Feedback Topics in Modern Code Review: Automatic Identification and Impact on Changes

Janani Raghunathan, Lifei Liu, Huzefa Kagdi
Department of Electrical Engineering and Computer Science
Wichita State University
Wichita, Kansas 6760, USA
Email: {jxraghunathan, lxliu2, huzefa.kagdi}@wichita.edu

Abstract— **Recent empirical studies show that the practice of peer-code-review improves software quality. Therein, the quality is examined from the external perspective of reducing the defects/failures, i.e., bugs, in reviewed software. There is a very little to no investigation on the impact of peer-code-review on improving the internal quality of software, i.e., what exactly is affected in code, due to this process. To this end, we conducted an empirical study on the human-to-human discourse about the code changes, which are recorded in modern code review tools in the form of review comments. Our objective of this study was to investigate the topics which are typically addressed via the textual comments. Although, there is an existing taxonomy of topics, there is no automatic approach to categorize code reviews. We present a machine-learning-based approach to automatically classify reviewer-to-reviewer and reviewer-to-developer comments on proposed code changes. We applied this approach on 468 code review comments of four open source systems, namely *eclipse*, *mylyn*, *android* and *openstack*. The results show that *Evolvability* categories are dominating topics. In an attempt to verify these observations, we analyzed the code changes that developers performed on receiving these comments. We identified several refactorings that are congruent with the topics of review comments. Refactorings are mechanisms to improve the internal structure of software. Therefore, our work provides initial empirical evidence on the effectiveness of peer-code-review on improving internal software quality.**

I. Introduction

Different types of maintenance help improve sustainability and quality of large-scale software systems. *Corrective* maintenance helps eliminating or reducing defects in the software; thereby, improving the external software quality. *Preventive* or *Perfective* types of maintenance may not necessarily address the defects or features at hand directly; however, they improve the design or code structure; thereby, improving the internal software quality. Software testing is primarily used to identify the defects and is often considered as an ubiquitous mechanism to improve the external quality. Code review is a rejuvenated phenomena with benefits in improving the software quality.

Extensive research shows the usefulness of code review in improving the external quality [1], [2], [3], [4], which is an important result. Unfortunately, there is little to no work on investigating how code review improves the internal software quality. Previous studies [5], [6], [7], [8], [9], [10], [11], [12] show that the internal quality is also critical and equally (or more) important. For example, evolvability or internal design issues could lead to code decay and/or premature degradation.

DOI reference number: 10.18293/SEKE2018-097

Peer-code review is the process of reviewers critiquing the code changes that developers submit to decide if those changes are of acceptable quality and can be integrated to the main code base of a software system. Nowadays, it is often lightweight, informal and tool-based, which is termed as Modern Code Review (MCR) [13]. MCR is popularly used in industrial and open source software-development paradigms [14], [2], [15]. The primary discourse between reviewers and developers is in the form of textual feedback about the code changes at the line level or collectively within an enabling tool, e.g., *Gerrit*.

The primary objective of this paper is to investigate how code review is effective in improving the internal software quality, which we substantiated with three research questions:

RQ1: What are the common topics, related to both internal and external qualities, discussed during code review?

RQ2: To what extent these common topics can be automatically classified?

RQ3: How do the developers address the review comments that reviewers provide to revise their code changes?

With respect to RQ1, we analyzed the code review repositories of four open source projects, namely *eclipse*, *mylyn*, *android* and *openstack*, which are archived in *Gerrit*. We manually investigated the three most relevant attributes of information from *Gerrit*: the patch description as it contains the reason the patch is submitted, the changed lines of code by the developers and the review comments from the reviewers for the proposed code changes. We classified each of the review comments into the most appropriate topic based on the taxonomy of Mantyla et al. [16]. Our results indicate that topics related to both external and internal qualities are found during the code review; however, those related to the internal quality are dominating. As a prime example, 75% of the defects identified during code review are evolvability type of defects and hence the majority of the comments raised by the reviewers are focussed on improving the internal software quality.

With respect to RQ2, we present an automatic approach to identify the topics found in code reviews. Our approach builds a machine-learning-based classifier to automatically categorize the code change into its appropriate topic. The results of our automatic classifier suggest that evolvability type of issues in a code can be well predicted with an average precision and recall of 0.45 and 0.41 respectively. Although, Mantyla et al. [16] proposed the taxonomy, they did not offer any automatic solutions to classify topics or issues typically identified during the code review. We also discuss application scenarios of this automatic classification (see Section IV).

