On the density and diversity of degradation symptoms in refactored classes: A multi-case study

Willian Oizumi
Informatics Department – PUC-Rio
Rio de Janeiro, Brazil
woizumi@inf.puc-rio.br

Leonardo Sousa, Anderson Oliveira
Informatics Department – PUC-Rio
Rio de Janeiro, Brazil
{lsousa, aoliveira}@inf.puc-rio.br

Luiz Carvalho
Informatics Department – PUC-Rio
Rio de Janeiro, Brazil
luizmatheus.ac@gmail.com

Alessandro Garcia
Informatics Department – PUC-Rio
Rio de Janeiro, Brazil
afgarcia@inf.puc-rio.br

Thelma Colanzi
Informatics Department – UEM
Maringa, Brazil
thelma@din.uem.br

Roberto Oliveira
Campus Posse – UEG
Posse, Brazil
roberto.oliveira@ueg.br

***Abstract*—Root canal refactoring is a software development activity that is intended to improve dependability-related attributes such as modifiability and reusability. Despite being an activity that contributes to these attributes, deciding when applying root canal refactoring is far from trivial. In fact, finding which elements should be refactored is not a cut-and-dried task. One of the main reasons is the lack of consensus on which characteristics indicate the presence of structural degradation. Thus, we evaluated whether the density and diversity of multiple automatically detected symptoms can be used as consistent indicators of the need for root canal refactoring. To achieve our goal, we conducted a multi-case exploratory study involving 6 open source systems and 2 systems from our industry partners. For each system, we identified the classes that were changed through one or more root canal refactorings. After that, we compared refactored and non-refactored classes with respect to the density and diversity of degradation symptoms. We also investigated if the most recurrent combinations of symptoms in refactored classes can be used as strong indicators of structural degradation. Our results show that refactored classes usually present higher density and diversity of symptoms than non-refactored classes. However, root canal refactorings that are performed by developers in practice may not be enough for reducing degradation, since the vast majority had little to no impact on the density and diversity of symptoms. Finally, we observed that symptom combinations in refactored classes are similar to the combinations in non-refactored classes. Based on our findings, we elicited an initial set of requirements for automatically recommending root canal refactorings.**

***Index Terms*—refactoring; root canal refactoring; code smell; object-oriented design principles; dependability attributes; source code degradation**

I. Introduction

As software systems evolve, they can go through changes that can lead to their structural degradation. Unfortunately, the structural degradation can lead software systems to the discontinuation or at least either significant maintenance effort or the complete redesign [1]–[4]. This degradation occurs when stakeholders make decisions that have a negative impact on dependability-related attributes [5]–[7]. An example of this scenario is when a stakeholder decides to create a common system interface to provide access to different unrelated services. This decision is likely to harm the system maintainability and extensibility [8].

Developers constantly have to improve the internal structure of software systems to, in the worst case scenario, repair a deteriorated code. For this purpose, they have been relying on one of the most common activities applied during software maintenance and evolution: refactoring [9]. Refactoring is a transformation in the source code structure without changing the functional behavior of the system [9]–[11]. A commonly applied refactoring tactic is known as root canal refactoring, which involves a process of exclusively applying refactorings to reduce the structural degradation [10], [11]. Despite being an activity aimed at improving dependability-related attributes of the system's design, deciding when applying root canal refactoring is not trivial [12].

Developers need to know where they should refactor the source code; more specifically, they have to find first what code elements (packages, classes, methods, and the like) need to be refactored to reduce the structural degradation [12]. To this end, developers can find and monitor indicators of structural degradation in the source code, i.e., they need to rely on symptoms of structural degradation [13]. Code smell is an example of a symptom. It is a structure in the system implementation that represents a surface indication of structural degradation [9]. An example of code smell is *Long Method*, which indicates a method that is too long to understand [9].

After the degradation symptoms have been found, developers can reduce the structural degradation by applying root canal refactoring [10]. Hence, one might expect that developers often apply refactoring in code elements that contain either multiple symptom instances (*density*) or different types of symptoms (*diversity*). Unfortunately, there is little information and no much consensus whether the density and diversity of multiple automatically detected symptoms can be consistent indicators of the need for root canal refactoring.

This work was supported by CNPq (grants 153363/2018-5, 434969/2018-4, 140919/2017-1, and 312149/2016-6), CAPES (grant 175956) and FAPERJ (grant 22520-7/2016).

Existing studies are mostly focused in investigating the impact of any refactoring kind in the density of symptoms [14], [15]. However, none of them investigated the relation of root canal refactorings with the density and diversity of symptoms. Thus, we investigated to what extent the density and diversity of symptoms indicate the need for root canal refactoring. We also investigated whether root canal refactoring impacts the density and diversity of symptoms.

To better understand the density and diversity of degradation symptoms, we conducted a multi-case exploratory study in which we observed the root canal refactorings applied by developers. This study involves eight software system: six open source systems and two systems from our industry partners. For each software system, we found the classes that were changed through one or more root canal refactorings, and then we collected the structural degradation symptoms in all classes from the system. After that, we compared refactored and non-refactored classes with respect to the density and diversity of symptoms. We also evaluated the impact of root canal refactorings on the density and diversity of these detected symptoms. Finally, we investigated if the most recurrent combinations of symptoms in refactored classes are different from the recurrent combinations in non-refactored classes.

Upon data analysis, we found that refactored classes usually present higher density and diversity of symptoms than other classes. After investigating what happens with the refactored classes, we did not find a consistent reduction in the density or in the diversity of symptoms, leading us to conclude that refactorings cause little to no positive effect in degradation symptoms. In fact, the effects of refactorings are practically nonexistent in the core classes that are constantly modified. We also found that the recurrent combinations of symptoms in refactored classes are similar to the recurrent combinations in non-refactored classes. Thus, the combinations by themselves are not good indicators of structural degradation. Based on such findings, we elicited an initial set of requirements for automatically recommending root canal refactorings.

II. Background

A. Structural Degradation

Structural degradation occurs when stakeholders make decisions that negatively impact dependability-related attributes [5]–[7]. An example of structural degradation is the so called Fat Interface [16]. This form of structural degradation occurs when a single interface provides multiple and unrelated operations, making it difficult to use and increasing the chance of introducing defects to its clients. Due to the negative impact caused by structural degradation, software systems have often been discontinued or redesigned when structural degradation was allowed to persist [3]. Thus, to be able to maintain the system's quality, developers need to identify and to confirm the existence of structural degradation. Next, we present an example to illustrate how dependability-related attributes may be impacted by structural degradation.

Figure 1 shows a partial view of the OpenPOS system before and after a degraded structure has been refactored. OpenPOS is a system that provides sales features. One of the functionalities of OpenPOS comprises the generation of payment slips. In the country where OpenPOS is used, payment slips serve for clients to make payments at any bank. Developers of OpenPOS implemented this feature in the *PaymentSlip* sub-component. To protect system information, this sub-component was strongly dependent from the *Authentication* component.

