How do Code Smell Co-occurrences Removal Impact Internal Quality Attributes? A Developers' Perspective

Júlio Martins
Federal University of Ceara
Quixadá, CE, Brazil
juliomserafim@gmail.com

Carla Bezerra
Federal University of Ceara
Quixadá, CE, Brazil
carlailane@ufc.br

Anderson Uchôa
DI - PUC-Rio
Rio de Janeiro, RJ, Brazil
auchoa@inf.puc-rio.br

Alessandro Garcia
DI - PUC-Rio
Rio de Janeiro, RJ, Brazil
afgarcia@inf.puc-rio.br

ABSTRACT

Code smells are poor code structures that might harm the software quality and evolution. However, previous studies has shown that only individual occurrences of smells may not be enough to assess the real impact that these smells can bring on systems. In this context, the co-occurrences of code smells, i.e., occurrences of more than one code smell in the same class or same method, can be better indicators of design problems for software quality. Despite its importance as an indicator of design problems, we have little known about the impact of removing the co-occurrence of smells via software refactoring on internal quality attributes, such as coupling, cohesion, complexity, and inheritance. It is even less clear on what is the developers' perspective on the co-occurrences removal. We aim at addressing this gap through a qualitative study with 14 developers. To this end, we analyze the refactorings employed by developers during the removal of 60 code smells co-occurrences, during 3 months in 5 closed-source projects. We observe (i) impact of code smells co-occurrences on internal quality attributes, (ii) which are the most harmful co-occurrences from the developers' perspective, (iii) developers' perceptions during the removal of code smells co-occurrence via refactoring activities; and (iv) what are the main difficulties faced by developers during the removal of code smells co-occurrences in practice. Our findings indicate that: (i) the refactoring of some types of code smells co-occurrences (e.g., Dispersed Coupling–God Class) indicated improvement for the quality attributes; (ii) refactoring code smells co-occurrences according to the developers is difficult mainly due to the understanding of the code and complexity refactoring methods; and (iii) developers still have insecurities regarding the identification and refactoring of code smells and their co-occurrences.

CCS CONCEPTS

• **Software and its engineering → Software evolution**; **Maintaining software**.

KEYWORDS

Code Smells Co-occurrences. Refactoring. Internal Quality Attributes.

ACM Reference Format:
Júlio Martins, Carla Bezerra, Anderson Uchôa, and Alessandro Garcia. 2021. How do Code Smell Co-occurrences Removal Impact Internal Quality Attributes? A Developers' Perspective. In *Brazilian Symposium on Software Engineering (SBES '21), September 27-October 1, 2021, Joinville, Brazil.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3474624.3474642

SBES '21, September 27-October 1, 2021, Joinville, Brazil

1 INTRODUCTION

Throughout its evolution, the software systematically undergoes changes that can lead to the deterioration of its quality structure [16, 42, 44, 45]. In this context, the concept of code smells arises, which are anomalous code structures that represent symptoms that affect the maintainability of systems at different levels, such as classes and methods [14, 20]. However, these anomalies are considered weaknesses in the software design that can delay development or increase the threat of failure or errors in the future [14]. In some cases, code smells are introduced into the source code through poor implementation choices caused several times by the developers' rush to deliver the functionality of a system [1].

Some works have evaluated the individual occurrences of code smells and the relationships between these smells and their impact on the software quality [13, 24, 28, 34, 51]. Despite the large number of studies that investigate the effects of individual occurrences of code smells [35], few studies investigate the effects of code smells co-occurrences [24]. This is an important topic, because the removal of co-occurrence can have a greater impact on code quality than removing individual occurrences of code smells [13, 24, 50].

Another gap pointed out by literature reviews [16, 17] is that most studies that investigate the effects of code smells and their co-occurrences use open-source systems and that few studies consider industrial systems [2]. Besides, the reviews point out that more empirical studies are needed in the area of code smells refactoring that take into account the developers' perception about the removal of these anomalies in the code [9, 17, 32].

This paper addresses the aforementioned limitations through a qualitative and slightly controlled study that aims to investigate the impact of code smells co-occurrences removal on four internal quality attributes – cohesion, coupling, complexity, and inheritance. To this end, we analyzed data from five closed-source software systems. We identify the code smells co-occurrences that are most harmful to the internal quality attributes under the developers' perspective. Moreover, we analyze the developers' perception on the removal of code smells co-occurrences via refactoring activities. Such perception was capture using the diary technique [15], in which the developers documenting their perception during the removal of each code smell co-occurrence. Finally, we analyze the main difficulties faced by developers during the co-occurrences removal. We summarize our study findings as follows.

(1) The removal of *Dispersed Coupling–God Class* and *God Class–Long Method* co-occurrences improved all internal quality attributes;

(2) Developers should remove and avoid introducing the following co-occurrences into the code: *Dispersed Coupling–God Class*, *Feature Envy–God Class* and *God Class–Long Method*;
(3) One of the main difficulties for developers is to understand the code that contains the code smells co-occurrences; and
(4) Developers still have insecurities in the identification and removal of code smells co-occurrences.

2 BACKGROUND

2.1 Code Smells

Code smells are poor code structural that might indicate design problems that harm the software maintenance and evolution [30]. The code smells may indicate design problems at multiple levels of granularities, i.e., they can indicate design problems at method and class levels. *Long method* is an example of a code smell that may indicate a design problem at the method level since that smell represents long and complex methods. On the other hand, *God Class* is a code smell that can indicate problems at the class level, by representing classes that are complex and difficult to modify [14].

Software developers often rely on code smells as indicators of code quality [14, 37, 42, 45]. For instance, developers have often used tools like Stack Overflow to ask about code smells, anti-patterns and even to identify these anomalies in their own source code [39]. There are several tools for detecting code smells [12]. In this study, we rely on the JSpIRIT [46] and JDeodorant [41] tools to detect different types of code smells at the method and class level. Such tools have been used by several studies in the literature [24, 25, 43]. Table 1 presents the code smells considered in this study. The first column names the code smells. The remainder columns present: a short description, and the tools used to detect them.

Table 1: Code smells detected in this study

Code Smells	Description	Tool
Feature Envy	Method "envying" other classes' features [14]	JSpIRIT
God Class	Too many software features into a class. It tend to be very large and hard to read and understand [14]	JDeodorant
Dispersed Coupling	Method that calls too many methods [14]	JSpIRIT
Intensive Coupling	Method that depends too much from a few others [14]	JSpIRIT
Shotgun Surgery	Method whose changes affect many methods [14]	JSpIRIT
Long Method	Too long and complex method [14]	JDeodorant

2.2 Code Smell Co-occurrences

Code smells co-occurrences occur when there are relationships and dependencies between two or more code smells [33, 51]. For instance, the same class that is *God Class* and also has a *Duplicated Code* [33]. Previous studies have used the code smells co-occurrences to provide a better understanding of the impact that code smells interactions can cause on software quality [20, 28, 51]. For instance, Yamashita et al. [51] observed that code smells interactions in the same file (*collocated smells*) and that interactions in coupled files (*coupled smells*), have shown problems with maintenance activities and software quality. On the other hand, Oizumi et al. [28] investigated how the relationships between *code smells* can support developers in locating problems in the software design. The authors propose a strategy for code smells co-occurrence groups called code smells agglomerations.

