An Empirical Study of Design Discussions in Code Review

Farida El Zanaty
McGill University
Montréal, Canada
farida.elzanaty@mail.mcgill.ca

Toshiki Hirao
Nara Institute of Science and Technology
Nara, Japan
hirao.toshiki.ho7@is.naist.jp

Shane McIntosh
McGill University
Montréal, Canada
shane.mcintosh@mcgill.ca

Akinori Ihara
Wakayama University
Wakayama, Japan
ihara@sys.wakayama-u.ac.jp

Kenichi Matsumoto
Nara Institute of Science and Technology
Nara, Japan
matumoto@is.naist.jp

ABSTRACT

Background: Code review is a well-established software quality practice where developers critique each others' changes. A shift towards automated detection of low-level issues (e.g., integration with linters) has, in theory, freed reviewers up to focus on higher level issues, such as software design. Yet in practice, little is known about the extent to which design is discussed during code review.
Aim: To bridge this gap, in this paper, we set out to study the frequency and nature of design discussions in code reviews.
Method: We perform an empirical study on the code reviews of the OpenStack Nova (provisioning management) and Neutron (networking abstraction) projects. We manually classify 2,817 review comments from a randomly selected sample of 220 code reviews. We then train and evaluate classifiers to automatically label review comments as design related or not. Finally, we apply the classifiers to a larger sample of 2,506,308 review comments to study the characteristics of reviews that include design discussions.
Results: Our manual analysis indicates that (1) design discussions are still quite rare, with only 9% and 14% of Nova and Neutron review comments being related to software design, respectively; and (2) design feedback is often constructive, with 73% of the design-related comments also providing suggestions to address the concerns. Furthermore, our classifiers achieve a precision of 59%–66% and a recall of 70%–78%, outperforming baselines like zeroR by 43 percentage points in terms of F1-score. Finally, code changes that have design-related feedback have a statistically significantly increased rate of abandonment (Pearson χ^2 test, DF=1, $p < 0.001$).
Conclusion: Design-related discussion during code review is still rare. Since design discussion is a primary motivation for conducting code review, more may need to be done to encourage such discussions among contributors.

ESEM '18, October 11–12, 2018, Oulu, Finland

ACM ISBN 978-1-4503-5823-1/18/10…$15.00
https://doi.org/10.1145/3239235.3239525

KEYWORDS

Code Review, Software Design, Mining Software Repositories

ACM Reference Format:
Farida El Zanaty, Toshiki Hirao, Shane McIntosh, Akinori Ihara, and Kenichi Matsumoto. 2018. An Empirical Study of Design Discussions in Code Review. In *ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM) (ESEM '18), October 11–12, 2018, Oulu, Finland.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3239235.3239525

1 INTRODUCTION

Code reviewing is a broadly adopted practice where one developer, the author, makes a change to the code, and other developers, the reviewers, critique the premise, content, and structure of the change [14]. Code review practices are realized in a variety of ways, such as formal code inspection [35], informal walkthroughs [46], lightweight tool-based reviews [3], or checklist-based reviews [36].

Since code review is expensive in terms of human effort [5], recent code review systems are shifting towards automated detection of low-level issues. For example, the Review Board project[1] incorporates linting and static code analysis results directly in the code review interface [5]. This trend towards automation of menial tasks, has, in theory, freed reviewers up to focus on higher level issues, such as design.

Software design plays a central role in the success of the produced software [42]. As a result, design decisions are essential during development, since design decisions are linked to the maintainability and extensibility of a codebase [10, 16, 22]. From an internal perspective, better design has been linked to lower rates of restructuring, better understandability, reduced coding effort and faster reworks of the system [15, 16, 42]. Moreover, from an external perspective, design quality has been linked to the quality of the end-product, both quantitatively and qualitatively [13, 42].

Despite the importance of design quality, little is known about the extent to which design is discussed during code reviews. We therefore set out to study the frequency and nature of design discussions in code reviews. To achieve our goal, we perform an empirical study on the code reviews of the Nova and Neutron projects from the OpenStack community. We first set out to better understand the frequency and type of design-related feedback through a qualitative analysis. More specifically, we manually classify 2,817 review

[1] https://www.reviewboard.org

comments from a randomly selected sample of 220 code review discussions to address the following two research questions:

(RQ1) How often is design discussed during code review?
Motivation: A common motivation for code review is to improve code quality [3]. Since design is an important aspect [10, 16, 22], we set out to study how often design issues are discussed during code review.
Results: Design concerns are not commonly discussed in code reviews. Although 33% of Nova and 35% of Neutron reviews have at least one design-related comment, only 9% of Nova and 14% of Neutron review comments are related to design.

(RQ2) Which design issues are discussed most often?
Motivation: Since design is a multi-faceted concept, to deepen our results from RQ1, we would like to know which aspects of design are being discussed during code review.
Results: Most of the raised design issues are due to an author's lack of awareness of the global context of the codebase. Indeed, of the seven design issues that are present in the analyzed review discussions (i.e., "Module Coupling", "Unnecessary Complexity", "Shallow Fix", "Redundant Code", "Performance", "Side Effect", and "Code Misplacement"), five can be linked to this lack of global awareness. Moreover, 73% of the design-related comments are constructive, offering alternative solutions that mitigate the raised design issues.

Next, we set out to better understand the nature of design-related feedback. We use the manually classified sample to train and evaluate classifiers to automatically label review comments as design related or not. Those classifiers are applied to a large sample of 2,506,308 review comments, and the results are used to address the following two research questions:

(RQ3) How accurately can our classifiers identify design-related comments?
Motivation: Classifiers that can automatically identify design-related comments would be helpful to study the characteristics of code changes that elicit design-related feedback. Since the usefulness of classifiers depend on their accuracy, we first set out to train and evaluate classifier performance.
Results: Our classifiers can achieve up to 59%-66% precision and 70%-78% recall, which outperform naïve classifiers such as zeroR by 43 percentage points in terms of F1-score.

(RQ4) Is design-related discussion correlated with the characteristics of code changes?
Motivation: Design-related discussion may be more likely to appear under certain conditions. For example, since novices may not understand the global context of the codebase (See RQ2), design discussions during their reviews may occur more frequently. Moreover, design-related discussion may be associated with the outcome of code changes. For example, design-related discussion may be associated with higher rates of abandonment, since design issues may require a rewrite of the code change. We set out to test such hypothesis using the automatically classified review data.
Results: We find that the authors of code reviews that include design discussions are statistically significantly less experienced than the authors of code reviews without design discussion (two-tailed unpaired Mann-Whitney U test, $p<0.001$). However, the difference is practically negligible (Cliff's delta = -0.06 Nova and -0.09 Neutron).
Moreover, the acceptance rate in code reviews with design-related feedback is 4 and 9 percentage points lower than code reviews without design-related feedback in Nova and Neutron, respectively. A Pearson χ^2 test indicates that this drop is statistically significant (DF=1, $p < 0.001$).

