Metrics for Gerrit code reviews

Samuel Lehtonen

University of Tampere
School of Information Sciences
Computer Science
M.Sc. thesis
May 2015

UNIVERSITY OF TAMPERE, School of Information Sciences
Computer Science
Lehtonen, Samuel: Metrics for Gerrit code reviews
M.Sc., 62 pages
May 2015

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 without physical meetings or mailing lists. In large software projects, tens or even hundreds of code changes are uploaded daily and following the code review process becomes increasingly hard for managers and developers themselves.

To make monitoring the review process easier, this thesis introduces review metrics for Gerrit code review. The metrics can be used, for example, to follow the amount of code changes and how long each activity within the review process take.

The metrics were implemented in a case company and data was collected from few different projects. The metrics were crafted based on measurement goals that were defined after conducting desk research on existing metrics and code review practices and after interviewing developers and managers.

The most notable benefits of the metrics are the ability to recognize the most time consuming areas of Gerrit code reviews and the simplicity to monitor the total amount of open changes. The review and code change quality can also be measured with the metrics by calculating the amount of positive and negative reviews given and received by developers.

By collecting data with the metrics new best practices for Gerrit code reviews can be established. With the help of the metrics, weaknesses are identified and the process can be optimized. Metrics can also be used to motivate developers for better performance when the metrics are publicly available for developers to see their own performance.

Contents

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 live environment. The main motivations for reviews are to detect software defects and to improve code quality while sharing knowledge among the developers. Reviews were originally introduced by Fagan [1976] already in 1970's. Since then the practice of code reviews has evolved into few different variants while the original, formal type of code inspections are also still used in many companies.

Modern code review, or tool assisted review which is investigated in this thesis is one of the variants and is a light weight version of the original code inspections. In modern code reviews people can work from different locations utilizing often web based software tools together with version control to manage review related tasks [Thongtanunam *et al.*, 2014]. As reviews are done individually and people may take part from locations around the world, the process could be hindered by the loss of control over the process. To solve this issue, in this thesis I am investigating how to apply metrics for measuring code reviews conducted in Gerrit [2015] code review tool.

This thesis focuses on modern code reviews (later code reviews or reviews), which are light weight and easily customizable variant of the traditional, formal code reviews or inspections (later code inspections). Special attention is on reviews done with Gerrit code review system, which is a free open source code review tool. Code reviews in Gerrit are an interesting research target because of Gerrit's current position as popular review tool among some of the leading open source projects and is used in projects like LibreOffice, Wikimedia, Eclipse and Android Open Source Project [Gerrit 2014b].

Reviews are known to be very efficient in defect detection and prevention especially in large software projects [McConnell 2004; Bachelli and Bird, 2013]. For the review process to be successful, reviews need to be done with enough care and without unnecessary delays. To ensure that, review process has to be motivating for developers while also being monitored and controlled to ensure the review quality. One way of doing this, is by measuring different activities taking place during the review process. This thesis proposes ways to measure those activities in order to provide metrics for better control over the reviews. At first fundamentals of software development is discussed and quality assurance in general. Then importance of code reviews as a part of quality assurance is explained and various activities involved are identified.

Good programming practices in code review context are discussed in the Chapter 4. The focus is on providing understanding of how good code is written and how that can be promoted during the code reviews and how teams can form their own set of standards. As a conclusion a set of general guidelines are proposed to act as a good starting point for writing easily readable and clean code.

The metrics for measuring code reviews are introduced in the Chapter 5 after explaining the case where the metrics were introduced. The need for the metrics is discussed and the reasoning why certain metrics were selected is explained.

The introduced metrics are used to examine data that is automatically stored during different phases of Gerrit code review. This information is then processed and prepared to be presented as graphs and key figures on a web site. The metrics are implemented as a web page so that anyone in the organization can easily take a look.

The goal of these metrics is to present the data extracted from the review process to developers and managers in a way which helps them to pick out problems in the review process, but also to motivate for better performance and give insights for further understanding of the review process. Metrics also help monitoring and controlling the reviews 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.

In a case company where the metrics introduced in this thesis were introduced, there are numerous Gerrit projects with different sizes and varying complexity. The code review flow is a bit more detailed than the default flow that Gerrit has out of the box. The introduced metrics can also deal with projects where the review flow is simpler. Considering this and the variance between the projects that the metrics are used for now, I am confident that these metrics can be copied and utilized in majority of Gerrit projects throughout the industry.

In order to be able to handle the lack of control in modern code reviews we need to gain understanding of the different activities within the code review process. To get better knowledge of what is happening and how the time is allocated during the reviews, different measurements and metrics are needed. From that raises the research question, how to measure code reviews for better control and quality in Gerrit code reviews?

This problem is addressed in this thesis by offering a set of review metrics. The areas of measurement include different phases of the review process and review outcomes. The focus

is on measuring the time that is spent on each activity and monitoring the number of reviews that lead into further code changes, also known as patch-sets in Gerrit. 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.

2. Metrics in software development

Software development is often considered to be synonymous with programming. However, a lot more is involved in a typical software project. Depending on the development style, the process is slightly different, but in most cases the process can be roughly divided into four different main parts: software specification, software construction, software validation, and software evolution [Sommerville, 2011]. The specification is the driver for the rest of the development process and is the main consideration when deciding how to proceed with the development process. Specification defines the problem to be solved, and what are the system requirements for solving the identified problem. [McConnell, 1996]

Software construction is a process where the actual programming is done and software is constructed to meet the specifications. The type, size and requirements of software to be developed set some restrictions for choosing the software process model [McConnell, 1996], but in modern software development, most of the software is developed following agile methodologies where the construction is done piece by piece with small increments. In larger systems it is typical to combine different processes like the waterfall model with agile, incremental development. [McConnell, 2004; Sommerville, 2011]

The waterfall process is rigid and slow to react on changes, but gives clear structure to projects. Pieces of larger project can be divided into smaller projects which may utilize some other software development process like agile software development. While waterfall process can be considered clumsy and heavy process [Haikala ja Märijärvi, 2004], agile methods value individuals and interactions, working software, customer collaboration, and responding to change over processes, documentation, contracts, and plans. In addition to that, agile development emphasizes frequent deliveries with short development cycles, continuous learning and face-to-face communication, and learning [Beck *et al.*, 2001]. The idea in agile development is to produce continuously new working versions of software that can be tested at any time. As soon as a developer has finished a piece of code, it can be integrated into the rest of the code and tested immediately.

The practice of integrating and testing each piece of new software continuously is called continuous integration. Whenever a new piece of software is integrated, the whole program is tested against unit tests, which all test certain functionality in the software. This enables immediate feedback of functional errors. These tests are usually automated and are quick way to validate functionality of new code and its performance together with the rest of the system. However, writing a test for every possible combination of functions takes a lot of

resources and in a typical project, not every possible situation has its own test. [Ganssle, 2012; McConnell, 2004]

Continuous integration is effective way to ensure absence of a majority of functional defects, but to make sure that the delivered product meets quality expectations, wider measures of quality assurance are needed. The focus in this thesis is in modern code reviews, which are recognized to be important part of the modern quality assurance practices. [Rigby *et al.*, 2014; Wang *et al.*, 2008; Sommerville, 2011]

Code reviews are usually performed after unit tests have passed. Reviews focus on problems with program logic, performance and evolvability. Unit tests only test that the program gives the expected answer to a certain input. Unit tests don't, for example, tell anything about program's complexity or efficiency. When in contrast a reviewer can quickly spot code that has complex structure and which is difficult to maintain.

Additional quality assurance measures that should be taken include activities like design inspections, new function tests, regression tests, performance tests, system tests, and external beta tests. To ensure minimum amount of defects, all these activities should be included. However, even when applying all of these, it is pretty hard to achieve defect detection rate higher than 95 %. [Jones, 2008]

Defect count of software is a measurement of software quality, but for deeper understanding and for better control of software quality the International Organization for Standardization (ISO) has crafted a set of quality standards, known as ISO 9000. The standard provides general guidelines for establishing quality measures in an organization. It is applicable to wide range of industries and scales to software development, too. It sets out general quality principles that should be used in the organization following the standard. ISO, [2015a] describes the standards to:

> "Provide guidance and tools for companies and organizations who want to ensure that their products and services consistently meet customer's requirements, and that quality is consistently improved."

From the ISO 9000 set, the ISO 9001 is the most important standard in software industry [Haikala ja Märijärvi, 2004]. It applies to organizations that design, develop, and maintain products, including software. It is not itself a standard for software development but is a framework for developing software standards [Sommerville, 2011]. Following an ISO standard in software development is a sign for customers that the product is developed under certain, well defined quality standards. In some cases customers may even request that a company qualifies for the standard [Haikala ja Märijärvi, 2004].

As the standards can be used in different industries, ISO 9001 does not directly give good tools for measuring software project quality or code quality but to address that, there is another standard called ISO/IEC 25000 developed together by ISO and International Electrotechnical Commission (IEC) [ISO, 2015b]. ISO/IEC 25000 is a new set of standards including the older ISO 9126 which has been widely used in software projects. While using standards as a starting point for software metrics is one approach it has sometimes been seen problematic and hard to apply effectively [Jung, 2004]. However, Kanellopoulos *et al.* [2010] have proposed methodology for using ISO/IEC 9126 to evaluate the code quality and static behavior of a software system so that the software can be measured using the guidelines provided by ISO.

The quality assurance as a part of software development is important, and its importance is ever increasing as programs are increasing in size and more software than ever is produced [Wang *et al.*, 2008]. In practice, quality assurance is important, because when software suffers from quality problems; more resources are used fixing the software than was actually spent when constructing it. The best solution would be avoiding that situation in the first place and take corrective actions early enough [McConnell, 1996; Ganssle, 2012]. The best way to do this is to carefully apply quality assurance principles straight from the beginning. Sooner the defects are found, the cheaper they are to fix. [Black, 2007] Still there are projects which try to save resources by cutting the quality assurance activities like code reviews and unit testing [McConnell, 1996]. According to Jones [1994] shortcutting one day of quality assurance activity at early stage of the project is likely to backfire with 3 to 10 days of bug fixing activity later in the project.

Metrics are an important part of software development as without any metrics, measuring improvement and identifying the best practices would be impossible. The metrics usage in software development is motivated by various reasons, process improvement being the most popular. Other common usages for metrics are to support communication and help with the decision making. [Kupiainen *et al.*, 2015]

Before any significant measurements can be done there needs to be a clear understanding of what is measured and what the measurement goals are. In a review of data collection at NASA's software engineering laboratory, it was concluded that the most important lesson learned in 15 years was that measurement goals have to be well defined before any measurements can be done. [McConnell, 1996]

While choosing the measurement goals carefully is essential to get good results, it is also important to dismiss measurements which are not needed. Collecting data that serves no purpose for the project is waste of money and time and should be avoided. When a new measurement is introduced, it has to have a measurement goal but also a reason why that

measurement goal is important. Basil and Weiss [1984] have introduced a process for checking measurement validity:

- *Set goals.* Determine clearly how you want to improve projects and products. Your goal might be for example, reducing the number of defects put into the software in the first place so that you don't spend so much time debugging and correcting the software downstream.

- *Ask questions.* Determine what questions you need to ask in order to meet your goals. A question might be, "What kinds of defects are cost the most to fix?"

- *Establish metrics.* Set up metrics that will answer the questions. You might start collecting data on defect types, creation times, detection times, cost to detect, and cost to correct.

Measured data should also be presented so that it is interesting for developers to follow and easy enough to understand. If metrics are too complex or measure something that is not relevant to daily work, they are not interesting for developers to follow. For example, measuring of development activities provides useful information for developers who want to plan and track their projects. Developers can also use metrics to track quality and progress in order to improve their performance and influence the team they are working with. [Kupiainen *et al.*, 2015]

Metrics are not only for management, but they can help both, developers and management. For example, developers themselves can draw conclusions from a situation where metrics indicate the number of integrated changes for the past few days is less than the number usually is. This implies that there might be problems somewhere in the development or review process, which should be investigated. By examining the metrics, management, or developers themselves can take corrective actions immediately to improve the situation. [Cheng et al., 2009]

One way of making metrics more interesting among developers is to place monitors showing metrics to hallways. When, for example, open defects and development progress are shown publicly, it motivates developers to react to new defects quickly and they also fix them faster. In a case when fix times of broken builds were shown next to the coffee machine it made developers fix the builds faster and issues were discussed more openly. [Kupiainen *et al.*, 2015] In another study where failed tests where visible to whole organization, teams worked faster to fix them, because they didn’t want those bad results to be seen. [Cheng *et al.*, 2009]

When anything new is introduced, there is always a possibility for resistance for change. At the manager side this could show as a lack of interest towards the use of metrics or at the developer side as a fear of unnecessary monitoring and setting too strict controls, which results as less freedom. Easy way to fight the resistance is to promote clear communication when any new practices are taken into use. In this case it should be made clear that the metrics are not used to monitor individual developers or restricting existing practices, but they are used to help everyone to see how the projects are progressing and to help to see where there is room for improvement. [Burnes, 2009]

Even when right things are measured it doesn't necessarily mean the metrics are good. There are many things that contribute to the overall success of software metrics. Metrics are for no great benefit if the data gathering is slow, metrics being out dated when published or if maintenance generates too high costs.

These problems can be avoided when, in addition to the measurement goals, also the implementation is well planned. Tomas *et al.* [2013] have researched the characteristics of a good software metric tool and they have listed several key attributes which a successful metric tool should have. They start by stating that a good software metric tool gathers data automatically and provides valid, up to date data on regular basis and the tool shouldn't require much effort to maintain and it is useful to have a possibility to easily add new metrics if needed.

Automatic data gathering and presentation of the metrics is important as the data is about to change constantly and collecting it manually would require additional resources. In an ideal situation, after setting up the system, the graphs are built automatically from constantly updated data and there is very little human effort required. When there is no human factor involved in the graph making nor in the metrics calculation, there is also very little chance of calculation errors or erroneous data. When the metrics process is automated, it allows people to focus on results rather than data collection and acquisition of results. There can also be thresholds set and any data points exceeding these minimum or maximum values can be ignored. Warnings can also be automatically raised when certain observations in the data are made or a threshold is passed. [Tomas *et al.*, 2013]

The properties a successful software metric should have are summarized in Figure 1. First of all, the metric has to have a purpose, meaning that there is a true need for the metric results and the measurement goals are well defined. Secondly, the data should be up to date. If metrics are using data which is no more valid, there is not much value for anyone. To enable continuously valid metrics, the data is best collected automatically, while at the same time minimizing maintenance costs and human errors. Finally, the metrics should be interesting to look at, so that the audience wants to follow how the metrics change during the time.

