions. For instance, as presented below, participant 10 mentioned that even though the class analyzed was indeed big and complex, it was an intentional decision.

> *The methods of the analyzed class are difficult to read. However, the code writing style (causing message chains) is justified by the transactional nature inherent in the database domain. I believe that, in the case of this specific domain, refactorings would only worsen the readability of the code.*

In the cases where the participant partially accepted the refactoring, we noticed that this is related to the complexity of the problem. For instance, participant 3 suggested the implementation of a design pattern to start solving

the problem. This was suggested since the analyzed class was doing duplicated functions. To implement this pattern, the participant suggested applying *Extract Class* to create new separated classes. In this case, refactorings suggested to remove *long parameter lists* and *long methods* would not be enough. Similar to the case where they do not accept the refactorings, the domain of the system was also impacted when they only partially accepted the refactorings.

Based on the aforementioned results, we conclude that the refactorings associated with code smells can contribute to the removal or, at least, the partial removal of degradation problems. Nevertheless, removing architectural problems tends to be more challenging. We have identified multiple scenarios in which more complex refactorings (e.g., introduction of a design pattern) are required. Therefore, we conclude that our smells-based approach could be more useful if it was able to provide automated support for performing the suggested refactorings. To assist in removing architectural problems, we believe that the suggested refactorings could be customized and optimized using machine learning or search-based algorithms [2]. This discussion leads to our last finding:

Finding 8: Refactorings associated with code smells contribute to the (partial) removal of degradation problems. However, more complex problems, such as the architectural ones, require a better support.

6.4 Threats to Validity

The time allocated for analyzing the cases can pose a threat to the validity of this quasi-experiment. To mitigate this threat, we performed a pilot study to adequate the number of cases and the time that would be spent on analyzing each case. We also present the cases of different groups in random order to avoid that cases of a certain group always remain at the beginning or at the end of the experiment. Finally, we recorded the time spent analyzing each case and asked the participants if they felt tired during the experiment. The analysis of each case lasted an average of about 11 minutes. Most participants reported that they did not feel tired. Two participants asked to partition the experiment in two parts to prevent tiredness.

As the experiment was carried out remotely, it was not possible to control external variables such as interruption by third parties or the occurrence of technical problems. We mitigate this threat by asking each participant to report the occurrence of any interruptions or technical problems. Two participants suffered interruptions caused by third parties and one of the participants' internet went down for 10 minutes. Nevertheless, the participants informed us that such problems did not hinder their performance during the experiment.

The number of participants represents another threat to validity. We would need a larger sample of participants to achieve statistically significant results. To mitigate this threat, we complemented our quantitative analysis with a

systematic qualitative analysis. In fact, qualitative research requires the study of specific situations and people, complemented by considering specific contextual conditions [71]. We selected 13 professional developers with diverse background and experience, which are representative individuals of our target population. Thus, we consider that this threat was properly mitigated.

As we use an automated tool to collect metrics, detect smells and find patterns, our results are influenced by the accuracy and reliability of the tool. To mitigate this threat, we extended a tool that was extensively used in previous studies for collecting metrics and detecting code smells (e.g., [8,59,40]). We also conducted manual tests with multiple open source projects to identify and remove possible defects in the tool. Finally, we conducted a qualitative assessment that helped us to identify cases in which the accuracy of the tool influenced the results.

7 Related Work

This paper is an extension of a previous work [58] in which we investigated when smells are indicators of architectural refactoring opportunities. We analyzed 52,667 refactorings from 50 open source projects. The main contribution of this previous work was the identification of smell patterns that are often associated with architectural problems. In this paper, we complement our previous work by conducting an evaluation on the use of smell patterns in practice. Our evaluation was based on a quasi-experiment involving 13 professional software developers. Our quasi-experiment helped us to find several new findings. For example, we investigated whether the patterns are more often associated with architectural problems or with implementation problems. In addition, we identified when patterns are actually useful in practice. We have also identified several factors that must be taken into account in order for developers to be more precise in identifying architectural problems through code smells. Next, we discuss other studies that are closely related to this work.

