Code Inspections: An In Depth Review of Techniques

Daniel A. Carton

Department of Computer Science & Engineering,
University of South Carolina

cartond@email.sc.edu

May 2, 2016

ABSTRACT

Code inspections -- synonymously referenced as code reviews -- are systematic techniques to analyze code quality and have strong support for their fundamental assistance of discovering faults, transferring information, and promoting company norms. This paper seeks to answer questions (1) what are code inspections and how are they implemented, (2) can every company benefit from code inspections, and (3) how can the programming community improve the current code review process. Since its formation in the late 70's, various segments of the original code inspection process has been reviewed, scrutinized, and reworked by many researchers and practitioners. Sections of the process such as team meetings are surveyed to determine cost and effectiveness. I found that code inspections are implemented in a number of ways amongst different companies. In addition, every research paper surveyed did not question the effectiveness towards quality assurance that code inspections produce. Companies also implement a wide array of techniques to ensure rigorous code reviews such as coverage and paired programming techniques. Lastly, the software engineering community has improved techniques by creating tools to automate the inspection process and help inspectors better organize concerns and changes.

{ }

1 INTRODUCTION

Code inspections -- synonymously referenced as code reviews -- are systematic techniques to analyze code quality that have strong support for their fundamental assistance of discovering faults, transferring information, and promoting company norms. There are a variety of techniques and variety of processes to apply to software code to improve the quality of the resulting product. The main goal of any code review is ultimately to improve the quality of the software produced. In the process, junior developers can consume new knowledge such as company norms and new approaches to problems. After being implemented in companies for over 40 years, code review continues to develop cleaner, more consistent, and more reliable code.

2 USAGE

The undisputed value that code inspection improves quality of software is evident to all studies that were surveyed in this paper. Code inspections are most helpful when more automated analyses are not applicable to verify a property. An example would be checking whether a programmer used camelCase consistently

through their application. Code reviews collaborative approach to reviewing is applicable to any kind of software, because it relies on humans to carry out the task.

2.1 What is Addressed

Addressing issues like efficiency, scalability, and repetition methods, inspections encourage developers to think with these goals in mind [**18.3 Pezzè]**. From personal experience, I believe this is best addressed as the last step of a code segment. The first two being a working algorithm, faultless implementation, and lastly efficiency. Furthermore code inspections can be implemented earlier than testing can be. This in turn can create ease for the development team with more tests passing. In addition, unit tests can't always be applied to artifacts while inspections can.

3 METHODS

Whether the inspection process is classic or modern, it consists of three main phases: preparatory, review, and follow-up **[18.3 Pezzè]**. Differences in the implementation of can

3.1 Process

The first step of the inspection process, or the preparatory phase, an inspector will make certain that the segments to be reviewed are all satisfactory. After ensuring acceptable artifacts, the inspector will (1) assign inspection roles, (2) acquire the information needed for the inspections and inspectors, (3) plan individual activities of the inspector roles, and (4) schedule the inspection meeting **[18.3 Pezzè]**. As discussed in section 3.2 and section 8, meetings are one of the big debates of research in code inspection techniques, because of the time it requires to schedule a meeting amongst reviewers, and time wasted in the actual meeting.

Succeeding the preparatory phase is the cardinal and most important step, the review phase. Classic inspection techniques are based on checklists and communication from said checklists **[18.4 Pezzè]**. The basic outline of a checklist item would range from checking class name capitalization to design decisions. From checking if comments properly describe imported classes to the file footer's revision log. The reviewer can check either yes or no, and leave comments describing what they liked, didn't like, or any concerns. These systematic and consistent processes have proved to set a foundation of structure for inspection teams **[18.4 Pezzè]**. A similar approach for review is to focus on user-flow or user-stories, as it can discover new issues with the software.