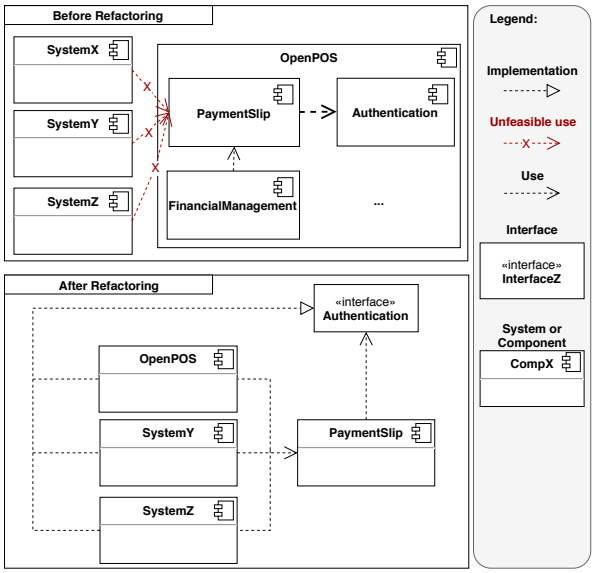

Fig. 1. Example of structural degradation impacting Reusability

Unfortunately, the strong dependency with the *Authentication* component led to a side effect on the reusability of *PaymentSlip* sub-component. Reusability is a sub-category of maintainability that indicates the degree to which a component can be re-used in two or more systems [17]. Since *PaymentSlip* was so coupled to the *Authentication* component, it could not be reused in other systems. In this context, developers have to refactor the *PaymentSlip* sub-component to reduce the coupling with *Authentication* component. Additionally, refactoring this kind of structural degradation is fundamental to avoid code duplication among systems and rework. In Section II-C, we will explain the structure obtained after the refactoring.

B. Degradation Symptoms

Sousa et al. [13] identified five categories of symptom upon which developers frequently rely to identify structural degradation. Similarly to other related work [18]–[21], they observed that developers tend to combine multiple symptoms, taking into account dimensions such as diversity and density to decide if there is a degradation or not. In this work, we selected a sub-set of two symptom categories that can be automatically detected using state-of-the-practice tools, which are the code smells and the principle violations.

Code smell is a surface indicator of possible structural degradation [9]. This symptom category have been extensively investigated by different researchers (e.g., [22]–[25]). Recent studies [13], [18], [20], [21] suggest that combining multiple

code smells may improve the precision when identifying structural degradation. An example of code smell type is the Long Method. This type of smell usually leads to structural degradation related to modifiability.

In object-oriented systems, structural degradation usually impact object-oriented design characteristics, such as abstraction, encapsulation, modularity, and hierarchy. Therefore, the second symptom category we used comprises the **principle violations**, which are symptoms that may indicate the violation of common object-oriented principles [16]. An example of object-oriented principle is the *Single Responsibility Principle* (SRP). The SRP determines that each class should have a single and well defined responsibility in the system [16]. An example of symptom that may be used for finding SRP violations is the *Insuficient Modularization* [26]. This symptom occurs in classes that are large and complex, possibly due to the accumulation of responsibilities.

Table I shows the descriptions for the 17 types of principle violations and 10 types of code smells used in this study. The descriptions are based on the taxonomy of symptoms provided by Sharma and Spinellis [27], [28].

C. Refactoring

Refactoring consists in transforming the source code structure without changing the functional behaviour of the system [9]. Thus, we consider that refactoring is any structural software change that is aimed at improving dependability-related attributes of the system's design.

According to Murphy-Hill and Black [10], refactoring can be classified into two tactics, which are floss refactoring and root canal refactoring. On one hand, floss refactoring is characterized by refactoring changes intermingled with other kinds of source code changes, such as adding new features and fixing bugs. The aim of floss refactoring is to keep structural quality as a means to achieve other goals. On the other hand, root canal refactoring aims at exclusively reducing structural degradation. A root canal refactoring consists of only refactoring changes; it is not performed in conjunction with other non-refactoring changes. Thus, in this paper, our focus is on root canal refactorings as they are explicitly aimed to reduce structural degradation. Thus, from now on, whenever we talk about refactoring in this paper, we'll be referring to root canal refactoring.

To illustrate our definition of refactoring, let's return to the example of Figure 1. As previously discussed, multiple different systems of the same company began to require a payment slip feature. Therefore, developers were asked to remove the reusability degradation by refactoring the *PaymentSlip* sub-component. The refactoring consisted of introducing an interface for authentication. This way, each system that needs to use the *PaymentSlip* component must specify an authentication component that meets the interface specifications required by *PaymentSlip*. After refactoring the *PaymentSlip* sub-component, besides fixing the structural degradation, it is expected the removal of symptoms such as the *Hub-Like Modularization* (Table I).

TABLE I
SHORT DESCRIPTION FOR THE SYMPTOMS USED IN THIS STUDY

Symptom Type	Description
Category 1 - Code Smells	
Abstract Function Call From Constructor	A constructor that calls an abstract method
Complex Conditional	A conditional statement that is complex
Complex Method	A method that has high cyclomatic complexity
Empty Catch Block	A catch block of an exception that is empty
Long Identifier	An identifier that is excessively long
Long Method	A method that is too long to understand
Long Parameter List	A method that accepts a long list of parameters
Long Statement	A statement that is excessively long
Magic Number	When an unexplained number is used in an expression
Missing Default	A switch statement that does not contain a default case
Category 2 - Principle Violations	
Broken Hierarchy	A supertype and its subtype that conceptually do not share an "is a" relationship
Broken Modularization	When data and/or methods that should have been into a single abstraction are spread across multiple abstractions
Cyclic Dependent Modularization	When two or more abstractions depend on each other directly or indirectly
Cyclic Hierarchy	A supertype in a hierarchy that depends on any of its subtypes
Deep Hierarchy	An inheritance hierarchy that is excessively deep
Deficient Encapsulation	The accessibility of one or more members of an abstraction is more permissive than actually required
Hub Like Modularization	An abstraction that has dependencies with a large number of other abstractions
Imperative Abstraction	When an operation is turned into a class
Insufficient Modularization	An abstraction that has not been completely decomposed
Missing Hierarchy	When a design segment uses conditional logic instead of polymorphism
Multifaceted Abstraction	An abstraction that has more than one responsibility assigned to it
Multipath Hierarchy	A subtype that inherits both directly as well as indirectly from a supertype
Rebellious Hierarchy	A subtype that rejects the methods provided by its supertype(s)
Unexploited Encapsulation	A client class that uses explicit type checks instead of exploiting the variation in types already encapsulated within a hierarchy
Unnecessary Abstraction	An abstraction that is actually not needed in the system
Unutilized Abstraction	An abstraction that is left unused
Wide Hierarchy	An inheritance hierarchy that is too wide

III. STUDY DESIGN

A. Goal and Research Questions

Several studies (e.g., [29]–[31]) have proposed and evaluated techniques for the detection of structural degradation. Nevertheless, in practice, most of them are not applied by developers. One of the issues of existing techniques is the high amount of false positives [21], [32], which may lead developers to have little confidence in the presented symptoms. Another problem is that most techniques are based on a single category of symptom. However, according to the literature, developers may combine multiple and diverse symptoms for confirming the existence of a structural degradation. In this sense, there are techniques that work with multiple symptoms [19], [21]. Still, unlike what was observed in the study of Sousa et al. [13], existing techniques only combine symptoms of the same category (e.g., code smells). In addition, their efficiency to reveal classes impacted by structural degradation has not been exhaustively validated. Finally, there is little evidence on the impact of root canal refactoring on symptoms such as principle violations. Thus, in this paper, *we aim at*

evaluating the relation of root canal refactorings with the occurrence of multiple and diverse degradation symptoms. To achieve our goal, we defined the following research questions:

> RQ1. Are the density and diversity of degradation symptoms in (root canal) refactored classes different from the density and diversity in other classes?

