On the Influential Interactive Factors on Degrees of Design Decay: A Multi-Project Study

Daniel Coutinho*, Anderson Uchôa†, Caio Barbosa*, Vinícius Soares*,
Alessandro Garcia*, Marcelo Schots‡, Juliana Pereira*, Wesley K. G. Assunção*
*Informatics Department – Pontifical Catholic University of Rio de Janeiro (PUC-Rio), Rio de Janeiro, Brazil
†Federal University of Ceará (UFC), Itapajé, Brazil
‡Informatics and Computer Science Department – Rio de Janeiro State University (UERJ), Rio de Janeiro, Brazil

***Abstract*—Developers constantly perform code changes throughout the lifetime of a project. These changes may induce the introduction of design problems (design decay) over time, which may be reduced or accelerated by interacting with different factors (*e.g.*, refactorings) that underlie each change. However, existing studies lack evidence about how these factors interact and influence design decay. Thus, this paper reports a study aimed at investigating whether and how (associations of) process and developer factors influence design decay. We studied seven software systems, containing an average of 45K commits in more than six years of project history. Design decay was characterized in terms of five internal quality attributes: cohesion, coupling, complexity, inheritance, and size. We observed and characterized 12 (sub-)factors and how they associate with design decay. To this end, we employed association rule mining. Moreover, we also differentiate between the associations found on modules with varying levels of decay. Process- and developer-related factors played a key role in discriminating these different levels of design decay. Then, we focused on analyzing the effects of potentially interacting factors regarding slightly- and largely-decayed modules. Finally, we observed diverging decay patterns in these modules. For example, individually, the developer-related sub-factor that represented first-time contributors, as well as the process-related one that represented the size of a change did not have negative effects on the changed classes. However, when analyzing specific factor interactions, we saw that changes in which both of these factors interacted tended to have a negative effect on the code, leading to decay.**

***Index Terms*—design decay, software quality, software metrics**

I. Introduction

Developers constantly change software systems, submitting their contributions to version control repositories, and allowing other developers to review [1] and discuss [2] their contents. Often, these changes may induce design decay over time [3], [4], whose symptoms manifest when the software modules become increasingly complex, large, coupled, and incohesive [5]. Decay levels can also vary among different modules of a system, since they evolve asymmetrically. Also, developers commonly apply certain repair actions (e.g., refactorings) in an attempt to potentially revert design decay [6], which can further contribute to this asymmetry.

Design decay is often studied as a cut-and-dried issue [7]. Still, a myriad of factors can influence – either alone or simultaneously – how decay occurs, and to which extent it can be slowed down or accelerated. These influence factors can vary in nature and can also affect, to a greater or lesser extent, the effectiveness of repair actions. As such, the quality of a change might be influenced by the developer(s) working on it [8], [9] and the change process itself [10], [11]. Process-related factors include actions and outcomes associated with the resulting software changes [9], [12], [13], e.g., refactorings. Conversely, developer-related factors include ways in which they discuss and contribute to a system [14], e.g., number of pull requests that have modified a file.

Previous studies investigate whether these factors, in isolation, affect design decay [15]. Some studies focus on analyzing either developer- [14], [16] or process-related [15], [17] factors. These studies often disregard that modules are subjected to varying levels of decay and to evolving combinations of influential factors. These studies also do not differentiate preliminary or advanced stages of design decay [18]. Thus, better understanding of potential associations between those factors, as well as to which levels they can affect design decay, are currently missing in the literature. For example, it is unknown whether and how these complementary factors simultaneously and progressively influence decay. Thus, one cannot effectively explore which factors (or associations between them) should be monitored to avoid decay.

In this paper, we address these gaps through a multi-project study that aims to reveal characteristics of certain influential factors, and their interactions, in varying levels of design decay. For this purpose, we divided modules into two groups based on how intensely they were affected by decay, namely slightly-decayed and largely-decayed classes. To analyze how developer- and process-related factors can relate to differences between these groups, we use association rule mining [19]. We mined associations from nearly 45k commits made to seven systems. We analyzed 38 software metrics to characterize the design decay of modules with respect to five internal quality attributes. To support our observations, we also analyzed 12 (sub-)factors, quantified by metrics, representing process- and developer-related factors.

From these observations, we have derived several findings that include: (i) we observed how design decay behaved differently on seven different systems and observed specific behaviors to certain decay levels or systems; (ii) we found that seven out of the 12 sub-factors analyzed are good individual indicators of design decay; (iii) we observed that

 To be published in the Proceedings 29th International Conference on Software Analysis, Evolution, and Reengineering (SANER), 2022 in Honolulu, Hawaii.

extraction refactorings surprisingly have a negative effect on the internal quality attributes (IQAs); and (iv) we discovered several negative effects pertaining to changes executed by newcomers. Besides those major findings, a myriad of other minor findings are presented in the remaining sections of this work. The contributions of this paper also include discussions on the implications of the aforementioned findings and a novel dataset [20], allowing researchers to further investigate how these factors influence the IQAs over time.

II. Background

A. Design Decay and Internal Quality Attributes

Design decay is a phenomenon in which developers progressively introduce design problems in a system [4]. It is caused by design decisions that negatively impact aspects such as maintainability and extensibility [7], [21]. Given the potential harmfulness of design decay, developers often need to identify and refactor impacted source code locations [22]. A key indicator used by developers to reduce design decay is the improvement of internal quality attributes (IQAs), such as *coupling*, *cohesion*, and *complexity* [7], [23]. Two other IQAs considered by developers are: *size* and *inheritance* [7], [24], [25]. They are described as follows.

Coupling represents the degree of interdependency between classes [24]. In this sense, a high *coupling* negatively affects the maintainability of a set of classes, making so that even a small change in a method in one highly coupled class can have a large, and possibly hard to predict, effect on many other classes. In the case of *cohesion*, it represents the degree to which the internal elements of a module are related to each other. Low *cohesion* within a class might lead to developers having difficulty understanding the responsibilities of that class, thus potentially leading that class to become hard to change and bug-prone. Regarding *complexity*, it represents the level of overload of responsibilities and decisions of a module or class. It causes difficulties in the reading and understanding of classes by developers, who end up making mistakes.

In the case of *inheritance*, it represents the complexity of a class's inheritance tree – due to the number of superclasses and subclasses in the hierarchy. A class with a complex *inheritance* tree may cause additional difficulty during maintenance. Finally, as for the *size* attribute, it represents the physical size of a code structure (e.g. method, class), often measured by the number of lines of code. It is often considered as a measure of code quality, due to its connection to bad code structures, such as long methods and classes. However, the semantics of high and low values of the metrics related to this size vary widely throughout a software project's evolution and across different projects. Thus, care should be taken when considering size changes as being a positive or negative influence to the code's quality. Their meanings are hard to interpret in each context. Due to the importance of IQAs for detecting design decay, in this work, we selected a set of 38 metrics from the literature that quantify a variety of properties of each IQA. These metrics are explained in more detail in Section III-B.

B. Influential Factors along Software Development

Software development is constantly affected by a myriad of factors. For convenience, these factors can be grouped into a number of key categories, e.g. process-related factors, developer-related factors, technical factors, organizational factors, among others. In this work, we focus on two of these groups, namely, *process-related factors* and *developer-related factors*. They were chosen because there are known ways to observe and measure them during a project's evolution, without requiring runtime information, observations external to the repository or that the changes have been fully integrated into the software system. Thus, by leveraging these factors, a preventive approach can be used for design decay avoidance.

Process-related factors reflect the changes themselves and their outcome, and can be seen as representing the universe of files, code segments, code changes and repair actions (e.g. refactorings). On the other hand, *developer-related factors* emerge from collaboration among developers [26], and represent the human-centered actions in code contributions, e.g., code review discussions. Previous studies [14], [17], [27], [28], consistently report that those groups of influential factors may contribute to avoiding, reducing, or accelerating design decay.