Lastly, the third juncture of code review is the follow-up phase. This stage of the process is similar to getting English teacher's notes back from a rough draft. The code producer -- and sometimes testers-- receive a summary of concerns from the review team. They subsequently solve each concern by either adding, removing, or editing the segments. Follow-up yields situational results on a pre-review basis. For example, if there are logical concerns, the review team might ask to re-review this developer's code. This is expensive as it starts the full three step process over. In contrast, it might be something simple like an uninitialized variable or missing segment of code. These would not require consideration of re-review **[18.3 Pezzè].**

In Fagan's popular 1976 publication discussing design and code inspection he includes a more in depth five-step process; this is 32 years before Pezze describes the condensed three step process:

Process Operations	Objectives of the operation
1. Overview	Communication: the author presents an overview of the scope and purpose of the work product.
2. Preparation	Education: reviewers analyze the work product with the goal of understanding it thoroughly.
3. Inspection	Find Errors: the inspection team assembles and the reader paraphrases their work for the producer. Reviewers raise issues that are subsequently recorded by the scribe.
4. Rework	Rework: the author revises the and resolves errors to the product found by inspection
5. Follow-up	The moderator verifies the quality of rework and that all errors, problems, and concerns have been resolved and decides if reinspection is required.

Figure 1: Fagan's five step inspection process. (3 Johnson, 11 Fagan)

3.2 Modern Approach

The modern approach to code review is not as tight or formal and utilizes more tools after over 40 years of practice. The use of meetings is far less emphasized. In the 1998 paper *Does Every Inspection Really Need a Meeting* , Johnson states that research concludes that meetings are a "costly component" shown to add 15-20% overhead onto the development process. He continues to conclude that meetings distract and retracts employees from their current work (2 Johnson). By using automated software, inspectors can omit meetings and leisurely address concerns in a comment-like setting that would highlight the line or lines of code in question. In addition it is thought to be more "lightweight" than techniques used in early days of inspection occurring in the 70s and 80s (1 Bird, 1 McIntosh). Most large companies -- Microsoft, Google, and Facebook to name a few -- designate some level of code review and there are many ways to implement review, various tools, and many life-cycle phases to perform inspection (1 Bacchelli).

For example, Google believes that its developers are best suited with a high priority of code review. Although a lenient philosophy to allow developers to edit anything in the massive stack of Google's code repository, the inspection schema is not so relaxed. Every programmer's change to code must undergo a series of web-based tests and reviews with automatically generated test results such as "simulating tens of thousands of users after just minutes of prep work" (Vallone). Likewise, it is interesting that Google also requires that test code go through a set of test cases and the same code review process.

4 TECHNIQUES

Team members that are chosen for selection must have a balance of perspectives, background, and cost [**18.2 Pezzè]**.

Similarly -- and applicable to any field of profession -- if one feels that there is an evaluation of their performance being surveyed, the motivation to follow guidelines more specifically can skew results of their average performance.

4.1 Group Inspections

4.1.1 The Classic Group Inspection

The classic approach for group inspection, dating back to the late seventies, focused on code artifacts with four to six inspectors [**18.2 Pezzè].** Inspectors will be selected in a way that balances perspectives, background knowledge, and cost. Cost is important because a junior developer compared against a senior developer will likely have found different results and process through the review differently -- at a different cost. Again cost is a problem when the consideration of a larger inspection team is suggested to receive different levels of expertise and varying perspectives. Furthermore, classic approaches did not span all the approaches of a modern review. Efficiency was questioned during this period and eventually summoned an array of research on the process.

4.1.2 The Review Meeting

Another process, known as The Review Meeting, is a formal technical review process that demands more preparation from reviewers. The four roles of this method are producer, review leader, reviewers, and recorder (234, Pressman). The producer develops the code and contacts the review leader establishing they are ready for review. The leader then prepares materials to distribute to the review team. Reviewers, usually two to three developers, are given appropriate materials. After spending no more than a couple hours on the product the meeting is held. A recorder will log any issues brought forth from the review team during this meeting. The big difference between these two group inspections is that the Review Meeting method allows the producer to interact with the panel. This means the code manufacturer might try to sway the panel in one direction, maybe not in the best intentions. A Microsoft team manager stated “discipline of explaining your code to your peers that drives a higher standard of coding. I think the process is even more important than the result” (5 Bacchelli).