With respect to RQ3, we investigated the review comments from *mylyn*, *eclipse* and *android*. We manually studied the changes in the code for the review comments raised by the reviewers. As every review comment corresponded to a specific topic, we were able to correlate the action of a developer to a particular topic. We observed that developers adopted different refactorings to address different defect topics in code review. We investigated the structural changes that took place in different revisions of code review as a result of review comments and observed that developers did perform refactoring to address the review comments. By comparing those refactoring techniques with the respective topics, we observed a substantial congruence between the topics of feedback from reviewers and the specific (categories) of refactorings developers performed to address the review feedback in revising their code changes. Refactoring is a mechanism to improve the internal design of the software. Therefore, it is evident from our results that code review is effective in improving the internal software quality.

II. Background on MCR and Taxonomy of Topics

We define the key concepts involved in the modern code review process, which is driven by supporting infrastructure and tools, e.g., *Gerrit*.

Code Change: A code change is a set of modified source code files submitted in order to fix a bug or add a feature.

Patch Description: A brief information about the patch and the reason it is submitted. For instance, it contains information about the bug id from *Bugzilla*, if the patch is submitted to address a particular bug.

Review: A code review is a record of the interactions between the owner of a change and reviewers of the change including comments on the code and signoffs from reviewers.

Owner: An owner is the developer who makes the change in the source code and submits it for review.

Reviewer: A reviewer on a particular review is a developer who is assigned to and/or contributes to that review.

Review Comment: A review comment is textual feedback written by a reviewer about the code change during the review process. A review comment may be about the change in general or may be explicitly tied to a particular part of the change called the in-line comment.

The life-cycle of a review is as follows: Initially a developer (owner) makes changes to the source code in response to a bug report or feature request. Once complete, they submit the code change for review. The owner may indicate the intended reviewers, who are subsequently notified about the review invitation. It should be noted that the invited reviewers do not necessarily accept the invitation and contribute to the review. Reviewers then inspect the change through the code review tool (a web page in the case of *Gerrit*) and provide feedback in the form of review comments to the owner. The owner may update the change and submit the update to the review as a result of such feedback. The code change is typically depicted by showing the difference of the code before and after the change. Eventually, a reviewer signs-off on the review, once they believe the code change is of sufficient quality to be checked into the code repository. If a change never received sign-off, it is abandoned. It is critical that code review is both effective (actually improves code changes and blocks poor code from being checked into the repository) and timely (does not act as a bottle-neck too by slowing down changes). Therefore, automation and tool support are key to its success.

Mantyla et al. [16] divided defects detected from code review into the following parent groups: evolvability, functional, and false positives. Evolvability defects are sub-divided into Documentation, Visual Representation, and Structure. The Functional category has seven groups: Resource, Check, Interface, Logic, Timing, Support and Larger defects. Each of these defect categories have a definition as defined by Mantyla et al. and C. Bird et al. in their works [16] and [17] respectively. We have used the taxonomy proposed by them to classify the reviews in our dataset. Mantyla et al. [16] have broadly referred to all types of issues found in code during code review as defects, irrespective of whether it is an external defect affecting the functionality of the software or an internal design flaw affecting the internal quality of the software. For consistency and simplicity, we adopted the same terminology. We have referred to all types of issues found by the reviewers as different types of defects or topics found in the code.

We define an evolvability defect as a defect in the code that makes the code less compliant with standards, more error-prone, or more difficult to modify, extend, or understand. The functional defects are those that cause a system failure or fail in their business logic. False positives are those class of defects which were initially suspected to be defects but later on were discovered to be as no defects during team meetings. Each category is further divided into sub-categories.

III. Empirical Study: Formulation of Benchmark

The purpose of this study was twofold: 1) to determine which specific categories were prevalent in the open-source systems under study, and 2) to curate a benchmark to assess our automatic approach for classification (see Section IV). There is no established dataset nor benchmark for our context in the literature. Additionally, our effort can be considered as an independent empirical verification or replication of the categories of Mantyla et al. [16]. That is, we address:

RQ 1: What are the common topics, related to both internal and external qualities, discussed during code review?

