Open Source Peer Review – Lessons and Recommendations for Closed Source

Peter C. Rigby, Software Engineering Group, University of Victoria, Canada
Brendan Cleary, CHISEL Group, University of Victoria, Canada
Frederic Painchaud, DRDC Valcartier, Department of National Defence, Canada
Margaret-Anne Storey, CHISEL Group, University of Victoria, Canada
Daniel M. German, Software Engineering Group, University of Victoria, Canada

> *Use asynchronous, frequent, incremental peer reviews conducted by invested experts supported by lightweight tools and non-intrusive metrics*

Although the effectiveness of software inspection (formal peer review) has a long and mostly supportive history, the thought of reviewing a large, unfamiliar software artifact over a period of weeks is something dreaded by both the author and the reviewers. The dislike of this cumbersome process is natural, but neither the formality nor the aversion are fundamental characteristics of peer review. The core idea of review is simply to get an expert peer to examine your work for problems that you are blind to. The actual process is much less important for finding defects than the expertise of the people involved [1].

While developers will acknowledge the value of inspection, most avoid it and adoption of traditional inspection practices remains relatively low [2], [3]. Surprisingly, peer review is a prevalent practice in Open Source Software (OSS) projects, which have minimalist processes and consist of intrinsically motivated developers. We examined over 100K peer reviews in case studies on the Apache httpd server, Subversion, Linux, FreeBSD, KDE, and Gnome and noted an "efficient fit" between the needs of OSS developers and the evolution and minimalist structure of the peer review processes they use [4].

On these OSS projects, changes are asynchronously broadcast (usually on a mailing list) to the development team, and reviewers self-select changes that they are invested in and competent to review. Changes that are uninteresting often remain unreviewed. To find changes in what can be an overwhelming broadcast of information, OSS developers rely on simple email filters, descriptive email subjects, and detailed change logs. These change logs represent the "the heart beat of the project." Through them, developers maintain a conceptual understanding of the whole system and participate in the threaded email discussions and reviews for which they have the required expertise.

This natural fit contrasts sharply with an enforced, but often misapplied inspection "best practice" that can give a false assurance of quality, frustrate developers, and lengthen the development cycle. As Michael Fagan, the father of formal inspection laments, "even 30 years after its [Fagan inspection's] creation, it is often not well understood and more often, poorly executed" [2]. In this article, we contrast OSS peer review with the following traditional inspection process that is widely acknowledged in the inspection literature. Inspections are performed on large, completed artifacts at specific checkpoints. The inspectors are often unfamiliar with the artifact under inspection, so an individual preparation stage precedes the synchronous, co-located meeting. During the meeting, defects are recorded but not discussed or fixed. After the meeting, the author is tasked with fixing the defects.

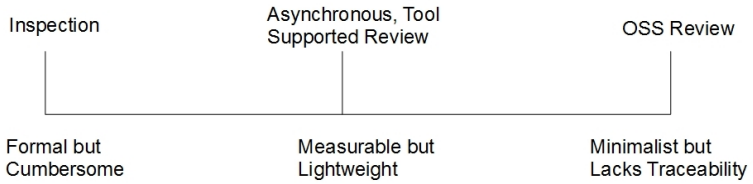

Figure 1. The spectrum of peer review techniques: From formal inspection to minimal-process OSS review. Tool supported, lightweight review provides a flexible, but traceable middle ground.

While there are obvious differences between OSS and proprietary development (*e.g.* intrinsic versus extrinsic motivation), our goal is to present the practices that are transferable to proprietary projects. We also discuss adaptations that make OSS review practices more traceable and appropriate for other organizations, while keeping them lightweight and non-intrusive for developers. The first five practices are presented as lessons from OSS. The last three practices are presented as recommendations that blend lightweight review practices (that are starting to see adoption in industry *e.g.* review at processor manufacturer AMD [5] and Cisco Systems [6]) with OSS practices.

Lesson 1: Asynchronous reviews – Asynchronous reviews allow for team discussion of solutions to defects and find the same number of defects as co-located meetings in less time. They also allow authors and passively-"listening" developers to learn from the discussion.

