Don't Forget the Exception!

Considering Robustness Changes to Identify Design Problems

Anderson Oliveira[*], João Correia[*], Leonardo Sousa[†], Wesley K. G. Assunção[*][§], Daniel Coutinho[*], Alessandro Garcia[*], Willian Oizumi[*], Caio Barbosa[*], Anderson Uchôa[‡], Juliana Alves Pereira[*]
[*]Informatics Department – Pontifical Catholic University of Rio de Janeiro (PUC-Rio), Rio de Janeiro, Brazil
[†]Department of Electrical and Computer Engineering - Carnegie Mellon University, Pittsburgh, United States
[§]Institute of Software Systems Engineering – Johannes Kepler University (JKU), Linz, Austria
[‡]Federal University of Ceará (UFC), Itapajé, Brazil

Abstract*—Modern programming languages, such as Java, use exception-handling mechanisms to guarantee the robustness of software systems. Although important, the quality of exception code is usually poor and neglected by developers. Indiscriminate robustness changes (*e.g.*, the addition of empty catch blocks) can indicate design decisions that negatively impact the internal quality of software systems. As it is known in the literature, multiple occurrences of poor code structures, namely code smells, are strong indicators of design problems. Still, existing studies focus mainly on the correlation of maintainability smells with design problems. However, using only these smells may not be enough since developers need more context (*e.g.*, system domain) to identify the problems in certain scenarios. Moreover, these studies do not explore how changes in the exceptional code of the methods combined with maintainability smells can give complementary evidence of design problems. By covering both regular and exception codes, the developer can have more context about the system and find complementary code smells that reinforce the presence of design problems. This work aims to leverage the identification of design problems by tracking poor robustness changes combined with maintainability smells. We investigated the correlation between robustness changes and maintainability smells on the commit history of more than 160k methods from different releases of 10 open-source software systems. We observed that maintainability smells can be worsened or even introduced when robustness changes are performed. This scenario mainly happened for the smells *Feature Envy*, *Long Method*, and *Dispersed Coupling*. We also analyzed the co-occurrence between robustness and maintainability smells. We identified that the *empty catch block* and *catch throwable* robustness smells were the ones that co-occurred the most with maintainability smells related to the *Concern Overload* and *Misplaced Concern* design problems. The contribution of our work is to reveal that poor exception code, usually neglected by developers, negatively impacts the quality of methods and classes, signaled by the maintainability smells. Therefore, existing code smell detecting tools can be enhanced to leverage robustness changes to identify design problems.

***Index Terms*—empirical study, design problems, robustness, exception handling, code smells**

I. Introduction

Exception-handling mechanisms, commonly utilized in modern programming languages, promote the robustness and stability of the software systems [1], [2]. The proper use of these mechanisms aims to guarantee the software integrity when unexpected events or behaviors happen [3]. However, most software systems do not offer detailed documentation of the design decisions related to the exception handling implementation [4], [5]. This lack of information encourages developers to focus solely on the normal behavior of the software system [6], leaving the exception handling behavior poorly implemented [1], [7] or even neglecting the exceptional code [5], [8]. This neglection can impact the software system's robustness and might also be a sign of problems in the design of the software system [9]–[11].

A design problem results from one or more design decisions that negatively impact the system's non-functional requirements (NFRs), which include robustness and maintainability [12]–[14]. The most critical design problems often affect how the system is modularized into components and how these components interact with each other. To identify such design problems, developers have to analyze several elements (*e.g.*, classes and packages), which is a laborious activity. Thus, in this study, we focus on design problems related to system modularity. An example is the design problem *Concern Overload* that occurs when a component is responsible for realizing multiple concerns [15]. This design problem can also make it difficult for developers to know which concerns they should focus on to create the proper exception-handling logic. Furthermore, when neglected, these design problems can lead the system to undesired consequences such as irrecoverability from the faults, increasing the maintenance cost, and speeding up software erosion [16]–[18].

Since these design problems can affect multiple NFRs, they must be identified and removed from the systems as soon as possible [19]. Multiple studies explored the use of maintainability smells as symptoms of design problems [20]–[22]. A recent study presented a catalog with patterns of maintainability smells that indicate multiple design problems [23]. However, more smells may be considered, since developers may need more context regarding the class, component, or system. Moreover, this can cause an incomplete identification of the design problem. In addition, these studies do not explore how exceptional code (*i.e.*, code inside the catch block) can be combined with maintainability smells to identify design problems. Given the different natures of normal and exceptional code, it is possible that they can complement each other to identify design problems. Developers could benefit from tools that, besides detecting multiple symptoms, also combine them for revealing design problems [21]. Therefore, in this study, we aim to understand how the poor changes in exception handling can be used as symptoms of design problems and how they can be combined with maintainability smells to identify these problems.

For this study, we consider the changes related to robustness as the changes performed within the catch blocks, since the exceptional code is in this part of the implementations. We analyzed over 160k class methods from 10 open-source software systems. For our analysis, we collected (i) maintainability smells based on insights from a related study [23], (ii) robustness changes in methods, and (iii) robustness smells. In the first analysis, we explored how robustness changes could correlate with maintainability smells. Furthermore, we looked for maintainability smells that were introduced through robustness changes. Our goal was to identify how these two factors were correlated. In a second analysis, we identified whether robustness changes could have a negative impact on classes with methods that underwent this kind of change. Hence, we could understand how these changes impacted the method's degradation. Finally, we investigated which poor robustness changes (signaled through robustness smells) could be used with patterns of maintainability smells to identify design problems.

We identified that a method with robustness changes would also be affected by a *Feature Envy*, *Dispersed Coupling*, or *Long Method* maintainability smells. By manually analyzing the methods with robustness changes, we identified cases where these changes introduced the maintainability smells, especially the *Feature Envy*. We also identified that classes with robustness changes had a higher density of smells when compared to classes without such changes. Hence, these robustness changes could indicate these classes' degradation. We also observed that the robustness smells *catch of generic exceptions* and *empty catch block* tended to co-occur with the patterns of maintainability smells that help in the identification of design problems such as *Concern Overload* and *Unwanted Dependency*.

Our results support the community in understanding how poor robustness changes can complement the information given by maintainability smells to identify design problems. In practice, developers can use this information to be aware that even minor modifications made to catch blocks can potentially affect or reveal underlying design issues within the system. Moreover, by identifying the robustness smells as a new symptom of design problems, tools can be developed to reinforce the presence of these problems.

II. Background

This section describes the concepts for understanding the relationship between poor robustness changes, and maintainability code smells, so they can be used to identify design problems.

A. Exception Handling

An *exception* is an unexpected event that occurs during the execution of a program, interrupting its normal behavior [24], [25]. Usually, exceptions manifest through errors. When an error happens, the method where it appears creates an *exception object* including information, such as the program's state and details about the error. The *exception object* is then delivered to the runtime system, completing the initial routine that *throws an exception* [24].

Developers use exception-handling mechanisms to ensure that the system will be in a consistent state, even after errors that occur at runtime [25], assuring that the system will be robust. In Java, any code snippet likely to throw exceptions must be placed inside a `try-catch` block. The `try` block defines the `normal code` of a method. Respectively, each `try` is followed by one or more `catch` blocks, which handle specific exceptions thrown inside the associated `try`. The `catch` block has the code responsible for handling the specific exception types that can emerge from the enclosed instructions in the `try` block. In addition, the `catch` block encompasses the method's `exceptional code`. These blocks can also be followed by a `finally` block that always executes after the `try` and the `catch` if an exception is raised.

B. Robustness Changes and Code Smells

In this study, we consider as *robustness changes* those performed within the catch block, since the poor use of exception-handling mechanisms can harm the software robustness [9]–[11]. We consider the *poor robustness changes* as the changes in the catch block that are affected by robustness smells. An example of a robustness smell is the *Empty Catch Block*, which occurs when a developer creates a catch statement but leaves its content empty. This is a problem because the catch block should be where the developer handles exceptions thrown by the system, which is not occurring. Unfortunately, developers tend to ignore these smells and only deal with them reactively when they face errors [1]. Albeit ignored, robustness smells may indicate the decay of the software [26]. Other indicators of software degradation are the maintainability smells [27]–[29], that can also signal design problems [21], [23], [30]–[32]. However, they are not always sufficient for this identification task. Hence, since both maintainability and robustness smells can indicate design problems, combining them could reinforce its presence.

