Code reviewing reviewed: recommendations for improving the efficiency and effectiveness of modern code reviews

Chun Fei Lung
Open University of the Netherlands
Heerlen, The Netherlands
cflung@acm.org

ABSTRACT
Code reviews are a way to maintain software quality. This article summarises two studies on modern code reviews: The first studies the effect of code review coverage and participation, and finds that the number of post-release defects is negatively correlated with both coverage and participation. The second studies the perceived usefulness of review comments within *Microsoft*, and finds that reviewer experience, and the size and composition of review requests affect the useful comment density. These findings are discussed, and a suggestion for future work is made.

Keywords
Code review, software economics, software evolution, software maintenance, software quality

1. INTRODUCTION
Many software projects make use of version control systems to keep track of changes. Changes to the source code are typically made in feature branches and reviewed by peers. Only when the changes are approved can they be integrated into the release branch.

Reviews are a relatively simple way to maintain software quality, and therefore make for an interesting research topic. For instance, McIntosh et al. [4] have studied how such modern code reviews influence software quality in open source projects, while Bosu et al. [3] have studied the perceived usefulness of review comments within *Microsoft*.

The remainder of this article is structured as follows: Section 2 describes the study conducted by McIntosh et al., while section 3 describes the study by Bosu et al. Section 4 discusses the results and poses a new research question.

2. PREVENTING POST-RELEASE DEFECTS
McIntosh et al. [4] hypothesise that modern code reviews that are not comprehensive enough will negatively impact software quality in the form of defects in the released version. The authors aim to disprove this hypothesis by addressing two research questions:

1. Is there a relationship between code review coverage and post-release defects?
2. Is there a relationship between code review participation and post-release defects?

2.1 Method
McIntosh et al. answer the research questions by performing a case study on three projects: *Qt*, *VTK*, and *ITK*. These projects were chosen because they are large, successful open source projects for which the majority of code changes were systematically reviewed using the *Gerrit* code review tool[1].

2.1.1 Data extraction
For each project the reviews are first extracted from *Gerrit*, while code changes ("patches") are extracted from the version control system. Only changes that happen on or have been merged into the release branch are considered for the case study: these are more likely to be carefully reviewed.

Furthermore, the authors identified a set of common metrics that are known to affect defect-proneness of code. Such metrics include the lines of code and cyclomatic complexity, the amount of change that happens during a release, and human factors such as developer expertise and code ownership. These metrics are calculated for each proposed code change, so that they can be controlled for during analysis.

Finally, it must be possible to determine how many post-release defects have occurred. These are found by retrieving all commits made within six months after a release, and searching the commit messages for occurrences of keywords such as "bug" or "fix"; such keywords typically appear in commits that (are intended to) fix defects.

2.1.2 Model construction
The explanatory variables consist of two classes: metrics related to the proportion of code covered by reviews and metrics related to the proportion of code review participation. Multiple Linear Regression (MLR) models are used to determine which of these metrics influence the occurrence of post-release defects. A log transformation is applied on all values to reduce the impact of outliers.

To keep the models simple, the number of metrics needs to be minimised. A Spearman's rank correlation is used to find metrics that are strongly correlated with each other. Then, a Variance Inflation Factor (VIF) score is calculated for each of the metrics. A VIF score indicates the collinearity, i.e. the extent to which a metric highly correlates with other metrics. High-scoring metrics are removed from the model.

[1] `https://www.gerritcodereview.com`

2.1.3 Model analysis

Once the models have been built, the goodness of the fit is evaluated using the Akaike Information Criterion (AIC) and Adjusted R^2. These two methods take bias into account that can occur due to the introduction of too many metrics. χ^2 tests are applied to each model's values to determine which metrics do not provide a significant contribution to the model and thus can be safely dropped.

Finally, the impact of each code review coverage and participation metric on the number of post-release defects is studied by calculating the expected number of defects using the models. The resulting number of defects is always rounded up to the nearest non-negative integer, as fractional number of defects do not exist in practice.

