Code review with Git

Joacim Åström, atp10jas@student.lu.se
Per-Victor Persson, dat12pp1@student.lu.se

Abstract

Code review is commonly prescribed today to help teams keep defects in the code to a minimum, and help the collective ownership. We will analyze the use of Git, with integration of third-party software, while following some best-practice reviewing methods. We will measure whether or not this makes for more efficient code reviews, quality code and a streamlined workflow.

Introduction

Reviewing code before final commit to a project will significantly increase code quality, and is a great way to find bugs early on [1]. In spite of this, code review has a stigma in agile development as time consuming and largely dependant on tools and processes, and is often not a priority. Some argue that code review can be sufficiently performed while programming in pairs, where there is already an extra set of eyes continuously inspecting the code whilst developing. While this may find some errors, we believe that having a secluded peer review could identify mistakes that slip by the authors of the code.

In a study by Winkler and Biffl [2], they compare pair-programming with a form of code inspection, and conclude that pair-programming is far more efficient in small teams, and find more defects than inspection. However, when the team and the project grows larger, a mix between inspection and pair-programming yielded the best results. The defined size of a team is a bit abstract, but we decided to use a mixture of the two, and combine it with tools available through the use of Git.

We provided our students with easy-to-use tracking tools for code coverage and automated testing, and presented a workflow with story branches, which forced the team to review code before it is merged into the project. By doing this, we hope that the team will get motivated to perform efficient code reviews, and to maintain a clean repository.

This report will contain the following sections:
Background, containing background information on the report and the tools we use in the study.
Methods, describing our methodology with sections on the workflow and also how we use the tools we have chosen.
Results, discussing and commenting on the results of the survey we sent out and the project as a whole.
Conclusion, where we present our conclusions based on the results.

Background

When we took the Software Development in Teams course, there was always a bottleneck in production, namely code reviews. They were time consuming and often unproductive, which resulted in review tasks being piled up and never really dealt with, which in turn caused the repository to repeatedly fail its tests and include a lot of bad smelling code. This was able to occur since all code was pushed directly into the master branch, and reviews were not necessary to progress in the project. In this study, we aim to get the team more motivated to

performing code reviews by providing tools that hopefully makes the process easier. Furthermore, using Git branches, we will ensure that the team never merges code that has not been reviewed, thus forcing the team to make them a priority to be able to progress with the project.

In a white paper written by Smartbear [3], they explain how a team can properly perform efficient code reviews, without the use of extensive meetings between developers. Through lightweight code reviewing, performed in small iterations, they managed to decrease the time used for reviews by 75%, whilst still maintaining the same level of quality. In this study, we will incorporate three of the lightweight methods they mention, mainly

- Pair programming
- Peer review
- Tool assisted review

Pair programming

This form of code review is the fastest form of feedback available, and is performed through pair programming where one developer writes code, while the other navigates and gives direct feedback on the code that is being written.

Peer review

By passing the code to an independent party, the code is being reviewed with a clean slate. This can help identify problems that the developer had not thought of, or issues that the developer simply became blinded for when writing the code. Furthermore, peer review is a very effective way to ensure collective code ownership. The peer review will be handled with the use of git branches, which are not to be merged until thoroughly reviewed.

Tool assisted review

To make the review easier and more more efficient, the use of third party tools can help gather useful metrics. Besides using Git for the ability to create branches, we set up two additional tools, CircleCI and Coveralls. CircleCI is not yet available for integration with Bitbucket, which was provided for the course, hence our team instead used Github.
We used both of these services as a quick sanity-check for the reviewers, so that they can easily see that the code has been tested.

CircleCI

CircleCI allowed us to run all tests in the cloud every time a commit was pushed to a branch. GitHub then integrated with CircleCI to show if the pull-request (essentially a merge-request) for a branch contained any failing tests. This will enforce that all tests are run continuously and alert if a story is not yet ready for review or merge.

Coveralls

We also integrated Coveralls, which adds code-coverage information on the pull-request as a comment. By displaying this information, the reviewer can quickly see how the total code-coverage will be affected by the committed code, and thereby if the code has been properly tested. If the branches code-coverage is not greater or equal to the master branch, better tests needs to be written.
However, due to missing functionality in the java test-running libraries we were not able to do this and had to settle for a coverage counter for the master branch on the repository web page. Code coverage could be viewed for a branch by visiting the coveralls website, which we did not inform the students of until a few iterations had passed, to be able to measure whether this tool was useful or not.