7.1 Code Smells and Degradation Problems

Code smells have been a well-researched topic over the last decade [28,31,37, 36,40,66,43,65]. For example, Tufano *et al.* [65] 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 [46,69, 65]. Yamashita and Moonen conducted an exploratory survey with developers about their knowledge and concern with code smells [69]. The authors investigated through developers' perspective if code smells should be considered

meaningful conceptualization of degradation 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 degradation problems [46]. 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 degradation problems. Our results are consistent with these studies. We showed that smells like *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 degradation problems.

Fontana *et al.* [3] conducted an empirical study to evaluate if architectural smells are independent from code smells. Their results indicate that there is no strong correlation between architectural smells and code smells. However, they used an automated tool for detecting architectural smells. Thus, their architectural smells do not necessarily represent architectural problems, as there can be false positives and false negatives. Differently from them, we evaluated the use of a smell-based approach with a multi-case study and a quasi-experiment. In the former, we used root-canal refactorings as indicators of architectural problems, while, in the latter, we relied on the expertise of professional developers. Therefore, our work evaluates the relationship between smells and architectural problems from a different perspective.

7.2 Applying Refactorings

Silva, Tsantalis and Valente [56] investigated the reasons why developers refactor their code. They identified refactoring operations in 748 Java projects in the GitHub repository. Then, they asked developers why they performed the

refactoring operations. Their results indicate that refactoring operations are mainly driven by fixing a bug or changing the requirements, such as adding a new feature, and much less by code smells. Although developers did not mention smells explicitly as their intention to refactor, their results show that the refactored code may contain code smells that may have been the motivation for refactoring. For instance, developers said that they apply the *Move Class* refactoring when they want to move a class to a package that is more functionally or conceptually relevant to the class. Even though developers did not mention smells in this description, it is possible that the moved class had smells such as *Feature Envy* and *Intensive Coupling*. These results suggest that developers are also motivated (at least implicitly) by smells, or ultimately by architectural problems, which is aligned with our results. Our data showed that most refactoring operations (79.48%) are applied to smelly elements. Even though the original motivation of developers is unknown, we actually observed a similar behavior when they applied both root-canal and floss refactoring. Therefore, they might not be applying refactoring to remove implementation problems or architectural problems; nonetheless, they are still applying refactoring to elements that present signs of degradation. As we mentioned before, developers are not removing these architectural problems.

Bavota *et al.* [5] investigated whether refactorings occur in elements in which certain indicators suggest that might be a need for refactoring. Their indicators include structural quality metrics and the presence of code smells. According to their results, quality metrics do not show a clear relationship with refactoring, and only 42% of the refactorings are applied to smelly elements – in contrast with the 79.48% that we found. Different from them, with a multi-case study, we investigated a large set of software systems, and we also performed the validation and classification of a subset of refactorings. Moreover, we have considered their 11 types of code smells plus 6 other types of code smell deemed relevant in the literature. Thus, our data sample is much larger than the sample analyzed by them.

Cedrim *et al.* [8] investigated the frequency that refactoring operations are applied to smelly elements. They found that almost 80% of refactoring operations are applied to smelly elements. Even though, we found a similar result, the studies are completely different. In their study, the authors focused on the relation between smells and refactoring. We focus on the relation between smells and architectural problems, in which we conveniently use refactoring to find relevant architectural problems. Additionally, we analyzed more than twice as many projects, and we also considered more smells and refactoring types.

Additionally, differently from Bavota *et al.* and Cedrim *et al.*, we focused on analyzing the root-canal refactorings. This refactoring tactic focuses on the design structure improvement; thus, this tactic has a high chance to target architectural problems. Considering root-canal refactoring, we could also better explain the relation of smells and architectural problems. Finally, differently from them, we conducted a quasi-experiment to assess the usefulness of a smell-based approach for finding architectural refactoring opportunities.

7.3 Identification of Architectural Problems

Studies have investigated approaches to identify architectural problems [35, 67]. Mo *et al.* [35] 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.* [67] 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 [59]. Sousa *et al.* [59] 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 opportunity.

