Design and code reviews in the age of the Internet

Bertrand Meyer

ETH Zurich and Eiffel Software

Bertrand.Meyer@inf.ethz.ch
http://se.ethz.ch http://eiffel.com

Code reviews are one of the standard practices of software engineering. Or let's say that they are a standard practice of the software engineering literature. They are widely recommended; how widely practiced, I am not sure.

Eiffel Software has applied code reviews to the recent developments of EiffelStudio, a large IDE (interactive development environment) whose developers are spread over three continents. This distributed setup forced us to depart from the standard scheme as described in the literature and led to a fresh look at the concept; some of what appeared initially as constraints (preventing us from ever having all the people involved at the same time in the same room) turned out to be beneficial in the end, encouraging us in particular to emphasize the written medium over verbal discussions, to conduct a large part of the process prior to the actual review meeting, and to take advantage of communication tools to allow several threads of discussion to proceed in parallel during the meeting itself. Our reviews are not just about code, but encompass design and specification as well. The process relies on modern, widely available communication and collaboration tools, most of them fairly recent and with considerable room for improvement. This article describes some of the lessons that we have learned, which may also be useful to others.

Code review concepts

Michael Fagan from IBM introduced "code inspections", the original name, in a 1976 article[1]. Inspection or review, the general idea is to examine some element of code in a meeting of (typically) around eight people, with the aim of finding flaws or elements that should be improved. This is the *only* goal:

- The review is not intended to assess the programmer — although in practice this is not so easy to avoid, especially if the manager is present.
- The review is not intended to *correct* deficiencies, only to uncover them.

The code and any associated elements are circulated a few days in advance. The meeting typically lasts a few hours; it includes the author, a number of other developer competent

[1] M.E. Fagan, *Design and Code inspections to reduce errors in program development*, IBM Systems Journal, Vol. 15, No 3, 1976, pages 182-211; at www.research.ibm.com/journal/sj/153/ibmsj1503C.pdf.

to assess the code, a meeting chair who moderates the discussion (and should not be the manager), and a secretary who records it, producing a report with specific recommendations. Some time later, the author should respond to the report by describing whether and how the recommendations have been carried out.

Such is the basic idea of traditional reviews. It is often criticized on various grounds. Advocates of Extreme Programming point out that reviews may be superfluous if the project is already practicing pair programming. Others note that when it comes to finding code flaws — a looming buffer overflow, for example — static analysis tools are more effective than human inspection. In any case, the process is highly time-consuming; most teams that apply it perform reviews not on the entire code but on samples. Still, code reviews remain one of the tools in a battery of accepted "best practices" for improving software quality.

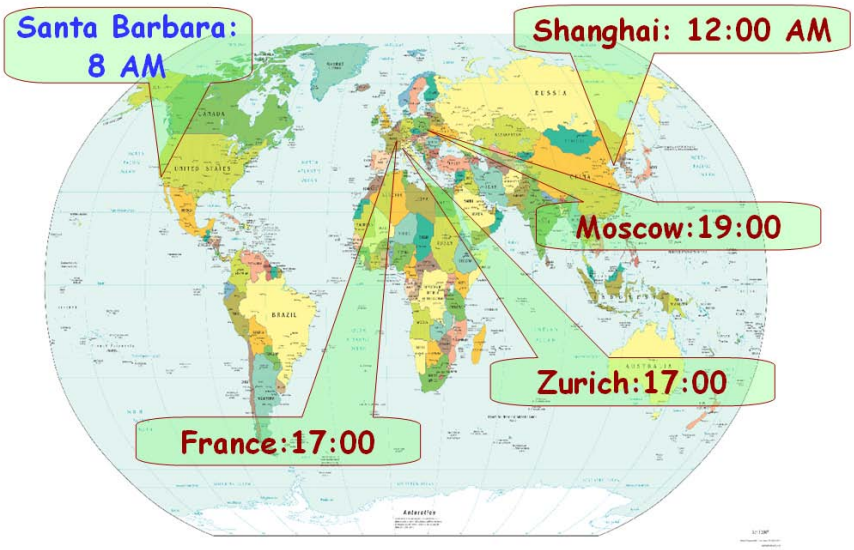
We have found that the exercise is indeed useful if adapted to the modern world of software development. The first extension is to include design and specification. Many of the recent references on code reviews focus on detecting low-level code flaws, especially security risks. This is important but increasingly a task for tools rather than humans. We feel that API (abstract program interface) design, architecture choices and other specification and design issues are just as worth the reviewers' time; in our reviews these aspects have come to play an increasingly important part in the discussions.