2.2 Results

Both code review coverage and participation influence the number of post-release defects.

2.2.1 Code review coverage

McIntosh et al. found that components with a higher review coverage tend to have a lower number of post-release defects: the impact of review coverage is statistically significant for the *Qt* and *ITK* projects.

For instance, in the *VTK* project a code review coverage below 0.29 will typically result in at least one post-release defect, whereas in the *Qt* project a code review coverage below 0.6 is already likely to yield one or more post-release defects. A review coverage below 0.06 is expected to result in two post-release defects. It should be noted that the models never appear to expect more than two post-release defects.

The code review coverage does not *always* have a significant impact on the number of post-release defects. This leads McIntosh et al. to believe that there may be other factors that influence software quality. For instance, there were also components that had a review coverage of 1 – i.e. were fully reviewed – but still showed post-release defects.

McIntosh et al. therefore conclude that code review coverage is negatively associated with post-release defects[2], but that there are likely other factors as well.

2.2.2 Code review participation

Because the results for the first research question showed that a relationship exists between the proportion of reviewed changes and the number of expected post-release defects, that is a variable that needs to be controlled for. McIntosh et al. therefore choose to consider only those components that have a code review coverage of 1. For the *VTK* dataset this means that it is has become too small for statistical analysis; it is therefore omitted from the analysis.

The authors look at three key metrics: 1) the proportion of changes that have only been approved by the change author themselves; 2) the proportion of changes that were approved by *other* team members without any discussion whatsoever; and 3) the proportion of changes that were likely hastily reviewed, i.e. there is little time between the creation of the review request and the approval of the code change.

Most importantly, McIntosh et al. found that components with a *high* level of review participation tend to have a *lower* number of post-release defects. The opposite is true as well: components with *low* code review participation tend to have a *higher* number of post-release defects.

McIntosh et al. therefore conclude that that low code review participation negatively impacts software quality.

2.3 Limitations

McIntosh et al. mention three threats to validity.

Firstly, the study only focusses on three open source systems. While Gerrit was used for all three systems, only a part of the proposed changes were actually reviewed through Gerrit. This can be a threat to the *external* validity.

Secondly, the constructed models assume that all defects are equally severe, but this is clearly not the case. The authors do note that many tools offer ways to assign severity levels to issues: however, these assignments are rarely reliable. This may threaten *construct* validity.

Finally, the *internal* validity may be threatened due to the assumption that a review is rushed if a code change is approved in a relatively short time: it is still possible that reviewers may postpone reviewing a code change, and rush the review at a later time.

2.4 Conclusions

McIntosh et al. conclude that code review coverage possibly has a positive influence on software quality, while code review participation likely has a positive influence on software quality. Overall, they believe that the findings provide evidence that code review policies should ensure a sufficient level of code review coverage and participation.

3. PERCEIVED USEFULNESS OF REVIEWS

Whereas McIntosh et al. focussed on post-release defects, Bosu et al. [3] studied how change authors perceive usefulness of review comments on their proposed code changes.

Despite the fact that modern code reviews are generally less rigorous than formal code inspections of yore, developers still spend almost an entire day every week reviewing code written by their peers [2]. Bosu et al. therefore aim to "*identify the factors that impact the usefulness of code reviews, and to derive recommendations for effectiveness improvements*" [3]. This goal is decomposed into three research questions:

1. What are the characteristics of code review comments that are perceived as useful by change authors?
2. What methods and features are needed to automatically classify review comments into useful and not useful?
3. What factors have a relationship with the density of useful code review comments?

[2]The article actually states that coverage is negatively associated with *software quality*, but this is probably incorrect.

A three-phase study was conducted to answer these questions. Each phase is described in further detail below.

3.1 How change authors perceive usefulness

To discover which characteristics make review comments useful for developers, Bosu et al. first conducted an exploratory study, which consists of semi-structured interviews.