7.4 Investigation of Architectural Refactoring

Few studies have investigated architectural refactorings [25, 51, 22, 72, 49]. For example, Kumar and Kumar [22] 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.* [25] 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 step-wise code refactorings that will change the source code to meet the target architecture.

Zimmermann [72] positioned architectural refactoring as a task-centred 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 [51] proposed a tool prototype to remove the *Cyclic Dependency* problem. Their tool suggests the refactorings steps that the developer should follow to remove the *Cyclic Dependency*. Recently, Rachow [49] 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. Al-

though his framework has not been implemented, the author indicates seven architectural problems that the framework will provide support.

Differently from the aforementioned studies, we investigated the use of a smell-based approach for finding architectural refactoring opportunities. Besides evaluating the use of a smell-based approach, we also developed an automated tool that is fully available to be extended and used by other researchers and developers [4].

8 Conclusion

We conducted a multi-case 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 smell-based approach for finding architectural refactoring opportunities. This approach relies on a set of smell patterns that are very likely to indicate the presence of architectural problems. We are the first one to catalog these patterns.

To assess the use of smell patterns in practice, we conducted a quasi-experiment with 13 professional developers. Each experiment's participant evaluated multiple cases of smell patterns and other combinations of smells regarding their relation with design degradation problems. This evaluation was conducted in the context of software projects they were familiar with. The participants also indicated whether the detected degradation problems were at the architectural or implementation level. Finally, they indicated whether the refactorings associated with the smells would be sufficient to remove each degradation problem.

Based on both studies, 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*, *Scattered Concern*, and *Fat Interface*. We also observed that a smell-based approach may be improved by exploring additional information, such as the mapping of concerns and the graph of dependencies of each code element.

[4] The source code is available at https://github.com/wnoizumi/degradation-experiment

Finally, we observed that several factors influence in the developers' decision to classify a set of code elements as being affected by architectural problems or not. Examples of such factors are: development platform (e.g., Android), type of functionality, and effort for refactoring the problem.

As future work, we intend to use the findings presented in this paper to improve our smell-based approach. We also intend to conduct a deep evaluation of the improved approach with new empirical studies.

Acknowledgements This work is funded by CNPq (grants 434969/2018-4, 312149/2016-6), CAPES (grant 175956), and FAPERJ (grant 22520-7/2016).

Conflict of interest

The authors declare that they have no conflict of interest.

References

1. Abbes, M., Khomh, F., Gueheneuc, Y., Antoniol, G.: 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, pp. 181–190 (2011)
2. Amal, B., Kessentini, M., Bechikh, S., Dea, J., Said, L.B.: On the use of machine learning and search-based software engineering for ill-defined fitness function: A case study on software refactoring. In: C. Le Goues, S. Yoo (eds.) Search-Based Software Engineering, pp. 31–45. Springer International Publishing, Cham (2014)
3. Arcelli Fontana, F., Lenarduzzi, V., Roveda, R., Taibi, D.: Are architectural smells independent from code smells? an empirical study. Journal of Systems and Software **154**, 139 – 156 (2019). DOI https://doi.org/10.1016/j.jss.2019.04.066. URL `http://www.sciencedirect.com/science/article/pii/S0164121219301013`
4. Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice. Addison-Wesley Professional (2003)
5. Bavota, G., Lucia, A.D., Penta, M.D., Oliveto, R., Palomba, F.: An experimental investigation on the innate relationship between quality and refactoring. Journal of Systems and Software **107**, 1 – 14 (2015). DOI http://dx.doi.org/10.1016/j.jss.2015.05.024. URL `http://www.sciencedirect.com/science/article/pii/S0164121215001053`
6. Bertran, I.M.: Detecting architecturally-relevant code smells in evolving software systems. In: Proceedings of the 33rd International Conference on Software Engineering, ICSE '11, pp. 1090–1093. ACM, New York, NY, USA (2011). DOI 10.1145/1985793.1986003. URL `http://doi.acm.org/10.1145/1985793.1986003`
7. Budd, T.A.: An Introduction to Object-Oriented Programming, 3rd edn. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2001)
8. Cedrim, D., Garcia, A., Mongiovi, M., Gheyi, R., Sousa, L., de Mello, R., Fonseca, B., Ribeiro, M., Chávez, A.: 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, pp. 465–475. ACM, New York, NY, USA (2017). DOI 10.1145/3106237.3106259. URL `http://doi.acm.org/10.1145/3106237.3106259`
9. Ciupke, O.: Automatic detection of design problems in object-oriented reengineering. In: Proceedings of Technology of Object-Oriented Languages and Systems - TOOLS 30 (Cat. No.PR00278), pp. 18–32 (1999)