In conclusion, the classic group inspection allows less tampering of panel judgement. Ultimately the developer can only weep in hope for sympathy on their code review. This might motivate engineers to perform better, but would discourage and scare others. The Review Meeting process is more unharming as an activity. Although intentions of developers are studied later, for now the panel must trust the producer. It is more conservative and a safer approach that does not result in belittlement to the engineer being reviewed.

4.2 Pair Programming

Pair Programming, part of the Extreme Programming (XP) agenda, is associated with agile processes utilizes two programmers side-by-side on one workstation **[18.5 Pezzè].** Much like going through training for a driver’s license, one person will steer while the other consults on actions being taken. It is evident to see that if one of the developers is a junior or intern, there will be a large amount of knowledge transferred and company norms inherited -- a topic studied further in Section 4.4.

Furthermore, because Pair Programming is part of the Extreme Programming itinerary it follows that it takes a more modern approach to review. Instead of checklists, the review process is a dynamic entity always being applied to segments. The chances that both developers do not adhere to a known practice, norm, or coding principal is lowered, and thus surmises healthier code.

During quantitative research on Pair Programming, the results are very promising. A substantial 90% of professional programming that work in pairs said they enjoy their jobs more, than when they worked alone (8, Williams). Furthermore, inconsiderate project managers that cast emotions aside will be satisfied to know that paired programming also yields better, faster results after the adjustment time of operations (4, McDowell, 4, Williams). This adjustment time, or change from unaccompanied to sociable programming, has also been researched. It showed a decrease of 60% more programmer-hours than lone developers, to 15% programmer-hours after their first paired assignment.

	Individuals	Collaborative Teams
Program 1	73.4%	86.4%
Program 2	78.1%	88.6%
Program 3	70.4%	87.1%
Program 4	78.1%	94.4%

Figure 2: Percentage of Test Cases Passed from page 6, Williams, showing the percentage of unit-tests passed from a timed programming exercise. "Collaborative Teams", or paired-programmers show evidence of more sound programs than developers working alone.

4.3 Productive Learning

A third approach to code inspection techniques is to teach during the inspection process. For example, pair programming assigning a senior engineer with a junior engineer. In this avenue of approach the senior developer would keep authority and the junior developer would participate in the discussion **[18.2 Pezzè]**. The goal of this is to see the reason behind actions that come from someone holding more precedence in the software, the senior developer in this case, for future problems like integration of other fragments of the big-picture. Research based on over one thousand Microsoft support my hypothesis suggesting that this junior developer, and all inspectors in this case, will learn from the inspection alone or with a senior inspector and increase their social awareness, a knowledge transfer, and improved problem solving skills (5 Bacchelli). Along with this conclusion, the vast array of developers were surveyed on what motivations they had for code review. Interestingly enough, the data implies that knowledge transfer was more important to reviewers than improving the development process and avoiding build breaks (figure 3).

