Metrics for Gerrit code reviews

Samuel Lehtonen and Timo Poranen

University of Tampere, School of Information Sciences, Tampere, Finland
samuel.lehtonen@gmail.com,timo.t.poranen@uta.fi

Abstract. Code reviews are a widely accepted best practice in modern software development. To enable easier and more agile code reviews, tools like Gerrit have been developed. Gerrit provides a framework for conducting reviews online, with no need for meetings or mailing lists. However, even with the help of tools like Gerrit, following and monitoring the review process becomes increasingly hard, when tens or even hundreds of code changes are uploaded daily. To make monitoring the review process easier, we propose a set of metrics to be used with Gerrit code review. The focus is on providing an insight to velocity and quality of code reviews, by measuring different review activities based on data, automatically extracted from Gerrit. When automated, the measurements enable easy monitoring of code reviews, which help in establishing new best practices and improved review process.

Keywords: Code quality; Code reviews; Gerrit; Metrics;

1 Introduction

Code reviews are a widely used quality assurance practice in software engineering, where developers read and assess each other's code before it is integrated into the codebase or deployed into production. Main motivations for reviews are to detect software defects and to improve code quality while sharing knowledge among developers. Reviews were originally introduced by Fagan [4] already in 1970's. The original, formal type of code inspections are still used in many companies, but has been often replaced with more modern types of reviews, where the review is not tied to place or time. Code reviews discussed in this paper are tool assisted modern code reviews.

Reviews are known to be very efficient in defect detection and prevention especially in large software projects [10, 1]. For the review process to be successful, reviews have to be done carefully and without unnecessary delays. To foster that, review process has to be motivating for developers while also being monitored and controlled to ensure the review quality. In this paper we are proposing a way to increase monitoring possibilities and controls by introducing different metrics to follow activities that take place during the review process. The introduced metrics are used to examine data that Gerrit [5], a web based light-weight code review tool, automatically stores during different events in Gerrit code review. This information can then be processed and prepared to be presented as graphs or key figures.

Code reviews in Gerrit are an interesting research target because of Gerrit's current position as a popular review tool among some of the leading open source projects. It is used in projects like LibreOffice, Wikimedia, Eclipse and Android Open Source Project [5].

The areas of measurement include different phases of the review process and review outcomes. The biggest focus area is on measuring the time that is spent on each activity and monitoring the number of reviews that lead into further code changes. The data provided by the metrics also establishes foundations for possible future research on how to improve the review processes even further, by optimizing individual activities.

After analyzing the metrics results from four different projects, we identified typical time division in code reviews, which was similar in all of the studied projects. Also we identified that review results depend on the person who is making the review.

The rest of the paper is organised as follows. In Section 2 we give a deeper introduction to code reviews and effects of code reviews to code quality. New metrics for measuring Gerrit code reviews are given in Section 3 and in Section 4 these metrics are applied to sample projects and findings are discussed. Last section concludes the work.

2 Code Reviews

Originally the reason for code reviews was mainly to spot defects and prevent any bad quality code from getting into codebase. Bachelli and Bird [1] reveal that while finding defects is still the main reason and motivation for reviews, there are also many other benefits that act as motivation to conduct reviews. These include at least sharing of knowledge, improved communication among developers and better overall software quality.

Successful code reviews enable a product with less defects and improved maintainability [16]. Improvements are not limited to code, but the whole project may benefit from the reviews as programming practices and other ways of work are discussed and refined during the reviews. Future work becomes easier as the development becomes smoother. Reviews are also known to speed up the whole project at later stages. When reviews are done at early stage, for example, when 10 % of the project is complete, rather than later stage when a lot of work has been done already, the lessons learned in reviews let the rest 90 % of the work to be better executed. This also mean better integrity and any fixes are easier to apply and are less likely to mess up the code structure. [6, 18]

Code reviews are demonstrated to be efficient way of detecting defects, but every developer does not spot defects with the same efficiency. Rigby et al. [14] and Porter et al. [13] researched the number of defects found per developer and found that more experienced developers discovered more defects than less experienced developers, which is natural because senior developers have more experience with the program than junior developers. The amount of code reviewed also has a big effect on review results. Statistical analysis of code reviews by McIntosh et al. [11] shows that there is a significant correlation between code review coverage and the amount of post release defects. However, even with 100 % review coverage it is likely that some defects remain. On the other hand, when reducing the review coverage to around 50 %, it is almost certain that at least one defect per component will get through.