While managers tend to believe that new defects and other benefits will arise out of co-located, synchronous group meetings, in 1993, Lawrence Votta [7] found that almost all defects could be discovered during the individual preparation phase, where a reviewer prepares for an inspection meeting by thoroughly studying the portion of code that will be reviewed. Not only were few additional defects found during co-located meetings, but scheduling these meetings accounted for 20% of the inspection interval, adding hidden length to the development cycle. Subsequent studies have replicated this finding in both industrial and research settings. This result lead to tools and practices that involved developers interacting in an asynchronous, distributed manner. Further, the hard time constraints imposed by co-located meetings, the rigid goal of finding defects, and the measuring of success based on the number of defects found per source line lead to a mentality of "Raise issues, don't resolve them" [3]. This mentality has the effect of limiting the ability of the group to collectively problem solve and mentor developers.

By conducting reviews in an asynchronous manner and eliminating the rigid constraints of inspection, in OSS, we see the synergy between author, reviewers, and other stakeholders as they discuss the best solution, not the existence of defects. The distinction between author and reviewer can blur; there are many instances where a reviewer re-writes the code, and the author now learns from and becomes a reviewer of the new code.

Lesson 2: Frequent review – The earlier a defect is found the better. OSS developers conduct "continuous" asynchronous reviews, which can be seen as a form of asynchronous pair programming.

The longer a defect remains in an artifact, the more embedded it will become and the more it will cost to fix. This rationale is at the core of the 35-year-old Fagan inspection technique [2]. However, the term "frequent" in traditional inspection processes means that large, completed artifacts are inspected at specific checkpoints which can be many months apart. The calendar time to inspect these completed artifacts is on the order of weeks.

In contrast, most OSS peer reviews begin within hours of completing a change and the full review discussion usually takes one to two days. Indeed, the feedback cycle is so fast, we consider it a form of "continuous" asynchronous review, which often has more similarities with pair programming than with inspection [8].

As an illustration, the following quotation is taken from Rob Hartill of the Apache project:

> `I think the people doing the bulk of the committing appear very aware of what the others are committing. I've seen enough cases of hard to spot typos being pointed out within hours of a commit.`

Lesson 3: Incremental review – Reviews should be of changes that are small, independent, and complete.

The development of large software artifacts by individuals or relatively isolated groups of developers means that the artifacts are unfamiliar to the reviewers tasked with inspecting them. As Parnas and Weiss first noted, the resulting inspections are done poorly, by unhappy, unfocused, overwhelmed inspectors [9].

To facilitate early, frequent feedback, changes on OSS projects tend to be smaller than proprietary projects [10] and range from 11 to 32 lines in the median case [4]. The small size allows reviewers to focus on the entire change, and the incrementality reduces the reviewers' preparation time and lets them maintain an overall picture of how the change fits into the system.

Equally important is the requirement that each change must be logically and functionally independent. A divide-and-conquer style of review is the result. For example, a change that combines refactoring a method with a bug fix in the refactored method will not be reviewed until it is broken into two changes. These independent changes could be submitted as a group of conceptually-related changes or combined on a single "topic" or feature branch. While an individual may have all the required expertise to perform the review, it is equally possible that one person will have the required system-wide expertise to understand the refactoring and another will have the detailed knowledge of a particular algorithm that contains the bug fix. The intelligent splitting of changes allows stakeholders with different areas of expertise to independently review aspects of a larger change, which reduces communication and other bottlenecks.

Finally, changes must be complete. While discussing each step of a solution in a small group can be very effective, it can be also be tiring and there are times when individual focus can be more effective. Pair programming can force two people to be involved in each step of a solution. In contrast, frequent asynchronous review requires a reviewer to examine only the author's best attempt. However, group problem solving can still occur during the review.

Lesson 4: Invested experienced reviewers – Reviews should be conducted by invested experts, ideally co-developers, who already understand the context in which the change is being made.

Without intimate knowledge of the module or subsystem under review, reviewers cannot be reasonably expected to understand a large, complex artifact they have never seen before. While checklists and reading techniques may force inspectors to focus during an inspection [9], these techniques cannot turn a novice or incompetent inspector into an expert.

