Teaching Code Review Management using Branch Based Workflows

Stephan Krusche
Technische Universität München
Munich, Germany
krusche@in.tum.de

Mjellma Berisha
Technische Universität München
Munich, Germany
m.berisha@tum.de

Bernd Bruegge
Technische Universität München
Munich, Germany
bruegge@in.tum.de

ABSTRACT

Developing software with high code quality in a university environment is a challenge for instructors of software engineering capstone courses. Teaching students how to achieve quality and how to conduct code reviews in projects is often neglected, although it helps to improve maintainability.

In this paper we describe an informal review technique: branch based code reviews. Developers realize requirements in feature branches. Before they integrate source code into the main codebase, the code is reviewed asynchronously over the Internet in a quality gate to identify defects, design flaws and code flaws. Traditionally reviewing has been a task of the instructor. In our course, we delegate it to students of the development team. Our hypothesis is that students learn and adapt best practices from each other, improving code quality and their coding skills.

We applied this technique in a workflow during a project-based capstone course over the period of three semesters. 300 students conducted 2939 code reviews with 4665 comments in 33 projects with industry customers. We evaluated the workflow in a qualitative study using interviews. Our key findings are that students do not longer see reviews as a bureaucratic burden and improve their skills through the comments of experienced team members. They are convinced that reviewing code leads to higher code quality and 89 % want to use the workflow again in future projects.

CCS Concepts

•**Software and its engineering** → **Software configuration management and version control systems; Software version control; Software development process management;** *Agile software development; Software evolution; Programming teams;*

Keywords

Peer Learning, Capstone Course, Code Quality, Branching Model, Pull Request, Merge Request, Pair Programming, Distributed Version Control

To appear in the 38th International Conference on Software Engineering - ICSE, May 2016, Austin, TX, USA.

1. INTRODUCTION

The organization of multi-customer capstone courses with real industry customers is challenging for instructors. They often choose toy projects with fabricated problem statements to minimize the effort. However, we claim such setups do not motivate the students and result in low learning experience. We consider large project courses with multiple customers the most promising way to teach industry relevant software engineering practices, in particular agile methodologies and continuous delivery [7]. We conduct such courses since 2008 with up to 100 students, ranging from sophomores to master students in their final semesters with some years of development experience. Students develop mobile applications for industry clients, e.g. BMW, Siemens and T-Systems, during one semester [7]. This allows students to experience and apply agile methodologies with changing requirements and real deadlines, which in fact leads to real team experience. Students face typical communication problems and improve their non technical skills as well.

In 2012, we incorporated continuous delivery into the capstone course [25]. Its introduction increased the number of releases significantly [26] and led to many benefits, also described by others [9]: accelerated time of releases, improved product quality and customer satisfaction and reduced risk of release failure. However it also posed new challenges. The focus on concurrent feature development complicates collaboration on source code. The client's expectation for multiple releases and the inclusion of feedback to improve the product increments increased the time pressure during development. This pressure resulted in less attention to object design and source code quality, which ought to be important when teaching software engineering: Students should learn to write maintainable code with high quality.

Before we introduced continuous delivery, we held code reviews with each team two weeks before the course end. The course instructors reviewed the code since they were experienced with the development environment and programming language. Reviewing the codebase and searching for common design flaws and unfulfilled coding guidelines consumed a lot of time. In retrospectives with students, we asked for feedback about the code reviews. They stated that late code reviews led to almost no learning experience. As the end results of the course projects further improved, more customers want to build upon the codebase of the projects and extend the applications. They hire students after the course to finalize and release the software. In such cases, further developing the software was challenging and led to higher maintenance costs due to low code quality. To address

this problem, we introduced a new approach for source code collaboration and reviews with the following goals:

1. **Early stage reviews**: The code is reviewed from the beginning of the project to adapt the fail early principle and to learn from mistakes as soon as possible.
2. **Continuous reviews**: Reviews are conducted regularly to guarantee high quality in the main codebase.
3. **Review responsibility**: Students conduct the review themselves to improve their learning experience and to reduce the effort for the instructors.
4. **High quality releases**: Only reviewed code is integrated to the main codebase and is present in product increments.
5. **Efficient reviews**: Each change is only reviewed once before it is integrated.
6. **Fast development process**: Reviews do not slow down the development process and the ability to release new features quickly.

In this paper we present a branch based code review workflow that fulfills these goals. We define the term quality, present a taxonomy of review techniques and describe the capstone course in which we applied the workflow, in Section 2. In Section 3, we give an overview of the workflow and describe benefits and challenges, while Section 4 provides details for each activity of the workflow. We describe the approach how we introduced the workflow in a capstone course with a project-based organization using introduction courses and interactive tutorials in Section 5. We evaluated our approach via interviews qualitatively and measured the usage of the workflow quantitatively. We present observations and statistics in Section 6, including key findings and limitations. Section 7 describes similarities and differences to related work, while Section 8 concludes the paper.

2. BACKGROUND

In this section we define quality, explain our taxonomy for review techniques shown in Figure 1 and describe our capstone course briefly. The topic of product quality has been investigated from various perspectives. Renown quality experts either take the stance that quality means "conformance to requirements" [13] or define it relative to the user's needs [15] and their "stated or unstated, conscious or merely sensed" [19] requirements. Another component of quality includes patterns, which describe generic solutions for recurring problems within a particular context using proven concepts [21]. The pattern solution has consequences in addition to the benefits. When change occurs and the consequences become "decidedly negative" [6], patterns devolve into antipatterns. An antipattern has a refactored solution: a "commonly occurring method in which the antipattern can be resolved and reengineered into a more beneficial form" [6]. Another knowledge base for recognizing mistakes are code smells, defined as "indicators that usually correspond to a deeper problem in the system" [20].

We define quality as conformance to flexible specifications that respond to the changes of the user's needs, in addition to the usage of corresponding patterns to address nonfunctional requirements if applicable, while avoiding antipatterns. We consider code quality to be a subclass of quality, focusing on functional requirements, system architecture, design patterns and coding guidelines, avoiding development antipatterns and code smells. Refactoring becomes essential for improving quality, as it helps to remove both code smells and problematic solutions from antipatterns.