With RQ1, we aim at understanding if the degradation symptoms are denser and more diverse in refactored classes when compared to other classes. As root canal refactorings should be applied to classes impacted by structural degradation [10], we need to know if such classes, before being refactored, present higher density and diversity of symptoms than most regular classes. Answering this question will be helpful for evaluating whether combining multiple and diverse symptoms is indeed an effective strategy for identifying and confirming the existence of structural degradation.

> RQ2. Do classes modified by root canal refactorings present structural improvement in the medium term?

With RQ2, we want to observe whether the expected improvements of root canal refactorings impact the density and diversity of symptoms. This question will help us to understand if symptoms disappear, decrease, or increase in the medium term after the application of root canal refactorings. In this context, we consider medium term as being the next release after refactoring. With this research question we will also be able to better understand if the refactorings performed in practice have been effective, according to the measurement provided by the investigated symptom categories.

> RQ3. Are the combinations of symptoms in (root canal) refactored classes different from the combinations in other classes?

The aim of RQ3 is to investigate whether combinations of symptoms can be used to differentiate refactored classes from other classes. In addition, this research question will help us to understand which combinations of code smells and principle violations are often refactored by the developers of our target systems. Based on the findings, it may be possible to prioritize degraded classes based on the combinations of symptoms that developers refactor more often.

To answer our research questions, we conducted a case study involving multiple and diverse software systems. We collected and analyzed source code changes due to root canal refactoring , i.e., changes that were exclusively dedicated to fixing structural degradation. After that, we collected multiple types of code smells and principle violations and conducted our data analysis. Next, we provide details about the target systems and about our procedures for data collection and analysis.

TABLE II
CHARACTERISTICS OF TARGET SYSTEMS

Name	Platform	Domain	Size (LOC)	# of Commits	Releases
Partners' Systems					
OpenPOS	.Net/C#	Enterprise	97,000	3,318	67, 68
UniNFe	.Net/C#	Enterprise	492,000	2,373	345, 362
Open Source Systems					
Achilles	Java	Tool	83,124	1,188	1.0-beta, 3.0.0, 5.1.0
Ant	Java	Tool	137,314	13,331	15_ 141, 163_ 170, 180_
Derby	Java	Database	1,760,766	8,135	10.3.2.1, 10.5.3.0, 10.7.1.1
Elasticsearch	Java	Engine	578,561	23,597	1.2.2, 1.5.0, 2.3.0
MPAndroidChart	Android/Java	Library	23,060	1,737	1.0.1, 2.1.0, 2.2.4
Tomcat	Java	Middleware	668,720	18,068	7.0.0-RC1, 7.0.8, 7.0.35, 7.0.57, 7.0.67, 8.5.9

B. Target Systems

Table II shows the target systems of this paper. Columns two to five of Table II show respectively the: platform, the system domain, the size in Lines of Code (LOC), and the number of commits. Column six shows the releases of each system in which we collected degradation symptoms.

Open Source Systems. As presented in Table II, we selected six open source systems for this study: Apache Ant, Apache Derby, Apache Tomcat, Achilles, Elasticsearch, and MPAndroidChart. To select these systems, we first selected 50 open source systems in which we applied a set of filtering criteria (Section III-C). We aimed at selecting a set of representative systems from different domains.

Partners' Systems. To make our data sample more heterogeneous, we selected two C# systems from our industry partners. The first system is OpenPOS, a desktop system that provides sales features, such as sales registration and cashier closing. UniNFe is a background service that sends and receives electronic invoices. These projects are suitable for this study because each of them presents more than one hundred classes that were refactored due to structural degradation. In addition, their root canal refactorings are documented in specific refactoring tasks. Finally, we had full access to their original developers for questions and clarifications.

C. Data Collection and Analysis

We followed three main steps for data collection and analysis: (a) finding root canal refactorings, (b) collecting information about structural degradation symptoms, and (c) running data analysis. Next we present details about each step.

*a) **Finding root canal refactorings**:* In the first step, we searched for source code changes that were exclusively intended to fix structural degradation. To achieve this goal, we adopted different procedures for the partners' systems and for the open source systems. To select the open source systems, we started by analyzing a database containing information about 50 open source projects. We created and validated this database in previous studies [15], [33] in which we collected information about the projects' history of changes, commit messages, and performed refactorings. We used Refactoring Miner 0.2.0 [34] to automatically detect refactorings of 11

different types. Due to space constraints, the description of refactoring types are presented in our replication package[1].

Since Refactoring Miner is unable to differentiate root canal from floss refactoring, we identified and filtered the root canal refactorings based on the following filter: (1) the selected refactorings should be occurring in groups of two or more refactorings, and (2) the refactorings withing a group should have been detected in the same commit or in sequential commits. As a result, this filter has helped us to find refactorings that changed multiple source code structures and, therefore, had a greater chance of being root canal refactorings. After filtering, we discarded the systems with less than 10 refactored classes since it would be a very small sample of classes. In this way, we have reduced our sample of systems to the 6 open source systems presented in Table II. We also tried to apply a second filter by considering the commit message associated with each refactoring change. With this filter we would only select the refactorings for which the associated commit message contained any variation of the word refactor (e.g., refactoring, Refactor, etc). However, the resulting sample of refactorings was very small, which meant that the results would not be statistically significant. Thus, we decided to use this second filter only as a parameter of comparison for the results obtained with the first filter.

To find refactorings in partners' systems, we asked two original developers of each project to provide us with a list of tasks aimed at root canal refactoring. After that, we conducted an automated search in the issue tracking system to complement the lists of tasks provided by developers. Our automated search was based on a set of keywords that are often associated with structural degradation (e.g., structure, interface, and duplicate). We have defined those keywords based on the analysis of task descriptions from 50 open source projects from our previously mentioned database. These keywords often occur in the description of tasks that aim at improving dependability-related attributes. After the automated search with the keywords, we, together with the two developers, analyzed the resulting list of tasks. We discarded those that could not be characterized as root canal refactorings.

*b) **Collecting information about degradation symptoms**:* As presented in Section II, we collected two categories of structural degradation symptoms: code smells and principle violations. We used the Designite tool to collect these symptoms [35]. We selected this tool because it detects the same set of symptoms for both C# and for Java programs, thus, keeping the consistency regarding the detection strategies for both programming language. Detailed descriptions, detection strategies, and thresholds for all types of symptoms are available in our replication package.

As illustrated in Figure 2, we collected the symptoms in the last release of the system before refactorings and in the first release after refactorings. The releases presented in the last column of Table II are the ones that we collected the symptoms. We are aware that this makes refactoring changes to be mixed with other changes. Nevertheless, we have chosen this approach intentionally, since we wanted to evaluate if the possible structural improvements caused by root canal refactorings persist in the medium term.

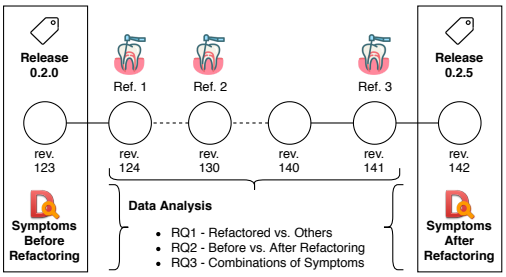

Fig. 2. Collection of symptoms for data analysis

*c) **Running data analysis**:* After collecting data about tasks, source code changes and symptoms, we conducted the data analysis to answer our research questions. To answer RQ1, we divided our dataset into two groups: Refactored Classes and Other Classes (or simply, Others). The first one is composed by all classes for which we found one or more refactorings. The second group is composed by all classes in the systems that are not in the former group.