To understand this relationship between poor robustness changes and maintainability smells, let us consider the following example illustrated by Figure 1[1]. This figure displays the HTTPHandler system, which implements an HTTP client. Figure 1(a) shows the layered architectural style followed by the architecture of the system we analyzed, which consists of four layers: `Interface`, `Cache`, `Connection`, and `Control`. According to the architectural style, each layer is in charge of its responsibility, thus following the *Separation of Concerns* (SoC) principle [33], [34].

[1]For simplicity, we adapted this example from one of the analyzed systems.

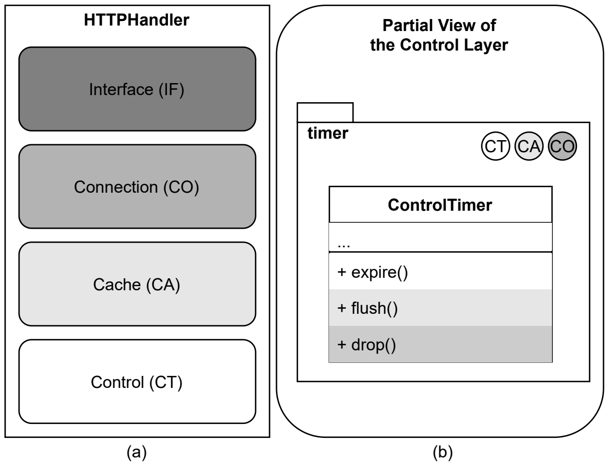

Fig. 1. Partial View of the HTTPHandler System

Like other design decisions, one might expect that every exception handled in each layer is directly related to the responsibility (*i.e.*, the concern) implemented in the layer. However, this is not always the case. For example, Figure 1(b) shows the `ControlTimer` class with three methods, where `flush()` and `drop()` implement responsibilities from two other layers: `Cache` and `Connection`, respectively. When we look at the `expire()` method (illustrated in Listing 1), we see a code snippet that shows a change was performed in the class in which the developer tried to handle an error. However, as observed in the Listing 1, it catches a generic exception to handle the error, which is a bad practice since it hides the error that the catch block should have captured and handled. In addition, within the catch block, nothing is handled, only logged with a generic message.

Listing 1
EXAMPLE OF METHOD CATCHING A GENERIC EXCEPTION

```
public class ControlTimer{
  (...)
  public void expire(Control ct,
                     Connection conn){
    try {
      (...)
    } catch (Throwable t) {
      log.debug(ct.getServiceKey()
      + "and" + conn.getUrl())
    }
}
```

We hypothesize that this happened due to the several concerns intermingled in the implementation of that class. This method is also affected by the code smell *Intensive Coupling*, which indicates that this class has a high coupling with the other classes in the module. At the time, the developer could not know which specific exception the block should handle. As aforementioned, the class implements concerns from other layers. Consequently, the many concerns caused the developer to neglect proper exception handling in the method. Furthermore, besides just logging the error instead of handling it, the log method also calls multiple methods from the other classes, introducing a *Feature Envy* and reinforcing the *Intensive Coupling* smell that the method already had. Additionally, these two maintainability smells are part of a pattern that can strongly indicate the design problem *Concern Overload* [23], [35]. The patterns considered in our study are presented in Table I. Therefore, in this example, the poor robustness change and the smell pattern together reinforced the design problem's presence. Moreover, this study explores the potential of using these co-occurring factors to identify design problems.

TABLE I
DESIGN PROBLEMS AND THEIR SMELL-PATTERNS

Design Problem	Smells Pattern
Ambiguous Interface	Long Method, Feature Envy, and Dispersed Coupling
Cyclic Dependency	Intensive Coupling and Shotgun Surgery
Concern Overload	Divergent Change, Feature Envy, God Class/Complex Class, Intensive Coupling, Long Method, and Shotgun Surgery
Fat Interface	Shotgun Surgery or Divergent Change, Dispersed Coupling, and Feature Envy
Misplaced Concern	God Class/Complex Class, Dispersed Coupling, Feature Envy, and Long Method
Scattered Concern	Dispersed Coupling, Divergent Change, Feature Envy, God Class/Complex Class, Intensive Coupling, and Shotgun Surgery
Unwanted Dependency	Feature Envy, Long Method, and Shotgun Surgery

III. RELATED WORK

Multiple studies have explored the relationship between maintainability smells and design problems [15], [21], [23], [31], [32], [35], [36]. Oliveira *et al.* [32] identified the criteria that developers used to prioritize the classes that, with respect to their degradation, were most critical in a system. The authors found that developers tend to consider the quantity and diversity of maintainability smells in a class as an important factor in its prioritization. Thus, the developers should focus their effort on these degraded classes. The limitation of these studies regards using only the maintainability smells as the symptom of design problems. Sousa *et al.* [21] developed a theory on how developers identify design problems. They identified the developers' use of multiple symptoms, including maintainability smells and violation of non-functional requirements (NFRs). Thus, our study explores the NFR robustness (by considering the robustness changes) and its combination with the symptom maintainability smell.

Studies have shown that maintainability smells can be indicators of design problems [22], [30], [37], [38]. A pattern of maintainability smells (*i.e.* groups of one or more types of smells) can be a strong indicator of the presence of design problems [23], [35]. For instance, when the maintainability smells *Feature Envy* and *Intensive Coupling* occur together, they indicate that the method, affected by these smells, is more interested in data from others, calling many methods from unrelated classes [28]. Hence, these smells can indicate the presence of a *Scattered Concern* design problem [39]. However, these smells may not be sufficient to confirm the presence of design problems [23]. Among the reasons, more information regarding the context of the method and class is needed, such as the system's domain. Thus, in this study, we expand the use of these smells with information on the exceptional code, which can introduce more context (*e.g.*, through the type of exception handled) about the method and class analyzed. Best of our knowledge, the relationship between robustness changes and code smells still needs to be explored.

The use of exception-handling by developers is extensively explored [26], [40]–[46]. Melo *et al.* qualitatively analyzed the use of exception-handling guidelines by surveying 98 developers [40]. The authors identified that in 70% of the developers' responses, there was a guideline to be followed. However, these guidelines tend to be implicit and undocumented. Cacho *et al.* [46] presented a study with C# projects where they identify the relationship between software systems changes and their robustness. They analyzed 119 software versions extracted from 16 systems from different domains. They identified that C# developers often unconsciously traded robustness for maintainability in various program categories. Finally, it was identified that a high number of uncaught exceptions were also introduced when the catch blocks were changed. Other studies explored the faults and anti-patterns commonly related to the exception-handling in the code [26], [42]. In this study, we explore how poor changes in the exceptional code can impact the maintainability smells of the system.

IV. Study Design

In summary, we analyzed commits from 10 open-source software systems. We started by collecting maintainability smells, robustness smells, and robustness changes in the commits between selected pairs of releases from those systems. We detail the remainder of the study design as follows. It is also summarized in Figure 2.

A. Research Questions

Our goal is to *understand how poor robustness changes can be combined with maintainability smells as complementary symptoms of design problems*. With this goal in mind, we defined three research questions.

RQ$_1$: How often do robustness changes co-occur with maintainability smells?

We hypothesize that robustness changes and maintainability smells can be considered in combination (Section II-B) and have the potential to reveal design problems. Therefore, we performed a statistical analysis of the correlation between robustness changes and maintainability smells using Fisher's exact test [47]. We started by dividing methods regarding robustness changes they underwent between two releases and the presence of maintainability smells in these methods.

Alongside understanding whether those robustness changes and maintainability smells correlate, we also want to understand if these changes could introduce the maintainability smells. Thus, we selected the methods with robustness changes and verified whether the maintainability smells were introduced during those changes (see Section IV-D - Step 1).

