Predicting Design Impactful Changes in Modern Code Review: A Large-Scale Empirical Study

*Informatics Department, Pontifical Catholic University of Rio de Janeiro (PUC-Rio), Brazil
†Computer Science Department, Federal University of Paraná (UFPR), Brazil

Abstract—Companies have adopted modern code review as a key technique for continuously monitoring and improving the quality of software changes. One of the main motivations for this is the early detection of design impactful changes, to prevent that design-degrading ones prevail after each code review. Even though design degradation symptoms often lead to changes' rejections, practices of modern code review alone are actually not sufficient to avoid or mitigate design decay. Software design degrades whenever one or more symptoms of poor structural decisions, usually represented by smells, end up being introduced by a change. Design degradation may be related to both technical and social aspects in collaborative code reviews. Unfortunately, there is no study that investigates if code review stakeholders, e.g., reviewers, could benefit from approaches to distinguish and predict design impactful changes with technical and/or social aspects. By analyzing 57,498 reviewed code changes from seven open-source systems, we report an investigation on prediction of design impactful changes in modern code review. We evaluated the use of six ML algorithms to predict design impactful changes. We also extracted and assessed 41 different features based on both social and technical aspects. Our results show that RandomForest and Gradient Boosting are the best algorithms. We also observed that the use of technical features results in more precise predictions. However, the use of social features alone, which are available even before the code review starts (e.g., for team managers or change assigners), also leads to highly-accurate prediction. Therefore social and/or technical prediction models can be used to support further design inspection of suspicious changes early in a code review process. Finally, we provide an enriched dataset that allows researchers to investigate the context behind design impactful changes during the code review process.

Index Terms—design changes; code review; machine learning

I. INTRODUCTION

Modern code review is a practice that has been widely adopted by major companies [1], [2]. It is typically a lightweight, informal, asynchronous, and tool-assisted practice aimed at monitoring, detecting and removing issues that were introduced during development tasks [2]. Supported by platforms such as Gerrit and GitHub, the code review process is initiated by one developer referred to as the owner, which modifies the original codebase and submits a new code change to be reviewed by other developers – the so-called reviewers.

A key concern of all stakeholders involved in a code review, including code owners, reviewers, and team managers, is to become aware of ongoing changes impacting the design [3]–[5]. The underlying motivation is to monitor and inspect those design impactful changes so that stakeholders can anticipate, find, and remove signs of design degradation before the end of a code review. Otherwise, those design harmful changes can become prevalent after the code review [6], [7]. If design impactful changes are not discriminated and brought to attention early, it will increase the likelihood of those changes finding their way into the software system for several reasons. These include reviewers deviating their effort to other quality checks in later review stages or even starting to focus only on the main purpose of the issue being resolved. Moreover, design-related changes become harder to revert towards the end of the review as many other inter-related modifications were already realized as the review progresses. In fact, these reasons explain why many design impactful changes get unnoticed as nearly 40% of pull request rejections are related to design issues [8].

Early identification of impactful changes that degrade the software design is important during code review [3], [9], [10]. Code reviewers are expected to inspect ongoing changes and provide prompt feedback to code owners in the form of comments. In turn, the code owner should fix and forward the new version of the code for inspection. Such a procedure is repeated in multiple iterations, which are called revisions. This sequence of revisions ends up with either the acceptance or rejection of the change into the codebase [11], [12].

Despite its importance, recent studies found these modern code review practices are far from being sufficient to prevent design-degrading changes [3], [13]. Design degradation occurs whenever a change introduces poor structural decisions, i.e., design smells [14]–[17]. Tools for detecting design smells tend to be inefficient when a change is still at its early stages. For instance, the full addition of a new feature can be complex and its realization needs many revisions to be accomplished. Moreover, other types of change can be complex as well. In fact, several changes in software projects are fully realized only after many revisions in a single review.

If these harmful changes are not reversed early, i.e., before a code review is ended, rework will be necessary after the changes of the last merged revision. Further changes with time-consuming refactorings will have to be applied later. Given the costs of design refactorings, they are unlikely to be applied and smells will be compounded over time, thereby accelerating the design degradation [3], [4], [18].

Due to the importance of the early identification of design relevant changes during the review process, stakeholders must use all available information during the code review process. In code review platforms, stakeholders have either technical
or social information at their disposal to be used as additional information, both before or after each change. Social information are often available as soon as code review starts. Social information includes: number of prior code changes submitted by the code owner, and centrality of the code owner on the collaboration graph [19]–[22]. Technical information are available after changes and revisions done during the review. Examples of technical information include: number of times a file has been changed and types of change [19], [23]–[25].

The advantage of using technical and social metrics to characterize and predict failures have widely been studied [26]–[29]. However, their use to discriminate and predict design impactful changes is rarely studied [3], [13]. In fact, such metrics can act as indicators of design impactfulness of ongoing changes along the code review process. Hence, to the best of our knowledge, there is no study on which types of metrics can be used as effective features in Machine Learning (ML) algorithms to accurately predict design impactful changes.

This paper presents results of a large-scale empirical study that investigates whether and how technical and social metrics can be used to predict design impactful changes. To this end, we analyzed more than 50k code reviews of seven real-world systems from two large open source communities. We mined and examined if a comprehensive suite of technical and social metrics can discriminate design (un)impactful changes. Then, we explored the use of these metrics, as features for six interpretable ML algorithms, which tend to offer an effective prediction for different tasks and contexts, e.g., [29], [30]. Finally, we evaluated the predictive power of the selected features and algorithms to assist developers to automatically determine whether a code change is impactful.

Our key findings and contributions are: (1) both social and technical metrics are able to distinguish design (un)impactful changes; (2) the use of technical features results in more accurate predictions, when compared to the social ones; (3) features related to the code change, commit message, and file history dimensions are effective for differentiating (un)impactful changes; (4) Random Forest and Gradient Boosting have shown to be the most accurate in predicting design impactful changes; and (5) an enriched dataset and replication package that allows researchers to investigate the context and motivations behind design impactful changes during code reviews.

II. MOTIVATING EXAMPLE

Next we show the importance of considering design impact and using social and technical aspects during code review. To this end, we rely on two scenarios of the jgit system in which code review is conducted on the Gerrit platform.