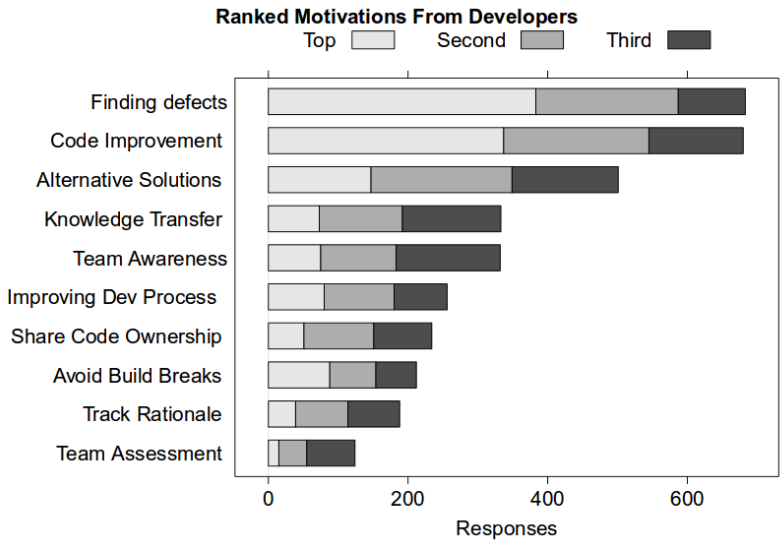

Figure 3: Developers' motivations for code review from page 5 of Bacchelli's "Expectations, outcomes, and challenges of modern code review"

5 DISADVANTAGES

According to Pezze and Young, authors of Software Testing and Analysis, "inspection is not a full time job" **[18.2 Pezzè]**. They further claim that a lot of research has shown that productivity drastically decreases after two hours of work. Similarly a study focusing more on the economics of software engineering has shown that there are large divergences between accuracy and productivity when looking at a single individual (1 Da Cunha). A personal hypothesis to this is that, after time, the inspectors have caught up to the most recent segments of code. Unless the inspector is at a company with hundreds of developers, he will have no new code to review.

5.1 Expense

Machiavelli, attributed founder to modern political science, one said "Doctors say some infections at their beginning are easy to cure, but difficult to recognize. But, when they are not first recognized and treated, become easy to recognize but difficult to cure." In other words that relate to software testing, the later in the lifecycle a disease, or bug, is discovered results in the more opportunity for the disease, or bug, to affect other segments of its source. Both cases end in a more costly resolution to fix.

Although inspections will yield a healthier end product in theory, they will take programmers, project managers, and program testers away from their current tasks (2 Johnson). This of course promotes the use of a devoted inspector, but as we discussed in section 4's introduction the efficiency of a full time inspector is not supported by research. In fact multiple studies suggest the opposite **[18.2 Pezzè]**. Furthermore, code inspections are not incremental, or does not allow reinspection of code. In other words the reinspection of code is "nearly as expensive as inspection of the original artifact" **[18.3 Pezzè].** The basis for this argument is that inspecting source code too early will have to undergo inspection again if it changes any main functionality, something developers cannot consistently predict.

Moreover by testing too late, faults might have already been resolved by developers requiring proper functionality. For example, think of a web-based application where many libraries are included throughout

the application stack. A developer with incorrect data being displayed on the front end will be forced to debug HTML, various JavaScript functions, ajax server calls, and the backend of the server. All because something is not working properly. It is easy to argue that the correct implementation of inspection throughout the stack at an earlier date would have avoided his troubles and efforts debugging other components of code.

In conclusion, implementation of code inspection in respect to project cycle is a problem of intermittent, or irregular, results. Research concludes that no matter when inspection is implemented, it will result in a lower cost of removing and correcting faults later in development **[1 & 11 Barnard, 18.1 Pezzè, 8 Boehm, 231 Pressman].** Some research indicates that as inspection efficiency increases, cost generally decreases **[8 Barnard]**. Moreover, it is evident that assigning a low-level employee, like an intern, can achieve training while also checking over source code. Discovering company norms and new approaches to problem solving, the company will save money in training and inspection, although the inspection might not yield astonishing results.

5.2 Fault Bounty

A reward mechanism might cause malicious behavior, like poor developers attitudes, from fault density or fault discovery compensation proposals. Fault density is the idea that if a bug sneaks through the process of inspection, it will be of higher value if found afterwards. A selfish engineer -- like the junior engineer learning from the senior engineer previously mentioned -- might not engage in inspection discussions. Subsequently on a later date the junior reveals the bug for praise and a reward. This progresses into the senior developer receiving hassle.

In addition, fault discovery's method of rewarding fault exposure will unintentionally identify inspectors that review high quality, sound code [**18.2 Pezzè]**. Although this might come off as an approach to some, it is clearly a disadvantage because of the unjust repercussions and malicious intent generated.