For both groups, we calculated the density and diversity of symptoms. *Density* represents the number of individual instances of symptoms occurring in a class, while *diversity* represents the number of different symptom types occurring in a class. We compared the density of both groups by computing the code smell and principle violation distributions. We compared the diversity of symptoms by calculating the distribution of symptom types quantity. As previously explained, we considered 10 types of code smells and 17 types of principle violations. Based on the collected data about density and diversity, we used the *Mann-Whitney Wilcoxon* statistical test to check whether there was a significant difference in the distributions presented by both groups.

To answer RQ2, we collected and compared the same data used to answer RQ1. The difference here is that we compared the density and diversity of classes collected in releases before and after the execution of root canal refactorings. Hence, for this question, we only considered classes that were changed by root canal refactorings. Additionally, with the help of developers from our industry partners, we conducted further analysis to better understand the obtained results.

Finally, to answer RQ3, we performed a threefold analysis. First, we investigated the number of classes affected by 170 pairwise combinations of code smell types with principle violation types. Second, we generated two rankings for the combinations based on the number of refactored (1^{st} ranking) and non-refactored (2^{nd} ranking) classes affected by each combination. Then, we applied the *Spearman's rank correlation rho* statistical test to compare both rankings. Finally, we evaluated the relevance of symptom combinations for recommending root canal refactorings. Our rationally for combining symptoms from different categories is that they could be stronger indicators of degradation. We also tried to investigate

[1]http://wnoizumi.github.io/ISSRE2019/

combinations with more than two symptoms. However, those combinations were rare and not observed in more than two systems. Conversely, combinations with only two symptoms from different categories occurred frequently.

IV. Results

A. Density and Diversity as Consistent Indicators

Table III shows the mean density of symptoms in the classes of both groups (refactored and others). Each line shows, for a symptom category, the mean density of symptoms in refactored classes (Ref.) and in other classes (Others). This information is provided for each system before and after refactorings.

For code smells, we observed a notable difference when comparing refactored classes with others in all target systems. This difference indicates that the density of smells can be used as a strong indicator of structural degradation. For the most extreme cases (OpenPOS and UniNFe), the density of smells was more than 8 times higher in refactored classes. Analyzing the dataset, we also observed several outliers in the distribution of code smells for Others. Many of these outliers may be classes affected by structural degradation that were not changed in root canal refactorings.

Principle violations, in general, were denser in refactored classes when compared to other classes. However, for all target systems the observed difference was small. The system that presented the greatest difference regarding principle violations was the UniNFe, where the density of violations was almost two times higher in refactored classes. Nevertheless, in all target systems, the density of principle violations in refactored classes (before refactoring) was higher than in other classes.

We applied the *Mann-Whitney Wilcoxon test* to check whether there was a statistically significant difference in the density distribution. Table IV summarizes the results for all systems. The results related to density, in the context of RQ1, are presented in the second (for principle violation) and fourth (for code smell) columns and in lines four to eleven of Table IV. A p-value smaller than 0.05, means that the distribution of density of symptoms in refactored classes is different from the distribution of density in other classes. The raw data and the detailed results of this statistical test are available in our replication package.

The tests showed that, for all systems, **the smell density in refactored classes was significantly different from the smell density in other classes**. On the other hand, when we ran the same test for the density of principle violations, we cannot reject the null hypothesis for the OpenPOS and MPAndroidChart systems. Therefore, the density distribution of principle violations in refactored classes may be equal to the density distribution of principle violations in other classes for some systems. For MPAndroidChart, this result is partially explained by the sample size, since we found only 18 refactored classes in this system. The explanation for OpenPOS, is the fact that its refactorings were more focused on fixing modifiability degradation. Such issues generally do not manifest themselves in the form of principle violations, since they do not affect aspects such as abstraction and hierarchy. In any case, to achieve greater generalization, the density of principle violations must be further investigated in the context of other systems.

Diversity of symptoms is also more significant for code smells. Table V shows the diversity of symptoms for code smells and for principle violations. This table follows the same organization of Table III, providing the mean diversity of each symptom category. In all systems, the diversity of code smells was significantly higher in refactored classes when compared to other classes in the systems. For refactored classes, the diversity of code smells was more than five times higher in Ant and more than seventeen times higher in UniNFe, for example. Similarly to what we observed regarding the density of symptoms, the statistical tests (3^{rd} and 5^{th} columns of Table IV) revealed that, for all target systems, the diversity mean of code smells in refactored classes is different from the diversity mean of code smells in other classes. However, regarding diversity of principle violations, we cannot reject the null hypothesis for the OpenPOS and MPAndroidChart systems. The rationale for explaining the diversity results in these two systems is the same as that used to explain the density results.

When we applied the second filter in the refactorings of open source systems (Section III-C), the density and diversity averages remained similar. The refactored classes continued to present higher density and diversity of symptoms when compared to other classes. This second filtering made the averages of most open source systems more similar to the averages observed in the partner systems. For the Derby system, for example, the mean density of both smells and violations in refactored classes became significantly higher: 2.50 for violations and 29.25 for smells. Therefore, based on our analyses, we conclude that **the density and diversity of symptoms in refactored classes are, indeed, different from the density and diversity of symptoms in other classes.** However, for two out of eight target systems, principle violations did not show significant differences for both density and diversity of symptoms. This indicates that we should, in future work, test our hypotheses on another set of systems to verify if the results converge. In addition, the results observed here indicates that the density and diversity of code smells may be considered a more reliable indicator of structural degradation, according to the criteria adopted by developers to decide when to conduct root canal refactorings.

B. Low Reduction of Symptoms After Refactoring

Table III shows the mean density of symptoms before and after the application of refactorings. It is possible to observe that refactorings caused little impact on all symptom categories. In some cases the mean value increased while in others it decreased after the refactorings. However, for most of them, there was not a significant difference. To confirm whether there is a significant difference between the two groups – before refactoring and after refactoring –, we applied the *Mann-Whitney Wilcoxon* test. In all target systems, the test

TABLE III
MEAN DENSITY OF SYMPTOMS IN REFACTORED CLASSES AND IN OTHERS

Category	Mean Density of Symptoms															
	OpenPOS		UniNFe		Achilles		Ant		Derby		Tomcat		Elasticsearch		MPAndroidChart	
	Ref.	Others	Ref.	Others	Ref.	Others	Ref.	Others	Ref.	Others	Ref.	Others	Ref.	Others	Ref.	Others
Before Refactoring																
P. Violation	0.806	0.756	2.238	1.134	1.632	1.073	1.802	1.068	1.802	1.068	1.801	1.032	1.632	1.042	1.250	1.138
Code Smell	35.843	4.102	41.642	1.415	16.775	4.578	10.985	1.810	10.985	1.810	23.72	3.731	14.475	6.380	21.62	9.317
After Refactoring																
P. Violation	0.725	0.768	2.404	1.497	1.666	1.043	1.921	1.070	1.921	1.070	1.794	1.020	1.557	1.082	1.625	1.127
Code Smell	35.462	3.765	41.357	1.719	32.055	4.463	12.156	1.840	12.156	1.840	21.339	3.691	24.242	7.440	18.687	8.475