10. Curtis, B., Sappidi, J., Szynkarski, A.: Estimating the size, cost, and types of technical debt. In: Proceedings of the Third International Workshop on Managing Technical Debt, MTD '12, pp. 49–53. IEEE Press, Piscataway, NJ, USA (2012). URL http://dl.acm.org/citation.cfm?id=2666036.2666045
11. Eick, S.G., Graves, T.L., Karr, A.F., Marron, J.S., Mockus, A.: Does code decay? assessing the evidence from change management data. IEEE Trans. Softw. Eng. **27**(1), 1–12 (2001). DOI 10.1109/32.895984. URL https://doi.org/10.1109/32.895984
12. Ernst, N.A., Bellomo, S., Ozkaya, I., Nord, R.L., Gorton, I.: 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, p. 50–60. Association for Computing Machinery, New York, NY, USA (2015). DOI 10.1145/2786805.2786848. URL https://doi.org/10.1145/2786805.2786848
13. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley Professional, Boston (1999)
14. Garcia, J., Popescu, D., Edwards, G., Medvidovic, N.: Identifying architectural bad smells. In: CSMR09; Kaiserslautern, Germany. IEEE (2009)
15. Garcia, J., Popescu, D., Edwards, G., Medvidovic, N.: Toward a catalogue of architectural bad smells. In: R. Mirandola, I. Gorton, C. Hofmeister (eds.) Architectures for Adaptive Software Systems, pp. 146–162. Springer Berlin Heidelberg, Berlin, Heidelberg (2009)
16. Godfrey, M., Lee, E.: Secrets from the monster: Extracting Mozilla's software architecture. In: CoSET-00; Limerick, Ireland, pp. 15–23 (2000)
17. Greenwood, P., Bartolomei, T., Figueiredo, E., Dosea, M., Garcia, A., Cacho, N., Sant'Anna, C., Soares, S., Borba, P., Kulesza, U., Rashid, A.: 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, pp. 176–200. Springer-Verlag, Berlin, Heidelberg (2007). URL http://dl.acm.org/citation.cfm?id=2394758.2394771
18. van Gurp, J., Bosch, J.: Design erosion: problems and causes. Journal of Systems and Software **61**(2), 105 – 119 (2002)
19. Hozano, M., Garcia, A., Antunes, N., Fonseca, B., Costa, E.: Smells are sensitive to developers! on the efficiency of (un)guided customized detection. In: 2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC), pp. 110–120 (2017). DOI 10.1109/ICPC.2017.32
20. Kim, M., Zimmermann, T., Nagappan, N.: A field study of refactoring challenges and benefits. In: Proceedings of the 20th International Symposium on the Foundations of Software Engineering, FSE '12, pp. 50:1–50:11. ACM, New York, NY, USA (2012). DOI 10.1145/2393596.2393655. URL http://doi.acm.org/10.1145/2393596.2393655
21. Kim, M., Zimmermann, T., Nagappan, N.: An empirical study of refactoringchallenges and benefits at microsoft. IEEE Transactions on Software Engineering **40**(7), 633–649 (2014)
22. Kumar, M.R., Kumar, R.H.: Architectural refactoring of a mission critical integration application: A case study. In: Proceedings of the 4th India Software Engineering Conference, ISEC '11, p. 77–83. Association for Computing Machinery, New York, NY, USA (2011). DOI 10.1145/1953355.1953365. URL https://doi.org/10.1145/1953355.1953365
23. Lanza, M., Marinescu, R.: Object-Oriented Metrics in Practice. Springer, Heidelberg (2006)
24. Lazar, J., Feng, J.H., Hochheiser, H.: Research methods in human-computer interaction. Morgan Kaufmann (2017)
25. Lin, Y., Peng, X., Cai, Y., Dig, D., Zheng, D., Zhao, W.: 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, p. 535–546. Association for Computing Machinery, New York, NY, USA (2016). DOI 10.1145/2950290.2950317. URL https://doi.org/10.1145/2950290.2950317
26. Lippert, M., Roock, S.: Refactoring in Large Software Projects: Performing Complex Restructurings Successfully. Wiley (2006). URL https://books.google.com.br/books?id=bCEYuB83ROcC