Dataset and Methodology: For our study, we collected the patches from four open source systems namely *eclipse*, *mylyn*, *android* and *openstack* between the periods Jan 2014 and Feb 2016 and classified the review comments into either of the categories proposed by Mantyla et al. [16] and Bosu et al. [17]. Each patch has a brief textual description called the patch description, the owner who submitted it, the files that are modified as part of the patch, the list of reviewers selected to review the patch, the line of code that changed and the review comment that the reviewers made on the changed line of code. In order to create the benchmark, we considered the following patch selection criteria: 1) patches that were merged or abandoned, 2) that had either in-line or general review comments from the reviewers. We also considered those patches that had a relevant bug id in bug tracking system like *Bugzilla* to better understand the patch and thereby classify the review more accurately. Review #22722[1] from *Mylyn*, is an example of how we classified an individual review into its appropriate defect category. The reviewer *Sam Davis* commented on the changes in the file *BugzillaRestPostNewTask.java*: *This is creative but I'd rather use ImmutableList from Guava.* The owner *Frank Becker* had used an ArrayList and the feedback suggested the use of ImmutableList instead. It is evident from the changed line of code and the keywords in the review comment, namely, ”use ImmutableList” that the reviewer is proposing an alternate approach to the ower's solution. As *Mantyla et al.* mentioned in their work [16] that the comments that suggested function call changes or a complete rethinking of the current implementation, belonged to *Solution Approach* defect category. We classified this review into *Solution Approach* category. In another example, review #52373, the reviewer commented *When you copy a big chunk of code like this, it would help to add a comment saying where it's copied from because it's a sign that we might want to create a common implementation in the future.* Similarly, in this case, after inspecting the changed line of code and the keywords from the review comment, namely, ”add comment” suggest that reviewers are concerned about the documentation. Hence, we categorized this review into *Documentation* category.

Results: While consolidating the results of our manual analysis, we observed a total of 17 defect categories which covered the majority of the defects identified during code review. They are: Check Function, Check User Input, Check Variable, Compare, Compute, Data and Resource Manipulation, Wrong Location, Algorithm/Performance, Organization, Parameter, Solution Approach, Support, Supported by Language, Textual, Variable Initialization, Visual Representation and Compiler Error. Table I shows the distribution of defect categories within and across projects. They show that topics across evolvability and functional categories are found in code review. They cover both external and internal aspects of the reviewed code. It is also evident from these results that 75% (352 out of 468 review comments) of the defects identified during code review are evolvability type defects. Overall, we see that the internal quality aspects are dominant.

IV. Approach: Automatic Classifier

As discussed in the previous section, the review topics pervade both external and internal software qualities; however, identifying them is non-trivial, tedious, non-scalable, among other things. Therefore, we need to consider their automatic identification. That is, we address:

RQ2: To what extent can we automatically classify the common topics?

[1]https://git.eclipse.org/r/#/c/22722/4

TABLE I
Defect category/Topics commonly found in each project

	***Eclipse Platform*(86)**		***Mylyn*(108)**		***Android Platform*(98)**		***OpenStack*(177)**	
Defect categories	**# of reviews**	**%**	**# of reviews**	**%**	**Total No. of reviews**	**%**	**# of reviews**	**%**
Evolvability Defects								
Textual	39	45.35	25	23.36	38	38.8	51	28.8
Supported by language	8	9.3	7	6.54	10	10.2	8	4.54
Organization	6	7	14	13.08	8	8.16	36	20.45
Solution Approach	9	10.50	27	25.23	9	9.18	18	10.22
Visual Representation	8	9.3	3	2.8	18	18.37	9	5.11
Functional Defects								
Compare	3	3.48	4	3.73	1	1.02	3	1.7
Compute	6	6.97	2	1.86	5	5.1	6	3.4
Check Function			9	8.41	3	3.06	33	18.8
Check Variable	4	4.65	11	10.28	6	6.12	8	4.54
Check User Input								
Algorithm/ Performance							1	0.56
Wrong Location	1	1.16	3	2.8				
Data and Resource Manipulation	1	1.16						
Variable Initialization			1	0.93			2	1.13
Parameter			1	0.93				
Support			1	0.93				
Timing							1	0.56
Compiler Error	1							

Methodology. We developed a classifier to automatically categorize the reviews into appropriate defect categories or topics, using natural language processing and machine learning. The patch description contains information about the code changes (i.e., patch) the developer submitted for review. The lines of code suggest the changes in the source file that were submitted as part of the patch. The review comments contain the textual feedback the reviewers provide on the code changes. We considered these three features from the code review repository to train our model.