In this work, process-related factors are composed of two factors: *Change outcomes* and *Refactoring actions*. *Change outcomes* represent the properties of each change performed by developers during development [9], [12], [13]. *Refactoring actions* represent code transformations usually made by developers to avoid or mitigate design decay [29]. On the other hand, developer-related factors are comprised of two factors: *Discussion activity* and *Contribution*. *Discussion activity* represents the developer interaction during the exchange of messages and the contents of messages [14], [30]. *Contribution* focuses on code ownership, based on the number of their previous commits to each file in the software project and their level of contribution to a specific class [16], [31].

Such factors can also be further grouped into more specific sub-factors that can be captured by the computation of metrics. In this work, we rely on meaningful interactions between a set of these sub-factors (see Section III-B) to understand how changes relate to decay. These interactions can happen between sub-factors grouped in the same factor. For example, a change where a new developer (i.e., it is his first change to the software project) modifies a large portion of a single class. This change would be identified as an interaction between *contribution* sub-factors, as that new developer would gain ownership of that code. However, they can also happen between sub-factors grouped in different factors. For example, that same aforementioned change could also, at the same time, interact with a *change outcomes* factor, since it could be a fairly large change. Another example in which this could happen would be a change that contained an extraction refactoring, was extensively discussed by developers, and was done by a new developer. This change would be identified as an interaction between sub-factors grouped in three different factors: *refactoring actions*, *discussion activity*, and *contribu-*

tion. By observing how their interactions are related to changes to the internal quality attributes, we are able to discern the influence these interactions exert over design decay. In order to identify these interactions and their influence, we utilize a technique called Associated Rule Mining, which is introduced in the following section.

C. Association Rule Mining

In this work, we utilized association rule mining [32] to investigate whether and how process- and developer- related factors interact to influence varying levels of design decay. Those interactions can be observed through the associations resulting from this technique, explained as follows.

Association rule mining is a data mining technique that aims to discover associations among a large set of data items [33]. This technique is used to detect patterns of values occurring together in a given dataset [33], [34]. To illustrate association rule mining, consider a set of data items $I = \{i_1, i_2, ..., i_n\}$. Let D be a set of transactions (i.e., dataset), in which each transaction T is a subset of I, i.e., $T \subseteq I$. An association rule is an implication expressed by $A \Rightarrow B$ where $A \subset I$, $B \subset I$ and $A \cap B = \emptyset$. In other words, when A occurs, B tends to occur (the opposite is not necessarily true). More specifically, A and B are disjointed sets of data items, in which A is called the *antecedent* and B is called the *consequent*.

There are three key measures commonly used to filter the relevant association rules: *support*, *confidence* and *lift* [34]. Support determines how frequent the rule is applicable in the transaction set D. It is expressed as $Support(A \Rightarrow B)$, and represents the percentage of transactions that contain both A and B. Confidence, on the other hand, measures the strength of the rules. It is expressed as $Confidence(A \Rightarrow B)$, and represents how frequent B appears in transactions that contain A. Finally, lift is expressed as $Lift(A \Rightarrow B)$, and represents how strongly a rule influences a potentially random occurrence – if a rule's lift is equal to 1, it means that a rule's consequent is independent from its antecedent, thus being a random result. Having a lift value higher than 1 means that the antecedent being fulfilled likely causes the consequent to appear, while lift values below 1 mean that the condition being fulfilled likely causes the inverse, i.e., the consequent to not appear.

In the context of this work, since those associations happen inside each of the transactions, which in the case of this work are the commits, those associations can be seen as representing the interactions between the different factors analyzed.

III. Study Settings

A. Goal and Research Questions

Our study can be described using GQM [35], as follows: **Characterize** interactions between process- and developer-related factors **for the purpose of** identifying whether and how they simultaneously influence design decay, **with respect to** four IQAs, **from the viewpoint of** developers, tool builders, and researchers, **in the context of** software evolution. Thus, we tackle the following research questions (RQs).

RQ_1: *Can process- and developer-related factors be used to indicate decay levels?* This RQ aims to evaluate if it is possible to distinguish between classes with varying levels of decay by looking at process- and developer-related factors. We analyze decay at the class-level, as classes represent the main abstractions of project's object-oriented designs.

RQ_2: *What associations between factors can be inferred from classes that suffered distinct levels of design decay?* This RQ aims to identify associations between either process- and developer-related factors, and changes to the IQAs. Those associations will provide us with information on how the decay process occurred. We also aim to identify if there are significant differences in the associations found for the analyzed levels of decay, and how often those associations can be found. Thus, by answering $\mathbf{RQ}_2$, we can better understand how design decay happens by establishing how responses to different interactions between factors can affect decay.

To answer both RQs, we executed the steps and procedures described in the following section.

B. Study Steps and Procedures

Step 1: Selecting systems for analysis: To select the open-source systems, we used the following criteria based in a prior study [36]. The systems must be: (i) mostly written in the Java programming language, due the availability of robust tools for software measurement; (ii) at least four years old, have more than 1k commits, and 250 closed pull requests[1]; and (iii) in active development, to capture current development practices. These criteria were used in order to avoid known mining perils [37] and to provide enough data to effectively observe and measure our selected sub-factors (Step 4). Table I details each of the seven selected systems. The last system is a closed-source system developed by our industrial partners. We chose to include it as it may hint at possible differences between decay in open and closed-source projects.

TABLE I
General data of the target software systems

System	Domain	# Commits	# Pull Requests	Time span
Fresco	Library	2291	361	4.8 years
Netty	Framework	9322	5086	10 years
OkHttp	HTTP Client	3655	2709	7.8 years
RxJava	Library	5723	3407	7.7 years
Presto	Query Engine	16895	10039	7.5 years
Dubbo	Framework	4105	2462	8 years
S1	Web Application	2548	-	1,5 years

Step 2: Collecting software metrics of internal quality attributes. We utilized Understand [38] and Organic [39] to collect a total of 38 metrics representing different properties of each IQA. For example, LCOM2 and TCC measure the lack of cohesion from different viewpoints. They were selected since they implement already validated metrics for design decay [40]–[44]. We will only present a subset of the metrics, and the size attribute was omitted since it is not used directly to detect design decay. The full list is available in the replication package [20]. From the **cohesion** category, we

[1]These thresholds were selected subjectively, and were based in criteria used in a prior work [14]

have *LCOM2* (Lack of Cohesion in Methods 2), *LCOM3* (Lack of Cohesion in Methods 3) and *TCC* (Tight Class Cohesion). As for **coupling**, we have *CBO* (Coupling Between Object), *FANIN*, and *FANOUT*. In the **inheritance** category, we have *IFANIN*, *NOC* (Number of Children), and *DIT* (Depth of Inheritance Tree). For **complexity**, we have *CC* (Cyclomatic Complexity), *ev(G)* (Essential Complexity), and *MAxNEST* (Maximum Nesting). Additionally, we emphasize that the five IQAs used in this work have been chosen based on a variety of studies that use these metrics to characterize and quantify software quality [25], [40].

Step 3: Tracking system file evolution. As we aim to analyze factors related to decay at a class-level, we collected the evolution history for each file present in a specific version of the system. We considered the most current versions at the time of data collection (detailed in the replication package [20]). To mine the git history, we utilized Pydriller [45] and built a list of modifications for each file, from its introduction in the system to the most recent commit analyzed.

Step 4: Collecting process- and developer-related metrics for tracked files. In this step, we implemented and collected the metrics representing the process- and developer-related sub-factors, introduced in Section II-B, for each of the file changes previously collected. Table II presents the 12 sub-factors used in this work. These metrics were chosen as they represent different dimensions of each factor, and therefore complement each other. Specifically for refactorings, i.e., the implementation of metrics in the *Refactoring action* factor, we have utilized Refactoring Miner 2.0 tool [46], due to its high precision and recall levels (98% and 87%, respectively).