Participants were shown a series of comments that were made about one of their proposed changes, and asked to classify each comment as either "Useful", "Somewhat useful", or "Not useful" and explain their reasoning for the classification. Furthermore, the participant was asked to categorise the comment type, e.g. a request for documentation, report of a defect, or false positive.

Prior to the exploratory study, a number of pilot interviews were held with other developers to assess whether questions were clear and answerable within reasonable time. These pilot interviews were solely intended to improve the actual study: the results were not included in the analysis.

3.1.1 Results

Change authors generally classified comments that point out defects or edge cases where the implementation may not behave correctly as "useful". In the case of junior change authors, advice by senior reviewers on designs, which APIs to use, team conventions, and so on was also considered to be useful.

On the other hand, comments about issues that do not affect the functionality of the current version are merely found to be "somewhat useful". Examples of such issues include suggestions for alternative implementation methods or typical "nit-picking issues", such as indentation, naming, and style.

Finally, comments that falsely point out issues, point out issues which are not part of the reviewed change, contain compliments, or are questions to gain a better understanding of the implementation, are classified as "not useful".

3.1.2 Discussion

The classifications made by the change authors largely support findings from existing studies that modern code reviews are used to prevent defects and increase maintainability. However, knowledge dissemination does not seem to be valued as much at Microsoft.

Furthermore, the findings show that useful comments later often result in further code changes near the line where the comment was made, presumably in response to feedback given in the review.

Finally, Bosu et al. note that the status of a comment is a reasonable indication of its perceived usefulness: "Resolved" comments are usually found useful, whereas comments that are marked with "Won't fix" are often *not* useful.

3.2 Classification of comment usefulness

In the second phase Bosu et al. construct an automated classifier that retroactively distinguishes useful from not useful comments based on evidence found within review comments and any code changes that were likely made *because* of those review comments. This is a multi-step process.

3.2.1 Manual classification

Based on insights gained from the exploratory study with change authors, Bosu et al. manually classified an additional 844 comments from five projects within *Microsoft*: *Azure*, *Bing*, *Exchange*, *Office*, and *Visual Studio*. These projects all use the same in-house code review tool, *CodeFlow*, and were selected such that they differ in scope and type. Comments are now only classified as "Useful" and "Not useful", as "Somewhat useful" comments can still be considered as "Useful". This is presumably done to simplify classification.

The manual classification work was divided among the authors. However, 100 comments were randomly selected for classification by all authors. This allowed for calculation of the inter-rater reliability of these classifications using Fleiss' Kappa. The result was a κ value of 0.947, which indicates that manual classifications were done consistently among the authors and are therefore valid.

The comments classified by change authors and Bosu et al. are combined into an oracle that is later used to train the automated classifier.

3.2.2 Attributes of useful comments

Based on insights gained during the interviews with change authors and the manual classification, Bosu et al. identified a number of comment attributes which often only co-occur with useful or non-useful comments. Examples include the number of review participants, the number of comments, whether the change author replied to a comment, whether the comment resulted in a code change, and whether the comment has a positive or negative tone.

3.2.3 Attribute calculation

To calculate the effect that each of the identified attributes has, Bosu et al. determine for each comment whether it is a change trigger, which is when a code change occurs after the review and before approval of the patch, in close proximity to the line where the comment was made.

A classifier based on the multi-variate Bernoulli document model is used to keep track of keywords within review comments. To prevent the list of keywords from becoming too large and uninformative, review comments are first preprocessed by removing whitespace, punctuation, and numbers, lowercasing, and stemming. Furthermore, only keywords that appear in at least ten comments are considered.

The sentiment of review comments was automatically calculated using the *Microsoft Research Statistical Parsing and Linguistic Analysis Toolkit*[3] (MSR-Splat) service.

3.2.4 Classification process and validation

A classification tree algorithm is used to build a decision tree model for comment classification.