Each review comment along with its patch description, line of code and the respective category (label) was considered a document. The patch description and review comment of the document were preprocessed by removing the stop words and stemming. We did not perform any processing of the line of code feature because they were programming syntax and we had to preserve the information as it is required to accurately identify the defect category. After preprocessing, we performed the term-weighting where we produced a dictionary from all of the terms in our document and assigned a unique integer Id to each term appearing in it using the *tf-idf* metric.

The model was trained on the 7 most common defect categories namely Visual Representation, Supported by Language, Solution Approach, Textual, Logic, Check and Organization because our dataset did not have enough samples for all of the 17 defect categories to train our model. The labels (topics) for each review were derived from our manual investigation (see Section III). Of the 468 review comments from our manual investigation across subject systems combined, we considered 240 review comments from three open source projects namely *eclipse*, *mylyn* and *android* to train our model.

After the model was trained, we tested it with our test data that consisted of 50 test cases. The test data consisted of only the patch description and the line of code that changed. The reason for that was to not include any forward looking information as they would invalidate the results. For example, review comments are available once the code review is already under way. Therefore, there might be very little benefit in predicting the review topics at that stage. We would like to predict the topics as early as possible and with as little information as possible. Similar to our training data, the test data also underwent preprocessing like removing stop words and stemming of the patch description. We fed the processed training data to our classifiers and predicted its performance on test data. Once the automatic classification for the test data was performed, its predicted label was compared with the identified label and the accuracy was estimated. We adopted three different machine learning algorithms that are commonly used for text classification: KNN (K Nearest Neighbors) with a K value of 7, Naive Bayes and Support Vector Classification. We observed that KNN performed the best.

Results: With KNN, we observed an accuracy of 20% and an average precision and recall of 0.45 and 0.41 respectively. Table II shows the results of our automatic recommendation model. Our results indicate that we were able to predict both evolvability and functional types of defects. Also, we observe that the evolvability categories namely *Textual*, *Organization*, *Visual Representation* and *Solution Approach* have much more promising levels of precision and recall. Therefore, we surmise that automatic topic classification holds a much better promise in internal quality than external quality topics. Our effort is a first step in automating the topic identification as soon as a code change is submitted for review.

TABLE II
RESULTS OF AUTOMATIC DEFECT CLASSIFICATION

Defect categories	Precision	Recall
Textual	0.73	0.50
Supported by Language	0.00	0.00
Organization	0.31	0.50
Solution Approach	0.27	0.80
Visual Representation	1.00	0.50
Check	0.00	0.00
Logic	0.60	0.38
Avg/ Total	0.45	0.41

Application Scenario: This investigation also gave us an insight on the reviewers' expertise in identifying a specific type of defect. Table III shows a list of all the reviewers that participated in reviewing the 108 review comments from *mylyn* project and the number of defects they identified under each category. It is evident that *Sam Davis* has actively participated in the code review process and he is more experienced in identifying defect categories like *Solution Approach* and *Textual*. On the other hand, with *Sam Davis* as the owner for two large patches #47888[2] and #61091[3] of the total 20 patches that were investigated in *mylyn*, there were only 2 defects identified on his patch which were of the type *Visual Representation* and *Check*. This suggests that active participation in code review helps in building knowledge, improves the quality of the developer and thereby significantly reduces the defect likelihood in one's code. This table also shows that many reviewers did not contribute enough during the code review process. However, this can be prevented by recommending appropriate reviewers i.e. reviewers who are capable of identifying specific defects in a patch, based on what defect types the patch is prone to.

Other application scenarios include prioritizing the code changes (patches) based on the topics of concern, predicting their acceptance likelihood and completion time, topic-specific knowledge transfer, and overall project's development and maintenance status and overall maturity.