5.3 Self-Inspection

Any developer that has written a program can explain how familiar it is to start looking and inspecting at *your* own code. This is supported by Pressman's research where he states "although people are good at catching some of their own errors, large classes of errors escape the originator *more easily than they escape anyone else*" (230 Pressman). You don't need to read the comments, you don't need to look at the outside function contexts, and you don't require an explanation of why something was written a certain way. The long hours, complex logic, and optimization a programmer commits to effort can consciously affect their evaluation of their self, because they know the work deposited. So a project manager cannot simply ask developers to review themselves. Similarly, self-evaluation can result in company norms not being enforced. Pressman suggests that any reviews using a diverse group of people results in more uniform code through the company, making it more manageable later in the software lifetime (230, Pressman). We can conclude from his research that the idea of using a third party developer will find errors with more ease and is suggested for the program review process.

6 MOTIVATION

Efficiency and cost are two burdens that all companies face. These are two main motivations behind a code review. Project managers and senior developers alike can feel at ease with confidence of a process that proves both qualities. In professions like sales, for example, efficiency of an employee can be measured by their sales reports. Fault detection is not as easily scaled.

One motivation to conduct research on code inspection is to discover a metric of efficiency of code inspection. A technique is to compare inspection with unit, integration, and system tests. It is apparent that a metric for unit testing efficiency is the cardinality of the tests per fault found. For manual inspection, a metric is not so obvious. Barnard proposes using the amount of effort per fault detected as shown in *figure 4*; one of his **nine** metrics **[10].** *How do you measure human effort?* Clearly factors like their experience and familiarity effect their metric, although some studies. This solution-caused-problem, which Barnard solves by using his equation, ultimately proves to be less efficient than all other methods of fault detection [**10 Barnard]**.

Average effort per fault detected: The average number of hours spent in inspection activities by the inspection team for a single detected fault:
$average\ effort\ per\ fault\ detected = \frac{\sum_{i=1}^{N} \boldsymbol{Inspection\ effort\ i}}{\sum_{i=1}^{N} \boldsymbol{Total\ faults\ detected\ i}}$ *Where N is the total number of inspections.*
As with the average effort per thousand lines of code, this effort computation includes only time spent by the inspection team preparing for the meetings, holding the meetings, and correcting the detected faults. The computation includes faults of all severities.

Figure 4: How to compute the metric Average Effort per Fault Detected. Taken from page 3, Barnard

7 RESEARCH

The first formal measures to ensure quality assurance were established at Bell Labs in 1916, and continue to this day (228, Pressman). Now, quality assurance is measured through many techniques such as code review.

7.1 Automation

As previously discussed on group inspections in section 4, and disadvantages in section 5, there is a large amount of effort spent on scheduling, organizing, mediating, and conducting a group meeting. Research on inspection automation, automating the process not the actual review, has promising progress in delegation. In 2006, Python creator Guido van Rossum, delivered his first project at Google, a code review system named Mondrian (Kennedy). With all of the code of Google living in a repository available to anyone within the company, it was now exposed to their own internal tool to handle code review. Before Mondrian, Google handled code review with a tool that started email threads. These word documents containing concerns of code understandably can get out of control, lost, and updates of documentation could be overlooked. Mondrian displayed a straightforward dashboard depicting code changes for the user to review, and the user's changes awaiting to be reviewed. In addition, the tool told user's what their code review accomplished such as: negative, positive, or neutral. By allowing team members to collaboratively work on reviews on their own schedule,

The efficiency of Mondrian created a desire from the programming world, but Google's internal tool was not open source, like many of their other projects. This spawned the creation of one tool, Diffy, integrating

with source control tool Git. Members of a repository can easily highlight and comment on lines of code. Subsequently, the author can comment back with agreeance, or accept the request to modify the code.

