



Figure 1: Code reviews go electronic and global.

DOI:10.1145/1378727.1378744

New collaboration tools allow geographically distributed software-development teams to boost the venerable concept of code review.

BY BERTRAND MEYER

Design and Code Reviews in the Age of the Internet

CODE REVIEWS ARE A standard practice of software engineering. Rather, they are a standard practice of the software engineering literature. Widely recommended, certainly, but how widely practiced, I am not sure.

In the development of EiffelStudio, a large integrated development environment (IDE), Eiffel Software has begun to apply code reviews with the added twist that the developers work on three continents. A distributed setup is increasingly common in the IT industry, though not always in such an extreme form. It naturally leads to distributing the reviews as well. This decision forced our development group to depart from the standard scheme described in the literature² and take a fresh look at the concept. Some of what initially appeared as constraints—

the impossibility of ever having all the people involved at the same time in the same room—turned out in practice to be beneficial, encouraging us to emphasize the written medium over verbal discussion, conduct much of the process prior to the actual review meeting, and take advantage of communication tools to allow several threads of discussion to proceed in parallel during the meeting itself. We also expanded our reviews, beyond just code, to cover design and specification as well. The process relies on modern, widely available communication and collaboration tools, most introduced over the past few years with considerable room for improvement. This article describes some of the lessons our team has learned with the hope they will be useful to other teams practicing distributed development.

Michael Fagan of IBM introduced the concept of “code inspections,” his original name, in a 1976 article.¹ Whether inspection or review, the idea is to examine some element of code in a meeting typically attended by perhaps eight people, with the aim of finding flaws or elements that should be improved. This is the meeting’s only goal. It is not intended to assess the code’s author, though in practice it is not easy to avoid doing so, especially when the manager is present; neither should it serve to correct deficiencies, only to uncover them.

The code and any associated elements are circulated a few days in advance. The meeting, which typically takes a few hours, includes the author, other developers competent to assess the code, a meeting chair (not the manager) who moderates the discussion, and a secretary who records it, producing a report with specific recommendations. Some time later, the author responds to the report by describing whether and how the recommendations have been carried out.

This is the basic idea behind a traditional code review and is often criticized on a variety of grounds. Advocates of extreme programming point

out that reviews may be superfluous if the project practices pair programming. Others note that when it comes to finding code flaws (such as a lurking buffer overflow) static analysis tools are more effective than human inspection. In any case, the process is time-consuming; most teams apply it on the entire code, as well as on samples. Still, code reviews remain an important tool in the battery of accepted “best practices” for improving software quality.

Our group has found that the exercise is indeed useful when adapted to the modern world of software development. The first extension is to include design and specification. Many recent

references concerning code reviews focus on detecting low-level code flaws, especially security risks. This is important but is increasingly a task for automated tools, not for humans. Our experience suggests that abstract program interface (API) design, architecture choices, and other specification and design issues are just as worthy of a reviewer’s time and play an increasingly important part in our reviews.

Among these traditional principles, one that should definitely be retained in the new distributed context is that reviews must focus on identifying deficiencies, not attempt to correct them. With the advent of better

software technology it may be tempting to couple review activities with actual changes in software repositories; one environment that supports Web-based review—Code Collaborator, www.smartbear.com—allows this by coupling the review tools with a configuration-management system. Such coupling may be risky; updating software—even for simple code changes with no effect on specification and design—is a delicate matter best performed in an environment free from the time pressure inherent in a review session.

Distributed Review

All the descriptions of code reviews I have read in the literature present a review as a physical meeting among people in the same room. This is hardly applicable to today’s increasingly dominant model of software development: distributed teams spread over many locations and time zones.³ At Eiffel Software, we were curious to see whether we could indeed apply the model; our first experience—still only a few months old and subject to refinement—suggests that thanks to the Internet and modern communications technology a distributed setup is less a hindrance than a benefit and that today’s technology provides a strong incentive to revive and expand the idea of the review.

