Effective Peer Reviews: Role in Quality

Anil Chakravarthy (Anil_Chakravarthy@mcafee.com)

Sudeep Das (Sudeep_Das@mcafee.com)

Nasiruddin S (nasiruddin_sirajuddin@mcafee.com)

Abstract

The utility of reviews, specifically of artifacts related to software development cannot be overemphasized. However, adopting a consistent and fruitful review process is difficult. There are many challenges to streamlining a review process to be widely accepted and followed in a software development team.

In this paper, we look at the learning gleaned from trying to adopt a consistent, constructive and effective review process across multiple teams. We look at the pitfalls, travails, shortcomings and mistakes while adopting a review process across various stages of the development process. We also highlight things that actually help a review process to be a "baked in" part of development process rather than being a "bolted on". This paper is primarily based on the authors' first hand experiences on introducing, sustaining and benefitting from peer reviews in software development.

The paper presents a holistic view of review rather than focusing on just code reviews. Review of other artifacts like designs, plans, and schedules shall also be discussed. It will be useful to managers and development team members in leadership roles. The paper provides insight from practical and firsthand experience about one of the important aspects of a software development process.

Biography

Anil Chakravarthy is a Senior Technical Lead at McAfee with more than seven years of software development experience. Highly passionate about quality, he strives to look for ways and methods to improve software reliability and usability. As an inventor of key technology solutions, his interests include security management, content distribution and updating.

Sudeep Das is a Software Architect at McAfee, with more than ten years of experience designing and implementing software. He started with a three person team and over time has seen it to grow to more than twenty, while improving upon development and testing processes incrementally over the years. His areas of interest span security management, content updating mechanisms, data protection and virtualization.

Nasir is a Principal Engineer at McAfee, with more than eight years of software development experience. He is highly passionate about working on emerging and innovative technologies.

Introduction

Peer review is one of the most effective ways of ensuring quality. Not just in software, but also in research and academic publications. Software that doesn't use reviews has defects discovered at later stages of development and potentially post shipping. It has also been established that late detected defects are expensive to fix. Yet, software engineering teams struggle to adopt a consistent and effective peer review process. For teams that do adopt a review process, at times, they fail to find enough early defects to justify the investment in a review process. Given the vast and diverse literature on peer reviews, software engineering teams often struggle to adopt a review process that works best for them.

In this paper, we start with presenting the goals of a peer review. We discuss effectiveness of peer reviews. Finally, we discuss factors that contribute to effectiveness of reviews and factors that hamper the effectiveness of peer reviews

Our aim is to share the lessons we learned while trying to adopt a peer review process in multiple teams in our organization. Over the years, we have tried different review processes and methods and refined our processes. We continued the things that worked, and eliminated parts that didn't work. This paper is the outcome of our experience and learning on peer reviews, and is based entirely on our experience

We believe that leaders of software engineering teams would find this paper most useful. However, we also believe that our experiences will be beneficial to other software engineering roles as well.

Structure

While many software engineering teams equate peer review to code review, the fact is that peer reviews apply to all artifacts of software engineering process. Artifacts like requirements, specifications, estimates, design, test plans and documents are all candidates for review. A peer review for each of these artifacts can help with early detection and removal of defects.

Not restricting peer reviews to code review brings up a few fundamental questions. What should be reviewed? By Who? Why? When? How?

We start with trying to answer the "What". What software engineering artifacts should be peer reviewed? Having identified the review targets, we then try to answer the "Why" of that particular review target. It establishes the goal of the peer review for that particular artifact. Finally, we look at the "Who, "When" and "How" of the peer review for each of the items

The What:

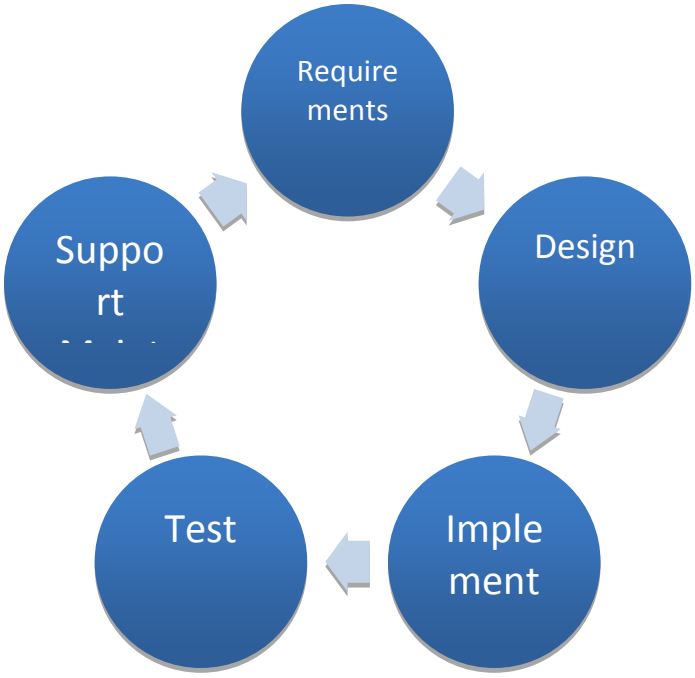
Software engineering encompasses multiple stages and functions, namely, Requirements, Design, Implementation, Testing and Support. While the stages may overlap in time, the primary output of each stage is distinct. It is the output of each of these stages that should be peer reviewed

So, the artifacts that are subjects of peer reviews are

1. Concept Commit
2. Requirements
3. High level feature design and architecture
4. Low level feature design
5. Feature Implementation
6. Review of static analysis findings
7. Regular code review
8. Test plans
9. Test cases

The Why:

Each of the artifacts generated above have a different purpose, and the peer review goals of each of them are different. The key goals of reviewing each of the artifacts are as below.



Concept review goals:

Leaders of a software engineering group formulate a business concept based on a multitude of factors such as market direction, business strategy, and technology advances. While there are a multitude of factors to be considered, the primary purposes of a concept review are to ensure that

1. The concept aligns with the larger strategic imperatives of the business
2. The concept does not adversely impact other successful offerings

3. The concept fills an existing gap in the market
4. The concept is sized for a reasonable time to market

Requirements review goals:

Once business leaders commit to a concept, it is broken down by the concept champion into deliverable features in a way that fulfills the vision behind the concept. The features or epics are then further broken down into user stories to achieve a high degree of granularity. It is at this stage that a thorough review with engineering leadership is warranted to realize the following important goals

1. Ensure technical feasibility of the features
2. Ensure engineering team capacity
3. Prevent feature creep
4. Prevent scope creep
5. Ballpark sizing and alignment with time to market needs
6. Enhance clarity and minimize ambiguity of features
7. The engineering cost of developing the solution.

High level design review goals:

A high level design serves as a straw man, and provides a technical view of the solution being developed. This phase needs in depth review, as the technical choices made during this phase have a very significant bearing on downstream quality. The goals of this review are

1. Ensure that the technology choice can meet scale and performance needs
2. Ensure that the design accounts for security aspects, like authorization, authorization, audit, regulatory compliance
3. Ensure that the design can be implemented on all desired platforms
4. Ensure that economics of technology choices are acceptable. Some technology choices, while easier to work with, force the end user to license building block technologies.
5. Discover if the end user is more familiar with one technology over another that may be used in the solution
6. Ensure that the design covers all use cases agreed to in the requirements review
7. Ensure modularity and extensibility of components

In our experience, we have found that design processes are iterative in nature and multiple revisions are a norm. A design that is subjected to multiple iterations is not symptomatic of a deficient design, but is rather indicative of a fuller and better understanding of the requirements

Low level design review goals:

A low level design is highly implementation centric and is the blueprint of the software being developed. Low-level designs are a magnified view of high-level designs and have a lot in common with the high level design. The key goals of a design review are

1. Ensure alignment with high level design
2. Ensure appropriate choice of algorithms
3. Ensure modularity
4. Ensure extensibility
5. Ensure conformance to established and standard design patterns
6. Improve the accuracy of estimates

Implementation review goals:

Implementation - better known as code. Literature abounds with stated, perceived and purported goals of code review. Goals of code review vary widely and depend on the specific domain and application of the code being developed. Code for a banking application will go through significantly more significant review criteria than, for example, a desktop game application.