For addition to defect finding, code reviews have become an important knowledge sharing tool especially in projects where people are working in different locations. Reviews can substitute a huge amount of formal documentation what would have otherwise been needed to describe to other developers what changes have been made since the last version. Modern review tools like Gerrit are able to show all the changes made on a side by side comparison with color codes. When changes are seen by several developers, it increases their awareness and understanding of the software being developed. Communication between developers is also enhanced, especially when review tools are used, which make it easy for developers to leave inline comments or questions. As frequent communication is used to discuss problems, there is also better atmosphere to generate new ideas for implementing the code [Bachelli and Bird, 2013; Markus, 2009]. To further emphasize the importance of knowledge sharing during the reviews, the knowledge sharing and frequent communication among developers is known to improve the software quality. [12]

Because of the idea of constant feedback and sharing of written code with others, code reviews are also a powerful learning tool. New developers get feedback from more seasoned programmers which enable them to fortify their skills and to learn from each other's work. To summarize, code reviews can be considered as a very important defect detection tool and even more important when working with large projects, where software evolves and new versions rely on earlier work. Code reviews are considered to be the best tool for detecting evolvability defects while also being effective tool on finding functional defects.

Review processes in Gerrit [5] can vary between projects, but the basic workflow is always the same. There is the author of the code change, one or more reviewers and possibly approver and integrator. The most basic use case of Gerrit only involves the author and reviewer, but if we want to measure the process with better accuracy, at least integrator role is needed to give details about integration times. The approver is usually a senior developer who makes the final acceptance decision for the code change.

While the the metrics proposed in this paper are best used with projects which utilize approver and integration flags, it is also possible to use them without those roles, but the measurement results will be more limited.

Typical code review process proceeds as follows:

1. Author uploads the code change. This step could also include automatic unit testing.
2. When author considers that the change is ready, he/she assigns reviewers for the change and gives the change a code review with a score of +1.
3. Reviewers review the code and give it either +1 or -1. If the reviewer doesn't find anything to improve, a score of +1 is given. Whenever there is something

to improve, -1 should be given. The author has to address the issues raised by the reviewer before the process can continue.

4. When there are enough reviews, approver makes the final decision to pass the change for integration by giving it +1 or sending it back for rework by giving it -1. If approver is not used, the code change goes straight into integration.
5. After the change is approved, integrator starts the integration process and gives the change +1 to mark the beginning of integration. If the code cannot be integrated, the integrator gives -1 and sends the change back to author for rework. Any merge conflicts also result as integrator -1, which typically means that the change goes back to author for fixing.
6. If the change can be merged and passes final testing, it is merged into the repository.

3 Metrics for measuring Gerrit code reviews

The motivation to have quality and efficiency metrics is that without them, process improvement and identification of the best practices would be impossible. [Sommerville, 2011; Kupiainen et al., 2015] While Gerrit gives good tools to organize reviews, it only provides limited amount of metrics about the changes. To improve code reviews in Gerrit, sufficient metrics are essential. Measuring activities in Gerrit allows more control over the code reviews which helps managers to ensure steady flow of code changes. Although Gerrit doesn't provide metrics, it has all the necessary data for building the metrics stored in a MySQL database.

The goal of the metrics is to present the data extracted from the review process to developers and managers, in a way which helps them to monitor and control the review process, in order to pick out problems, but also to motivate for better performance. Based on the knowledge gained from the figures, managers and developers themselves could take action to improve the process or make corrective actions. Based on the knowledge gained from the metrics, it is possible to refine the review process for better overall software quality with fewer defects. Metrics also provide valuable information of the whole process, which supports the improvement of code review process in the company.

The measurements can be divided into two categories: time and quality. The time metrics are focused on measuring the velocity of different activities and the time spent for different tasks. The quality metrics are measuring various items which can hint on process quality and code quality. In the following the metrics are discussed in detail.