Both authors and reviewers on OSS projects tend to have at least one to two years of experience with the project, and many reviewers have more than four years, with a few having been with the project since its inception [4]. We also found that maintainers of a particular section of code provided detailed reviews when another developer made a change. The maintainer would often have to interact with, maintain, or evolve the changed code. Since co-developers are dependent on each other, they have a vested interest in ensuring that changes are of high quality. Furthermore, co-developers are already experts in the part of the system under review, so their preparation time is reduced to the time taken to understand how the small change affects the system.

While co-developers have the highest level of investment, many organizations cannot afford to have more than one developer working on the same part of a software system. A simple alternative is to assign regular reviewers to particular subsystems. While the reviewers are not responsible for making changes, they follow and review changes incrementally. This technique also spreads the knowledge across the development team, mitigating the "getting-hit-by-the-bus" risk.

In a small start-up organization, the cost of doing any review can be prohibitive. One of the authors, Brendan, used the cost-effective strategy of "reviewer as bug fixer" in his company. Here, developers are periodically assigned a bug to fix in another developer's code. Through this process, the bug fixer becomes a co-developer as he or she reads, questions, understands, and reviews the code that is related to the bug. This technique combines turnover risk management (*i.e.* developers gain a wider understanding of the system), peer review, and development work (*i.e.* bug fixing). In Table 1, we use the literature and our research findings to provide a comparison of five different types of reviewers.

Lesson 5: Empower expert reviewers – Allow experts to self-select changes they are interested in and competent to review. Assign reviews that nobody selects.

Poorly implemented, prescriptive, heavyweight processes can give the illusion of following a "best practice", while realizing none of the advertised benefits. Just as checklists cannot turn novices into experts, a formal process cannot make up for a lack of expertise. Adam Porter and colleagues reported that the most important predictor of the number of defects found during review is the expertise of the reviewers involved; the process has a very minimal impact [1].

In a development environment where reviews are assigned, it can be difficult to know who should be performing a review and how many reviewers should be involved. The author or manager must balance the candidates' expertise with their workloads and other factors. In the inspection literature, there is a rule-of-thumb that two reviewers find an optimal number of defects. The cost of adding more reviewers is not justified by the number of additional defects found [11]. In OSS, there is a median of two reviewers per review. These reviewers are not assigned, instead broadcasting and self-selection lead to natural load balancing among the development team. Furthermore, dictating a constant number of reviewers for each change ignores the fact that some changes are simple and can be "rubber stamped" by one reviewer, while others can be very complex and may require a discussion with the whole development team. The advantage of self-selection is that it is up to the developers with the most intimate knowledge of the system to decide on the level of review given to each change.

One difficulty with self-selection is that changes can be ignored. Tools can be used to automatically assign unreviewed changes to reviewers; however, unselected changes might indicate areas of the codebase that are only understood by a single developer or some other equally troubling problem.

Recommendation 1: Lightweight review tools – Tools can increase traceability for managers and help integrate reviews with the existing development environment.

OSS developers rely on the broadcast of development information and use minimalistic tool support. For example, on the Linux Kernel Mailing List (LKML), there is a median of 343 messages per day, and the OSS developers we interviewed received thousands of messages per day [12]. While there are techniques to manage this barrage of email, it is difficult to track the review process for reporting and quality assurance, and it is easy to inadvertently ignore reviews. Furthermore, the frequency of these small changes can lead to fragmentation, which makes it difficult to find and review a feature that consists of multiple changes.

Introducing tool support can help structure reviews and integrate them with other development systems. Tools typically provide:

- side-by-side highlighted changes to files (*i.e.* diffs),

Table 1. Types of reviewers based on the cost, the level of investment the reviewer has in the code, the quality of the review, and the amount of knowledge transfer and community development that occurs during the review.

Reviewer Type	Cost	Investment	Quality	Team Building
Independent Reviewer	very high	low	med	low
Pair Programming	very high	very high	high	high
Reviewer is a Co-Developer	high	high	high	high
Regular, Incremental Reviewer	med	med	med	med
Reviewer as Bug Fixer	low	med	low	med

- inline discussion threads that are linked to a line or file,
- the ability to hide and show additional lines of context and to view the diff in the context of the whole file,
- the ability to update the code under review with the latest revision in the version control system,
- a central place to collect all artifacts and discussions relating to a review,
- a dashboard that shows pending reviews and that alerts authors and reviewers who have not responded,
- integration with email and development tools,
- notification and assignment of reviews to individuals and groups of developers, and
- collection of metrics to gauge review efficiency and effectiveness.