RQ$_2$: What impact can robustness changes have on the degradation of classes?

Once we answer RQ$_1$, we should have indications of whether robustness changes and maintainability smells are correlated and if these changes can introduce smells. After knowing that, we aim to understand whether performing robustness changes to methods can impact the degradation of classes. For RQ$_2$, similarly to previous work [21], [31], [32], we considered degradation as the number of maintainability smells a method has (*a.k.a.* density of smells). Developers use this metric to prioritize classes with a high number of different smells when looking for design problems [32] (see Section IV-D - Step 2).

RQ$_3$: How do robustness smells give evidence of design problems?

After identifying if robustness changes can have a negative impact on maintainability smells, we also want to investigate further how we can use poor robustness changes as symptoms of design problems. To answer RQ$_3$, we analyze when robustness smells co-occur with maintainability smells that are part of patterns that indicate design problems (see Section II-B). With this RQ, we aim to identify which robustness smells can complement these patterns of maintainability smells, reinforcing the presence of design problems (see Section IV-D - Step 3).

Consequently, by addressing all three research questions, we will gain insights into how and which poor robustness changes can be used with the patterns of maintainability smells to assist developers in detecting design problems.

B. Defining Subject Software Systems and Releases

We firstly selected projects from a list of projects used in related studies [22], [23], [36], [48]. Then, we filtered the systems to be used in this study using the following criteria:

Open Source. We selected open-source software systems to allow full replication of this work, since access to closed software systems is usually very limited. Open-source software systems often rely on version control systems (*e.g.*, Git) to track the evolution of their source code. This gives us access to the complete history of changes (*i.e.*, commits) to the source code of a software system, allowing us to perform the multiple analysis required to address our three research questions.

Java Language. We consider systems written in Java since it provides an exception-handling mechanism designed to help developers to build robust systems [49]. For instance, the use of checked exceptions forces developers to write handlers for certain errors. In addition, we selected projects from different domains that are more inclined to follow robustness requirements (*e.g.*, distributed computing, and big data processing).

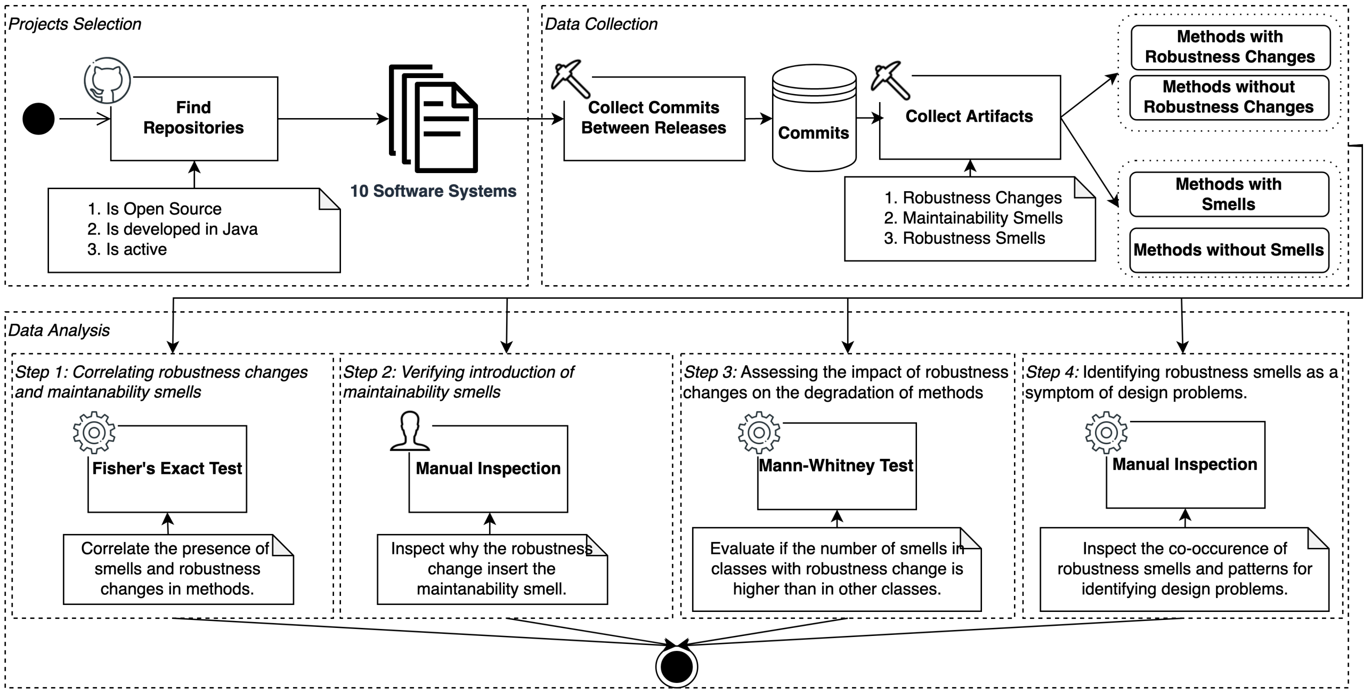
Projects Selection
Find Repositories
1. Is Open Source
2. Is developed in Java
3. Is active
10 Software Systems
Data Collection
Collect Commits Between Releases
Commits
Collect Artifacts
1. Robustness Changes
2. Maintainability Smells
3. Robustness Smells
Methods with Robustness Changes
Methods without Robustness Changes
Methods with Smells
Methods without Smells
Data Analysis
Step 1: Correlating robustness changes and maintanability smells
Fisher's Exact Test
Correlate the presence of smells and robustness changes in methods.
Step 2: Verifying introduction of maintanability smells
Manual Inspection
Inspect why the robustness change insert the maintanability smell.
Step 3: Assessing the impact of robustness changes on the degradation of methods
Mann-Whitney Test
Evaluate if the number of smells in classes with robustness change is higher than in other classes.
Step 4: Identifying robustness smells as a symptom of design problems.
Manual Inspection
Inspect the co-occurence of robustness smells and patterns for identifying design problems.

Fig. 2. Workflow of our Study Design.

Moreover, Java has a wide availability of static analysis tools and libraries that can automatically identify source code problems [50]–[52].

Active Software Systems. To consider a software system *active* we considered four criteria: (i) its Git repository has more than 1,000 commits, (ii) should contain commits pushed a month before our data collection period, (iii) should have recent discussions on pull requests and issues, and (iv) should have recent releases. These criteria ensure that the systems still have a development activity and relevance to the developers participating. After following these criteria, we settled on 10 software systems: APM Agent, Dubbo, Elasticsearch Hadoop, Fresco, Netty, Spring Boot, Spring Security, Spring Framework, RxJava, and OkHttp.

Given that the releasing strategy can differ across software systems, we selected start and end releases that span at least 1,000 commits. Our goal is to avoid the bias of having only a few robustness changes and thus being unable to perform reliable conclusions. Therefore, we can detect subtle patterns and correlations in the data that may not have been apparent with fewer robustness changes. This led us to more generalizable, accurate, and robust conclusions. In Table II, we present the systems and the releases selected.

C. Collecting Artifacts Data

To answer our RQs, we collected robustness changes, maintainability smells, and robustness smells from 10 target systems. We first collected the maintainability smells using *Organic* [52]. *Organic* is a static code analyzer that collects software metrics [28] for maintainability smells detection. For our first two RQs, we only consider method-level smells (*e.g.*, Feature Envy, and Dispersed Coupling). We selected these smells since they are part of the patterns that help identify design problems (see Table I). When evaluating those design problem patterns (*i.e.*, RQ$_3$), we also consider class-level maintainability smells (*e.g.*, God Class, and Complex Class).