Table 1: An overview of the subject systems.

Product	Scope	Studied Period	#Code Changes	#Devs
Nova	Provisioning management	09/2011 to 01/2018	30,972	1,852
Neutron	Networking abstraction	07/2013 to 01/2018	16,894	1,177

Our results suggest that the modern shift to adopt code review automation is not enough to ensure that design is discussed during code reviews. Our design classifiers could be integrated into the reviewing interface to increase the transparency of the reviewing process. This could also be integrated with risk estimates [43] to understand the rigour of the reviewing process (e.g., whether design was discussed during a given review). By combining design-related classifiers with risk estimators, software organizations could make more pragmatic integration decisions (e.g., if a highly risky code change did not discuss design, integration may not be recommended).

Paper Organization

The remainder of this paper is organized as follows: Section 2 describes the design of our empirical study. Sections 3 and 4 present the results of our qualitative and quantitative studies, respectively. Section 5 discusses the broader implications of our results. Section 6 discloses the threats to the validity of our study. Section 7 surveys related work. Lastly, Section 8 draws conclusions.

2 EMPIRICAL STUDY DESIGN

In this section, we present our rationale for selecting the subject projects, as well as our approach for preparing our qualitative and quantitative data.

2.1 Subject Projects

To address our research questions, we aim to perform an empirical study on large and actively maintained software projects that regularly perform code reviews. Our qualitative analysis (Section 3) requires a large amount of manual effort, which makes large-scale analysis of data from several software projects impractical. Therefore, we select for analysis the OpenStack community—a software community that has heavily invested in (a) its code review process (e.g., all changes must be approved for integration by two core reviewers[2]) and (b) the integration of automation into its code review process (e.g., before integration, all changes are scanned by and must pass code style and static analysis verification bots).

[2]https://docs.openstack.org/infra/manual/developers.html#project-gating

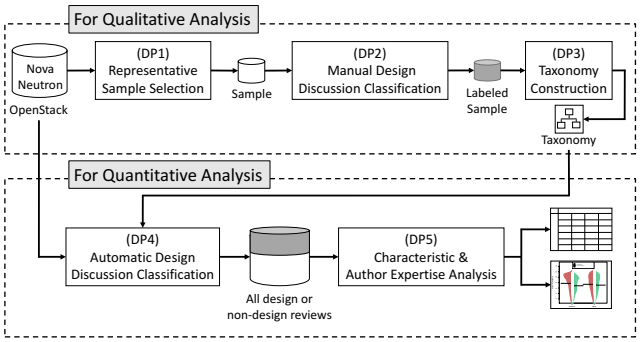

Figure 1: An overview of our data preparation approach

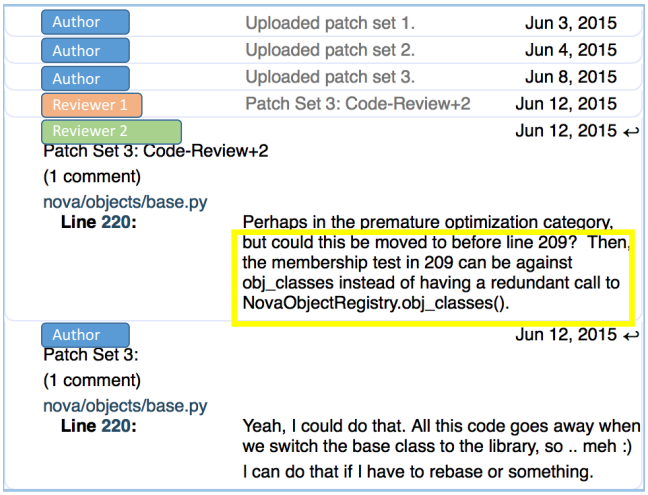

Figure 2: An Example of a review discussion

The OpenStack develops a set of software tools that provision and manage virtual and physical machines that comprise a cloud computing environment.[3] The code reviewing process of OpenStack projects is managed by the Gerrit Code Review tool,[4] and its data can be accessed via a REST API.[5] We select the two most active OpenStack projects for analysis. Table 1 provides an overview of the studied projects. Nova is responsible for provisioning management, while Neutron manages networking abstraction.

2.2 Code Review Process

Gerrit is a web-based tool that facilitates code review and integration management. The Gerrit worklflow begins with an author uploading an initial revision of their code changes for review. These changes are then scanned by verification bots that check for code style issues using linters, common mistakes using static analysis, and functional issues using automated tests. After these checks are complete, other developers can review the changes, writing comments for the author to consider.

Authors may address the feedback of verification bots and reviewers by uploading new revisions of their changes. This process is repeated until either (1) verification bots and reviewers are satisfied, and the code changes are approved for integration into project repositories; or (2) the author abandons the changes.

2.3 Qualitative Data Preparation

Figure 1 provides an overview of our preparation processes (DP1 and DP2) for addressing RQ1 and RQ2 of our qualitative analysis.

Representative Sample Selection (DP1). Using the Rest API provided by Gerrit, we download 527,287 reviews within the studied time frame, which begins with community adpotion of Gerrit (July 2011) and ends at the time of data collection (January 2018). We store the downloaded data in a NoSQL database using MongoDB.[6]

Since manual classification of a database of this size is impractical, we randomly sample reviews in each subject project.

We use the *PyMongo*[7] and *random*[8] Python modules to randomly generate Nova and Neutron Review IDs for analysis. To cover as broad of a set of design issues as possible, similar to prior work [33], we aim to achieve *saturation* [28] with our sample. To do so, we continue to classify randomly selected review comments until we find no new types of design issues for 50 reviews. We achieved saturation after classifying 100 Nova reviews with 1,357 comments and 120 Neutron reviews with 1,460 comments.

Manual Design Discussion Clarification (DP2). We identify design-related review comments by manually classifying review comments as design related or not. Design related review comments are identified according to the definition of design introduced by Brunet *et al.* [10]: *"As an artifact, design is a representation of how a portion of the code should be organized. As an activity, design is the process of discussing the structure of the code to organize abstractions and their relationships."*

The first and second authors participated in the manual classification process. The process involved classifying each comment in the discussion of the sampled review as design related or not. When necessary, the code changes themselves were analyzed for context. When the authors disagree, it is discussed until a consensus is reached.