TABLE IV
P-VALUES OF THE MANN-WHITNEY WILCOXON TEST FOR RESEARCH QUESTIONS RQ1 AND RQ2

System	Principle Violation		Code Smell	
	Density	Diversity	Density	Diversity
RQ1 - refactored classes and others				
OpenPOS	**0.18**	**0.12**	<0.01	<0.01
UniNFe	<0.01	0.03	<0.01	<0.01
Achilles	<0.01	0.01	<0.01	0.01
Ant	<0.01	<0.01	<0.01	<0.01
Derby	<0.01	<0.01	<0.01	<0.01
Elasticsearch	<0.01	0.01	<0.01	<0.01
MPAndroidChart	**0.76**	**0.52**	<0.01	0.04
Tomcat	<0.01	<0.01	<0.01	<0.01
RQ2 - before and after refactoring				
OpenPOS	0.62	0.68	0.54	0.32
UniNFe	0.98	0.88	0.72	0.68
Achilles	0.72	<**0.01**	0.06	0.06
Ant	0.33	0.30	0.59	0.54
Derby	0.63	0.61	0.96	0.46
Elasticsearch	0.73	<**0.01**	0.10	<**0.01**
MPAndroidChart	0.35	<**0.01**	0.64	0.07
Tomcat	0.96	0.11	0.20	0.34

revealed p-values higher than 0.05 for both code smells and principle violations, indicating that we cannot reject the null hypothesis. Based on such result, we concluded that root canal refactorings applied in the systems of this case study did not reduce the density of any of the investigated symptoms.

Regarding diversity, we carried out a similar analysis. Table V shows the mean number of different types of code smells and principle violations. In this case, the difference was often marginal for both symptom categories. Moreover, we did not observe similar trends for most systems. The mean diversity for both symptom categories was reduced in some systems but increased in other systems. The statistical test revealed p-values higher than 0.05 for both code smells and principle violations in most systems. The only systems in which we observed a statistically significant difference for principle violations were Achilles, MPAndroidChart, and Elasticsearch. For code smells, only in Elasticsearch the diversity before and after refactoring was statistically different. Thus, it is not possible to state that diversity of any symptom category always reduces or increases after refactorings.

As we could not observe a significant and consistent reduction in the density or in the diversity of symptoms, the answer of our second research question RQ2 is that **refactorings cause little to no impact on symptoms of structural degradation in the medium term**. This trend was maintained even after applying the second filter (Section III-C) to the refactorings of open source systems.

To better understand why most refactorings did not remove symptoms, we decided to take a close look at refactored classes with the help of our industry partners. We selected and analyzed two sub-sets of refactored classes from OpenPOS and UniNFe: (1) classes with increased density, and (2) classes with decreased density. The former is composed by refactored classes that, after refactorings, presented higher density of symptoms. The latter is composed by refactored classes that, after refactorings, presented lower density of symptoms. With the first set of refactored classes, we expected to identify and analyze the classes that, even after refactorings, have continued to worsen the structural quality. On the other hand, with the second set, we intended to find cases of success in which the refactorings fixed structural degradation.

Symptoms tend to increase in core classes. Table VI shows the classes of OpenPOS and UniNFe that presented higher density for both symptom categories. Analyzing the classes in which there was an increase in the density of all symptoms, we asked the developers to describe what they remember about the implementation and maintenance of each class. Based on their observations, we noted that many of the refactored classes are also frequently changed in other tasks. In addition, many of the refactored classes that presented higher density after refactorings are considered core classes of the system. That is, they are linked to fundamental functionalities of the system. The *frmVendaCF* class, for example, is a core class that was changed in 306 different commits, while most classes in the OpenPOS system were not changed more than 20 times. Such changes may be often conducted without proper concern for structural quality.

As a result, any improvement promoted by refactorings ends up getting lost with the structural degradation. Thus, even being refactored in three different tasks – for improving modifiability and reusability –, *frmVendaCF* continued to present high density and diversity of symptoms. In fact, analyzing these results from the perspective of the refactoring literature, other studies [14], [15] have pointed out that refactoring in general (not just root canal) does not usually remove symptoms such as code smells. We conjecture that this is due to the fact that carrying out structural transformations is usually costly. Therefore, developers end up performing root canal refactorings only to avoid increasing degradation in classes that (1) have been poorly designed, or (2) undergo constant modifications related to changing requirements.

The effects of refactorings only persist in classes that are not often modified. In OpenPOS and UniNFe, only the *frmCliente* class from OpenPOS presented a decreased

TABLE V
MEAN DIVERSITY OF SYMPTOMS IN REFACTORED CLASSES AND IN OTHERS

Category	Mean Diversity of Symptoms															
	OpenPOS		UniNFe		Achilles		Ant		Derby		Tomcat		Elasticsearch		MPAndroidChart	
	Ref.	Others	Ref.	Others	Ref.	Others	Ref.	Others	Ref.	Others	Ref.	Others	Ref.	Others	Ref.	Others
Before Refactoring																
P. Violation	0.650	0.701	1.166	0.815	0.500	0.218	1.157	0.540	1.725	0.876	1.113	0.568	1.143	0.828	0.555	0.363
Code Smell	1.581	0.444	3.285	0.190	0.437	0.175	1.931	0.352	3.412	1.247	2.581	0.529	2.151	0.770	1.388	0.411
After Refactoring																
P. Violation	0.600	0.722	1.142	1.076	0.187	0.366	1.280	0.644	1.825	0.974	1.272	0.619	0.459	0.477	1.444	0.765
Code Smell	1.387	0.424	3.523	0.245	0.237	0.280	2.093	0.418	3.700	1.365	2.795	0.569	0.877	0.441	2.555	0.822

TABLE VI
CLASSES WITH INCREASED DENSITY AND DIVERSITY OF SYMPTOMS

System	Class
OpenPOS	OpenPOS.Data.Abstract.Faturamento.Lancamento.Movimento.NF.NFBase
	OpenPOS.Data.Regra.CFOP.CFOPRegraFiltro
	OpenPOS.Data.Abstract.Cadastro.Item.ItemBase
	OpenPOS.Desktop.Forms.FrenteCaixa.Lancamento.frmVendaCF
UniNFe	UniNFe.Service.TFunctions
	UniNFe.Service.Processar
	UniNFe.Service.TaskAbst
	UniNFe.ConvertTxt.UniNFeW
	UniNFe.Service.TaskConsultarLoteeSocial
	NFSe.Components.SchemaXMLNFSe_TIPLAN

TABLE VII
TOP-10 COMBINATIONS OF SYMPTOMS IN REFACTORED CLASSES

Position	Combination	# of Affected Classes
1	Long Statement-Insufficient Modularization	216
2	Complex Method-Insufficient Modularization	212
3	Magic Number-Insufficient Modularization	178
4	Long Statement-Deficient Encapsulation	145
5	Complex Conditional-Insufficient Modularization	135
6	Complex Method-Deficient Encapsulation	126
7	Long Statement-Unutilized Abstraction	111
8	Magic Number-Deficient Encapsulation	109
9	Complex Method-Unutilized Abstraction	102
10	Long Parameter List-Insufficient Modularization	99

number of symptoms. We observed that the structural quality of this class was indeed improved. However, the developers of OpenPOS revealed to us that *frmCliente* is not often modified – it was changed less than 20 times along source code history. Therefore, a natural conclusion is that the low volume of changes allowed this class to maintain a good structure. Nevertheless, structural degradation is critical when impacting the core classes of the system, such as the ones presented in Table VI. Thus, developers still need help to effectively identify and refactor the most relevant structural degradation.