TABLE II
DETAILS ON THE SOFTWARE SYSTEMS ANALYZED

Project	Start Release	End Release	Commits		Methods
			With Any Changes	With Robustness Changes	
apm-agent-java	v0.5.0	v1.28.0	1,025	484	9,949
dubbo	dubbo-2.6.12	dubbo-3.0.0	1,382	1,069	26,613
elasticsearch-hadoop	v1.3.0.M1	v8.0.0	1,120	503	3,894
fresco	v1.0.0	v2.6.0	1,165	791	7,700
netty	netty-4.1.31.Final	netty-4.1.75.Final	995	694	19,067
okhttp	parent-2.3.0	parent-3.14.0	586	455	5,933
RxJava	v1.0.10	v3.1.0	1,169	799	28,004
spring-boot	v2.7.6	v3.0.0	1,380	618	19,330
spring-framework	v5.3.24	v6.0.0	1,119	794	38,429
spring-security	5.1.0.RELEASE	5.6.0.RELEASE	1,426	690	10,062

Commits with any changes: Number of commits with any kind of change
Commits with robustness changes: Number of commits with robustness changes
Methods: Total number of methods between releases

To collect robustness changes, we developed a Python script that calculates the difference between two commits and identifies any change within the catch block of a method body. We also filtered out the test code. Finally, we collected nine robustness smells (*e.g.*, empty catch block, and catch generic exception), using *PMD* [50], which is a static code analyzer, often used to find flaws in source code. Details about the smells and the script developed can be found in our replication package [53].

D. Data Analysis

In this section, we present the steps to the analyses executed to answer our research questions as follows (see Figure 2).

Step 1: Correlating robustness changes and maintainability smells: To answer RQ$_1$, we first divided methods as follows: (i) methods with at least one robustness change, (ii) methods that changed but did not have any robustness change, (iii) methods that were affected by at least one maintainability smell, and (iv) methods that were unaffected by maintainability smells. Considering this division, we defined the following pair groups to be used in Fisher's exact test [47].

- **Smelly + Changed (SML + CH)**: Methods with maintainability smells and robustness changes
- **Smelly + Not Changed (SML + NoCH)**: Methods with maintainability smells without robustness changes
- **Not Smelly + Changed (SML + CH)**: Methods without maintainability smells and with robustness changes
- **Not Smelly + Not Changed (NoSML + NoCH):** Methods without maintainability smells and without robustness changes

With this statistical test, we can define whether robustness changes performed to a method are related to the presence of maintainability smells in that same method. To understand how these two factors could be correlated, we performed a manual inspection analysis of randomly selected 206 methods (equally distributed between the authors) with maintainability smells and robustness changes. First, we selected the methods in which robustness change and maintainability smell were present. Then, for each analysis, the participants filled out a form detailing how the robustness changes could be related to the maintainability smells. All participants have experience with exception handling and code smell detection and at least a Master's degree in Software Engineering. Whenever authors had even a slight doubt about the validity of a case, they took note of it, and another author was asked to confirm their findings. This process allowed a comprehensive review of each case, ensuring that mistakes were avoided and the verdicts were reliable. Details about this manual inspection and the protocol used can be found in our replication package [53].

Step 2: Verifying the introduction of maintainability smells. In this step, we complemented the analysis of RQ$_1$ with a qualitative analysis now looking at cases in which maintainability smells were introduced through robustness changes. For that purpose, we identified the methods by which this event occurred and inspected them, looking for why the robustness change introduced the smells. For this analysis, we prioritized cases with a higher quantity of smells introduced. Furthermore, we analyzed 30 methods, equally divided between the software systems. To select these methods, we divided them into quartiles, considering the number of smells introduced in that commit. Furthermore, four authors analyzed the methods presented in the 1st, 2nd and 3rd (together), and 4th quartiles.

Step 3: Assessing the impact of robustness changes on the degradation of methods: To answer RQ$_2$, we evaluate whether the number of maintainability smells in a class with methods that underwent robustness changes is significantly higher than the number of smells in classes that do not have methods with this kind of change. First, we compared the number of smells in both groups of classes. Next, we considered the total number of maintainability smells a class has in the end release, considering the smells on the method-level. Finally, to analyze the statistical significance of our results, we applied the Mann-Whitney test [54].

TABLE III
THE RELATION OF MAINTAINABILITY SMELL (S) AND ROBUSTNESS CHANGES (C) - ($p < 0, 05$)

Software System	Odds Ratio	SML + CH	SML + NoCH	NoSML + CH	NoSML + NoCH
apm-agent-java	4.755	50	941	99	8,859
dubbo	6.995	219	1,956	385	24,053
elasticsearch-hadoop	2.377	33	139	338	3,384
fresco	5.243	25	35	916	6,724
netty	6.902	45	63	1,778	17,181
okhttp	3.351	32	75	658	5,168
RxJava	5.938	48	105	1,991	25,860
spring-boot	2.192	10	62	1,320	17,938
spring-framework	4.504	31	73	3,302	35,023
spring-security	2.458	50	234	782	8,996

Odds Ratio: Odds ratio identified through the Fisher test
SML + CH: # of methods with maint. smells and robust. changes
SML + NoCH: # of methods with maint. smells and without robust. changes
NoSML + CH: # of methods without maint. smells and with robust. changes
NoSML + NoCH: # of methods without maint. smells and robust. changes

Step 4: Identifying robustness smells as a symptom of design problems. To answer RQ$_3$, we analyzed the methods that underwent robustness changes between the pair of releases defined (see Section IV-B), resulting in 4,758 methods. First, we collected the robustness smells for each method and verified whether the robustness smells co-occurred with the patterns for identifying design problems. For each smell, we verified how many times a pattern co-occurred with the robustness change. Finally, we analyzed 80 cases in which this event occurred to understand whether this robustness change and the pattern could be related to a possible design problem.

V. ANALYSIS AND RESULTS

This section presents and discusses the results to answer our research questions. First, we explored the correlation between robustness changes and maintainability smells. Next, we explored the density of smells compared to classes with robustness changes and without this kind of change. Finally, we explored the type of robustness changes that lead to the presence of design problems.

A. How often do robustness changes co-occur with maintainability smells?

To answer **RQ$_1$**, we analyzed if there is a correlation between robustness changes and maintainability smells. For that purpose, we first relied on Fisher's exact test [47] (see Section IV-A). Table III provides an overview of this analysis.

Chances of methods with robustness changes being affected by maintainability smells. Considering the commits between the two releases, we computed a correlation for the methods' robustness changes and maintainability smells. In our analysis, all systems had $p < 0.05$, meaning that there is an association between the occurrence of robustness change and maintainability smell in methods. The odds ratio assumes values from 0 to infinity. When the OR is greater than 1, it indicates that methods that underwent robustness changes will likely be affected by maintainability smells.

In this test, we used the robustness change as a predictor and maintainability smell as the outcome. For instance, for the system Dubbo, the Fisher test computed an odds ratio of 6.995. This means that the odds of a maintainability smell occurring are 6.995 higher in cases where the robustness change occurs than in cases where this change does not affect the method. Thus, this indicates a strong correlation between robustness change and maintainability smells. In this example, 26,613 methods were changed between the releases, thus considered in the analysis. From these methods, 604 had robustness changes (219 with maintainability smells and 385 without them). Also, we discuss the cases in systems with the highest odds ratio. For instance, in Table III, the lowest odds ratio is 2.192 in the spring-boot system; hence methods with robustness changes are at least twice as likely to have maintainability smells compared to methods without such changes. Therefore, developers should consider these changes when introducing or changing exceptional code.

Finding 1: When robustness changes are performed, developers should be aware of the maintainability smells introduced or already present in the method.

To better understand which maintainability smells correlated with the robustness changes, we did an analysis specifically for each smell. Table IV presents the results for this analysis. Each column represents a maintainability smell. Each cell represents the odds ratio for the specific smell. When we did not reach statistically relevant results, we filled the cell with 'NR' (Not statistically Relevant).

We can highlight three smells: *Feature Envy*, *Dispersed Coupling*, and *Long Method*. The results of these three smells were statistically significant in all software systems, except for *spring-boot*. Dispersed Coupling had an odds ratio higher than five in eight systems, while Feature Envy and Long Method had in six and eight systems, respectively. That means a strong correlation between the robustness change and the presence of these smells on methods. The smell Shotgun Surgery had an odds ratio of 21.686 on *RxJava*. However, the results for this smell were relevant only for five systems.