3.1 Review time

The review time is one of the most fundamental metrics we have. It describes how long it takes for the change to get the necessary amount of positive code reviews and an approval by the nominated approver. The review time starts when a developer has uploaded a change and has given it code review +1 in

Gerrit, to acknowledge that the change is ready to be reviewed. The +1 given by the author is also a sign that the author has tested the change and considers it ready to be submitted to the baseline. The +1 that the change owner gives is an agreed practice used in the case company. It is not a forced process by Gerrit, but a good way to communicate that the author approves the change ready for review. This is especially useful if continuous integration and unit testing is used alongside Gerrit.

The review time consists of the time it takes for reviewers to start the review after they have been invited combined with the time it actually takes to complete the review. Measuring the actual review time, meaning the time how long it takes for the reviewer to actually review the code, cannot be automatically measured and it has not been seen convenient to add an additional flag that the reviewer has to set before he or she starts to review the code, especially if the review is not completed within a session.

The review time also includes the time that the author uses for fixing the change whenever it receives a -1. There could be multiple fixing rounds until the change can be fully accepted by the reviewers. This loop continues until the change has received required amount of positive reviews without any -1's and is thought to be ready for merge. The final decision is done by the approver whose +1 also ends the review time. If approver is not used, integrator +1 could also mark the end of review time.

Review time is an important measure of how quickly new changes are ready for integration. Long review times slow down the whole process, especially when there are dependencies present. There is a couple of ways reducing the review time. The most obvious way is that the reviewers should review the change soon after being invited for a review. Another way to easily reduce the review times is to promote the communication during the code reviews so that everyone involved is aware when reviews or fixes are needed. In urgent issues it is best to send email directly or have talk face to face to get the things moving. Messages left to single change in Gerrit may remain unseen for a while or are easily forgotten when developers are simultaneously involved in numerous open changes.

The review time metric allows managers to follow review times and react to any longer than usual times. Long review times could be caused by a reviewer forgetting or neglecting the review, or delay may be due to an absence of a developer. If reviews pile up because of absence of some person, change author should nominate another reviewer, so that the process can move on. Long review times can also be a result of poor code quality which generates multiple -1's. Received reviews by a developer are also measured and that can be used to complement review times measurement, to better understand where the problem with long review times lies. Within the company it is useful to compare review times of different projects and discuss why some projects achieve better review times than others. Without these measures it would be inconvenient to try to determine whether reviews take reasonable amount of time or compare them with other projects.

3.2 Pre-review time

When the change is first uploaded to Gerrit the pre-review time starts. The purpose of measuring the pre-review time is to separate the time that is used by the author to fix the change after integration and tests from the time that is used to complete the code reviews. After uploading the change, the source is built and tested for errors by using automated tools. If the patch passes the tests it is possible to set the patch ready for review by giving it code review +1. Before giving +1 the author can still make changes by uploading a new patch-set to address any issues the author may have come across during the tests. Pre-review time ends at the moment the code review +1 is given by the author. This also marks the beginning of review time.

The pre-review time measures how quickly uploaded change is ready for review. Long pre-review time indicates that there has been need for rework after the initial upload and first tests. Typically this time should be less than a day and any longer times need a good reason and managers should take action whenever long pre-review times are noticed. The owner +1 is not a mandatory action in Gerrit, but is applied practice in the case company where the metrics were introduced. Although it is optional in Gerrit for the author to give +1, it is a good indicator that the author wants the change to be reviewed. It is especially useful if change is queued for tests or continuous integration after the upload. The author can then give the +1 when all the tests have fully passed. By separating these test activities and possible fixing to pre-review time, we can achieve more accurate review time.

3.3 Pre-integration time

The pre-integration time is the time after the review is completed, but the actual integration process is yet to start. This time should be minimized as it is time when no one is working with the change, but it is just waiting for integrator to start the integration process. Basically pre-integration time describes how long it takes for the integrator to start the integration process. The integration process should start as soon as the review is completed and approver has given +1. This means that the pre-integration time should not be long as there is no reason for the integrator not to start the integration. Typically the integration should be started the same day as the review is completed or the next day at the latest. The pre-integration time shouldn't therefore be more than 24 hours excluding weekends.