2.1 Reviews

The Oxford Dictionary defines review as "formal assessment of something with the intention of instituting change if necessary" [35]. This definition also applies to software engineering, where reviews are an important quality assurance method that check for defects, deviations from development standards, and other problems in products [10, 31]. Weinberg states how early software developers, even the likes of von Neumann and Babbage, understood that correctness was too difficult a task to master by oneself, and sought their colleagues' feedback [38]. These initial reviews were informal in nature, as there was no defined or agreed upon process. In fact, well defined reviews eluded research interests well into the 1970s. The explanation Weinberg [38] offers is that "the need for reviewing was so obvious to the best programmers that they rarely mentioned it in print, while the worst programmers believed they were so good that their work did not need reviewing".

An early approach to reviews found in literature is a formal, well defined and heavyweight process called inspection [30]. Software inspections were developed under the direction of Michael Fagan in an effort to improve quality and increase productivity. Fagan's inspection process consists of six activities: planning, overview, preparation, inspection, rework and follow-up. The first three lay out the foundation: the author determines what materials are to be inspected and ensures that they meet predefined entry criteria. Then, a meeting is scheduled, the participants are chosen and each is assigned one of the four inspection roles: designer, coder, tester and moderator. This set of roles ensures that the material is reviewed from various perspectives to identify different bugs. [17, 18]

Walkthroughs are lower in formality than inspections [30]. There are different approaches to define the process ranging from formal ones such as Yourdon's structured walkthrough [40] or the IEEE Standard 1028 [24], to informal ones that focus mainly on the walkthrough meeting [34, 3]. They all have in common the fact that the author walks an audience of reviewers step-by-step through the material, while explaining the purpose and reasoning behind it. There is consensus on the purpose for walkthroughs to evaluate and improve the quality of the materials by finding defects, suggesting (alternative) solutions, checking conformance to standards, educating the audience on the materials and training new team members.

We differentiate informal reviews based on their flexibility to forego formal meetings, to be conducted as-needed and to tailor the processes by making it more lightweight. Moreover, planning is limited to choosing the reviewers and asking them for feedback. They often only include the code author and a few reviewers with programming or testing background [30]. The review results need not be explicitly documented. Listing the remarks or revising the document is in most cases sufficient. Thus, informal reviews can be simplified down to a cross reading in an author-reader cycle [34]. Informal reviews have mainly focused on source code, due to the tools developed to help conduct code reviews and the techniques specific to programming. Figure 1 shows a taxonomy for reviews including variations for informal code reviews.

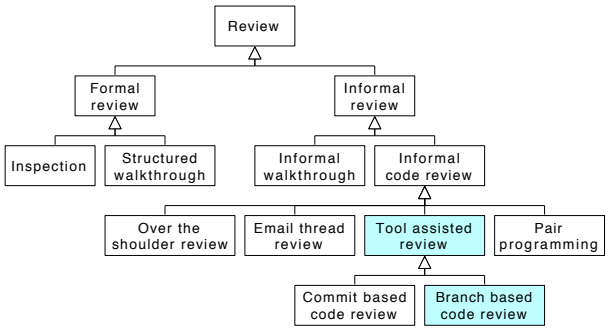

Figure 1: Review taxonomy

A code review takes into account the architecture and code guidelines of the particular project. An additional activity specific to code reviews is the integration needed to merge the individual code changes with the project's shared codebase. The rest of the review process is tailored to apply to source code. When adopted to incorporate integration, the informal review process typically includes the following four activities as shown in Figure 2: preparation, examination, rework, and integration. Depending on the review type, activities are skipped to make the process lighter and more flexible.

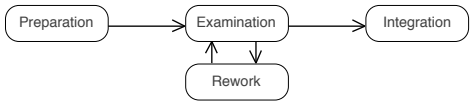

Figure 2: Activities for informal code reviews (adapted from [12])

There are variations of informal code reviews as shown in Figure 1, which Cohen [12] categorizes as: Over-the-Shoulder Review, E-mail Thread Review, Tool Assisted Review and Pair Programming. We can distinguish two types of tool assisted reviews: commit based and branch based code reviews. The difference between them is whether the review is conducted on a single commit or on an entire branch that may contain multiple commits. This paper describes an informal review technique for tool assisted reviews that we integrated into our capstone course: branch based code reviews.

2.2 Capstone Course

We conduct a capstone course each semester. It consists of 10-12 software projects with industry partners who expect a functioning application at the end of the course. Typically, 75 % of the students in our course are graduates while 25 % are undergraduates. The majority of the students study computer science. The challenge is to work with customers from industry to solve a real problem meeting real deadlines in one semester. This setup is ambitious and requires high effort from students and teaching assistants who prepare and organize the course. However, such a course lets the students experience real communication in a real project team with up to 10 developers, a coach and a project leader.

We describe the course in [8], [25] and [26]. In this paper we summarize the aspects that influence the code review workflow. The lifecycle model of the course includes different workflows such as review management. Before the capstone course starts, we conduct a one week introduction in which we teach advanced programming techniques. After that, the actual course starts with a course-wide kickoff where everybody gets to know each other. In this meeting, customers present their project ideas and try to convince students for their project. After the team allocation the students start to work on the project. The first milestone is an initial empty release built on the continuous integration server and delivered to the customer [25]. The teams develop in an agile way using the concept of sprints following Rugby [26], an adapted version of Scrum [33]. Each sprint leads to a product increment, a mature executable prototype. While we except each team to deliver at least one release at the end of each sprint, we motivate students to deliver the current version of the software to the customer whenever they want to obtain feedback (event-based).

After two third of the course we synchronize all course participants in the design review, an important milestone. In this meeting each team presents its current development state to all other teams and to all customers. At the end of the course, all teams present and demonstrate their final application in another course-wide meeting, the client acceptance test (CAT). Two instructors organize the course. Each project team includes a project leader, a coach, the development team as well as a customer. The project leader is a teaching assistant, his role is comparable to a scrum master [33]. The coach is an experienced student, who took the course as a developer. He learns agile project management by observing the project leader and helps the team with design questions such as the selection of an architectural style or design patterns.