Methods

This section will discuss and motivate the tools and the workflow that the team used, and how the methodology was introduced to the team.

Team introduction

Many team members had, as could be expected, very little previous experience with Git apart from obligatory exercises in the course Software Development in Teams. During the very first meeting with the team, every member was assigned to create an account on github, if they did not have one already. After a short discussion, we concluded that a brief introduction to Git was necessary and consequently performed, where we ran the team through the basics of branching and merging.

Workflow

Before we met the team, we devised a workflow that would then drive the code review process. This workflow was based on the one in Vincent Driessens “A successful Git branching model” [5], but we simplified it to make it easier for the developers to work with since we wanted to test the essence of the workflows. We did this by cutting down the use of branches to only have a master branch and story branches (what Driessen calls feature branches).
The code was stored in a git repository hosted on GitHub.

Git

Git provides very simple and efficient branching, since a branch is just a pointer to a commit, it is very easy to change branches. We leveraged this by creating a new branch for every story, and merging the changes back into master when the story had passed review.
The story branches then only contained the differences in the code needed for the story, and since branching and switching branches is simple, it becomes easier for the reviewers to switch branches and only review the changes for that particular story.
We also tried to get the team using git in the terminal, instead of using built-in plugins in eclipse. Our rationale for this was that it would help the developers get familiar with the tool and its concepts faster if they did not work with it through an abstraction.

Kanban

When a pair of developers start working on a story, they first pick move the card on the Kanban board from “Todo” to “In Development”. The Kanban board is the first part of the workflow and structures the rest of the process. Having the story cards physically available in the room helps keep the entire team aware of the current state of development. We began with five steps on the kanban board:

1. Todo
2. In development
3. Done
4. Reviewing
5. Merged

After iteration 3 we also added a sixth step “Accepted by customer”, due to customer demand. For the story to be able to move from step 2 to step 3 (Done) the story had to be finished, tested and refactored. After the story was marked as done a new pair had to move it to the review stage and review the code. The review step consisted of having the new pair pull the code the their local machine, view the diff of the code, verify that the tests passed and also verify that it matched the story specification. In contrast to many similar workflows we also had the pair that performed the review merge the branch into master if it passed the review, else it was their responsibility to fix anything wrong with the code.

Git Cheat-Sheet

Before you get a routine and get familiar with all useful commands, git can be rather confusing. Since tools and methods should not consume valuable time that could be spent on development, we initially intended to author an easy-to-follow cheat-sheet for the team to use whenever they needed to perform a git-related task. However, during the first planning session, we realized that coaches as developers often do not speak the same language when it comes to technical terms. After discussing it with the team, we decided it would be appropriate to delegate composing said cheat-sheet to the group in the form of a spike

assignment. Requirements for the cheat-sheet was to cover set up and installation, as well as any or all operations needed to follow the workflow we wanted them to follow, as described in the next section.

Initial architecture

The team was provided a simple skeleton application containing very simple classes with faked functionality and a test-running harness, testing the fake functionality.
It also contained basic configuration and the needed configuration to run the test suite online with CircleCI. This allowed us to easily demonstrate the functionality of Coveralls and CircleCI, and what to look for.

Survey

To measure if our method and workflow had a positive impact on the project, we released a survey to our team close to the end. These included questions of how many reviews the individual performed, how much time was spent, how many errors were identified etc. It also included general attitude towards the methodology, and invited to explain why something worked well or not. The survey provided us with the metrics presented in the results section.

Results

Github as tool

During the first programming session, a lot of time was spent trying to set up the work environment. However, thanks to the cheat-sheet provided by the team, any operations related to github, e.g cloning the repository and branching out, was relatively painless. Instead, it was eclipse as an IDE that created problems when importing the project and including libraries. At this point, the cheat-sheet was a great success to get everyone started.

Bottlenecks

During the course of the project, the team never had more than one or two stories on hold to be reviewed, no bottleneck arose were reviews stacked up and hindered the progression. This was likely a direct result of using story branches instead of pushing directly to master. Since many stories have dependencies, they could not be initialized, much less finished, without certain stories merged to the master branch, and to do so, it would have to be reviewed. Thus, reviews were viewed as high priority tasks and appreciative work.