Each comment may be tagged with multiple design issues; however, we find that multiple design concerns are rarely raised within one comment (only 18 instances in our sample of 2,817 comments). The whole discussion is then tagged with all of the unique labels of its comments. For example, Figure 2 shows that the comment surrounded by the yellow square is tagged as design related due to a "Code Misplacement" issue.

In addition to labelling every code review discussion with one or more raised design issues, we also record if alternative solutions have been provided. For instance, a comment in a discussion can be labelled as "code misplacement" without a solution (e.g., "This

[3] https://www.openstack.org
[4] https://www.gerritcodereview.com/
[5] https://gerrit-review.googlesource.com/Documentation/rest-api.html
[6] https://www.mongodb.com
[7] https://api.mongodb.com/python/current/
[8] https://docs.python.org/2/library/random.html

Table 2: Design Categories

Design Label	*Definition*	*Example*
Module Coupling	Comments that point out unnecessary dependencies between software artifacts.	*"so my proposal was to have the resource tracker create the ResourceProvider object (separately from the ComputeNode object). That way the ComputeNode and ResourceProvider objects don't have any interaction needed between them."* link
Redundant Code	Comments that identify duplicated or unnecessary code that needs to be eliminated. Usually due to not reusing existing functionality or, simply, addition of unwanted code.	*"nit: This is the default, and already set by the super class. It would be cleaner to omit it."* link
Performance	Comments that point out execution issues in terms of time and memory usage, which can be optimized using alternative approaches, while still maintaining correct functionality.	*"[This alternative] uses a much more efficient exponential doubling algorithm. It runs in 0.03 seconds versus a very long time for your implementation (I stopped waiting after a couple of minutes)."* link
Shallow Fix	Comments that point out limitations in the breadth of a fix in terms of the design of the code. This could be a fix that has a ripple effect on the code, or a fix that could potentially be generalized to fix other areas in the code suffering from the same problem.	*"The VIP port is not too different from any other port, so why checking only VIP's ip then? I think such check needs to be added to create_port method of db_base_plugin_v2, and not to LBaaS plugin only."* link
Side Effect	Comments pointing at code changes that might potentially break other parts of the code due to its design	*"Hardcoding this breaks non-qemu libvirt hypervisors. [..]"* link
Unnecessary Complexity	Avoidable complications in the code that can and should be further simplified without affecting the correct functionality of it.	*"The code here is going to query subnets table and fetch subnets objects, but those subnet objects are only used to check whether their enable_dhcp attribute is True or not. So, why not directly query Subnets table [..], and check whether at least one record exist."* link
Code Misplacement	Unsuitable or wrong code placement of software artifacts. Better placement might render the artifacts reachable and reusable by other modules or artifacts.	*"[..] We should mode that map from neutron.agent.firewall to some security group common module and re-use it. "* link
Non-Design Labels	These labels can be confused for design, but are not. They are shown as way to better understand the difference between what is considered as Design-related, according to our taxonomy.	
Readability	Not considered design. Related to the cosmetics of the code. It includes but is not limited to: indentation, naming conventions and spacing.	*"I think this should be renamed to SimpleTenantUsage"* link *"recommend just overriding the numaTopology variable instead of the awkward "fittedTopology" variable name :)"* link
Documentation	Intuitiveness, understandability and writing style of commit messages, code comments and code documentation.	*"So I think this might be change that doesn't require an API version bump, but we need a lot more context in the commit message to make that clear."* link *"you removed this function [..] if it's your desired action, at least should be mentioned in commit message"* link

method should not be in this class"), or with a solution (e.g., "This method should be in Class X, not Class Y").

To ensure that the labels that appeared later in the sample were not overlooked in the earlier comments, we perform a second pass over all of the labels after the initial classification pass. This classification process took seven days of full-time work of both authors. We use this classified sample to address RQ1 and RQ2.

Taxonomy Generation (DP3). After our manual classification (DP2), we apply open card sorting to construct a taxonomy from our tag data. This taxonomy helps us to extrapolate general themes from our detailed tag data. During the card sorting process, the tagged comments are merged into cohesive groups.

2.4 Quantitative Data Preparation

Figure 1 provides an overview of our preparation processes (DP4—DP6) for addressing RQ3 and RQ4 of our quantitative analysis.

Automatic Design Review Classification (DP4). To address RQ3, we train classifiers that can automatically identify design-related comments by applying machine learning techniques to our original review data. We select five popular techniques for our study: Multinomial Naïve Bayes (MNB), Support Vector Machines (SVM), Decision Tree, k-Nearest Neighbours, and Random Forest.

We begin the automatic classification by pre-processing the raw comment text. First, we remove stop words, i.e., words that add little semantic meaning to a document, using the stopword list of the

Python NLTK module.[9] Next, to mitigate the impact of conjugation, we apply the Porter stemmer to each surviving word.[10] Finally, we convert each comment to a vector that is composed of the Term Frequency-Inverse Document Frequency (TF-IDF) weights of its words. In a nutshell, terms that rarely appear in comments, and often appear within a comment are of higher weight for a comment.

To estimate the performance of our models on unseen data, we perform 10-fold cross validation, where the sample data is split into ten folds of equal size, nine folds are used to train the model, and the remaining fold is used to test the model. The process is repeated ten times (where each fold is treated as the testing fold once), and the mean performance results are reported.

Characteristic Analysis (DP5). To address RQ4, we study the relationship between incidences of design-related discussion and pre-/post-integration code change characteristics. In terms of pre-integration properties, we analyze author experience because we conjecture that authors with low experience may be more susceptible to making design mistakes. We estimate the author expertise using heuristics that we derive from prior work [40]. Since the primary focus of our study is on reviewing, our author expertise heuristic counts the number of prior reviews that the author has written or reviewed.

In terms of post-integration properties, we analyze the rate of review abandonment, since it is the primary post-integration property. We identify reviews that are abandoned (i.e., not integrated). To do so, we classify the design-related/unrelated comments that DP4 generates into accepted or abandoned categories. We exclude ongoing reviews (i.e., those with the "OPEN" status) from this analysis because it is not clear if they will be accepted or abandoned yet.

3 QUALITATIVE STUDY RESULTS

In this section, we present the results of our qualitative analysis with respect to RQ1 and RQ2. For each question, we describe our approach for addressing it followed by the results that we observe.

(RQ1) How often is design discussed during code review?

Approach. To address RQ1, we examine how often reviewers discuss design issues in the Nova and Neutron projects. To do so, we manually tag comments that are related to design issues (See Section 2 DP2). The tags focus on the design concerns of authors and reviewers. They also differentiate between authors and reviewers to capture the difference in participation behaviour for both roles.