A group of 6-10 students form the development team, which is self-organizing, cross-functional and responsible for development and delivery. The customer is an employee of the company and takes over the role of the product owner [33]. Project leaders and coaches report their project status weekly and discuss important issues. Additionally, we build cross-project teams consisting of a developer of each project team, each responsible for a certain topic. More customers of the course now build upon the codebase of the projects, therefore code quality became a focus in recent years. We introduced a cross-project team which coordinates code review related workflows, coding guidelines, the detailed software architecture and how the architecture is mapped to code.

3. WORKFLOW

In this section we describe a branch based code review workflow. Using distributed version control systems (VCS), developers share the same codebase, but separate their work into branches. While centralized VCSs such as Subversion are easier to use and faster to learn, distributed VCSs such as git provide more possibilities, in particular to commit locally (offline) and to create and merge branches fast and easily [5]. Lightweight branching allows context switches and exploratory coding [29]. A branching workflow defines when new branches are created, merged and deleted. Branch management is the activity of defining and controlling these workflows [23, 27, 37]. There are different branching models, e.g. git-flow [16], which handle feature, bugfix, release and hotfix branches. We use a simplified version of the git-flow branching model, shown in Figure 3.

Developers realize requirements on feature branches, use a development branch for the integration of realized requirements and a master branch to store released versions. When implementing a new feature, developers create a new feature branch and commit all related changes into the branch. Meanwhile other developers may work on other feature branches

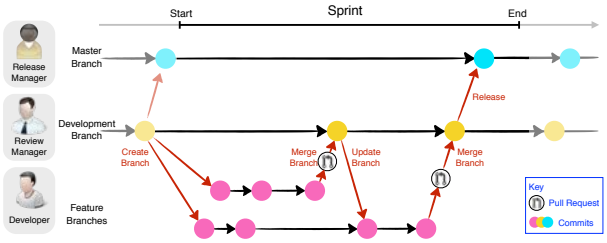

Figure 3: Branching Model (adapted from [16])

and may have already integrated their changes back into the development branch. The developers then need to pull these changes and merge them into their feature branch. When they have realized the feature, they run unit tests locally and on the continuous integration server (not shown in Figure 3). If all test cases pass, they request a merge into the development branch to integrate their changes. This merge request, also called *pull request*, is supported by several platforms such as GitHub, BitBucket and Stash. It acts as quality gate that prevents code with bad quality from being integrated into the main codebase in the development branch. When a developer files a pull request, he is requesting that the review manager accepts and then pulls the changes from the feature branch into the development branch, therefore the name pull request is used. We use pull requests to implement a code review workflow as shown in Figure 4.

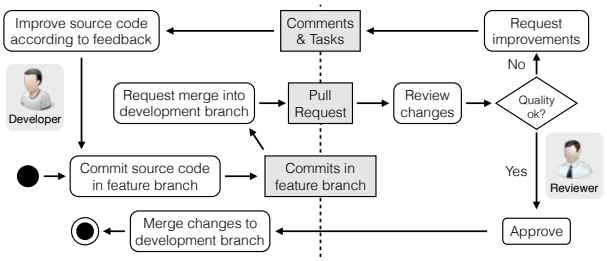

Figure 4: Overview of the Code Review Workflow

Reviewers address the following questions in the review:

- **Design Traceability**: Is the code traceable to the specified system and object design? Does it fulfill design principles such as low coupling and high cohesion?
- **Use of Patterns**: Does the code contain design or architectural patterns? Does it avoid software development antipatterns [6]? Does it avoid code smells?
- **Maintainability**: Does the code adhere to coding guidelines? Is it easy to read and understand?
- **Review History**: Does the code address feedback from previous reviews?

Pull requests show the accumulated changes of the feature branch. Changes in different commits that neutralize each other (e.g. the addition of a method that was removed later on) are not shown. Only if the code meets defined criteria, reviewers approve the request. Thus, the workflow prevents feature branches with poor code quality or bad architectural decisions from being merged with the development branch. Problems or misunderstandings found by reviewers cause a comment added directly to the changes. The developer, who had requested the merge, reads these comments and improves the code in response commits. The pull request is updated automatically. When all comments are addressed, reviewers approve the request. Compared to other collaboration models, this solution for sharing and reviewing commits creates a streamlined workflow. While git could send notification e-mails with a simple script, it becomes haphazard when developers discuss changes and have to rely on e-mail threads, in particular when response commits are involved. Pull requests put a discussion platform on top of commits and branches into a web interface next to the repository.

To improve its visibility, the status of code reviews can be tightly integrated with the issue tracker. When developers start to implement a feature, the corresponding issue transitions into the state *In Progress*. This transition is initiated automatically when the developer creates a feature branch for the issue. When he has realized the feature (i.e. resolved the issue), he opens a merge request to automatically transition to the state *In Review*. After the reviewers approved the merge request and the feature branch was merge, the issue transitions to *Closed*. Automatic transitions synchronize the state of the VCS and the issue tracker. The status of multiple pull requests can be visualized on a digital taskboard.

The branch based code review workflow has several advantages: (1) Only changes in the feature branch must be reviewed. If the change set of a feature branch is small, the workload for reviewing is small. (2) If an experienced programmer reviews the code for errors, there will be less defects in the code. [11] (3) Developers prevent "broken windows" in the development branch, if they use this workflow from the beginning: "Don't leave broken windows (bad designs, wrong decisions, or poor code) unrepaired. Fix each one as soon as it is discovered." [22] Conducting code reviews avoids that bad design and poor code are distributed to the whole development team, and is potentially being reused in other places in the system. This alleviates the risks of the broken window theory in programming. (4) The workflow increases collaboration and knowledge transfer between developers, because pull requests facilitate conversations about actual source code. This improves peer learning [4]. Inexperienced developers can learn best practices and coding guidelines while doing asynchronous pair programming over the Internet. [39] This is especially helpful for balanced teams with beginners and advanced programmers. While pull requests allow for asynchronous pair programming, developers should also build pairs for synchronous pair programming [2].