7.2 Personality Correlation

Of course in an interview for any software engineering job the interviewees are assessed, at least a little, on their personalities. What if personalities had more impact than just "water-cooler" conversations?[1] Two researchers in the Centre for Software Reliability at the University of Newcastle upon Tyne have starting looking at why "some professionals show skill in debugging while others are less successful" with developers' personality types in particular (2 Da Cunha). This study used sixty-four second-year undergraduate students inspecting two hundred and eighty two lines of Java code. Along with their inspection results looking for sixteen faults, the students were rated on a scale of Sensing-Intuition (SN) vs Thinking-Feeling (TF) of the Myers-Briggs Type Indicator for computing personality (2 Da cunha). The results of the test suggests that NT, or intuition and thinking, personalities were best at finding faults. Intuition and feeling type personalities came second. This suggests that developers that put intuition first, then rationalize, should be deployed to code inspection teams. Another researcher implies that personalities cause consequences on producers stating "people and egos" get involved in reviewing the product the producer, instead of just the product (235, Pressman).

A company can deduce that inspection roles must capitalize on the strengths of their employees' performance and personality. It does not make sense to assign a poor back-end developer to the back-end developer position. Similarly, product managers should not ignore any tense or immature inner-office relationships when assigning inspection roles.

7.3 Inspector Characteristics

During the review and interview of inspectors of five major open source software (OSS) organizations, research has been conducted on the types of inspectors that review code. This study found two main categories: reviewer and outsider (6, Rigby). These two classifications were expanded upon by interviewees to add characteristics to learn about the interaction between them and developers. These characteristics led to positive and negative personas of reviewers.

The three positive reviewer personas include objective analyzer, expert or fatherly adviser, and enthusiastic support or champion. An objective analyzer is utilized for in depth analysis of the goals that inspection attempt to accomplish. As discussed previously in the paper, code inspection greatly values the "expert or fatherly adviser." Junior developers can learn greatly from these personas and utilize the idea of transferred knowledge such as that OSS community's culture or historic inspection issues similar to the problem being faced. The champion is a dedicated, enthusiastic, and passionate reviewer that takes responsibility where no other team members approach.

The two negative personas include the grumpy cynic and the frustrated and resigned. A grumpy cynic is typically depicted as a senior developer. For example consider a junior developer approaching a problem in a way that has already proved to fail. A grumpy cynic, being around the project for a long period of time, might lose temper that the new developer has not researched previously attempted results. Similarly short tempered is the frustatedly resigned, because of the length of particular reviews. The researchers depict this personality by simply ignoring an issue after they become uninterested with the discussion. Something

1. Some research has already looking into this, but this is not my main focus of this paper. Here are some more publications found: Donaldson, *Successful Software Development*; Kidder, *The Soul of a New Machine;* Weinberg, *The Psychology of Computer Programming*; Furnham, *Do personality factors predict job satisfaction?* Shneiderman, *Software Psychology: Human Factors in Computer and Information Systems*;

interesting that is pointed out is that “silence implies consent” in relation to forum discussions in the open source community (6 Rigby).

Outsiders to the OSS development team were said to have positive and negative aspects resulting in debate amongst developers to whether they are needed or not. The helpful outsiders were said to give real world examples, clean bug reports, test system application program interface (API), and initial reviews. Most of these tasks can be performed without code experience, and leads to believe that the cost of hiring such a team would be low. Negative aspects of the outsiders consist of rude requests for fixes, infeasible objectives, and overly specific use cases.

Regardless of the two categories, characteristics amongst reviewers still yielded helpful results to the inspection process. Furthermore, both categories desired the more relaxed process of inspection that OSS preferred.

7.4 Code Review Coverage

Something I studied extensively for this paper is code coverage as a review metric to delineate quantitative results as opposed to the extravagant focus on qualitative research. One researcher stated “In reality, some medium risk code might be high risk with respect to certain types of failures, and I would tailor the type of testing (and the amount and rigor of code inspections) to the blend of risks“ (9, Marick). Although few of the papers surveyed included insight with respect to code inspections, I think it is something that could be utilized in this field as a metric to support its effectiveness of healthy code.