Similar to prior studies [3, 31, 38], we apply open card sorting to construct a taxonomy from our tag data. This taxonomy helps us to extrapolate general themes from our detailed tag data. The card sorting process is comprised of three steps. First, the first and second authors perform the classification of review comments as per DP2. Second, we recursively group related labels according to topic to produce a hierarchy of design issues. Finally, we count the occurrences of design-related comments in each studied project.

[9]https://www.nltk.org/api/nltk.corpus.html#module-nltk.corpus
[10]http://www.nltk.org/howto/stem.html

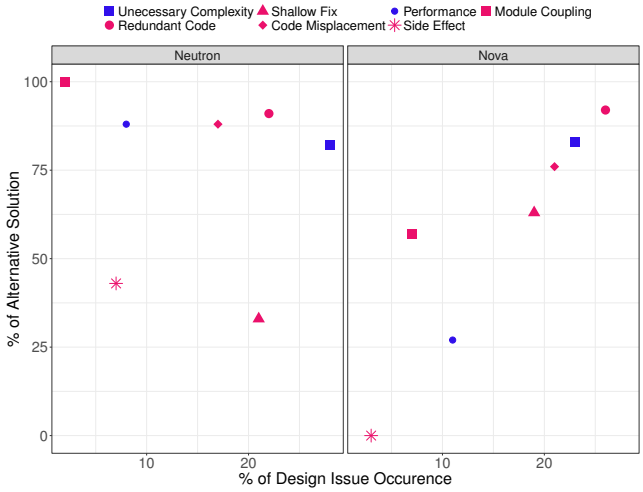

Figure 3: The rate of each design category across design-related comments

In a sample of 50 review comments that the raters individually labelled, a Cohen's Kappa of 0.72 was observed, which indicates substantial agreement.

Results. Observation 1—Design issues are not commonly discussed. Figure 3 shows a distribution of design issues that emerged during manual classification and the percentage of alternative solutions that were provided for each of the identified issues. We find that 33% and 35% of reviews have at least one design-related comment in Nova and Neutron, respectively. However, we find that only 9% and 14% of Nova and Neutron review comments are related to design, which implies that even in design-related code reviews, design discussions are short and scarce.

Observation 2—Authors rarely engage in design-related discussions. Most of the design-related comments are initiated by reviewers, whereas 11% and 21% of those comments elicit authors' responses in Nova and Neutron, respectively. This is alarming as low review participation has been linked to poor software quality [24].

We suspect that this tendency may have to do with the power that (core) reviewers hold over authors. Indeed, authors need reviewers to approve their changes in order for them to be integrated into the project. Thus, authors may feel compelled to simply do what the reviewer asks. A more rigorous analysis is needed to confirm (or reject) this suspicion.

> Design concerns are not commonly discussed in code reviews. Moreover, authors rarely engage in design-related discussions with reviewers.

(RQ2) Which design issues are discussed most often?

Approach. To address RQ2, we examine the types of design issues in the studied projects. To do so, we examine the properties of

the taxonomy that our open card sorting produced (See Section 2, DP2—DP3).

Results. Table 2 shows our resulting hierarchy-less taxonomy of mutually exclusive design issues that have been encountered and identified during manual classification [34]. Furthermore, we provide some non-design labels, namely "Readability" and "Documentation", both of which can easily be confused with design. Despite being unrelated to the functionality of the code, these categories are concerned with the code cosmetics and understandability, respectively, rather than design.

Observation 3–Design issues are often due to lack of awareness of the global context of the codebase. Figure 3 shows the type of design issues that emerge during the sampled code reviews. We observe that all of the design issues except for "Performance" and "Unnecessary Complexity" (blue shapes in Figure 3) are linked to an incomplete understanding of the codebase. Altogether, a lack of global context awareness accounts for 67% and 65% of the sampled design issues in Nova and Neutron, respectively (red shapes in Figure 3). For instance, duplication may occur because the author is not aware of the existing functionality, while shallow fixes, code misplacement, side effects, and coupling concerns may occur because the author is not aware of the hierarchy and relationships between modules in the codebase. However, their emergence in code reviews are a positive indication that these types of issues are being noticed and corrected during the reviewing process.

Observation 4—Most of the design-related discussions are constructive, offering alternative solutions that mitigate the raised design issues. Collectively, 73% of the sampled design comments provide an alternative solution to help authors to address the raised design issues. We believe that this is an indication of a healthy reviewing community, since such feedback is likely more constructive than simply pointing out flaws.

As Figure 3 shows, reviewers are less likely to provide alternative solutions for design issues like "Code Misplacement", "Performance", "Module Coupling", and "Side Effect" than the other design issues. Indeed, 93% of the time, reviewers provide alternative solutions to address "Redundant Code".

For example, in review #319327,[11] the reviewer pointed out that the functionality already exists: "We actually have a method to do that in [..]". The author fixed the issue and was eventually able to merge the code change.

> Most design issues are raised due to lack of awareness of the global context of the codebase. Moreover, most of those comments are constructive, providing alternative solutions.

4 QUANTITATIVE STUDY RESULTS

In this section, we present the results of our quantitative analysis with respect to RQ3 and RQ4. For each question, we describe our approach for addressing it followed by the results that we observe.

[11] https://review.openstack.org/#/c/319327/

(RQ3) How accurately can our classifiers identify design-related comments?

Approach. To address RQ3, we train classifiers models that can automatically identify design-related comments. Our models learn a word vector of a comment where words are weighted by TF-IDF scores (see Section 2.4). For this analysis, we use the comments that were not included in the sample that was analyzed in Section 3.

Our dataset is imbalanced—only 9% and 14% of Nova and Neutron comments are design related. To reduce bias towards the majority class (non-design), the training data must be re-sampled. Our re-sampling process is implemented using a combination of oversampling of the minority class and undersampling of the majority class. We do so because prior work has shown that such a combination performs better than either oversampling or undersampling in isolation [1, 11].

We use the common classifier performance measures of recall, precision, and F1-score. Recall is the ratio of actual design-related comments that a classifier can detect ($Recall = \frac{TP}{TP+FN}$), and precision is the ratio of correctly predicted design-related comments to the total number of reviews that were predicted to be design-related ($Precision = \frac{TP}{TP+FP}$). There is a tradeoff between recall and precision. To account for this, we use the F1-score, which is the harmonic mean of recall and precision: $F1 = \frac{2 \times Precision \times Recall}{Precision + Recall}$

The above mentioned performance metrics (i.e., recall, precision, and F1-score) depend on the threshold that is selected to classify instances as design related or not (typically set to 0.5). Choosing different thresholds may yield different results. To get an overall idea of the performance across thresholds, we additionally use the Area Under the receiver operating characteristics Curve (AUC). AUC ranges from 0 to 1, and larger AUC values indicate better prediction performance. The advantage of AUC is its robustness toward imbalanced data, since the curve is obtained by varying the classification threshold over all possible values.