However the workflow also poses challenges. If there are too few experienced developers who have to review many pull requests, they shortly become a bottleneck for the development progress. When features are too large, developers need several days or weeks to finish them. The change set to be reviewed becomes large so that reviewing the code needs a lot of time. In addition, improving the code upon the review comments takes more time and might delay the development process. In such situations, the likelihood increases, that merge conflicts occur, in particular when parallel feature branches have overlapping changes. To alleviate such problems, features should be small. Additionally, pull requests can be integrated into a continuous integration system. When a pull request is created, the integration server detects it, checks if a merge is possible without conflicts and if the merged code builds and all tests pass.

4. WORKFLOW DETAILS

In this section we present more details about each activity of the workflow according to Figure 2. The workflow starts

when the developer creates the source branch, as shown in Figure 5. Next comes the development work for the feature. Both during the implementation and throughout the rest of the workflow, the developer should be aware of new, relevant commits in the development branch and pull them to update his branch, as shown in Figure 3. The merge may result in conflicts, which are resolved locally before the workflow is continued. Once the developer finished the work, he commits to the branch, resulting in one or more initial commits, depending on the feature size. Typically, the commits are immediately pushed to the remote repository for storage, however, this step can be delayed until the merge request.

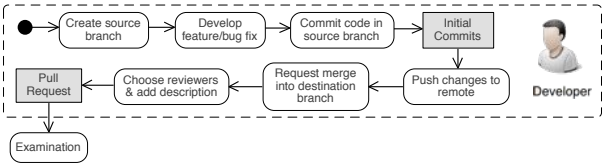

Figure 5: Workflow for the preparation activity

When the developer implemented the feature, he requests a merge into the development branch. For the pull request to be created, the developer needs to select reviewers and add a brief description of the changes and their purpose. At least one experienced reviewer that is familiar with the purpose of the changes should be included among the participants. Nevertheless, the pull request is open to the rest of the team, and any other member that would like to review the code, learn how to conduct reviews or simply is curious can join and contribute to the review as well. Once the developer has completed the above steps, the examination activity begins that is shown in Figure 6 following either preparation or rework. The first point of interest is ensuring that the code was able to build successfully and passed the test cases. If either the built or a test case failed, the developer is notified and is expected to fix the code and update the pull request before the reviewers can examine it. Usually, this step is automated by using a continuous integration server and an integrated notification mechanism such as e-mail.

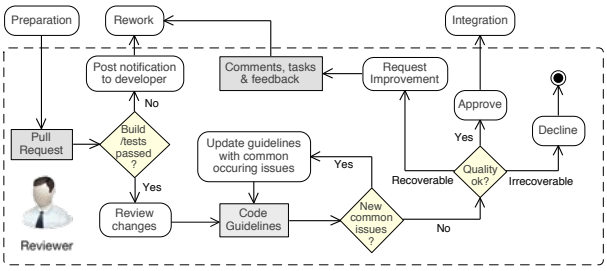

Figure 6: Workflow for the examination activity

If the build is successful and all test cases pass, the actual review takes place. The assigned reviewers are automatically notified about the pending request and can follow a link to begin reviewing. Their goal is to find anomalies by answering the questions detailed in Section 3. Since branch based reviews are asynchronous over the Internet, the reviewers conduct the examination independently of each other. When an anomaly is identified, it is documented using comments or tasks, which directly reference the code. Comments support discussion threads, whereas tasks are used to track anomalies that must be refactored. If the examination follows the preparation activity, the reviewers have to examine all changes, however, if it comes after rework, they focus on the detected anomalies. The reviewers check if their prior feedback was implemented correctly and ensure that no other problems were introduced. The code guidelines are an essential source for the reviewers. A good opportunity to continuously improve and keep them relevant, is to update the document with common anomalies found after each review.

Once a reviewer has examined the changes, he decides whether the source code fulfills the team's quality standards. In the best case, the merge request is directly approved and the workflow proceeds to integration. However, if the reviewer believes that code quality should be further improved through rework, he requests improvements by noting the anomalies in comments and tasks. The third possibility is to outright decline the pull request, either because the developer is unwilling or incapable of implementing the feedback, or because the code quality is so poor that the improvement cost outweighs the benefit of having the work. Declined pull requests, though, are very rare and should be treated as an indicator for a bigger problem concerning work allocation or lack of motivation. The more common case are recoverable pull requests, where with one or more improvement cycles of examination and follow-up, the code reaches a state where it satisfies the quality requirements for approval.

The developer is immediately notified about incoming feedback. Ideally, he first responds to comments that require clarification or discussion threads about possible or alternative solutions. Once these are resolved, he starts the rework activity shown in Figure 7. He then improves his source code according to the feedback documented in tasks and comments. This includes fixing bugs, adjusting coding style to guidelines and restructuring the design to fit to the system architecture. If antipatterns or code smells were detected, the developer must implement refactored solutions or refactor the code smells. The second case for entering rework is when the pull request has a build or test case failure. In this context, the developer needs to find and fix the problems before the reviewers have a chance to examine his changes.

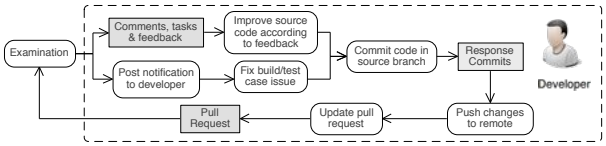

Figure 7: Workflow for the rework activity

Rework results in changes that are committed as a response commit. The commit must be pushed to the remote repository in order to automatically update the pull request with the new changes. Typically, the tool used to conduct the review is integrated with the VCS, so that the commit automatically triggers the update, which alerts the reviewers to return back to the pull request and conduct the examination activity. This examination-review cycle is repeated until the request is approved, at which point the review enters the last step in the workflow. The work that remains for the integration activity, is for the developer to merge the changes into the development branch, as shown in Figure 8.

Possible impediments that could arise from the merge attempt are merge conflicts. They are present when the source branch has diverged from the destination branch and the

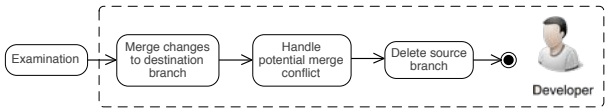

Figure 8: Workflow for the integration activity

two contain different changes in same areas. To resolve the conflict, the developer can either use tools designed for this purpose, or, in the worst case, do it manually. The actual resolution involves choosing which version of the changes to keep: the developer's changes, the ones in the destination branch or a combination thereof. If the merge conflict involves large changes to the codebase, the reviewer should examine the code again. Once all conflicts are resolved, the code can be successfully merged into the development branch. The developer can then delete the source branch to clean up the repository, which marks the end of the review workflow.