The code smells co-occurrences can be occur at class and method levels [31]. At the method level, a co-occurrence exists when there are two or more method code smells in a given method. For instance, when in the same method there are occurrences of *code smells* such as *Feature Envy* and *Long Method*. Similarly, a class-level co-occurrence exists when there are two or more code smells in a given class. For instance, the occurrence of a *God Class* together with another code smells in the same class.

Table 2 shows examples of code smells co-occurrences at the class and method levels. In the first example, there is a code smell co-occurrence (*Long Method and God Class*), i.e., Class1 which is a God Class (CG) has Method1 which is Long Method (LM). In the second example, there is a co-occurrence at the method level in which the two code smells Long Method and Feature Envy (FE) are "together" in Method2. This example represents how we identified the code smells co-occurrences in our study [33].

Table 2: Examples of code smells co-occurrences

Class	Method	LM	FE	GC
Class1	method1()	X		X
Class2	method2()	X	X	

2.3 Internal Quality Attributes

Internal quality attributes are key indicators of code structural quality [11]. Examples of internal quality attributes include coupling, cohesion, complexity, and inheritance. *Coupling* is the degree of interdependence between modules or classes. High coupling affects maintainability and reusability, for instance. *Cohesion* is the degree to which the internal elements of a module are related to each other. *Low cohesion* may lead to high complexity and bug proneness. *Complexity* is the measure of the overload of responsibilities and decisions of a module. It affects the code readability, for instance. *Inheritance* represents relationships between superclasses and subclasses. Finally, *Inheritance* enables software reusability, but large hierarchies may lead to software maintenance problems [5].

Previous studies [3, 11, 23] apply software metrics to capture specific internal quality attributes. For instance, Mens and Tourwé [27] apply different software metrics for measuring internal quality attributes. In our study, we used 13 software metrics of internal quality attributes (see Table 4) well known in the literature [6, 10, 21, 26]. To collect the metrics we use the Understand tool to measure all internal quality attributes [5].

3 STUDY SETTINGS

3.1 Goal and Research Questions

We followed the Goal Question Metric template [48] to define our goal as follows: *analyze* the removal of code smells co-occurrences; *for the purpose of* understating their impact on internal attributes of software quality; *with respect to* (i) the most harmful code smell co-occurrences; (ii) the most harmful from the developer's perspective; and (iii) difficulties and perceptions of developers during the removal of code smell co-occurrences; *from the viewpoint of* software developers; in the context of five closed-source systems. Our research questions (RQ_s) are discussed as follows.

RQ_1: *Which code smells co-occurrences are most harmful to the internal quality attributes?* – **RQ_1** aims at understanding which are the most harmful code smells co-occurrences for internal quality

attributes. Answer this research question, it is important because it allows us to know and understand which are the most harmful co-occurrences for software quality. Thus, by answering $\mathbf{RQ}_1$, we can provide guidance on which co-occurrences should be avoided.

$\mathbf{RQ}_2$: *Which code smells co-occurrences are considered most harmful from the developer's perspective?* – $\mathbf{RQ}_2$ aims at assessing which code smells co-occurrences are harmful from the developer's perspective. This research question is important, since previous studies [23–25, 49] have considered the impact of co-occurrences on maintainability, without considerate the developer's perspective. Thus, by answering $\mathbf{RQ}_2$, we can understand developers' perceptions about code smells co-occurrences.

RQ3: *What are the main difficulties faced by developers during the removal of code smells co-occurrences in practice?* – $\mathbf{RQ}_3$ aims at identifying the main difficulties faced by developers during the removal of code smells co-occurrences via software refactorings. By answering $\mathbf{RQ}_3$, we can understand what are the main difficulties and what criteria the developers take into account to explain the difficulty of removing code smells occurrences.

3.2 Study Steps

This section describes the study steps, in order to support the investigation of code smells co-occurrences.

Step 1: Selecting closed-source systems for analysis. We have selected five closed-source systems that are being developed by our industrial partners. To select the systems, we used the following criteria: (i) the system must be mostly written in the Java programming language, due to the availability of software metric and refactoring mining tools; (ii) systems that are already in a production environment; and (iii) systems with the most lines of code. Table 3 presents general data per system. The first column names the system[1]. The remainder columns present: system domain; number of classes; and, number of lines of code (LOC). We collected all data via Understand tool.

Table 3: General data of the target software systems

System	Domain	# of classes	# of methods	# LOC
S1	Management of Events	78	612	4790
S2	Extension and Research Projects	110	706	6464
S3	Risk Management	106	698	4296
S4	Employee Competency	183	1247	8384
S5	Student Activities	48	250	1910

The S1 aims to allow the management of public and private events. S2 aims to store and manage extension actions and research projects developed by university employees. S3 aims to enable risk management at a university. S4 aims to enable employee competency management. Finally, S5 aims to facilitate the management of complementary student activities. All the target systems are web-based and developed using Spring framework, Thymeleaf, Vue.js, and Jquery technologies.

Step 2: Detecting code smells co-occurrences. Before detecting the code smells co-occurrences, we identified the individual occurrences of six types of code smells: *Feature Envy*, *God Class*, *Dispersed Coupling*, *Intensive Coupling*, *Shotgun Surgery* and *Long Method*. The code smells were collected using the JDeodorant [18] and JSpIRIT [46] tools. Next, we detected the code smells co-occurrences in each target system. The types of relationships used to identify the co-occurrences are described in Section 2.2.

Step 3: Measuring internal quality attributes. Table 4 presents the 13 software metrics that we used to measure internal quality attributes [6, 10, 21]. In total we measuremed four quality internal attributes: cohesion, coupling, complexity, and inheritance. The first column lists the internal quality attributes. The second column presents the software metrics related to each internal quality attribute. Finally, the third column describes each metric.

Table 4: Metrics of the internal quality attributes analyzed in this work [6, 10, 21, 26]

Attributes	Metric	Description
Cohesion	Lack of Cohesion of Methods (LCOM2) [6]	Measures cohesion of a class. The higher the value of this metric, less cohesive is the class.
	Lack of Cohesion of Methods (LCOM3) [6]	Number of disjoint components in the graph that represents each method. The higher the value of this metric, less cohesive is the class.
Coupling	Coupling Between Objects (CBO) [6]	Number of classes that a class is coupled. The higher the value of this metric, more coupling is the classes and methods.
	Coupling Between Objects Modified (CBO Modified) [6]	Number of other classes coupled to. The higher the value of this metric, more coupling is the classes and methods.
	Fan-in (FANIN) [6]	Number of other classes that reference a class. The higher the value of this metric, more coupling is the classes and methods.
	Fan-out (FANOUT) [6]	Number of other classes referenced by a class. The higher the value of this metric, more coupling is the classes and methods.
Complexity	Weighted Method Count (WMC) [26]	Sum of cyclomatic complexity of all nested functions or methods. The higher the value of this metric , more complex is the classes and method.
	Sum Cyclomatic Complexity (SCC) [26]	Sum of cyclomatic complexity of all nested methods. The higher the value of this metric, more complex is the classes and methods.
	Nesting (MaxNest) [21]	Maximum nesting level of control constructs. The higher the value of this metric, more complex is the classes and methods.
	Essential Complexity (EVG) [26]	Measure of the degree to which a module contains unstructured constructs. The higher the value of this metric, more complex is the classes and methods.
Inheritance	Number Of Children (NOC) [6]	Number of subclasses of a class. The higher the value of this metric greater is the degree of inheritance of a system.
	Depth of Inheritance Tree (DIT) [6]	The number of levels that a subclass inherits from methods and attributes of a superclass in the inheritance tree. The higher the value of this metric greater is the degree of inheritance of a system.
	Bases Classes (IFANIN) [10]	Immediate number of base classes. The higher the value of this metric greater is the degree of inheritance of a system.