C. Combinations as Indicators of Degradation?

In the previous research questions we observed that developers tend to apply root canal refactorings in classes with high density and diversity of symptoms, and that most root canal refactorings present little to no persistent positive effects on the density and diversity of symptoms. Thus, aiming at improving the effectiveness of existing detection techniques for structural degradation, we investigated which combinations of symptoms are more likely to indicate structural degradation.

As explained in Section III, we identified the number of classes affected by 170 pairwise combinations of code smell types with principle violation types both for refactored classes and for other classes. Table VII shows the top-10 pairwise combinations of code smells and principle violations in refactored classes. Each line corresponds to a pairwise combination of code smell and principle violation. Column 3 shows the number of refactored classes, considering all target systems, affected by each pairwise combination. The complete rankings of combinations for both groups (refactored and other classes) are available in our replication package.

To observe whether the recurrent combinations can be used as indicators of structural degradation, we applied the *Spearman's rank correlation rho* test to compare the ranking of combinations in refactored classes and in other classes. With a confidence level of 95% and p-value smaller than 0.00001, we obtained 0.90 as the correlation coefficient. This result indicates a strong correlation between the two rankings, which means that **we cannot use the combinations of symptoms to differentiate degraded classes from other classes**. On the other hand, based on results from the literature (e.g., [36]), we believe that combinations of symptoms may still be useful in other contexts, as we will describe in Section V.

V. REQUIREMENTS FOR RECOMMENDING ROOT CANAL REFACTORINGS

Given the results presented in this paper, we envision a technique for automatically recommending root canal refactorings for degraded classes. This technique should take into consideration our insights on the density, diversity, and combination of symptoms. Therefore, based on such insights, we elicited four main requirements for the recommendation of root canal refactorings. As illustrated by Figure 3, requirements involve the activities of symptom collection, filtering of refactoring candidates, prioritization of the most relevant candidates, and summarizing of information about structural degradation. Below we present the detailed description of each requirement.

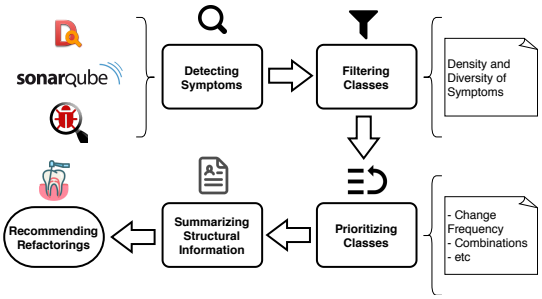

Fig. 3. Steps taken by a recommender technique based on our proposed requirements

*a) **Collecting multiple symptom categories**:* We observed in this study that, according to the theory proposed by

Sousa et al. [13], we should rely on two or more categories of symptoms to identify degraded classes. Each symptom category will reveal degradation aspects in dependability-related attributes that other symptom categories may not be able to capture. For example, while most code smells investigated in this study are related to modifiability and analyzability, the principle violations are mostly linked to modularity and reusability. Our results showed that high density and diversity in both symptom categories is usually associated with deep structural degradation that is difficult to fix even after successive refactorings. Therefore, a recommender technique should combine information about both symptom categories to provide precise recommendations. In this study, we used only symptoms provided by the Designite tool. However, recommendation techniques can implement their own detection strategies or can be based on symptoms provided by other tools, such as SonarQube [37] and SpotBugs [38].

*b) **Filtering classes**:* Due to several constraints, developers cannot waste time with the analysis of classes that do not need to be refactored. Thus, developers would benefit from a technique that automatically filters and selects only the classes that have the greatest chance of presenting structural degradation. Our findings revealed that combining density and diversity of symptoms such as code smells and principle violations can be an effective strategy for selecting degraded classes. The filter may consider only one category of symptom or it may combine multiple symptom categories. Some state-of-the-practice tools (e.g., SonarQube) already considers the density of symptoms to filter and to prioritize elements for refactoring. However, our study shows that diversity of symptoms should also be considered as degraded classes tend to present higher diversity of symptoms. The combination of both information could make filtering even more stringent, helping to save time for developers.

*c) **Prioritizing classes**:* Even after filtering classes, there will be too many candidates for refactoring in medium- and large-sized systems. Thus, it is fundamental to prioritize the refactoring candidates according to their relevance. Our study provided evidence that the core classes of the system should receive special attention because they are often involved with the main changes made throughout the system evolution. One of the characteristics that differentiate these core classes is the change frequency. Thus, this information can be used as a prioritization criterion for ranking the refactoring candidates. The symptom combinations may also be explored as a criterion to prioritize refactoring candidates. For instance, degraded classes can be prioritized based on the combinations of symptoms that developers, of the team, refactored more often in previous projects and previous tasks. The effectiveness of this idea is supported by findings from the literature (e.g., [36]).

Depending on the context, other factors can also be taken into account to prioritize classes for root canal refactoring. For instance, developers may want to prioritize refactoring of degraded classes that will be modified in future tasks. One of the challenges for the implementation of this criterion is the identification of which classes will be modified by future tasks. In our future work, we plan to address this challenge using the strategy adopted by Kim et al. [39] to locate bugs based on bug reports. We believe that our proposed criteria could be helpful for generating more accurate rankings, since existing prioritization criteria (e.g., [40], [41]) are unable to present consistent results for every system.

*d) **Summarizing structural information**:* One problem that often hinders the adoption of automated tools is the difficulty in understanding, exploring, and combining different symptoms. Knowing beforehand which combinations of symptoms are most recurrent, the tool can be prepared to explore the characteristics of the most recurrent combinations, providing detailed information about the types of degradation indicated by each combination. In addition, this information may be used by the tool to recommend specific refactoring operations, according to the refactoring operations that are usually associated with each combination. To provide a more readable and easy to understand summary, the recommender technique may apply an approach similar to the one designed by Moreno et al. [42]. This summarized and readable structural information would help developers to reason about each degraded class and to perform more effective refactorings.

Although we have focused our research mostly on maintainability, we believe that these requirements can be applied in the context of other attributes, such as reliability, security, and performance. We leave for future work the evaluation of this technique in the context of maintainability and other dependability-related attributes.

VI. Threats to Validity

The first threat to validity is regarding our dataset. Analyzing data from eight systems may not be enough for finding generalizable results. We mitigated this threat by selecting systems with different characteristics, developed in different platforms and with different practices. Another threat is related to the method used to find the root canal refactorings. For the systems of our industry partners, we may have missed refactorings that were not remembered by the developers or did not contain the searched keywords. To mitigate this threat, we asked for at least two developers of each system to provide us a list of root canal refactorings. All participating developers have knowledge about refactoring and have worked in the systems since their inception. Still related to the refactorings, it is possible that a task description demonstrates the intention to remove structural degradation but this does not occur in practice. We mitigate this threat by checking, with the help of developers, whether there was any discrepancy between the task description and the actual changes made.

In the context of open source systems, we may have missed several root canal refactorings after applying multiple filters. This may have influenced the number of outliers observed in the set of non-refactored classes. However, this has helped us to drastically reduce the possibility of false positives. In addition, we discarded systems that had a very low number of refactored classes after the filters were applied.