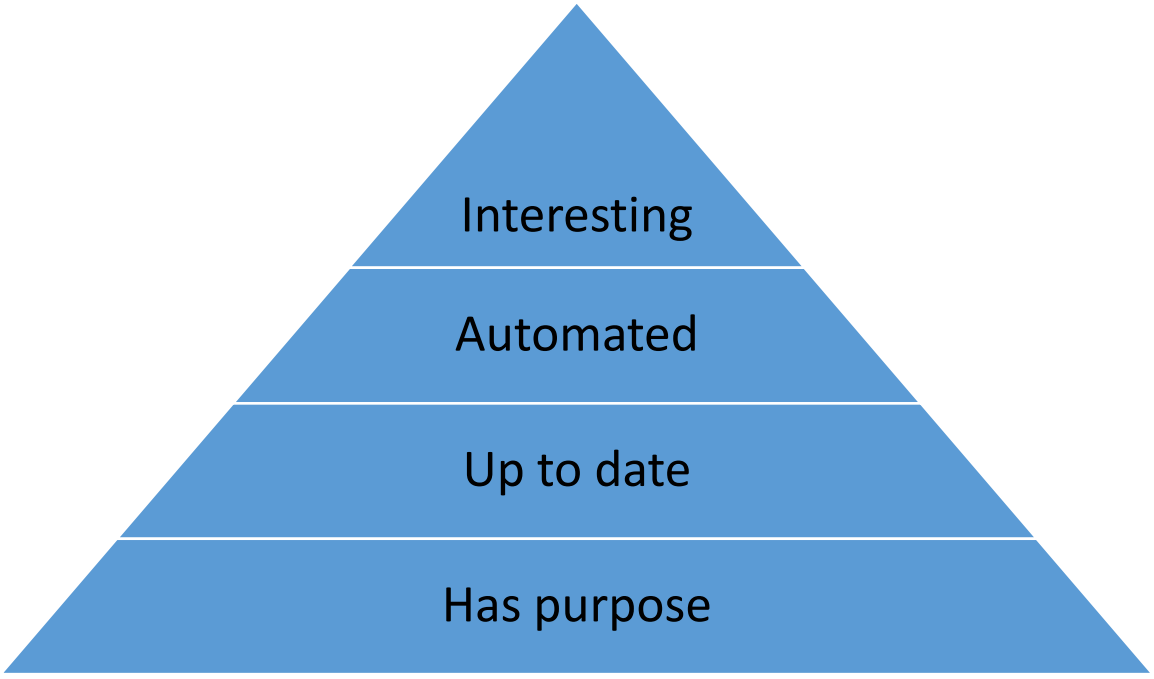

Figure 1: Software metrics success factors.

Management has an essential role in determining how well an organization's review activities work. Key issues include management participation in reviews and how the data is used, or misused. Management also has to show commitment to the review practice and promote its usefulness. [Wiegers, 2002]

Metrics in software development are a powerful tool for process improvement. By comparing the generated metrics with each other along the organization, it is possible to draw conclusions about the efficiency and quality of certain software processes, tools and methods. As a result, the best performing projects can be benchmarked and best practices established [Sommerville, 2011]. These best practices can then be applied in other projects in the organization. Measuring the activities also provide basis for discussion which could result on ideas for better practices leading to better overall performance.

Metrics can also be misused. Assessing developer performance based on collected defect data is often cited as a culture killer and should be avoided. It could lead in to a situation where measurements motivate people to change their behavior to do things differently in order to make the statistics look better. In this case the original goal of the metrics is no more met. Wiegers [2002] discusses a case where a manager in one company declared that finding more than five defects during a code inspection would count against the author's performance evaluation. This kind of strategy which emphasizes the evaluation could lead into bad review practices where developers do not want to submit their work for review or they try to split them into small chunks to avoid finding too many defects in any review. Reviewers might

also point out defects to authors off-line, rather than during the review so that there won't be a mark in the statistics. As a result, organization's culture might turn into promoting an atmosphere of not finding defects during a review. Such evaluation also criminalizes the mistakes that we all make and motivates participants to manipulate the process to avoid being hurt by it. [Wiegers, 2002]

3. Code reviews

The main idea of a code review is that any produced code is read through and assessed by at least two persons, the author and one or more reviewers. This means that every line is read through by two or more persons, which makes it hard for defects to hide [McConnell, 2004]. The usefulness of code reviews was first recognized already in the 1970's when Fagan [1976] first demonstrated the practice of code review and its benefits. At the time the reviews were organized in a form of formal code inspections. In these inspections several people were invited into a meeting where the code was inspected and commented line by line. [Fagan, 1976; Bachelli and Bird, 2013]

Since then, the concept has evolved into different types of review activities while the original style inspections are also still in use [Cohen *et al.*, 2013]. Lately a type of modern code review has risen, which with a help of review tools has taken the practice into more agile and light weight direction [Thongtanunam *et al.*, 2014]. Modern code reviews are still following the same basic principles as the original inspections, but the framework is light weight, with formal meetings most often being replaced with online tools. [Beller *et al.*, 2014]

The difference between code inspections and code reviews is that code inspections are typically formal with very strict agenda and are performed in planned meetings where each participant has his/her own role. Modern code reviews on the other hand do not need to happen in a planned meeting and are not tied to a location. Reviews can be done by using mailing lists, separate software or online tools. Online review tools like Gerrit are popular, because while they make it possible for people from different locations to participate, they also make reviews more organized than, for example, mailing lists [Gerrit, 2014a]. When reviews are done outside formal meetings, it eliminates scheduling problems which are common when formal inspections are used [Cohen *et al.*, 2013].

The code review practice could vary a lot between organizations. In some organizations all code changes are not reviewed but only some of them. Although it is known that less review coverage often means more unnoticed defects [Beller *et al.*, 2014; Cohen *et al.*, 2013]. If every change is not reviewed, various methods for selecting the code for review can be used. The code to be reviewed could be chosen randomly or there could be an algorithm which chooses changes with highest complexity or other attributes which hint for problematic code. [Tomas *et al.*, 2013]

Code reviews in its different forms are nowadays widely used and accepted best practice across the industry [Bird and Rigby, 2013; Beller *et al.*, 2014]. Although reviews take additional resources, there are many benefits that they can provide.

3.1 The benefits of code review

In this section the benefits of code reviews are discussed. Since the introduction of reviews in the1970's the main motivation has changed a bit. Originally the reason for code reviews was mainly to spot defects and prevent any bad quality code from getting into codebase. Bachelli and Bird [2013] reveal that while finding defects is still the main reason and motivation for reviews, there are also many other benefits which act as motivation to conduct reviews. These include at least sharing of knowledge, improved communication among developers, better overall software quality.

3.1.1 Defect finding and removal

A successful code review enables a product with less defects and improved maintainability [Thongtanunam *et al.*, 2014]. Improvements are not limited to code, but the whole project may benefit from the reviews as coding 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 structure. [Jones, 2008; Wiegers, 2002]

The importance of defect removal during code reviews is well demonstrated in Figure 2. In the requirements capture stage, finding and fixing a bug establishes the base level for how much fixing a defect costs. That is marked as multiplier one in Figure 2. In the design stage the cost of defect removal doubles and when we get to system implementation stage, the cost is already fivefold when compared to the requirements capture stage. Code reviews typically occur between unit testing and integration testing, which means that fixing defects found in review is likely to be ten to hundred times cheaper than fixing defects found after the integration or deployment. If a defect gets into deployed system, the fixing of defect could become thousand times more expensive than fixing it right in the beginning.

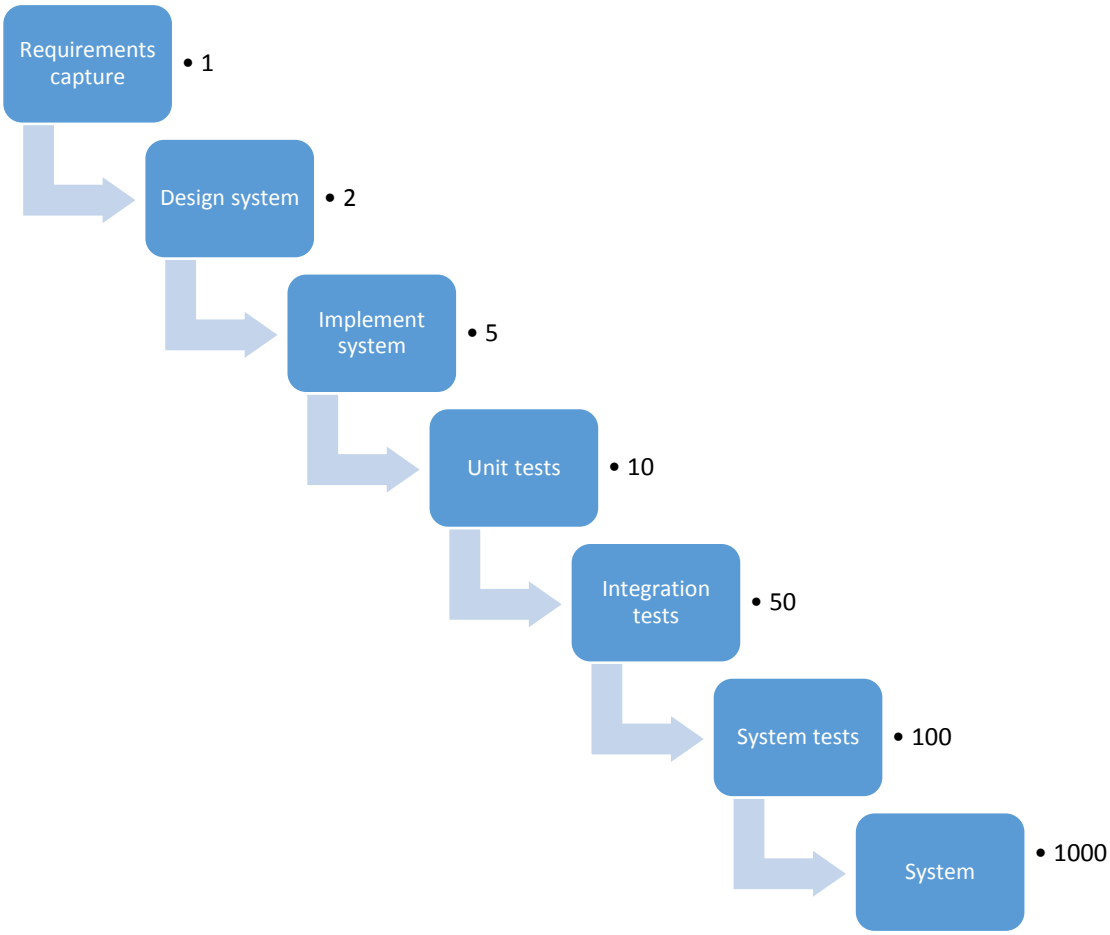

Figure 2: Different phases of development and the cost multiplier in a software project [Black, 2009].

According to Jones [2013] software testing typically averages at around 30 - 35 percent of defect removal efficiency and rarely exceeds 50 percent. Formal code inspections are known to hit quite often defect removal rates as high as 85 % and the average is around 65 %. Informal code reviews are not known to be that efficient, but in a study where formal inspections scored 80 % the defect removal rate of informal code reviews was still around 60 % [Wiegers, 2002].

The importance of code reviews as a part of the whole quality assurance is best demonstrated by comparing its efficiency against the total effectiveness of all the quality assurance activities. Typical averages of potential defects and their removal efficiency are presented in Table 1. Removal efficiency is the rate of how effectively code reviews are able to remove specific type of defects. The defect potential is the average of defects per function point [Jones, 2008]. Function point measures the size and complexity of the software product by

quantifying the functions which are meaningful to the end user [IFPUG, 2015]. Function point based metrics are considered to be the most accurate and effective method for calculating software size developed [Jones, 2013]. It is usual that the number of defects per function point increases as the size of program grows. The defect removal rate is the average percent of the defects spotted and fixed in reviews. Jones [2008] found that the best companies have defect removal efficiencies up to 95 % while the average figure is around 65 %. From that 95 % defect removal rate, the code reviews represent 80 %, while most of the rest is detected in testing. If functional defects are taken into consideration, the importance of reviews is not that big, as 75 % of the defects are evolvability defects and functional defects accounting only 25 %.

Defect origin	Defect potential	Removal efficiency	Defects remaining
Design defects	1.25	85 %	0.23
Coding defects	1.75	95 %	0.09
Bad fixes	0.40	70 %	0.12

Table 1 - Defect removal efficiency of different type of defects [Jones, 2008].

McConnell [2004] compared code reviews with unit testing and found that the defect detection rate of unit testing was around 25%, while in code reviews the defect detection rate was 60%. He also described a number of case studies including an example where the introduction of peer reviews led to a 14% increase in productivity and a 90% decrease in program defects.

By combining different defect removal activities it is possible to reach a defect removal rate of around 95 %. This is a realistic target with typical resources when all the following quality assurance processes are used [Jones, 2008]:

- Design inspections
- Code inspections
- Unit tests
- New function tests
- Regression tests
- Performance tests
- System tests
- External beta tests

Targeting defect removal rate of 100 % is not seen feasible as the costs would become too high [Jones, 2008]. To support the code reviews, many researches indicate that formal design and code reviews can be effective ways to identify defects so that they can be fixed early in the development cycle [Jones 2008; Tanaka *et al.*, 1995]. It is recommended that code reviews target every line of changes made, because fixing any unnoticed defect becomes increasingly expensive as the software matures. [Tanaka *et al.*, 1995; Sommerville, 2011; Black, 2007]

Software defects are often considered only as bugs which pass down to the final product and are then found. However, defects can also be evolvability defects, which make maintaining, testing or further development of the software harder. This type of defects are around 70 % - 75 % of all defects found [Mantyla and Lassenius, 2009; Siy and Votta, 2001; Beller *et al.*, 2014] and these cannot be found anywhere but in reviews, because unit tests only test the software for functionality [Mantyla and Lassenius, 2009]. For tailored software where extensive future development is not expected finding all the evolvability defects is not that important, but when same codebase is used for longer time and software constantly evolves, these defects are equally important to functional defects, because software which has bad maintainability, becomes very costly on longer term. In experiments by Bandi *et al.* [2003] and Rombach [1987] similar software systems with different evolvability were compared. In less evolvable system it took 28 percent longer to add new functionality and fixing a defect took 36 % longer.

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.* [2014] and Porter *et al.* [1998] 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 big effect on review results. Statistical analysis of code reviews by McIntosh et al. [2014] 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.