Step 5: Differentiating classes based on how they were affected by design decay. In order to differentiate the files based on how design decay affected their IQAs, we combined the metrics collected in Step 2 with the file history built in Step 3. First, we observed the variation of the metrics pertaining to four IQAs – cohesion, coupling, complexity, and inheritance – taking into account the difference between the creation of the file and its most recent version.

For instance, an increase in the cyclomatic complexity represents a deterioration (i.e., an observable event of decay), whereas a decrease represents an improvement. As seen, this interpretation can vary depending on the semantics of each metric. When needed, we used normalized variations of some metrics (e.g., LCOM2). We also did not consider size-related metrics in this step, since they, in isolation, may not reflect what developers consider as quality [25].

Thus, we ranked the classes in terms of how many[2] metrics indicated a deterioration in an IQA. To this end, we calculated the percentage of metrics from an IQA that have deteriorated (see Equation 1) and computed the mean of this value for those four attributes, resulting in the final score.

$$Attribute\% = \frac{Number\ of\ deteriorated\ metrics}{Total\ number\ of\ metrics\ in\ this\ attribute} \quad (1)$$

[2]Previous studies have associated this diversity in the deterioration of IQAs with design decay [40].

In general, code smells' detection strategies also consider multiple attributes, albeit not necessarily all at the same time or in the same smell [47]. Due to that, we have considered changes to all four attributes equally. It has recently been found that smells often co-occur in the same class, thus affecting the four IQAs [48]. In cases where it did not affect all four attributes, it would at least impact three of them, with inheritance being the exception. We've also executed an additional analysis to ascertain the frequency of changes in a subset of changes in our dataset. Both this analysis, and another additional analysis on the distribution of decay scores between project are described in our replication package [20].

For each system, this score was used to determine quartiles that classify the classes based on how their IQAs decayed. Thus, if a class was below the 25th percentile, it was considered a *slightly-decayed class*. Conversely, if it was above the 75th percentile, it was considered a *largely-decayed class*.

Step 6: Assessing the relationship between individual process and developer-related sub-factors and design decay. To determine if (and which) process- and developer-related metrics are able to distinguish between slightly- and largely decayed classes, we used the *Wilcoxon Rank Sum Test* [49], since our metrics were not normally distributed [50].

Our test used the customary 95% significance level. We needed to adjust the *p-values* of the *Wilcoxon Tests*, since we performed multiple comparisons. For this adjustment, we used the *Bonferroni correction* [50], which controls the *familywise* error rate. We also used the *Cliff's Delta (d)* measure [51] to investigate the magnitude of the difference between the two groups of classes. To interpret the Cliff's Delta (d) effect size, we employed a well-known classification [52] that defines four categories of magnitude: *negligible*, *small*, *medium*, and *large*.

Step 7: Tagging commits to track influential factors on design decay. To identify which factors interacted in each single commit and how this affected the IQAs, we applied a set of tags to each individual change based on related process- and developer-related metrics and on how the IQAs (and their respective metrics) were affected by the change. Table III lists these tags and the conditions under which they are applied. The different states (i.e., +/- or High/Low) of certain tags are mutually exclusive. These tags allow us to identify interacting developer activities that happen prior or along the commit process, as well as how the change affected the IQAs. Thus, an antecedent-consequent relationship spanning the history of the project can be established through association rules.

Step 8: Mining association rules from code changes. Using the changes collected per project (Step 3) and their associated tags (Step 7) as input, we executed the Apriori algorithm. For this step, we used an open-source library called Apyori [53], that implements this algorithm. As we were only interested in identifying meaningful association rules, we chose three criteria as the thresholds for rule creation: **(i)** only the tags related to the process and developer sub-factors should appear as antecedents, otherwise, the model would try to associate tags relating to the IQAs to themselves, creating a situation where they would be dependent and independent

TABLE II
PROCESS AND DEVELOPER METRICS USED IN THIS STUDY

Factor Type	Metric	Description	Rationale
Process-related	**Change outcome**		
	Change Set	Number of files modified alongside a file in a commit.	Large change sets are more likely to be reviewed by developers, due to having high chances of causing decay [9].
	Hunks Count	Number of distinct code segments modified in a file.	A higher number of code segments being modified in a class might indicate complex changes [12].
	Code Churn	Sum of the number of lines added and deleted in a file.	Classes having design problems are more change-prone. [13]
	Refactoring action		
	Move Refatoring Count	Sum of the number of refactoring operations related to motions, i.e, Move Method, Move Class and Move Attribute.	The amount of refactoring actions made on classes may indicate that developers are combating or minimizing design decay [29].
	Extraction Refactoring Count	Sum of the number of refactoring operations related to extractions, i.e, Extract Method, Extract Superclass, Extract Interface, Extract Attribute, Extract Class, Extract Subclass, and Extract Variable.	
	Hierarchical Refactoring Count	Sum of the number of refactoring operations related to hierarchies, i.e, Pull Up Method, Pull Up Attribute, Push Down Method, and Push Down Attribute.	
	Rename Refactoring Count	Sum of the number of refactoring operations related to renames, i.e, Rename Method, and Rename Class.	
Developer-related	**Discussion activity**		
	Number of Associated Pull Requests	Number of pull requests associated with a code change.	Changes with more pull requests associated are more complex, that may lead to design decay.
	Number of Words in discussion	Sum of the all words of each comment inside a Pull Request. Here we applied the preprocessing in the text removing contractions, stop words, punctuation, and replacing numbers.	Discussions with a high number of words are related to more complex changes, that may lead to design decay [14], [30].
	Number of Words per comment	Sum of the all words of each comment inside a Pull Request weighted by the number of comments. Here we applied the preprocessing in the text removing contractions, stop words, punctuation, and replacing numbers.	Discussions with a high weighted number of words are related to more complex changes, that may lead to design decay [14], [30]
	Contribution		
	Contributor Count	Number of distinct developers that have modified a file.	Classes touched by multiple developers are more prone to being degraded [16], [31].
	File Contribution Percentage	Percentage of the lines of a file that is by the same author as the current commit.	Minor contributors tend to degrade more the source code than code changed only by major contributors [16], [31].

TABLE III
COMMIT TAGS USED IN THIS STUDY

Tag	Condition
(+/-) Coupling (+/-) Cohesion (+/-) Complexity (+/-) Size (+/-) Inheritance	Generated based on the variation of the related metrics. Positive changes lead to a positive score, and negative changes to a negative score. The resulting tag is based on whether or not the score sum is above or below zero.
+NumPullRequests +WordsInDiscussion (+/-) WordsPerComment	Applied if there was a variation in the metric's value between two commits. Since the NumPullRequests and WordsInDiscussion metrics are cumulative, however, they cannot have negative values.
(High/Low) ChangeSet (High/Low) HunksCount (High/Low) CodeChurn	These tags were applied using a quartile-based strategy, per project. If a change is in the top 25% (considering all classes) in a metric, it is tagged as High. If it is at the bottom 25%, it is tagged as Low.
+Contributor	This tag indicates the presence of a brand-new contributor in the class.
MajorContributor	This tag indicates that the contributor is responsible for over 5% of the class's code.
MinorContributor	This tag indicates that the contributor is responsible for less than 5% of the class's code.
MoveRefactoring ExtractionRefactoring HierarchicalRefactoring RenameRefactoring	This tag indicates that a refactoring from the specified category was detected in the class's code. This detection is done by RMiner.

variables at the same time; **(ii)** the consequent must only contain one or more tags related to IQA changes, thus, the rules inferred in this step do not consider cases in which there are no modifications to the IQAs; and **(iii)** minimum support = 1% and minimum confidence = 30%. Those thresholds were iteratively defined, initially starting from the default values of the used implementation. We arrived at these final values from the subjective representativeness of the observations made on the results.[3] These observations made sure that those thresholds were adequate to create rules that, while maybe infrequent, are present in multiple of our software projects.

In order to verify if there are associations common to all projects, we also repeated the same aforementioned steps, considering changes in classes from all software projects. We refer to this set of association rules as the *All projects* dataset. Thus, using this procedure, we mined three sets of rules: rules found in slightly-decayed classes, rules found in largely-decayed classes, and rules found in all classes.