Scenario A. Let us consider the review 3345 [31], composed of seven revisions, in which two developers performed a major change to “Replace TinyProtobuf with Google Protocol Buffers”. After the last revision, 12,215 insertions and 2,404 deletions were performed in 58 files. Additionally, 104 design smells were introduced, leading to structural degradation related to the lack of abstraction (46), encapsulation (17), and modularity (42). Interestingly, design impact was not mentioned during revisions of the reviews of this major change.

In other words, the replacement of a third-party component was conducted without an explicit concern with possible side effects that the change could introduce into the system.

Scenario B. Now consider the review 825 [32], which aims to “Implement a Dircache checkout (needed for merge)”. This review had four reviewers and 18 revisions. After the first revision (R1), we observed a inclusion of three smells: two Unutilized Abstraction and one Insufficient Modularization. Such smells were perceived by a developer, according to the following comment: “One problem I faced here: we do have an abstraction to access the WorkTree when walking (reading) on it.” Additionally, during the revisions (R3 to R10), we observed removals and reintroductions of the smells Unutilized Abstraction and Insufficient Modularization, characterizing a high fluctuation of design degradation during revisions. In other words, despite the developers identify degradation symptoms, they still are not able to see all the ramifications and impacts of their changes along with revisions.

Both scenarios illustrate the need for mechanisms to aid developers on identifying and preventing design impactful changes. Such mechanisms can rely on the large, diverse, rich information from social and technical aspects of the system and stakeholders in the code review. In Scenario A, when developers are unaware of the design impact of their change, a mechanism could have supported reviewers by automatically analyzing the discussions that took place on previous revisions and the components involved in the changes to predict the impact of changes in the current revision on the system design. In Scenario B, a mechanism could have analyzed the previous behavior of the code owner, and reviewers could better understand which changes are harming the source code.

In this context, one could argue: why not using existing tools, such as Designite [33], to identify design degradation? Despite useful, such tools are limited. Besides the reasons already mentioned in Section I, they rely only on static analysis in which detection strategies do not adapt per revision, not exploring the history of changes and multiple sources of information. They ignore the variation of technical and social aspects inherent to the code review process [3], [13], [20].

In summary, the motivations for our work are: (i) a real-world need for mechanisms to aid stakeholders in identifying impactful design changes during code review, (ii) availability of large and rich sources of information that can be used to make stakeholders aware of their changes’ impact, and (iii) limitation of existing tools on using historical and dynamic information to support stakeholders during code review.

III. STUDY SETTINGS

A. Research Questions

Our study is guided by four research questions (RQs).

RQ1: Are design impactful changes significantly different from unimpactful ones in terms of social and technical metrics? – Social and technical aspects may be avoiding or amplifying design degradation. To capture such aspects, we
used a set of metrics detailed in Section III-E. \textbf{RQ}_1 aims at investigating which metrics are able to distinguish between design impactful changes and unimpactful ones.

\textbf{RQ}_2: \textit{What is the performance of ML algorithms to predict design impactful and unimpactful changes?} – Once we show empirical evidence that distinguishes impactful and unimpactful changes, \textbf{RQ}_2 aims at investigating the use of supervised ML techniques to assist developers in automatically make their decisions. In practice, some prediction algorithms perform better than others, depending on the task. Thus, we compare the performance of six interpretable ML algorithms: Logistic Regression, Naive Bayes, SVM, Decision Tree, Random Forest, and Gradient Boosting. We chose these algorithms since they provide an intuitive and easy to explain model \cite{34, 35}.

\textbf{RQ}_3: \textit{How effective are the social and technical features as a proxy to the design impactfulness changes?} – \textbf{RQ}_3 aims at evaluating and comparing the performance of both kinds of features. To this end, we applied the ML algorithm using three feature sets: a set using only social features, a set using only technical ones, and a set using technical and social features together. By answering \textbf{RQ}_3, we will be able to identify which kind of features are the best predictor, as well as the effectiveness of combining social and technical features. Furthermore, we also evaluated the effectiveness of a feature selection step for the three sets.

\textbf{RQ}_4: \textit{What features are the best indicators of impactful design changes?} – \textbf{RQ}_4 aims at understating which features are considered the most relevant by the models. Such knowledge is essential because, in practice, a model should be as simple as and require as little data as possible. By answering \textbf{RQ}_4, we will be able to provide insights to practitioners and researchers as to what factors best indicate design impactful changes.

\subsection*{B. Code Review Data}

To answer our \textbf{RQ}_s, we need not only information of the source code to distinguish design impactful changes, but also to analyze every code revision submitted along the code review process and investigate technical and social information related to those revisions. Thus, instead of mining code review data ourselves, we used the data provided by the Code Review Open Platform (CROP) \cite{36}, an open-source dataset that links code review data to software changes. All systems in CROP employ Gerrit as their code review tool. Hence, by using CROP, we have access to a rich dataset of code changes. To this end, given a certain system, CROP provides a complete copy of the entire codebase for each revision and its respective parent, which represents the system’s codebase at the time of review. In other words, unlike the Git repository of a system, which contains only accepted revisions (i.e., the changes in the final revisions in a review), the CROP stores all revisions.

In our study, we adopt all Java systems included in the CROP dataset: four systems from the Eclipse community and three systems from the Couchbase community, as presented in Table I. For sake of completeness, we remove the reviews whose status is “Open” since they may not have been assigned to reviewers, and the set of reviewers may still change.

\begin{table}[h]
\centering
\caption{Software systems investigated in this study}
\begin{tabular}{|l|l|l|l|}
\hline
Community & System & \# of Reviews & \# of Revision \& Time span \& \hline
Eclipse & jgit & 5,290 & 3,872 to 11/17 & \hline
& egit & 5,220 & 12,841 to 11/17 & \hline
& platformui & 4,527 & 11,414 to 11/17 & \hline
& functions & 4,270 & 10/13 to 11/17 & \hline
Couchbase & java-client & 909 & 12/11 to 11/17 & \hline
& jenkins & 828 & 11/13 to 11/17 & \hline
& spammemcached & 536 & 11/13 to 11/17 & \hline
\hline
\end{tabular}
\end{table}

\subsection*{C. Detection of Degradation Symptoms within Code Reviews}

We investigated two categories of degradation symptoms, which are fine-grained (FG) and coarse-grained (CG) smells \cite{37}. Although we do not focused on architectural smells, we empirically observed they follow similar trends in a complementary analysis. \textit{FG smells} are indicators of structural degradation in the scope of methods and code blocks \cite{37}. For instance, the Long Method is a FG smell that occurs in methods that contain too many lines of code. \textit{CG smells} are symptoms that may indicate structural degradation related to object-oriented principles, e.g., abstraction, encapsulation, and modularity \cite{37, 38}. An example of CG smell is Insufficient Modularization \cite{37}. This symptom occurs in classes that are large and complex due to the accumulation of responsibilities. Such categories encapsulate a set of symptoms that are more perceived and used by developers in practice to identify and refactor source code locations degraded \cite{15, 39, 41}.