3.4 Integration time

The integration time is a measure of how long it takes for the change to be merged into the trunk after the review process is completed. Integration time includes various phases which take place during the integration. Depending on a project, integration time could include very different tasks. Validation processes and build processes depend on platform and the nature of code being tested.

Some changes may get integrated pretty fast, while in some other projects it is a slow process. Comparing integration times between projects is therefore not very useful. Comparison should rather be done within the same project.

At its best, case scenario integration starts almost immediately when the change has been approved. During the integration time the integrator tries to merge the patch with rest of the code and if there are no conflicts and the change passes tests, it can be integrated.

However, sometimes there are merge conflicts which need to be resolved before the change can be fully merged and tested. When a conflict appears, the integration process is stopped and integrator gives the change -1. If the conflicts are small, the integrator can then fix them and a review of changes by the change author is enough. If there are major conflicts then the change needs rework and it has to go through the review process all over again. Also if change cannot pass integration testing, it is returned to the author for rework. The time spent to rework can be measured in a different metric called integration fix time.

Ideal integration time is less than 24 hours, which means that the integration is done the next day at the latest, after the change has been approved. There should not be any excuse for the integrator not to start the integration as soon as the code change has been approved for integration. For meaningful results, integration times should only be compared within same project or within projects with similar workflows. This is because projects working with for example with application layer or firmware have very different validation processes during the integration.

Whenever the change gets rejected during the integration, the integration reject time meric can be used to calculate time wasted to rejected changes. It is simply a measurement of how long it takes, before the change is rejected after being approved by the approver. There are many reasons why a change could be rejected during the integration. The reason can be some known issue in the change which prevents integration or the integrator -1 could be a result from merge conflicts or failed tests. Any change receiving integrator -1 needs fixing. Often the reason is of merge conflicts. In a case of simple conflicts the integrator can fix them immediately, but most often the problems are more profound and the change is sent back to the author for rework. The time used for reworking with the change is another measure called integration fix time.

The reject time depends on at what point the integrator notices that the change cannot be merged. The most common situation is that the change is based on too old baseline which is no more compatible with the current codebase. This results as errors when merging or building. Typically integrator starts the build at the end of the day and reviews the results the next morning at the latest. Because of this, the reject time is typically around 12-60 hours including weekends. The integration fix time is the additional time that is used to fix the change after it has been rejected during the integration. This time includes the time the developer uses to actually fix the change, but also the time it takes to review and approve this new fix.

After measuring the reject time and fix time, it is clear that any rejected change becomes expensive as it takes many days for the change to be fixed and reviewed again. If the developer used few extra hours to rebase and validate the change to be ready for integration at the first place, it would reduce the overall lifetime by tens of hours at the minimum.

3.5 Quality metrics

The purpose of quality metrics is to accompany time measurements by giving a perspective on how code reviews are made. When review times of a project suddenly fall while reviews constantly result with only +1's, it hints for change in review motivation and should be investigated. To maintain good code quality reviews have to be performed with care. Reviews are successful when defects are found. However if someone's patches constantly gets a lot of minuses there might be a need for discussion with the developer why that is happening. Also if certain reviewer always finds something to improve, it is useful to investigate if all those minus ones are necessary.

Metrics were also established to ensure that the reviews are done properly. When schedules are tight and deadlines are closing in, it is tempting to skip the reviews. However, any time saved here could cause even bigger delays and costs later in the development lifecycle. To recognize if any team is trying to catch its schedule by hurrying or ignoring the reviews, a few different quality measurements were introduced.

The main motivation for quality measurements is monitoring that the reviews are conducted following the good code review practice. The measurements used are the number of reviews given and received. The motivation for not giving negative reviews is often a need to speed up the review process. When a project is in hurry it may be tempting to hurry the code reviews too or skip it altogether so that feature complete milestones can be achieved. Ignoring code reviews or testing is very shortsighted and it is likely that any schedule advantage gained by reducing reviews or testing, will later be lost because the end product is too buggy or hard to maintain. By measuring the review activity, problems like this are tried to be avoided. To complement these, there is also a metric to measure review results. Reviews should have quite steady amount of +1s and -1s. If some project starts to receive only +1's it might be better to investigate what is the reason.