Though the EiffelStudio development team includes members in California, China, Russia, and Western Europe, it still manages to have a weekly technical meeting, with some members agreeing to stay up late; for example, in the winter, 8 A.M. to 9 A.M. in California means 5 P.M. to 6 P.M. in Western Europe, 7 P.M. to 8 P.M. in Moscow, and midnight to 1 A.M. in Shanghai (see Figure 1). We devote every second or third such meeting to a design and code review.

Although many of the lessons we have learned should be valid for any team, some of the specifics of our reviews may influence our practice and conclusions. Our meetings, both ordinary ones and those devoted to reviews, last exactly one hour. We are strict on the time limit, obviously because it’s late at night for some team members but also to avoid wasting the time of a group of highly competent develop-

Figure 2: Excerpts from a chat session during a review (names blocked out).



ers. This constraint is an example of a limitation that has turned out to be an advantage, forcing us to organize both the reviews and other meetings carefully and professionally.


Almost all of our development is done in Eiffel; one aspect of this choice that influences the review process is that Eiffel applies the “seamless development” principle, treating specification, design, and analysis as a continuum rather than as a sequence of separate steps (using, for example, first UML then a programming language); the Eiffel language serves as the common notation throughout. This has naturally caused the extension of the traditional review to design reviews, an extension that may also be desirable for teams using other development languages and a less-seamless process. Another aspect that influences EiffelStudio development is that, since IDEs are our business, the tool we produce is also the tool we use (following the “eat your own dog food” principle); we again feel the results would not fundamentally change for other kinds of software development.

Distributed reviews need support from communication and collaboration tools; we essentially rely on four such tools:


Voice communication similar to a conference call. We started with Skype but now use it only as a backup; our primary VoIP solution is a tool called X-Lite; such technology choices are subject to reassessment as the tools evolve;

Written communication. We retain Skype’s chat mechanism, so a window available to all participants is active throughout the review;

Google Docs for shared documents. Providing a primitive Microsoft-Word-like editing framework, Google Docs works on the Web so several people are able to update a given document at the same time. The means of resolving editing conflicts is fine-grained: most changes go through, even if someone else is simultaneously modifying the document; only if two people are changing the very same words does the tool reject the requests. While not perfect, Google Docs is an effective tool for collaborative editing, with the advantage that text can be pasted into and from Microsoft Word documents with approximate preservation of format;



What is remarkable in the current setup is that we have not yet identified a need for specialized review software, being content enough with general-purpose widely available communication and collaboration tools.



WebEx sharing tool for sharing screens. We find this tool (one of several, including Adobe Connect, on the market) especially useful for running a demo of, say, a new proposal for a graphical-user-interface idea or other element developers might have just put together on a workstation; and

Wiki pages. The EiffelStudio community, involving both Eiffel Software developers and numerous outside contributors, has a Wiki-based site (dev.eiffel.com) with hundreds of documentation and discussion pages. The site is useful, although the Wiki mechanism, with its traditional edit cycle—start editor, make changes, update page, refresh—is less convenient than Google Docs for working on a common document during a meeting.

Among the ideas described here, the choice of tools is the most likely candidate for quick obsolescence. The technology is evolving so quickly that a year or two from publication the solutions might be fairly different. What is remarkable in the current setup is that we have not yet identified a need for specialized review software, being content enough with general-purpose widely available communication and collaboration tools.

Lessons Learned

Here are some of the lessons we have learned from our distributed code review:

First, *scripta manent*, or “prefer the written word.” We have found that a review works much better when it is organized around a document. The team members produce a shared document (currently in Google Docs) ahead of the meeting and update it in real time during the meeting.

The “unit of review” is a class or sometimes a small number of closely related classes. A week ahead of the review, the software’s author, following a standard structure described in the following sections, prepares the shared document with links to the actual code.

One way this process differs from a traditional review is a practice we had not planned for when we first put reviews in place but which quickly imposed itself through experience: Most of the work is done offline before the meeting. Our original thinking was

that it was preferable to limit the advance work and delay written comments until a couple of days before the meeting to avoid reviewers influencing one another too much. Experience showed that such concern was misplaced; interaction among reviewers, before, during, and after the meeting, is one of the most effective aspects of the process.