For automatically detecting symptoms of such categories, we used a state-of-the-practice tool called DesignJava \cite{33}, which detected a total of 27 degradation symptoms types: 17 CG smells, and 10 FG smells. Hence, for each system, we identified these symptoms by considering each submitted revision that has undergone the code review process. Thus, we used CROP to access the versions of the system before and after the revision took place. Next, we detected the degradation symptoms in each version before and after revision. By following this methodology, we are guarantee that the introduced degradation symptoms between each version were solely introduced by the code changes in the revision. Table II lists the 27 symptoms types investigated in our study. We provide all descriptions, detection strategies, and thresholds for each type of symptom in the replication package \cite{42}.

\begin{table}[h]
\centering
\caption{Degradation symptoms investigated in this study}
\begin{tabular}{|l|l|l|}
\hline
Coarse-grained Smells &  &  \\
Inappropriate Abstraction, Multifaceted Abstraction, Unfinished Abstraction, Unnecessary Abstraction, Deficient Encapsulation, Unexploited Encapsulation, Broken Modularization, Insufficient Modularization, Hub Like Modularization, Cyclic Dependend Modularization, Rebellious Hierarchy, Wide Hierarchy, Deep Hierarchy, Multipath Hierarchy, Cyclic Hierarchy, Missing Hierarchy, Broken Hierarchy. \\

Fine-grained Smells &  &  \\
\hline
\end{tabular}
\end{table}

\subsection*{D. Identification of Design Impactful Change Instances}

We identified design impactful change instances in two steps: (i) identification of smelliness files by considering each revision of a code change, where each revision was compared to its parent, i.e., the codebase’s version before any revision; and (ii) the computation of design degradation indicators.

To illustrate the first step, let $R_s = \{r_1, r_2, ..., r_s\}$ be the set of submitted revisions for a given review $s$. For each revision in $R_s$ (i.e., $\forall r_i$, $r_i \in R_s$), we use the CROP to retrieve the system’s versions before and after $r_i$. Next, we check the
presence of degradation symptoms (both CG and FG smells) in the files of $r_i$. The two file sets (before and after) might not be exactly the same, due to files created and deleted during the review process. Since the before version of each revision is its parent, we guarantee that the introduced degradation symptoms between each version were solely introduced by the code changes in $r_i$, avoiding the collateral effects of the rebase [43]. The output of this step is, for each revision in $R$, the version of the files impacted before and after the revision.

In the second step, we rely on an existing grounded theory [13] that explains that developers tend to consider multiple degradation characteristics in terms of density and diversity of symptoms. In addition, the use of density and diversity of symptoms for detecting design degradation is supported by other studies [18], [40]. Such studies show that degraded code elements tend to be affected by higher diversity and density of symptoms when compared to other code elements. However, before selecting diversity and density of symptoms as metrics, we compared the two lists of smells (by density and diversity) before and after revisions. As a result, we observed a small average variation ($<$1 type of smell/revision). Thus, computing the degradation increase/crease with density imposes only a minor threat. Therefore, we take into account only the density, as a metric to measure the level of design degradation.

Therefore, for each selected system, we computed this characteristic in the context of each symptom category (CG and FG smells), for all the collected revisions. Density was computed for each version before and after revision, as the sum of the number of symptom instances in the set of smelliness source code files. The computation of density represents the differences in density of FG and CG smells.

In summary, a positive difference in the density of symptoms indicates an increase in the degradation as a result of the review, harming the design. Similarly, a negative difference indicates a reduction of the degradation as a result of the revision. Finally, no variation indicates that there has been no structural design change. We consider that a design change is impactful when an increase in the degradation as a result of submitted changes. Table III shows the number of revisions identified as design (un)impactful changes for each system and symptom category.

### TABLE III

<table>
<thead>
<tr>
<th>Project</th>
<th>Coarse-grained Smells</th>
<th>Fine-grained Smells</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Impacted</td>
<td>Unimpacted</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Impacted</td>
</tr>
<tr>
<td>jar-client</td>
<td>78 (22%)</td>
<td>47 (14%)</td>
</tr>
<tr>
<td>jvm-core</td>
<td>1,480 (25%)</td>
<td>53 (15%)</td>
</tr>
<tr>
<td>platforms</td>
<td>423 (11%)</td>
<td>44 (13%)</td>
</tr>
<tr>
<td>engines</td>
<td>2,431 (15%)</td>
<td>32 (15%)</td>
</tr>
<tr>
<td>Total</td>
<td>14,824 (22%)</td>
<td>297 (15%)</td>
</tr>
</tbody>
</table>

E. Features for Design Impactful Change Prediction

We extracted a set of features able to capture both technical and social aspects of the changes involved in each revision of a code review. Each feature corresponds to a metric. We detail each feature and its description per dimension on Table IV.