3.6 Average number of patch-sets

One straight forward metric for measuring change quality is the number of patch-sets. In addition to change quality it can be used for measuring process smoothness. Smaller amount of patch-sets means that the code change has needed less fixes until it has been integrated in to the codebase. The number of patch-sets is measured with two different ways. There is the absolute number of patch-sets per change, which makes it possible to easily point out individual changes with larger than normal count of patch-sets. Then there can be a histogram chart which shows the distribution of number of patch-sets needed for a change. The histogram can quickly tell that the most of the changes go through with only one patch-set, which may be due to very trivial nature of the change or +1's are given too softly in code review. For better understanding these patches need to be investigated individually. By following this distribution over time, a normal value for each frequency can be established and big variances to that could then be a sign to start investigation. A good number of patch-sets is around 2-5, which implies that the initially uploaded patch did not need numerous fixes, but the reviewer has found something to comment or improve. As each patch-set represents an improvement to the earlier patch, this can be used to give an estimate on how effectively code reviews are used to improve the code quality. The interesting thing to follow are situations where change receives very few or large amount of patch-sets. If the change receives only one or no fixes at all, it indicates that reviewers may not have looked the code at all, or have decided not to give feedback on minor issues. It is rarely the case that the first version of the code change is perfect and there is no room for improvement.

On the other hand a change with a lot of patch-sets indicates that the change has needed many fixes and has probably been under development for a long time. This can be because the change is big and requires a lot of work phases or the change could be fundamentally problematic and needs a lot of fixes to get it working with the rest of the code. If the number of changes with large number of patch-sets increases, it should be investigated whether the changes are too complex and if they should be divided into smaller ones. Sometimes changes need to wait for other changes being finished first. For example, in the case of dependencies the change might need numerous rebases while it waits the parent change being finished. Rebasing means applying latest changes from the master branch in Git. This can happen multiple times and that increases the number of patch-sets although there is no quality issues whatsoever. Other explanation is that the change has needed many fixes and has been somehow problematic.

If there is always only one patch-set, it means that reviews may not have been done properly or with enough care. While the main motivation of code reviews is to find defects, it is also important to try to improve the code. The code is not improved if the reviewer doesn't provide any feedback on the proposed change. Therefore, it is positive to see more than just one patch-set. From the code improvement point of view, it may not be the ideal situation to have minimum number of patch-sets. On the other hand, if the amount of patch-sets is very high, it often means that the patch is too complex or that the programmer is not doing good enough job. Comparison of number of patch-sets should be done within a same project on weekly or monthly basis, as the amount of patch-sets vary a lot between projects due to the different nature of development. High patch-set count is a good indicator of change complexity as more complex changes typically require more revisions, which increases the complexity even more. The complexity should be reduced by making smaller changes, which would make reviews easier and dependent changes would also get merged faster.

An important metric for determining the state of the code reviews is the count of currently open code changes. The difference between opened and closed items gives the number of currently open items, which is a useful measure to get an insight of current pace of opening and closing changes. If number of open patches goes up for extended period of time, it immediately tells that there are some problems in the process which need attention. If open items pile up, they are harder to manage and dependency management and the number of rebases needed become an issue. This metric can be accompanied with a count of abandoned changes, which measures how many of the uploaded code changes are abandoned and not integrated.

3.7 Measuring given and received reviews

Measuring review results shows how much negative and positive reviews are given. As part of the motivation for code review is code improvement, it is important that changes receive also negative reviews. Whenever a change receives -1, it means that reviewer has noticed something to comment. Comments are usually a good thing, because that implies that the code is being improved and discussed. Only time when comments may be useless are false positives, where reviewer thinks something as a defect when it is not. When this was discussed with developers, the consensus was that false positives are very rare. Measuring the amount of positive and negative reviews given by developer is also an important measure from the quality control perspective, because if some reviewer gives constantly only +1's he/she may not review thoroughly or is too soft in giving reviews. It is unlikely that the changes are always perfect.