3.1.2 Knowledge transfer

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. Now review tool is 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 comments or questions. While 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. [Perry *et al.*, 1994]

One aspect to the importance of knowledge sharing is the pass down and division of information to several developers. When more than one person has understanding of the code, maintaining and development is not dependent on a single developer. If the developer decides to leave the project other developers are able to continue the development or maintenance that component. If there is no knowledge sharing whatsoever, it would take a lot of effort for a new programmer to get grips with the old code and continue development. [McConnell, 2004] Knowledge sharing should therefore be taken seriously and, for example, in Linux kernel project, which is one the most active open source projects, insufficient discussion is the most frequently cited reasons for the rejection of a patch. [McIntosh *et al.*, 2014; Wiegers, 2002]

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.

Knowledge sharing raises another benefit that code reviews bring. When code is reviewed before integration, it means increased controls and possibility to improve the code quality before the change is integrated to the trunk [Tanaka, 1995]. This is valuable especially in systems where maintainability is important [Mantyla and Lassenius, 2009]. Therefore, one objective in reviews is that reviewers discuss openly about the proposed changes with the author and try to improve the code whenever possible. Objectives of improvement are to make the code more readable, better commented and more consistent. The code should also be testable with clear input and output and it should be easy to maintain. [Markus, 2009; Bird and Rigby, 2013; McConnell, 2004]

Introducing strict quality standards may face resistance or might be tiresome to follow, but code reviews leave a bit more freedom to the developers as the quality is ensured in the reviews while developers can also give feedback about the quality to each other. When

decisions are made by the group, they will eventually develop their own idea of what constitutes as a good code. These standards may include formatting, naming of variables, structure etc. [McConnell, 2004; Bird and Rigby, 2013; Bird and Bachelli, 2013]. Good programming practice and its role in code reviews is discussed more thoroughly in Chapter 4.

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.

3.2 Good code review practice

Code reviews are useless if they are used the wrong way. Too large code changes in too many files make reviewer's job a nightmare and the ability to spot defects drops. Studies have shown that a good amount of code to be reviewed at one time is few files at maximum with no more than 200-300 line changes [Cohen *et al.*, 2013; Porter *et al.*, 1998]. Defects found per 1000 lines of code start to near zero rapidly when changes are larger than 400 lines [Cohen *et al.*, 2013]. The relationship of defects found and lines of code under review is shown in Figure 3.

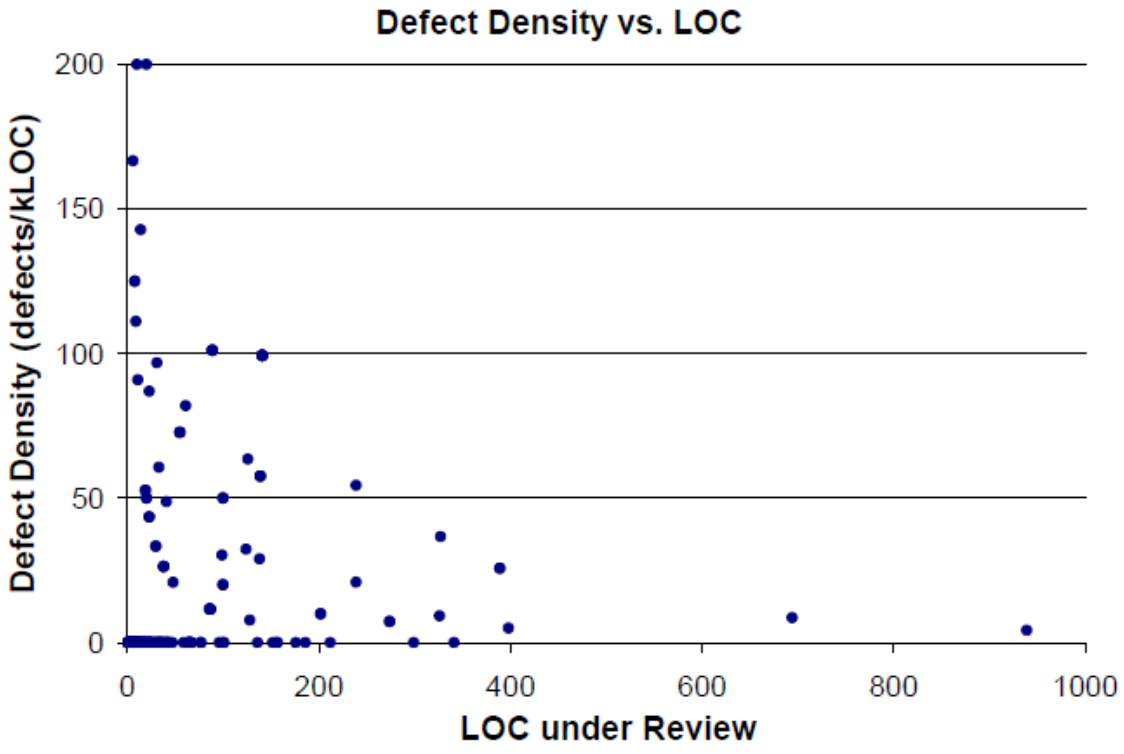

Figure 3: Defects found per 1000 lines of code [Cohen *et al.* 2013].

Dividing changes into smaller segments, typically leads in to less complexity. Less complexity means that reviewer needs less effort to understand the changes and can give feedback faster [Mishra and Sureka, 2014]. Rigby *et al.* [2014] investigated change sizes of 11 to 32 lines of code and noticed that changes of this size allow almost continuous feedback loops, which speed up the development and improve the code quality. Changes with smaller size were also integrated faster.

Of course sometimes there is a need for much larger changes. These can be very simple, like adding a list of items, or the change can include refactoring or removing functionality. Larger changes may also be due to an addition of a completely new module or feature. A good code review practice is that every change should be a complete, working piece of code that does not break the build [AOSP, 2014]. This can be made sure by forcing continuous integration before a change can be submitted. If unit tests have passed, the reviewer's task is mainly to hunt down problems with logic and evolvability defects.

The optimal time spent for a review of one change is less than an hour at once. There are two reasons for this. During an hour a reviewer should have found most of the defects and after that the reviewer typically just looks for same things all over again. This is especially true if a review checklist is used. Checklist is a tool used in reviews which lists the most common causes of defects which should be checked during the review. Also the ability to focus starts to fade during longer periods. However, reviews shouldn't be done too quickly either as more defects are found with more time used on the review. [Cohen *et al.*, 2013]

3.3 Review tools

To make code reviews easier various tools have been developed. Large amount of them are online tools which allow modern web based reviews while some are simple plugins to integrated development environments. The idea of the tools is to make reviews more organized and easier by providing features like code comparison, reviewer roles and integrated version control. In interviews done during this research it was also pointed out that review tools are easier for reviewers to start with than, say, more old fashioned mailing lists which are most notably still used in Linux Kernel development. [Kernel, 2014]

The most popular online code review tools at the moment are Review board [2015], Phabricator [2015] and Gerrit [2015]. Review board is, for example, used by professional networking site LinkedIn. Phabricator is known to be used by social networking site Facebook. Gerrit is probably best known for its wide usage in Android community [AOSP, 2014]. Currently all of these are available as free software and each can support projects with various sizes. This research focuses on the use of Gerrit code review and the review metrics

proposed in this thesis are designed to integrate with Gerrit. Gerrit was the chosen target for metrics because it is one the most popular code review systems [Mishra and Sureka, 2014] and it provides good infrastructure to build the metrics.

3.4 Code reviews in Gerrit

Gerrit code review is a tool which is designed to support projects of any size to implement well organized reviews. It is published as free software under Apache 2 license and it can be installed on any web server requiring only PHP and MySQL support. Its purpose is to provide online code reviews for projects using the Git version control system. Gerrit is used for example in Android open source project (AOSP) and in projects like LibreOffice and Eclipse [Gerrit, 2014b].

Gerrit takes full advantage of version control and is strongly integrated with Git. Git is a version control system originally developed by Linus Torvalds for managing code repositories of Linux kernel project [Chacon & Straub, 2014]. Git allows distributed development with fast performance and wide variety of tools for controlling the workflow. Due to its quickness and ability to scale for projects of any size, Git has become one the most popular version control systems out there. Github which is a web service offering and hosting Git repositories for anyone who is interested currently has over 20 million repositories on its servers which gives quite a good perspective on Git's popularity nowadays. [Github, 2015]

The Gerrit code review process follows Git's best practices where new change is always started by creating a new branch which in Gerrit is called a topic. Developer then works with that branch and whenever the change is ready, the developer uploads it into Gerrit. All changes which are submitted for a review should be relatively small with low level of complexity so that they are easy for the reviewer to understand. When size of the change and its complexity increases it becomes more tedious to review [Mishra and Sureka, 2014]. Having small commits in version control is also a good software engineering practice as it makes the maintenance and repairs easier. Gerrit can be used together with continuous integration system, which makes it possible to automatically run unit tests after the change has been uploaded.

Git stores a lot of information into repositories including commit messages, notes and code changes so there is not much that Gerrit needs to save to its own database. The Gerrit's database is mainly used to store review and user data, which cannot be retrieved from Git. All of the other information it can fetch from Git. Thanks to its tight integration with Git, Gerrit is simple for developers to take into use and it doesn't complicate developers' work much as code is submitted into version control pretty much like if there was no review system

whatsoever. Interviews which were done during this research revealed that one doesn't require much training to get started with Gerrit, which could be one reason for its popularity.

Gerrit uses some terminology which may cause confusion. Throughout the software industry code changes are called patches, but in Gerrit they are called changes. The fixes which complement the change in Gerrit are called patch-sets. As this thesis is about code reviews in Gerrit, the term change is used instead of patch.

As Gerrit is a code review tool, its most important job is to support the code review. Gerrit provides reviewer with a side-by-side display of changes where additions and deletions are color coded. It also allows inline comments to be added by any reviewer which makes information sharing easy. All open changes can easily be browsed and their statuses are clearly visible. A user can quite easily navigate through all of his/her project's open changes and review them.

In addition, Gerrit brings many other tools, too. It helps the author to assign reviewers and keeps track of all the reviews made. There is even a possibility to assign reviewers automatically based on different variables that can be set. This often eliminates the need for the author to assign the reviewers manually. Sometimes the reviewer may not even know who the reviewers should be, for example, if the code change is submitted into project the developer does not usually work for.

Developers are not restricted to view their own project's changes, but if they have sufficient privileges, they can look at other project's changes too. The possibility to inspect the code is therefore not limited only to the people who are invited to review the change. The freedom to browse through changes, even in other projects, is a good way to share information between developers and projects and it allows developers to learn from others.

Gerrit also simplifies Git based project maintainership by permitting any authorized user to submit changes to the master Git repository, rather than requiring all approved changes to be merged in by hand by the project maintainer through pull requests, which would require more work. This functionality enables a more centralized usage of Git.

Just like in the original 1970's style code inspections, Gerrit code reviews also have different roles for different participants. These roles are by default author, reviewer, approver and maintainer. It can be decided within the project what roles are used. Only mandatory roles for reviews to work are author and reviewer. In a typical Gerrit code review use-case authors upload their proposed code changes to Gerrit for review. Each uploaded change should be a complete feature or bug-fix, which does not break the build. After the change is uploaded the author invites other developers to review the code. Gerrit could also add reviewers automatically based on given parameters. [Gerrit, 2014a]

During the review, a reviewer goes through the files which are included in the change he/she is assigned to review and finds the changes made and reads them through line by line. Reviewer inspects the logic and formatting and tries to find places of improvement or criticism from the code. Whenever reviewer finds something to comment, he/she leaves comments to the author. Gerrit shows these comments together with the changed lines. Author can then address these comments by replying or by uploading a new patch-set which fixes the issues addressed.

There can be any amount of patch-sets in a change, but if the number starts to goes over ten, it usually indicates that the patch is fundamentally problematic. Sometimes large number of patch-sets may be due to removal of a functionality which is removed piece by piece. Often, when the number of patch-sets gets high, abandoning the change and starting over with simpler change should be considered as increasing the number of patch-sets makes the change more complicated with many layers of comments and reviews. This issue is investigated in detail in Chapter 5. After the reviewer has completed the review he/she gives a score for the change. In Gerrit there are four different scores that the reviewer can give for the code after completing the review. These options, by default, are -2, -1, +1, and +2.

If the reviewer considers the code well done and has nothing to say about it, he/she gives it a review score of +1, which means that reviewer approves the code to be merged in to the code base and doesn't have suggestions for improvement. Reviewer gives -1 whenever there is anything in the change that needs extra attention. If a change receives -1, it doesn't necessarily mean that there is anything fundamentally wrong, it could be just a note that reviewer wants to discuss on some things or that there might be a better way to implement something. Any given -1 during the review halts the review process and forces the author to check the comments and act accordingly. If the reviewer gives comments, but still approves the code by giving it +1, the author never sees the comment as the review process moves forward if no -1 is given. That is why it is extremely important to give -1 whenever there are any comments.

Whenever a clear bug is detected during the review, the reviewer has to give the change a score of -2, which forces the author to revise the code and upload a new patch-set. If the initial change has many problematic parts, it can also be considered to abandon the whole change and start all over again with a rewritten change.

The policy of how many +1's a code change needs before it is allowed to be integrated can be decided within the project. A usual policy is to require two +1's from the review. To add more control, an approver role can be used which means that in addition to code review +1's, an approver is required. Approver is typically a senior engineer who has deep knowledge of

the project. If approver is used, the change needs Approver+1 before it can move to integration.

Usual Gerrit workflow where Approver and two times code review +1 are used is shown in Figure 4. Every project may not have all of the phases shown in the picture, like testing and approver. Every Gerrit project doesn't either require two times +1 in code review. These are all customizable features of Gerrit and vary between projects.