To compute each metric, we used a non-commercial license of the Understand tool. We selected theses metrics because they enable us to assess different properties of each attribute [4, 6], such as LOC and CBO that measures the size and coupling, respectively. Therefore, these code metrics can reveal the quality of the target system before and after the code smells co-occurrences removal in terms of internal quality attributes.

Step 4: Performing the removal of co-occurrences of code smells with software developers. This step aims to conduct the removal of co-occurrences of code smells identified in Step 2. For this purpose, we have recruited developers who contribute to the development of each selected software system to participate as subjects in the study. Thus, we sent a *Characterization Form* for each developer. This form aimed to characterize the developer regarding education, experience with software development, and their projects. Their answers were analyzed to determine which of them were eligible to participate in the study. Table 5 summarizes the characteristics of each developer selected for the experiment.

[1]We omitted their names due to intellectual-property constraints.

All the developers are from the same company, but not everyone was aware of all systems; 3 had prior knowledge of S1, 4 of S2, 4 of S3, 2 of S4, and 2 of S5. The company released the developers as a regular part of the job.

Table 5: Characterization of developers

ID	Experience in years	Education Level	Quality Metrics	Code Smells	Java
P1	3 years	Graduate Degree	Basic	Basic	Intermediary
P2	6 years	Graduate Degree	Basic	Basic	Intermediary
P3	4 years	Graduate Degree	Advanced	Advanced	Advanced
P4	2 years	Graduate Degree	Intermediary	Intermediary	Advanced
P5	4 years	Master Degree	Basic	Intermediary	Advanced
P6	4 years	Graduate Degree	Basic	Basic	Intermediary
P7	2 years	Graduate Degree	Intermediary	Intermediary	Intermediary
P8	2 years	Graduate Degree	Basic	Basic	Intermediary
P9	2 years	Graduate Degree	Intermediary	Intermediary	Advanced
P10	3 years	Graduate Degree	Basic	Basic	Intermediary
P11	3 years	Graduate Degree	Basic	Basic	Intermediary
P12	5 years	Graduate Degree	Intermediary	Intermediary	Advanced
P13	8 years	Master Degree	Advanced	Advanced	Advanced
P14	4 years	Graduate Degree	Basic	Basic	Intermediary

After selected the developers we asked them to perform the removal of code smells co-occurrences (both method and class levels) in their systems thought manual software refactoring. We explain in more detail the experimental procedure used to remove the code smells co-occurrences in Section 3.3.

Step 5: Documenting the developer's perspective during the removal of code smell co-occurrences. We rely on the diary technique for document the developers' perception concerning the removal of code smell co-occurrences during the refactoring applications. The diary technique consists of a data collection method in which participants record in a form, their daily activities about any event that has affected them positively or negatively. This technique is a way to understand the participant's behavior, minimizing the influence of researchers [15]. In this study, the developers used the diary technique during the removal of code smell co-occurrences to record the answers to the following questions: (1) *I am currently working on refactoring which co-occurrence?*; (2) *What are my main difficulties in removal this co-occurrence?*; (3) *What is the most harmful co-occurrence that I am removing?*; (4) *Why did I choose this co-occurrence as the most harmful?*; and (5) *What refactoring operations am I using to remove co-occurrences?*. Thus, by using the diary technique we can capture the perception of the developers at the time of removal of code smell co-occurrences.

Step 6: Analyzing the removal of code smell co-occurrences and the developers' perception. After the removal of code smell co-occurrences, we performed new measurements of the internal quality attributes. Our goal was to verify if the system quality improved or worsened after the removal of co-occurrences via refactoring. To this end, we used the same strategy of Tarwani and Chug [40], in which the sum of the metrics of each internal quality attribute is used. Therefore, if the value of the sum of the metrics of a given internal quality attribute increases, it means that this attribute has worsened. We measured and calculated the value of each metric in each class before and after the refactoring commits related to the complete removal of a given code smell co-occurrence.

For instance, we used two metrics to calculate the cohesion (see Table 2.3), we measured and calculated the sum of the value of each metric of this attribute in the class that contained the co-occurrence before the removal and after the removal of the co-occurrence. Next, we analyzed three possible scenarios: (1) If the value of the two metrics has decreased, then cohesion has increased/improved; (2) If the value of the two metrics has increased, then the cohesion has decreased/worsened; and (3) If there was no change in the value of the metrics, then the cohesion remained unaffected. In summary, we used this approach to all other metrics and internal quality attributes. More details on the detection of code smell co-occurrences and measuring systems before and after removing code smells co-occurrences are found in our research website[2].

Finally, we analyzed the responses of developers. To this end, we rely on Grounded Theory procedures [38]. More specifically, we used the *open* and *axial* coding procedures to analyze the types of occurrences considered most harmful, and the main difficulties faced by developers throughout the refactoring process.

3.3 Experimental Procedures

The study was composed by a set of four activities.

Activity 1: Training session. We conducted a training session with all participants about essential concepts for the study, such as code smells and their co-occurrences, internal quality attributes, and refactoring operations. We also trained the participants about how to identify the code smells co-occurrences. We spent four hours. We presented a set of practical examples that illustrate refactoring operations that could be applied in each co-occurrence presented in the first part of the training. Next, we provide a set of toy examples for developers to apply refactoring methods to remove code smells. We also explained to the developers how the diary technique worked so that they could use this technique during the refactorings of each code smell co-occurrence. We decided to provide a training session to level up their knowledge about the main concepts regarding our study. Thus, we tried to reduce the bias by focusing on main concepts and presenting theoretical and practical examples.

Activity 2: Removal of co-occurrences of code smells via manual refactoring. We asked developers to perform the removal of the complete code smells co-occurrences, i.e., the two code smells that characterize the co-occurrence, in both at the method and class level thought manual software refactoring. To support the removal of code smells co-occurrences, we provided participants a list that summarized the name of methods or classes in which the co-occurrences of code smells were identified from Step 2.

Additionally, for each code smell co-occurrence, we created issues on Github related to refactoring activities. Each issue contained information about the class and the method affected by a code smell. Thus, the developers were free to choose issues, and which code smell co-occurrences to refactor. We conducted weekly meetings to check the progress of the activities and if the developers founded any type of difficulty or obstacle in the refactoring process. Despite the freedom of developers to choose issues, we were concerned about the bias of the choice of developers, and in training, they were asked to choose different types of co-occurrences to remove. We instructed developers to make it clear which commits were related to a refactoring activity. Thus, each commit has tagged to with the label representing the name of the code smell co-occurrence to be

[2]https://julioserafim.github.io/SBES2021/

refactored. Additionally, separate branches were created for each of the refactoring activities.

Activity 3: Perception documentation along the removal of code smells co-occurrences. During the entire process of removal co-occurrences via refactoring, developers were instructed to document their perceptions using the diary technique [15]. Thus, each developer documented which co-occurrence he was working on at the time, which co-occurrence was considered to be the most harmful during removal, and the reason for this choice. In addition, the developers explained the main difficulties related to the removal of co-occurrence.