[2]https://git.eclipse.org/r/#/c/47888/
[3]https://git.eclipse.org/r/#/c/61091/

TABLE III
REVIEWERS AND CONTRIBUTED TOPICS IN MYLYN

Reviewer	Textual	Organization	Visual Representation	Solution Approach	Supported by Language	Check	Logic
Sam Davis	17	13		27	4	16	9
Steffen Pingel	2	1		3	1		
Landon Butterworth	3						
Frank Becker			1				
Doug Janzen					1		
Colin Ritchie	2				2		
Blaine Lewis	2		2		1		

Discussion: We discuss the evolution of our automatic classifier. We made several attempts to build a reasonably strong model. In our first attempt, we considered the data collected from *eclipse*, *mylyn* and *android* projects from our manual investigation and considered all the 17 defect categories that were identified during manual analysis. The training data set consisted of 240 review comments and the test data set consisted of 50 review comments. To build our classifier, we chose the following features from *Gerrit*; patch description, file name as it would be a good analysis to see the defect type with respect to the file, owner name as it would give information about the quality of the developer, line of code that was changed in order to address the bug or implement a new feature and the review comment written for the specific line of change in the code. We processed the dataset to remove the stop words and stemming and then trained our classifier to predict the defect categories of the test dataset. We used KNN, Naive Bayes and Support Vector Classification for our classifier to automatically recommend the defect category. Our first attempt was not successful. The accuracy came out to be as low as 4%. The main reason was the insufficient dataset. The samples for each defect category in the dataset were not sufficient. Hence the model could not be trained well.

Because the size of dataset was small and the number of defect categories was large, for our second trial, we reduced the total number of defect categories by combining a few of the similar categories and eliminating some of the rare defect categories. We eliminated those defect categories that had very few occurrences in the entire dataset. For instance, we combined the categories *Check User Input* with *Check Function* as they are similar. Check Function checks for the return value of a function and Check User Input asks for a test case to verify the return value of a function. Similarly, we combined *Data and Resource Manipulation* with *Variable Initialization* as both of them are related to Resource management. The *Compiler Error category* is a very rare scenario because it is highly unlikely for one to submit a file with compilation errors for code review because of the automatic pre-checks typically in place. Only one review comment belonged to the Compiler Error category of all the 468 review comments that were investigated. Hence, we eliminated this category in the second round. In total, we considered a list of 13 defect categories for the second trial to train our model and they are as follows: *Check Function, Check Variable, Compare, Compute, Organization, Parameter, Resource, Solution Approach, Supported by Language, Textual, Timing, Visual Representation, and Wrong Location*. We considered the same set of features as in our first trial (namely patch description, owner name, file name, line of code and review comment). The training and test datasets were the same from the first trial (i.e 240 and 50 respectively). The only difference was with the number of target defect categories on which the model was trained. This time we obtained an accuracy of 6%. The accuracy was still low because of the small dataset and also the noise introduced in the dataset in the form of file and owner names.

After these two unsuccessful trials, we realized that there was a need to further reduce the total number of defect categories. This time we considered only the most common defect categories for our classifier. We chose only those categories that were identified to be more prevalent in code review. In other words, we chose those target categories that had enough samples in our dataset. We also removed the features like file name and owner name which were mostly acting as noise in our dataset. In our third trial, we eliminated those features and considered only the patch description, line of code and review comment to train our classifier. This was the trial that gave a significant improvement in the accuracy, precision and recall and it was discussed in detail in the previous section.

For our next trial, we considered the data from all the four projects. The training data consisted of 360 records and the test data consisted of 108 records. We considered the same features as 22 our last trial (patch description, line of code and review comment). But this time, the average precision and recall fell from our previous attempt. Upon analyzing the results, we realized that the low accuracy in general was due to the following reasons. Firstly, there was insufficient data set and insufficient samples for each defect category. Secondly, the results of the manual investigation could not be verified by the concerned developer or the reviewer, hence the accuracy of the manual investigation results, on which the automatic classifier was built, could not be verified. Thirdly, our model provides good precision and recall to identify evolvability defects; however, there are not sufficient features in the code review tool to help us perform automatic classification of functional type of defects effectively. Another important reason is the difference in the line of code feature which is an important feature in our automatic model. *openstack* is a completely different project with python as its programming; *eclipse* and *mylyn* use the same programming language which is java, whereas *android* has both C++ and Java code.