The time for a story being idle as “done” on the Kanban-board could vary a lot. This was in no way a result of some stories being less important to review, but simply that the developers preferred to finish their own tasks before engaging in a review. That being said, a story

awaiting review was never overlooked by an available developer to prioritize other stories, but dealt with as soon as an opportunity arose. Because stories sometimes had to wait quite a while, this form of review in a way contravenes the values of extreme programming [6], which encourages fast feedback. Therefore, it is a good practise to combined peer review with pair-programming, which provides instantaneous feedback.

Red repository

Using story branches that have to pass all tests before merging into master, it should theoretically be impossible for the master branch to fail its build or tests. It did, however, occur twice during the projects course. The first time was because a class did not have a valid package declaration. This error ought to have been spotted during review, since attempting to compile the code would have thrown the error. This can be attributed to the developers not being experienced enough, but also us as coaches who should have worked with the developers to establish a better definition of done that included running the code during review.

A second time occurred due to the Eclipse Java compiler accepting code that the regular javac compiler would not, thus the code could not be built when CircleCI performed its tasks on the master branch.

Overall, since the master branch was mostly green, it was very easy for the team to quickly produce a release.

Survey results

During the second to last week we sent out a survey to gather data from the developers to gauge the effectiveness of the workflow and using git to perform code review.

The average time spent on reviews hovered below 35min, though there is an outlier at three hours. This extremely time consuming review was a result of a story that a programming pair thought was sufficiently developed, but in fact had completely misunderstood. Since a majority of the functionality had to be reworked, this should probably have become its own story, and not logged as a review. In a study by Bacchelli and [4], they conclude that a lot of the time spent on reviews is to simply understand the code. This number is likely higher in our study, since the team exists of junior developers with little experience. A crucial factor to reduce the time for understanding is writing proper comments explaining any or all methods in the code.

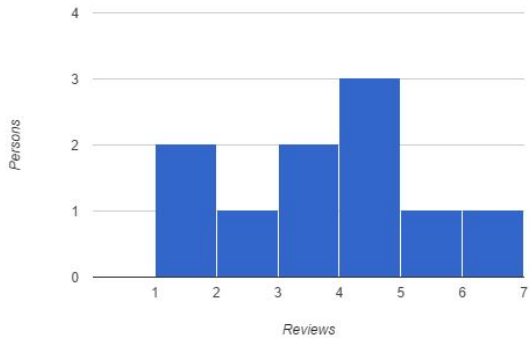
We also asked the developers about the amount of reviews they had done, and the majority of the developers had done three or more reviews each.

As we also had the reviewing developers fix any issues that came up during the reviews we were also interested on how much actually had to be fixed after the story was “done” and in the review phase. The answer was that most of the time was spent on finding problems, but some amount of fixes had to be done. Combining this result with the time spent explains the outliers, the result is however somewhat skewed since some of the stories required extensive

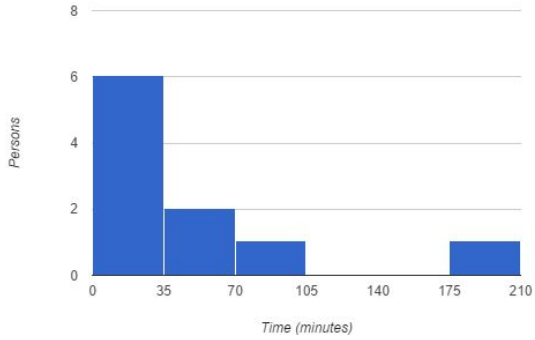
fixing before they were actually done. This was likely a result of the lack of developing experience by the team, combined with a lack of communication with the rest of the team, which could have helped by sorting out what was required for the story to pass. However, the majority of the stories worked and passed the review.

Lastly, we asked a few questions regarding the developers opinions on the workflow.
We asked them on a scale of 1-5 if the peer reviews had increased the quality of the code, which the majority agreed with. The overwhelming majority also found story-branches and being able to quickly see that the tests had passed when reviewing to be beneficial.

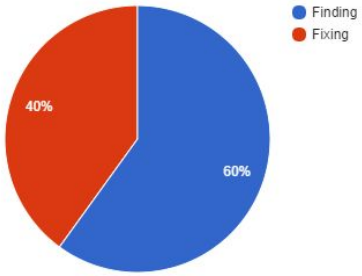
How many reviews have you performed?