Activity 4: Validation of complete removal of code smells co-occurrences. We analyzed the commits to verify if the code smells co-occurrences were completely removed by the developers. Two researchers carried out the analysis and review of the results until reaching a consensus. We also analyzed: (1) the impact of code smells co-occurrence removal on the internal quality attributes (cohesion, coupling, inheritance, and complexity); (2) the number of days which developers spent on each issue removing co-occurrences; and (3) the number of commits it took for the developer to refactor the code smells co-occurrences. The analysis of the projects was performed project-by-project, starting with the S1 system and finish with the S5 system. The entire removal of code smells co-occurrences via refactoring for all target systems took three months and involved 14 developers. Our analysis took 2 months to completed. Table 6 shows the number of code smells co-occurrences, the number of refactoring commits and the number of total commits.

Table 6: Number of refactoring commits and number of co-occurrences removed

System	# co-occurrences	# refactoring commits	# total of commits
S1	16	92	1597
S2	30	132	1056
S3	12	70	1196
S4	20	106	2471
S5	4	20	111

4 RESULTS AND DISCUSSION

4.1 Co-occurrences of Code Smells that are more harmful to quality attributes (RQ_1)

We address $\mathbf{RQ_1}$ by analyzing the impact of code smell co-occurrences removal on four internal quality attributes: coupling, cohesion, complexity, and inheritance. The removal of co-occurrences was performed via refactorings employed by 14 developers. Table 7 shows the impact of code smell co-occurrences removal for the internal quality attributes considering all target systems. The first column represents the co-occurrence. The remainder columns present the impact of code smell co-occurrences removal for cohesion, complexity, coupling, and inheritance, respectively. The ↑ symbol indicates an increase in the value of the attribute after the co-occurrences removal, the ↓ symbol indicates a decrease in the value of the attribute after the co-occurrences removal, and the – symbol indicates that the value of the attribute remained unaffected.

We emphasize that if the cohesion increases (by decreasing the values of the metrics LCOM2 and LCOM3)), it means that this attribute has been improved because of the greater the cohesion of a class or method the better the system quality. Conversely, attributes such as complexity and coupling should be kept as small as possible to indicate an improved code quality. In fact, a high complexity may indicates a code more difficult to understand and a high coupling may indicate a code more difficult to modify. Thus, for these two attributes, their decreases mean an increase in quality. Finally, an increase in the inheritance attribute can mean greater reusability in the code and, consequently, a better quality. However, care must be taken as excessive inheritance can lead to heavy coupling and be detrimental to the software [14].

Co-occurrences that were removed and improved all internal quality attributes. Results of Table 7 reveal some interesting observations. The removal of *Dispersed Coupling–God Class* and *God Class–Long Method* improved all internal quality attributes. More specifically, the removal of *Dispersed Coupling–God Class* increased the cohesion by *3.16%*, decreased the complexity by **24.59%**, and increased the inheritance by **3.57%**. These observations indicate that the removal of these co-occurrences brings a significant improvement of system quality or that the presence of these co-occurrences indicates a degradation of the system quality. Thus, we believe that it is worthwhile for developers, quality analysts, managers, and other professionals to focus on removing these specific co-occurrences to improve the quality of the systems.

> **Finding 1**: The removal of *Dispersed Coupling-God Class* and *God Class-Long Method* has improved all internal quality attributes, this suggests that the presence of these co-occurrences are harmful to the system quality.

Co-occurrences that were removed and improved at least three internal quality attributes. Backing to Table 7, we can observed that the removal of some co-occurrences, such as *Feature Envy-God Class*, and *God Class–Shotgun Surgery* improved the cohesion, complexity e coupling attributes. With exception of inheritance attribute that has remained unaffected. More specifically, the removal of *Feature Envy-God Class* caused an increase of **10.51%** in cohesion, a decrease of **30.98 %** in complexity and a decrease of **21.52%** in coupling. On the other hand, the removal of *God Class-Shotgun Surgery* increased cohesion by **2.59%**, decreased by **27.61%** complexity and decreased by **19.14%** the coupling. Finally, the removal of *Dispersed Coupling-Long Method* resulted in decreased of complexity in **9.62%**, coupling in **15.48%**, and increased the inheritance in **5.56%** and worsened the cohesion attribute in **30.80%**.

> **Finding 2**: The removal of *Feature Envy-God Class* and *God Class-Shotgun Surgery* has improved three internal quality attributes. This also indicates that these co-occurrences need more attention from developers.

Co-occurrences that were removed and improved only one, and at least two internal quality attributes. The removal of *Intensive Coupling-Long Method* worsened the cohesion in **19.90%**, and complexity in **12.63%**, however, the coupling decreased **6.81%**. On the other hand, the removal of *Feature Envy-Intensive Coupling* resulted in a decrease for the coupling in **2.59%** and has not significantly changed any other attributes. Additionally, the removal of the co-occurrence *Feature Envy-Long Method* was the only one that did not improve any internal quality attribute, since it resulted in

Table 7: The impact of code smell co-occurrences removal for internal quality attributes.

Co-occurrence	Cohesion	Complexity	Coupling	Inheritance
Feature Envy–God Class	↑ 10.51%	↓30.98%	↓21.52%	-
God Class–Shotgun Surgery	↑ 2.59%	↓27.61%	↓19.14%	-
Dispersed Coupling–God Class	↑ 3.16%	↓24.59%	↓20.00%	↑ 3.57%
Feature Envy–Long Method	↓16.17%	↑ 2.95%	↑ 4.99%	-
Intensive Coupling–Long Method	↓19.90%	↑ 12.63%	↓6.81%	-
Dispersed Coupling–Long Method	↓30.80%	↓9.62%	↓15.48%	↑ 5.56%
Dispersed Coupling–Feature Envy	↓39.76%	↓22.12%	↓13.50%	-
Feature Envy–Intensive Coupling	-	-	↓2.59%	-
God Class–Long Method	↑ 19.97%	↓41.59%	↓33.98%	↑ 11.00%

the worsening of the cohesion in **16.17%**, the complexity at **2.95%** and coupling increased by **4.99%**. Finally, the removal of *Dispersed Coupling-Feature Envy* has improved 2 attributes: decreased the complexity in **22.12%** and coupling in **13.50%**. However, but it worsened the cohesion in **39.76%**.

> **Finding 3**: The removal of *Feature Envy-Long Method* suggests a negative effect on cohesion, complexity and coupling.

Implications of RQ$_1$. Our findings suggests that the following co-occurrences *Dispersed Coupling-God Class*, *God Class-Long Method*, *Feature Envy-God Class* and *God Class-Shotgun Surgery* are extremely harmful to the software quality and that their removals results in an improvement in the internal quality attributes. Such findings also suggest that the removal of certain co-occurrences improves the software quality, and thus confirms the observation of prior studies on the removal of code smell co-occurrences [13, 24, 50]. Additionally, the removal of co-occurrences such as *Dispersed Coupling-Feature Envy* and *Intensive Coupling-Long Method* improves certain attributes and worsens others. Thus, project managers and developers can choose to remove or not these co-occurrences aimed to improve a certain internal quality attribute. Finally, the removal of *Feature Envy-Long Method* did not benefit any of the internal quality attributes, suggesting that the removal of some co-occurrences might result in a negative effect on quality [25].

4.2 Most Harmful Co-occurrences from the Developers' Perspective (RQ$_2$)

We address **RQ$_2$** by analyzing the commits of refactoring of code smell co-occurrences, and the developers' responses that were written using the diary technique.