Table 2 compares popular peer review tools.

Recommendation 2: Unintrusive metrics – Mine the information trail left by asynchronous reviews to extract lightweight metrics that do not disrupt developer workflow.

Metric collection is an integral part of controlling, understanding, and directing a software project. However, metric collection can disrupt developers' workflows and get in the way of their primary task of producing software. For example, formally recording a defect is a cognitively expensive task. Developers discussing a change will be sidetracked when they have to stop and formally agree on and record a defect. Tool support does not fix this problem. At AMD, Julian Ratcliffe found that despite the simple mechanism for reporting defects in CodeCollaborator, defects were under reported. "A closer look [by Julian] at the review archive shows that reviewers were mostly engaged in discussion, using the comment threads to fix issues instead of logging defects" [5].

Is the defect or the discussion more important? In the Linux community, the amount of discussion on a particular change is an indicator of code quality. Indeed, code can be rejected, not because it is incorrect, but simply because not enough people have tried it out and discussed it on the mailing list. The risk of accepting it does not outweigh the benefit that it brings to the system.

Turning "the amount of discussion during a review" into a metric is trivial if discussions are recorded and associated with file changes. Before a release, a manager might ask developers, "do you think that we have adequately discussed this section of the system?" In our work, we have demonstrated that many acceptable proxy metrics can be non-intrusively extracted from review archives [13].

Recommendation 3: Implementing a review process – Large, formal organizations might benefit from more frequent reviews and more overlap of developer work to produce invested reviewers. However, this style of review will likely be more amenable to Agile organizations.

OSS has much in common with Agile development and the Agile manifesto [14], [15]. For example, both prefer working software over documentation, both empower individuals rather than having a rigid process, both respond to change by working in small increments rather than rigidly following a plan, and both work closely with the customer rather than negotiating contracts.

While there are certainly differences between the two methodologies, the most striking is that Agile is used by small, co-located teams of developers, while OSS projects can scale to large, distributed teams that rarely, if ever, meet in a co-located setting. All communication, including discussion, code changes, and reviews, is broadcast to the entire community. The broadcast mentality is so strong that when a company pays co-located developers to work on an OSS project, the developers are required to summarize and broadcast all in-person discussion to the community.

The introduction of this practice in most software development companies would likely not be welcomed by software developers who are accustomed to communicating in-person. However, since peer review has been shown to be more effective in an asynchronous environment than in a synchronous, co-located one, it could benefit from this style of communication. Instead of using pair programming, developers could use asynchronous "continuous" review, where each small, functionally independent change is discussed by invested co-developers shortly after it is completed.

Table 2. Popular peer review tools

Tool	Main Advantages	Main Disadvantage
CodeCollaborator	Instant message style discussion of lines of code, metric reporting, and tight integration with multiple development environments (*e.g.* Eclipse and Visual Studio)	Commercial license fee
Crucible	Integration with the JIRA bug tracker and other products made by Atlassian	Commercial license fee
ReviewBoard	Free, full-featured web interface for review	Must setup and maintain it on an in-house server
Rietveld	Runs on top of Google App Engine, so it is quick and easy to start reviewing. Gerrit is a git specific fork of Rietveld	Project must be publicly hosted on Google Code or the review system must be setup on an in-house server
CodeStriker	Web interface that supports traditional inspection	An older tool that lacks good support for lightweight review techniques

Conclusion

Developers and researchers alike agree that code review is an important and effective method for improving code quality and sharing knowledge. However, a lot of software development companies struggle with implementing effective code review using traditional formal code review processes.

OSS code review practices have evolved as an efficient fit to the problem of maintaining code quality within a distributed group of developers and demonstrate that a robust code review practice (with its associated benefits in code quality) can work as part of an otherwise agile development method.

For software companies struggling to implement formal code review, the OSS practice of having invested co-developers perform asynchronous, frequent and incremental review might be complemented by lightweight tools and metrics to allow software teams to improve code quality while reducing the pain traditionally associated with more heavyweight practices.