One such study found notable correlation between code review coverage in particular that more thoroughly reviewed code and higher participation from inspectors produced a “significant link with [positive] software quality” (9 McIntosh). Similarly, the three-way collaboration research in this paper concludes that modern approaches to inspection provide positive outcomes on software standards. Amongst the three software projects, Qt, VTK, and ITK, the study deduces that on average, projects contained up to five more defects when inspection had poor participation from inspectors (2 McIntosh).

7.5 Open Source Software

Open source software is software that anyone can use, edit, distribute, and contribute to. Focusing just on the contributing part, open source software encourages collaborative development. This means anyone can edit the code and submit their changes for consideration. With this lies problems like malicious intent, but we are interested in the productive changes to code. Like any big company, an open source software organization would need to regulate things such as, testing and code review. How well can you hold code inspection standards when your employee base can be potentially millions of internet users?

In over 400 individual reports of code inspections reviews for five high profile open source software projects such as: Apache, the Linux kernel, and FreeBSD, a research team in Canada examined reports to discover trends in how developers communicate during the review process (Rigby). In addition they sought explanations on why patches are ignored, how stakeholders interact with review requests, and what happens when there are too many opinions. In all five projects, the software review process consists of email based communication. These emails could potentially contain pages of conversation, debate, explanations, and code differences. Researchers proceeded to summarize these reports’ different abstract themes.

Patches, or bug fixes, that need to be addressed from inspection are not assigned to any single developer. Instead the reviewer gets to choose what he/she decides to patch. One contributor interviewed states that they review a specific segment “because of interest or experience in the subject area of the patch” (3, Rigby).

This supports the idea of using inspectors with relatable experience. As discussed previously, this can cause problems with cost when more than a few inspectors are necessary to review the work.

Another question seeking wisdom for the Canadian researching duo was "why are patches ignored?" and "how much does it impact the project?" Large companies, such as Google, Microsoft, and similar must set deadlines for their project inspection meetings to avoid wasting cost, but open source projects do not. Without the pressure of traditional methodologies these asynchronous programming projects allow for more relaxed inspection processes. A conclusion drawn was that although open source projects relax their guidelines on timelines, the quality of review is not compromised (5, Rigby). Furthermore, it suggests that setting a timeline goal might cause poor inspection results to accompany for interval objectives.

In conclusion of the open source research, investigators discovered that although scalability of code inspection was hypothesized to cause problems, it did not. Additionally, open source software continues to produce "mature and successful products" (10, Rigby)

8 FUTURE WORK

Future work and scrutiny of code inspection includes topics like: languages, ownership, scheduling, and meeting effectiveness. This paper discusses a little on how an inspector can be subconsciously ignoring code review on their own code. Analysis should be produced on this subject, because of the possibility to compromise and wound software quality. Similarly I desire more examination of comparing inspection results of independent program languages. For example, Python's clean and easy syntax compared to Java's distinct set of rules.

8.1 Scheduling

Nearing the end of this paper, the question: "what stage should my team start worrying about inspections" has not been answered. In layman's terms, *we don't know*. IBM argues to start late in the project. Their logic summarized is that the project will be mature, the test cases will have less errors to correct and thus less faults throughout the code source [**Fagan]**. Pezze and Young argue the opposite claiming "it must be placed to reveal faults as early as possible, but late enough to avoid excessive repetition" [**18.3 Pezzè]**. This is supported by section 4.1 of this document, exploring how repetitious inspections lead to a higher cost. In addition, Barnard of AT&T suggest applying inspection as early as possible **[8 Barnard]**. He continues to say the sooner it is implemented early to development cycle, the more successful the fault finding outcomes will be. Constraints like team knowledge, project difficulty, and metrics to review are the main factors when conducting research on scheduling. I believe that by running an experiment on a junior level computer science class, with similarly knowledgeable teams, could produce significant results proving the effectiveness of test case and inspection scheduling.