[3]The possibility that some relationships were not detected due to the thresholds used is an inherent threat to the validity of studies of this nature.

IV. RESULTS AND DISCUSSION

A. The Relationship Between Sub-Factors and Decay

To answer $\mathbf{RQ}_1$, we needed to first assess the relationship between the metrics and sub-factors, according to Step 6 of our methodology. Table IV shows the results in which each column lists, respectively: the type of factors, the sub-factors split by their types, the classification of the Wilcoxon Test and *d* results for each system, and, finally, a summarization of the results considering the data from all systems. We employ a well-known classification [52] for representing the results, as shown in the table's legend. Cells containing N/A represent cases without enough data to execute the test (further discussed in Section VI). We discuss the metrics associated with factors that have been shown to differentiate between classes with different levels of design decay as follows.

Process-related factors and design decay. From Table IV, we can observe that process-related metrics were the most statistically significant, i.e., able to differentiate between levels of decay. Overall, metrics related to change outcomes were good indicators, having at least a small magnitude in all cases and medium magnitude in 11 out of 24 cases. These results are in line with the findings of previous studies [54] that also investigated process metrics as indicators of design decay. Regarding refactoring actions, while they had worse overall results, two sub-factors stood out: *Extract Refactoring Count* and *Rename Refactoring Count*. Both presented magnitudes higher than medium on 10 out of 16 cases and reached a large magnitude on Presto, OkHttp, and S1.

Finding 1: Sub-factors that represent change outcomes are strong indicators to distinguish the level of design decay. Moreover, some sub-factors related to refactoring actions also stand out as good indicators of design-level change.

TABLE IV
RESULTS OF THE STATISTICAL SIGNIFICANCE (P-VALUE) OF THE WILCOXON RANK SUM TEST AND THE CLIFF'S DELTA (D).

Factor	Metric	dubbo	fresco	netty	okhttp	presto	RxJava	S1	All
Process-related	**Changes outcome**								
	Change Set	(−)**	(−)**	(−)*	(−)*	(−)*	(−)*	(−)***	(−)**
	Hunks Count	(+)**	(+)**	(+)*	(+)*	(+)*	(+)*	(+)***	(+)*
	Code Churn	(−)**	(−)**	(−)*	(−)*	(−)*	(−)*	(−)**	(−)**
	Refactoring action								
	Move Refactoring Count	(+)	(+)	(+)*	(+)*	(+)*	(−)	(+)*	(+)*
	Extraction Refactoring Count	(+)*	(+)**	(+)**	(+)**	(+)***	(+)	(+)	(+)**
	Hierarchical Refactoring Count	(+)	(+)	(+)	(+)	(+)	(+)	(−)	(+)
	Rename Refactoring Count	(+)*	(+)*	(+)**	(+)***	(+)**	(+)	(+)***	(+)**
Developer-related	**Discussion activity**								
	# of Associated Pull Requests	(+)**	(+)	(+)	(−)	(+)*	N/A	N/A	(+)*
	# of Words in Discussion	(+)**	(+)*	(+)**	(+)***	(+)***	N/A	N/A	(+)**
	# of Words per Comment	(+)*	(+)*	(+)*	(+)	(+)*	N/A	N/A	(+)*
	Contribution								
	# of Contributors	(+)***	(+)**	(+)**	(+)***	(+)***	(+)*	(+)***	(+)***
	File Contribution Percentage	(+)	(+)	(−)	(−)*	(−)	(+)	(−)*	(−)

[a] Gray cells represent statistically significant differences, with (*p-value* < 0.05) between the two levels of design decay.
[b] The *Cliff's Delta (d)* effect size is shown as 4 categories of magnitude: *negligible* (no symbol), *small* (*), *medium* (**), and *large* (***).
[c] The positive *d* magnitudes are represented by the (+) symbol, and negative ones are represented by the (−) symbol.

Developer-related factors and design decay. Back to Table IV, we observed that in general, the metrics presented statistical significance on almost all projects. However, the magnitudes were medium and large in only 13 out of 34 cases. More specifically, for metrics that consider discussion factors, we observed that the *Number of Words in Discussion by Class* metric stood out in comparison to the others, which presented a small magnitude in only one system (Fresco) and large in two systems (OkHttp and Presto). This result might indicate that classes containing large discussions are related to more complex changes, which are frequent in largely-decayed classes (as will be discussed in Section IV-B1). At the same time, the *contribution* factor is mostly represented by the *Contributor Count* sub-factor, which presented a large magnitude on five of the eight data samples, while the *File Contribution Percentage* only presented negligible and small magnitudes. In other words, the volume of contributions affecting the classes is a decisive factor in the characterization of largely-decayed classes. Those results lead us to this next finding.

Finding 2: For developer-related factors, the density (i.e., number of words) of discussions and the number of contributors stood out as decisive sub-factors to differentiate between decay levels.

By leveraging these results, we are able to better study the influential associations discussed in the following section. Since some sub-factors are individually related to design decay, they could be used as early indicators that a change should be carefully monitored, before certain interactions that depend on factors that can only be identified later can happen.

B. Influential Associations and Design Decay

As the result of Step 8 of our methodology, we collected many association rules regarding three groups of code changes: changes made to all classes, changes made to slightly-decayed classes, and changes made to largely-decayed classes. Those groups were then composed of eight subsets, seven pertaining to each of the projects analyzed in this work and an eighth that contains an aggregate of all projects. In total, those groups amount to 9131 rules, namely 3245 for all classes, 5096 for slightly-decayed classes, and 790 for largely-decayed classes. Due to the large number of rules, those rules are available in the replication package. However, findings presented in this subsection will provide examples of rules that are related to them. To answer $\mathbf{RQ}_2$, we organized those association rules and manually analyzed the results. To find relevant associations between specific antecedents and consequents, we utilized a variety of visual aids, such as tables and visualizations (e.g., those generated by the arulesvis R package [55], which are available in our replication package [20]).

Due to the large number of association rules, we performed two different analyses. First, we obtained an overview of the entire ruleset by looking at the rules generated with the aggregated data from all projects. Second, we looked at the rules specific to each project. In cases where many rules were present (over 100 rules), we analyzed the top 100, ordered by their *lift* value (see Section II-C). We used the *lift* measure since it favors rules that had a bigger influence on the existence of the consequences. Such strategy also was used in previous studies [55], [56].

We then separated the results between *general associations*, i.e., present in three or more projects, and *specific associations*, i.e., only appeared in one or two projects. By this division, we were able to better understand which associations could be caused by general practices, and which could be caused by project-specific practices or contexts. Finally, we have also performed a manual validation on 96 commits with tags related to some results presented in this section, aiming to understand how our findings correlate to the intent specified by the developers in commit messages and issue discussions. The full details and data for the validation are available in the replication package [20].

1) General Associations Across Projects: We analyzed the set of association rules found in the *All projects* dataset. We emphasize that the following findings only apply to changes in which at least one IQA was affected.[4] We observed that in largely-decayed classes, only a small amount of rules (39) reached our support and confidence thresholds when compared to slightly-decayed classes. Moreover, most of those rules only altered the size IQA. The only other attribute modified was coupling. We conjecture that this low amount of rules is due

[4] 31% of the mined changes did not affect any of the IQAs and another 18% only affected the size attribute.

to how varied the characteristics of changes made to this group of classes were, making it harder for rules to reach our support and confidence thresholds. This variance can be observed in the decay distribution visualization available in our replication package [20]. Nonetheless, for slightly-decayed classes, many rules were generated (847 rules), with great variety of consequents. For rules related to all classes, all of them had consequents pointing towards negative effects.