Table 8: The most harmful code smell co-occurrences under the developers' perception

Co-occurrences	Developer	#Developers
Feature Envy–God Class	P1,P5,P6,P11,P12,P13	6
Dispersed Coupling–God Class	P3,P8,P9,P14,P7	5
God Class–Long Method	P2,P4,P10	3

Table 8 presents the most harmful co-occurrences according to the developers' perception. The first column shows the type of co-occurrences and the second column lists the developers who considered the co-occurrence as most harmful. Finally, the last column summarizes the number of developers who considered the co-occurrence as most harmful.

The most harmful co-occurrences under developers' perception. Table 8 allows us to observe that the co-occurrences considered most harmful under the developers' perception were: *Feature Envy-God Class*, mentioned by five developers; *Dispersed Coupling–God Class* mentioned by four developers; and *God Class–Long Method* mentioned by three developers. We can also observe that the code smell *God Class* is present in the three co-occurrences mentioned by the developers. This suggests that a co-occurrence containing this smell can be considered harmful by developers. Such observation was mentioned by the developers as follows.

P11: "Feature Envy and God Class are the most harmful, and in my opinion the God Class smells is most harmful due to the difficulty [in removing them]"

P5: "The class that has the God Class would need many refactorings to decrease its size considerably."

> **Finding 4**: The presence of the God Class smell in a co-occurrence suggests that co-occurrence is more likely to be considered harmful by software developers.

Although the aforementioned co-occurrences have been considered to be the most harmful in the developers' perception, the removal of these co-occurrences via refactoring had a positive impact on the internal quality attributes. In fact, in Table 7, we observed that the removal of *Dispersed Coupling–God Class*, and *God Class–Long Method* improved all internal quality attributes, and the removal of *Feature Envy–God Class* improved the attributes of cohesion, complexity and coupling. Such observations suggest that the presence of these co-occurrences is harmful to quality internal attributes and developers. Therefore, these co-occurrences must be removed as possible. Additionally, software developers should be more attentive not to introduce these co-occurrences during the development process, since these co-occurrences tend to appear frequently in software systems [22, 24, 29, 51].

> **Finding 5**: Developers should pay attention to remove and avoid the introduction of *Dispersed Coupling–God Class*, *Feature Envy–God Class* and *God Class–Long Method* co-occurrences during the software development.

Co-occurrences that needed more commits to be removed. Table 9 overviews the number of days and commits required to remove each code smell co-occurrence. The first column names the co-occurrences. The second column shows the number of occurrences by type considerate all systems. The third column shows the total number of commits to remove a type of co-occurrence. The

Table 9: Information on number of days and commits to remove co-occurrences in the five target systems

Co-occurrence	Number of occurrences	Total Commits	Total Days	Average Commits	Average Days
Feature Envy–God Class	24	106	50	4.41	2.08
God Class–Shotgun Surgery	6	12	12	2	2
Dispersed Coupling–God Class	13	84	30	6.46	2.3
Feature Envy–Long Method	11	50	13	4.54	1.18
Intensive Coupling–Long Method	8	46	25	5.75	3.12
Dispersed Coupling–Long Method	11	69	17	6.27	1.54
Dispersed Coupling–Feature Envy	2	7	2	3.5	1
Feature Envy–Intensive Coupling	1	4	1	4	1
God Class–Long Method	6	42	12	7	2

fourth column shows the total number of days to remove a type of co-occurrence. Finally, the fifth and sixth columns show the average of commits and the average number of days required to remove one occurrence per type. For instance, the *Feature Envy-God Class* co-occurrence had appeared 24 times on the five target systems and required 106 commits and 30 days to be completely removed and to remove an instance of this co-occurrence, the developers took an average of 4.41 commits and 2.18 days.

By looking at Table 9, we can observe that the *Feature-Envy–God Class* co-occurrence was the one that had the most commits for its removal, but also it was the one that had more appear (24) in the five systems. Thus, we used the average as a measure to evaluate the number of commits that are required to remove a co-occurrence. Therefore, the *God Class–Long Method* co-occurrence had the highest average with **seven commits**, i.e., was required an average of seven commits for removing an instance of *God Class-Long Method* and a total of 42 commits were needed to refactor all six occurrences of this co-occurrence. Other co-occurrences that had high average commits were *Dispersed Coupling-God Class* with an average of **6.46 commits**; *Dispersed Coupling-Long Method* with an average of **6.27 commits**; and *Intensive Coupling-Long Method* with an average of **5.75 commits**.

Co-occurrences that needed more time to be removal. Similar to the number of commits, we analyzed the average days instead of total days. The co-occurrence with the highest average of days was *Intensive Coupling-Long Method* with **3.12 days**, i.e., every 3.12 days it was possible to remove one out of eight occurrences of *Intensive Coupling-Long Method*. This is an interesting result because despite this co-occurrence have the highest average number of days to be removal via refactoring, no developer has mentioned this co-occurrence as the most harmful. Other co-occurrences that had high average days were: *Dispersed Coupling-God Class* with **2.3 days** average, *Feature Envy-God Class* with **2.08 days** and *God Class-Long Method* with **2 days** average. Such results are in line with the developers' perception, since these three co-occurrences were mentioned by the developers as the most harmful.

> **Finding 6**: The *Intensive Coupling-Long Method*, *Dispersed Coupling-God Class*, *Feature Envy-God Class* and *God Class-Long Method* co-occurrences are the ones that take the longest to be removed by software developers.

Implications of RQ_2. Our findings suggest that the presence of the God Class smell can lead to harmful co-occurrences. Additionally, developers should prioritize the removal of co-occurrences such as *Dispersed Coupling-God Class*, *Feature Envy-God Class* and *God Class-Long Method*. Moreover, if the software development is at the beginning, it is recommended to avoid the introduction of these smells as they are harmful, and because they take the longest to be removed, together with *Intensive Coupling-Long Method*.

4.3 Main Difficulties Faced by Developers During the Co-occurrences Removal (RQ_3)

We address $\mathbf{RQ_3}$ by analyzing the developers' responses that were written using the diary technique. Thus, from the responses collected, we made a qualitative analysis and four categories were identified: (1) *Difficulty in Understanding the Source Code*; (2) *Complexity of Methods and Functions*; (3) *Refactoring Effort*; and (4) a *Large Amount of Source Code*. Table 10 describes the categories identified during the analysis of the developers' responses. The first column refers to the category and the second column its description.

Table 10: Description of the categories

Category	Description
Difficulty in Understanding the Source Code	Refers to the difficulty of understanding the source code by the developer.
Complexity of Methods and Functions	It is the definition of Methods or functions that make many calls to other methods in the source code.
Refactoring Effort	Refers to a high level of work and rework in the refactoring activities of the source code by the developer.
Large Amount of Source Code	It refers to the massive amount of source code written in software, either in a class or in a method.

The main difficulties faced by software developers. Figure 1 shows the categories and their relationships with the main difficulties faced by developers during the removal of co-occurrences.

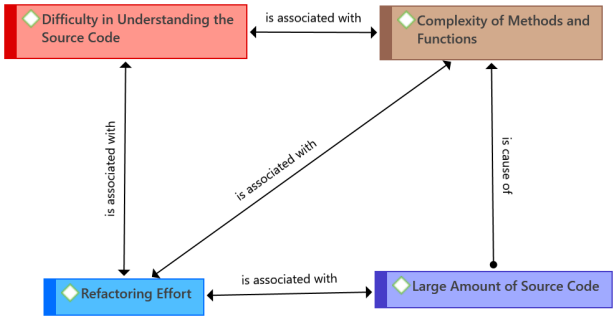

Figure 1: Categories and relationships identified for developers' difficulties

Looking at Figure 1, we can identify the categories and relationships associated with the main difficulties faced by developers

during the removal of code smell co-occurrences. The association between *Difficulty in Understanding the Source Code* and *Complexity of Methods or Functions* is an aspect highlighted in our research, and this is reinforced by:

P4:"One method doing the responsibilities of another method, makes it a little tricky to understand what is going on."