### TABLE IV

<table>
<thead>
<tr>
<th>Dimension</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>NFA</td>
<td>Number of added files in this code change</td>
</tr>
<tr>
<td></td>
<td>NF</td>
<td>Number of deleted files in this code change</td>
</tr>
<tr>
<td></td>
<td>NLDF</td>
<td>Number of added lines to and removed in this code change</td>
</tr>
<tr>
<td></td>
<td>NLDF</td>
<td>Number of deleted files in this code change</td>
</tr>
<tr>
<td>Diffusion</td>
<td>NC</td>
<td>Number of changed files in the code change</td>
</tr>
<tr>
<td></td>
<td>NC</td>
<td>Number of modified directories in the code change</td>
</tr>
<tr>
<td></td>
<td>NC</td>
<td>Distribution of modified code across files in this code change</td>
</tr>
<tr>
<td></td>
<td>NLANG</td>
<td>Distribution of modified code across files in this code change</td>
</tr>
<tr>
<td></td>
<td>NLANG</td>
<td>Number of programming languages used in this code change</td>
</tr>
<tr>
<td>Complexity</td>
<td>NS</td>
<td>Number of added code segments in this code change</td>
</tr>
<tr>
<td></td>
<td>NS</td>
<td>Number of deleted code segments in this code change</td>
</tr>
<tr>
<td></td>
<td>NS</td>
<td>Number of updated code segments in this code change</td>
</tr>
<tr>
<td>File history</td>
<td>FD</td>
<td>Number of developers who changed files in this code change</td>
</tr>
<tr>
<td></td>
<td>MDL</td>
<td>Number of words in description of this code change</td>
</tr>
<tr>
<td>Textual</td>
<td>BUG</td>
<td>Whether description of this code change contains word &quot;bug&quot;</td>
</tr>
<tr>
<td></td>
<td>FEATURE</td>
<td>Whether description of this code change contains word &quot;feature&quot;</td>
</tr>
<tr>
<td></td>
<td>IMPROVE</td>
<td>Whether description of this code change contains word &quot;improve&quot;</td>
</tr>
<tr>
<td></td>
<td>DOCS</td>
<td>Whether description of this code change contains word &quot;document&quot;</td>
</tr>
<tr>
<td></td>
<td>RELATE</td>
<td>Whether description of this code change contains word &quot;relate&quot;</td>
</tr>
</tbody>
</table>

From the technical perspective, we extracted 21 features related to source code, modification history of the files, and the textual description of the change. Moreover, these features were categorized into five dimensions: (i) Size consists of features related to source in their smallest granularity. Prior studies have found that large patches may need more effort to review [47]; (ii) Diffusion comprehends the features about changes distributed on two or more files (e.g., number of changed files). Prior studies also found that revisions, where their changes scatter across a large number of files or directories, may need more effort to review [19], [25]. Thus, we expected that the diffusion of a change could influence the likelihood of the change being impactful; (iii) Complexity comprehends the features on the complexity of a change. A code change with more code segments modified is likely more complex and requires more effort and time to be reviewed [48]; (iv) File history is composed of features related to the history of the files. The number of prior changes to a file can be a good indicator to detect degraded files. Moreover, files that are previously touched by more developers are more likely to introduce degradation symptoms [25]; and (v) Textual consists of features that capture textual characteristics of the commit message. Previous studies [49], [50] found that the description length of a patch is related to its likelihood of receiving poor comments. Additionally, the commit message may contain more information about a code change that may...
help reviewers comprehend the change more easily.

From the social perspective, we extracted 20 features that characterize the developer’s experience, collaboration network, and participation in discussions. Moreover, these features were grouped into three dimensions: (i) **Developer’s experience** comprise the features related to the previous experience of the code change owner. Previous studies [19–21] found that developer experience is essential information for predicting design issues. Such studies claim that if a developer often submits changes in recent times prior to the change, they will be more familiar with the recent developments of the system, and thus the code change may be fewer design issues; (ii) **Discussion activity** comprise the features of communication between developers and reviewers. In fact, classes having degraded symptoms can create more discussion among the reviewers [3, 13]. As well as, discussions with a high number of comments around code changes would find possible design symptoms, improving or maintaining the quality; and (iii) **Collaboration networks** consists of features of social networks. Previous studies [20, 22, 48] found that collaboration factors (i.e., level of participation within the system) could influence code review outcomes. For this reason, we constructed a network based on the collaboration of owners and reviewers to use the features proposed by [22].

**F. Development of the Impactfulness Prediction Models**

We experimented with six different (binary classification) supervised ML algorithms: Logistic Regression, Naive Bayes, SVM, Decision Tree, Random Forest, and Gradient Boosting.

**Training and testing the models.** We trained and tested the models as follows. Firstly, we collected the design (un)impactful changes instances for a given system. We merged them into a single dataset, where design impactful changes instances are marked with a true value, and design unimpactful changes instances are marked with a false value. Secondly, before training the models, we dealt with imbalanced data, a common issue with software engineering data [51]. In our case, the number of design impactful changes instances varies; i.e., the design impactful changes instances might be greater than or smaller than the number of unimpactful ones. To that end, we relied on the undersampling algorithm, which randomly selects instances of the oversampled class. Thirdly, we scaled all the features to a [0, 1] range to speed up the learning process of the algorithms [52]. Fourthly, we used the grid search to tune the hyperparameters of each model using five folds. Grid search is an exhaustive search that examines all of the combinations of a specified set of candidate settings to find the best combination [53].

Finally, to train the model, we employed a 10-fold cross-validation strategy using the hyperparameters established by the search [54]. This strategy randomly partitions the dataset into 10 folds of equal size, in which each fold has the same proportion of the various criticality classes. A single fold is then used as a test set, while the remaining ones are employed for training the model, i.e., they are independent of each other.

**Performance evaluation.** We evaluated the performance of each generated model, by analyzing confusion matrices, obtained from the testing strategy described above, and reporting the values of well-known measures [55]. Precision is the percentage of detected code changes that are actually impactful \( Pr = \frac{TP}{TP+FP} \). Recall is the percentage of correctly predicted impactful design change relative to all of the changes that are actually unimpactful \( Re = \frac{TP}{TP+FN} \). The \( F1 \)-score \((F1)\) is the harmonic mean of precision and recall. Additionally, to mitigate the limitation of choosing a fixed threshold when calculating precision and recall, we compute the Area Under the ROC Curve (AUC) values. AUC is computed by measuring the area under the curve that plots the true positive rate against the false positive rate, while varying the threshold that is used to determine if a design change is predicted as impactful or unimpactful.

**Replication package.** All data described previously, features used for training and testing the ML algorithms, hyperparameters analyzed, generated ML models, as well as the confusion matrix and statistical analysis are available in [42].

**IV. Results and Discussions**

**A. Design impactful changes vs. unimpactful ones**

To answer **RQ1**, we used the Wilcoxon Rank Sum Test [50] and the Cliff’s Delta \((d)\) measure [57] to verify which metrics are able to discriminate between impactful and unimpactful design changes. To this end, we explore each metric described in Section [II-E]. The Cliff’s Delta \((d)\) measure [57] shows how strong is the difference between design impactful changes and unimpactful ones in terms of the analyzed metrics. Since we are performing multiple comparisons, we need to adjust the \(p\)-values to consider the increased chance of rejecting the null hypothesis simply due to random chance. To do so, we apply the widely used Bonferroni correction [58], which controls the familywise error rate. For this method, we consider that each system is a family, which means that we perform the correction in the \(p\)-values of the features at the system level. Table [V] shows the results obtained for coarse- (CG) and fine-grained (FG) smells, where the 1st column contains the type of metric while the 2nd column shows the evaluated metric. The 3rd to 16th columns show the Wilcoxon Test and \(d\) results for each system, each group of two columns represents the results for CG and FG smells, respectively. Statistical significant differences \((p\text{-value} < 0.05)\) are highlighted as gray cells. To interpret the Cliff’s Delta \((d)\) effect size, we employ a well-known classification [59], that defines four categories of magnitude, which are represented in Table [V] negligible (without symbol), small (*), medium (**), and large (***)

The positive \( d \) magnitudes are represented by the (+) symbol and the negative ones are represented by the (−) symbol.