Finding 2: Methods that underwent robustness changes mostly co-occur with the maintainability smells *Dispersed Coupling*, *Feature Envy*, and *Long Method*.

We decided to manually evaluate cases involving the three maintainability smells mentioned with the highest odds ratio. For that, we selected a sample of 206 methods. To select them, we chose the commit in which the class had methods affected by the maintainability smells and had the robustness change. On a manual analysis, we identified that in at least 74 (35.91%), the maintainability smell was directly related to the robustness change, meaning that either the robustness change introduced the smell or worsened the smell already present in the method. More details about this validation can be found in our replication package [53]. Besides these direct cases, we observed that this relation could also be indirect.

Indirect relation between robustness changes and maintainability smells. One example is case in the method `drainLoop` from *RxJava* [55]. We identified that the *Feature Envy* and *Message Chain* smells were related to the operation performed, which was a parsing that resulted in new exceptions being raised. Therefore, there is an indirect relation between the smells and the exceptional change. The indirect relation can also be related to *Long Methods*. In this scenario, it is natural that the excess of code statements will be more complex, leading to multiple catch implementations of generic catch blocks or even empty catch blocks.

Developers can use the logging mechanism as a source of information for minor exception-handling improvements. Changes in the logging regarding the errors handled were also constant in our analysis. We observed that 37 (from 206) cases were related to logging in the catch block. At first sight, this may be seen as a bad practice, as apparently nothing is handled. However, the log messages can serve as documentation for the developers, since they can provide information to the developers that will maintain this exceptional code. Furthermore, when a developer inserts a log message with proper information regarding the exception, it can indicate that he intends to improve the code in the future. However, in our analysis, we observed that the inclusion of *Feature Envy* was mainly related to changes in the log messages, as we described above on a method from *dubbo*. Hence, developers need to be careful when adding these messages since they can add new maintainability smells. For instance, the developer should give more context in the log message, which can be reinforced using more specific exceptions rather than generic ones. In addition, the developer needs to be sure about the log level for the messages, which would avoid smells such as the *Feature Envy*.

Maintainability smells can be included during robustness changes. We identified that from the 4,758 methods with robustness changes, 774 (16.26%) co-occurred with the introduction of maintainability smells. Therefore, we analyzed the cases in which this happened. We focused on cases with the introduction of *Feature Envy*, *Dispersed Coupling*, and *Long Method*, which had statistically relevant results in 9 out of 10 systems (see Table IV). On *dubbo*, we observed a case in which the developer added log messages to the catch block. However, these messages heavily relied on calls to methods from other classes, adding a *Feature Envy* and a *Dispersed Coupling*. We observed a similar case on *RxJava*, but in which the catch block introduced the smell *Dispersed Coupling*. In addition, the block called multiple classes to close

TABLE IV
THE RELATION OF SPECIFIC SMELLS (S) AND ROBUSTNESS CHANGES (C). (P $< 0,05$)

Software System	Odds Ratio							
	Brain Method	Dispersed Coupling	Feature Envy	Intensive Coupling	Long Method	Message Chain	Long Parameter List	Shotgun Surgery
spring-security	NR	11.002	2.477	NR	6.291	NR	NR	11.206
apm-agent-java	NR	6.952	8.153	9.058	13.431	2.482	NR	NR
dubbo	10.330	14.171	8.198	10.191	9.305	3.168	1.936	10.330
elasticsearch-hadoop	NR	7.214	2.968	4.607	3.971	3.002	NR	8.032
fresco	NR	9.299	8.878	NR	7.205	NR	2.780	NR
netty	NR	13.033	6.126	9.930	14.293	NR	2.461	NR
okhttp	NR	3.862	5.933	9.743	9.059	2.929	NR	9.011
RxJava	NR	10.049	9.820	NR	11.272	NR	3.387	21.686
spring-boot	NR	NR	NR	NR	NR	NR	3.682	NR
spring-framework	NR	15.411	4.517	NR	5.447	3.425	4.124	NR

NR: Not statistically relevant

the resources used. On *okhttp*, we identified a case of bad practice on exception handling that led to the introduction of a *Long Method*. In this class, the developer added multiple catch blocks for different exceptions. However, the handling code was the same in every case, which led to duplicated code, making the method unnecessarily long. In that scenario, the developer should use the same code on the same block, since the Java language allows the developer to do that. These changes also can impact the robustness of the software system in the future, making it difficult for the developer to understand the source of that problem.

Finding 3: Robustness changes can introduce smells such as *Feature Envy* and *Dispersed Coupling* on the methods, which can negatively impact their maintainability.

We observed that these cases in which the robustness changes come along with the introduction of maintainability smells happened when the method was introduced in the commit. Therefore, both the change and the smell are inserted simultaneously. Still, the robustness change can indicate the presence of these smells in the normal code. For instance, let us consider the method `onBeforeExecute` from *apm-agen-java* [56]. This method is part of the *ElasticsearchRestClientInstrumentation* class, which provides the instrumentation for the Elasticsearch Java Rest Client. The method is called before a request is executed using this client. In this case, the catch block in the method was empty, and the developer left a comment warning that nothing should be handled there. Through this message, it can be the case that the exception is not in the correct class. This can be directly related to the presence of the *Feature Envy*, indicated through the multiple calls to data from the `Span` class. This excessive calling also introduced the smells *Intensive Coupling*, *Long Method*, and *Message Chain*. Therefore, even though it was a simple change in the catch block, it signaled the other maintainability smells.

Finding 4: Even small changes in the exceptional code (*e.g.*, a comment in an empty catch block) can be an indicator of maintainability smells such as *Feature Envy* and *Intensive Coupling*.

In summary, in our first RQ, we observed the maintainability smells that commonly co-occur or are introduced with robustness changes performed in the code. Hence, developers should be aware of these smells when performing this kind of change on a method, even when small. This way, it is recommended that the developers plan ahead the changes that will be performed. Furthermore, these changes should be considered even in the early stages of software development, when the decisions regarding exception handling should be defined appropriately. Developers could also benefit from using tools that could warn them about the possible maintainability smells that could be introduced with that robustness change. Alternatively, even warning them about possible smells already present in that method, giving the change performed, is informative.

B. What impact can robustness changes have on the degradation of classes?

To answer **RQ$_2$**, we computed the density of maintainability smells in methods with and without robustness changes (see Section IV-D. Figure 3 presents box plots representing the density of smells per method group and software system. Inside each one, we have white dots representing the mean density. In parentheses, we show the systems in which $p - value < 0.05$, meaning that the results were statistically significant for the Mann-Whitney statistical test applied. With this test, we want to confirm whether the density of maintainability smells in classes with methods that underwent robustness changes is greater than the density of smells in classes without these methods.

Density of maintainability smells in classes with and without methods that underwent robustness changes. Looking at the box plots, we can observe that the density median was higher in six systems, while the median was the same in the other four. In addition, the mean density of smells is higher in seven systems. Thus, we can observe through the mean and median that classes with methods that underwent robustness changes tend to have a higher density of maintainability smells than classes without these methods. More details about the statistical values are available in our replication package [53].