5. TEACHING APPROACH

In this section we explain three ways to teach the workflow described in Section 3 to students in our capstone course.

5.1 Introduction Course

Before the kickoff of the capstone course, all students attend an introduction course, in which we teach the programming language, common design patterns and important frameworks of the development platform. The example code of the introduction course is often reused by the students in their projects. Therefore, we check all materials against a comprehensive set of code guidelines. During the introduction course, students learn the branching model and use the coding guidelines. They develop the solution to programming exercises in feature branches in their own repository and open a pull request to submit their solution. Tutors review and approve the pull request, if the solution is correct. Otherwise, if anomalies or errors occur, they write comments to ask for changes. If the tutors finally accept the solution, the student merges the changes. With this method the students apply the code review workflow more than 10 times during the introduction course and incorporate it in an organic manner.

5.2 Cross-Project Team

The cross-project review team is led by the review coordinator, an experienced programmer who also has good communication skills, e.g. a teaching assistant. One member of each project team is part of this cross-project team and has the role of the review manager which is responsible for all activities to achieve high code quality. Their first meeting is early after the course started and the infrastructure has been setup. After becoming acquainted, the review coordinator explains goals and responsibilities of the review team. He gives a tutorial about branch management and the review workflow using an example project and shows all steps from the developer and reviewer perspectives. The review coordinator is the main contact person for the review managers. He needs technical expertise to understand the process and common errors, e.g. how to fix a merge conflict. He also needs skills to communicate with the team of review managers and to track the status of all projects.

The review coordinator also introduces the coding guidelines and asks the review managers to discuss and agree upon them with their development team. He takes care that all review managers understand and apply the workflow in their teams. The review team meets biweekly to share knowledge about tools and workflows, to synchronize their understanding and to discuss and resolve potential issues with workflows and tools. The coordinator uses different techniques to further build knowledge in the review team. He assigns small challenges to review managers such as to present best practices, code smells or antipatterns that were reviewed within their team. Another task is to describe how the development team actually uses the workflow and why they might differ from the presented one. This facilitates that review managers take responsibility for their role and internalize the knowledge required for peer learning with the rest of the team. The review coordinator regularly checks whether the review managers fulfill the mentioned tasks by talking to coaches and project leaders.

5.3 Interactive Tutorials

The capstone course includes a weekly course-wide meeting for milestone events like kickoff, design review and CAT, and is also used in between to introduce workflows and best practices, and to reflect over the current status. In these course-wide meetings, the instructor teaches software and usability engineering concepts to the students in short live tutorials, e.g. meeting management, agile methods or prototyping. The review coordinator uses one of these meetings to hold an interactive tutorial about review management and the code review workflow. The main goal of the tutorial is to create a common understanding with all students.

He explains version control, branch management and discusses important best practices such as having small commits, using meaningful commit messages or only committing if the code compiles without errors. Then he shows how to use pull requests to conduct code reviews in an asynchronous way and introduces best practices such as short branch lifetimes and how to handle non-mergeable files. He also introduces coding guidelines, typical code smells and antipattern as well as examples for refactored solutions. He intermixes theory with practical exercises where the students build pairs to try out the concepts: one student is the developer and another one is the reviewer. They apply the whole workflow as described in Section 3 and then switch the roles to apply it again from the other perspective.

6. EVALUATION

In this section we present the results of our evaluation and the observations in the 2014, 2014-15 and 2015 capstone course labeled iOS14, iOS1415 and iOS15, respectively. After the CAT, we conducted an online survey with the participants of iOS14. All students, including review managers and coaches, were invited to answer a 20-minute questionnaire. It contained 33 questions and was not mandatory. We received 81 full responses out of 90 invitations. We used anonymous tokens that allowed us to send e-mail reminders only to students who did not fill out the survey. Additionally, we conducted personal interviews with participants of all three courses. We investigated the following hypotheses:

(H1) **Review Practice**: Students understand, appreciate and apply the review workflow.

(H2) **Peer Learning**: Students learn and adapt best practices from each other, improving their code quality and coding skills.

(H3) **Handling Workflow Problems**: Students encounter and can handle typical workflow problems.

To answer H1, we counted pull requests, comments, etc., looked at the review interval and asked whether the students are convinced about the workflow. With respect to H2, we investigated what students learn from one another and if this knowledge leads to quality improvements. The workflow may pose challenges such as merge conflicts due to parallel feature development in different branches. Therefore, we asked about the students' experience with the problem and how they were able to handle it (H3). We present findings for each hypothesis and discuss threats to validity.

6.1 Review Practice

Table 1 contains the number of repositories, pull requests, comments, commits for the three courses. Additionally, we computed the interval (i.e. the amount of time in hours) for each pull request, from when it was created until it was merged. We also included the average measures per pull request as well as standard deviation and coefficient of variation to compare the results. We filtered out declined reviews and did not count merge commits. In iOS14, the teams created and approved 1053 pull requests, on average 96 pull requests per team. In iOS1415, the average per team was 74, while in iOS15 it was 97. This indicates the teams' frequent use of the review workflow. There were differences between teams, which we attribute to the following three reasons: (1) The number of pull requests depends on the partition of requirements into sprint backlog items such as features; some teams came up with a high amount of small features, while others created a small amount of large features. (2) Some teams also used hotfix branches for each bug, which then consisted of only a very small amount of changes. (3) A few teams had a larger codebase than others.

Some teams made little use of comments in pull requests because they worked together in the same room or used other channels to communicate the feedback. Other teams used comments quite often when they worked distributed. Most teams had a similar number of commits per pull request (between three and five). Only a few teams had a lot more commits per pull request since they used larger features that needed more code changes. Then, multiple features branches were open for a long time because the developers needed more time to actually implement and review the functionality of the feature. This in turn leads to longer review intervals and a higher chance for merge conflicts, although most teams managed to review and approve pull requests within less than one day on average. Only a few teams needed more time due to fewer reviewers or larger amounts of changes.