We emphasize that **RQ2** does not consider the FEAT, IMPR, IMPR, DOC, and REFC metrics since the Wilcoxon Test and the Cliff’s Delta measure are not suitable for boolean metrics.

**Correlation between inline comments and impactful changes.** Among all metrics in the dimension of discussion, we highlight those related to inline comments, inline comments
### Table V
Statistical Significance of the Wilcoxon Rank Sum Test and the Cliff’s Delta (d) for Coarse and Fine-Grained Smells

<table>
<thead>
<tr>
<th>Dimension</th>
<th>Metric</th>
<th>java-client</th>
<th>java-core</th>
<th>jvm-core</th>
<th>platformed</th>
<th>jgit</th>
<th>jbg</th>
<th>all</th>
</tr>
</thead>
<tbody>
<tr>
<td>Developer’s Experience</td>
<td>NIC</td>
<td>(+)</td>
<td>(+)</td>
<td>(+)</td>
<td>(+)</td>
<td>(+)</td>
<td>(+)</td>
<td>(+)</td>
</tr>
<tr>
<td></td>
<td>NRC</td>
<td>(+)</td>
<td>(+)</td>
<td>(+)</td>
<td>(NFA)</td>
<td>(+)</td>
<td>(+)</td>
<td>(+)</td>
</tr>
<tr>
<td></td>
<td>NGC</td>
<td>(+)</td>
<td>(+)</td>
<td>(+)</td>
<td>(NFA)</td>
<td>(+)</td>
<td>(+)</td>
<td>(+)</td>
</tr>
<tr>
<td></td>
<td>NR</td>
<td>(+)</td>
<td>(+)</td>
<td>(+)</td>
<td>(NFA)</td>
<td>(+)</td>
<td>(+)</td>
<td>(+)</td>
</tr>
<tr>
<td></td>
<td>MR</td>
<td>(+)</td>
<td>(+)</td>
<td>(+)</td>
<td>(NFA)</td>
<td>(+)</td>
<td>(+)</td>
<td>(+)</td>
</tr>
<tr>
<td></td>
<td>DMR</td>
<td>(+)</td>
<td>(+)</td>
<td>(+)</td>
<td>(NFA)</td>
<td>(+)</td>
<td>(+)</td>
<td>(+)</td>
</tr>
<tr>
<td>Discussion Activity</td>
<td>NIC</td>
<td>(+)</td>
<td>(+)</td>
<td>(+)</td>
<td>(NFA)</td>
<td>(+)</td>
<td>(+)</td>
<td>(+)</td>
</tr>
<tr>
<td></td>
<td>NRC</td>
<td>(+)</td>
<td>(+)</td>
<td>(+)</td>
<td>(NFA)</td>
<td>(+)</td>
<td>(+)</td>
<td>(+)</td>
</tr>
<tr>
<td></td>
<td>NGC</td>
<td>(+)</td>
<td>(+)</td>
<td>(+)</td>
<td>(NFA)</td>
<td>(+)</td>
<td>(+)</td>
<td>(+)</td>
</tr>
<tr>
<td></td>
<td>NWGC</td>
<td>(+)</td>
<td>(+)</td>
<td>(+)</td>
<td>(NFA)</td>
<td>(+)</td>
<td>(+)</td>
<td>(+)</td>
</tr>
<tr>
<td></td>
<td>PWGC</td>
<td>(+)</td>
<td>(+)</td>
<td>(+)</td>
<td>(NFA)</td>
<td>(+)</td>
<td>(+)</td>
<td>(+)</td>
</tr>
<tr>
<td></td>
<td>DL</td>
<td>(+)</td>
<td>(+)</td>
<td>(+)</td>
<td>(NFA)</td>
<td>(+)</td>
<td>(+)</td>
<td>(+)</td>
</tr>
<tr>
<td>Collaboration Networks</td>
<td>SD</td>
<td>(+)</td>
<td>(+)</td>
<td>(+)</td>
<td>(NFA)</td>
<td>(+)</td>
<td>(+)</td>
<td>(+)</td>
</tr>
<tr>
<td></td>
<td>SCLOS</td>
<td>(+)</td>
<td>(+)</td>
<td>(+)</td>
<td>(NFA)</td>
<td>(+)</td>
<td>(+)</td>
<td>(+)</td>
</tr>
<tr>
<td></td>
<td>SE</td>
<td>(+)</td>
<td>(+)</td>
<td>(+)</td>
<td>(NFA)</td>
<td>(+)</td>
<td>(+)</td>
<td>(+)</td>
</tr>
<tr>
<td></td>
<td>SCLUST</td>
<td>(+)</td>
<td>(+)</td>
<td>(+)</td>
<td>(NFA)</td>
<td>(+)</td>
<td>(+)</td>
<td>(+)</td>
</tr>
<tr>
<td></td>
<td>SKC</td>
<td>(+)</td>
<td>(+)</td>
<td>(+)</td>
<td>(NFA)</td>
<td>(+)</td>
<td>(+)</td>
<td>(+)</td>
</tr>
</tbody>
</table>

### Finding 1: The usefulness of the metrics from the discussion activity, developer’s experience, and collaboration networks dimensions to differentiate design impactful changes is limited. However, the number of directory changes presents promising results.

### Code metrics as strong indicators of impactful changes.

The most relevant metrics for distinguishing between impactful and unimpactful changes were the ones related to code. Their results were not statistically significant in only three cases for FG smells. Moreover, with the exception of the files deleted (NFD) metric in the java-client system, all code metrics presented positive magnitudes. Among all code metrics, we highlight the lines added (NAL), changed lines (CHURN), and segments added (NSA) metrics, which presented statistically significant results with large magnitude in all cases, for both types of smells. If we restrict our analysis to CG smells, the files added (NFA), segments deleted (NSD), and modify entropy (ME) metrics also stand out. The highlighted metrics are closely related to the size (NAL, CHURN, NFA) and complexity (NSA, NSD, ME) of the changes. Thus, such metrics can be used by reviewers to decide when a change requires more attention to the design impact.

