Contemporary Peer Review in Action:
Lessons from Open Source Development

Peter C. Rigby, Concordia University, Montreal, Canada

Brendan Cleary, University of Victoria, Canada

Frederic Painchaud, Department of National Defence, Canada

Margaret-Anne Storey and Daniel M. German, University of Victoria, Canada

// Open source development uses a rigorous but agile review process that software companies can adapt and supplement as needed by popular tools for lightweight collaboration and nonintrusive quality assurance metrics. //

SOFTWARE INSPECTION IS a form of formal peer review that has long been recognized as a software engineering "best practice." However, the prospect of reviewing a large, unfamiliar software artifact over a period of weeks is almost universally dreaded by both its authors and reviewers. So, even though developers acknowledge the value of formal peer review, many also avoid it, and the adoption rates for traditional inspection practices are relatively low.[1,2]

On the other hand, peer review is a prevalent practice on successful open source software (OSS) projects. We examined more than 100,000 peer reviews in OSS case studies of the Apache httpd server, Subversion, Linux, FreeBSD, KDE, and Gnome and found an efficient fit between OSS developers' needs and the minimalist structures of their peer review processes.[3] Specifically, the projects broadcast changes asynchronously to the development team—usually on a mailing list—and reviewers self-select changes they're interested in and competent to review. Changes failing to capture a reviewer's interest remain unreviewed. Developers manage what can be an overwhelming broadcast of information by relying on simple email filters, descriptive email subjects, and detailed change logs. The change logs represent the OSS project's heart beat, through which 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.

The OSS process evolved naturally to fit the development team and contrasts with enforced inspections based on best practices that are easily misapplied and end in false quality assurances, frustrated developers, and longer development cycles. As Michael Fagan, the father of formal inspection, lamented about the process he developed, "Even 30 years after its creation, it is often not well understood and more often, poorly executed."[1]

In this article, we contrast OSS peer review with a traditional inspection process that's widely acknowledged in the literature—namely, inspections performed on large, completed software artifacts at specific checkpoints. The inspectors are often unfamiliar with the artifact under inspection, so they must prepare individually before the formal review by thoroughly studying the portion of code to be reviewed. Defects are recorded subse-

0740-7459/12/$31.00 © 2012 IEEE

quently at the formal review meeting, but the task of fixing a recorded defect falls to the author after the meeting.

Some intrinsic differences between open source and proprietary development projects, such as self-selected versus assigned participation, suggest inspection processes at opposite ends of a continuum (see Figure 1). However, neither formality nor aversion is fundamental to peer review. The core idea is simply to get an expert to examine your work to find problems you can't see. Success in identifying defects depends less on the process than on the expertise of the people involved.[4]

We present five lessons from OSS projects that we think are transferable to proprietary projects. We also present three recommendations for adapting these practices to make them more traceable and appropriate for proprietary organizations, while still keeping them lightweight and nonintrusive for developers.

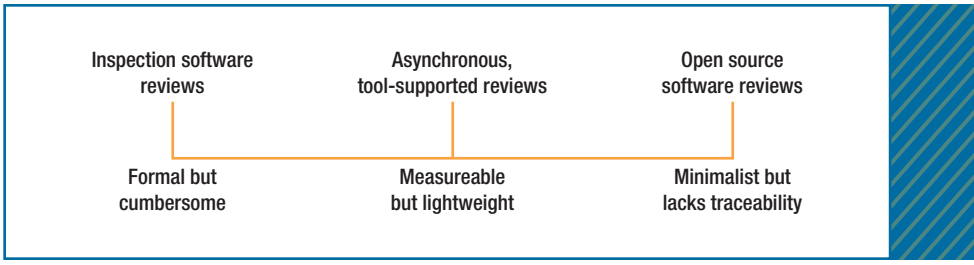


FIGURE 1. The spectrum of peer review techniques, from formal inspection to minimal-process OSS review. Tool-supported, lightweight review provides a flexible but traceable middle ground.