Figure 9 depicts the pull request interval throughout the courses. One observation is that pull requests appear earlier in iOS15. This is the result of introducing the workflow during the introduction course in 2015, which occurs much earlier than the weekly course-wide meetings that were used to present pull requests in the previous iterations. The early introduction and organic manner of incorporating the workflow in iOS15 enabled the students to start using pull requests sooner than their peers previously had. The graph in Figure 9 indicates that the average pull request interval is shortened in the time leading to design review and CAT. One explanation is that reviewers got used to the workflow over time and therefore conducted reviews quicker; however, the increase in review interval after the design review, when there was again more time for reviewing and less pressure to present features, shows additional factors were involved.

From the interviews we found that the students used more pair and team programming before the milestones. In such cases, they already reviewed the code before opening the pull request and therefore closed it quickly without comments. A few students reported that milestones are a difficult time to maintain the balance between reviewing and finishing features that they want to present in a demo. Some teams responded to the pressure by prioritizing implementation and therefore conducing less thorough reviews. This way, they both saved time and did not block developers working on intersecting parts of the codebase.

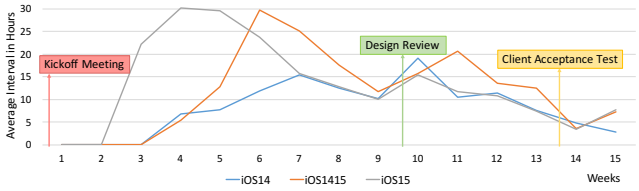

Figure 9: Average pull request interval in hours per week

Figure 10 shows the average number of comments per pull request. It follows a trend similar to the time interval. Most of the feedback was given at the early stages of the pull request introduction, when the developers were only beginning to learn about code quality and arguably made more mistakes. However, the increase in comments after the design review indicates that the lack of feedback did not only stem from the developers learning how to write better quality code. Team programming and time pressure before milestones led to less thorough reviews in the pull requests, with lower number of comments.

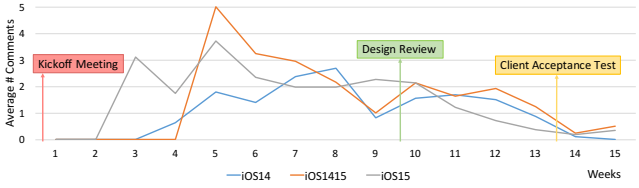

Figure 10: Average number of comments in pull requests per week

Further evidence that the students used and appreciated the workflow comes from the online survey. When asked whether the branching model helps them to map sprint backlog items to branches in version control, 85 % of the students agreed. Moreover, a majority of 98 % of the students thinks that the branching model supported development with multiple persons on the shared codebase. In conclusion, we have found supporting evidence for H1 that the review workflow was understood and valued enough to be adopted and regularly used throughout the course, notwithstanding the pressure during milestones, which affected the usage.

6.2 Peer Learning

Figure 11 shows interview findings from participants who ranked their motivation for conducting reviews and the achieved benefits they perceived. Ensuring architecture compliance and improving code readability and maintainability were the main motivators, and likewise, the most prominent benefits of branch based reviews. The interviewees agreed that reviews had a bigger impact with sharing system knowledge among the team than initially believed. Educating

Course	# Projects	# Repos	# Pull Requests	Comments				Commits				Interval		
				#	avg	sd	cv	#	avg	sd	cv	avg	sd	cv
iOS14	11	35	1053	1569	1,49	± 4,03	2,70	4402	4,18	± 6,05	1,45	11,19	± 22,49	2,01
iOS1415	11	32	819	1497	1,83	± 3,93	2,15	3472	4,24	± 5,37	1,27	16,62	± 27,87	1,68
iOS15	11	33	1067	1599	1,50	± 5,03	3,35	4105	3,85	± 5,37	1,40	12,88	± 25,36	1,97

Table 1: Comparison of measurements regarding code review workflow usage of students in three capstone courses (avg = average per pull request, sd = standard deviation, cv = coefficient of variation)

novice developers about best practices and quality standards was also a significant motivator and benefit. While the students did find defects using reviews, it was not much more than they had expected.

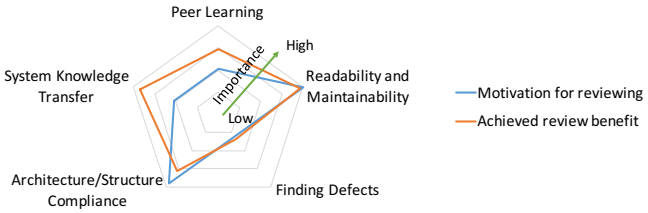

Figure 11: The students' view on motivations and benefits of reviews

Another focus in our evaluation was the students' learning experience. Figure 12 shows that more than two thirds of the students could improve the code quality of their own code with the feedback of experienced developers. The students reported about mentoring relationships between experienced and inexperienced developers where both learned something.

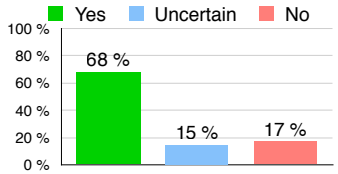

Figure 12: Comments by reviewers helped me to improve my code quality

We received the same results when we asked whether the code review workflow helped the team sustain good code quality in the development branch. As shown in Figure 13 two thirds think that the workflow had an important impact in preventing the negative effects of the broken window theory.

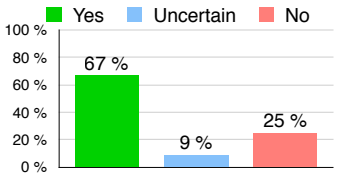

Figure 13: My team was able to maintain high quality code in the main codebase

We asked the students whether they would use the workflow again in future projects. Only one student replied that he would not do so. Figure 14 shows that 43 % of the students definitely want to use the workflow again, 27 % very likely and 19 % likely. Given the above, we conclude that students adapted programming best practices and improved their skills resulting in better quality code, which supports our peer learning hypothesis H2. Moreover, they learned about different parts of the system from reviewing their peers.