8.2 Meeting Effectiveness

Traditional approaches include steps for group meetings in the inspection process. As the internet has grown and produced communication tools like Skype and development tools such as GitHub, a collaborative code repository, there is an increase in remotely working developers. This brings a problem in physical meetings, and if meetings in general are useful. "Perhaps the most fundamental procedural constant of Fagan inspection and its many variants is the review meeting" (1 Johnson). Johnson and Tjahjono of the University of Hawaii seek to rest the controversies around traditional review practices that require group meetings. Their experiment contained twenty-four 3 person groups composed of student programmers.

Conclusions of this experiment include insight of cost, preferences, accuracy, and effectiveness. Interestingly enough, groups that held inspection meetings showed no significant difference in cost, but did require more total effort and effort per defect. Individuals generated over 4 times more false positives than the groups, on average (17, Johnson). As far as the effectiveness of performance on defect discovery, traditional group based meetings did not gain support from the research. Instead, the research workers support the proposal to improve non-meeting-based reviews.

9 CONCLUSIONS

In conclusion of this paper, it is evident that code inspections result in more reliable and mature software. Regardless of cost, effort, or efficiency of how inspection is implemented, this is true. With research in pair programming giving very positive results for code development in regards to the quality of code, it should be implemented in schools and internship programs alike. In addition, review meetings should be reevaluated on a per-company basis. Some organizations do not benefit from them. Code review is useful for discovering faults, transferring information, and promoting company culture.

Works Cited

1. Barnard, Jack, and Art Price. "Managing code inspection information." *Software, IEEE* 11.2 (1994): 59-69.
2. Bacchelli, Alberto, and Christian Bird. "Expectations, outcomes, and challenges of modern code review." *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 2013.
3. Boehm, Barry W. "Software Engineering Economics." Software Engineering Economics (2011): n. pag. 28 June 1983. Web. 28 Apr. 2016.
4. Da Cunha, Alessandra Devito, and David Greathead. "Does personality matter?: an analysis of code-review ability." Communications of the ACM 50.5 (2007): 109-112.
5. Fagan, Michael E. "Design and code inspections to reduce errors in program development." IBM Systems Journal. Springer Berlin Heidelberg, 1978. 182-211.
6. Gerrit Code Review "Code Review for Git" https://www.gerritcodereview.com/index.md Web. 02 May 2016.
7. Johnson, Philip M., and Danu Tjahjono. "Does every inspection really need a meeting?." Empirical Software Engineering 3.1 (1998): 9-35.
8. Johnson, Philip M. "Reengineering inspection." Communications of the ACM 41.2 (1998): 49-52.
9. Kennedy, Niall. "Google Mondrian: Web-based Code Review and Storage." Niall Kennedy Google Mondrian Webbased Code Review and Storage Comments. N.p., 30 Nov. 2006. Web. 02 May 2016.
10. Marick, Brian. "How to misuse code coverage." Proceedings of the 16th International Conference on Testing Computer Software. 1999.
11. McDowell, Charlie, et al. "The effects of pair-programming on performance in an introductory programming course." *ACM SIGCSE Bulletin*. Vol. 34. No. 1. ACM, 2002.
12. McIntosh, Shane, et al. "The impact of code review coverage and code review participation on software quality: A case study of the qt, vtk, and itk projects." Proceedings of the 11th Working Conference on Mining Software Repositories. ACM, 2014.
13. Pezzè, Mauro, and Michal Young. Software Testing and Analysis: Process, Principles, and Techniques. Hoboken, NJ: Wiley, 2008. Print.
14. Rigby, Peter C., and Margaret-Anne Storey. "Understanding broadcast based peer review on open source software projects." *Proceedings of the 33rd International Conference on Software Engineering*. ACM, 2011.
15. Vallone, Anthony. "The Google Test and Development Environment - Pt. 3: Code, Build, and Test." Google Testing Blog. Google, 21 Jan. 2014. Web. 27 Apr. 2016.
16. Williams, Laurie, et al. "Strengthening the case for pair programming." *IEEE software* 17.4 (2000): 19.