Among the traditional review principles that should in our view be retained is the rule that the review should only identify deficiencies, not attempt to correct them. With the advent of better software technology it may be tempting to couple review activities with actual changes to the software repositories; one tool that supports web-based review, Code Collaborator[2], allows this through coupling with a configuration management system. We feel that this is risky. Updating software — even for simple code changes, not touching specification and design — is a delicate matter and should be performed carefully, outside of the time pressures inherent in a review.

A distributed review?

All the descriptions of code reviews I have seen in the literature talk of a review as a physical meeting with people sitting in the same room. This is hardly applicable to the model of software development that is increasingly dominant today: distributed teams, split over many locations and time zones. At Eiffel Software we were curious to see whether we could apply the model in such a setup; our first experience — still fresh and subject to refinement — suggest that thanks to the Internet and modern communication mechanisms this setup is less a hindrance than a benefit, and that today's technology actually provides a strong incentive to revive and expand the idea of the review.

[2] http://www.smartbear.com/



The EiffelStudio development team has members in California, Western Europe, Russia and China. In spite of this we manage to have a weekly technical meeting, with some members of the team agreeing to stay up late. (In the winter 8 to 9 AM California means 5 to 6 PM in Western Europe, 7 to 8 PM in Moscow, and midnight to 1 AM in Shanghai.) We are now devoting one out of three such meetings to a code review.

Constraints and technology

Although many of the lessons should be valid for any other team, here are some of the specifics of our reviews, which may influence our practice and conclusions.

Our meetings, whether ordinary ones or for code reviews, last one hour. We are strict on the time limit, obviously because it's late at night for some of the participants, but also because it makes no sense to waste the time of a whole group of highly competent developers. This schedule constraint is an example of limitation that has turned out to be an advantage, forcing us to organize the reviews and other meetings seriously and professionally.

Almost all of our development is done in Eiffel; one aspect that influences the review process is that Eiffel applies the "seamless development" principle which treats specification, design and analysis as a continuum, rather than a sequence of separate steps (using, for example, first UML then a programming language); the Eiffel language serves as the single notation throughout. This has naturally caused the extension to design reviews, although we think that this extension is desirable for teams using any other development language and a less seamless process. Another aspect that influences our process is that, since IDEs are our business, the tool we produce is also the tool we use (this is known as the "eat your own dog food" principle); but again we feel the results would not fundamentally change for another kind of software development.

Distributed reviews need support from communication and collaboration tools. At the moment we essentially rely on four tools:

- The reviews require voice communication, similar to a conference call. We started with Skype but now use it only as a backup since we found too many problems in intensive use. Another Voice over IP solution (X-Lite) is the current voice tool.
- In parallel, we use written communication. For this we retain Skype's chat mechanism. A chat window involving all the participants remains active throughout the review.
- For shared documents, we use Google Docs, which provides a primitive Microsoft-Word-like editing framework, but on the Web so that several people can update a given document at the same time. The conflict resolution is fine-grained: most of the time changes are accepted even if someone else is also modifying the document; only if two people are changing exactly the same words does the tool reject the requests. While not perfect, Google Docs provides a remarkable tool for collaborative editing, with the advantage that texts can be pasted to and from Microsoft Word documents with approximate preservation of formats.
- It is also important to be able to share a screen, for example to run a demo of a new proposal for a GUI (graphical user interface) idea or other element that a developer has just put together on his or her workstation. For this we use the WebEx sharing tool.
- Wiki pages, especially the developer site at http://dev.eiffel.com, are also useful, but less convenient than Google Docs to edit during a meeting.

Clearly the choice of tools is the part of this article that is most clearly dependent on the time of writing. The technology is evolving quickly enough that a year or two from now the solutions might be fairly different. What is remarkable in the current setup is that we have not so far found a need for specialized reviewing software but been content enough with general-purpose communication and collaboration tools.

Reviews for the 21st century

Here are some of the lessons we have learned.

First, *scripta manent*: prefer the written word. We have found that a review works much better if it is organized around a document. For any meeting we produce a shared document (currently Google Docs); the whole meeting revolves around it. The document is prepared ahead of the meeting, and updated in real time during the meeting (while, as noted, the software itself is not).

The **unit of review** is a class, or sometimes a small number of closely related classes. A week in advance of the review the code author prepares the shared document with links to the actual code. It follows a standard structure described below.