[3] `https://www.microsoft.com/en-us/research/project/msr-splat/`

The model is validated in two ways: Firstly, using 10-fold cross-validation, which is repeated 100 times. Secondly, the authors asked 5 developers to manually classify review comments that they had recently received on their proposed changes, and compared these classifications with those made by the model.

3.2.5 Results

Because evaluation of the automated classifier is not the primary goal of the study, Bosu et al. do not provide an exhaustive overview of the results. Instead they merely present a few interesting findings about the impact that certain review comment attributes have on their perceived usefulness.

Comments that trigger changes are likely to be found useful. The same is true for comment threads where the author does not respond – as this implies agreement with the commenter – unless the author explicitly responds using keywords such as "done" or "fixed". Comment threads with only one comment or participant are also more likely to be useful, as are comments which are change requests and comments which have a "Resolved" or "Closed" status. Finally, comments with a neutral or somewhat – but not extremely – negative tones are generally found to be useful as well.

The 100 10-fold cross-validations showed that the model has a mean precision rate of 89.1%, a mean recall of 85.1%, and a mean classification error rate of 16.6%. When compared to the manual classifications made by developers, the model has a 86.7% precision and 93.8% recall.

3.3 Empirical study of comment usefulness

Bosu et al. look at two factors that may influence the perceived usefulness of review comments: characteristics of *reviewers* and characteristics of *reviewed changes*. They examine the relation between these two factors and the comment usefulness density, which is derived from the classifications made by the decision tree model.

3.3.1 Factor 1: Reviewers

Reviewers who have made at least one *modification* to source code files they are commenting on, tend to have a higher proportion of useful review comments. Curiously enough, the total amount of experience with a source code file, i.e. the number of times the reviewer has previously made modifications to it, barely affects the proportion of useful review comments.

Similar results are found for reviewers who have *reviewed* a source code file they are commenting on at least once, although in this case Bosu et al. report an increase in the proportion of useful review comments of up to 80% when a reviewer has reviewed the same source code file around five times[4].

In practice, this means that developers at *Microsoft* have low proportions of useful comments in their first three months, as they are still learning the system's design and constraints, and show a gradual improvement up to the end of the first year, when it plateaus.

Overall, Bosu et al. conclude that developers who have experience changing or reviewing a software artefact *generally* provide more useful comments. Up and downs in useful comment density may still occur periodically due to circumstances unrelated to the reviewing process.

Another interesting observation is that while most review comments (76%) are made by reviewers from the same team as the change author, cross-team reviewers have a slightly higher useful comment density. The difference is very small however.

3.3.2 Factor 2: Reviewed changes

As the number of source code files included in a review increases, the useful comment density drops, as reviewers may overlook changes or spend more time asking questions about the implementation.

The type of files appears to influence the density as well: source code files appear to have a high useful comment density, whereas build and configuration files have a low proportion of useful comments.

3.4 Limitations

All studied projects were from the same organisation and reviewed using the same code review tool. Bosu et al. argue that this is not a major issue, as the studied projects are for different types of software and prior research shows that open source projects and commercial projects use similar informal review practices [3, 5].

Another factor that may have influenced the results is the quality of the constructed model: even though the precision and recall are high, the mean classification error rate is still around 15%. The authors do not believe that the error is systematic, i.e. consistently causes a skewing of results towards a particular outcome.

3.5 Conclusions

Based on the findings from the empirical study, Bosu et al. make the following three recommendations.

Firstly, experienced reviewers contribute much more useful reviews than inexperienced reviewers. Thus the latter should be involved as early as possible to let them gain that experience. Nevertheless, in order to maintain a sufficient amount of useful comments in the short term, at least one experienced reviewer should be involved.

Secondly, change authors should commit small and incremental changesets whenever possible, so as to improve the ratio of useful comments. Furthermore, care must be taken when non-code files are committed: change authors should help reviewers by providing additional explanation for the changes in such files.