V. Impact on Revising Code Changes

We wanted to investigate how developers receive the review feedback and what actions they take to improve their code changes. Was there any relationship between the topics they were critiqued on and the corresponding actions they took to resolve them. That is, we address:

RQ 3: How do the developers address the review comments that reviewers provide to revise their code changes?

By analyzing the code review comments, we identified that reviewers are more inclined towards identifying the internal design flaws in the code as it is evident from our manual investigation that 75% of the defects identified by the reviewers are about the internal quality of the code. As a next step, to validate these results, we wanted to study the steps that were taken to address those review comments. We carefully studied each revision of the patch, analysed the review comment and therefore the underlying defect category and observed the developer's code change so as to address that defect for the following projects: *eclipse*, *mylyn* and *android*.

From our empirical study, we observed that developers incorporated specific refactorings that are congruent to the defect category of the review comments. For instance, in the review #22719 [4], the reviewer *Steffen Pingel* suggested in his review comment that a class be split into a separate class. This comment is clearly about an issue in the organization of the code as it is about rearranging the code such that the software is more comprehensible and maintainable, and hence can be categorized as *Organization*. If we observed the next revision of the patch, we see that the developer has extracted that part of the code and created a separate class as suggested by the reviewer. In this case, the developer has incorporated *Extract class* refactoring technique.

Similarly, in another review #60407 [5], the reviewer suggested a naming issue in a developer's code. This defect can be compared with the Textual defect category where emphasis is given to proper naming or comments in the code which otherwise can cause misleading information. In the next revision of the changed code (patch), we observed that the developer had changed the name of the method as per the reviewer's comments. This action can be compared to the *rename* refactoring method as it is about renaming a class or a variable or a method in order to make its purpose clear.

It can be seen from our observation that majority of the defects identified during code review are evolvability type of defects and in order to address those, developers adopted refactoring techniques. Since refactoring is a mechanism to improve the internal structure of the code, it is evident that code review is useful in improving the internal software quality or the evolvability of the software. Table IV shows the mapping between various defect topics and the refactorings adopted by the developers for each of those topics.

TABLE IV
Mapping between topics and refactorings

Defect categories	Refactorings
Textual	Rename method
Organization	Extract method, Extract class, Move method
Solution Approach	Substitute Algorithm
Supported by Language	Hide Method

VI. Threats to Validity

We discuss internal, construct, and external threats to validity.

Misclassification of review comments: The results of the manual investigation could not be verified by the original developer/reviewer or other proficient software engineers thereby raising the risk of researcher's bias.

Heterogeneous Dataset: Although all the open source projects that we considered for our research uses the same code review tool and the mechanism, the projects themselves are different in nature (e.g., their main programming languages).

[4]https://git.eclipse.org/r/#/c/22719/1
[5]https://git.eclipse.org/r/#/c/60407/3

The line of code is a primary feature in our classifier, which could have impacted our results.

Insufficient Dataset: Since manual investigation is a tedious process, we could not gather enough data for all the defect categories. Hence our dataset was relatively small.

Insufficient Features: Code review tool has possibly insufficient features for our automatic recommendation model to classify functional defects effectively.

Generalization: Although we investigated four open source systems, we do not claim that our results would generalize to every single software system.

VII. Related Work

There have been many efforts on studying the effectiveness of code review in improving the external quality. However, very few efforts have been done to emphasize the importance of code review in improving the internal software quality.

Siy and Votta [15] proposed that 75 percent of the defects found during code reviews are evolvability defects that affect the evolution of the software instead of runtime behavior. C. Bird et al. [17] in their work, identified the factors that led to useful code review. They investigated the usefulness of code review by performing an empirical study in Microsoft projects, built and verified a classification model that can distinguish between useful and not useful code review feedback. Recently, McIntosh et al. [2] empirically showed that that poor code review negatively affect software quality. In another study, McIntosh et al. [18] reported that the percentage of reviewed changes a code component underwent correlates inversely to its chance of being involved in post-release fixes.

Rigby et al. [19] examined two peer review techniques: review-then-commit and commit-then-review used by Apache server project. They measured the frequency of reviews, the level of participation in reviews, and the size of artifacts under review in their studies. Beller et al. [20] found that the types of changes due to modern code review in Open source software are similar to those in the industry and academic systems from literature, featuring a similar ratio of maintainability-related to functional problems. Kemerer et al. [3] showed that code review reduces the amount of defects in student projects. With the available data they were also able to study the impact of review rate on the inspection performance. They found high review rates (i.e., a high number of reviewed LOC/hour) to be associated with a decrease in inspection effectiveness.