### Textual dimension.
The message length (ML) metric showed statistically significant results with small or medium positive magnitudes in most cases. We conjecture that this is because changes with detailed descriptions tend to be more complex, leading to a higher impact on design.

### Finding 2: Code, textual, and file history dimensions contain metrics that are relevant to differentiate impactful changes. The most relevant ones can be combined to predict changes that require more attention to design.

### B. ML performance for predicting design impactful changes.

We address RQ₂, by reporting and comparing different models after the 10 stratified cross-fold executions. We apply stratified sampling in all the cross-fold executions to make sure both training and test datasets contain the same amount...
of design (un)impactful changes instances. Table VI shows the precision (Pr), recall (Re), F1-score (F1), and AUC values of each ML algorithm for each target system and smell levels. The row "all" represents the generated model, when training and testing in the entire dataset and using all features.

The performance of ML algorithms for predicting design impactful changes. Table VI shows that for coarse-grained smells, the average of precision values across models ranges between 0.74 and 0.95, while the recall values range between 0.56 and 0.95. Similarly, the F1-score values range between 0.57 and 0.95, while the AUC values range between 0.65 and 0.95. A similar observation applies when we consider fine-grained smells, in which the average of precision, recall, F1-score, and AUC values ranges between 0.83 and 0.96, 0.50 and 0.96, 0.59, and 0.96, and 0.69, and 0.96, respectively.

To assess statistical differences between the models, we applied Friedman non-parametric test [60] with Nemenyi’s post hoc multiple pairwise comparison (p-value ≤ 0.05). We observed that both Random Forest and Gradient Boosting outperformed the other ML models at coarse-grained (CG) and fine-grained (FG) levels, with significant difference according to the statistical test. Furthermore, both Random Forest and Gradient Boosting present a similar performance, i.e., without a statistical difference, considering the CG level, with a similar average of 0.95 for both F1 and AUC. However, with a slight difference of 1% for the average of precision and recall values. A similar observation applies when at FG level. Moreover, to ascertain if the level of accuracy is adequate, we compared our model performance results with other approaches both for ML-based smell detection [61], [62] and code review analysis [63]. On average, our two best models achieved similar or better performance results than those previous works.

Finding 3: Random Forest and Gradient Boosting are the most accurate in predicting design impactful changes within code reviews. Both algorithms achieve an average of F1-score of 0.95 and 0.96, for predicting design impactful changes at a CG and FG smell levels, namely (i) social features only, (ii) technical features only, and (iii) social + technical features together. Additionally, and given the great number of features, we investigate the difference of the impactfulness prediction between to use or not a step for feature ranking and selection.

Instead of simply removing the highly-correlated features by following a filtering method, we decided to apply a wrapper method to feature selection. We applied feature ranking with recursive feature elimination and cross-validated selection of the best number of features for our data. For feature ranking and selection we used the RFECV function available in the scikit-learn’s feature selection package [64]. We run RFECV using 5-fold cross-validation and SVC linear as the estimator. After the cross-validation process, and RFECV recommendations, three new sets, namely feature selection sets, were generated, a set with 19, 9, and 40 features respectively for the social, technical, and social + technical dimensions. Similarly to RQ2, we check the statistical difference of the results using Friedman test [60] and Nemenyi’s post hoc multiple pairwise comparison, with a confidence level of 95%.

The effectiveness of social and technical features as predictors to design impactful changes. To evaluate the effectiveness of social and technical features, we rely on the best ML algorithms, i.e., Random Forest and Gradient Boosting, based on the RQ2 results. Table VII shows the mean values of precision (Pr), recall (Re), F1-score (F1), and AUC for both algorithms, grouped by dimension and feature set.

C. The role of social and technical features as predictors

We address RQ3, by investigating the performance of different feature sets as a proxy to predict design impactful changes at CG and FG smell levels, namely (i) social features only, (ii) technical features only, and (iii) social + technical features together.

We observed that social features for impactful changes, at both levels of granularity, reached mean values of Pr, Re, F1, and AUC.
and AUC around 0.8. A similar performance is reached when feature selection was used. Technical features reach mean values of Pr, Re, F1, and AUC around 0.96 and 0.94 for impactful changes at, respectively, coarse- and fine-grained levels. But we also observed that the use of features selection at the coarse-grained level leads to a performance reduction, decreasing values of Pr, F1, and AUC. The Friedman test did not point a significant difference when using feature selection compared to all features, as well as between the two levels of granularity, in the same set of features. In summary, we can conclude that both sets of features are good predictors. They reach values of Pr, Re, F1, and AUC ≥ 0.79 at both levels.

Finding 4: Both social and technical features are effective as a proxy to detect impactful changes. In this way, code review stakeholders may choose the set of features to be used according to their interests and roles.

Social features vs. technical features vs. social + technical features. The set of technical features are better predictors than the set of social features in terms of Pr, Re, F1, and AUC for coarse-grained and fine-grained smells, and both algorithms, with or without feature selection. We observe a significant statistical difference between both sets. Nevertheless, there is no statistical difference, in terms of performance, between technical and social + technical sets. The combination of both kinds of features leads to results that are statistically equivalent to the best results, obtained by the technical feature set. This happens even for the set of social and technical features together when the double number of features is used.

Finding 5: The use of technical features leads to the best results. Moreover, such kind of features can be used in combination with social features without reducing the performance of the ML algorithms.

D. The best features for predicting design impactful changes

We address RQ₁, by reporting how often each feature appears among the top-1 and top-5 most important features of all the generated models without feature selection. To better understand the importance of social and technical features for predicting design impactful changes per symptom category, we generate different sets of rankings. To this end, we vary the configuration of each model according to different feature sets, i.e., social features only, technical features only, and both together, and reported the five most frequent features per ranking and feature sets. We used scikit-learn’s implementations to extract the feature importance of the SVM (linear), Decision Tree, Random Forest, Gradient Boosting, and Logistic Regression models [64]. We highlight that some models, e.g., SVM, might return the importance of a feature as a negative number, indicating that the feature is important for the prediction of the design unimpactful changes, in our case. Thus, we consider such a feature also important to the models, and thus, we build the ranking using the absolute value of feature importance returned by the models.