There is another threat related to the tools used for detecting symptoms. Aspects such as precision and recall may have influenced the results of this study. We mitigate this threat by selecting Designite, the only tool we know that is capable of detecting the investigated symptoms both in C# and in Java systems. Moreover, Designite has been used successfully in other recent studies [35], [43]. Finally, there is a threat regarding reproducibility as we are not allowed to publish detailed information about the refactorings and about the issue tracking system of our industry partners. Thus, to mitigate this threat, we created a replication package containing, among other information, the description of each root canal refactoring identified in their systems.

VII. Related Work

Symptoms of structural degradation. Different studies have proposed and evaluated techniques that use different symptoms for identifying structural degradation [20], [23], [25], [30], [44]. Many of them use only code smells as symptoms for the identification of structural degradation. Nevertheless, there is consistent evidence [19], [32], [45], [46] that individual code smells are not enough to accurately indicate the presence of structural degradation. For this reason, Oizumi et al. [20] proposed an alternative approach to identify structural degradation with the combination of multiple code smells. In their study, they investigated to what extent code smells could "flock together" to realize a structural degradation. They concluded that certain combinations of code smells are consistent indicators of structural degradation. Despite such result, in a subsequent study [21], they observed that their technique may not be effective in practice. In addition, they did not verify whether multiple code smells occur more frequently in classes that are actually refactored by developers.

As mentioned, Sousa's et al. [13] revealed that, in practice, developers use multiple heterogeneous symptoms to diagnose structural degradation. However, although they have used a systematic methodology to propose their theory, it was not validated. We overcome such limitation by analyzing the density and diversity of symptoms in refactored classes.

Incidence of symptoms as the system evolves. Mamdouh and Mohammad [43] conducted an study involving six open source systems to investigate if degradation symptoms are removed as the system evolves. They observed that, in general, the density of symptoms undergo few changes throughout the systems evolution. We conducted a similar analysis in our research question RQ2. Nevertheless, our study was focused on refactored classes, while they analyzed all classes of systems indistinctly.

Impact of refactoring on symptoms. The impact of refactoring in symptoms was also vastly studied by different researchers [14], [15], [47]. Similarly to us, Bavota et al. [14] investigated whether refactorings occur in classes with symptoms such as the code smells. Overall, they observed that none of the investigated symptoms are strong indicators of need for refactoring. In our case study, unlike them, both code smells and principle violations showed to be strong indicators of the need for refactoring. Finally, like us, they also observed that code smells are not usually removed by means of refactorings. However, we are the first to investigate the impact of refactorings on principle violations.

Cedrim et al. [15] also investigated the impact of refactorings on code smells. Different from us, they evaluated the impact of refactorings by each individual commit. As in the work of Bavota et al. [14], they collected the refactorings using an automated tool. Although this approach results in the collection of a larger volume of refactorings, it is not able to collect only the refactorings intentionally performed by the developers. Also, they did not analyze whether density and diversity of symptoms in refactored classes is different from the density and diversity in other classes.

Finally, a closely related short paper [48] presents preliminary results about the impact of refactorings in the density and diversity of symptoms. However, this work analyzed a set of only two systems. In addition, the relation of symptom combinations with structural degradation was not investigated.

In a nutshell, our work differs from the existing ones in the following points: (1) we investigated systems developed with C# and Java while most studies investigated only Java systems; (2) we did not use an automated approach for finding refactorings in the systems of our partners, while other studies used only automated approaches; (3) we restricted our analysis to root canal refactorings; (4) while most studies are focused only on code smells or on code metrics, we investigated two different symptom categories, and (5) we are the only ones to investigate whether combinations of different categories of symptoms can be used as indicators of structural degradation.

VIII. Conclusion

In this paper, we investigated whether symptoms of structural degradation appear with higher density and diversity in classes changed by root canal refactorings. After that, we investigated if root canal refactorings have a positive impact on the density and diversity of symptoms. We also investigated the combinations of symptoms that often occur in classes that were changed by root canal refactorings. To achieve our goal, we conducted a case study involving two C# systems from our industry partners and six open source Java systems.

Our results indicate that refactorings caused almost no positive impact on the density and diversity of any category of symptom. Nevertheless, we also observed that refactored classes have higher density and diversity of code smells when compared to other classes in the target systems. This result indicates that, despite not being removed by refactorings, some categories of symptom may be indeed strong indicators of structural degradation. Based on our insights, we proposed a set of four requirements for automatically recommending root canal refactorings. These requirements are related to the tasks of: (1) symptom collection, (2) filtering, (3) prioritization, and (4) summarizing of information about degradation. As future works, we aim at proposing and evaluating a semi-automated technique based on the requirements for recommending root canal refactorings.

REFERENCES

[1] M. Godfrey and E. Lee, "Secrets from the monster: Extracting Mozilla's software architecture," in *CoSET-00; Limerick, Ireland*, 2000, pp. 15–23.

[2] J. van Gurp and J. Bosch, "Design erosion: problems and causes," *Journal of Systems and Software*, vol. 61, no. 2, pp. 105 – 119, 2002.

[3] A. MacCormack, J. Rusnak, and C. Baldwin, "Exploring the structure of complex software designs: An empirical study of open source and proprietary code," *Manage. Sci.*, vol. 52, no. 7, pp. 1015–1030, 2006.

[4] S. Schach, B. Jin, D. Wright, G. Heller, and A. Offutt, "Maintainability of the linux kernel," *Software, IEE Proceedings* -, vol. 149, no. 1, pp. 18–23, 2002.

[5] Z. Li, P. Avgeriou, and P. Liang, "A systematic mapping study on technical debt and its management," *J. Syst. Softw.*, vol. 101, no. C, pp. 193–220, Mar. 2015. [Online]. Available: http://dx.doi.org/10.1016/j.jss.2014.12.027

[6] E. Lim, N. Taksande, and C. Seaman, "A balancing act: What software practitioners have to say about technical debt," *IEEE Software*, vol. 29, no. 6, pp. 22–27, Nov 2012.

[7] T. Besker, A. Martini, and J. Bosch, "Time to pay up: Technical debt from a software quality perspective," in *Proceedings of the XX Iberoamerican Conference on Software Engineering, Buenos Aires, Argentina, May 22-23, 2017.*, 2017, pp. 235–248.

[8] R. Martin, *Agile Principles, Patterns, and Practices*. New Jersey: Prentice Hall, 2002.

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

[10] E. Murphy-Hill and A. P. Black, "Refactoring tools: Fitness for purpose," *IEEE Software*, vol. 25, no. 5, pp. 38–44, Sep. 2008.

[11] E. Murphy-Hill, C. Parnin, and A. P. Black, "How we refactor, and how we know it," *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 5–18, Jan 2012.

[12] G. Bavota, A. De Lucia, A. Marcus, and R. Oliveto, *Recommending Refactoring Operations in Large Software Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 387–419.

[13] L. Sousa, A. Oliveira, W. Oizumi, S. Barbosa, A. Garcia, J. Lee, M. Kalinowski, R. de Mello, B. Fonseca, R. Oliveira, C. Lucena, and R. Paes, "Identifying design problems in the source code: A grounded theory," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. New York, NY, USA: ACM, 2018, pp. 921–931. [Online]. Available: http://doi.acm.org/10.1145/3180155.3180239

[14] G. Bavota, A. D. Lucia, M. D. Penta, R. Oliveto, and F. Palomba, "An experimental investigation on the innate relationship between quality and refactoring," *Journal of Systems and Software*, vol. 107, pp. 1 – 14, 2015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121215001053

[15] D. Cedrim, A. Garcia, M. Mongiovi, R. Gheyi, L. Sousa, R. de Mello, B. Fonseca, M. Ribeiro, and A. Chávez, "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*, ser. ESEC/FSE 2017. New York, NY, USA: ACM, 2017, pp. 465–475.