4 Findings

To understand how time is allocated on average between the four major activities in Gerrit code review process, data was collected from four different Gerrit projects with over 200 changes per month each. The project sizes varied from 30 to 100 developers per project. Each researched project had teams based in several locations around the world. More detailed illustrations of project data can be found from Lehtonen's [8] thesis.

The results gained from the four projects are shown on Table 1. The integration reject time and integration fix time are eliminated because the event of integrator intervention is rare and hardly ever happens in most of the projects. Project number one is suffering from a couple of changes where integration time is extraordinary long, which partly explains the share of integration time being 36 % from the change life time.

From the analysis of four projects with different number of people and different tasks it can be seen that the division of time spent is still quite similar in all of the projects. The times in each project are also fairly long at some stages, so there is clearly some room for improvement. The change stays in Gerrit for around one day before the author declares it ready for review. In the investigated

Table 1. Distribution of time in Gerrit code review. Times are hours with percentage of total change life cycle.

	Pre-review time	Review time	Pre-integration time	Integration time
Project 1	28 (10 %)	99 (36 %)	49 (18 %)	101 (36 %)
Project 2	30 (11 %)	166 (62 %)	28 (10 %)	44 (17 %)
Project 3	14 (7 %)	103 (54 %)	16 (14 %)	49 (25 %)
Project 4	14 (12 %)	119 (59 %)	20 (10 %)	38 (19 %)
Average	24 (10 %)	122 (53 %)	31 (13 %)	58 (24 %)

projects the actual review time took 36 – 62 % from the total time, including possibly multiple rounds of reviews, which makes it the most time consuming activity. This is no surprise as that is when most of the work during review process is done. However, the hours spent is currently quite a lot. As a percentage, the share of reviews could increase, as now around 10 % or 30 hours is spent during the pre-integration time, which is the time the change waits for integration to begin. This is long time, which partly is explained with people working in different time zones, but still there is clearly room for improvement. In integration times there is quite a bit of variance, which is because of the different nature of the projects and amount of required integration testing.

The integration fix time was excluded from the individual projects due to too small sample size, but when a group of projects was investigated where the total number of changes goes beyond 1000, the typical time needed to fix a change after rejection in integration, requires 100 – 150 hours, before it is again ready for integration. From that finding, it is safe to say that it is worth investing time to get the change ready for integration at the first place. The fix time quickly accumulates because developer needs time to do the fixing, but the change also needs a new round of code reviews and an approval.

Another place where time can be saved in researched projects, is in the code reviews. Average time for the code change to complete the code review stage is 122 hours, which equals to roughly 5 days. The time a reviewer actually uses to review the change is typically less than an hour, which means that the change waits several days for reviews to be completed. Even in the best of the researched projects, the average review time was 99 hours. When examining the frequency of minus ones in code reviews, the variance between projects is noticeable. The share of changes going through without any negative reviews is shown in Table 2.

Table 2. Percentage of changes which receive at least one code review -1.

	Share of rejects
Project 1	73 %
Project 2	62 %
Project 3	35 %
Project 4	27 %

The project which has received the most rejects in code reviews has almost three times the number of rejects than the project with least rejects. However, this does not always correlate with the number of patch-sets. For example, in a project where only 27 % of changes received a negative review, every change had two or more patch-sets, meaning that the fixes were self-motivated or due to comments given together with positive review, which is against the agreed practice. In the project where 35 % of the changes went through with at least one negative review, share of changes with only one patch-set was around 20 %.

The number of patch-sets is another quality metric and is good in pointing out complex code changes. In an optimal case the change receives two or three patch-sets which indicates, that there has been some improvements to the initial code change. If changes get continuously integrated with only one patch-set, it should be brought to discussion, because the idea of code reviews is to improve the code and when the first version of the code is always accepted, there will be no improvement. On the other hand if a change has much more patch-sets than two or three, it tells that the change has some serious problems, because so many fixes has been needed.