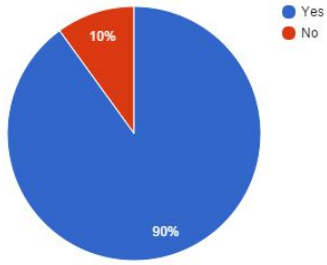
How much time do you spend on a review, on average?



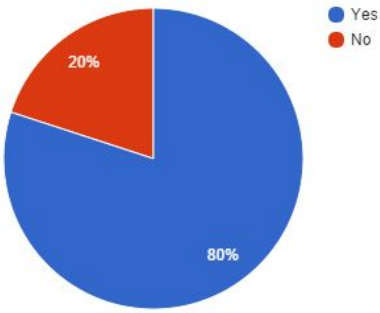
Do you spend more time finding or fixings bugs in a review?



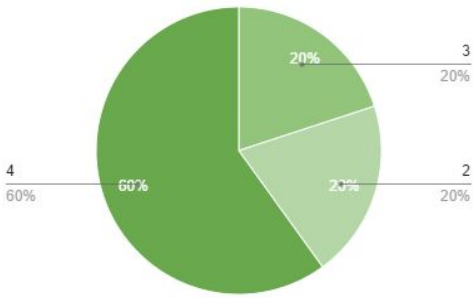
Do you think having story branches instead of working in a master branch has worked well?



Has it helped you to be able to see that the tests pass on the branch you are reviewing?



Do you feel that peer review have increased the code quality, on a scale from 1-5?



Conclusions

All in all, the workflow and the reviews were successful, the team had mostly clean code and could quickly produce a release knowing all code had been thoroughly reviewed and tested. The workflow did however require some extra technical knowledge with branching, which at times was confusing for the team, how to merge, rebase etc. Therefore, a more extensive introduction to git could be preferred, with some practical and visual examples of how it works.

That being said, using story branches makes for easier reviews, since all the code and changes pertaining to the story is localized to the story branch. This means that the reviewers don't have to filter through the commits for the ones they want to review, or that they have to look at code that is not part of the story.
Another strong case for using branches is being able to hold off on merging code with the master until it has been reviewed. Using story branches instead of always pushing to master also allowed the team to choose when to merge and take the merge conflicts while still having the code available for the rest of the team to view.

60% of the team agreed that code quality was greatly increased through peer reviews, and almost everyone had a good attitude towards story branches and reviews before merging. This is crucial, so that the team feels committed to enthusiastically perform reviews instead of just performing them to get on with the work.

Some reviews reportedly took way too much time, but during these reviews, majority of the time was spent on writing code and fixing problems. Had these reviews not been enforced by the workflow, a lot of faulty code would have entered the master branch and greatly increased the time to troubleshoot it.

Ultimately we find that using Git to streamline code reviews was a success, and delivered great experience to the team on how to properly use version control systems. The team made reviews a priority and an obvious part of the workflow. Reviews were performed as soon as a team was available and the repository was almost always clean. The only downside was that the feedback from peer reviews is rather slow, and not in line with the extreme programming fundamentals of fast feedback.

Future Work

Easy avenues for future studies could be changing the tools or workflow in the methodology. More interesting paths could be to combine this workflow other processes like continuous integration or working remotely, since this would present an increased need for collective code ownership.

References

1: Bernhart, Mario, Andreas Mauczka, and Thomas Grechenig. "Adopting Code Reviews for Agile Software Development." *2010 Agile Conference*, 08 2010. doi:10.1109/agile.2010.18.

2: Winkler, Dietmar, and Stefan Biffl. "An Empirical Study on Design Quality Improvement from Best-Practice Inspection and Pair Programming." *Product-Focused Software Process Improvement Lecture Notes in Computer Science*, 2006, 319-33. doi:10.1007/11767718_27.

3: "Code Review: An Agile Process." Accessed March 06, 2016. https://smartbear.com/learn/code-review/agile-code-review-process/.

4: Bacchelli, Alberto, and Christian Bird. "Expectations, Outcomes, and Challenges of Modern Code Review." *2013 35th International Conference on Software Engineering (ICSE)*, 05 2013. doi:10.1109/icse.2013.6606617.

5: Vincent Driessen. "A Successful Git Branching Model." Nvie.com. Accessed March 06, 2016. http://nvie.com/posts/a-successful-git-branching-model/.

6: Beck, Kent. “Extreme Programming EXplained: Embrace Change.” 2000.