However, there are goals that are common across the board…namely

1. Readable code
2. Maintainable code
3. Documented and well-commented code
4. Adherence to design
5. Logical correctness
6. Error handling

Test plan review goals:

A test plan details the high level steps that need to be taken to ensure low defect software.

Engineering team members rely on the test plan to determine if the software has achieved its stated quality goal. The goals of reviewing a test plan are to ensure that

1. Quality goals are reasonable and achievable
2. All relevant quality indicators are spelled out.
3. All testing phases accounted for… unit test plan, integration test plan, acceptance test plan
4. All testing domains accounted for, for example, security testing, performance testing
5. All features and use cases are covered
6. Automation has been accounted for

Test case review goals:

Test cases lay down the steps that verify that a feature satisfies its stated requirements and quality goals. It is important for test case reviewers to ensure that

1. Test cases are accurate
2. Test cases are exhaustive
3. Test cases are complete
4. Test cases are comprehensive

The How:

The generally prevalent peer review methodology is a formal in-person review where the owner of the work presents his/her work, and multiple reviewers engage in finding shortcomings in the work. The shortcomings are then debated, argued, and some kind of resolution to each shortcoming is expected. Reviewers across multiple functions are involved, ostensibly, to keep all stakeholders aware of important decisions. This approach, in our experience, is found to be severely limiting, and is highly ineffective for the following reasons:

1. Reviewers may have opposing views on issues, and an in-person review with 5 participants turns to a technical duel of 2 seasoned professionals that continues for hours.
2. In-person reviews are found to be intimidating by junior team members, particularly in the presence of reporting managers.
3. Managers are judgmental about individual performance based on code review feedback.

4. For distributed teams, synchronizing time for multiple team members over multiple time zones is a challenge.

Our recommendations for peer review are as follows:

Pre Review :

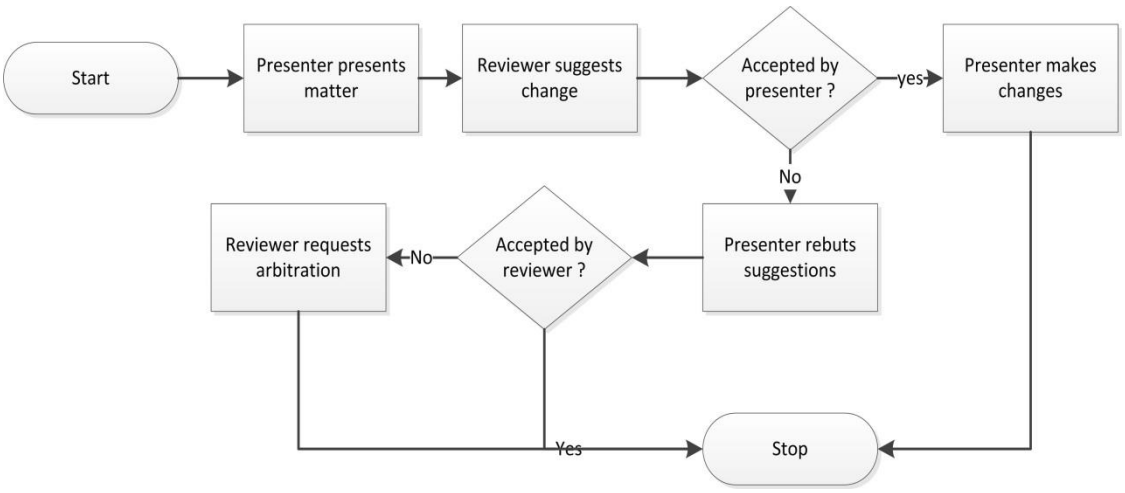
1. Train the reviewers. Reviewers should be trained on what to review, and be sensitive to the goals and purpose of the review
2. Designate an arbitrator for each phase and artifact. Someone in the team should have the final word in face of disagreements.
3. Utilize automated tools where possible. For example, for code reviews, readability, and maintainability and to some extent, error handling may be enforced and detected using static code analysis tools.
4. Avoid reviewer fatigue. Content to be reviewed should be small. If there is a lot of matter to be reviewed then it should be broken down and each part should be reviewed separately.
5. Avoid multi-level reviews. Some teams have an initial peer review that is then followed up by a “superior review” of sorts, where a “senior” and “expert” person reviews the already reviewed content.