Lesson 1: Asynchronous Reviews

Asynchronous reviews support team discussions of defect solutions and find the same number of defects as colocated meetings in less time. They also enable developers and passive listeners to learn from the discussion.

Managers tend to believe that defect detection and other project benefits will arise from colocated, synchronous meetings. However, in 1993, Lawrence Votta found that reviewers could discover almost all defects during their individual preparations for an inspection meetings, when they study the portion of code to be reviewed.[5] Not only did the meetings generate few additional defects, but the scheduling for them accounted for 20 percent of the inspection interval, lengthening the development cycle.

Subsequent studies have replicated this finding in both industrial and research settings. This led to tools and practices that let developers interact in an asynchronous, distributed manner. Furthermore, the hard time constraints imposed by colocated meetings, the rigid goal of finding defects, and the sole metric of defects found per line of source code encouraged a mentality of "Raise issues, don't resolve them."[2] This mentality limits a group's ability to collectively solve problems and mentor developers.

By conducting asynchronous reviews and eliminating rigid inspection constraints, OSS encourages synergy between code authors, reviewers, and other stakeholders as they discuss the best solution, not the existence of defects. The distinction between author and reviewer can blur such that a reviewer rewrites the code and an author learns from and becomes a reviewer of the new code.

Lesson 2: Frequent Reviews

The earlier a defect is found, the better. OSS developers conduct all-but-continuous, asynchronous reviews that function as a form of asynchronous pair programming.

The longer a defect remains in an artifact, the more embedded it becomes and the more it will cost to fix. This rationale is at the core of the 35-year-old Fagan inspection technique.[1] However, the term "frequent" in traditional inspection processes means that large, completed artifacts are inspected at specific checkpoints that might occur 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—which involves multiple exchanges—usually takes one to two days. Indeed, the feedback cycle is so fast, we consider it a form of continuous review, which often has more similarities with pair programming than with inspection.[6]

To illustrate, we quote Rob Hartill, a former core developer of the Apache project and a founding developer of the Internet Movie Database: "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 developer groups means that the artifacts are unfamiliar to the reviewers tasked with inspecting them. David Parnas and David Weiss first noted

that the resulting inspections are done poorly by unhappy, unfocused, overwhelmed inspectors.[7]

To facilitate early and frequent feedback, OSS projects tend to review smaller changes than proprietary projects,[8] ranging from 11 to 32 lines in the median case.[3] The small size lets reviewers focus on the entire change, and the incrementality reduces reviewers' preparation time and lets them maintain an overall picture of how the change fits into the system.

Equally important is the OSS divide-and-conquer review style that keeps each change logically and functionally independent. For example, a change that combines refactoring a method with fixing a bug in the refactored method won't be reviewed until it's divided into two changes. Developers can either submit these independent changes as a sequence of conceptually related changes or combine them on a single topic or feature branch. Although one developer might have all the required expertise to perform the review, it's also possible that one person will have the required systemwide expertise to understand the refactoring and another will have detailed knowledge of a particular algorithm that contains the bug fix. Intelligently splitting changes lets stakeholders with different expertise independently review aspects of a larger change, which reduces communication and other bottlenecks.

Finally, changes must be complete. Discussing each solution step in a small group can be very effective, but it can be also be tiring. Furthermore, certain problems can be more effectively solved by a single focused developer. Pair programming involves two people in each solution step, but with frequent asynchronous reviews, reviewers only see incremental changes that the author feels are small, independent, and complete solutions.

Lesson 4: Invested, Experienced Reviewers

Invested experts and codevelopers should conduct reviews because they already understand the context in which a change is being made.

Without detailed knowledge of the module or subsystem, reviewers can't reasonably be expected to understand a large, complex artifact they've never seen before. Checklists and reading techniques might force inspectors to focus during an inspection,[7] but they won't turn a novice or incompetent inspector into an expert.

The developers involved in OSS review tend to have at least one to two years' experience with the project; many reviewers have more than four years, and a few have been with the project since its inception.[3] In the OSS projects we studied, we also found that maintainers of a particular code section provided detailed reviews when another developer made a change. The maintainer often had to interact with, maintain, or evolve the changed code. Because codevelopers depend on each other, they have a vested interest in ensuring that the quality of changes is high. Furthermore, because codevelopers are already experts in part of the system under review, they take less time to understand how a small change affects the system.