27. MacCormack, A., Rusnak, J., Baldwin, C.: Exploring the structure of complex software designs: An empirical study of open source and proprietary code. Manage. Sci. **52**(7), 1015–1030 (2006)
28. Macia, I.: On the detection of architecturally-relevant code anomalies in software systems. Ph.D. thesis, Pontifical Catholic University of Rio de Janeiro, Informatics Department (2013)
29. Macia, I., Arcoverde, R., Cirilo, E., Garcia, A., von Staa, A.: Supporting the identification of architecturally-relevant code anomalies. In: ICSM12, pp. 662–665 (2012)
30. Macia, I., Arcoverde, R., Garcia, A., Chavez, C., von Staa, A.: On the relevance of code anomalies for identifying architecture degradation symptoms. In: CSMR12, pp. 277–286 (2012)
31. Macia, I., Garcia, J., Popescu, D., Garcia, A., Medvidovic, N., von Staa, A.: Are automatically-detected code anomalies relevant to architectural modularity?: An exploratory analysis of evolving systems. In: AOSD '12, pp. 167–178. ACM, New York, NY, USA (2012)
32. Marinescu: Detection strategies: metrics-based rules for detecting design flaws. In: Proceedings of 20th IEEE International Conference on Software Maintenance (ICSM); Chicago, USA, pp. 350–359 (2004)
33. Martin, R.C., Martin, M.: Agile Principles, Patterns, and Practices in C# (Robert C. Martin). Prentice Hall PTR, Upper Saddle River, NJ, USA (2006)
34. Martins, J., Bezerra, C., Uchôa, A., Garcia, A.: Are code smell co-occurrences harmful to internal quality attributes? a mixed-method study. In: Proceedings of the 34th Brazilian Symposium on Software Engineering, SBES '20, p. 52–61. Association for Computing Machinery, New York, NY, USA (2020). DOI 10.1145/3422392.3422419. URL `https://doi.org/10.1145/3422392.3422419`
35. Mo, R., Cai, Y., Kazman, R., Xiao, L.: Hotspot patterns: The formal definition and automatic detection of architecture smells. In: Software Architecture (WICSA), 2015 12th Working IEEE/IFIP Conference on, pp. 51–60 (2015)
36. Moha, N., Gueheneuc, Y., Duchien, L., Meur, A.L.: Decor: A method for the specification and detection of code and design smells. IEEE Transaction on Software Engineering **36**, 20–36 (2010)
37. Moha, N., g. Gueheneuc, Y., Leduc, P.: Automatic generation of detection algorithms for design defects. In: 21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06), pp. 297–300 (2006). DOI 10.1109/ASE.2006.22
38. Murphy-Hill, E., Parnin, C., Black, A.P.: How we refactor, and how we know it. In: Proceedings of the 31st International Conference on Software Engineering, ICSE '09, pp. 287–297. IEEE Computer Society, Washington, DC, USA (2009). DOI 10.1109/ICSE.2009.5070529. URL `http://dx.doi.org/10.1109/ICSE.2009.5070529`
39. Oizumi, W., Garcia, A., Colanzi, T., Staa, A., Ferreira, M.: On the relationship of code-anomaly agglomerations and architectural problems. Journal of Software Engineering Research and Development **3**(1), 1–22 (2015)
40. Oizumi, W., Garcia, A., Sousa, L., Cafeo, B., Zhao, Y.: Code anomalies flock together: Exploring code anomaly agglomerations for locating design problems. In: The 38th International Conference on Software Engineering; USA (2016)
41. Oizumi, W., Sousa, L., Oliveira, A., Carvalho, L., Garcia, A., Colanzi, T., Oliveira, R.: On the density and diversity of degradation symptoms in refactored classes: A multi-case study. In: 2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE), pp. 346–357 (2019). DOI 10.1109/ISSRE.2019.00042
42. Oizumi, W.N., da Silva Sousa, L., Oliveira, A., Garcia, A., Agbachi, O.I.A.B., Oliveira, R.F., Lucena, C.: On the identification of design problems in stinky code: experiences and tool support. J. Braz. Comp. Soc. **24**(1), 13:1–13:30 (2018). DOI 10.1186/s13173-018-0078-y. URL `https://doi.org/10.1186/s13173-018-0078-y`
43. Oliveira, A., Sousa, L., Oizumi, W., Garcia, A.: 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, pp. 83–92. ACM, New York, NY, USA (2019). DOI 10.1145/3357141.3357142. URL `http://doi.acm.org/10.1145/3357141.3357142`
44. Package, .R.: `http://wnoizumi.github.io/EMSE2021` (2021)