In the top 100 rules, we observed that most associations skewed towards negative effects, with 68 out of the top 100 causing an increase in code complexity. Additionally, in this subset, the antecedents had little variety, with major contributors (i.e., contributors responsible for over 5% of the changed class's code) being present in 75 of the top 100 rules. While not as predominantly as in the slightly-decayed classes, the same behavior was also seen in the group of rules mined from all classes. It was also observed in the manual validation.

Finding 3: Slightly-decayed classes were frequently negatively affected by changes, even if marginally. However, changes to largely-decayed classes had no specific patterns, with their effects being mostly mixed. In the group containing all classes, all the rules mined pointed towards negative effects.

We observed that changes that have positive effects on the IQAs were affected more variously by the sub-factors. Therefore, those changes tend to not reach our detection thresholds. Negative changes, however, tend to occur in more uniform patterns. The same could not be observed in the case of largely-decayed classes because changes in the group are so varied that neither positive nor negative changes reached our thresholds. We believe this could be a possible reasoning for these findings. This information could be considered in the development of strategies to mitigate design decay. Since largely-decayed classes tend to be more varied, perhaps more targeted approaches could be more adequate when trying to perform repair actions in these classes.

Change outcomes and largely-decayed classes. Regarding the change outcomes, we observed that the presence of complex or large changes, i.e., high amount of hunks or high code churn, often happened in both types of classes, and mostly lead only to changes in the size of the affected code and almost always reduced the size of the changed code. In fact, in largely-decayed classes, we noticed that around 50% of the rules caused a reduction of code size, which were always preceded by complex changes. Furthermore, when these large changes underwent a pull request process, which might imply a code review, they sometimes improved other IQAs, such as reducing code coupling. However, when these large changes are done in a small amount of classes (low change set), they instead tended to increase the size of the changed code.

Table V provides a list with the top 4 rules (ordered by lift) from largely-decayed classes in the All Projects dataset. The second and third rules are the only ones in this group that modified IQAs other from size.

TABLE V
TOP 4 RULES FROM LARGELY-DECAYED CLASSES FROM ALL SYSTEMS.

Antecedents	Consequents	Support	Confidence	Lift
HighHunksCount, LowChangeSet	+Size	1.35%	43.99%	2.38
+NumPullRequests, HighHunksCount, +WordsInDiscussion	-Coupling	1.33%	31.42%	2.32
+NumPullRequests, HighHunksCount	-Coupling	1.77%	30.24%	2.23
MajorContributor, LowChangeSet	+Size	3.44%	38.32%	2.07

2) The Most Common Associations Across Projects: We also performed a cross-project analysis to understand which associations commonly appear in multiple projects. Overall, even when looking at a project-level scope, the results described in Finding 3 continue to appear. Even with the larger sample size of rules, the effect of changes to largely-decayed classes remained mixed, with changes having either a positive or negative effect on code quality based on the project and the performed actions. Slightly-decayed classes also maintain the negative effect observed in Finding 3.

Here, refactoring actions stood out. We observed that, in general, refactorings were one of the two main motivators of changes to IQAs, and rules without refactorings tended to only affect size. Differently from their expected goals, extraction refactorings tended to have a non-positive effect, either only changing the code's size (both positively and negatively), or sometimes even worsening the other IQAs. These negative effects, while not very common, happen more in situations where certain (sub-)factors interact, such as when these refactorings are performed with complex changes (high amount of hunks), or in a few classes (low change set). On the other hand, move refactorings usually had a positive effect on the IQAs. Moreover, we also observed that the interaction between movement and extraction refactorings in the same change is generally more positive than the latter alone. This pattern was observed in the entire dataset, albeit not present in largely-decayed classes (as previously stated, this could be due to our thresholds). It also presented itself more strongly in the rules from all classes than in the slightly-decayed ones. Regarding the intent, we observed in our manual validation that a majority of the changes in our data set had non-refactoring goals, with refactorings only used as a step in a larger process. However, we noticed that when move refactorings are performed with an explicit refactoring goal, they tend to have positive effects. Table VI provides examples of rules used to formulate Findings 4 and 5.

Finding 4: Extraction refactorings often had a non-positive effect on the IQAs. Conversely, move refactorings had a mostly positive effect. Specific interactions between (sub-)factors enables one to better understand those effects. For instance, extraction refactorings being more likely to be negative when applied in complex changes, and being less likely when applied alongside a move refactoring.

Previous code contributions did not reduce the likelihood of extraction refactorings being negative. Another interesting interaction observed, was that even when the developers

TABLE VI
EXAMPLES OF RULES USED TO FORMULATE FINDING 4 AND 5

Finding	Group	Antecedents	Consequents	Support	Confidence	Lift
4	Slightly-Decayed	HighHunksCount, ExtractionRefactoring	-Cohesion, +Coupling	1.04%	32.24%	2.80
4	Slightly-Decayed	HighHunksCount, MajorContributor, ExtractionRefactoring	-Cohesion, +Complexity	1.08%	37.43%	2.73
4	All Classes	ExtractionRefactoring, HighHunksCount, MajorContributor	-Cohesion	1.10%	48.80%	3.42
4	All Classes	ExtractionRefactoring, MajorContributor	-Cohesion, +Complexity	1.01%	31.32%	3.30
5	Slightly-Decayed	+Contributor, MajorContributor, LowChangeSet	-Cohesion, +Size, +Complexity	1.01%	38.28%	2.86
5	Slightly-Decayed	+Contributor, HighHunksCount, MajorContributor	-Cohesion, +Size, +Coupling	1.39%	30.94%	2.83
5	All Classes	LowChangeSet, MajorContributor, +Contributor	+Size, +Complexity, +Coupling	1.29%	47.40%	3.08
5	All Classes	LowCodeChurn, LowChangeSet, MajorContributor, +Contributor	+Size, +Complexity	1.06%	61.12%	2.85

were experienced and major contributors to a class, their extraction refactorings (often in combination with complex changes, a third interacting factor) also frequently caused negative effects on the code. It is noteworthy that these negative effects may be minor in slightly-decayed classes, given that they are in the group that decayed the least. However, these effects might still mean a gradual decay of the affected classes, and are still potential causes for future concern. In the manual validation, changes with both refactoring and non-refactoring goals, respectively, were common with major contributors. The latter mostly had negative effects, while the former was usually positive, yet coupled with a side-effect that could be seen as negative (i.e., an increase of inheritance tree depth through the usage of *Extract Superclass* refactorings).

Level of contribution and slightly-decayed classes. We observed that, for developer-related factors, interactions between certain (sub-)factors frequently appeared. There were often cases in slightly-decayed classes in which new contributors changed over 5% of the code. Those changes usually interacted with additional factors as well, and modified only a few classes. While less frequent, this pattern also presented itself in the rules mined from all classes. Those interactions showed a variety of negative effects, and specifically in slightly-decayed classes, occurring regardless of whether the changes interacted with the associated pull requests (sub-)factor, which may imply that a code review process did not influence this. In the manual validation, the most common changes by new contributors were small functional changes, followed by refactorings. Rarely were large functional changes performed. Functional changes usually had negative effects on code quality, while refactorings had a more mixed effect. Table VI exemplifies rules that resulted in the following finding.

> **Finding 5**: In slightly-decayed classes, a specific interaction, first time contributors that changed a significant (over 5%) portion of a class' code, tended to cause decay – even if these changes might have gone through a pull request process.

3) Specific Associations per Project: For this set of associations, we analyzed each project individually to investigate more context-specific associations. Thus, we found that certain associations behaved differently than the general associations, or even those that were found in other projects. One such case of this was in Fresco's slightly-decayed classes, where changes often had improvements to code quality – when other projects, in general, had mostly negative effects. Another example is related to OkHttp's slightly-decayed classes, which, differently from other projects, had large changes mostly having negative effects, increasing the code's complexity.

There were also interesting observations due to project-specific contexts, providing potential insight into how certain development practices can lead to certain results. For instance, in Dubbo's low decay classes, new contributors often brought negative consequents – if the number of changes was low, the complexity increased; if it was high, the cohesion and coupling worsened. Project S1's largely-decayed classes were the only case in which inheritance metrics were changed by rules with a considerable lift, with these changes being negative, and caused alongside extraction refactorings.