We compare the performance of our classifiers to that of traditional baselines like random guessing and zeroR. By construction, random guessing would achieve an AUC of 0.5. Hence, AUC values above 0.5 indicate performance that suggests the classifier contains some information. In addition, a zeroR classifier is a classifier that always reports the class of interest. Hence, in our scenario, a zeroR classifier achieves 100% recall and a precision equal to the rate of design-related comments. Since these values are unreasonably high (recall) and low (precision), we compare our classifiers to the F1-score of the zeroR classifier to counteract the extreme values.

Results. Observation 5—Multinomial Naïve Bayes achieves the best recall values. Table 3 shows that MNB achieves recall values of 69% and 76% in Nova and Neutron, respectively. As for the general performance of all classifiers, the precision ranges from 59% to 70%, whereas the recall ranges from 55% to 76%. Overall, we observe that Neutron has better performance values than Nova. We suspect that this is due to Neutron having more design comments in its training data (14%) compared to Nova (9%).

Observation 6—Our classifiers outperform zeroR by 43 percent points in terms of F1-score. Since zeroR achieves 9% (Nova) and 14% (Neutron) of precision, the best classifier MNB outperforms zeroR by 50 (Nova) and 52 (Neutron) percentage points in terms of precision. Moreover, the AUC ranges from 0.55 to 0.85,

Table 3: Binary Classification 10-fold Cross-Validation Results with balanced datasets

Algorithm	Neutron				Nova			
	Prec.	***Recall***	***F1***	***AUC***	***Prec.***	***Recall***	***F1***	***AUC***
MNB	0.66	0.76	0.68	0.85	0.59	0.69	0.60	0.79
SVM	0.70	0.71	0.70	0.84	0.63	0.62	0.62	0.76
Decision Tree	0.65	0.66	0.65	0.66	0.59	0.61	0.59	0.61
KNN	0.59	0.60	0.36	0.70	0.63	0.55	0.56	0.55
Random Forests	0.69	0.64	0.65	0.80	0.66	0.55	0.56	0.75

indicating that our models outperform random guessing by 35 percentage points.

Observation 7— Our classifiers strengthen the finding that design discussions are scarce. Following Observation 1, our classifiers suggest that 43% and 44% of all code reviews contain at least one design comment in Nova and Neutron, respectively. However, said design comments are so rare that they only amount to 4% and 6% of review comments in Nova and Neutron, respectively.

> Multinomial Naïve Bayes achieves the best recall values among our classifiers, outperforming a zeroR classifier by 43 percent points in F1-score. Moreover, our automatic classification complements our manual classification finding, suggesting that design discussions are indeed rare.

(RQ4) Is design-related discussion correlated with the characteristics of code changes?

Approach. To address RQ4, we first apply our top-performing classifier from RQ3 to the entire set of review comments from Nova and Neutron to classify them as design related or not. Then, we compare the author experience cores and rates of the abandonment of code changes in the reviews that contain design-related comments and those that do not.

The author experience score is the number of times that an author has participated in the given project prior to the review at hand. We then compare the author expertise values, using bean plots. Bean plots are boxplots in which the vertical curves summarize and compare the distributions of different data sets [19], which in our case correspond to the author expertise of design-related reviews and reviews without design comments. The horizontal black lines indicate the mean of the author expertise for design-related reviews (red) and non-design reviews (green), shown in Figure 4.

We estimate the statistical significance of the difference using two-sided, unpaired Mann-Whitney U tests ($\alpha = 0.05$) and the practical significance of the difference using Cliff's delta (δ). The effect is considered negligible when $0 \leq |\delta| < 0.147$, small when $0.147 \leq |\delta| <$, medium when $0.33 \leq |\delta| < 0.474$, and large otherwise. We use Mann-Whitney U tests and Cliff's delta since these are non-parametric tests and our data is not normally distributed. We check the statistical significance of that change in rate of abandonment using a Pearson χ^2 test (one degree of freedom, $\alpha = 0.05$).

Results. Observation 8—Authors of design-related reviews tend to have lower expertise. Table 4 shows that design-related reviews tend to have lower author expertise than reviews without design comments in both studied projects. Mann-Whitney U tests

Table 4: Summary statistics of accepted and abandoned reviews

	Neutron		Nova	
	Design	***Not***	***Design***	***Not***
Merged Reviews	5100	7387	9344	13303
Abandoned Reviews	2340	2067	3835	4490
% of Abandonment	31%	22%	29%	25%
p-value	$<2.2\times10^{-16}$		3.67×10^{-14}	

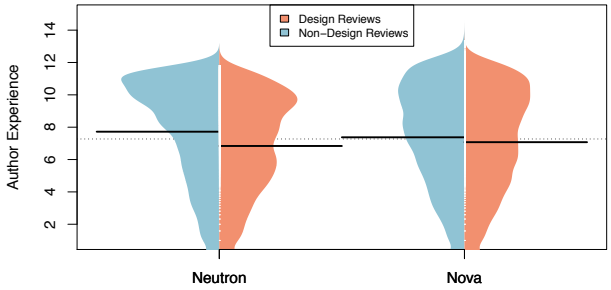

Figure 4: Difference in author experience in design-related and design-unrelated code reviews

show that there is a statistically significant difference in expertise between design-related and design-unrelated reviews in Nova and Neutron ($p < 2.2 \times 10^{-16}$ in both cases). However, the Cliff's delta values suggest that the practical difference is negligible (-0.09 in Neutron and -0.06 in Nova). Although the difference is practically negligible, the direction generally agrees with Observation 3, i.e., the authors of design-related reviews are less likely to be aware of the global context of the codebase.

Observation 9—Design-related reviews are more likely to be abandoned. Table 4 shows that 29% and 31% of design-related reviews are abandoned in Nova and Neutron, respectively. When compared with reviews without design comments, we find that design-related reviews account for an increase of 4 and 9 percentage points in Nova and Neutron, respectively. A Pearson χ^2 test shows that there is a statistically significant difference between design-related and reviews without design comments in terms of the rate of abandonment in both studied projects. These results complement those of Tao *et al.* [45], who observed that design-related reviews were less likely to be accepted in Mozilla.