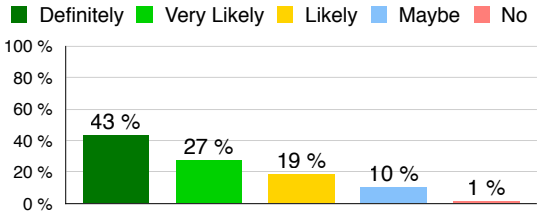

Figure 14: Would you use the workflow again in a future project?

6.3 Handling Workflow Problems

We asked whether students encountered specific problems while using the workflow, e.g. if they encountered simple or complex merge conflicts or whether the build in the continuous integration server broke. Figure 15 shows that simple merge conflicts occurred quite often; students reported that they could easily resolve them. Complex merge conflicts such as when multiple developers worked on non-mergeable files, happened less often, nonetheless, three out of four students experienced them. Some students had problems resolving complex merge conflicts and needed help from experienced team members. We also asked them how to prevent such errors and some common answers were to improve team communication, to only change non-mergeable files on the development branch and to pull changes from development into feature branches more often. One important answer was that they should try to minimize the lifetime of branches.

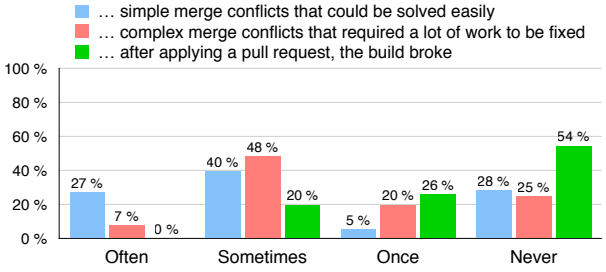

Figure 15: How often did you encounter...?

One problem that we identified in the interviews with some of the review managers, was that the code review workflow needs additional time and can slow down the team's progress. This occurs especially right before a sprint review meeting with the customer or a demo for the design review or CAT. In such situations, some teams did not thoroughly review all pull requests. However, Figure 16 shows that only 22 % of the students believe the workflow affected the team's progress. Our findings support H3: students encounter problems and can

handle them with the exception of complex merge conflicts where they need help from experienced developers.

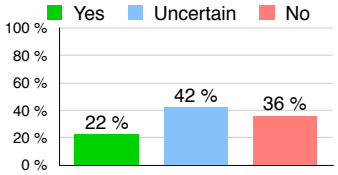

Figure 16: The workflow slowed down the progress of my team

6.4 Threats to Validity

There are threats to the validity in the methodology of our evaluation that we discuss briefly. First, we might have the problem of selection bias. While most of the course participants took part in the online questionnaire, some teams used the workflow more frequently than others because of more experienced students. To alleviate selection bias we asked students in which team they worked. We have at least five responses from each team, so the threat of selection bias is small. Additionally we observed the same results in the interviews where interviewees agreed with our findings.

A problem might be that participants gave answers which do not reflect their work practice, because they knew we would publish the results. This threat was addressed by guaranteeing anonymity. We know that our findings might not be generalizable to industry projects because of the different setup at university. However, we think our course is similar to a project based organization in industry so that most results would remain valid. The amount of code lines changed in a pull request is not shown. It does not reflect the size of the code changes as developers added frameworks, used automatic code formatting and refactored code.

A limitation is that we did not compute the number of response commits after the review. It would be interesting to evaluate how many review comments developers addressed in response commits. We were not able to measure the actual code quality and how much it improved in contrast to previous courses. We reviewed random samples and observed that it increased. Our evaluation focused on the benefits through the process aspects of the workflow and it was not our aim to measure code quality in a quantitative manner.

7. RELATED WORK

In this section, we review related work. We discuss similarities and differences to other code reviews studies and describe the use of pull requests in the open source community.

7.1 Code Review Studies

Our results regarding review motivation differ from publications such as [1] where finding defects ranks first. However, [1] and [31] come to the same conclusion that improving readability and maintainability counts for the majority of the feedback in reviews. The studies also confirm our findings that reviewing increases exposure and enables developers to learn more about the system [1, 31]. There is also agreement that reviewing enables teaching novice developers about quality and best practices [1]. Similarly, the interviews highlighted that the mere knowledge of code being reviewed and criticized leads to developers paying extra attention to quality and therefore writing better code. Developers reported a remaining need for direct communicate during reviews [1].

Developers need incentives for reviews, otherwise they do not like to spend time on reviewing code. Team members who develop many features and fix a lot of bugs are seen as heroes, reviewers do not get noticed so much. [1] Teams should be able to adapt the process to their own needs, e.g. to allow certain changes to happen on the development branch or time-critical bugfixes not to be reviewed. In contrast to formal reviews, modern code review approaches involve less formal practices [12, 32]. Informal practices enable teams to adapt code reviews to their needs and to switch to other forms such as over-the-shoulder reviews or pair programming.

7.2 Pull Requests in Open Source Community

In open source communities such as GitHub, repository owners manage incoming code contributions using pull requests. Developers without write access fork the repository and implement their contribution in their fork. If they want to merge back their contribution, they create a pull request including their changes. The owner of the source repository can review the changes and ask for improvements before accepting the change. Publications show that pull requests are an important part of the social coding community in GitHub and improve transparency, learning and collaboration in open software repositories [14].

Tsay and his colleagues present a study on open source contribution in GitHub that evaluates pull requests, which are the primary method for contributing code [36]. They analyzed the association of technical and social measures with the likelihood of acceptance. They found that repository owners use information about the technical contribution and the personal connection to the submitter when reviewing. In some cases, multiple rounds of reviewing and comments were necessary to establish a shared understanding. This is in line with Marlow who found that uncertain pull requests need negotiation and explanation [28]. Pull requests with many comments tend to signal controversy and were less likely accepted by repository owners [14]. Popular projects were more conservative in accepting pull requests because it poses a higher risk for the code users if defects get through.

8. CONCLUSION

We described a branched based workflow for code review management that we successfully introduced in a capstone course with 100 students and industry clients. The workflow fulfills its main goals: code is reviewed regularly from the beginning and students learn to peer review themselves asynchronously over the Internet. It ensures that only reviewed code which meets quality standards, including design patterns and avoiding development anti patterns and code smells, is present in the main codebase. This prevents the broken window theory in programming without slowing down the development process.