For Netty's largely-decayed classes, we observed that when a specific class appears more frequently on pull requests, the size of that class tended to increase. In Presto's largely-decayed classes, we observed that changes associated with short comments, with a few words (i.e., comments that reduced the mean number of words per comment), always had positive effects, reducing both coupling and size. Finally, in Presto's slightly-decayed classes, even though major contributors to such classes were very present, changes made by them had mostly negative effects on the code.

V. STUDY IMPLICATIONS

The findings of our study lead to implications for researchers, tool builders, and developers, as discussed next.

Researchers can use a variety of interacting (sub-)factors for differentiating design decay: Findings 1 and 2 show that process-related factors and some types of developer-related factors are strong indicators for distinguishing design decay. First, these observations confirm the findings of previous studies [9], [27], [54] on the use of process-related metrics as indicators of complex changes leading to design decay. Second, they confirm findings of recent studies [14], [27] reporting that the number of words in developer discussions could be used to distinguish different levels of design decay.

Our study also advances the body of knowledge by revealing that the number of associated pull requests, either high or low, is unable to distinguish different levels of decay. Initially, one could suspect that as the number of pull requests grows – with more features, bug fixes, and refactorings being requested – design would progressively decay, which would be in line with Lehman's software evolution law [3]. However, that was not observed in the results. This is an important finding, given that the introduction of a change via a pull requests could imply a code review process occurred. Moreover, our study indicates that the high number of refactoring actions grouped by transformation nature, i.e., extraction and move, performed

on a class can be used as indicators of design decay. Finally, we showed how several key interactions between factors could affect design decay. These results can be used by tool builders to create preventive approaches to avoid design decay. For example, these interactions could be used as early indicators that a change or class should be carefully monitored.

Researchers might also be able to explore other types of factors (and their interactions) that may help to further identify and differentiate design decay. It is also important to investigate the actual main root-causes for design decay to occur – these causes certainly go beyond the characteristics of both the changes themselves, as well as the developers that perform them. Our study hints at some of these causes, confirming that certain kinds of changes require more attention from developers as they are highly related to design decay.

Developers lack awareness about refactorings' side effects: Finding 4 shows that, while developers are performing a variety of refactorings, some of them (in most cases, extraction refactorings) are often having a non-positive effect on the code. This happens even when this refactoring interacts with other factors such as it being applied by developers with high ownership of the code. Our observations contradict previous studies on refactorings [22], [41], [57]. Still, a recent study linked extraction refactorings to negative effects [58]. One interesting example found in our manual validation shows an extraction refactorings worsening a class's IQAs.[5] In this change, developers performed a variety of complex extraction refactorings, which slightly improved a few of the affected classes, but caused bloat on another class, by adding a plethora of new extracted methods to its method list.

Conversely, move refactorings had a positive effect on the code quality; almost all the cases change IQAs positively. We have also observed that whenever there is spatial interaction (i.e., same class and same commit) or temporal interaction (i.e., same class and different commits) between extraction refactorings and move refactorings (two process-related sub-factors), that change is less likely to cause decay. Extraction-only refactorings often impacts the target class's interface and how it communicates with its clients [59]. However, this sub-factor alone may not be a consistent indicator of design decay (as we observed here), as previous studies have reported [41].

Thus, developers might need to consider the implications of an extraction refactoring's application, to prevent it from potentially bloating the target class or method – especially in cases of specific interactions, such as the changes being applying alongside other, already complex, changes. More attention should also be given by researchers and tool builders to better support this type of situation. For instance, tools should be adapted to track problems in larger, more complex changes, which are the refactorings that tend to have the most negative effects. Refactoring recommendation systems should also consider there side effects when suggesting changes.

Special attention should be taken when reviewing code by first-time contributors: In Finding 5, we also highlighted another frequent interaction between factors, that when first-time contributors make large changes to a class's code, that change tends to lead to decay, regardless of code review. This might mean that developers should be more cautious while reviewing changes performed by contributors who have not performed previous changes to the target class, especially if this new change affects a large section of the class or a large variety of classes. Where possible, the task distribution to first-time contributors should also be carefully considered. For instance, granting them smaller tasks first, while providing constructive feedback in reviews, so they gain a better understanding of the code in question, to then later grant them tasks that require larger changes to the code.

Reviewers should remain attentive even when reviewing code by experienced contributors: We've also observed that one of the biggest motivators for changes to the IQAs are changes made by major contributors (are authors of at least 5% of a class' code). We conjecture this could happen due to the developer already knowing the class and being more confident to perform changes to the design of a class. The most frequent interactions we encountered involving major contributors were negative. This happened especially if this sub-factor interacted with the sub-factor representing extraction refactorings, which was the strongest interaction observed in terms of lift. Thus, we believe that reviewers should have the same care they would have when reviewing code by a newcomer, even if the author of the changes is already knowledgeable about a class.

VI. Threats to Validity

We discuss threats to the study validity [35] as follows.

Construct and Internal Validity: We analyzed design decay in terms of five IQAs. Thus, our findings might be biased by them, even though they are commonly investigated in other works [40], [41], [60]. The metrics chosen to capture properties of the IQAs might not be appropriate. In fact, metrics alone might not be enough to capture external factors that may influence decay, e.g., developer intentions and design decisions. However, we were particularly interested in the quantified version of decay, represented by these metrics. To mitigate this, we chose a non-random set of metrics that assess different properties of each IQA based on well-known catalogs [24], [61]–[63]. Regarding process- and developer-related metrics, some of them are based on heuristics, e.g., we have assumed that the number of major contributors to a class is the number of developers that contributed at least 5% LOC to this class. Although this is a limitation, we rely on known heuristics to recover this information [17], [27], [31].

As mentioned in Section III, while we have meticulously chosen our thresholds, it is possible they might cause some relationships to not be found by the Apriori algorithm. We also evaluated the Pearson correlation coefficient to measure the correlation between metrics for each factor, and found that some metrics were correlated. However, we tested our models without their presence and found that their absence weakens the models, and thus, maintained them. Concerning the lack of discussion activity metrics on some projects: (i) RxJava classes

[5]https://github.com/square/okhttp/commit/9575377

where decay was found were not inserted on the main branch by pull requests; (ii) since S1 is closed-source, we only had access to the source code. Thus, we lacked discussion data for these projects. Moreover, because of this lack of data, we were not able to apply the analysis of the RQ1, on the Discussion activity factor, for the RxJava and S1. In the future, we plan to improve our project selection to avoid this type of threat.

Conclusion and External Validity: All results were double-checked by two paper authors, to mitigate biases and the misapplication of procedures. Our study focuses on investigating the design decay of Java projects only. Nevertheless, we highlight that Java is one of the most popular programming languages in both industry and academia. Additionally, although we have assessed both open and closed source projects, the number of closed source projects is quite low (only one project). Hence, collecting data from more (in particular closed source) projects and conducting such additional analyses is part of future work.

VII. Related Work

This section describes previous work related to this study.

Empirical Studies on Factors that Affect Design Decay: There are multiple recent studies about factors related to design decay [14], [16], [17], [27], [40], [60]. Many of them only use code smells [14], [17], [27] and software metrics [16], [40], [60] as symptoms for the identification of design decay. Uchôa et al. [17] observed that the majority of code changes performed by developers in code reviews have an invariant impact on the evolution of design decay. They also analyzed the relationship between design decay and the influence of factors related to developer's participation, code review intensity, and reviewing time. They concluded that certain code review practices, such as long discussions and a high rate of reviewers' disagreement, might increase design decay risk. Despite such results, the author does not consider factors related to refactoring actions and contribution. In addition, they do not measure the decay in terms of IQAs.