Authors of design-related reviews tend to have less expertise than reviews without design comments. Moreover, design-related reviews are more likely to be abandoned in our subject projects.

5 PRACTICAL IMPLICATIONS

In this section, we discuss the broader implications of our observations for software organizations, tool developers, and authors of code changes.

5.1 Software Organizations

Software organizations should encourage reviewers to take design aspects into account. Only 9% and 14% of code review comments are design related in Nova and Neutron, respectively (Observation 1). Moreover, 11% and 21% of those comments elicit authors' responses in Nova and Neutron (Observation 2). Despite increasing rates of automation of reviewing tasks in the OpenStack community, these observations suggest that design-level discussions are still rare. Indeed, automation of menial tasks is not enough to achieve high rates of high-level review discussion.

Design-related reviews are likely to struggle with their integration. We find that 67% and 65% of the design issues observed are due to a lack of awareness of the global context of the codebase in Nova and Neutron, respectively (Observation 3). This may explain why we observe that those design-related reviews are significantly more likely to be abandoned (Observation 9).

5.2 Tool Developers

Integration of design classifiers into the reviewing interface would increase the transparency of the reviewing process. Our design classifiers can detect design discussions in past code reviews. If these classifiers are integrated into the code review user interface, integration decisions could be based on a clearer understanding of the review process. This could be combined with risk estimates (e.g. the impact of system deliverables [43]) to help determine whether the rigour of the code reviewing process is too low to integrate the corresponding code changes into the codebase.

5.3 Authors of Code Changes

Authors should not be intimidated to engage in design-related discussions. Authors rarely engage in design-related discussion (Observation 2). We conjecture that author-reviewer power dynamics may be at play, since reviewers have authority over the acceptance of code changes. However, in design-related reviews, reviewers mostly offer constructive feedback to help authors handle their design issues (Observation 4), which in return will broaden their global knowledge of the codebase (Observation 3).

6 THREATS TO VALIDITY

In this section, we describe the threats to the validity of our study.

6.1 External Validity

External validity concerns have to do with the generalizability of our study. Due to the manually intensive nature of our qualitative study, we chose to focus our analysis on two open-source projects from one community (OpenStack). Due to our small sample size, it is difficult to draw general conclusions about software projects. However, the goal of the study is not to provide a theory of design discussions in code reviews that applies to all systems. Instead we believe that our study could serve as a baseline for design classification and design involvement in code reviews. Future replication studies will help to broaden our insights to more general trends.

6.2 Internal Validity

Internal validity concerns may be raised if other plausible hypotheses may explain our observations. Since we select a sample of review comments in our qualitative analysis, sampling bias might have an impact on our conclusions. For example, the studied projects may have dedicated explicit effort to design improvement at particular time periods. If those periods were over or undersampled, our results may be skewed. To mitigate this threat, we randomly selected the reviews for manual classification, so that our findings are not bound to a certain period in the project history.

6.3 Construct Validity

Construct validity concerns may creep into our study if our measurements are misaligned with the phenomena we set out to study. Our detection of design-related comments is a manual process which may be subjective. However, we mitigated this threat by having two raters agree about each classified result. The raters are both graduate students with industrial experience in code reviewing practices. Moreover, an individually classified sample yielded a Cohen's Kappa of 0.72 (substantial agreement). To facilitate further refinement of our classification, we provide our definition and shared understanding of each design category, as well as examples in Table 2. To further strengthen our observations, as suggested by Ralph [34], in future work, we will conduct developer interviews to see if our observations resonate with the developers.

7 RELATED WORK

In this section, we present the related work with respect to code review practice, design quality assurance, and code review characteristics dimensions.

7.1 Code Review Practice

Code review is a common practice that is introduced to improve the code quality [4]. Adequate code reviews can detect bugs, increase productivity, and even improve documentation [17]. Several studies set out to explore the factors of code reviews that affect software quality. For instance, reviewer involvement has been linked with incidence rates of post-release defects [24, 25, 39], defect proneness [21], design anti-patterns [29], and security vulnerabilities [26]. Moreover, the documentation level of code reviews has been linked with the level of maintainability of the produced code [7, 30].

While prior work has analyzed the broad aspects of code review related to code quality, little is known about the presence of design elements and design discussion in code reviews. Better design has been linked to lower rates of restructuring, better understandability, reduced coding effort, and faster reworks of the system [15, 16, 42]. This inspired our study, which focuses on feedback content by

analyzing how often reviews discuss design issues and what kinds of design issues are raised.

Code reviews have a much broader scope [2, 6] than only improving sotware quality. Czerwonka *et al.* [12] found that code reviews often do not find functionality issues that should block a code submission. Bacchelli and Bird [3] found that code reviews at Microsoft aim at not only bug fixing, but also knowledge transfer among team members. Mäntylä and Lassenius [23] found that there are three evolvability issues raised for every functionality issue during code reviews. Beller et al. [7] found a similar ratio of evolvability and functional issues are fixed during code reviews. These findings motivated the search for design discussions in code reviews, which are also considered non-functional issues.

7.2 Design Quality Assurance

Software design is essential to ensure the quality of the software product [10, 16, 22]. For instance, Kemerer and Paulk [20] showed that adding design reviews to projects at the SEI led to software products that were of higher quality. Brunet *et al.* [10] found that only 25% of discussions are design related in commits, issues, and pull requests. Our study complements prior work, observing that only 9% and 14% of comments are design related in Nova and Neutron.

Design-related discussions may include various types of issues. Tao *et al.* [45] found that code changes that comprise design issues like suboptimal solutions and incomplete fixes (e.g., "Shallow Fix") are often linked with the rejection of the code changes in the Eclipse and Mozilla projects. We also find that, in general, design issues present in code reviews, including "Shallow Fix", are correlated with the abandonment of code changes in Nova and Neutron.

Sedano *et al.* [37] found that duplicated work like "Redundant Code" and "Unnecessary Complexity" is a frequently occurring type of software development waste. Our results show that design-related comments are raised due to not only "Redundant Code" and "Unnecessary Complexity", but also "Module Coupling", "Performance", "Side Effect", "Shallow Fix", and "Code Misplacement". Yamashita and Moonen [44] found that 32% of developers are not aware of the code smells in their own code. Paixao *et al.* found that 62% of the time, developers do not discuss the architectural impact of their changes on the system, suggesting a lack of awareness of them [32]. We also find that a lack of awareness of the global context of the codebase is linked to incidence rates of design comments.