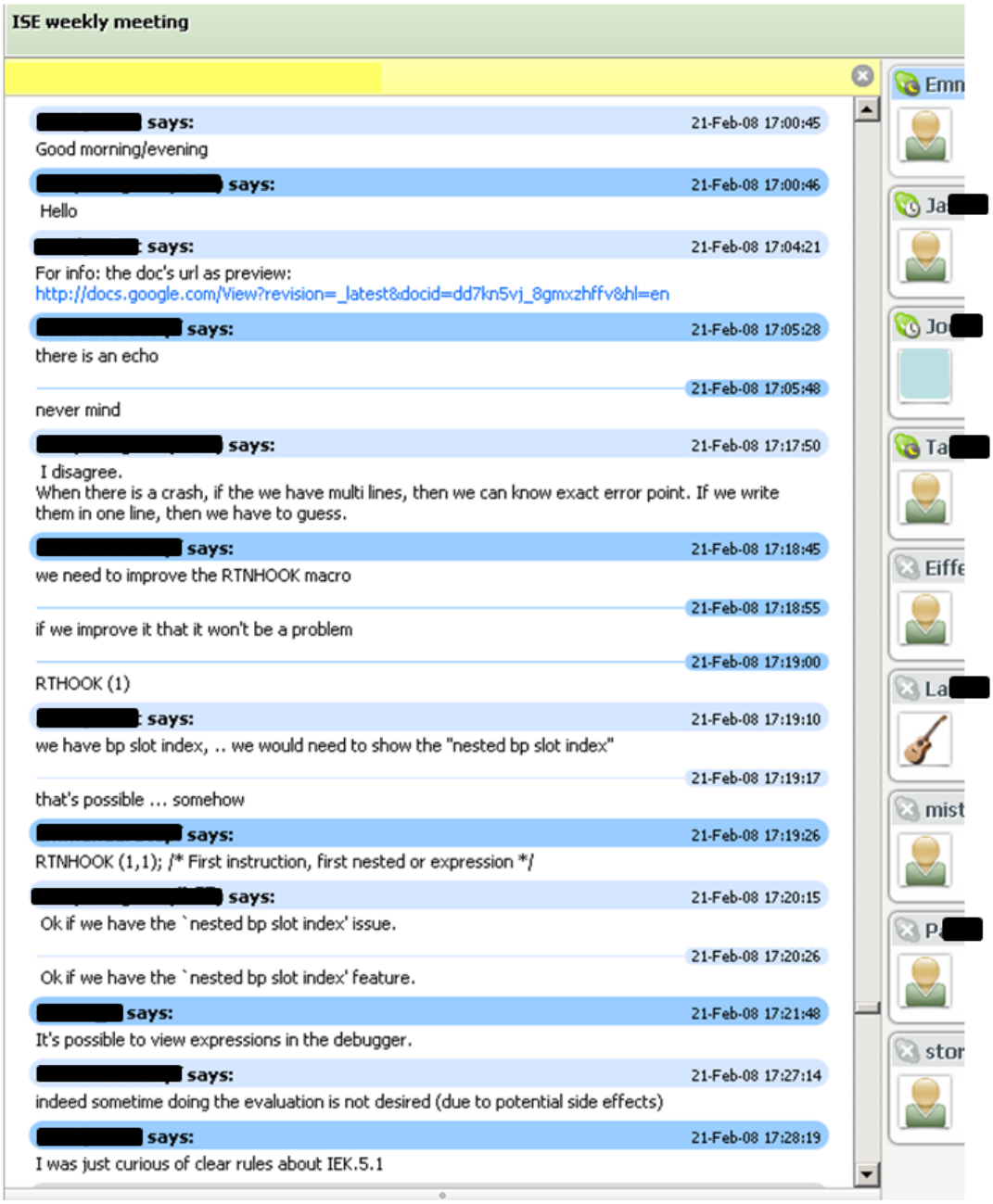
One of the differences with a traditional review is a practice which we hadn't planned, but which quickly imposed itself through experience: most of the work occurs off-line, before the meeting. Our original intuition was to limit the amount of advance work and delay written comments to a couple of days before the meeting, to avoid reviewers influencing each other too much. This was a mistake; interaction between reviewers, before, during and after the meeting, has turned out to be one of the most effective aspects of the process.

The reviewers provide their comments on the review page (the shared document); the code author can then respond at the same place. The benefit of this approach is that it saves considerable time. Before we systematized it we were spending time, in the actual meeting, on non-controversial issues; in fact, our experience suggests that with a competent group most of the comments and criticisms will be readily accepted by the code's author. We should instead spend the meeting 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*. (There are two items on the agenda: 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 with this all too common pattern, the verbal exchanges can target the issues that truly warrant discussion.

For an actual example of a document produced before and during one of our code reviews, see

http://dev.eiffel.com/reviews/2008-02-sample.html

which gives a good idea of the process. For a complete picture you would need to see the full discussion in the chat window (see a small extract below, names blacked out) and hear a recording of the discussion.