Although codevelopers have the highest level of investment, many organizations can't afford to keep more than one developer working on the same part of a software system. A simple alternative is to assign regular reviewers to particular subsystems. The reviewers aren't responsible for making changes, but they follow and review changes incrementally. This technique also spreads the knowledge across the development team, mitigating the risk of "getting hit by a bus."

In a small start-up organization, any review costs can be prohibitive. One of the authors of this article, Brendan Cleary, solved this problem in his company with what he called a "reviewer as bug fixer" strategy, in which he periodically assigned one developer to fix a bug in another developer's code. As a bug fixer, the developer becomes a codeveloper as he or she reads, questions, understands, and reviews the bug-related code. This technique combines peer review with the primary task of fixing bugs. It also helps manage turnover risk by giving all developers a broader understanding of the system.

In Table 1, we use the literature and our research findings to compare five reviewer types.

Lesson 5: Empower Expert Reviewers

Let expert developers self-select changes they're 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 can't turn novices into experts, a formal process can't make up for a lack of expertise. Adam Porter and his colleagues reported that the most important predictor of the number of defects detected during review is reviewer expertise; the process has minimal impact.[4]

In a development environment where the artifact author or manager assigns reviews, it can be difficult to know who should perform a review and how many reviewers to involve. The candidates' expertise must be balanced with their workloads and other factors. A rule-of-thumb in the inspection literature is that two reviewers find an optimal number of defects—the cost of adding more reviewers isn't justified by the number of additional defects detected.[9] In OSS, the median is two reviewers per review. These reviewers aren't assigned; instead,

TABLE 1

Reviewer types and their costs, investment level in the code, review quality, and amount of knowledge transfer and community development that occurs during the review.

Reviewer type	Cost	Investment	Quality	Team building
Independent reviewer	Very high	Low	Medium	Low
Pair programming	Very high	Very high	High	High
Codeveloper reviewer	High	High	High	High
Regular incremental reviewer	Medium	Medium	Medium	Medium
Reviewer as bug fixer	Low	Medium	Low	Medium

broadcasting and self-selection lead to natural load balancing across the development team.

Dictating a constant number of reviewers for each change ignores the difference between a simple change that one reviewer can rubber stamp and a complex one that might require a discussion with the whole development team. The advantage of self-selection is that it's up to the developers, who have the most detailed knowledge of the system, to decide on the level of review given to each change.

On the other hand, self-selection can end in some changes being ignored. Managers can use tools to automatically assign unreviewed changes to reviewers. However, unselected changes might indicate areas of the code base that pose a problem, such as areas that only a single developer understands.

Recommendation 1: Lightweight Review Tools

Tools can increase traceability for managers and help integrate reviews with the existing development environment.

OSS developers rely on information broadcast and use minimalistic tool support. For example, the Linux Kernel Mailing List has a median of 343 messages per day, and the OSS developers we interviewed received thousands of messages per day.[10] There are techniques to manage this email barrage, but it's difficult to track the review process for reporting and quality assurance, and it's easy to inadvertently ignore reviews. Furthermore, the frequency of small changes can lead to fragmentation, which makes it difficult to find and review a feature that consists of multiple changes.

Tools can help structure reviews and integrate them with other development systems. Typically, they provide

- side-by-side highlighted changes to files (diffs);
- inline discussion threads that are linked to a line or file;
- capability to hide or show additional lines of context and to view a diff in the context of the whole file;
- capability 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 to show pending reviews and alert code authors and reviewers who haven't responded to assignments;
- integration with email and development tools;
- notification and assignment of reviews to individuals and groups of developers; and
- metrics to gauge review efficiency and effectiveness.

Table 2 compares some popular peer review tools.

Recommendation 2: Nonintrusive Metrics

Mine the information trail left by asynchronous reviews to extract lightweight metrics that don't 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 to produce software.