Beller et al. [2] reported a case where the figure of undocumented fixes by the author accounted only 10 – 22 % which is significantly less than in most projects that were researched. That might be due to different development processes, because in the projects the unit tests and continuous integration is done after the change has been uploaded into Gerrit.

A reason which may explain the numbers, is that the sizes of changes in the projects with high rate of zero negative reviews was often very small, 5-10 lines. This means that there is very little space for defects. To support this observation, the project that scored the highest number of rejections during the code review, also has considerably larger change sizes on average. Contrary to the findings by Beller et al. [2], who examined small software projects, it was found that reviewer do have significant effect on review results. Some reviewers truly are stricter than others. There are also other factors which influence the number of negative reviews, like cultural background. Beller's study is based on two small projects while our sample size is hundreds of employees working within one company, but in very different projects with their own ways of work.

5 Conclusions and discussion

Code reviews have been around for almost 40 years, but the practice is still evolving. Modern code reviews are not much studied and without many concrete best practices being established. In an effort to recognize new best practices for code reviews in Gerrit code review, this paper has studied code reviews in a case company and introduced different metrics for measuring the reviews in a number of projects.

The main measurement areas are the time spent in review process and the frequencies of various actions in order to establish foundations for measuring code review velocity and quality.

In this paper we have demonstrated that measuring code review process with various metrics, can help controlling the review process on whole, while enabling improvement of the review process. The metrics give valuable information to management and developers including how quickly new code changes are reviewed and how long it takes for them to become integrated. It was also examined what are the stages in the process that need the most improvement. The metrics are most useful in large projects where keeping track of single changes would be too laborious.

When places of delays are identified and other metrics like number of negative and positive reviews given are measured, it is possible for the management to improve the efficiency of the review process. In a case of long change life times the key is to identify the factors which slow the process down and apply corrective actions. By measuring the number of positive and negative reviews per developer, it is possible to monitor the review activity and quality. If changes constantly receive low amount of negative reviews management should encourage stricter review policies so that more negative reviews are given and the code gets therefore improved more. In addition, the number of given feedback can be monitored and if very little feedback is given, the reviewers are encouraged to give more comments on the code changes.

When reviews and testing are applied properly, the speed of development becomes the most important thing to measure, as in the end the ability to create software fast is the thing that allows to gain the competitive advantage in the industry. From these time measurements, the most interesting one measures how quickly new code changes can be integrated after being uploaded to Gerrit.

Measuring code quality is also very important as code with lot of defects, functional or evolvability will cause additional costs later in the lifecycle. These metrics do not directly measure the code quality. The focus is more on measuring areas which do have secondary effect on quality. These include the number of rejects during review and number of patch-sets. From these two measurements it is possible to draw conclusions of the initial quality of the changes. When there are a lot of patch-sets, it typically means problems. When compared with data of received reviews it is possible to tell whether the change was well crafted. A lot of negative reviews combined with long change lifetime is a signal of bad code quality.

5.1 Limitations

The usage of the metrics introduced here is partly limited to Gerrit and some of the metrics require certain configurations. However, especially the velocity metrics are quite generic, and can be also used with other code review systems. While the interface could be different, the basic principles are typically similar, with change author and reviewers. By using the time stamps from reviews it is possible to calculate at least pre-review and review times. The proposed quality metrics are more Gerrit specific, but the concept is not tied to any unique functionality and can be copied to be used with other systems as well.

The sample size in the findings is four projects, all from the same organization, which sets some limitations to the credibility of the study. On the other hand, the studied projects are largely self organized, working with different software components, which makes them heterogeneous. Therefore, the sample data can be considered diverse enough to validate the observations.

5.2 Further research

In addition to the measurement areas covered in this paper, there are number of other possibilities for measurements. The metrics that were presented do not go deep into the factors behind the results. There are, for example, many variables that affect the code review process and could be used in later studies to complement the proposed metrics.

Tomas et al. [17] list and discuss few of the variables in detail including code complexity, code dependencies, number and size of comments, amount of code smells and code design. These additional metrics can be used to give more detail on why reviews take certain time or how effective they are in improving the code quality. To get deeper understanding of the factors affecting the metrics results, the future work with the metrics could, for example, take the variables studied by Tomas et al. and combine them with the metrics introduced in this paper.