Review scope

The standard review page structure consists of 8 sections, dividing the set of aspects to be examined:

1. Choice of abstractions
2. Other aspects of API design
3. Other aspects of architecture, e.g. choice of client links and inheritance hierarchies

4. Implementation, in particular choice of data structures and algorithms
5. Programming style
6. Comments and documentation (including indexing/note clauses)
7. Global comments
8. Coding practices

This goes from more high-level to more implementation-oriented. Note in particular sections 1 to 3, making it clear that we are talking not just about code but about reviewing architecture and design:

- The choice of abstractions (1) is the key issue of object-oriented design. Developers will 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 (2) is essential to the usability of software elements by others, and in particular to reuse. We enforce systematic API design conventions, with a strong emphasis on consistency across the entire code base. This aspect is particularly suitable for review.
- Other architectural issues (3) are also essential to good object-oriented development; the review process is useful, both during the preparatory phase and during the meeting itself, to discuss such questions as whether a class should really inherit from another or instead be a client.

Algorithm design (4) is also a good item for discussion.

In our process the lower-level aspects, 5 to 8, are increasingly handled before the review meeting, in writing, enabling us to devote the meeting time to the deeper and more delicate issues.

Making the process effective

.
We have identified the following benefits of the choices described.

- The group saves time. Precious personal interaction time is reserved for topics that require it.

- Discussing issues in writing makes it possible to have more thoughtful comments. Participants can take care to express their observations — criticism of design and implementation decisions, and the corresponding responses — properly. This is easier than in a verbal conversation.

- The written support allows editing and revision.

- There is a record. Indeed the review no longer needs a secretary or the tedious process of writing minutes: the review page in its final stage, after joint editing, *is* the minutes.

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

- In a group with contentious personalities, one may expect that expressing comments in writing will help defuse tension. (I am writing "may expect" because I don't know from experience — our group is not contentious.)

Through our electronic meetings — not just code reviews — another example has emerged of how constraints can become benefits. Individually, most of us apart from piano players may be 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 a one-hour meeting willingly focused at every minute on the issue then under discussion? Even the most attentive, careful not to let their minds wander off topic, react at different speeds from others: you may be thinking deeper about the previous item even when 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, whom you don't want to interrupt. But this requires multithreading and a traditional meeting is sequential. In our code reviews and other meetings we have quickly learned to practice a kind of organic multithreading: someone is talking; someone is writing a comment in the chat window (e.g. a program extract that illustrates a point under discussion, or a qualification of what is being said); a couple of people are updating the common document; someone else is preparing next week's document, or a Wiki page at http://dev.eiffel.com. It is amazing to see the dynamics of such meetings, with the threads progressing in parallel while everyone remains on target, and not bored.

An academic endeavor

Team distribution is a fact of life in today's software development, and as well as a challenge it can be a great opportunity to improve the engineering of software. I am also practicing it in an academic environment. ETH Zurich offers a course specifically devoted to studying, in the controlled environment of an academic environment, the issues and techniques of distributed development: DOSE (Distributed and Outsourced Software Engineering). It involved in 2007, for the first time, a cooperative project performed in common by several universities. This was a trial run, and we are now expanding the experience; for details see

http://se.ethz.ch/dose/

Participation is open to any interested university. The goal is to let students discover and confront the challenges of distributed development in the controlled environment of a university course.

Distributed and collaborative development

Not all of our experience, as noted, may be transposable to other contexts. Our group is small, we know each other well and have been working together for quite a while; we developed these techniques together, learning from our mistakes and benefiting from the advances of technology in the past years. But whatever the reservations I believe this is the way of the future. Even more so considering that the supporting technology is still in its infancy. Two years ago most of the communication tools we use did not exist; five years ago none did. (Funny to think of all the talks I heard over the years about "Computer-Supported Cooperative Work", fascinating but remote from anything we could use. Suddenly come the Web, Voice Over IP solutions for the common folk, shared editing tools and a few other commercial offerings, and the gates 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, remember to mute your microphone!"), calls get cut off, we don't have a really good equivalent of the shared whiteboard. Other aspects of the process still need improvement; for example we have not yet found a good way to make our review results seamlessly available as part of the open-source development site, which is based on Wiki pages. All this will be corrected in the next few years. I hope that courses such as DOSE and other academic efforts will enable us to understand better what makes collaborative development succeed or fail.

But this is not just an academic issue. Eiffel Software's still recent experience of collaborative development — every meeting brings new insights — suggests that something fundamental has changed, mostly for the better, in the software development process. As regards code reviews I do not, for myself, expect ever again to get stuck for three hours in a windowless room with half a dozen other programmers poring over some boring printouts.

Acknowledgment I am grateful to the EiffelStudio development team for their creativity and team spirit, which enabled the team collectively to uncover and apply the techniques described here. Earlier versions of this article appeared as a column in the EiffelWorld newsletter and in the SEAFOOD 2008 conference.