For example, formally recording a defect is a cognitively expensive task, sidetracking developers who are discussing a change and forcing them to formally agree on and record a defect. Tool support doesn't fix this problem. At AMD, Julian Ratcliffe found that defects were underreported despite the simple CodeCollabotor reporting mechanism: "A closer look at the review archive shows that reviewers were mostly engaged in discussion, using the comment threads to fix issues instead of logging defects."[11]

Is the defect or the discussion more important? In the Linux community, the amount of discussion on a

TABLE 2

Comparison of some popular peer review tools.

Tool	Main advantages	Main disadvantages
CodeCollaborator	Supports instant messaging-style discussion of LOC, metric reporting, and tight integration with multiple development environments, such as Eclipse and Visual Studio	Commercial license fee
Crucible	Integrates with the Jira bug tracker and other Atlassian products	Commercial license fee
ReviewBoard	Has a free, full-featured Web interface for review	Requires setup and maintenance on an in-house server
Rietveld	Runs on top of Google App Engine, so it's quick and easy to start reviewing; supports Subversion development (Gerrit is a git-specific implementation of Rietveld)	Requires public hosting on Google Code or setting up the review system on an in-house server
CodeStriker	Has a Web interface that supports traditional inspection	An older tool that lacks good support for lightweight review techniques

particular change is an indicator of code quality. Indeed, Linus Torvalds, who maintains the current release on the Linux operating system, has rejected code, not because it's incorrect, but because not enough people have tried it and discussed it on the mailing list. To Torvalds, the potential system benefit of accepting code that hasn't been discussed by a group of experts doesn't outweigh the risks.

Turning the amount of discussion during a review into a metric is trivial if a tool records discussions and associates them with file changes. On this basis, a manager might ask developers whether they think the group has adequately discussed a part of the system before its release. In our work, we've demonstrated the extraction of many nonintrusive, proxy metrics from review archives.[12]

Recommendation 3: Implementing a Review Process

Large, formal organizations might benefit from more frequent reviews and more overlap in developers' work to produce invested reviewers. However, this style of review will likely be more amenable to agile organizations that are looking for a way to run large, distributed software projects.

OSS has much in common with agile development and the Agile Manifesto:[13,14]

- a preference for working software over documentation and for empowering individuals over imposing a rigid process;
- handling changes by working in small increments rather than following a rigid plan; and
- working closely with the customer rather than negotiating contracts.

The most striking difference between the development methodologies is that agile supports small, colocated developer teams, while OSS projects can scale to large, distributed teams that rarely, if ever, meet in a colocated setting. OSS projects broadcast all communication—discussions, code changes, and reviews—to the entire community. The need for the entire community to see all communication is so strong that when a company pays colocated developers to work on an OSS project, it often requires them to summarize and broadcast all in-person discussion to the community.

Software developers in most development companies are accustomed to communicating in person, so they might not welcome this practice. However, peer review has proved more effective in an asynchronous environment than in a synchronous, colocated one. Companies with large, distributed development teams might consider using frequent, asynchronous reviews involving codeveloper discussions of small, functionally independent changes as a substitute for pair programming.

Practitioners from both the OSS community and software companies have driven the development of lightweight peer review and supporting tools. OSS practices have evolved to maintain code quality efficiently within a distributed development group, and many companies are already adopting a lightweight, tool-supported review approach, including AMD[11] and Cisco.[15] We're currently working with the Canadian defense department to develop an agile review style that fits its development teams. We're also actively seeking collaborations with developers and companies who use a lightweight peer review. Our

goal is provide a systematic and practical understanding of contemporary peer review.