VIII. Conclusions and Future Work

We conducted an empirical study on the types of topics in the reviewers' feedback provided to developers on their code changes. Four open source systems were the subject of this investigation: *eclipse*, *mylyn*, *android* and *openstack*. Furthermore, we presented an automated approach to predict potential topics of reviewers' feedback as soon as a developer submits their code changes for review. Lastly, we also examined the impact of these review comments on the revisions that developers perform on their changes. We found that topics relevant to both external and internal code qualities are discussed in code review; however, those on the internal quality are dominant. Also, developers use refactorings to address those topics. The specific refactorings seem to align with the specific nature of review feedback topics. In summary, we provide evidence of the benefits of code review on internal code quality. Our future work will be directed on improving the automatic detection of these topics (e.g., accuracy) and developing their applications to further empower the peer-code-review process.

To facilitate replication, among other things, we provide access to our online appendix http://serl.cs.wichita.edu/codereview/topicmodel.

References

[1] B. Boehm and V. R. Basili, "Software defect reduction top 10 list," *Computer*, vol. 34, pp. 135–137, Jan. 2001.

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

[3] C. Kemerer and M. Paulk, "The impact of design and code reviews on software quality: An empirical study based on psp data," *Software Engineering, IEEE Transactions on*, vol. 35, pp. 534–550, July 2009.

[4] O. Laitenberger, "Studying the effects of code inspection and structural testing on software quality," pp. 237–246, IEEE, 1998.

[5] R. S. Arnold, "Software restructuring," vol. 77, pp. 607–617, Apr 1989.

[6] "Refactoring: Improving the design of existing code," (Boston, MA, USA), Addison-Wesley Longman Publishing Co., Inc., 1999.

[7] N. Gorla, A. C. Benander, and B. A. Benander, "Debugging effort estimation using software metrics," vol. 16, pp. 223–231, Feb 1990.

[8] W. Li and S. Henry, "Object-oriented metrics that predict maintainability," vol. 23, pp. 111 – 122, 1993.

[9] R. J. Miara, J. A. Musselman, J. A. Navarro, and B. Shneiderman, "Program indentation and comprehensibility," vol. 26, pp. 861–867, Nov. 1983.

[10] P. W. Oman and C. R. Cook, "Typographic style is more than cosmetic," vol. 33, (New York, NY, USA), pp. 506–520, ACM, May 1990.

[11] H. D. Rombach, "A controlled expeniment on the impact of software structure on maintainability," vol. 13, (Piscataway, NJ, USA), pp. 344–354, IEEE Press, Mar. 1987.

[12] T. Tenny, "Program readability: procedures versus comments," vol. 14, pp. 1271–1279, Sep 1988.

[13] C. Bird and A. Bacchelli, "Expectations, outcomes, and challenges of modern code review," in *Proceedings of the International Conference on Software Engineering*, IEEE, May 2013.

[14] P. C. Rigby and C. Bird, "Convergent software peer review practices," in *Proceedings of the the joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC/FSE)*, ACM, 2013.

[15] V. Balachandran, "Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation," in *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pp. 931–940, 2013.

[16] M. Mantyla and C. Lassenius, "What types of defects are really discovered in code reviews?," vol. 35, pp. 430–448, May 2009.

[17] A. Bosu, M. Greiler, and C. Bird, "Characteristics of useful code reviews: An empirical study at microsoft," in *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pp. 146–156, May 2015.

[18] R. Morales, S. McIntosh, and F. Khomh, "Do code review practices impact design quality? a case study of the qt, vtk, and itk projects," in *Proc. of the 22nd Int'l Conf. on Software Analysis, Evolution, and Reengineering (SANER)*, pp. 171–180, 2015.

[19] P. C. Rigby, D. M. German, and M.-A. Storey, "Open source software peer review practices: A case study of the apache server," in *Proceedings of the 30th International Conference on Software Engineering*, pp. 541–550, ACM, 2008.

[20] M. Beller, A. Bacchelli, A. Zaidman, and E. Juergens, "Modern code reviews in open-source projects: Which problems do they fix?," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pp. 202–211, ACM, 2014.