45. Page-Jones, M.: Fundamentals of Object-oriented Design in UML. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2000)
46. Palomba, F., Bavota, G., Penta, M.D., Oliveto, R., Lucia, A.D.: Do they really smell bad? a study on developers' perception of bad code smells. In: 2014 IEEE International Conference on Software Maintenance and Evolution, pp. 101–110 (2014). DOI 10.1109/ICSME.2014.32
47. Parnas, D.L.: Designing software for ease of extension and contraction. In: Proceedings of the 3rd International Conference on Software Engineering, ICSE '78, pp. 264–277. IEEE Press, Piscataway, NJ, USA (1978)
48. Perry, D.E., Wolf, A.L.: Foundations for the study of software architecture. SIGSOFT Softw. Eng. Notes **17**(4), 40–52 (1992). DOI 10.1145/141874.141884. URL `http://doi.acm.org/10.1145/141874.141884`
49. Rachow, P.: Refactoring decision support for developers and architects based on architectural impact. In: 2019 IEEE International Conference on Software Architecture Companion (ICSA-C), pp. 262–266 (2019). DOI 10.1109/ICSA-C.2019.00054
50. Refactoring oracle. URL `http://refactoring.encs.concordia.ca/oracle/`
51. Rizzi, L., Fontana, F.A., Roveda, R.: Support for architectural smell refactoring. In: Proceedings of the 2nd International Workshop on Refactoring, IWoR 2018, p. 7–10. Association for Computing Machinery, New York, NY, USA (2018). DOI 10.1145/3242163.3242165. URL `https://doi.org/10.1145/3242163.3242165`
52. Sarkar, S., Ramachandran, S., Kumar, G.S., Iyengar, M.K., Rangarajan, K., Sivagnanam, S.: Modularization of a large-scale business application: A case study. IEEE Softw. **26**(2), 28–35 (2009). DOI 10.1109/MS.2009.42. URL `https://doi.org/10.1109/MS.2009.42`
53. Schach, S., Jin, B., Wright, D., Heller, G., Offutt, A.: Maintainability of the linux kernel. Software, IEE Proceedings - **149**(1), 18–23 (2002)
54. Sharma, T., Singh, P., Spinellis, D.: An empirical investigation on the relationship between design and architecture smells. Empirical Software Engineering **25**(5), 4020–4068 (2020)
55. Sharma, T., Spinellis, D.: A survey on software smells. Journal of Systems and Software **138**, 158 – 173 (2018). DOI https://doi.org/10.1016/j.jss.2017.12.034. URL `http://www.sciencedirect.com/science/article/pii/S0164121217303114`
56. Silva, D., Tsantalis, N., Valente, M.T.: Why we refactor? confessions of github contributors. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, pp. 858–870. ACM, New York, NY, USA (2016). DOI 10.1145/2950290.2950305. URL `http://doi.acm.org/10.1145/2950290.2950305`
57. Silva, M.C.O., Valente, M.T., Terra, R.: Does technical debt lead to the rejection of pull requests? In: Proceedings of the 12th Brazilian Symposium on Information Systems, SBSI '16, pp. 248–254 (2016)
58. Sousa, L., Oizumi, W., Garcia, A., Oliveira, A., Cedrim, D., Lucena, C.: When are smells indicators of architectural refactoring opportunities: A study of 50 software projects. In: Proceedings of the 28th International Conference on Program Comprehension, ICPC '20, p. 354–365. Association for Computing Machinery, New York, NY, USA (2020). DOI 10.1145/3387904.3389276. URL `https://doi.org/10.1145/3387904.3389276`
59. Sousa, L., Oliveira, A., Oizumi, W., Barbosa, S., Garcia, A., Lee, J., Kalinowski, M., de Mello, R., Fonseca, B., Oliveira, R., Lucena, C., Paes, R.: Identifying design problems in the source code: A grounded theory. In: Proceedings of the 40th International Conference on Software Engineering, ICSE '18, pp. 921–931. ACM, New York, NY, USA (2018). DOI 10.1145/3180155.3180239. URL `http://doi.acm.org/10.1145/3180155.3180239`
60. Sousa, L., Oliveira, R., Garcia, A., Lee, J., Conte, T., Oizumi, W., de Mello, R., Lopes, A., Valentim, N., Oliveira, E., Lucena, C.: How do software developers identify design problems?: A qualitative analysis. In: Proceedings of 31st Brazilian Symposium on Software Engineering, SBES'17 (2017)
61. Suryanarayana, G., Samarthyam, G., Sharma, T.: Refactoring for Software Design Smells: Managing Technical Debt, 1st edn. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2014)
62. Trifu, A., Marinescu, R.: Diagnosing design problems in object oriented systems. In: WCRE'05, p. 10 pp. (2005)