References

1. M. Fagan, "A History of Software Inspections," *Software Pioneers: Contributions to Software Engineering*, Springer, 2002, pp. 562–573.
2. P.M. Johnson, "Reengineering Inspection," *Comm. ACM*, vol. 41, no. 2, 1998, pp. 49–52.
3. P.C. Rigby, "Understanding Open Source Software Peer Review: Review Processes, Parameters and Statistical Models, and Underlying Behaviours and Mechanisms," 2011; http://thechiselgroup.org/rigby-dissertation.pdf.
4. A. Porter et al., "Understanding the Sources of Variation in Software Inspections," *ACM Trans. Software Eng. Methodology*, vol. 7, no. 1, 1988, pp. 41–79.
5. L.G. Votta, "Does Every Inspection Need a Meeting?" *SIGSOFT Software Eng. Notes*, vol. 18, no. 5, 1993, pp. 107–114.
6. L. Williams, "Integrating Pair Programming into a Software Development Process," *Proc. 14th Conf. Software Eng. Education and Training*, IEEE, 2001, pp. 27–36.
7. D.L. Parnas and D.M. Weiss, "Active Design Reviews: Principles and Practices," *Proc. 8th Int'l Conf. Software Eng.* (ICSE 85), IEEE CS, 1985, pp. 132–136.
8. A. Mockus, R.T. Fielding, and J. Herbsleb, "Two Case Studies of Open Source Software Development: Apache and Mozilla," *ACM Trans. Software Eng. and Methodology*, vol. 11, no. 3, 2002, pp. 1–38.
9. C. Sauer et al., "The Effectiveness of Software Development Technical Reviews: A Behaviorally Motivated Program of Research," *IEEE Trans. Software Eng.*, vol. 26, no. 1, 2000, pp. 1–14.
10. P.C. Rigby and M.-A. Storey, "Understanding Broadcast Based Peer Review on Open Source Software Projects," *Proc. 33rd Int'l Conf. Software Eng.* (ICSE 11), ACM, 2011, pp. 541–550.
11. J. Ratcliffe, "Moving Software Quality Upstream: The Positive Impact of Lightweight Peer Code Review," *Proc. Pacific NW Software Quality Conf.* (PNSQC 09), 2009, pp. 171–180; www.pnsqc.org/past-conferences/2009-conference.
12. P.C. Rigby, D.M. German, and M.-A. Storey, "Open Source Software Peer Review Practices: A Case Study of the Apache Server," *Proc. 30th Int'l Conf. Software Eng.* (ICSE08), IEEE CS, 2008, pp. 541–550.
13. K. Beck et al., *The Agile Manifesto*, 2001; http://agilemanifesto.org.
14. S. Koch, "Agile Principles and Open Source Software Development: A Theoretical and Empirical Discussion," *Extreme Programming and Agile Processes in Software Eng.*, LNCS 3092, J. Eckstein and H. Baumeister, eds., Springer 2004, pp. 85–93.
15. J. Cohen, *Best Kept Secrets of Peer Code Review*, white paper, Smart Bear, 2006; http://smartbear.com/solutions/white-papers/best-kept-secrets-of-peer-code-review

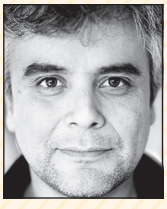
ABOUT THE AUTHORS

PETER C. RIGBY is an assistant professor of software engineering at Concordia University in Montreal, Canada. His research interests focus on understanding how developers collaborate to produce successful software systems. Rigby received his PhD in computer science at the University of Victoria, and the lessons and recommendations reported in this article are largely based on his dissertation. Contact him at peter.rigby@concordia.ca.

BRENDAN CLEARY is a research fellow at the University of Victoria. His research interests focus on managing commercial and research projects, and he's the founder of a university spin-out company. Cleary has a PhD in computer science from the University of Limerick, Ireland. Contact him at bcleary@ uvic.ca.

FREDERIC PAINCHAUD is a defence scientist at Defence Research and Development Canada and a part-time PhD student in computer science at Université Laval. His research interests include software architectural risk analysis, static and dynamic code analysis, and lightweight peer review. Painchaud has a master's degree in computer science from Université Laval. Contact him at 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, Canada. Her research interests center on technology to help people explore, understand, and share complex information and knowledge. Storey received her PhD in computer science from Simon Fraser University. Contact her at 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. German received his PhD in computer science from the University of Waterloo, Canada. Contact him at dmg@uvic.ca or through his website at turingmachine.org.