Biographies

Peter C. Rigby is a postdoctural researcher working with Dr. Robillard at McGill University in Montreal. His PhD at the University of Victoria examined the peer review practices of OSS projects. He is currently examining lightweight review in proprietary projects as well as software framework documentation practices. pcr@cs.mcgill.ca

Brendan Cleary is a research fellow at the University of Victoria; he has experience managing commercial and research projects and is founder of a university spin out company. Brendan holds a PhD in computer science. bcleary@uvic.ca

Frederic Painchaud is a defence scientist at Defence Research and Development Canada - Valcartier since 2002. His current research interests include software architectural risk analysis, static and dynamic code analysis and lightweight peer review. He is currently pursuing a part-time PhD degree in Computer Science working on hybrid static and dynamic Java code analysis for automatic verification. Frederic.Painchaud@drdc-rddc.gc.ca

Margaret-Anne Storey is a Professor of Computer Science and a Canada Research Chair in Human Computer Interaction for Software Engineering at the University of Victoria. Her research interest is to understand how technology can help people explore, understand and share complex information and knowledge. mstorey@uvic.ca

Daniel M. German is an Associate Professor of Computer Science at the University of Victoria, Canada. His research areas are Open Source Software Engineering and the impact of copyright in software development. He has authored over 80 papers. Further info at turingmachine.org/. dmg@uvic.ca

References

[1] A. Porter, H. Siy, A. Mockus, and L. Votta, "Understanding the sources of variation in software inspections," *ACM Transactions Software Engineering Methodology*, vol. 7, no. 1, pp. 41–79, 1998.

[2] M. Fagan, "A history of software inspections," *Software pioneers: contributions to software engineering, Springer-Verlag, Inc.*, pp. 562–573, 2002.

[3] P. M. Johnson, "Reengineering inspection," *ACM Communications*, vol. 41, no. 2, pp. 49–52, 1998.

[4] P. C. Rigby, "Understanding Open Source Software Peer Review: Review Processes, Parameters and Statistical Models, and Underlying Behaviours and Mechanisms," thechiselgroup.org/rigby-dissertation.pdf, Dissertation, 2011.

[5] J. Ratcliffe, "Moving software quality upstream: The positive impact of lightweight peer code review," in *Pacific NW Software Quality Conference*, 2009.

[6] J. Cohen, *Best Kept Secrets of Peer Code Review*. Smart Bear Inc., 2006.

[7] L. G. Votta, "Does every inspection need a meeting?" *SIGSOFT Softw. Eng. Notes*, vol. 18, no. 5, pp. 107–114, 1993.

[8] L. Williams, "Integrating pair programming into a software development process," in *14th Conference on Software Engineering Education and Training*. IEEE, 2001, pp. 27–36.

[9] D. L. Parnas and D. M. Weiss, "Active design reviews: principles and practices," in *ICSE: Proceedings of the 8th international conference on Software engineering*. IEEE Computer Society Press, 1985, pp. 132–136.

[10] A. Mockus, R. T. Fielding, and J. Herbsleb, "Two case studies of open source software development: Apache and Mozilla," *ACM Transactions on Software Engineering and Methodology*, vol. 11, no. 3, pp. 1–38, 2002.

[11] C. Sauer, D. R. Jeffery, L. Land, and P. Yetton, "The Effectiveness of Software Development Technical Reviews: A Behaviorally Motivated Program of Research," *IEEE Transactions Software Engineering*, vol. 26, no. 1, pp. 1–14, 2000.

[12] P. C. Rigby and M.-A. Storey, "Understanding broadcast based peer review on open source software projects," in *Proceeding of the 33rd international conference on Software engineering*, ser. ICSE '11. New York, NY, USA: ACM, 2011, pp. 541–550.

[13] P. C. Rigby, D. M. German, and M.-A. Storey, "Open source software peer review practices: A case study of the apache server," in *ICSE: Proceedings of the 30th international conference on Software Engineering*, 2008, pp. 541–550.

[14] K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries *et al.*, "The Agile Manifesto," 2001.

[15] S. Koch, "Agile principles and open source software development: A theoretical and empirical discussion," in *Extreme Programming and Agile Processes in Software Engineering*, ser. Lecture Notes in Computer Science, J. Eckstein and H. Baumeister, Eds. Springer Berlin / Heidelberg, 2004, vol. 3092, pp. 85–93.