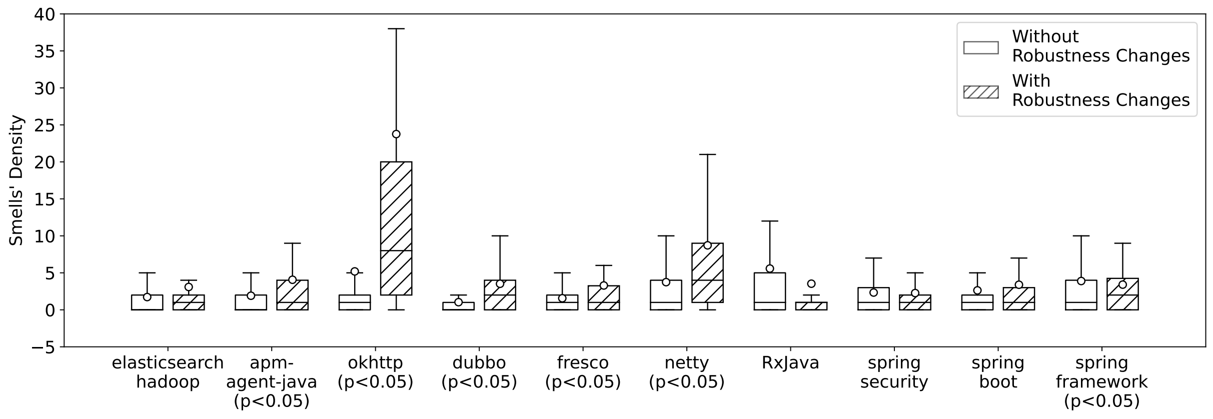

Fig. 3. Smells Density in Classes per Software System.

Finding 5: Classes with methods that underwent robustness changes tend to have a higher density of maintainability smells compared to classes without methods with this kind of change.

Robustness change can worsen the already existing maintainability smells. In our analysis, we observed a typical scenario for all systems. The introduced catch block does not originate the smells but, in some cases, contributes to their worsening (*e.g.*, making a method more coupled than earlier). It suggests that the robustness changes occur in methods with a high density of smells. For instance, we identified that catch blocks contribute to amplifying the smelly nature of *Long Methods*, *Message Chains*, and *Dispersed Couplings*. Therefore, the catch block contributes to expanding the smelly structure since the same practices of the smelling code are – to a minor or a large extent – also replicated for the exception handling code.

Well-written robustness changes may reduce maintainability smells. We observed that methods with a high number of smells had catch blocks with poor or no exception handling at all. For instance, on *elasticsearch*, we observed that the excess of maintainability smells on the methods could be related to the lack of proper error handling. Multiple instances of Feature Envy were found in methods without exception handling. These methods exhibited this code smell more frequently than the methods with catch blocks in the same class. We also observed similar cases on *Fresco* and *dubbo*. Since we are considering the system's version on the last release (*e.g.*, the last version of the class), we can conjecture that the methods in which the developers did not handle errors degraded more when compared with the classes that had proper error handling.

Finding 6: Robustness changes can worsen maintainability smells such as *Long Methods*, *Message Chains*, *Dispersed Couplings*, and *Feature Envy*, hampering the modularity of the system.

To sum up, with this RQ, we observed that the robustness changes could have a negative impact on classes with methods that underwent such changes. This impact is signaled through the high density of maintainability smells. Developers commonly use this density to prioritize classes likely to have design problems [21], [32]. Furthermore, the introduction and worsening of smells were mainly related to the maintainability smells that signal design problems related to the system's modularity (Table I). Hence, we highlight that when performing robustness changes, developers should be aware of the smells introduced or worsened since it may indicate deeper problems in the system. In addition, tools could use these metrics (*i.e.*, presence of robustness change and high density of specific smells) to signal possible design problems in the system.

C. How do robustness smells give evidence of design problems?

To answer **RQ**$_3$, we analyzed if robustness smells co-occurring with patterns of maintainability smells could help in the identification of design problems. Thus, we first identified when the robustness smells appeared in methods together with the maintainability smells that indicate design problems (see Table I). Table V presents the results. Each cell represents cases where the robustness smell co-occurs with the maintainability smells, forming a pattern. We can observe a high number of cases where *catch generic exception* (1,160 cases) and *empty catch block* (347 cases) co-occurred with the maintainability smells forming the patterns. We manually analyzed the patterns that happened the most with the robustness smells. They are highlighted in gray.

Generic and empty catches can indicate the presence of maintainability smell patterns. The robustness smell was not directly related to a design problem from the cases we analyzed. However, they could signal that maintainability smells were affecting the method. Let us consider the case of the method `getAsync` from fresco [57]. This method has nested try/catch blocks due to excess of verification performed. There are also catch generic exceptions that only return `Null` instead of handling the exceptions. We can hypothesize that this happened for two reasons: (i) the method was fulfilled with multiple concerns; hence the developer did not know which

TABLE V
CASES IN WHICH ROBUSTNESS SMELLS CO-OCCUR WITH PATTERN OF MAINTAINABILITY SMELLS

Pattern	Robustness Smell								
	Catch Generic Exception	Method Throws Exception	Empty Catch Block	Catch NPE	Rethrows Exception	Throw New Instance Of Same Exception	Throw Exception In Finally	Exception As Flow Control	Throw NPE
Concern Overload	212	1	54	2	7	0	3	7	3
Cyclic Dependency	74	5	6	0	6	0	3	7	1
Fat Interface	171	2	6	0	1	0	0	1	1
Misplaced Concern	394	5	199	3	8	1	0	1	7
Scattered Concern	35	0	1	0	0	0	0	1	0
Unwanted Dependency	274	2	81	2	7	0	0	1	4
Total	1160	15	347	7	29	1	6	18	16

exception should be handled, or (ii) the exception was handled on the wrong class. This can hint that this method could not be in the correct class. When we look at the maintainability smells, this method has a *Feature Envy* since it calls multiple methods from other classes, which also causes a *Dispersed Coupling*. Since the method uses data from other classes, it also causes a *Long Method*. Together, these smells can indicate the design problems *Misplaced Concern* or *Scattered Concern*. Both design problems suggest that this method should be moved to a more appropriate class.

Developers can be induced to introduce generic catches. Let us consider a case analyzed on *dubbo* [58] in the method `getProxy`, which was affected by a *catch generic exception* (*i.e.*, a *Throwable*). This is a bad practice since the exceptions should be adequately handled depending on their type. As the scope of *Throwable* is too broad, it may hide runtime issues that should be better handled. However, the developer may be induced to introduce these bad practices. This may occur on methods affected by a *Long Method*, combined with the fact that the developer does not know the software system's design well. Thus, he/she uses this generic handler to catch any exception.

Furthermore, this harmful practice can be the source of new maintainability smells, as we observed in this method, which was affected by *Dispersed Coupling*, *Feature Envy*, *Message Chain* and was part of a *Complex Class*. This happens because the developer tries to implement multiple concerns in this method. To do that, he/she may need to call multiple methods from other classes, hampering the modularity of the software system. In addition, the long methods also tend to have more dependencies on external elements. Thus, the class also has a higher chance of having exceptions handling errors from multiple contexts, which can explain why developers use generic catches. In this example, both maintainability and robustness smells could indicate the presence of a *Misplaced Concern* or *Scattered Concern*.

Generic catches indicating unwanted dependencies. We observed that developers tended to determine that all exceptions without a specific type would only be logged and lead to the system crash afterward. This happened when they were handling generic exceptions or in cases with an empty catch block, where only a comment was left in the handler. These smells could appear due to some projects' status as frameworks (*e.g.*, spring-framework and RxJava), meaning that the hot spot classes (*i.e.*, those that the framework's users will directly use) might have a different form of exception handling than the rest of the project. The design problem *Unwanted Dependency* seemed to appear for the same reason: the role of the classes as hot spots. Upon analysis, we noticed that these classes served as "import hubs". Each class imported several other package classes, allowing the user to have all the package's functionality by importing a single class. However, this bad practice leads to potentially unwanted dependencies among the classes. In addition, this caused the presence of the maintainability smells *Feature Envy*, *Shotgun Surgery*, and *Long Method*.

Finding 7: The robustness smells *empty catch block*, and *catch generic exception* can indicate the presence of maintainability smells mainly related to the design problems *Concern Overload*, *Misplaced Concern*, and *Unwanted Dependency*.