Prior to the meeting, the reviewers provide their comments on the review page (the shared document); the code author then responds just below the comments on the same page. The benefit of this approach is that it saves time. Before we systematized it we spent considerable time in the meeting on noncontroversial issues; in fact, our experience suggests that with a competent group of developers most comments and criticisms are readily accepted by the code's author. We should instead be spending our meeting time on the remaining points of disagreement. Otherwise we end up replaying the typical company board meeting as described in the opening chapter of C. Northcote Parkinson's *Parkinson's Law*.⁴ (The two items on the agenda are the color of bicycles for the mail messengers and whether to build a nuclear plant. Everyone has an opinion on colors, so the first item takes 59 minutes, ending with the decision to form a committee; the next decision is taken in one minute, with a resolution to let the CEO handle the matter.) Unlike this pattern, the verbal exchange in an effective review should target the issues that truly warrant discussion.

For an example of a document produced before and during one of our code reviews, see dev.eiffel.com/reviews/2008-02-sample.html, which gives a good idea of the process. For a more complete picture of what actually goes on during the meeting, also see the discussion extract from the chat window (names blocked out) in Figure 2.

Scope of Review

The standard review-page structure consists of nine sections dividing the set of software characteristics under review:

- ▶ Choice of abstractions;
- ▶ Other aspects of API design;
- ▶ Other aspects of architecture (such

as choice of client links and inheritance hierarchies);

- ▶ Contracts;
- ▶ Implementation, particularly the choice of data structures and algorithms;
- ▶ Programming style;
- ▶ Comments and documentation (including indexing/note clauses);
- ▶ Global comments; and
- ▶ Adherence to official coding practices.

The order of these sections goes from more high-level to more implementation-oriented. Note that the first four concern not just code but architecture and design as well:

- ▶ The choice of abstractions is the key issue of object-oriented design. Developers discuss whether a certain class is really justified or should have its functionalities merged with another's, or, conversely, whether an important potential class has been missed;
- ▶ API design is essential to the usability of software elements by other elements and as a consequence to reusability. We enforce systematic API design conventions through strong emphasis on consistency across the entire code base. This aspect of software development is particularly suitable for review; and
- ▶ Other architectural issues are also essential to good object-oriented development; the review process is useful (during both the preparatory phase and the meeting itself) to address such questions as whether a class should inherit from another or just be its client.

Algorithm design (the fifth section in the list) is another good candidate for discussion during the meeting.

In our process, the lower-level sections—programming style, comments and documentation, global comments, and coding practices—are increasingly handled before the review meeting (in writing), enabling us to devote the meeting itself to the deeper, more delicate issues.

The division into nine sections and the distribution of work through written comments prior to the meeting and in-meeting verbal discussions yield the following benefits:

- ▶ The group saves time; precious personal interaction time is reserved only for the topics that really need it;
- ▶ Discussing issues in writing makes

it possible to include more thoughtful comments; participants can take care to express their observations, including criticism of design and implementation decisions and corresponding responses, more easily than through a verbal conversation alone;

- ▶ The written support allows editing and revision;
- ▶ A record is produced. Indeed, the review needs no secretary or the tedious process of writing minutes. The review page (in its final stage after joint editing) is the minutes;

▶ Verbal discussion time is much more interesting since it addresses issues of substance. The dirty secret of traditional code reviews is that most of the proceedings are boring to most participants, each of whom is typically interested in only a subset of all the items discussed. With an electronic meeting each participant resolves issues of specific concern in advance and in writing; the verbal discussion is devoted to the controversial and hence most interesting stuff; and


▶ In a group with contentious personalities, one may expect that expressing comments in writing will also help defuse tension. (I say "may expect" because I don't know this from experience; our developer group is not contentious.)