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

[17] International Organization for Standardization, *ISO-IEC 25010: 2011 Systems and Software Engineering-Systems and Software Quality Requirements and Evaluation (SQuaRE)-System and Software Quality Models*. ISO, 2011.

[18] A. Yamashita, M. Zanoni, F. A. Fontana, and B. Walter, "Inter-smell relations in industrial and open source systems: A replication and comparative analysis," in *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on*, Sept 2015, pp. 121–130.

[19] I. Macia, R. Arcoverde, E. Cirilo, A. Garcia, and A. von Staa, "Supporting the identification of architecturally-relevant code anomalies," in *ICSM12*, Sept 2012, pp. 662–665.

[20] W. Oizumi, A. Garcia, L. Sousa, B. Cafeo, and Y. Zhao, "Code anomalies flock together: Exploring code anomaly agglomerations for locating design problems," in *The 38th International Conference on Software Engineering; USA*, 2016.

[21] W. Oizumi, L. Sousa, A. Oliveira, A. Garcia, A. B. Agbachi, R. Oliveira, and C. Lucena, "On the identification of design problems in stinky code: experiences and tool support," *Journal of the Brazilian Computer Society*, vol. 24, no. 1, p. 13, Oct 2018. [Online]. Available: https://doi.org/10.1186/s13173-018-0078-y

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

[23] E. Murphy-Hill and A. P. Black, "An interactive ambient visualization for code smells," in *Proceedings of the 5th international symposium on Software visualization; Salt Lake City, USA*. ACM, 2010, pp. 5–14.

[24] A. Yamashita and L. Moonen, "Do code smells reflect important maintainability aspects?" in *ICSM12*, 2012, pp. 306–315.

[25] N. Moha, Y. Gueheneuc, L. Duchien, and A. L. Meur, "Decor: A method for the specification and detection of code and design smells," *IEEE Transaction on Software Engineering*, vol. 36, pp. 20–36, 2010.

[26] G. Suryanarayana, G. Samarthyam, and T. Sharmar, *Refactoring for Software Design Smells: Managing Technical Debt*. Morgan Kaufmann, 2014.

[27] T. Sharma and D. Spinellis, "A survey on software smells," *Journal of Systems and Software*, vol. 138, pp. 158 – 173, 2018.

[28] T. Sharma, "A taxonomy of software smells," May 2019. [Online]. Available: http://tusharma.in/smells/index.html

[29] E. Murphy-Hill and A. P. Black, "Seven habits of a highly effective smell detector," in *Proceedings of the 2008 International Workshop on Recommendation Systems for Software Engineering*, ser. RSSE '08. New York, NY, USA: ACM, 2008, pp. 36–40.

[30] D. M. Le, D. Link, A. Shahbazian, and N. Medvidovic, "An empirical study of architectural decay in open-source software," in *2018 IEEE International Conference on Software Architecture (ICSA)*, April 2018, pp. 176–17 609.

[31] L. Xiao, Y. Cai, R. Kazman, R. Mo, and Q. Feng, "Identifying and quantifying architectural debt," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: ACM, 2016, pp. 488–498. [Online]. Available: http://doi.acm.org/10.1145/2884781.2884822

[32] I. Macia, J. Garcia, D. Popescu, A. Garcia, N. Medvidovic, and A. von Staa, "Are automatically-detected code anomalies relevant to architectural modularity?: An exploratory analysis of evolving systems," in *AOSD '12*. New York, NY, USA: ACM, 2012, pp. 167–178.

[33]

[34] N. Tsantalis, M. Mansouri, L. M. Eshkevari, D. Mazinanian, and D. Dig, "Accurate and efficient refactoring detection in commit history," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. New York, NY, USA: ACM, 2018, pp. 483–494.

[35] T. Sharma, P. Mishra, and R. Tiwari, "Designite: A software design quality assessment tool," in *Proceedings of the 1st International Workshop on Bringing Architectural Design Thinking into Developers' Daily Activities*, ser. BRIDGE '16. New York, NY, USA: ACM, 2016, pp. 1–4. [Online]. Available: http://doi.acm.org/10.1145/2896935.2896938

[36] M. Abbes, F. Khomh, Y. Gueheneuc, and G. Antoniol, "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*, 2011, pp. 181–190.

[37] G. Campbell and P. P. Papapetrou, *SonarQube in action*. Manning Publications Co., 2013.

[38] SpotBugs, "Spotbugs: Find bugs in java programs," May 2019. [Online]. Available: https://spotbugs.github.io/index.html

[39] D. Kim, Y. Tao, S. Kim, and A. Zeller, "Where should we fix this bug? a two-phase recommendation model," *IEEE Transactions on Software Engineering*, vol. 39, no. 11, pp. 1597–1610, Nov 2013.

[40] S. Vidal, E. Guimaraes, W. Oizumi, A. Garcia, A. D. Pace, and C. Marcos, "Identifying architectural problems through prioritization of code smells," in *SBCARS16*, Sept 2016, pp. 41–50.

[41] S. Vidal, W. Oizumi, A. Garcia, A. D. Pace, and C. Marcos, "Ranking architecturally critical agglomerations of code smells," *Science of Computer Programming*, vol. 182, pp. 64 – 85, 2019. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167642318303514

[42] L. Moreno, J. Aponte, G. Sridhara, A. Marcus, L. Pollock, and K. Vijay-Shanker, "Automatic generation of natural language summaries for java classes," in *2013 21st International Conference on Program Comprehension (ICPC)*, May 2013, pp. 23–32.

[43] M. Alenezi and M. Zarour, "An empirical study of bad smells during software evolution using designite tool," *i-Manager's Journal on Software Engineering*, vol. 12, no. 4, pp. 12–27, Apr 2018.

[44] R. Mo, Y. Cai, R. Kazman, and L. Xiao, "Hotspot patterns: The formal definition and automatic detection of architecture smells," in *Software Architecture (WICSA), 2015 12th Working IEEE/IFIP Conference on*, May 2015, pp. 51–60.

[45] I. Macia, R. Arcoverde, A. Garcia, C. Chavez, and A. von Staa, "On the relevance of code anomalies for identifying architecture degradation symptoms," in *CSMR12*, March 2012, pp. 277–286.

[46] W. Oizumi, A. Garcia, T. Colanzi, A. Staa, and M. Ferreira, "On the relationship of code-anomaly agglomerations and architectural problems," *Journal of Software Engineering Research and Development*, vol. 3, no. 1, pp. 1–22, 2015.

[47] A. Chávez, I. Ferreira, E. Fernandes, D. Cedrim, and A. Garcia, "How does refactoring affect internal quality attributes?: A multi-project study," in *Proceedings of the 31st Brazilian Symposium on Software Engineering*, ser. SBES'17. New York, NY, USA: ACM, 2017, pp. 74–83.

[48] A. Eposhi, W. Oizumi, A. Garcia, L. Sousa, R. Oliveira, and A. Oliveira, "Removal of design problems through refactorings: Are we looking at the right symptoms?" in *Proceedings of the 27th International Conference on Program Comprehension*, ser. ICPC '19. Piscataway, NJ, USA: IEEE Press, 2019, pp. 148–153. [Online]. Available: https://doi.org/10.1109/ICPC.2019.00032