63. Tsantalis, N., Guana, V., Stroulia, E., Hindle, A.: A multidimensional empirical study on refactoring activity. In: Proceedings of the 2013 Conference of the Center for Advanced Studies on Collaborative Research, pp. 132–146. IBM Corp. (2013)
64. Tsantalis, N., Mansouri, M., Eshkevari, L.M., Mazinanian, D., Dig, D.: Accurate and efficient refactoring detection in commit history. In: Proceedings of the 40th International Conference on Software Engineering, ICSE '18, pp. 483–494. ACM, New York, NY, USA (2018). DOI 10.1145/3180155.3180206. URL http://doi.acm.org/10.1145/3180155.3180206
65. Tufano, M., Palomba, F., Bavota, G., Oliveto, R., Di Penta, M., De Lucia, A., Poshyvanyk, D.: 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 (2015)
66. Vidal, S., Guimaraes, E., Oizumi, W., Garcia, A., Pace, A.D., Marcos, C.: Identifying architectural problems through prioritization of code smells. In: SBCARS16, pp. 41–50 (2016)
67. Xiao, L., Cai, Y., Kazman, R., Mo, R., Feng, Q.: Identifying and quantifying architectural debt. In: Proceedings of the 38th International Conference on Software Engineering, ICSE '16, pp. 488–498. ACM, New York, NY, USA (2016). DOI 10.1145/2884781.2884822. URL http://doi.acm.org/10.1145/2884781.2884822
68. Xing, Z., Stroulia, E.: Umldiff: An algorithm for object-oriented design differencing. In: Proc. of ASE '05, pp. 54–65 (2005). DOI 10.1145/1101908.1101919. URL http://doi.acm.org/10.1145/1101908.1101919
69. Yamashita, A., Moonen, L.: Do developers care about code smells? an exploratory survey. In: 2013 20th Working Conference on Reverse Engineering (WCRE), pp. 242–251 (2013). DOI 10.1109/WCRE.2013.6671299
70. Yamashita, A., Moonen, L.: 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, pp. 682–691 (2013)
71. Yin, R.K.: Qualitative research from start to finish. Guilford publications (2015)
72. Zimmermann, O.: Architectural refactoring for the cloud: a decision-centric view on cloud migration. Computing **99**(2), 129–145 (2017). DOI 10.1007/s00607-016-0520-y. URL https://doi.org/10.1007/s00607-016-0520-y