The metrics still do not measure how long it takes for the reviewer to start the review after the change has been made available for review. Neither do the metrics measure how long it actually takes to complete the review. This would require adding a new functionality to Gerrit where reviewer can set the review started. However this is a manual process and might be hard to measure accurately as it relies to the fact that reviewer always remembers to set review started. Also, if review is not completed at a one session, the measurement would be skewed. However, this is very interesting information and the possibility to measure this will be investigated. Another additional measure could be review start time. It would be a measure of how long it takes for the change to receive its first review. This would give some insight on how quickly developers are reacting to review requests as currently measured review time only measures the total time spent on reviews.

References

1. Bacchelli, A., Bird, C.: Expectations, outcomes, and challenges of modern code review. In Proceedings of the 2013 International Conference on Software Engineering (2013), IEEE Press, 712-721.
2. Beller, M., Bacchelli, A., Zaidman, A., Juergens, E.: Modern code reviews in open-source projects: which problems do they fix? In: Proceedings of the 11th Working Conference on Mining Software Repositories (2014), ACM, 202-211.
3. Cheng, T., Jansen, S., Remmers, M.: Controlling and monitoring agile software development in three Dutch product software companies. In: Proceedings of the 2009 ICSE Workshop on Software Development Governance, (2009), IEEE Computer Society, 29-35.

4. Fagan, M.E.: Design and Code Inspections to Reduce Errors in Program Development. IBM Systems Journal. 15(3) (1976), 182211.
5. Gerrit - web-based team code collaboration tool, `https://code.google.com/p/gerrit/`
6. Jones, C.: 2008. Measuring Defect Potentials and Defect Removal Efficiency, The Journal of Defense Software Engineering. June 2008. 11-13.
7. Kupiainen, E., Mäntylä, M.V., Itkonen, J.: Using metrics in Agile and Lean Software Development A systematic literature review of industrial studies. Information and Software Technology. 62 (2015), 143-163.
8. Lehtonen, S.: Metrics for Gerrit code reviews, University of Tampere, School of Information Sciences, master's thesis, 62 pages, 2015.
9. Markus, A.: Code reviews. SIGPLAN Fortran Forum 28, 2, (2009) 4-14.
10. McConnell, S.:Code Complete, 2d Ed.. Microsoft Press, 2004.
11. McIntosh, S., Kamei, Y., Adams, B., Hassan, A.E.: The impact of code review coverage and code review participation on software quality: a case study of the qt, VTK, and ITK projects. In: Proceedings of the 11th Working Conference on Mining Software Repositories, (2014), ACM, 192-201.
12. Perry, D.E., Staudenmayer, N.A., and Votta, L.G.: 1994. People, organizations, and process improvement. Software, IEEE. 11(4) (1994), 36-45.
13. Porter, A., Siy, H., Mockus, A., Votta, L.: 1998. Understanding the sources of variation in software inspections. ACM Transactions on Software Engineering Methodology. 7(1) (1998), 41-79.
14. Rigby, P.C., German, D.M., Cowen, L., Storey, M.: Peer Review on Open-Source Software Projects: Parameters, Statistical Models, and Theory. ACM Transactions on Software Engineering Methodology. 23(4) (2014), Article no. 35.
15. Sommerville, I.: Software engineering, Ninth edition. Addison-Wesley, 2011.
16. Thongtanunam, P., Kula, R.G., Cruz, A.E.C., Yoshida, N., Iida, H.: Improving code review effectiveness through reviewer recommendations. In: Proceedings of the 7th International Workshop on Cooperative and Human Aspects of Software Engineering, (2014), ACM, 119-122.
17. Tomas, P., Escalona, M.J., Mejias, M.: Open source tools for measuring the Internal Quality of Java software products. A survey, Computer Standards & Interfaces. 36(1) (2013), 244-255.
18. Wiegers, K.E.: 2002. Seven Truths About Peer Reviews, Cutter IT Journal, July 2002. Retrieved from http://www.processimpact.com.