Finally, if a component attracts a disproportionate amount of comments that are not useful (e.g. false positives, questions about the implementation), actions should be taken to make the implementation clearer.

[4]It should be noted that within *Microsoft* developers are not allowed to change code before they have reviewed it. This may have skewed the results somewhat.

4. DISCUSSION

Both studies are empirical in nature and provide insights into the value of modern code reviews from the perspective of product owners and software developers.

4.1 Similarities and differences

In essence, the study by McIntosh et al. on the impact of code review coverage and participation on the number of post-release defects is about optimising for the *effectiveness* of modern code reviews. In order to gain and maintain a high software quality level and prevent post-release defects as much as possible, they argue that all code changes must be thoroughly reviewed and discussed before integration into the main release branch.

In contrast, the study by Bosu et al. on what makes code review comments (perceived as) useful is about optimising for *efficiency* of modern code reviews. In the short term, this may lead to a streamlining of the software development process, as less man-hours are wasted on processing review comments that are not actually useful.

This sidesteps the impact on software quality however: for instance, the software developers interviewed by Bosu et al. did not find questions about the implementation for the purpose of knowledge dissemination to be useful, even though this is also one of the purposes of code review [1], as it may allow them to produce higher-quality code in the future. In the long term, a sole focus on achieving a high useful comment density may therefore have undesirable consequences.

Nonetheless, in business environments, practitioners such as software developers and software development managers will likely want to consider both the effectiveness and efficiency of their code review processes: ideally a sufficient number of experienced software developers should always be involved in reviews, and be given enough time to (mostly) comment on issues that can be directly resolved by the change author.

4.2 Limitations

The discussed studies have a few limitations that may have affected the findings and conclusions.

4.2.1 Small, non-representative samples

The projects studied by McIntosh et al. are primarily free and open source projects, whereas those studied by Bosu et al. are commercial and closed-source. In both cases, the projects that ended up being studied were chosen because they were easier to analyse.

A study by Rigby and Bird [5] suggests that many similarities exist in the code review processes used in open source and commercial projects. Bosu et al. argue that this mitigates the problem.

However, differences are likely to exist in other areas of the software development process. This can make it harder to generalise the findings by McIntosh et al. to commercial, closed-source projects with tighter time and budget constraints or the findings by Bosu et al. to open-source projects where deadlines are less strict and reviewers are "free" and work on the software voluntarily.

4.2.2 A good outcome – for whom?

McIntosh et al. have chosen to look primarily at the number of post-release defects as a measure of how effective review comments are: however, defect-free software might not always be the most important goal.

A similar observation can be made about the study by Bosu et al., who have chosen to classify a review comment as useful if the change author thinks it is useful. They claim that this is a sensible choice, because the change author is the only one who can accurately judge whether *they* found the comment to be useful.

However, it can also be argued that this is a somewhat limited view of "usefulness". This is especially apparent when a change author has to answer questions about their implementation, which is an action that indeed provides little benefit to the change author, but may be very beneficial for the development team and project as a whole.

4.3 Future work

In the latter case, it is clear that Bosu et al. have not yet taken into account the positive externalities that may be associated with knowledge dissemination via replies on review comments.

A follow-up study could therefore be conducted that answers the question "*how do questions posed in review comments about the design or implementation of a component affect the software development process in subsequent development phases?*".

It is possible that the presence of such questions may be correlated with a lower number of post-release defects or a shorter development time later, e.g. in subsequent sprints when the Scrum software process is used.

5. REFERENCES

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

[2] A. Bosu and J. C. Carver. Impact of peer code review on peer impression formation: A survey. In *Empirical Software Engineering and Measurement, 2013 ACM/IEEE International Symposium on*, pages 133–142. IEEE, 2013.

[3] 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 (MSR)*, pages 146–156. IEEE, 2015.

[4] 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*, pages 192–201. ACM, 2014.

[5] P. C. Rigby and C. Bird. Convergent contemporary software peer review practices. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 202–212. ACM, 2013.