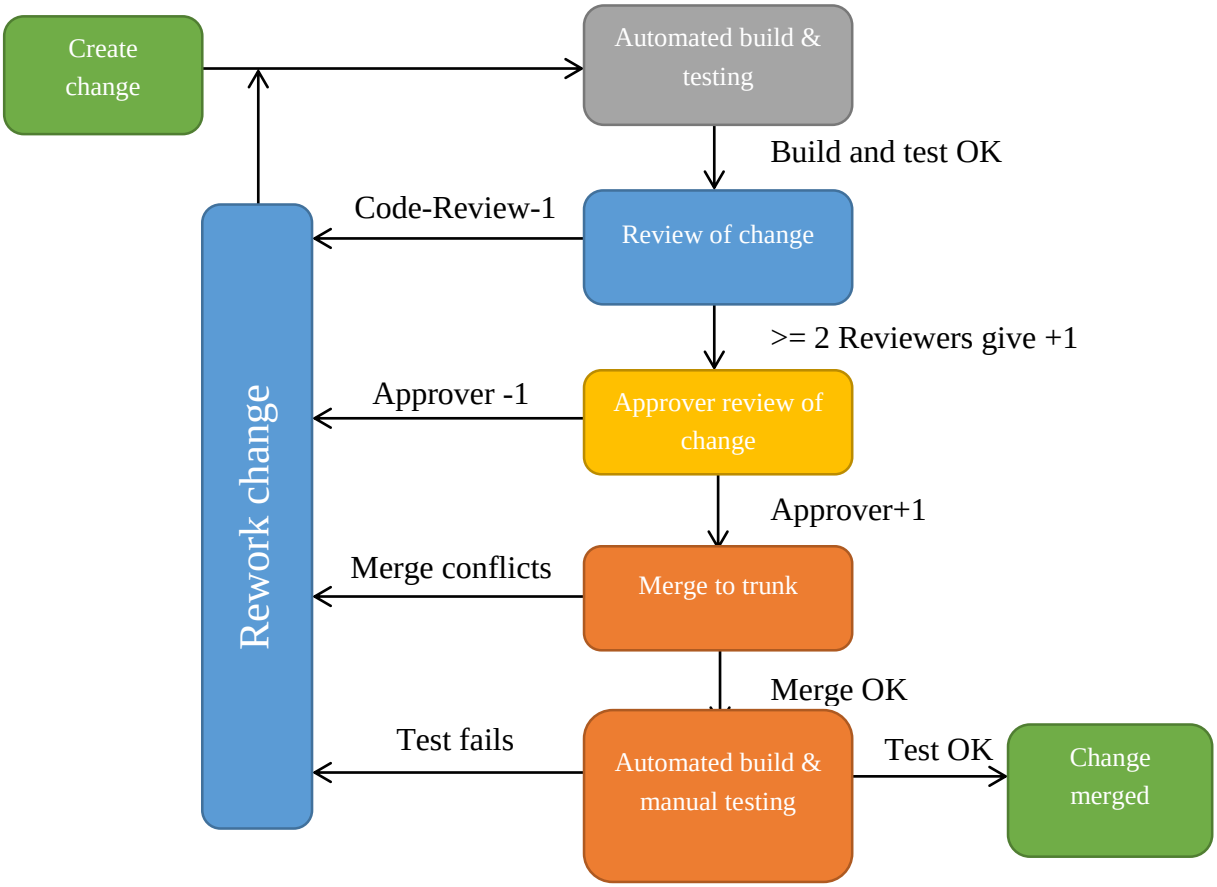

Figure 4: Typical review flow in Gerrit.

Gerrit does not come without problems. The main issue that came up when interviewing developers and was also an issue when MediaWiki [2015] compared alternatives to Gerrit, is the way Gerrit notifies users with emails. There is so much of them, and without sufficient informative content, that developers often ignore them.

Another issue, which is also related to the way Gerrit notifies developers is the following. Whenever reviewer leaves a comment in review, author will not get any notification unless reviewer also gives the change -1 review score. These comments may also remain unseen by other reviewers, and that limits the effectiveness of the knowledge transfer when compared

to, say, mailing lists, where the comments are made more visible. Gerrit UI also hides the messages from view unless user clicks them open.

Also Gerrit's performance is criticized and its tight integration with Git means that even a proposed change needs to be committed into version control, which unnecessarily populates the version control system. [MediaWiki, 2015]

3.5 Metrics in code reviews

While Gerrit gives good tools to organize reviews it only provides limited amount of metrics on the changes that are open in the system. To improve code reviews in Gerrit, sufficient metrics is essential. These metrics could include important key figures from the whole process. Measuring activities in Gerrit also allows more control over the code reviews which helps managers to maintain steady flow of patches. Metrics are also good source of knowledge for developers themselves. They can spot problems and places of improvement and come up with better ways of work.

When delays or disruptions in the review process are spotted early, an appropriate action can be taken to prevent any further delays down the chain. This becomes crucial when there are dependencies within patches. Dependencies in this context means, that if a patch is dependent on another one, it cannot be integrated before the other patch is integrated. If a patch from which many other patches are dependent on, gets stuck in the review process, it slows everyone down. Therefore, it is essential to get these patches ready as soon as possible.

Although reviews are known to be an efficient way of knowledge sharing and defect detection, many companies are still hesitant to introduce them. Often the need for additional resources and fear of slowing the development process down are reasons for management to abandon the idea of reviews [Sommerville, 2011]. One of the biggest obstacles for introducing code reviews is that the benefits come much later in the product life cycle and linking the gained value could be hard [Jones, 2009]. This ideology against reviews is strong even when it is known that the resources invested to reviews are likely to pay back many times in later phases of the software lifecycle. [Black, 2007]

It is true that code reviews may also slow the development process. The reason for that is often due to time management. While conducting this research, the interviews revealed variety of reasons for not performing reviews. The most cited reasons were hurry or not remembering to do the review. Sometimes the reason for not reviewing might be just ignorance, because of prioritizing something else. Sometimes the review requests from Gerrit are accidentally ignored because of the large amount of emails coming from Gerrit. The solution in case of lack of reviews for a change would be telling in person to reviewers that there is change waiting for review and action needs to be taken.

To have smoothly moving review process, developers need to be active as authors but also as reviewers. Otherwise, the lack of reviews can block the development of the component in question. To help this situation management should actively monitor progress of changes and make corrective actions when needed. If the issue with slow review times is time management it may be necessary to separately allocate time for reviews so that they are done quickly and with enough care after the change has been uploaded. Review times can also be affected if people who are not familiar with the component are invited to review the code. For them, understanding the code takes a lot longer than for people who have worked with the project a longer time. [Bacchelli and Bird, 2013]

Introduction of reviews could also receive resistance from software engineers who are grown in a culture of testing. They may have strong opinions that any defect can be spotted if tests are well made and they may sometimes be unwilling to accept that code reviews can be more effective for defect detection than testing. [Sommerville, 2011]

Reviews may also cause delays to development if there are dependencies between patches. This means that some change cannot be merged before the earlier one is ready and merged. Concrete example is that while the original change is still under review or author is doing fixes, someone else may need to create another change on top of that. If the original change is based on older version of the mainline, the new change cannot be integrated before the author of the first patch rebases his/her changes to match the current mainline. After that the change which was built on top of that can be integrated. Problems like this can be avoided by always using the latest version from the mainline and making small enough changes that they can be quickly merged.

Reviews can also lead to mutual misunderstanding of requirements or to a situation where the team is unwilling or reluctant to carefully inspect the code because they do not want to be responsible for slowing down the process. When reviewers work with the same code as original author, they may not be able to look the code objectively enough and some defects may remain undiscovered because of this [Sommerville, 2011]. One thing that was observed during this research was that especially in some cultures where everything negative communication is avoided in fear of losing face, like in China, people may feel ashamed of giving negative reviews. Also some may take received reviews too personally, although the idea of the reviews is not to rate the person, but the code and to help point out problems and improve the final product.

Some people may also think that managers are evaluating authors based on the number of defects found during the review. Another risk related to this is that reviewers might not be willing to point out problems if they think managers are keeping records of how everyone has performed and use that information on performance evaluations [Sommerville, 2011]. To

address this issue, managers must ensure that everyone knows that defects are positive. In the end, the purpose is to refine the code and make it better. Each defect that is found during the review is one defect less that could later be found by the customer. [Cohen *et al.*, 2013]

One argument against code reviews is false positives, which are argued to cause unnecessary delays and costs. False positives may occur when developers have different opinions how certain functionality should be implemented or when reviewer just misunderstands the logic [McConnell, 2004; Cohen *et al.*, 2013]. In interviews conducted during this research some argue the development is slowed down because sometimes too much attention is paid on even to the smallest evolvability defects, which probably doesn't have any effect on anyone. However, like explained earlier the purpose of reviews is to improve the code and documentation so that the maintainability can be done with the least effort. To handle issues like this, teams should agree on standards what to follow. At some points it may be difficult to assess whether a small defect can be ignored.

Generally programmers are proud of their own work as it is a result of a creative process and there are often many solutions to solve a problem. When code is made public for reviews some programmers might experience peer pressure and feel uncomfortable on receiving or giving feedback [McConnell, 2004]. To avoid problems that may arise, management has to communicate clearly what the purpose of the reviews is and that the goal is not to criticize individual's decisions but to improve the software altogether. Management also has to develop a culture where errors are acceptable and people are supported rather than accused for errors. [Sommerville, 2011]

When interviewing developers during this research I noticed that some people are unwilling to give negative review when there are only minor improvements, because negative reviews are considered to slow the process down. However, these improvements may not be noticed if the reviewer gives +1. As discussed earlier, one of the main motivations for reviews is code improvement. If improvement ideas, although small, are not noticed by the author, there will be no improvement and same mistakes are made later and passed forward.

3.7 Who benefits from code reviews

Code reviews can be useful for any software projects, but the biggest advantages are achieved with large projects where many developers work with a same code base. In large projects where amount of code and level of complexity is higher the knowledge sharing becomes a major part of reviews. Especially important reviews are in projects where the same software is modified and extended over time. [Jones, 2008; Mantyla and Lassenius, 2009]

Projects which are working with customer specific products with small need for evolutionary changes may not benefit as much from the reviews. However, defect detection and

knowledge sharing still motivate to utilize code reviews. Sometimes reviews can be targeted to complex parts of software or to those which are likely to be modified over time. [Mantyla and Lassenius, 2009]

4. Effects of code quality to code reviews

During code reviews reviewer's task is to read and assess the code and give comments to the author whenever a defect or something to improve is found. To make reviewer's job easier the code should be well structured and easy to read so that understanding it doesn't require too much effort.

This chapter discusses several widely accepted best practices of programming and gives an overview of what is considered good code and how to write it. Preferred programming practices are examined from different points of view. This chapter also makes an effort to form general guidelines for writing neat code.

When software projects get bigger, the importance of maintainability increases. Architecture lays the foundations and gives the structure, but that does not guarantee good maintainability [Sommerville, 2011]. Equally important to good foundations is to write the code in a way that it is easy for any developer to read and continue to work with. Well written, easy to read and well-structured code makes code reviews easier and faster to complete [Beller *et al.*, 2014; Mishra and Sureka, 2014]. That means defects are easier to spot and when the structure is clean, it less likely contains evolvability defects at the first place [Rigby *et al.*, 2014]. Clean code is also easy to maintain and debug, which decreases maintenance costs in the future [Martin, 2009].

Even a good program can turn into a nightmare to maintain if the code base is not kept clean. Software projects seldom get fully complete and are often maintained for some time after the first version is delivered. Therefore, the architecture has to be designed change in mind, so that the code doesn't get messy when new functions are added. Bad code is often a result from hurrying or trying to accomplish too many things at a time and not being able to fully concentrate on the task at hand. [Martin, 2009]

Every programmer tends to have a bit different style of programming and these variances may cause conflicts. In order to produce quality code effectively this issue has to be addressed. One way to solve this is that management introduces general guidelines which everyone has to follow. These should not be too strict so that programmer's freedom is not limited too much. These guidelines are best communicated through concrete examples which are publicly displayed and circulated. [McConnell, 2004]

To further unify the coding style a development team should agree on their own practices which supplement the general guide lines coming from the management. Team's own best practices can be commonly agreed in meetings, but often they take shape during the code

reviews when different coding styles can be publicly discussed in a real environment with live code examples. If these coding standards are coming straight from the manager there might be authority issues as sometimes managers may have become alienated from the programming or do not have the sufficient technical knowledge. It could be better if these standards come, for example, from architects. With the help of reviews, over time the team develops its own guide lines which eventually speed up reviews and make defect finding easier. [McConnell, 2004]

Standards developed by the team are discussed in code reviews and everyone in the team has to follow them. Even if some of the rules may appeal weird for some, it is best for everyone to use the same style. Code that is implemented with various styles doesn't look pleasant and is difficult to read. Issues in the code formatting can be taken into discussion rather than just solely formatting the code. [Markus, 2009]

Programming languages are constantly evolving. Some old languages are gradually forgotten while new languages are developed all the time. No matter what the language is, the principles of good programming style have little changed within the past decades. Already in the 1970's Kernighan and Plauger [1978] promoted clear structures and shallow functions to write code which is easy to maintain and understand. These same principles are promoted in many of the latest programming guides [Martin, 2009].

4.1 Clean code

There are many aspects in describing good and clean code. To get started a few different definitions produced by well-known programmers and listed by Martin [2009] are presented. The inventor and author of C++ Bjarne Stroustrup says that:

> "code should be elegant and efficient with straightforward logic in order to make it hard for bugs to hide. The dependencies should be minimal to ease maintenance, error handling complete and performance close to optimal without tempting people to make messy optimizations. And finally, clean code does one thing well."

Dave Thomas, founder of OTI and godfather of the Eclipse strategy describes clean code to be:

> "Something that can be easily enhanced by other developers, not only by its author. The code should also be literate and accompanied with enough comments to supplement the code."

In an interview by Feldman [2014], the creator of Smalltalk Alan Kay emphasizes the importance cleanliness of the code structure and compares it to user interface design:

> "Even if you're designing for professional programmers, in the end your programming language is basically a user-interface design. You will get much better results regardless of what you're trying to do if you think of it as a user-interface design."

Starting point for easily readable clean code are good variable and function names. Variable names should clearly tell what the variable is for and they should be distinct enough from each other. If variable needs commenting then there is something wrong with its name. The same applies to functions. Function's name should tell what the function does and why. Names should also be searchable and pronounceable. Variable names with, for example, only single letter 'a' are not searchable as the search would return every item containing the letter 'a'. Variable names which are native language and easily pronounceable are easy for developer to handle in his/her mind. Complex names make remembering them hard which tends to cause errors. [Martin, 2009]

The importance of good variable names is demonstrated with Code fragments 1 and 2. In Code fragment 1 the variable names doesn't tell anything about what is happening while in the Code fragment 2, the code becomes easier to understand thanks to variable names, which give a hint about their content and purpose.

```
public List<int[]> getThem() {
    List<int[]> list1 = new ArrayList<int[]>();
    for (int[] x : theList){
        if (x[0] == 4){
            list1.add(x);
        }
    }
    return list1;
}
```

Code fragment 1: Code where variable names doesn't give any hint about their purpose [Martin, 2009].

```
public List<int[]> getFlaggedCells() {
    List<int[]> flaggedCells = new ArrayList<int[]>();
    for (int[] cell : gameBoard) {
        if (cell[STATUS_VALUE] == FLAGGED)
            flaggedCells.add(cell);
        }
    }
    return flaggedCells;
}
```

Code fragment 2: Same code as above, but with easier to understand variable names [Martin, 2009].