![The best features using social and technical features in isolation](image)

Table VIII lists the ranking of the best features across ML algorithms and systems grouped by smell category, i.e., coarse-grained (CG) and fine-grained (FG) smells, and feature set, i.e., technical feature only, and social feature only. For each ranking, we show the corresponding dimension (Dim.) and the frequency (Freq.) in which each feature appears by feature set.

<table>
<thead>
<tr>
<th>Ranking</th>
<th>Technical Features Only</th>
<th>Social Features Only</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dim.</td>
<td>Feature Name</td>
<td>Freq.</td>
</tr>
<tr>
<td>Social</td>
<td># Files Added</td>
<td>11</td>
</tr>
<tr>
<td>Size</td>
<td># Changed Lines</td>
<td>6</td>
</tr>
<tr>
<td>Complexity</td>
<td># Segments Added</td>
<td>2</td>
</tr>
<tr>
<td>Defect</td>
<td># Changed Files</td>
<td>3</td>
</tr>
<tr>
<td>Size</td>
<td># Lines Added</td>
<td>29</td>
</tr>
<tr>
<td>Social</td>
<td># Files Added</td>
<td>28</td>
</tr>
<tr>
<td>Size</td>
<td># Changed Lines</td>
<td>23</td>
</tr>
<tr>
<td>Complexity</td>
<td># Segments Added</td>
<td>20</td>
</tr>
<tr>
<td>Diffusion</td>
<td>Modify Entropy</td>
<td>13</td>
</tr>
</tbody>
</table>

By considering FG smells with technical features only, we observed that features quantifying the size of the code changes such as, lines added (NAL), and changed lines (CHURN) frequently appear in the top-1 ranking, followed by features that quantify complexity (segments added (NSA)), history information about files modified (file modifications (FM)), and diffusion of a change (changed files (NCF)). A similar observation applies in the top-5 ranking, but with the presence of one textual feature, message length (ML), that appears 17 times. Interestingly, when we compare both types of smells, we observed that NAL, CHURN, NSA, NCF also appear as important for both symptom categories in the top-1 and top-5 rankings, except for the features lines added (NFA), and modify entropy (ME) that only appear for CG smells.

On the other hand, when we consider social features only: for FG smells, we observed that the feature directory changes (NDC), that quantifies the developer’s experience in terms of the number of prior code changes submitted by the owner that contains at least one directory affected by the current submitted code change, is the most important social feature in the top-1 ranking appearing 28 times across models. Next, less frequently, features that quantify inline comments (NIC) made by reviewers on the code change submitted by the owner, and features that quantify discussion activities among the owner and reviewers, such as, # words in general comments (NGC), % words in general comments (PWGC), and discussion length (DL) also appear as important features.

Interestingly, by looking at the top-5 ranking, we observed that 4 out of 7 (57%) features that quantify different aspects of the developer’s experience are considered as important across models, followed by the NIC that appears 11 times. By comparing both categories, we observed that NDC keeps its...
importance in the top-1 and top-5 rankings. Finally, two social features appear as important for CG smells, social closeness (SCLOS), and recent merged ratio (RMR). These observations also reinforce that senior developers should be allocated as reviewers to keep the quality of code review high [3]. [7] and promote the knowledge transfer along revisions [2]. [65].

**Finding 6**: In isolation, social features that quantify the developer’s experience and discussion activities are indeed considered important across models in both smells categories. Also, as expected, technical features that quantify size, complexity, and diffusion of the code changes are considered important across all models.

The best features using social and technical features together for predicting design impactful changes. Table IX also lists the ranking of the best features across ML algorithms grouped by CG and FG smells. However, in this table, we consider social and technical features as a single feature set. We also show the corresponding dimension (Dim.) and the frequency (Freq.) in which each feature appears.

**TABLE IX**

<table>
<thead>
<tr>
<th>The Ranking of the Most Important Features Across ML Algorithms Using Social and Technical Features Together</th>
</tr>
</thead>
<tbody>
<tr>
<td>Top-1 CG Smells</td>
</tr>
<tr>
<td>Top-1 FG Smells</td>
</tr>
<tr>
<td>Top-5 CG Smells</td>
</tr>
<tr>
<td>Top-5 FG Smells</td>
</tr>
</tbody>
</table>

For FG smells with social and technical features aggregated, we observe that the same technical features listed in Table VIII appears in the top-1 and top-5 rankings, i.e., when we consider the technical features in isolation, the same features also appear as the most important features for FG smells, except for the message length (ML), that only appears when the technical features are in isolation, and the number of languages (NLANG) feature, which only appears in the rankings with the combined features. Both of those features appear 17 times each in their respective top-5 rankings. We also observed that when the social and technical features are considered together, social features only appear in the top-5 ranking.

This result indicates that technical features, when combined with social ones, tend to be the most important features across models, especially in the top-1 ranking. However, social features that appear in the top-5 ranking are majority features that quantify the developer’s experience. This result, reinforces our previous finding, on the importance of the developer’s experience to predicting fine-grained design impactful changes.

We also observed similar behavior for CG smells, in which the same set of technical features when we considered the technical features in isolation, also appear as the most important features across models. The exception to this is the feature modified entropy (ME), which appears 1 time in the top-1 ranking in Table IX. Finally, similar to FG smells the social features only appear in the top-5 ranking, with the prevalence of features that quantify the developer’s experience.

**Finding 7**: Technical features tend to be considered the most important features across models when compared with social features. However, social features that quantify the developer’s experience are also considered important across models in both symptom categories.

Some features never appear in any of the rankings. Considering the full rankings of features importance, we observed that for both FG and CG smells, when technical and social features are considered in isolation, the technical features that capture textual characteristics of the commit message such as, has bug (BUG), has feature (FEAT), has improvement (IMPR), has document (DOC), and has refactor (REFC) do not appear even when we consider the top-5 ranking. A similar observation applies when we consider all social and technical features together, except for the feature FEAT. On the social features, for both smell types the features directory merged ratio (DMR) and social betweenness (SB) never appear in the top-1 and top-5 feature importance ranking. Additionally, the feature percentage words in general comments (PWGC) also does not appear in the rankings for CG smells.