7.3 Code Review Characteristics

Code review characteristics have been analyzed to understand their relationship with integration decisions. Tsay *et al.* [41] showed that code changes that attract many comments are less likely to be accepted. Indeed, McIntosh *et al.* [24, 25] and Thongtanunam *et al.* [39] have argued that the amount of discussion that was generated during review should be considered when making integration decisions. Gousios *et al.* [18] found that only 13% of pull requests are rejected due to technical reasons. Our results suggest that design-related reviews are likely to struggle with integration.

Prior work showed the importance of author expertise in code reviews. Bosu *et al.* [9] found that module-based expertise shares a link with code review usefulness (as expressed by authors of the code changes). Meneely *et al.* [27] examined the association between the number of commits of developers and security problems in the Red Hat Enterprise Linux 4 kernel. Bosu and Carver [8] found that code changes written by inexperienced authors tend to receive little review participation. Our results suggest that authors of design-related reviews have lower expertise than reviews without design comments. However, in those design-related reviews, reviewers mostly provide constructive feedback that includes suggestions to help inexperienced authors to resolve the design problems.

8 CONCLUSION

Code review is a common software quality assurance practice. A shift towards automated detection of low-level issues has, in theory, freed reviewers up to focus on higher level issues, such as software design. Yet in practice, little is known about the extent to which design is discussed during code review.

In this paper, we set out to better understand the frequency and nature of design-related discussions in code reviews. To achieve our goal, we qualitatively and quantitatively analyze design-related comments in the OpenStack Nova and Neutron projects. We make the following observations:

- Design concerns are not commonly discussed in code reviews. Moreover, authors rarely engage in design-related discussions with reviewers.
- Most design issues are raised due to lack of awareness of the global context of the codebase. However, most design-related comments are constructive, providing suggestions to mitigate the issue.
- Multinomial Naïve Bayes produces classifiers that acheive the best recall values in our samples. Moreover, those classifiers outperform a zeroR classifier by 43 percent points in terms of F1-score.
- Authors of design-related reviews tend to have lower expertise than those reviews without design-related comments.
- Design-related reviews are more likely to be abandoned than reviews without design-related comments.

Our results suggest that: (a) the modern shift code review automation is not enough to ensure that design is discussed during code reviews; (b) design classifiers could be integrated into the reviewing interface to increase the transparency of the reviewing process; and (c) authors should not be intimidated to engage in design-related discussions, since reviewers usually offer constructive suggestions for improvement.

Replication Package

To enable future work, we have made the manually classified reviews and the scripts that we used to analyze them available online.[12]

ACKNOWLEDGMENTS

This work was supported by the Al Ghurarir Foundation For Education STEM scholar program, and JSPS KAKENHI Grant Numbers 17J09333 and 17H00731.

[12] https://github.com/software-rebels/DesignInCodeReviews-ESEM2018

REFERENCES

[1] Rana Alkadhi, Manuel Nonnenmacher, Emitza Guzman, and Bernd Bruegge. 2018. How do developers discuss rationale?. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 357–369.

[2] Guzzi Anja, Bacchelli Alberto, Lanza Michele, Pinzger Martin, and van Deursen Arie. 2013. Communication in open source software development mailing lists. In *2013 10th Working Conference on Mining Software Repositories (MSR)*. 277–286. https://doi.org/10.1109/MSR.2013.6624039

[3] Alberto Bacchelli and Christian Bird. 2013. Expectations, outcomes, and challenges of modern code review. In *Proceedings of the 2013 international conference on software engineering*. IEEE Press, 712–721.

[4] Richard A Baker Jr. 1997. Code reviews enhance software quality. In *Proceedings of the 19th international conference on Software engineering*. ACM, 570–571.

[5] Vipin Balachandran. 2013. Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation. In *Software Engineering (ICSE), 2013 35th International Conference on*. IEEE, 931–940.

[6] Tobias Baum, Olga Liskin, Kai Niklas, and Kurt Schneider. 2016. Factors Influencing Code Review Processes in Industry. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016)*. ACM, New York, NY, USA, 85–96. https://doi.org/10.1145/2950290.2950323

[7] Moritz Beller, Alberto Bacchelli, Andy Zaidman, and Elmar Juergens. 2014. Modern code reviews in open-source projects: Which problems do they fix?. In *Proceedings of the 11th working conference on mining software repositories*. ACM, 202–211.

[8] A. Bosu and J. C. Carver. 2013. Impact of Peer Code Review on Peer Impression Formation: A Survey. In *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*. 133–142. https://doi.org/10.1109/ESEM.2013.23

[9] Amiangshu Bosu, Michaela Greiler, and Christian Bird. 2015. Characteristics of Useful Code Reviews: An Empirical Study at Microsoft. In *Proceedings of the 12th Working Conference on Mining Software Repositories (MSR '15)*. IEEE Press, Piscataway, NJ, USA, 146–156. http://dl.acm.org/citation.cfm?id=2820518.2820538

[10] João Brunet, Gail C Murphy, Ricardo Terra, Jorge Figueiredo, and Dalton Serey. 2014. Do developers discuss design?. In *Proceedings of the 11th Working Conference on Mining Software Repositories*. ACM, 340–343.

[11] Nitesh V Chawla, Kevin W Bowyer, Lawrence O Hall, and W Philip Kegelmeyer. 2002. SMOTE: synthetic minority over-sampling technique. *Journal of artificial intelligence research* 16 (2002), 321–357.

[12] Jacek Czerwonka, Michaela Greiler, and Jack Tilford. 2015. Code reviews do not find bugs: how the current code review best practice slows us down. In *Proceedings of the 37th International Conference on Software Engineering-Volume 2*. IEEE Press, 27–28.

[13] Fernando Brito e Abreu and Walcelio Melo. 1996. Evaluating the impact of object-oriented design on software quality. In *Software Metrics Symposium, 1996., Proceedings of the 3rd International*. IEEE, 90–99.

[14] Michael E Fagan. 1999. Design and code inspections to reduce errors in program development. *IBM Systems Journal* 38, 2/3 (1999), 258.

[15] Martin Fowler and Kent Beck. 1999. *Refactoring: improving the design of existing code*. Addison-Wesley Professional.

[16] Erich Gamma. 1995. *Design patterns: elements of reusable object-oriented software*. Pearson Education India.

[17] Tom Gilb, Dorothy Graham, and Susannah Finzi. 1993. *Software inspection*. Vol. 253. Addison-Wesley Reading.