When the naming is done correctly the code is much more readable. Studies have shown that people spend a lot of time scanning the variable names and memorizing them. This means that all local variables should be visible at the same time so that the need for scrolling back and forth is eliminated. The best practice to enforce this is to limit function sizes so that the names fit to the screen. [Cohen *et al.*, 2013]

Often the importance of code readability is not fully understood and the developer writes the code only for him/herself. However, there will be other readers also and even for the developer, it is easier to return to work with the code when it is neatly written and easy to read. The importance of readable code is emphasized by the fact that the ratio of time spent reading vs. writing code is well over 10:1 as we are constantly reading old code as part of the effort to write new code. [Martin, 2009]

The length of classes and functions should be kept as small as it is feasible. Sometimes it might be necessary to write a long function, so setting a hard line limit is not appropriate. However, if the function gets close to 50 lines, it is very likely that it can be divided into parts without harming the structure of the program. [Martin, 2009]

The reason why functions should be small is that small functions are easy to understand and read which means that they are cheap to maintain and require reasonable effort to fix [McConnell, 2004]. Small well-structured functions also reduce complexity, which makes understanding the whole program easy and code reviews fast [Mishra and Sureka, 2014]. Complex code requires always more effort to understand, which makes maintenance and future development difficult [Bandi, 2003].

It is always a good practice to separate even the smallest thing into a function, because it documents itself and makes it easier to read. Also in programming small things tend to grow into big things. [McConnell, 2004] Functions also save space and reduce duplicate code, which once again makes the code more logical to read.

There is a good principle that a function should do only one thing and do that one thing well. If it tries to do many things the logic gets easily confusing and also code reusability and maintenance suffers. Error handling should also be done in separate function, because error handling is one thing and function should do only one thing. Martin [2009] also states that optimal number of arguments for a function is zero and one or two are tolerable, but three or more needs special justification. Also passing boolean values to a function is seen as a bad practice indicating that a function does more than a one thing depending on the given value. Each function should also stay at consistent level of abstraction. It makes it easier to follow the program's flow. [Martin, 2009]

In a well-structured program each function should present the next so that the flow of the program logically progresses from top to bottom, making it easy to read. [Martin, 2009] Program's control flow and complexity can be measured, and used to evaluate the quality of the code and need for refactoring. The best known method is McCabe's cyclomatic complexity metric, which measures how deep the functions are and how many variables they use [McConnell 2004; McCabe 1976]. While the number of variables and function length give hint of complexity, the most important factor is most likely the control flow, which on the other hand is hard to measure automatically. [McConnell, 2004]

Good code has to be readable, but it also has to perform well and handle any error situations. If error handling is neglected, it becomes frustrating to debug any problems that could arise. Error handling practices and methods depend heavily on language, but for example Android developing guidelines strongly suggest handling every possible error case individually instead of catching every error as general exception. [AOSP, 2014]

While good and clean code is pretty much self-documenting, it is a good practice to have comments on every nontrivial function. In Android developer guidelines it is stated that every class and nontrivial public function a developer writes must contain a Javadoc comment with at least one sentence describing what the class or function does. This sentence should start with a third person descriptive verb. [AOSP, 2014] An example of good comment is shown on Code fragment 3.

```
/** Returns the correctly rounded positive square root of a double
value. */
static double sqrt(double a) {
    ...
}
```

Code fragment 3: Well commented function from the Android style guide [AOSP, 2014].

The things mentioned above are widely agreed best practices, but there are some, mainly styling issues which divide developers. One thing that can raise conversation is indention of the code. The question is whether to indent with spaces or with tabs and how many spaces to use. Tabs can be dynamically adjusted by developer, but spaces are always the same size. If spaces and tabs are used inconsistently within the code, it becomes pretty messy after time, so a styling issue as simple as this, is quite important.

4.2 Reasons for bad code

According to Bosu [2014], most vulnerabilities and errors found in code reviews are made by little experienced programmers. Experienced programmers make fewer mistakes than novices, but also most of these vulnerabilities are discovered by seasoned programmers who

are familiar with the project [Rigby *et al.*, 2014; Bosu, 2014]. Some projects like LibreOffice utilize this knowledge by enforcing code reviews only for code written by new developers [Beller *et al.*, 2014]. However, this is not the way reviews are recommended to be used as much of the code remains outside of reviews.

One reason for juniors writing code with more defects is that junior developers generally have less experience with the project and are not familiar with all of its functions, which makes them more likely to write duplicate code. Experienced programmers are also familiar with the most typical errors. [Mantyla and Lassenius, 2009]

Another issue which contributes to the number of errors in code is the number of lines and the number of files changed. The more there are changed lines and files, the more complicated the change becomes and more probable there are errors and it becomes harder to find these errors [Rigby *et al.*, 2014]. Generally new files which are added to the project don't increase the risk of vulnerabilities [Bosu, 2014]. This was also confirmed by Beller *et al.* [2014] who pointed out that changes which fix a bug typically contain fewer defects than changes which introduce a new function. The main reason is that change which fixes a bug is more focused on a single issue.

If a developer writes code in haste, without paying attention to quality and using variable and functions names which are not thought through it is likely that it will back fire later in the software lifecycle. In code reviews code with bad variable names and hard to read structure becomes tedious and time consuming to review. While coding is small percentage of all the life cycle costs and time, it has huge effect on following activities. It is therefore essential that coding and quality control at that stage is done right. [Haikala ja Märijärvi, 2006; Black, 2007]

To promote writing of professional and clean code there is a software development practice of code sign-offs, developed by Linux kernel project [Kernel, 2014] which means that every time a commit is made, the author signs the changes to verify that the code is ready and written with care and with the best of knowledge [McConnell, 2004]. This practice is also encouraged in Gerrit documentation [Gerrit, 2014] and is used, for example, in AOSP [AOSP, 2014] and Linux Kernel project [Kernel, 2014]. Initiatives like this are simple thing to take into use, but might have significant effect on developers' commitment.

5. Metrics for measuring Gerrit code reviews

The area of code reviews is very little studied and mostly concerned on what is achieved with code reviews and what kind of defects are found and fixed. The measurement of the actual process is even less studied. Rigby *et al.* [2014] studied review times in open source projects, but code review process in these projects differ a bit from the process used in Gerrit. There is also an ongoing study by Mukadam *et al.* [2013], where they are mining data from Android Open Source Project [AOSP, 2014], but their interest has been more on frequency of reviews. Cohen *et al.* [2013] hypothesize that one reason for the lack of public information is that the best modern review techniques and practices has been kept as competitive advantage which no one has wanted to reveal until lately.

After extensive search, I couldn't found software which would generate review metrics as extensively as are presented in this thesis. As the modern code review tools are relatively new, also measuring of the code reviews is a fresh topic with very little research [Rigby *et al.*, 2014; Beller *et al.*, 2014]. There are only few software products available which offer basic metrics, for example Bicho [2015], but for more extensive analysis there is no software publicly available.

The metrics I am proposing in this thesis are implemented in a large software company where sizes of projects and teams vary a lot. The most of these projects are doing the development in Scrum teams. These teams could be geographically scattered in multiple sites across the globe. Members of these teams could also be working from different locations which emphasize the importance of smooth code review process and good culture of communication during the reviews.

Code reviews in the projects researched in this thesis are done using Gerrit code review tool. The review process differs slightly from Gerrit's default setup. It follows a practice where every code change needs at least two +1's from the reviewers and in addition author has to declare the change ready for review by giving it +1. The author +1 is used because continuous integration and unit tests are run after the upload. By giving +1, the author indicates that the tests have passed and the change is ready. If the change doesn't pass the tests, the author needs to upload new patch-set to fix the change until the tests pass.

Every change must also receive +1 in code review from all teams working with the same code base before it is allowed to be integrated. This is to make sure that every involved team is aware about the changes being made. This is a practical example of how important tool code reviews are for sharing information.

The review process takes an advantage of Gerrit's approver role, meaning that after the code reviews are completed an Approver +1 is needed before the change is ready for integration. This is given by a selected person who has approver status on the project, usually a senior developer.

In the following is explained how the review process works in the case company. The process is also illustrated in Figure 5. The process proceeds as follows:

1. Author uploads the change and runs automated tests. This is considered as the first patch-set.
2. When author considers that the change is ready for merge he/she assigns reviewers for the change. Reviewers can also be determined automatically by Gerrit.
3. At least three +1's are needed and one person from every team using the same code base has to give review +1.
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 -1.
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. If the conflicts are simple, integrator can fix them. Otherwise, the change is again sent back to the author.
6. If the change can be merged and passes final testing it is integrated to the mainline.

In a large organization with many projects it is easy to lose control over the code reviews if they are not monitored in any way. Tools like Gerrit are very helpful in providing a platform to have reviews done in a sophisticated way, but Gerrit doesn't provide tools to monitor how quickly changes get merged or how long the change spends in different phases of the review process. So, there was a need for metrics to give deeper understanding on what is happening during the code reviews and how long it takes for a change to get integrated after its initial upload. Another motivation for the metrics was to have the status of code review process visible for developers in real time. By observing the status and pace, developers can spot problems and be pro-active and take corrective action when needed without being told to do so by the manager.

Measurement goals that are proposed for the metrics in this thesis were identified by analyzing the review process together with managers and developers. As a result a need for around ten different metrics was recognized. The goals of these metrics include improving

code quality, speeding up the review process and giving a better overview of the whole review process in order to help developing the code review practice at the company.

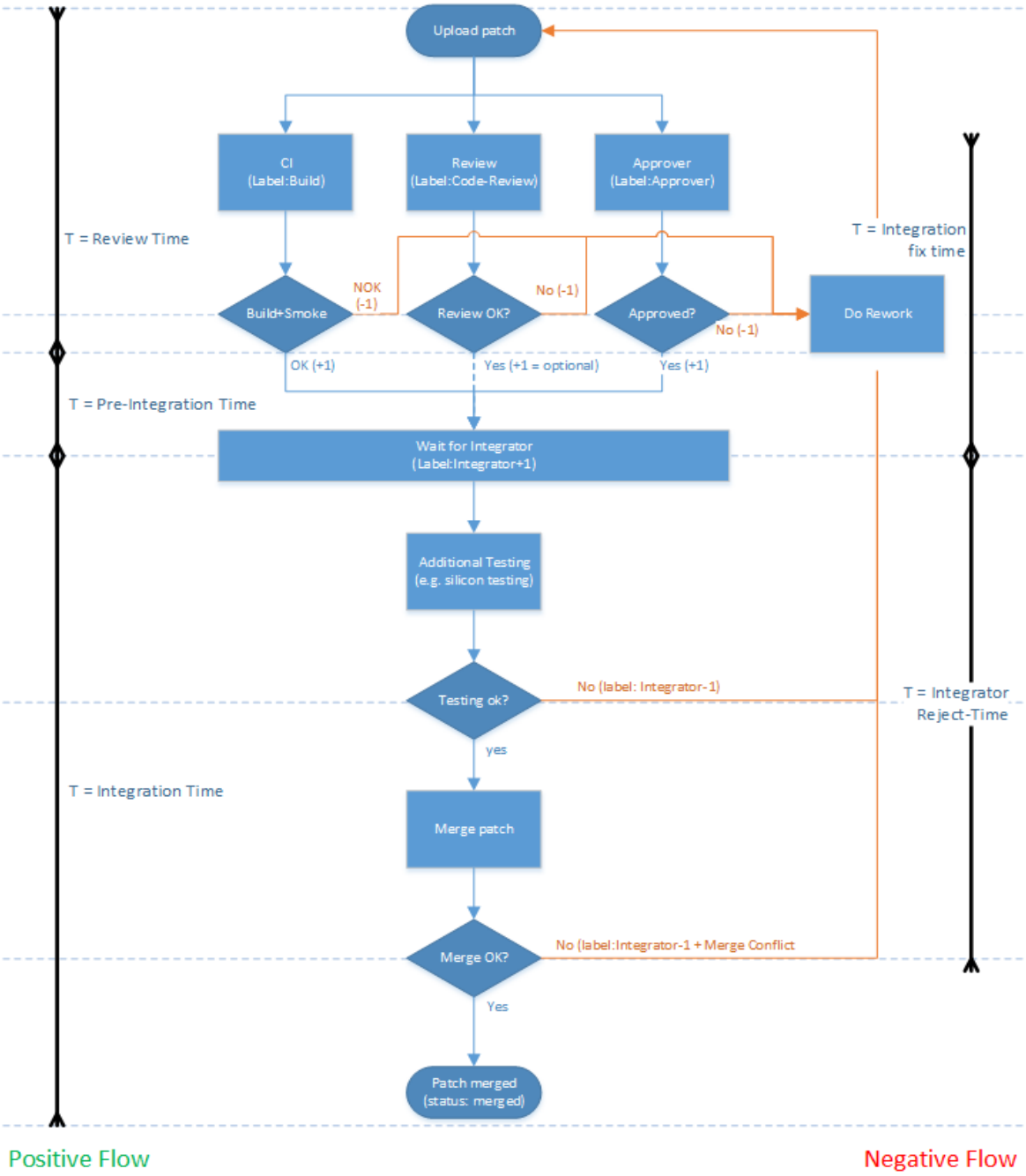

Figure 5: Change lifecycle in the case company and how each of its stage is measured.

5.1 The implementation

Initial requirements were to have couple of graphs ready and possibility to easily add more. All the graphs have to be visible for every developer so the natural decision was to create a web site where the metrics are shown (Figure 6). The web site consists of one page where different metrics can be selected from a select menu. The web page allows metrics to be seen by everyone involved in the projects. When the data is public, developers can follow the metrics themselves and find the areas of improvement, but developers also know exactly what is measured and can give feedback on the metrics. This enables constant development of the metrics and ensures that right things are measured the right way. The data is also made available as .csv for further analysis.