In review:

1. Avoid in-person reviews for code, test cases and design. It is useful to use collaborative review tools such as code collaborator, that lets presenters annotate and upload their work for review
2. It is good to have cross-functional reviewers, primarily with the purpose of identifying gaps in the subsequent development phases. For example, in a code review, developers may find the code suitable for production, but the code may not be testing and automation friendly. Review by testing team members would highlight this shortcoming.
3. Two is the optimal number of reviewers. The review benefits hit a point of diminishing returns with more than two reviewers. We have experienced that there is significant duplication in the issues discovered by additional reviewers.
4. Observers should only observe. If there are stakeholders who need to be kept informed, their role is that of observer who can only watch but cannot comment.

Post review:

1. Timely closure: Having review pending, and going ahead to the next step is a mistake which that leads to rework.
2. Avoid a rally. Unresolved disagreements should be arbitrated rather than being dragged on.



The Who:

The effectiveness of peer reviews has a strong dependency on the number and expertise of the reviewers. Some teams follow a pre-defined matrix of reviewers; some teams gate reviewers on experience and expertise. The key aspects to be considered while deciding on reviewers are

1. In general, reviewers should belong to the same functional group that produces the artifact, and observers should belong to immediately succeeding functional group that utilizes the artifact.
2. The concept review should have participation from product management, possibly early adoption users, and engineering leaders.
3. High level design reviews should have participation from architects and senior developers.
4. Low level design review should include developers as reviewers and senior testers as observers.
5. Test plan review should include senior developers and senior testers.
6. Test case review should include testers and developers.
7. Arbitrators should be senior and respected people in the same functional group that generates the artifact.

Teams generally have mixed composition of experience and expertise, and it's important to ensure that skills are organically developed within the team. The concept of observer provides a unique opportunity for achieving this, where a team member would learn to be more effective watching other team members perform the role

It is found to be detrimental when reviewers from a different functional group are actively involved in reviewing artifacts of a different functional group. A disproportionate of time is spent explaining and clarifying commonplace and obvious facts; context is not obvious to the reviewers. More reviewers do not lead to better reviews.

The When:

Review content can generally be plan-related content or action-related. Code, test cases, and designs may be classified as action-related content, whereas concept, test plans, high level architecture are plan-related content.

For action-relation content, the notion of “Review early, review often” is found to work the best. This is because review presenters, at times, create a large volume of work to make sure the work is complete before they present it for review. This deluges reviewers with too much content to be reviewed in a short time. It is useful to constitute review of the days’ work at the end of the day, to be reviewed first thing the next day.

For plan-related content, where completeness is a key aspect of review, it is more useful to have high bandwidth, engaged, potentially in person reviews rather than off line and passive review. Reviews should be completed before moving on the next task. Pending and unfinished reviews are a major source of rework and effort duplication. For example, if a high-level architecture review is unfinished and the next phase of development is already in progress, a change to architecture later would result in rework and effort duplication.

Concept, high-level design, low-level design, test plans and test cases should be reviewed once complete. Feedback should be incorporated and the artifacts reviewed before moving on the next phase. Subsequently, any modifications to these artifacts should also be similarly reviewed.

Conclusion

This paper presents our experiences on the “Who”, the “How” and the “When” aspects of different review contents that we have found effective for our teams over a significant period of time while using agile methodologies. Reviews are one of the key contributors to software quality. Reviews need to be effective to useful. Reviews can be made highly effective by following a few simple and easy to adopt guidelines. Different review artifacts may utilize different review strategies, like passive but arbitrated reviews and in person but open debate reviews. Adopting reviews initially present challenges because of its inherent nature of being a fault finding exercise, and teams should be eased into it over time. From a management perspective, it works better if reviews are a pulled process, as compared to being a pushed process. Teams need to be reasonable and guarded in their expectations from a review, as reviews do not guarantee zero defect software. Reviews alone however, do not ensure software quality. Complementary processes like continuous integration and automated verification tests are important and have their place in the software development process. Metrics are important.