Barbosa et al. [14] investigated the impact of 11 social metrics related to two social factors on design decay: communication dynamics among developers – who play specific roles; and the discussion content itself. The authors observed that many social metrics could be used to discriminate whether code changes had an impact on design decay. Finally, the authors noticed certain metrics tend to be indicators of design decay only when analyzing both aspects together. Similar to the previous study, the authors do not consider refactoring actions and contribution factors, and neither IQAs.

Capiluppi et al. [16] investigated whether the work of multiple developers and their experiences has an effect on the structural quality metrics. The authors observed that the experience of developers plays a key role: the more inexperienced developers tend to degrade more the source code than the code changed only by experienced developers. They also observed that the decay in structural quality metrics is linked to an increase in further maintenance: when more developers work on the same code, its structure degrades and the number of further commits needed increases. This is even more visible when less experienced developers have worked (or still work) on the code itself. However, the authors do not track how the influential factors are associated with the progression of the decay, and how other developer-related factors, e.g., discussion activities, might distinguish the levels of design decay.

Empirical Studies on the Use of Association Rules: Different studies have used association rules for finding patterns and to extracting knowledge about the different aspects of influencing software development and evolution [56], [64], [65]. For instance, Soares et al. [56] have used association rules to identify characteristics that influence the rejection of pull requests by team members in projects with high acceptance rates. The authors observed that some key factors increase the chances of having internal contributions rejected: (i) physical characteristics and complexity of changes, as well as the location of the modified artifacts; (ii) previous experience with pull requests; and (iii) the project's contribution policy.

Mondal et al. [64] identified co-change patterns to detect hidden dependencies among different parts of the system. More specifically, the author detected a co-change pattern to support the identification of methods logically coupled with other methods. Despite using association rules, their work is centered at the method-level while we focus on the class-level. We also considered changes to coupling as a consequent on the association rules. Finally, Zimmermann et al. [65] proposed an approach that relies on association rule mining to suggest possible future changes. For example, if class A usually co-changes with B, and a commit only changes A, a warning is given suggesting to check whether B should be modified too. Rather than recommend co-changes, our study aims to identify associations that affect design decay on classes.

In summary, our work differs from existing ones as follows: (i) we investigate how process- and developer-related factors can be used to distinguish between varying levels of class-level design decay; (ii) we track how multiple factors simultaneously can affect decay; and (iii) we use association rules to infer and assess relationships between factors and decay.

VIII. Conclusion and Future Work

In this work, we investigated how two groups of factors, developer- and process-related, relate to decay. Our results indicate that several sub-factors can be used to individually distinguish between different decay levels. We also reported several meaningful findings, such as the side-effects of extraction refactorings and negative effects pertaining to changes made by newcomers. The factors involved in these findings interacted in a myriad of ways that changes how they affected design decay. We expect these findings and the other findings and implications presented in the paper to aid developers, researchers, and tool builds in developing measures to avoid or mitigate design decay. As future work, we intend to explore differences between closed and open source systems. Moreover, we intend to explore whether classes introduced with preexisting design problems behave differently from our findings. Finally, we aim to investigate additional factors.

REFERENCES

[1] D. G. Feitelson, E. Frachtenberg, and K. L. Beck, "Development and deployment at Facebook," *IEEE Internet Computing*, vol. 17, no. 4, pp. 8–17, 2013.

[2] J. Tsay, L. Dabbish, and J. Herbsleb, "Let's talk about it: evaluating contributions through discussion in GitHub," in *22nd FSE*, 2014, pp. 144–154.

[3] M. M. Lehman, "Programs, life cycles, and laws of software evolution," *Proceedings of the IEEE*, vol. 68, no. 9, pp. 1060–1076, 1980.

[4] D. L. Parnas, "Software aging," in *16th ICSE*, 1994, pp. 279–287.

[5] D. M. Le, C. Carrillo, R. Capilla, and N. Medvidovic, "Relating architectural decay and sustainability of software systems," in *13th WICSA*, 2016.

[6] G. Bavota, A. De Lucia, M. Di Penta, R. Oliveto, and F. Palomba, "An experimental investigation on the innate relationship between quality and refactoring," *J. Syst. Softw. (JSS)*, vol. 107, pp. 1–14, 2015.

[7] 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 *40th ICSE*, 2018, pp. 921–931.

[8] F. Falcão, C. Barbosa, B. Fonseca, A. Garcia, M. Ribeiro, and R. Gheyi, "On Relating Technical, Social Factors, and the Introduction of Bugs," in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2020, pp. 378–388.

[9] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi, "A large-scale empirical study of just-in-time quality assurance," *IEEE Transactions on Software Engineering*, vol. 39, no. 6, pp. 757–773, 2012.

[10] F. Ebert, F. Castor, N. Novielli, and A. Serebrenik, "Confusion in code reviews: Reasons, impacts, and coping strategies," in *2019 IEEE 26th international conference on software analysis, evolution and reengineering (SANER)*. IEEE, 2019, pp. 49–60.

[11] S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan, "The impact of code review coverage and code review participation on software quality: A case study of the qt, vtk, and itk projects," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, 2014, pp. 192–201.

[12] Y. Fan, X. Xia, D. Lo, and S. Li, "Early prediction of merged code changes to prioritize reviewing tasks," *Emp. Softw. Eng. (ESE)*, vol. 23, no. 6, pp. 3346–3393, 2018.

[13] J. M. Bieman, A. A. Andrews, and H. J. Yang, "Understanding change-proneness in OO software through visualization," in *11th IWPC*, 2003, pp. 44–53.

[14] C. Barbosa, A. Uchôa, F. Falcao, D. Coutinho, H. Brito, G. Amaral, A. Garcia, B. Fonseca, M. Ribeiro, V. Soares, and L. Sousa, "Revealing the Social Aspects of Design Decay: A Retrospective Study of Pull Requests," in *34th SBES*, 2020, pp. 1–10.

[15] M. C. Ohlsson, A. Von Mayrhauser, B. McGuire, and C. Wohlin, "Code decay analysis of legacy software through successive releases," in *1999 IEEE Aerospace Conference. Proceedings*, vol. 5, 1999, pp. 69–81.

[16] A. Capiluppi, N. Ajienka, and S. Counsell, "The effect of multiple developers on structural attributes: A Study based on Java software," *J. Syst. Softw. (JSS)*, p. 110593, 2020.

[17] A. Uchôa, C. Barbosa, W. Oizumi, P. Blenilio, R. Lima, A. Garcia, and C. Bezerra, "How Does Modern Code Review Impact Software Design Degradation? An In-depth Empirical Study," in *36th ICSME*, 2020, pp. 1–12.

[18] I. Ahmed, U. A. Mannan, R. Gopinath, and C. Jensen, "An empirical study of design degradation: How software projects get worse over time," in *9th ESEM*, 2015, pp. 1–10.

[19] G. Piatetsky-Shapiro, "Discovery, analysis, and presentation of strong rules," *Knowledge discovery in databases*, pp. 229–238, 1991.

[20] "Replication Package," https://opus-research.github.io/decay_factors_replication/, 2021.

[21] Z. Li, P. Avgeriou, and P. Liang, "A systematic mapping study on technical debt and its management," *J. Syst. Softw. (JSS)*, vol. 101, pp. 193–220, 2015.

[22] M. Fowler, *Refactoring*. Addison-Wesley Professional, 1999.

[23] R. Malhotra, *Empirical research in software engineering: concepts, analysis, and applications*. CRC Press, 2016.

[24] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Trans. Softw. Eng. (TSE)*, vol. 20, no. 6, pp. 476–493, 1994.

[25] E. A. AlOmar, M. W. Mkaouer, A. Ouni, and M. Kessentini, "On the impact of refactoring on the relationship between quality attributes and design metrics," in *19th ESEM*, 2019, pp. 1–11.

[26] M.-A. Storey, A. Zagalsky, F. Figueira Filho, L. Singer, and D. M. German, "How social and communication channels shape and challenge a participatory culture in software development," *IEEE Trans. Softw. Eng. (TSE)*, vol. 43, no. 2, pp. 185–204, 2016.