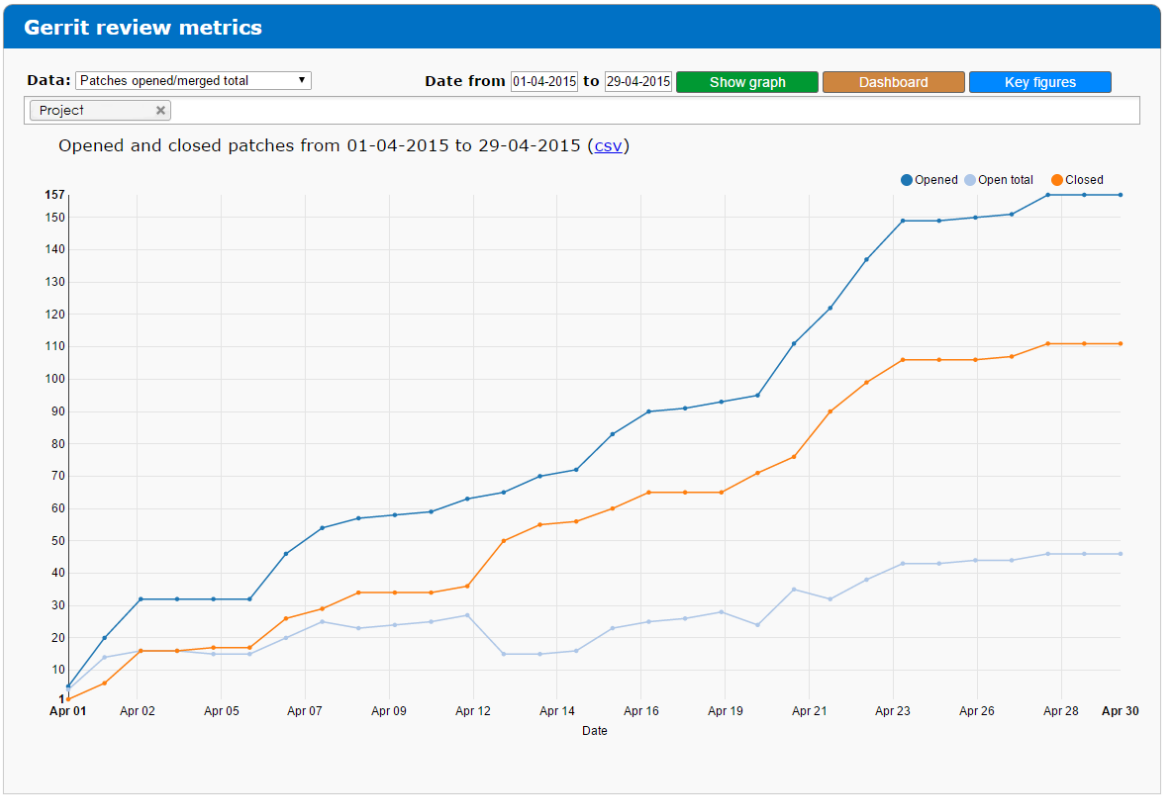

Figure 6: A screenshot of the metrics page showing opened and integrated changes for a sample project.

All the data is retrieved from Gerrit's database, which grows by hundreds or even thousands of rows per day in the use of the projects researched in this thesis. The total amount of changes in the database is tens of thousands. This sets certain requirements for the

performance. While backend needs to perform tasks economically also front needs to render even thousands of data points neatly into charts without crashing user's browser.

Backend

The main task for the backend is to make queries to Gerrit's MySQL database and encode returned data to JSON and serve it for the frontend. It is written with PHP, mainly because that is the language I am most familiar with and I knew it would suit the task very well. The architecture is fairly simple with one file being the target for API calls, which implements the Main class. Depending on the parameters given to the API, the Main class then creates corresponding objects to handle the request.

The Gerrit database is a MySQL database which has hundreds of thousands of rows and it grows constantly. With this amount of data, efficiency plays a big role. MySQL queries need to be well thought and getting the data with multiple queries was not really an option. Therefore data is always fetched with one single query, where efficiency is close to optimal. Still there are some metrics which were thought would be nice, but which were not feasible to implement because they would have required too much computing power.

Frontend

The frontend consists of a single PHP-page, which content is created dynamically with JavaScript. When user chooses a graph to be shown, JavaScript makes AJAX-calls to the backend, which provides the data which is placed to the html template.

While backend needs to efficiently retrieve the data, frontend needs to be equally efficient in converting the data into charts. The amount of data drawn set some limitations to which library to choose for drawing the charts. After comparing alternatives, the nvd3 JavaScript library was chosen due to its capability to produce different charts quickly without much configuration. The library utilizes powerful d3 charting framework, which on its own would have required too many programming hours before it could have been fitted for this project.

Further, nvd3 was selected because it was also used in the earlier version of the metrics site, so it was known to be able to handle the data and charts needed. Nvd3 is an open source library and is written with plain JavaScript and CSS, which made customizing it easy whenever needed.

Training needed?

The metrics page does not have many different options and usability is very simple as user just has to choose which graph to show and for which project and/or team. There is also an option to adjust how many days back the data is fetched. To clarify what each graph is about,

each chart has a short description which can be seen when user clicks question mark next to the select menu.

When the first requirements were closed and the website was running smoothly managers from different projects were questioned for more requirements, especially what kind of metrics they would like to see. From these new requirements, five more graphs were implemented. Some of the graphs would have been too complex and resource hungry to feasibly construct and were left outside of this project. For example measuring lines of code requires access to Git repositories which requires that the program is constantly aware of where the projects are located and then executes Git commands at the locations. While comparing lines of code against review data is interesting it is not ideal to combine Git with this tool because of performance and security issues. I implemented a separate tool to satisfy these requirements.

5.2 Applied metrics

This section explains how and with what kind of tools the code review process can be measured. The metrics that were designed to measure code reviews in Gerrit includes collecting and analyzing various key attributes present in a code review. These attributes can be divided into three main categories, time, quality and count. The goal of the metrics introduced here is to provide easily comparable, meaningful data which helps decision making and gives developers an overview of their current performance with an ultimate goal of speeding up the code review process while improving the software quality.

The time category of the metrics is focused on calculations of how long certain activity during the change life time in Gerrit takes. For example, there is a metric to measure how long it takes until a change has received required code reviews and Approver+1 after it has been declared ready by the author.

The quality category is about measuring the review quality. This doesn't mean software quality but what are the results of reviews and if reviews are done carefully enough and by following the guidelines. Interesting measurements here are for example amount of positive and negative reviews received by a developer or total positive and negative reviews given by a developer.

The count measurements represent the amounts or frequencies of various items present in the review process including the amount of open and closed items or count of patch-sets. These are important figures when analyzing the change lifecycles or quality of patches in terms of complexity.

To generate meaningful data it is essential that right things are measured in the right way. Metrics that are presented in this thesis were picked after discussing with managers and developers what they want to measure and how they want the data to be measured and displayed. Requests came from various projects with different amount of people and slightly different development processes. Metrics were also evaluated to assess their usefulness. It was immediately recognized that there are areas which need improvement and are therefore important to measure.

In general, the metrics that were seen as most important were the ones which measure the time spent for different review activities. Long review times delay the whole project and that is why it is essential to root out any delays. From the proposed requirements the ones that could be applied for most of the projects were prioritized to be implemented at the first version of the metrics. Project specific metrics were prioritized to be done later.

In this research the following measurement areas are discussed:

- Patch lifetime from creation to integration.
- Negative vs. positive reviews per change.
- Negative vs. positive reviews given per developer.
- Negative vs. positive reviews received by developer.
- Number of opened and closed changes.

There were existing implementations which were built for old code review system. These old metrics were used as a starting point when designing the new metrics tool. Also the same open source nvd3 JavaScript graph library that was used earlier was chosen to generate the new graphs as well.

The metrics site is built to easily accommodate new metrics and fresh data is automatically fetched straight from Gerrit's database so it is always up to date. The tool requires very little maintenance. The only maintenance is to adjust metrics to fit review process whenever there are changes to that. Gerrit's database has had only minor changes which have not affected the metrics.

5.3 Time Measurements

When the need for metrics was first discussed with managers, the most wanted metrics were different time measurements. People at all positions are interested in how long different activities take during the Gerrit code review process.

After the developer has uploaded a new change into Gerrit, the change goes into pre-review stage until author declares it to be ready for review. Pre-review stage in the case company

includes continuous integration (CI) and smoke tests. After that the actual code review takes place. That is measured with review time. When necessary amount of reviews are done, and the change has received Approver+1 from the approver, the change is ready for integration. If there is -1 in code review or approver give Approver-1, the change goes back to the author for rework. From the pre-review stage and code review stage come two metrics: pre-review time and review time.

After the review is completed and approver has given the +1 the pre-integration time starts. During the pre-integration time, the change is waiting in the system that the integration begins. In other words, in the pre-integration nothing happens and this time should be minimized. The integrator gives the change an Integrator+1 status as a sign that the integration has started. The integration process can stop to two different events. Either the change becomes successfully submitted or it encounters error during the integration. In case of error, the integrator gives the change -1 which sends the change back to the developer for fixing. In this case the review process starts all over again until the integration can be successfully completed and change submitted.

The integration process is measured with four different metrics. There is the pre-integration time, which is the waiting time before integrator does anything. Then there is the integration time when the actual integration process is happening. If change gets rejected during the integration, two other metrics are needed. Thirdly, there is the integrator reject time, which is the time from start of integration until integrator -1, which describes how long it has taken for the integrator to notice that the change cannot be submitted. Finally there is the integration fix time, which is the time that is needed for fixing the change after first integration failure to finally get submitted. All the times measured are real calendar values, so weekends and holidays are included in them.

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've 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.

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 Gerrit may remain unseen for a while or are easily forgotten.

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.

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 CI 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 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.

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 a 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 shouldn't be long as there is no reason for the integrator no 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.

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 Integrator-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 is measured in a different metric called Integration fix time.

Ideal integration time is less than 24 hours, which means that the integration is done 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.

Integration reject time

The integration reject time is used whenever a change gets rejected during the integration. 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.

Integration fix time

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 life time by tens of hours at the minimum.

The graphs that are built from these time measurements are shown in Figures 7 and 8. The first one shows the review and integration times sorted by value where the ones with the longest change life times are on the left. The second figure shows the same changes sorted by time, the changes being merged last are on the right. These two graphs give two different ways to interpret the data. From Figure 7 it is easy to see how the change life times are divided and the Figure 8 offers a view on what is the trend in life times.

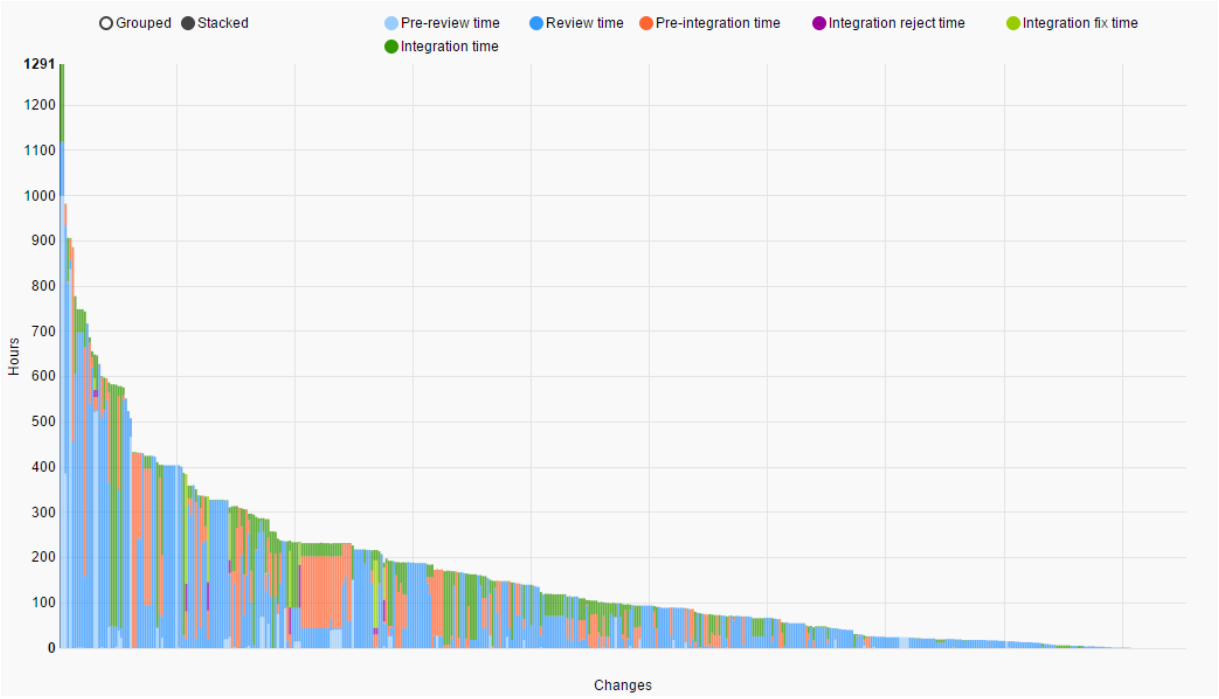

Figure 7: Duration of different phases in code review in in of the case company's projects, sorted by total time.

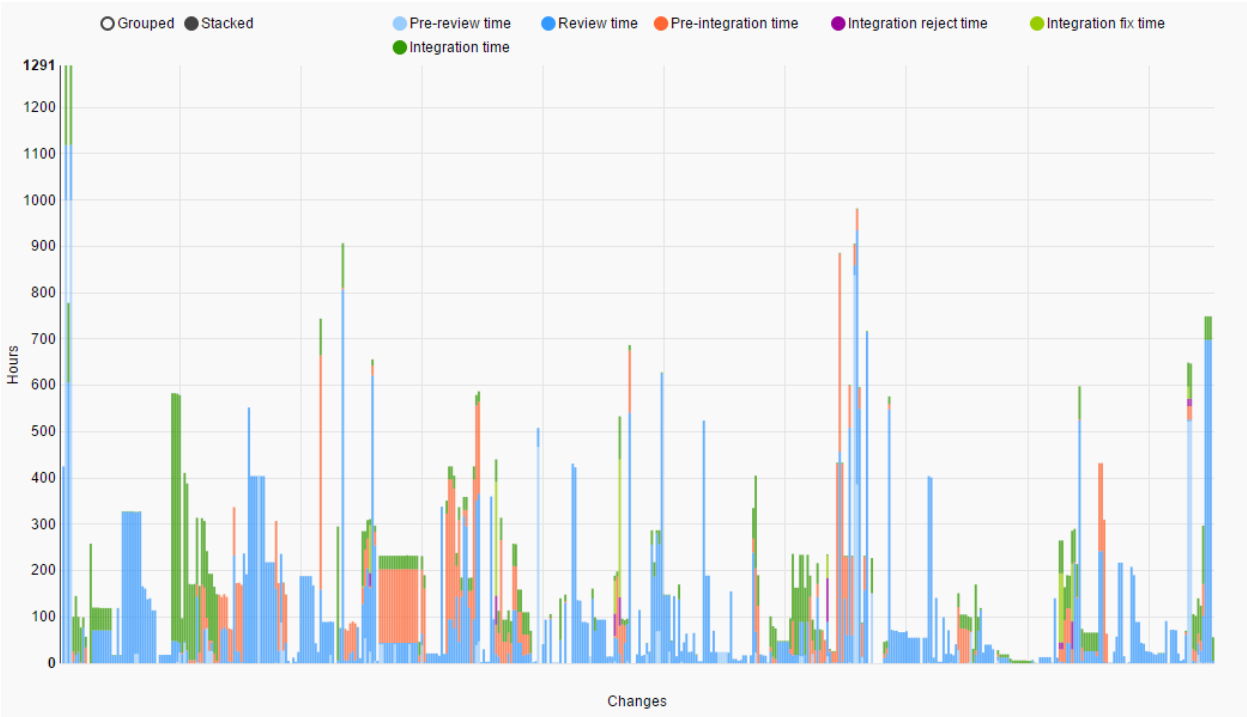

Figure 8: Duration of different phases in review process, sorted by date.