P8:"The co-occurrences that I am removing use several methods and functions of other classes, and this affects the legibility of the class and the method, decreasing the understanding of the code."

P6:"They affect the initial understanding of how the code works, given the excess of methods called in a class or method. "

These observations suggests that co-occurrences of code smells are usually associated with complex methods. We also identified a relationship between *Difficulty in Understanding the Source Code* and *Refactoring Effort* that can be highlighted through the following statements made by the developers:

P4:"The biggest difficulty is to understand the code, because to remove the anomalies it was necessary to study the logic of the code. In addition, I had difficulty in identifying the anomaly location."

P2:"It is more difficult to understand, because the co-occurrence increases coupling"

Thus, we can observe that the difficulty of understanding the code is a crucial factor for the co-occurrence removal. Our analysis suggests that the less the developer understands the source code with co-occurrence, the more work the developer will have to remove it.

Finding 7: One of the main difficulties for software developers is to understand the source code that contains co-occurrences.

The *Refactoring Effort* category relates to two other categories which are: *Complexity of Methods or Functions* and *Large Amount of Source Code*. An example of how *Refactoring Effort* is associated with a *Large Amount of Source Code*, and the relationship between *Refactoring Effort* and *Complexity of Methods or Functions* are highlighted as follows:

P11:"The co-occurrences made the code very large, and this made the code [more] complex for future maintenance"

P11:"The co-occurrence is harmful because it calls various methods for other functions, and thus requires more attention in its removal, as long as it does not interfere with the functioning of the system."

In this context, we can observe that complex methods and large code are directly related to a greater refactoring effort during the removal of code smells co-occurrences. Our analyzes suggest that the presence of code smells co-occurrences can lead to a massive increase in the source code, leaving the complex methods and consequently increasing the developers' refactoring effort [36]. We also observed a relationship between a *Large Amount of Source Code* and *Complexity of Methods or Functions*. We found that a large amount of code can mean greater complexity of methods or functions. Therefore, developers should pay attention to the source code size that they produce, since this code with co-occurrences can generate complex methods. This observation can be explained from the following comment:

P9:"The co-occurrences make the code large and disorganized, making classes and methods more complex."

Finding 8: The more complex and extensive the methods, the greater refactoring effort required to remove co-occurrences.

Developers still do not feel safe to identify and remove code smells co-occurrences. We observed that developers still have insecurities concern the identification and removal of co-occurrences of code smells. Some developers are not sure whether they removed the co-occurrence completely or if they removed it correctly. Such insecurities can be observed as follows:

P1:"I have difficulty removing the co-occurrence and checking if the solution that I made was adequate and if it actually solves the smell."

P6:"Sometimes it was difficult to identify where the anomaly was affecting."

P8:'I have difficulty analyzing what to do to refactor the anomaly, and decide what is the refactoring operation in each situation.'

These observations suggest that the developer does not feel safe to apply the refactoring operations and completely remove the co-occurrence of code smell. In addition, sometimes the developer does not know or is not sure whether the solution used for removal was the best one at that time, and this can negatively affect the quality of the software instead of improving it.

Implications of RQ_3. Our findings suggest that complex methods can hinder the developers' understanding during the removal of co-occurrences, suggesting a greater effort during the refactoring activities. In fact, one of the characteristics of co-occurrences is to make it difficult to understand the source code [34]. A large amount of source code can generate complex methods and thus indicate a greater refactoring effort. As such, we believe that developers should adopt source code optimization as much as possible in their daily practice. In addition, we can observe that developers still have insecurities in the removal and identification of co-occurrences of code smells. Despite our results on the developers' perception regarding the removal of co-occurrences of code smells, further studies are still needed to understand the developer during the removal of co-occurrences [7, 8, 16, 19].

5 THREATS TO VALIDITY

Internal validity. One of the threats is the small number of systems and code lines analyzed in our study. However, the systems are closed-source, and we wanted to have a deeper understanding of those systems. Another threat identified is that we analyzed only classes in production and not other types of classes (e.g., test classes) at the time of detecting co-occurrences and measuring quality. However, we consider that developers are more concerned with classes with features linked to the system than the test classes.

Construct validity. We use JSpIRIT and JDeodorant tools to detect code smells and their co-occurrences. Both tools have defined detection strategies, and this can be a potential threat to validity, as other tools with different approaches could identify different code smells. However, we chose these tools because they are accurate and well consolidated. Another threat to validity found is that the participants are the subjective filling of the diary by the developers.

However, to mitigate this threat, we explain in detail the goal of each question in the diary during the training session.

External validity. The results found in our study are used for systems implemented using the the Java programming language. Another limitation is that some developers had little knowledge of the diary technique, code smells, quality metrics and refactoring. To mitigate this threat, we conduct training with all developers.

6 RELATED WORK

Refactoring code smells and internal quality attributes. Fernandes et al. [11] conducted a quantitative study on the impact of refactoring on internal quality attributes. The results found by the authors pointed out that most refactorings have improved one or more internal quality attributes. Chávez et al. [5] conducted a study to investigate the impact of refactorings on the internal quality attributes in 23 open-source projects implemented in Java. They found that in 65% of the cases, the internal quality attributes improved while whereas in the other 35%, these attributes remained unchanged. However, these studies did not assess the impact of removing co-occurrences for internal quality attributes or developers' perceptions. In our study, we found that removing certain co-occurrences can improve internal quality attributes and that developers still have insecurities when removing co-occurrences.

Detection and analysis of co-occurrences of code smells. Palomba et al. [29] conducted a large-scale empirical study to quantify and analyze which code smells are most likely to co-occur during the software development cycle. As a result, the authors identified that 59% of the classes are measured by more than one occurrence of code smells. Walter et al. [47] performed an empirical analysis of collocated smells, which are interactions of code smells co-occurrences in the same file, and involved a set of 92 systems from different domains detecting 14 different code smells using 6 tools. As a result, the authors identified in all the domains analyzed that there is a certain group of code smells that tend to co-occur, such as: *Brain Class*, *God Class*, *Dispersed Coupling* and *Long Method*. Walter et al. [47] and Palomba et al. [29] take into account code smells co-occurrences. However, in both studies, an analysis is made of which smells are more likely to co-occur and do not investigate the impact of these anomalies on code quality.

Code smells co-occurrences and software quality. Yamashita and Moonen [49] analyzed the impact of co-occurrences for the maintainability of 4 medium-sized systems in Java. The authors noted that co-occurrences negatively affect maintainability and software maintenance activities. Oizumi et al. [28] carried out a study to investigate whether code smells co-occurrences, which the authors call agglomerations, which can mean software design problems. The results found that co-occurrences of code smells can cause problems in software design and are an effective approach to locating these problems. Politowski et al. [34] conducted a study with 133 participants and 372 comprehension activities involving co-occurrences of two code smells: *Blob* and *Spaghetti Code*. The study aimed to investigate the developers' understanding of the source code from the co-occurrences of these two anomalies. The results found by the authors showed that the readability and understanding of the code worsened as the developers took longer to finish their activities. The effort was greater to complete them.