The course instructor teaches knowledge about conducting reviews using different approaches: an introduction course, a cross-project review team and interactive tutorials. The combination of these approaches ensures that knowledge about required workflows, tools and guidelines is distributed to all participants in a project-based organization. We evaluated the review practice, peer learning and how students handle workflow problems in an online questionnaire and personal interviews. Our findings indicate that students understand

workflows and tools, appreciate their benefits and that code quality increases. They thoroughly use the review workflow during the whole semester expect before important milestones where they concentrate on getting features finished for the live demo. In the future, we want to introduce additional team reviews after milestones and evaluate their benefits.

The separation of requirements into small features is important because large features would require too much time to be realized. This would lead to longer review intervals slowing down the development process and increasing the likelihood of merge conflicts. It remains future work to find the right size of features as it can be hard to break down large and complex features into smaller ones. Code review workflows in capstone courses decrease the effort of the instructor and increase the learning experience of students. In one semester, 100 students managed to conduct 1000 code reviews with 1500 feedback comments contributing to a higher code quality than in the previous courses. 89 % of the students want to use the workflows again in future projects.

9. REFERENCES

[1] A. Bacchelli and C. Bird. Expectations, outcomes, and challenges of modern code review. In *Proceedings of ICSE*, pages 712–721. IEEE, 2013.

[2] K. Beck and C. Andres. *Extreme programming explained: embrace change*. Addison-Wesley, 2004.

[3] B. Blum. *Software engineering: a holistic view*. Oxford University Press, 1992.

[4] D. Boud, R. Cohen, and J. Sampson. *Peer learning in higher education: Learning from and with each other*. Routledge, 2014.

[5] C. Brindescu et al. How Do Centralized and Distributed Version Control Systems Impact Software Changes? In *Proceedings of ICSE*, 2014.

[6] W. Brown et al. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. Wiley, 1998.

[7] B. Bruegge, S. Krusche, and L. Alperowitz. Software engineering project courses with industrial clients. *ACM Transactions on Computing Education*, 2015.

[8] B. Bruegge, S. Krusche, and M. Wagner. Teaching tornado: from communication models to releases. In *Proceedings of Educators' Symposium*. ACM, 2012.

[9] L. Chen. Continuous delivery: Huge benefits, but challenges too. *Software, IEEE*, 32(2):50–54, 2015.

[10] M. Ciolkowski, O. Laitenberger, and S. Biffl. Software reviews: The state of the practice. *IEEE Software*, 20(6):46–51, 2003.

[11] A. Cockburn and L. Williams. The costs and benefits of pair programming. *Extreme programming examined*, 2000.

[12] J. Cohen, E. Brown, B. DuRette, and S. Teleki. *Best kept secrets of peer code review*. Smart Bear, 2006.

[13] P. Crosby. *Quality is free: The art of making quality certain*. Signet, 1980.

[14] L. Dabbish et al. Social coding in github: transparency and collaboration in an open software repository. In *Proceedings of CSCW*. ACM, 2012.

[15] W. Demming. Out of the crisis: Quality productivity and competitive position, 1986.

[16] V. Driessen. A successful git branching model, 2010. Retrieved January 08, 2016 from http://nvie.com/posts/a-successful-git-branching-model.

[17] M. Fagan. Design and code inspections to reduce errors in program development. *IBM Journal of Research and Development*, 15(3):182, 1976.

[18] M. Fagan. Advances in software inspections. *IEEE Transactions on Software Engineering*, 12(7):744–751, 1986.

[19] A. Feigenbaum. *Total quality management*. Wiley, 2002.

[20] M. Fowler. *Refactoring: improving the design of existing code*. Pearson, 1999.

[21] E. Gamma et al. *Design patterns: elements of reusable object-oriented software*. Pearson, 1994.

[22] A. Hunt and D. Thomas. *The pragmatic programmer: from journeyman to master*. Addison-Wesley Professional, 2000.

[23] IEEE. Guide to Software Configuration Management. *Std 1042-1987*, 1988.

[24] IEEE. *Standard for Software Reviews and Audits (Standard 1028-2008)*, Aug 2008.

[25] S. Krusche and L. Alperowitz. Introduction of Continuous Delivery in Multi-Customer Project Courses. In *Proceedings of ICSE*. IEEE, 2014.

[26] S. Krusche et al. Rugby: An Agile Process Model based on Continuous Delivery. In *Proceedings of the 1st International Workshop on RCoSE*. ACM, 2014.

[27] A. Leon. *A Guide to software configuration management*. Artech House, Inc., 2000.

[28] J. Marlow, L. Dabbish, and J. Herbsleb. Impression formation in online peer production: activity traces and personal profiles in github. In *Proceedings of CSCW*, pages 117–128. ACM, 2013.

[29] K. Muslu et al. Transition from Centralized to Decentralized Version Control Systems: A Case Study on Reasons, Barriers, and Outcomes. In *Proceedings of ICSE*, 2014.

[30] R. Patton. *Software Testing*. Sams, 2005.

[31] P. Rigby and C. Bird. Convergent contemporary software peer review practices. In *Proceedings of the Joint Meeting on FSE*, pages 202–212. ACM, 2013.

[32] P. Rigby et al. Contemporary peer review in action: Lessons from open source development. *IEEE*, 29(6):56–61, 2012.

[33] K. Schwaber and M. Beedle. *Agile software development with Scrum*. Prentice Hall, 2002.

[34] A. Spillner, T. Linz, and H. Schaefer. *Software testing foundations: a study guide for the certified tester exam*. Rocky Nook, 2014.

[35] A. Stevenson. *Oxford Dictionary of English*. OUP Oxford, 2010.

[36] J. Tsay et al. Influence of social and technical factors for evaluating contribution in github. In *Proceedings of ICSE*, pages 356–366. ACM, 2014.

[37] C. Walrad and D. Strom. The Importance of Branching Models in SCM. *Computing Practices*, 2002.

[38] G. Weinberg and D. Freedman. Reviews, walkthroughs, and inspections. *IEEE Transactions on Software Engineering*, SE-10(1):68–72, 1984.

[39] L. Williams and R. Kessler. *Pair programming illuminated*. Addison-Wesley, 2002.

[40] E. Yourdon. *Structured walkthroughs*. Prentice Hall, 1979.