5.5 Count measurements

Average number of patch-sets

The number of patch-sets is a measure of code quality and process smoothness. Smaller amount of patch-sets means that the code change has needed less fixes before 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 is a histogram chart which shows the distribution of number of patch-sets needed for a change. This chart is shown in Figure 9. It quickly tells 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.

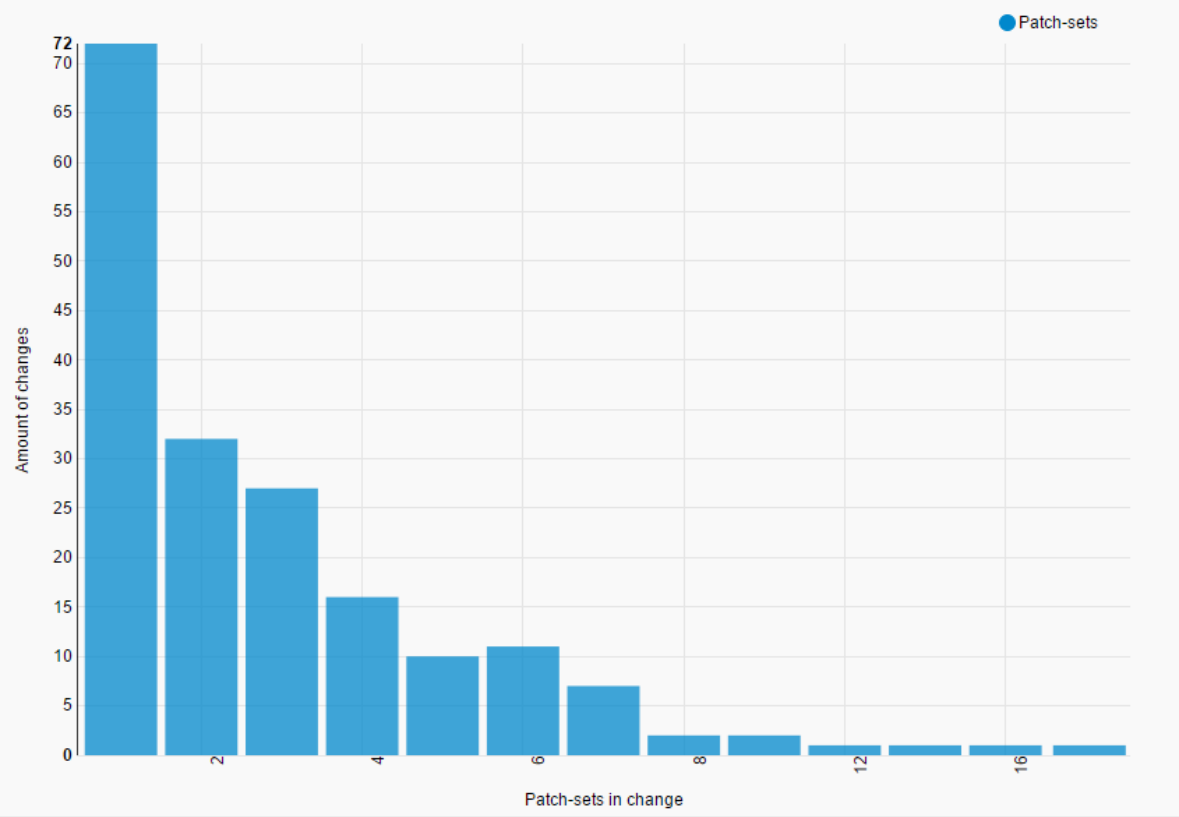

Figure 9: Distribution of patch-sets in an example project. In this project most of the changes have only one patch-set, but there are few changes with over ten patch-sets too.

A good number of patch-sets is around 2-5, which implies that the initially uploaded patch didn't 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 affectively 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. In some projects there are typically two or three patch-sets, but in some the average is around 4, while topping 10 is not unusual. This indicates that at least some of the patches in these projects might be too complex. The complexity should be reduced by making smaller changes, which would make reviews easier and dependent changes would also get merged faster.

Open items

Open items metric counts opened and integrated changes within the selected time period. The difference between opened and closed items gives the number of currently open items, which is a useful measure to get and insight of current pace of opening and closing changes. An example of this metrics can be seen in Figure 10. If number of open patches goes up for extended period of time, it immediately tells that there are some problems which need attention. If open items pile up, they are harder to manage and dependency management and the number of rebases needed become problematic.

As this method doesn't count any opened or closed items before the chosen date, any open items that fall beyond the starting date are not shown. However, if the timeframe is set, say, to around 90 days, any older items are almost certainly outdated as there should not be any older changes in the system than that. Otherwise, we cannot talk about rapid software development anymore.

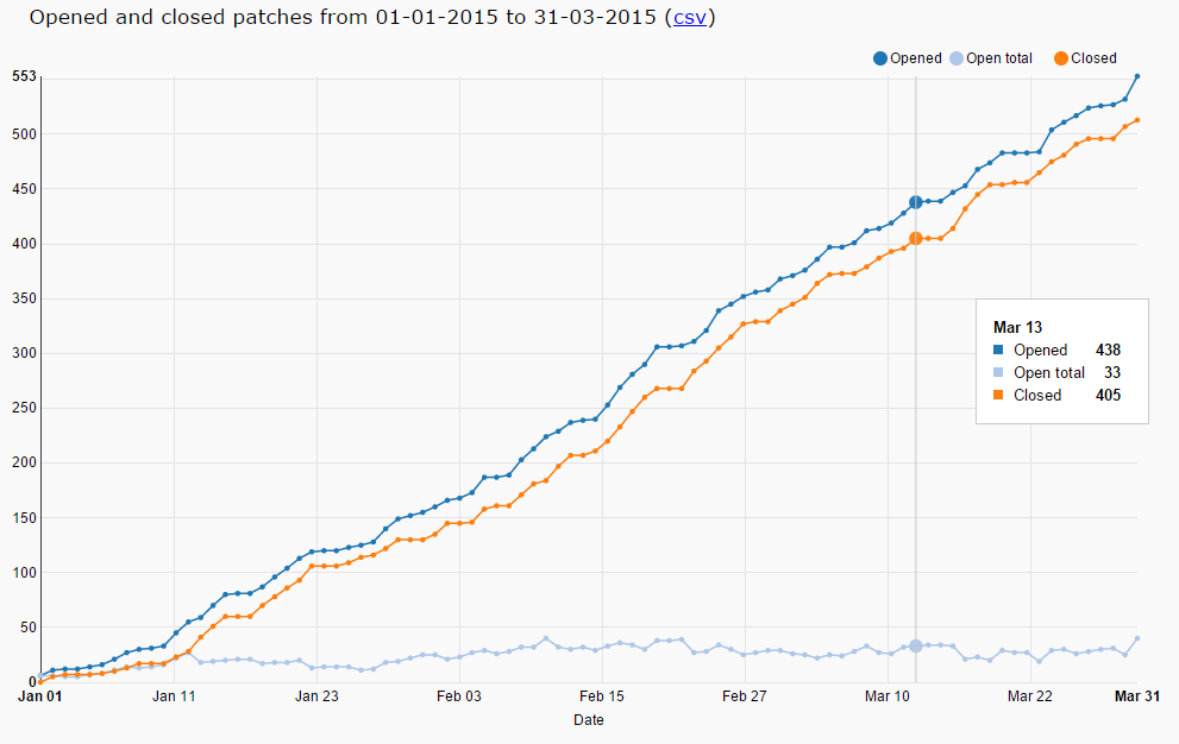

Figure 10: Opened, integrated and currently open changes in one the case company's projects.

5.6 Quality measurements

The purpose of quality metrics is to accompany the time and to count measurements by giving a perspective on what is happening in the code reviews. 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.

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.

Figure 11 describes how different people have given reviews during a three months period. In the Y axis there is the number of reviews done by the user while the X axis represents people involved in the project. It is clear that some reviewers give more negative reviews than others which is opposite than what Beller *et al.* [2014] found in small software projects. Review results of -2 are very rare and during the time period investigated, there were no -2's at all in Figure 11. The distinctly huge amount of reviews done by one user is because the user is in position where he/she has to give code review for every change in that project.

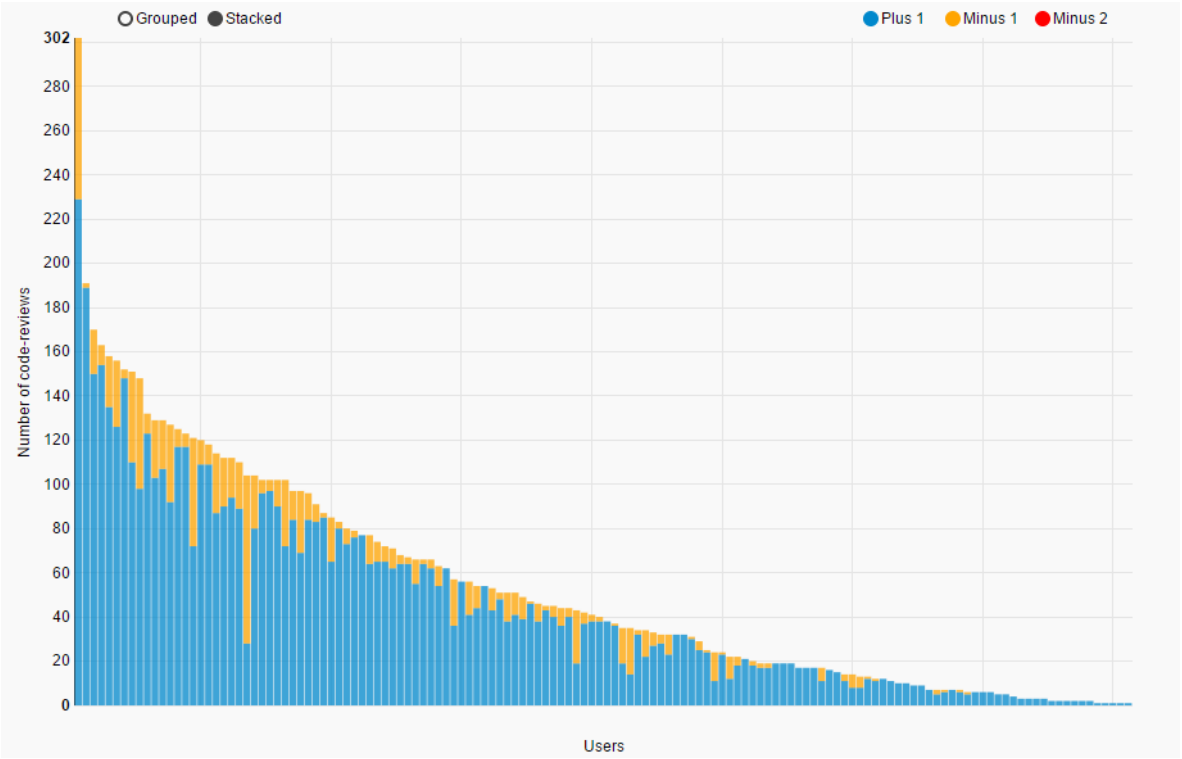

Figure 11: The number of give +1's and -1's per reviewer.

If we examine different project with totally different people, we find similar pattern of varying percentage of negative and positive reviews (Figure 12). From these observations it can be stated that some reviewers give negative reviews more easily than others.

The number of patch-sets can also be used as a quality metric. In an optimal case the change receives two or three patch-sets which indicate 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.

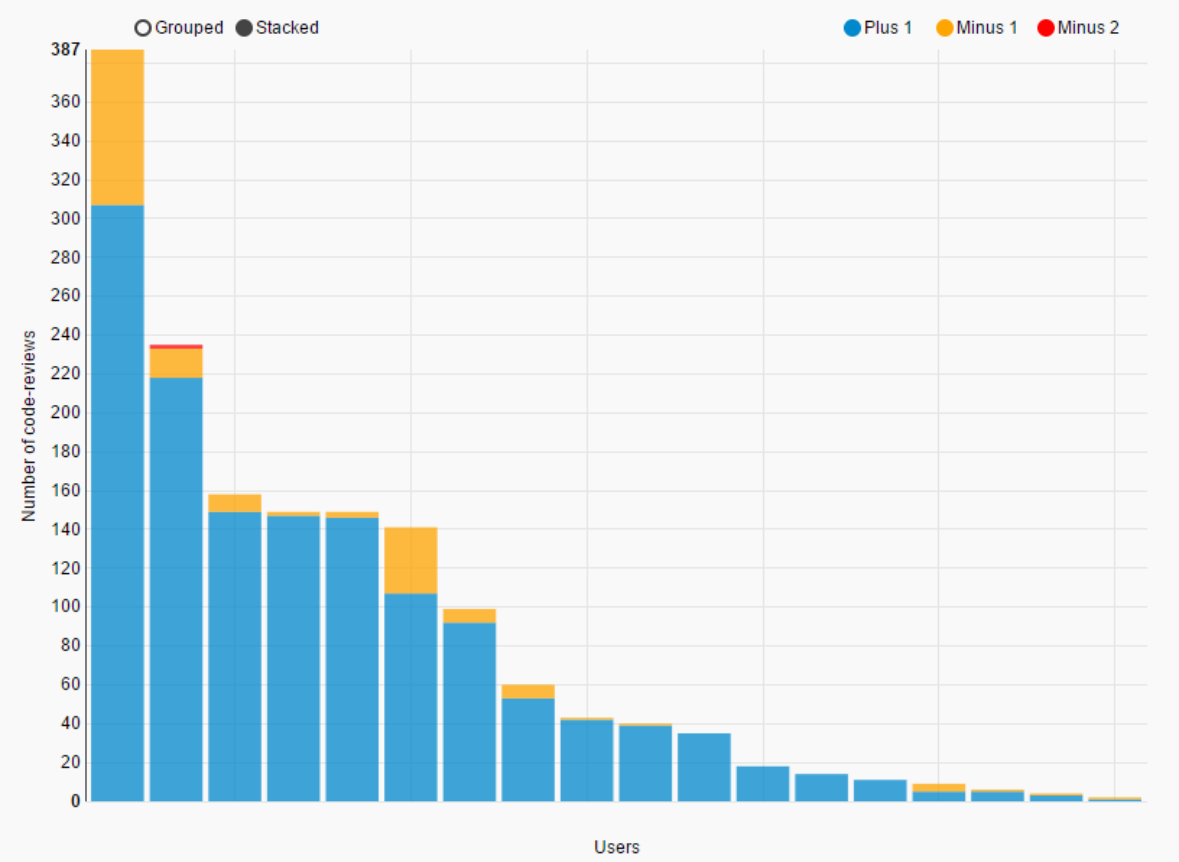

Figure 12: The number of given +1's and -1's per reviewer in another project.