Martins et al. [24] investigated the impact of code smells co-occurrences in the internal quality attributes in 3 Java closed-source systems. The authors identified which are the co-occurrences to be removed according to the developers' perspective. As a result, it was identified that *God Class-Long Method* and *Disperse Coupling-Long Method* are the most frequent co-occurrences in the three systems and also the most difficult co-occurrences to refactor in developers' perspective. Besides, removing these smells harms the cohesion and coupling attributes and suggests a significant decrease in the complexity of the systems. All previous studies analyze the impact of co-occurrences for some quality factor. However, none of the works identifies the most harmful co-occurrences taking into account the effort to remove these co-occurrences by the developers.

7 CONCLUSION AND FUTURE WORK

Our study considered 6 types of code smells and their co-occurrences in 5 Java closed-source systems and 4 internal quality attributes (cohesion, inheritance, coupling, and complexity). As the main goal of our study: (i) we investigated the impact of removing these smells co-occurrences for internal quality attributes and the most harmful code smell co-occurrences in these systems; (ii) we also identified the most harmful code smell co-occurrences from the developer's perspective; and (iii) we analyzed the main difficulties and perceptions of developers during the removal of code smell co-occurrences. Removing these co-occurrences of code smells took 3 months and happened at different times for each system. Were made in the study a total of 420 refactoring commits, and 14 developers removed 82 co-occurrences. During the entire process of removal co-occurrences via refactoring, we instruct developers to document their perceptions using the diary technique.

Our main findings were: (i) the removal of *Disperse Coupling-God Class* and *God Class-Long Method* has improved all internal quality attributes; (ii) developers should pay attention to remove and avoid the introduction of *Dispersed Coupling–God Class*, *Feature Envy–God Class* and *God Class–Long Method* co-occurrences during the software development; (iii) one of the main difficulties for developers is to understand the source code that contains co-occurrences; and, (iv) the developers still have insecurities regarding the identification and refactoring of code smells and their co-occurrences. The finding (iv) is interesting because removing certain co-occurrences improved the software quality, and thus confirms the observation of prior studies on the removal of code smell co-occurrences [13, 24, 50]. As future work, we intend to: (i) analyze the co-occurrence in open-source systems; (ii) reproduce the study with tools that detect other code smells; (iii) reproduce the study with systems written in other programming languages; (iv) study the co-occurrence phenomenon of code smells in android; and, (v) use automatic refactoring to remove code smells co-occurrences.

Acknowledgements. This work was partially funded by CNPq (434969/2018-4, 312149/2016-6, 141285/2019-2), FAPERJ (200773/2019, 010002285/2019), and CAPES/Procad (175956).

REFERENCES

[1] Walid Abdelmoez, Essam Kosba, and Ali Falah Iesa. 2014. Risk-based code smells detection tool. In *The International Conference on Computing Technology and Information Management (ICCTIM)*. Society of Digital Information and Wireless Communication, 148.

[2] Chaima Abid, Vahid Alizadeh, Marouane Kessentini, Thiago do Nascimento Ferreira, and Danny Dig. 2020. 30 Years of Software Refactoring Research: A Systematic Literature Review. *arXiv preprint arXiv:2007.02194* (2020).

[3] Ana Carla Bibiano, Vinicius Soares, Daniel Coutinho, Eduardo Fernandes, João Lucas Correia, Kleber Santos, Anderson Oliveira, Alessandro Garcia, Rohit Gheyi, Baldoino Fonseca, Márcio Ribeiro, Caio Barbosa, and Daniel Oliveira. 2020. How Does Incomplete Composite Refactoring Affect Internal Quality Attributes?. In *28th ICPC*. 149–159.

[4] James M Bieman and Byung-Kyoo Kang. 1995. Cohesion and reuse in an object-oriented system. *ACM SIGSOFT Software Engineering Notes* 20, SI (1995), 259–262.

[5] Alexander Chávez, Isabella Ferreira, Eduardo Fernandes, Diego Cedrim, and Alessandro Garcia. 2017. How does refactoring affect internal quality attributes?: A multi-project study. In *31st SBES*. ACM, 74–83.

[6] Shyam R Chidamber and Chris F Kemerer. 1994. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.* 20, 6 (1994), 476–493.

[7] Rafael de Mello, Anderson Uchôa, Roberto Oliveira, Willian Oizumi, Jairo Souza, Kleyson Mendes, Daniel Oliveira, Baldoino Fonseca, and Alessandro Garcia. 2019. Do research and practice of code smell identification walk together? a social representations analysis. In *13th ESEM*. 1–6.

[8] Rafael Maiani de Mello, Anderson Gonçalves Uchoa, Roberto Felicio Oliveira, Daniel Tenório Martins de Oliveira, Baldoino Fonseca, Alessandro Fabricio Garcia, and Fernanda de Barcellos de Mello. 2019. Investigating the social representations of code smell identification: a preliminary study. In *12th CHASE*. 53–60.

[9] Elder Vicente de Paulo Sobrinho, Andrea De Lucia, and Marcelo de Almeida Maia. 2018. A systematic literature review on bad smells—5 W's: which, when, what, who, where. *IEEE Trans. Softw. Eng.* (2018).

[10] Giuseppe Destefanis, Steve Counsell, Giulio Concas, and Roberto Tonelli. 2014. Software metrics in agile software: An empirical study. In *International Conference on Agile Software Development*. Springer, 157–170.

[11] Eduardo Fernandes, Alexander Chávez, Alessandro Garcia, Isabella Ferreira, Diego Cedrim, Leonardo Sousa, and Willian Oizumi. 2020. Refactoring Effect on Internal Quality Attributes: What Haven't They Told You Yet? *Inf. Softw. Technol.* (2020), 106347.

[12] Eduardo Fernandes, Johnatan Oliveira, Gustavo Vale, Thanis Paiva, and Eduardo Figueiredo. 2016. A review-based comparative study of bad smell detection tools. In *20th EASE*. ACM, 18.

[13] Eduardo Fernandes, Gustavo Vale, Leonardo Sousa, Eduardo Figueiredo, Alessandro Garcia, and Jaejoon Lee. 2017. No Code Anomaly is an Island. In *16th ICSR*. Springer, 48–64.

[14] Martin Fowler. 2018. *Refactoring: improving the design of existing code*. Addison-Wesley Professional.

[15] César França, Fabio Q. B. da Silva, and Helen Sharp. 2020. Motivation and Satisfaction of Software Engineers. *IEEE Transactions on Software Engineering* 46, 2 (2020), 118–140. DOI: https://doi.org/10.1109/TSE.2018.2842201

[16] Amandeep Kaur and Gaurav Dhiman. 2019. A review on search-based tools and techniques to identify bad code smells in object-oriented systems. In *Harmony search and nature inspired optimization algorithms*. Springer, 909–921.

[17] Satnam Kaur and Paramvir Singh. 2019. How does object-oriented code refactoring influence software quality? Research landscape and challenges. *J. Syst. Softw.* 157 (2019), 110394.

[18] Sharanpreet Kaur and Satwinder Singh. 2016. Spotting & eliminating type checking code smells using eclipse plug-in: Jdeodorant. *International Journal of Computer Science and Communication Engineering* 5, 1 (2016).

[19] Guilherme Lacerda, Fabio Petrillo, Marcelo Pimenta, and Yann Gaël Guéhéneuc. 2020. Code smells and refactoring: a tertiary systematic review of challenges and observations. *J. Syst. Softw.* (2020), 110610.

[20] Michele Lanza and Radu Marinescu. 2007. *Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems*. Springer Science & Business Media.