Through our electronic meetings, not just code reviews, another example has emerged of how constraints can be turned into benefits. Individually, most people (apart from, say, piano players) are most effective when doing one thing at a time, but collectively a group of humans is pretty good at multiplexing. When was the last time you spent an hour-long meeting willingly focused throughout on whatever issue was under discussion at the moment? Even the most attentive participants, disciplined to not let their minds wander off topic, react at different speeds; you may be thinking deeply about some previous item, while the agenda has moved on; you may be ahead of the game; or you may have something to say that complements the comments of the current speaker but do not want to interrupt her. This requires multithreading, but a traditional meeting is sequential. In our code reviews and other meetings we have learned to practice a kind of organic multithread-


ing: Someone may be talking; someone else may be writing a comment in the chat window (such as a program extract that illustrates a point under discussion or a qualification of what is being said); others may be updating the common document; and yet someone else may be preparing next week's document or a Wiki page at dev.eiffel.com. The dynamics of such meetings are amazing to behold, with threads progressing in parallel while everyone remains on target and alert.

Team distribution is a fact of life in today's software development and can be a great opportunity, as well as a challenge, to improve the engineering of software. I also practice distributed team development in an academic environment. ETH Zurich offers a course called Distributed and Outsourced Software Engineering, or DOSE, specifically devoted to the issues and techniques of distributed development. In fall 2007, it meant, for the first time, a cooperative project involving several universities. This was a trial run, and we are now expanding the experience; for details see se.ethz.ch/dose/. Participation is open to any interested university worldwide; the goal is to let students discover and confront the challenges of distributed development in the controlled environment of a university course.

Not all of our practice may be transposable to other contexts. The core EiffelStudio group includes only about 10 people who know each other well and have worked together for several years; we developed these techniques together, learning from our mistakes and benefiting from recent advances in the communication technology discussed here. But even if all the details of our experience cannot be generalized, distributed software development, and with it distributed reviews, are the way of the future. Even more so considering that the supporting technology is still in its infancy. As recently as 2006, most of the communication tools we use today did not exist; in 2003 none of them did. It is ironic now to recall the talks I heard over the years about "computer-supported cooperative work"—fascinating but remote from anything my colleagues or I could use. Suddenly comes the Web, VoIP solutions for common folk, shared editing



Interaction among reviewers, before, during, and after the meeting, has turned out to be one of the most effective aspects of the process.




tools, and new commercial offerings, and the gates to globalized cooperative development open without fanfare.

The tools are still fragile; we waste too much time on meta communication ("Can you hear me?", "Did Peter just disconnect?," "Bill, mute your microphone."), calls are cut off, and we lack a really good equivalent of a shared whiteboard. Other aspects of the process also still need improvement; for example, how can we make our review results seamlessly available as part of the EiffelStudio open-source development site based on Wiki pages? All this will be corrected in the next few years. I also hope that courses like DOSE and other academic efforts will enable the commercial world to understand better what makes collaborative development succeed or fail.

This is not just an academic issue. Eiffel Software's experience in collaborative development—whereby each meeting brings new insight—suggests that something fundamental has changed, mostly for the better, in the software-development process. As for code reviews, I do not expect ever again to get stuck for three hours in a windowless room with a half dozen other programmers poring over boring printouts.

Acknowledgment

I am grateful to the EiffelStudio developers for their creativity and the team spirit that enabled them collectively to uncover and apply the techniques discussed here. 

References

1. Fagan, M.E. Design and code inspections to reduce errors in program development. *IBM Systems Journal* 15, 3 (1976), 182–211; www.research.ibm.com/journal/sj/153/ibmsj1503C.pdf.
2. Ghezzi, C., Jazayeri, M., and Mandrioli, D. *Fundamentals of Software Engineering, Second Edition*. Prentice Hall, Upper Saddle River, NJ, 2002.
3. Meyer, B. and Piccioni, M. The allure and risks of a deployable software engineering project. In *Proceedings of the 21st IEEE-CS Conference on Software Engineering Education and Training* (Charleston, SC, Apr. 2008).
4. Parkinson, C. Northcote. *Parkinson's Law: The Pursuit of Progress*. John Murray, London, 1958.

Bertrand Meyer (Bertrand.Meyer@inf.ethz.ch) is a professor of software engineering at ETH Zurich, the Swiss Federal Institute of Technology, and chief architect of Eiffel Software, Santa Barbara, CA.

Copyright of Communications of the ACM is the property of Association for Computing Machinery and its content may not be copied or emailed to multiple sites or posted to a listserv without the copyright holder's express written permission. However, users may print, download, or email articles for individual use.