6. Findings

After implementation of the metrics system, the data has been gathered for more than 6 months and there are concrete measurement results. All the implemented metrics have been in use, but the most widely used metrics are the ones measuring the review and integration times and metrics concerning the amount of opened and integrated changes.

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. Each project has different teams developing them. The members of development teams are typically based in one country while integration or validation can be done somewhere else.

The results gained from the four projects are shown on Table 2. The integration reject time and integration fix time are eliminated because the event of integrator intervention is rare and hardly ever happens in some 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.

	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 %)	26 (14 %)	49 (25 %)
Project 4	14(12 %)	119 (59 %)	20 (10 %)	38 (19 %)
Average	**24 (10 %)**	**122 (53 %)**	**31 (13 %)**	**58 (24 %)**

Table 2: Distribution of time in Gerrit code review. Times are hours with percentage of total change life cycle.

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 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 a 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 the case company is in the 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 review the variance between projects is noticeable. The share of changes going through without any negative reviews is shown in Table 3.

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

Table 3: Percentage of changes which receive at least one code review -1.

The project which has received 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 %.

Beller *et al.* [2014] reported a case where the figure of undocumented fixes by the author accounted only 10-22 % which is significantly less than in most projects I researched. That

might be due to different development processes, because in the projects I studied 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*. [2014], I 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 my sample size is hundreds of employees working within one company, but in very different projects with their own ways of work.

7. Conclusion

It has been pointed out in many studies that code reviews and inspections, when done properly, are the most effective defect removal activities. The code improvement and knowledge sharing elements help also in constructing software which is easy to maintain.

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 established. In an effort to recognize new best practices for code reviews in Gerrit code review, this thesis has studied code reviews in a case company and introduced different metrics for measuring the reviews in a number of projects.

For the review process to be efficient in the first place, it is recommended to review every code change that has been made. By using Gerrit it is easy to make sure that every code change is reviewed, as nothing can be integrated without the code going through Gerrit. However, Gerrit doesn't provide a lot of control over the review process itself and therefore additional controls are needed to maintain the quality in reviews.

Measurement of code reviews is rather new area and there are limited number of tools available. Gerrit provides very few readymade key figures or measurements to be utilized and there was a need for set of metrics to increase the effectiveness and control over the review process. The main measurement areas are the time spent in review process and the frequencies of various activities.

In this thesis I have demonstrated that measuring code review process with various metrics, can help controlling the review process on whole while enabling improvement of the 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.

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 for shortening the time required to complete that phase. 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.

Metrics which are focusing on measuring time were considered the most important among the managers. 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.

The case company has constantly evolved the review process since the metrics were first introduced. This limits the amount of valid data that can be used for determining whether the metrics have helped to achieve faster code reviews and better code quality.

Because of that, the metrics were under heavy refinement during the whole project and formulas were adjusted many times to meet the new criteria. This was partly due to the introduction of new features in Gerrit and there was a need to measure new workflows. One of the key changes was the introduction of approver and integrator roles.

However, when investigating change life times from the three month time period where the data is valid, a trend of shorter times can be seen. To get verification to this observation manager and developers from different projects were questioned for their opinion on how they have utilized the metrics to improve the review process. These interviews confirmed that the metrics were successfully helped to identify problems in review process to achieve shorter change life cycles.

While the results have been good, there is still work to do. The metrics that were taken into use in the case company 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 reviewer as 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.

In addition to the measurement areas covered in this thesis, there are many 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 metrics presented in this thesis.

Tomas *et al.* [2008] 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 thesis.

References

[AOSP, 2014] Android Open Source Project (AOSP), http://source.android.com/, retrieved 10th Nov., 2014.

[Bacchelli and Bird, 2013] Alberto Bacchelli and Christian Bird, Expectations, outcomes, and challenges of modern code review. In *Proceedings of the 2013 International Conference on Software Engineering* (2013), IEEE Press, 712-721.

[Bandi *et al*., 2003] Rajendra K. Bandi, Vijay K. Vaishnavi, and Daniel E. Turk. Predicting Maintenance Performance Using Object-Oriented Design Complexity Metrics. *IEEE Trans. Softw. Eng.* **29** (2003), 77-87.

[Basili and Weiss, 1984] Victor R. Basili and David M. Weiss. A Methodology for Collecting Valid Software Engineering Data. *IEEE Trans. Softw. Eng.* **10** (1984), 728-738.

[Beck *et al.*, 2001] Kent Beck, James Grenning, Robert C. Martin, Mike Beedle, Jim Highsmith, Steve Mellor, Arie van Bennekum, Andrew Hunt, Ken Schwaber, Alistair Cockburn, Ron Jeffries, Jeff Sutherland, Ward Cunningham, Jon Kern, Dave Thomas, Martin Fowler, and Brian Marick. Manifesto for Agile Software Development. 2001.

[Beller *et al.*, 2014] Moritz Beller, Alberto Bacchelli, Andy Zaidman, and Elmar Juergens, 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.

[Bicho, 2015] Bicho, Gerrit metrics backend. https://github.com/MetricsGrimoire/Bicho/wiki/Gerrit-backend, retrieved on 12th Mar., 2015.

[Black, 2007] Rex Black, *Pragmatic Software Testing. Becoming an Effective and Efficient Test Professional*. Wiley Publishing, 2007.

[Bosu, 2014] Amiangshu Bosu, Characteristics of the vulnerable code changes identified through peer code review. In *Companion Proceedings of the 36th International Conference on Software Engineering*, (2014), ACM, 736-738.

[Burnes, 2009] Bernard Burnes, *Managing Change: A Strategic Approach to Organizational Dynamics*. Pearson Education. Essex, England, 2009.

[Chacon and Straub, 2009] Scott Chacon and Ben Straub, *Pro Git*. Apress, 2009.

[Cheng *et al*., 2009] Tjan-Hien Cheng, Slinger Jansen, and Marc Remmers, 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.

[Cohen *et al*., 2013 Jason Cohen, Steven Teleki, Eric Brown, Best Kept Secrets of Peer Code Review, Smart Bear, 2013.

[Fagan, 1976] Fagan, M. E. Design and Code Inspections to Reduce Errors in Program Development. *IBM Systems Journal*. **15**(3) (1976), 182–211.

[Feldman, 2004] Stuart Feldman. 2004. A Conversation with Alan Kay. *Queue.* **2**(9) (2004), 20-30.

[Ganssle, 2012] Jack Ganssle. A Guide to Inspections, 2012. http://www.ganssle.com/inspections.pdf, retrieved on 11th Jan 2015.

[Github, 2015] Github home page. https://github.com, retrieved on 12th Mar., 2015.

[Gerrit, 2014a] Gerrit documentation. https://Gerrit-review.googlesource.com/Documentation/index.html, retrieved on 11th Dec., 2014.

[Gerrit, 2014b] https://code.google.com/p/Gerrit/wiki/ShowCases, retrieved on 15th Nov., 2014.

[Gerrit, 2015] Gerrit home page. http://code.google.com/p/Gerrit, retrieved on 12th Mar., 2015.

[Haikala ja Märijärvi 2006] Ilkka Haikala ja Jukka Märijärvi. *Ohjelmistotuotanto*. Talentum, 2006.

[IFPUG, 2015] IFPUG International Function Point Users Group. http://www.ifpug.org/about-ifpug/about-function-point-analysis, retrieved on 13th Mar., 2015.

[ISO, 2015a] ISO 9000. http://www.iso.org/iso/iso_9000, retrieved on 13th Mar., 2015.

[ISO, 2015b] ISO 25000. http://www.iso.org/iso/home/store/catalogue_tc/catalogue_detail.htm?csnumber=64764, retrieved on 14th Apr., 2015.

[Jones, 2008] Capers Jones. 2008. Measuring Defect Potentials and Defect Removal Efficiency, *The Journal of Defense Software Engineering.* June 2008. 11-13.

[Jones, 2013] Capers Jones. 2013. Function points as a universal software metric. *SIGSOFT Software Engineering Notes*. **38**(4) (2013), 1-27.

[Jung *et al*., 2004] Ho-Won Jung, Seung-Gweon Kim, and Chang-Shin Chung. 2004. Measuring Software Product Quality: A Survey of ISO/IEC 9126. *IEEE Software*. **21** (2004), 88-92.

[Kanellopoulos *et al.*, 2010] Yiannis Kanellopoulos, Panos Antonellis, Dimitris Antoniou, Christos Makris, Evangelos Theodoridis, Christos Tjortjis, and Nikos Tsirakis. 2010. Code quality evaluation methodology using the ISO/IEC 9126 standard. *International Journal of Software Engineering & Applications*. **1**(3) (2010), 17-36.

[Kernel, 2014] Submitting patch to Linux kernel project. https://www.kernel.org/doc/Documentation/SubmittingPatches, retrieved on 4th Dec., 2014.

[Kernighan and Plauger, 1978] Brian W. Kernighan and P. J. Plauger. *The Elements of Programming style.* McGraw-Hill, 1978.

[Kupiainen *et al.*, 2015] Eetu Kupiainen, Mika V. Mäntylä, and Juha Itkonen, Using metrics in Agile and Lean Software Development – A systematic literature review of industrial studies. *Information and Software Technology*. **62** (2015), 143-163.

[Mantyla and Lassenius, 2009] Mika V. Mäntylä and Casper Lassenius. 2009. What types of defects are really discovered in code reviews? *IEEE Transactions on Software Engineering*. **35**(3) (2009), 430–448.

[Markus, 2009] Arjen Markus, Code reviews. *SIGPLAN Fortran Forum* 28, **2**, (2009) 4-14.

[Martin, 2009] Robert C. Martin, *Clean Code*. Prentice Hall, Pearson Education, 2009.

[McConnell, 1996] Steve McConnell, *Rapid Development: Taming Wild Software Schedules.* Microsoft Press, 1996.

[McConnell, 2004] Steve McConnell, *Code Complete, 2d Ed.*. Microsoft Press, 2004.

[McIntosh *et al.*, 2014] Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed 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*, (2014), ACM, 192-201.

[MediaWiki, 2015] Git/Gerrit evaluation. http://www.mediawiki.org/wiki/Git/Gerrit_evaluation, retrieved on 29th Apr., 2015.

[Mishra and Sureka, 2014] Rahul Mishra and Ashish Sureka. Mining Peer Code Review System for Computing Effort and Contribution Metrics for Patch Reviewers. In: *proceedings of 4th IEEE workshop on Mining Unstructured Data,* (2014), 11-15.

[Mukadam *et al.*, 2013] Murtuza Mukadam, Christian Bird, and Peter C. Rigby, Gerrit software code review data from Android. In: *Proceedings of the 10th Working Conference on Mining Software Repositories*, (2013), IEEE Press, 45-48.

[Perry *et al.*, 1994] Perry, Dewayne E. Perry, Nancy A.Staudenmayer, and Lawrence G. Votta. 1994. People, organizations, and process improvement. *Software, IEEE*. **11**(4) (1994), 36-45.

[Phabricator, 2015] Phabricator Code review tool. http://phabricator.org/, retrieved on 12th Mar., 2015.

[Porter *et al.*, 1998] Adam Porter, Harvey Siy, Audris Mockus, and Lawrence Votta. 1998. Understanding the sources of variation in software inspections. *ACM Transactions on Software Engineering Methodology.* **7**(1) (1998), 41-79.

[Review Board, 2015] Review Board. https://www.reviewboard.org/, retrieved on 12th Mar., 2015.

[Rigby and Bird, 2013] Peter C. Rigby and Christian Bird. Convergent contemporary software peer review practices. In: *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, (2013), ACM, 202-212.

[Rigby *et al.*, 2014] Peter C. Rigby, Daniel M. German, Laura Cowen, and Margaret-Anne Storey. Peer Review on Open-Source Software Projects: Parameters, Statistical Models, and Theory. *ACM Transactions on Software Engineering Methodology.* **23**(4) (2014), Article no. 35.

[Rombach, 1987], Dieter H. Rombach. A Controlled Experiment on the Impact of Software Structure on Maintainability, *IEEE Transactions on Software Engineering*, **13**(3), (1987), 344-354.

[Siy and Votta, 2001] Harvey Siy and Lawrence Votta. Does the modern code inspection have value? Software Maintenance. In: *Proceedings of the 2001 IEEE International Conference on* Software Maintenance, (2001), 281-289.

[Sommerville, 2011] Ian Sommerville, *Software engineering*, Ninth edition. Addison-Wesley, 2011.

[Tanaka *et al.*, 1995] Toshifumi Tanaka, Keishi Sakamoto, S. Kusumoto, Ken-ichi Matsumoto, and T. Kikuno. Improvement of Software Process by Process Description and Benefit Estimation. *In Proceedings of the 17th International Conference on Software Engineering*, (1995), 123–132.

[Thongtanunam *et al.*, 2014] Patanamon Thongtanunam, Raula Gaikovina Kula, Ana Erika Camargo Cruz, Norihiro Yoshida, and Hajimu Iida. 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.

[Tomas *et al.*, 2013] P. Tomas, M.J. Escalona, and M. Mejias, Open source tools for measuring the Internal Quality of Java software products. A survey, *Computer Standards & Interfaces*. **36**(1) (2013), 244-255.

[Wang *et al.*, 2008] Yanqing Wang, Li Yijun, Michael Collins, and Peijie Liu. 2008. Process improvement of peer code review and behavior analysis of its participants. In: *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science education* (2008). ACM, 107-111.

[Wiegers, 2002] Karl E. Wiegers. 2002. *Seven Truths About Peer Reviews*, Cutter IT Journal, July 2002. Retrieved from http://www.processimpact.com.