On the other hand, when we combine all social and technical features together, eight social features do not appear among the rankings for both symptoms category. These features, four are from the collaboration network dimension, social closeness (SCLOS), social clustering (SCLUST), social eigenvector, and social betweenness (SB). Another group of three features that do not appear are from the discussion activity dimension: inline comment (NIC), general comments (NGC), and discussion length (DL). Finally, reviews (NR) from the developer’s experience dimension also does not appear. Specifically for FG smells, we observed that merged ratio (MR), recent merged ratio (RMR), directory merged ratio (DMR), social k coreness (SKC), # words in inline comments (NWIC), # words in general comments (NWGC), and % words in general comments (PWGC) also never appear in the rankings.

**Finding 8**: Features from the textual dimension consistently did not appear in any of the rankings when technical features are used in isolation. Conversely, when both types of features are used in conjunction, features related to collaboration networks and discussion activity tend to not appear for both symptom categories.

### V. Threats to Validity

**Construct and Internal Validity.** The precision and recall of degradation symptoms detection may have influenced our results. We mitigate this threat by selecting a detection tool, successfully used in recent studies on design degradation [3]. [13]. [33]. [66], and previous work [67] indicated a precision of 96% and a recall of 99%. Moreover, there is evidence
that developers tend to refactor code elements with a high density and diversity of the selected symptoms \[18\]. Although, there are more instances of design unimpactful changes in our dataset we mitigate this imbalanced dataset by removing instances from the over-represented class through random under-sampling strategy. Moreover, the selection of the ML algorithms and their parameter settings may affect the accuracy and influence interpretability. We mitigate this threat by selecting the most widely used interpretable ML algorithms and, for a fair comparison, we searched for their best parameters through an extensive hyperparameter search via grid search, and 10-fold cross-validation strategy. Finally, we selected and computed a wide number of social and technical features that helped us measure different social and technical dimensions of the changes involved in each code revision, e.g., developer experience, and file history. The rationales for using metrics are supported by prior studies, e.g., \[22\], \[47\], \[48\], \[68\]. We wrote scripts to automate compute these metrics, and all implementations were validated by three paper authors.

**Conclusion and External Validity.** About the descriptive analysis, four paper authors contributed to the analysis of design (un)impactful changes. For the statistical analysis, we rely on the Wilcoxon Rank Sum Test, Bonferroni correction, and Cliff’s Delta \(d\) measure to verify which metrics are able to discriminate between (un)impactful design changes. We also computed largely used performance measures \[55\], precision, recall, F1-score, and AUC score. Furthermore, we rely on the Friedman non-parametric test, and Nemenyi-tests to avoid subjective opinion regarding the best accurate model and the role of social and technical features as predictors. About the feature importance, our ML pipeline does not currently have a way to get the feature importance of Naive Bayes, but we have no reason to believe the lack of this model affected the conclusion of RQ4. We are aware that there is an inherent threat to transferring our findings to other systems, i.e., our results may be subject to the degradation characteristics of these specific systems. However, we focused on seven systems to be able to make robust and reliable statements about if learning approaches can be used in such settings.

VI. RELATED WORK

Previous studies applied ML to help in the code review activity \[63\]. In fact, ML has been used intensively for smell characterization and detection \[61\], \[62\], \[69\], \[70\]. Recent reviews on this subject show that distinct algorithms and smells have been addressed. They differ in the set of metrics, regarding product and process \[71\], \[72\]. There are also works that investigate how smells affect program comprehension and bug proneness, and how code becomes smelly. Sae-Lim et al. \[73\], \[74\] proposes an ML approach that uses information from the current release to predict modules that will decay in the next release, i.e., modules that will become smelly. The prediction model is built based on source code metrics, but it can be improved by using developers’ context that is expressed by a set of modules to be modified by the developers. We have not found ML-based studies concerned with how changes performed by developers in the code review context can impact on the degradation symptoms like smells.

Works that study the impact of code review on the overall software quality do not usually apply ML. Some works investigate the relationship between code review and bug introduction \[75\], \[76\], and between code review and design quality \[6\], \[7\]. Morales et al. \[6\] studied the impact of code review coverage (proportion of changes code reviewed) and reviewer involvement on smells. They observe that high coverage and review participation can reduce the occurrence of smells. Pascarella et al. \[7\] investigated the influence of code reviews in the code smell severity. They conclude that higher values of code review quality dimensions (activity and participation) are related to a decrease in code smell severity.

In a previous work \[3\], we observed that the majority of code reviews are unimpactful on the evolution of design degradation and that certain code review practices can both combat or even accelerate design degradation. Our findings showed that design discussions may not be enough for avoiding degradation and that certain practices, such as long discussions and a high rate of reviewers’ disagreement, might increase the design degradation risk. We also observed a wide design degradation fluctuation during the revisions of each review, meaning that developers were both introducing and removing degradation symptoms along with a single code review. This fluctuation often resulted in the amplification of design degradation at the end of the review. This fact shows the importance of reviewers to know a priori the impact of their revisions. Our study is the first initiative to contribute in this direction. We propose an ML approach that uses social and technical features. This allows reviewers to distinguish design (un)impactful changes and prioritize revisions to avoid design degradation.

VII. FINAL REMARKS

In summary, our main findings pointed out that: (i) both social and technical features are able to distinguish between design impactful changes and unimpactful ones; (ii) Random Forest and Gradient Boosting are the most accurate algorithms; and (iii) both social and technical features are effective as a proxy to predict impactful changes. Our findings also provide insights for new studies and be the basis for tool builders creating a new generation of tools to aid developers in automatically predicting design impactful changes during code reviews. We also show that: (i) existing detection tools should be more interactive, in a stepwise manner, to anticipate, find, and remove signs of degradation before finishing a review; (ii) In addition to only technical features, the combination with social features is promising for predicting design (un)impactful changes; and (iii) qualitative studies should be performed to explain other aspects governing the decision-making process discriminating and predicting design impactful changes.

**Acknowledgment.** This work was partially funded by CNPq (434969/2018-4, 312149/2016-6, 140919/2017-1, 141285/2019-2, 131020/2019-6, 104254/2019-0), CAPES/Procad (175956), CAPES/Proex, and FAPE RJ (200773/2019, 010002285/2019, PDR-10 202073/2020).


G. Bavota and B. Russo, “Four eyes are better than two: On the impact of code reviews on software quality,” in 31st ICSME. IEEE, Sep. 2015, pp. 81–90.