In conclusion, when the code is too complex and affected by code smells on both normal and exceptional code, we observed that developers tend to modify only the normal code (the one inside the try block). This can happen since the exceptional code tends to be more complex due to the global nature of exceptions. This complexity is related to how the exceptions can traffic between system modules. Thus, when changing the exceptional code, the developer needs to be careful about how this change will impact the modularity of the software system. In addition to the complexity, the IDEs tend to provide refactoring suggestions only for the normal code. Due to its different nature, it would be ideal to have specific refactorings for the exceptional code. For instance, in some cases, the IDEs constrain the implementation of exception-handling mechanisms. The IDE does not let the developer choose which class should handle the exception. Therefore, the developer must be free to choose how to properly handle the exception before applying the refactoring. Hence, the refactoring needs to be manually performed, which can discourage the developer from implementing the improvement change.

We hypothesize that this behavior was also due to the need for proper refactoring recommendations by IDEs. These tools provide better refactoring options only for the normal code.

They do not consider the nuances the exception handling code can have, such as which class could or should handle the exceptions. Thus, these changes co-occurring with the code smells can indicate to developers that robustness refactoring should be done at that time. When neglected, these robustness changes can also increase the degradation of the method. This can complicate the code in the future, making it more complex and discouraging developers from maintaining the code.

As observed, the robustness smells can indicate the presence of patterns of maintainability smells that signal design problems related to the system modularity. These design problems are challenging to identify since they can be scattered through multiple software system components. By analyzing maintainability smells, developers may not be able to identify a design problem accurately. However, developers can find more reliable results when using both maintainability and robustness smells. Thus, the indicator given by the co-occurrence of robustness smells and these patterns can help the developer to identify this design problem and refactor the code to remove or reduce this problem. Hence, the developer will save time and effort on this identification.

VI. Threats to Validity

Our analyses were performed on a set of 10 software systems, which could be a threat. However, our selection was based on meticulously defined criteria to find software systems (see Section IV-B) relevant to our research questions. In addition, our criteria also focused on reproducibility, allowing other researchers to replicate our steps. Therefore, our study can be the starting point for other researchers to explore the types of relationships explored in this paper with other software systems with other purposes and domains.

Another threat is the detection strategies (and their specific thresholds) for code smells. To mitigate this threat, we selected a tool (Organic) that uses a set of strategies and thresholds commonly used by the software engineering community [28]. Moreover, this tool has been successfully used in multiple studies about software design (*e.g.*, [20], [21], [23], [59]). Finally, we mitigated this threat by performing a manual inspection analysis with experienced experts. For each analysis, the participants filled out a form detailing how the robustness changes could be related to the maintainability smells.

The selection of releases can also be another threat. However, we carefully selected these releases based on the number of commits that the two releases had between them. More specifically, we selected a pair of releases spanning at least 1000 commits. That allowed us to cover a significant part of the history of software systems. Furthermore, there is no bias in the selection of releases, since this metric of commits was used indistinctly from the software system.

The use of exception handlers affects the size and complexity of methods, potentially threatening validity. Therefore, we validated our dataset to ensure we were not performing unfair comparisons between methods with and without exception handlers. Hence, we observed that the presence of getters/setters was balanced between these two subsets per software system, accounting for $\sim$11% of getters and setters composing our complete dataset.

VII. Conclusion

We explore how the combination of poor robustness changes and maintainability smells indicates design problems. For that purpose, we analyzed over 160k methods from 10 open-source systems. We identified that the robustness changes could introduce or worsen maintainability smells such as *Dispersed Coupling*, *Feature Envy*, and *Long Method*. We also observed that classes with methods that underwent robustness changes tend to have a higher density of maintainability smells compared to classes without these methods. Finally, we observed that the robustness smells *empty catch block* and *catch generic exception* could be used with the maintainability smells to help the identification of design problems such as *Concern Overload*, and *Unwanted Dependency*.

Understanding the relationship between robustness changes and code smells can help developers identify design problems' presence. Such problems are harmful to the software and hard to identify. Using the knowledge about this relationship can be the first step toward their identification. Additionally, these problems can be the target of refactoring operations, thus improving the maintainability and system robustness. In practice, developers can now be aware that even small changes in the catch blocks can impact or indicate possible design problems in the system. Moreover, based on our findings, tools can be developed to assist the maintainability and evolution of systems.

In future work, we plan to explore other symptoms of design problems (*e.g.*, smells related to other NFRs). In addition, we will investigate the evolution of methods that underwent robustness changes. Thus, we will be able to understand how poor robustness changes can impact the quality of the method during the software evolution. Finally, we plan to explore cases where only the exception part has changed and observe its impact on the normal code unchanged.

Acknowledgment

This work is partially funded by CNPq (140770/2021-6, 140185/2020-8, 434969/2018-4, 312149/2016-6, 409536/2017-2, and 427787/2018-1), CAPES/Procad (175956), CAPES/Proex (88887.373933/2019-00), FAPERJ (22520-7/2016), FAPERJ/PDR-10 program (202073/2020), FUNCAP (BP5-00197-00042.01.00/22), and by the Instituto Estadual de Engenharia e Arquitetura (IEEA), Secretaria Estadual de Infraestrutura do Estado do Rio de Janeiro (001/2021).

References

[1] H. Shah, C. Gorg, and M. J. Harrold, "Understanding exception handling: Viewpoints of novices and experts," *IEEE Transactions on Software Engineering*, vol. 36, no. 2, pp. 150–161, 2010.

[2] W. Weimer and G. C. Necula, "Exceptional situations and program reliability," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 30, no. 2, pp. 1–51, 2008.

[3] I. S. C. Committee *et al.*, "Ieee standard glossary of software engineering terminology (ieee std 610.12-1990). los alamitos," *CA: IEEE Computer Society*, vol. 169, 1990.

[4] F. Ebert and F. Castor, "A study on developers' perceptions about exception handling bugs," in *2013 IEEE International Conference on Software Maintenance*. IEEE, 2013, pp. 448–451.

[5] R. de Lemos and A. Romanovsky, "Exception handling in the software lifecycle," *International Journal of Computer Systems Science and Engineering*, vol. 16, no. 2, pp. 167–181, 2001.

[6] M. P. Robillard and G. C. Murphy, "Designing robust java programs with exceptions," in *Proceedings of the 8th ACM SIGSOFT international symposium on Foundations of software engineering: twenty-first century applications*, 2000, pp. 2–10.

[7] D. Reimer and H. Srinivasan, "Analyzing exception usage in large java applications," in *Proceedings of ECOOP'2003 Workshop on Exception Handling in Object-Oriented Systems*, 2003, pp. 10–18.

[8] B. Jakobus, E. A. Barbosa, A. Garcia, and C. J. P. De Lucena, "Contrasting exception handling code across languages: An experience report involving 50 open source projects," in *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2015, pp. 183–193.

[9] M. Kechagia and D. Spinellis, "Undocumented and unchecked: exceptions that spell trouble," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, 2014, pp. 312–315.

[10] R. Coelho, L. Almeida, G. Gousios, and A. Van Deursen, "Unveiling exception handling bug hazards in android based on github and google code issues," in *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. IEEE, 2015, pp. 134–145.

[11] J. Oliveira, N. Cacho, D. Borges, T. Silva, and F. Castor, "An exploratory study of exception handling behavior in evolving android and java applications," in *Proceedings of the 30th Brazilian Symposium on Software Engineering*, 2016, pp. 23–32.

[12] J. Garcia, D. Popescu, G. Edwards, and N. Medvidovic, "Toward a catalogue of architectural bad smells," in *International conference on the quality of software architectures*. Springer, 2009, pp. 146–162.

[13] Z. Li, P. Avgeriou, and P. Liang, "A systematic mapping study on technical debt and its management," *Journal of Systems and Software*, vol. 101, pp. 193–220, 2015.

[14] 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, 2012.

[15] 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.

[16] J. Van Gurp and J. Bosch, "Design erosion: problems and causes," *Journal of systems and software*, vol. 61, no. 2, pp. 105–119, 2002.

[17] 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.

[18] B. Curtis, J. Sappidi, and A. Szynkarski, "Estimating the size, cost, and types of technical debt," in *2012 Third International Workshop on Managing Technical Debt (MTD)*. IEEE, 2012, pp. 49–53.