[21] Mark Lorenz and Jeff Kidd. 1994. *Object-oriented software metrics: a practical guide*. Prentice-Hall, Inc.

[22] Angela Lozano, Kim Mens, and Jawira Portugal. 2015. Analyzing code evolution to uncover relations. In *2nd PPAP*. IEEE, 1–4.

[23] Ruchika Malhotra and Anuradha Chug. 2016. An empirical study to assess the effects of refactoring on software maintainability. In *International Conference on Advances in Computing, Communications and Informatics (ICACCI)*. IEEE, 110–117.

[24] Júlio Martins, Carla Bezerra, Anderson Uchôa, and Alessandro Garcia. 2020. Are Code Smell Co-Occurrences Harmful to Internal Quality Attributes? A Mixed-Method Study. In *34th SBES (SBES '20)*. Association for Computing Machinery, New York, NY, USA, 52–61. DOI: https://doi.org/10.1145/3422392.3422419

[25] Júlio Martins, Carla Ilane Moreira Bezerra, and Anderson Uchôa. 2019. Analyzing the Impact of Inter-smell Relations on Software Maintainability: An Empirical Study with Software Product Lines. In *15th SBSI*. 1–8.

[26] Thomas J McCabe. 1976. A complexity measure. *IEEE Trans. Softw. Eng.* 4 (1976), 308–320.

[27] Tom Mens and Tom Tourwé. 2004. A survey of software refactoring. *IEEE Trans. Softw. Eng.* 30, 2 (2004), 126–139.

[28] Willian Oizumi, Alessandro Garcia, Leonardo da Silva Sousa, Bruno Cafeo, and Yixue Zhao. 2016. Code anomalies flock together: Exploring code anomaly agglomerations for locating design problems. In *38th ICSE*. IEEE, 440–451.

[29] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Fausto Fasano, Rocco Oliveto, and Andrea De Lucia. 2018. A large-scale empirical study on the lifecycle of code smell co-occurrences. *Inf. Softw. Technol.* 99 (2018), 1–10.

[30] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Fausto Fasano, Rocco Oliveto, and Andrea De Lucia. 2018. On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. *Empir. Softw. Eng.* 23, 3 (2018), 1188–1221.

[31] Fabio Palomba, Rocco Oliveto, and Andrea De Lucia. 2017. Investigating code smell co-occurrences using association rule learning: A replicated study. In *2nd MaLTeSQuE*. IEEE, 8–13.

[32] Jeremy R Pate, Robert Tairas, and Nicholas A Kraft. 2013. Clone evolution: a systematic review. *Journal of software: Evolution and Process* 25, 3 (2013), 261–283.

[33] Błażej Pietrzak and Bartosz Walter. 2006. Leveraging code smell detection with inter-smell relations. *Extreme Programming and Agile Processes in Software Engineering* (2006), 75–84.

[34] Cristiano Politowski, Foutse Khomh, Simone Romano, Giuseppe Scanniello, Fabio Petrillo, Yann-Gaël Guéhéneuc, and Abdou Maiga. 2020. A large scale empirical study of the impact of Spaghetti Code and Blob anti-patterns on program comprehension. *Inf. Softw. Technol.* 122 (2020), 106278.

[35] José Amancio M Santos, João B Rocha-Junior, Luciana Carla Lins Prates, Rogeres Santos do Nascimento, Mydiã Falcão Freitas, and Manoel Gomes de Mendonça. 2018. A systematic review on the code smell effect. *J. Syst. Softw.* 144 (2018), 450–477.

[36] Vinícius Soares, Anderson Oliveira, Juliana Alves Pereira, Ana Carla Bibano, Alessandro Garcia, Paulo Roberto Farah, Silvia Regina Vergilio, Marcelo Schots, Caio Silva, Daniel Coutinho, and others. 2020. On the Relation between Complexity, Explicitness, Effectiveness of Refactorings and Non-Functional Concerns. In *34th SBES*. 788–797.

[37] Leonardo Sousa, Anderson Oliveira, Willian Oizumi, Simone Barbosa, Alessandro Garcia, Jaejoon Lee, Marcos Kalinowski, Rafael de Mello, Baldoino Fonseca, Roberto Oliveira, and others. 2018. Identifying design problems in the source code: A grounded theory. In *40th ICSE*. 921–931.

[38] Klaas-Jan Stol, Paul Ralph, and Brian Fitzgerald. 2016. Grounded theory in software engineering research: a critical review and guidelines. In *38th ICSE*. 120–131.

[39] Amjed Tahir, Aiko Yamashita, Sherlock Licorish, Jens Dietrich, and Steve Counsell. 2018. Can you tell me if it smells? A study on how developers discuss code smells and anti-patterns in Stack Overflow. In *22nd EASE*. 68–78.

[40] Sandhya Tarwani and Anuradha Chug. 2016. Sequencing of refactoring techniques by Greedy algorithm for maximizing maintainability. In *Proceedings of the International Conference on Advances in Computing, Communications and Informatics (ICACCI)*. IEEE, 1397–1403.

[41] Nikolaos Tsantalis, Theodoros Chaikalis, and Alexander Chatzigeorgiou. 2018. Ten years of JDeodorant: Lessons learned from the hunt for smells. In *25th SANER*. IEEE, 4–14.

[42] Anderson Uchôa, Caio Barbosa, Daniel Coutinho, Willian Oizumi, Wesley KG Assunçao, Silvia Regina Vergilio, Juliana Alves Pereira, Anderson Oliveira, and Alessandro Garcia. 2021. Predicting Design Impactful Changes in Modern Code Review: A Large-Scale Empirical Study. In *18th MSR*. IEEE, 1–12.

[43] Anderson Uchôa, Eduardo Fernandes, Ana Carla Bibiano, and Alessandro Garcia. 2017. Do Coupling Metrics Help Characterize Critical Components in Component-based SPL? An Empirical Study. In *5th VEM*. 36–43.

[44] Anderson Uchôa. 2021. Unveiling Multiple Facets of Design Degradation in Modern Code Review. In *29th ESEC/FSE*. 1–5.

[45] Anderson Uchôa, Caio Barbosa, Willian Oizumi, Publio Blenílio, Rafael Lima, Alessandro Garcia, and Carla Bezerra. 2020. How Does Modern Code Review Impact Software Design Degradation? An In-depth Empirical Study. In *36th ICSME*. 1 – 12.

[46] Santiago A Vidal, Claudia Marcos, and J Andrés Díaz-Pace. 2016. An approach to prioritize code smells for refactoring. *Automated Software Engineering* 23, 3 (2016), 501–532.

[47] Bartosz Walter, Francesca Arcelli Fontana, and Vincenzo Ferme. 2018. Code smells and their collocations: A large-scale experiment on open-source systems. *J. Syst. Softw.* 144 (2018), 1–21.

[48] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. 2012. *Experimentation in software engineering*. Springer Science & Business Media.

[49] Aiko Yamashita and Leon Moonen. 2013. Do developers care about code smells? An exploratory survey. In *20th WCRE*. IEEE, 242–251.

[50] Aiko Yamashita and Leon Moonen. 2013. Exploring the impact of inter-smell relations on software maintainability: An empirical study. In *35th ICSE*. IEEE, 682–691.

[51] Aiko Yamashita, Marco Zanoni, Francesca Arcelli Fontana, and Bartosz Walter. 2015. Inter-smell relations in industrial and open source systems: A replication and comparative analysis. In *31st ICSME*. IEEE, 121–130.