[27] A. Uchôa, C. Barbosa, D. Coutinho, W. Oizumi, W. K. Assunçao, S. R. Vergilio, J. A. Pereira, A. Oliveira, and A. Garcia, "Predicting Design Impactful Changes in Modern Code Review: A Large-Scale Empirical Study," in *18th MSR*. IEEE, 2021, pp. 1–12.

[28] V. Soares, A. Oliveira, P. Farah, A. Bibiano, D. Coutinho, A. Garcia, S. Vergilio, M. Schots, D. Oliveira, and A. Uchôa, "On the Relation between Complexity, Explicitness, Effectiveness of Refactorings and Non-Functional Concerns," in *34th SBES*, 2020, pp. 788–797.

[29] I. Macia, R. Arcoverde, A. Garcia, C. Chavez, and A. von Staa, "On the relevance of code anomalies for identifying architecture degradation symptoms," in *16th CSMR*, 2012, pp. 277–286.

[30] N. Bettenburg and A. E. Hassan, "Studying the impact of social interactions on software quality," *Emp. Softw. Eng. (ESE)*, vol. 18, no. 2, pp. 375–431, 2013.

[31] C. Bird, N. Nagappan, B. Murphy, H. Gall, and P. Devanbu, "Don't touch my code! Examining the effects of ownership on software quality," in *19th FSE*, 2011, pp. 4–14.

[32] R. Agrawal, R. Srikant *et al.*, "Fast algorithms for mining association rules," in *20th VLDB*, vol. 1215, 1994, pp. 487–499.

[33] J. Han, J. Pei, and M. Kamber, *Data mining: concepts and techniques*. Elsevier, 2011.

[34] R. Agrawal, T. Imielinski, and A. Swami, "Mining associations between sets of items in large databases," in *ACM SIGMOD ICMD*, 1993, pp. 207–216.

[35] C. Wohlin, P. Runeson, M. Höst, M. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering*, 1st ed. Springer Science & Business Media, 2012.

[36] J. Tsay, L. Dabbish, and J. Herbsleb, "Influence of social and technical factors for evaluating contribution in GitHub," in *36th ICSE*, 2014, pp. 356–366.

[37] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian, "An in-depth study of the promises and perils of mining GitHub," *Empirical Software Engineering*, vol. 21, no. 5, pp. 2035–2071, 2016.

[38] I. Scientific Toolworks, "Understand Tool," 2021, available at: https://scitools.com/.

[39] W. Oizumi, L. Sousa, A. Oliveira, A. Garcia, A. Agbachi, R. Oliveira, and C. Lucena, "On the identification of design problems in stinky code: experiences and tool support," *JBCS*, vol. 24, no. 1, 2018.

[40] A. Chávez, I. Ferreira, E. Fernandes, D. Cedrim, and A. Garcia, "How does refactoring affect internal quality attributes? A multi-project study," in *31st SBES*, 2017, pp. 74–83.

[41] A. C. Bibiano, V. Soares, D. Coutinho, E. Fernandes, J. Correia, K. Santos, A. Oliveira, A. Garcia, R. Gheyi, B. Fonseca *et al.*, "How Does Incomplete Composite Refactoring Affect Internal Quality Attributes," in *28th ICPC*, 2020.

[42] E. Fernandes, A. Chávez, A. Garcia, I. Ferreira, D. Cedrim, L. Sousa, and W. Oizumi, "Refactoring effect on internal quality attributes: What haven't they told you yet?" *Information and Software Technology*, vol. 126, p. 106347, 2020.

[43] 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.

[44] A. C. Bibiano, E. Fernandes, D. Oliveira, A. Garcia, M. Kalinowski, B. Fonseca, R. Oliveira, A. Oliveira, and D. Cedrim, "A quantitative study on characteristics and effect of batch refactoring on code smells," in *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 2019, pp. 1–11.

[45] D. Spadini, M. Aniche, and A. Bacchelli, "PyDriller: Python framework for mining software repositories," in *26th ESEC/FSE*, 2018, pp. 908–911.

[46] N. Tsantalis, A. Ketkar, and D. Dig, "RefactoringMiner 2.0," *IEEE Trans. Softw. Eng. (TSE)*, pp. 1–1, 2020.

[47] R. Marinescu, "Detection strategies: Metrics-based rules for detecting design flaws," in *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.* IEEE, 2004, pp. 350–359.

[48] J. Martins, C. Bezerra, A. Uchôa, and A. Garcia, "How do Code Smell Co-occurrences Removal Impact Internal Quality Attributes? A Developers' Perspective," in *35th SBES*, 2021.

[49] E. Whitley and J. Ball, "Statistics review 6: Nonparametric methods," *Critical care*, vol. 6, no. 6, p. 509, 2002.

[50] J. H. McDonald, *Handbook of biological statistics*. sparky house publishing Baltimore, MD, 2009, vol. 2.

[51] R. J. Grissom and J. J. Kim, *Effect sizes for research: A broad practical approach*. Lawrence Erlbaum Associates Publishers, 2005.

[52] J. Romano, J. Kromrey, J. Coraggio, J. Skowronek, and L. Devine, "Exploring methods for evaluating group differences on the NSSE and other surveys: Are the t-test and Cohen'sd indices the most appropriate choices," in *Annual Meeting of the Southern Association for Institutional Research*, 2006.

[53] "ymoch/apyori: A simple implementation of Apriori algorithm by Python." https://github.com/ymoch/apyori.

[54] F. Palomba, G. Bavota, M. Di Penta, F. Fasano, R. Oliveto, and A. De Lucia, "On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation," *Emp. Softw. Eng. (ESE)*, vol. 23, no. 3, pp. 1188–1221, 2018.

[55] M. Hahsler and S. Chelluboina, "Visualizing association rules: Introduction to the R-extension package arulesViz," *R*, pp. 223–238, 2011.

[56] D. M. Soares, M. L. de Lima Júnior, L. Murta, and A. Plastino, "Rejection factors of pull requests filed by core team developers in software projects with high acceptance rates," in *14th ICMLA*, 2015, pp. 960–965.

[57] W. F. Opdyke, "Refactoring object-oriented frameworks," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 1992.

[58] A. C. Bibiano, W. Assunção, D. Coutinho, K. Santos, V. Soares, R. Gheyi, A. Garcia, B. Fonseca, M. Ribeiro, D. Oliveira, C. Barbosa, J. L. Marques, and A. Oliveira, "Look Ahead! Revealing Complete Composite Refactorings and their Smelliness Effects," in *37th ICSME*, 2021.

[59] M. Paixão, A. Uchôa, A. C. Bibiano, D. Oliveira, A. Garcia, J. Krinke, and E. Arvonio, "Behind the intents: An in-depth empirical study on software refactoring in modern code review," in *Proceedings of the 17th International Conference on Mining Software Repositories*, 2020, pp. 125–136.

[60] J. Martins, C. Bezerra, A. Uchôa, and A. Garcia, "Are Code Smell Co-occurrences Harmful to Internal Quality Attributes? A Mixed-Method Study," in *34th SBES*, 2020, pp. 1–10.

[61] T. J. McCabe, "A complexity measure," *IEEE Trans. Softw. Eng. (TSE)*, no. 4, pp. 308–320, 1976.

[62] S. Henry and D. Kafura, "Software structure metrics based on information flow," *IEEE Trans. Softw. Eng. (TSE)*, no. 5, pp. 510–518, 1981.

[63] M. Lorenz and J. Kidd, *Object-oriented software metrics: a practical guide*. Prentice-Hall, Inc., 1994.

[64] M. Mondal, C. K. Roy, and K. A. Schneider, "Insight into a method co-change pattern to identify highly coupled methods: An empirical study," in *21st ICPC*, 2013, pp. 103–112.

[65] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl, "Mining version histories to guide software changes," *IEEE Trans. Softw. Eng. (TSE)*, vol. 31, no. 6, pp. 429–445, 2005.