[19] R. de Mello, R. Oliveira, A. Uchôa, W. Oizumi, A. Garcia, B. Fonseca, and F. de Mello, "Recommendations for developers identifying code smells," *IEEE Software*, vol. 40, no. 2, pp. 90–98, 2023.

[20] W. Oizumi, A. Garcia, L. D. S. Sousa, B. Cafeo, and Y. Zhao, "Code anomalies flock together: Exploring code anomaly agglomerations for locating design problems," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 2016, pp. 440–451.

[21] L. Sousa, A. Oliveira, W. Oizumi, S. Barbosa, A. Garcia, J. Lee, M. Kalinowski, R. de Mello, B. Fonseca, R. Oliveira *et al.*, "Identifying design problems in the source code: A grounded theory," in *Proceedings of the 40th International Conference on Software Engineering*, 2018, pp. 921–931.

[22] D. Coutinho, A. Uchôa, C. Barbosa, V. Soares, A. Garcia, M. Schots, J. Pereira, and W. K. Assunçao, "On the influential interactive factors on degrees of design decay: A multi-project study," in *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2022, pp. 753–764.

[23] A. Oliveira, W. Oizumi, L. Sousa, W. K. Assunção, A. Garcia, C. Lucena, and D. Cedrim, "Smell patterns as indicators of design degradation: Do developers agree?" in *Proceedings of the XXXVI Brazilian Symposium on Software Engineering*, 2022, pp. 311–320.

[24] Java Documentation, "What is an exception?" 2021, https://docs.oracle.com/javase/tutorial/essential/exceptions/definition.html, Last accessed on 2021-07-16.

[25] J. B. Goodenough, "Exception handling: Issues and a proposed notation," *Communications of the ACM*, vol. 18, no. 12, pp. 683–696, 1975.

[26] F. Ebert, F. Castor, and A. Serebrenik, "An exploratory study on exception handling bugs in java programs," *Journal of Systems and Software*, vol. 106, pp. 82–101, 2015.

[27] M. Fowler, *Refactoring: Improving the Design of Existing Code*, 1st ed. Addison-Wesley Professional, 1999.

[28] M. Lanza, R. Marinescu, and S. Ducasse, *Object-Oriented Metrics in Practice*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2005.

[29] A. Uchôa, C. Barbosa, W. Oizumi, P. Blenílio, R. Lima, A. Garcia, and C. Bezerra, "How does modern code review impact software design degradation? an in-depth empirical study," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2020, pp. 511–522.

[30] L. Sousa, R. Oliveira, A. Garcia, J. Lee, T. Conte, W. Oizumi, R. de Mello, A. Lopes, N. Valentim, E. Oliveira *et al.*, "How do software developers identify design problems? a qualitative analysis," in *Proceedings of the 31st Brazilian Symposium on Software Engineering*, 2017, pp. 54–63.

[31] W. Oizumi, A. Garcia, L. Da Silva Sousa, B. Cafeo, and Y. Zhao, "Code anomalies flock together: Exploring code anomaly agglomerations for locating design problems," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, 2016, pp. 440–451.

[32] A. Oliveira, L. Sousa, W. Oizumi, and A. Garcia, "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*, 2019, pp. 83–92.

[33] E. W. Dijkstra, *A Discipline of Programming*, 1st ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1997.

[34] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Commun. ACM*, vol. 15, no. 12, pp. 1053–1058, Dec. 1972.

[35] L. Sousa, W. Oizumi, A. Garcia, A. Oliveira, D. Cedrim, and C. Lucena, "When are smells indicators of architectural refactoring opportunities: A study of 50 software projects," in *Proceedings of the 28th International Conference on Program Comprehension*, 2020, pp. 354–365.

[36] W. Oizumi, L. Sousa, A. Oliveira, L. Carvalho, A. Garcia, T. Colanzi, and R. Oliveira, "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)*. IEEE, 2019, pp. 346–357.

[37] 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.

[38] 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.

[39] J. Garcia, I. Ivkovic, and N. Medvidovic, "A comparative analysis of software architecture recovery techniques," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2013, pp. 486–496.

[40] H. Melo, R. Coelho, and C. Treude, "Unveiling exception handling guidelines adopted by java developers," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2019, pp. 128–139.

[41] J. Rocha, H. Melo, R. Coelho, and B. Sena, "Towards a catalogue of java exception handling bad smells and refactorings," in *Proceedings of the 25th Conference on Pattern Languages of Programs*, 2018, pp. 1–17.

[42] G. B. De Padua and W. Shang, "Studying the prevalence of exception handling anti-patterns," in *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*. IEEE, 2017, pp. 328–331.

[43] M. B. Kery, C. Le Goues, and B. A. Myers, "Examining programmer practices for locally handling exceptions," in *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*. IEEE, 2016, pp. 484–487.

[44] S. Nakshatri, M. Hegde, and S. Thandra, "Analysis of exception handling patterns in java projects: An empirical study," in *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*. IEEE, 2016, pp. 500–503.

[45] M. Asaduzzaman, M. Ahasanuzzaman, C. K. Roy, and K. A. Schneider, "How developers use exception handling in java?" in *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*. IEEE, 2016, pp. 516–519.

[46] N. Cacho, T. César, T. Filipe, E. Soares, A. Cassio, R. Souza, I. Garcia, E. A. Barbosa, and A. Garcia, "Trading robustness for maintainability: an empirical study of evolving c# programs," in *Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 584–595.

[47] R. Fisher, "On the interpretation of χ 2 from contingency tables, and the calculation of p," *J. R. Stat. Soc.*, vol. 85, no. 1, pp. 87–94, 1922.

[48] C. Barbosa, A. Uchôa, D. Coutinho, F. Falcão, H. Brito, G. Amaral, V. Soares, A. Garcia, B. Fonseca, M. Ribeiro *et al.*, "Revealing the social aspects of design decay: A retrospective study of pull requests," in *Proceedings of the 34th Brazilian Symposium on Software Engineering*, 2020, pp. 364–373.

[49] E. A. Barbosa, A. Garcia, and S. D. J. Barbosa, "Categorizing faults in exception handling: A study of open source projects," in *2014 Brazilian Symposium on Software Engineering*. IEEE, 2014, pp. 11–20.

[50] "Pmd," https://pmd.github.io/, (Accessed on 01/19/2023).

[51] "Sonarqube," https://www.sonarsource.com/products/sonarqube/, (Accessed on 01/19/2023).

[52] 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.

[53] Anonymous, "Complementary Material," https://github.com/robustnessdp/robustness-changes, 2023.

[54] D. J. Sheskin, *Handbook of parametric and nonparametric statistical procedures*. Chapman and Hall/CRC, 2003.

[55] "Rxjava/flowableflatmap.java at 2572fa74ab93c4f3ffe0ea51932eecc180873904 · reactivex/rxjava · github," https://github.com/ReactiveX/RxJava/blob/2572fa74ab93c4f3ffe0ea51932eecc180873904/src/main/java/io/reactivex/internal/operators/flowable/FlowableFlatMap.java, (Accessed on 01/11/2023).

[56] "Elasticsearchrestclientinstrumentation.java at apm-agent-java," https://github.com/elastic/apm-agent-java/blob/32a364/apm-agent-plugins/apm-es-restclient-plugin/apm-es-restclient-plugin-5_6/src/main/java/co/elastic/apm/es/restclient/v5_6/ElasticsearchRestClientInstrumentation.java, (Accessed on 01/11/2023).

[57] "Buffereddiskcache.java at facebook/fresco," https://github.com/facebook/fresco/blob/cc75c3/imagepipeline/src/main/java/com/facebook/imagepipeline/cache/BufferedDiskCache.java, (Accessed on 01/16/2023).

[58] "dubbo/stubproxyfactorywrapper.java apache/dubbo," https://github.com/apache/dubbo/blob/0e66de1c6/dubbo-rpc/dubbo-rpc-api/src/main/java/org/apache/dubbo/rpc/proxy/wrapper/StubProxyFactoryWrapper.java, (Accessed on 01/16/2023).

[59] 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*, 2017, pp. 465–475.