[18] Georgios Gousios, Martin Pinzger, and Arie van Deursen. 2014. An Exploratory Study of the Pull-based Software Development Model. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. ACM, New York, NY, USA, 345–355. https://doi.org/10.1145/2568225.2568260

[19] Peter Kampstra. 2008. Beanplot: A Boxplot Alternative for Visual Comparison of Distributions. *Journal of Statistical Software, Code Snippets* 28, 1 (October 2008), 1–9. http://www.jstatsoft.org/v28/c01

[20] Chris F Kemerer and Mark C Paulk. 2009. The impact of design and code reviews on software quality: An empirical study based on psp data. *IEEE transactions on software engineering* 35, 4 (2009), 534–550.

[21] Oleksii Kononenko, Olga Baysal, Latifa Guerrouj, Yaxin Cao, and Michael W Godfrey. 2015. Investigating code review quality: Do people and participation matter?. In *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on*. IEEE, 111–120.

[22] Craig Larman. 2012. *Applying UML and Patterns: An Introduction to Object Oriented Analysis and Design and Interative Development*. Pearson Education India.

[23] Mika V. Mäntylä and Casper Lassenius. 2009. What Types of Defects Are Really Discovered in Code Reviews? *IEEE Transactions on Software Engineering* 35, 3 (2009), 430–448.

[24] Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E Hassan. 2014. 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*. ACM, 192–201.

[25] Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E Hassan. 2016. An empirical study of the impact of modern code review practices on software quality. *Empirical Software Engineering* 21, 5 (2016), 2146–2189.

[26] Andrew Meneely, Alberto C. Rodriguez Tejeda, Brian Spates, Shannon Trudeau, Danielle Neuberger, Katherine Whitlock, Christopher Ketant, and Kayla Davis. 2014. An Empirical Investigation of Socio-technical Code Review Metrics and Security Vulnerabilities. In *Proceedings of the 6th International Workshop on Social Software Engineering (SSE 2014)*. ACM, New York, NY, USA, 37–44. https://doi.org/10.1145/2661685.2661687

[27] Andrew Meneely and Laurie Williams. 2009. Secure Open Source Collaboration: An Empirical Study of Linus' Law. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS '09)*. ACM, New York, NY, USA, 453–462. https://doi.org/10.1145/1653662.1653717

[28] Matthew B Miles, A Michael Huberman, Michael A Huberman, and Michael Huberman. 1994. *Qualitative data analysis: An expanded sourcebook*. sage.

[29] Rodrigo Morales, Shane McIntosh, and Foutse Khomh. 2015. Do code review practices impact design quality? a case study of the qt, vtk, and itk projects. In *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*. IEEE, 171–180.

[30] Stacy Nelson and Johann Schumann. 2004. What makes a code review trustworthy?. In *System Sciences, 2004. Proceedings of the 37th Annual Hawaii International Conference on*. IEEE, 10–pp.

[31] Kerzazi Noureddine, Khomh Foutse, and Adams Bram. 2014. Why Do Automated Builds Break? An Empirical Study. In *2014 IEEE International Conference on Software Maintenance and Evolution*. 41–50.

[32] Matheus Paixao, Jens Krinke, DongGyun Han, Chaiyong Ragkhitwetsagul, and Mark Harman. 2017. Are developers aware of the architectural impact of their changes?. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 95–105.

[33] Rigby Peter C. and Storey Margaret-Anne. 2011. Understanding broadcast based peer review on open source software projects. In *2011 33rd International Conference on Software Engineering (ICSE)*. 541–550. https://doi.org/10.1145/1985793.1985867

[34] Paul Ralph. 2018. Toward Methodological Guidelines for Process Theories and Taxonomies in Software Engineering. *IEEE Transactions on Software Engineering* (2018).

[35] Jason Remillard. 2005. Source code review systems. *IEEE software* 22, 1 (2005), 74–77.

[36] Guoping Rong, Jingyi Li, Mingjuan Xie, and Tao Zheng. 2012. The effect of checklist in code review for inexperienced students: An empirical study. In *Software Engineering Education and Training (CSEE&T), 2012 IEEE 25th Conference on*. IEEE, 120–124.

[37] Todd Sedano, Paul Ralph, and Cécile Péraire. 2017. Software Development Waste. In *Proceedings of the 39th International Conference on Software Engineering (ICSE '17)*. IEEE Press, Piscataway, NJ, USA, 130–140. https://doi.org/10.1109/ICSE.2017.20

[38] Mini Shridhar, Bram Adams, and Foutse Khomh. 2014. A Qualitative Analysis of Software Build System Changes and Build Ownership Styles. In *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM '14)*. ACM, New York, NY, USA, Article 29, 10 pages. https://doi.org/10.1145/2652524.2652547

[39] Patanamon Thongtanunam, Shane McIntosh, Ahmed E. Hassan, and Hajimu Iida. 2015. Investigating Code Review Practices in Defective Files: An Empirical Study of the Qt System. In *Proceedings of the 12th Working Conference on Mining Software Repositories (MSR '15)*. IEEE Press, Piscataway, NJ, USA, 168–179. http://dl.acm.org/citation.cfm?id=2820518.2820540

[40] Patanamon Thongtanunam, Shane McIntosh, Ahmed E. Hassan, and Hajimu Iida. 2017. Review participation in modern code review. *Empirical Software Engineering* 22, 2 (01 Apr 2017), 768–817. https://doi.org/10.1007/s10664-016-9452-6

[41] Jason Tsay, Laura Dabbish, and James Herbsleb. 2014. Influence of Social and Technical Factors for Evaluating Contribution in GitHub. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. ACM, New York, NY, USA, 356–366. https://doi.org/10.1145/2568225.2568315

[42] John Vlissides, Richard Helm, Ralph Johnson, and Erich Gamma. 1995. Design patterns: Elements of reusable object-oriented software. *Reading: Addison-Wesley* 49, 120 (1995), 11.

[43] Ruiyin Wen, David Gilbert, Michael G. Roche, and Shane McIntosh. 2018. BLIMP Tracer: Integrating Build Impact Analysis with Code Review. In *Proc. of the International Conference on Software Maintenance and Evolution (ICSME)*. To appear.

[44] Aiko Yamashita and Leon Moonen. 2013. Do developers care about code smells? an exploratory survey. In *Reverse Engineering (WCRE), 2013 20th Working Conference on*. IEEE, 242–251.

[45] Tao Yida, Han Donggyun, and Kim Sunghun. 2014. Writing Acceptable Patches: An Empirical Study of Open Source Project Patches. In *2014 IEEE International Conference on Software Maintenance and Evolution*. 271–280. https://doi.org/10.1109/ICSME.2014.49

[46] Edward Yourdon. 1989. *Structured walkthroughs*. Yourdon Press.