developerWorks®

11 proven practices for more effective, efficient peer code review

Using SmartBear CodeCollaborator for lightweight code review

Jason Cohen (smartbear@smartbear.com)
Original architect of CodeCollaborator
SmartBear Software

25 January 2011

These 11 proven practices for efficient, lightweight peer code review are based on a study at Cisco Systems using SmartBear CodeCollaborator. They can help you ensure that your reviews both improve your code and make the most of your developers' time.

Our team at SmartBear Software® has spent years researching existing code review studies and collecting "lessons learned" from more than 6000 programmers at more than 100 companies. Clearly, people find bugs when they review code, but the reviews often take too long to be practical. We used the information gleaned through years of experience to create the concept of lightweight code review. By using lightweight code review techniques, developers can review code in one-fifth the time needed for full, formal code reviews. We also developed a theory for best practices to employ for optimal review efficiency and value. This article outlines those practices.

To test our conclusions about code review in general and lightweight review in particular, we conducted the largest study ever done on code review. It encompassed 2500 code reviews, 50 programmers, and 3.2 million lines of code at Cisco Systems. For 10 months, the study tracked the MeetingPlace product team, which had members in Bangalore, Budapest, and San José.

At the start of the study, we set up these rules for the group:

- All code had to be reviewed before it was checked into the team's version control software.
- SmartBear's CodeCollaborator® code review software tool would be used to expedite, organize, and facilitate all code reviews.
- In-person meetings for code review were not allowed.
- The review process would be enforced by tools.
- Metrics would be automatically collected by CodeCollaborator, which provides review-level and summary-level reporting.

The 11 best practices, according to our study

It's common sense that peer code review (in which software developers review each other's code before releasing software to QA) identifies bugs, encourages collaboration, and keeps code more maintainable.

But it's also clear that some code review techniques are inefficient and ineffective. The meetings often mandated by the review process take time and kill excitement.Strict process can stifle productivity, but lax process means no one knows whether reviews are effective or even happening. And the social ramifications of personal critique can ruin morale.

This article describes 11 best practices for efficient, lightweight peer code review that have been proven to be effective by scientific study and by SmartBear's extensive field experience. Use these techniques to ensure your code reviews improve your code – without wasting your developers' time. And use the latest technology to do code review from within the IBM® Rational Team Concert® environment.

1. Review fewer than 200–400 lines of code at a time

The Cisco code review study (see the sidebar) showed that for optimal effectiveness, developers should review fewer than 200-400 lines of code (LOC) at a time. Beyond that, the ability to find defects diminishes. At this rate, with the review spread over no more than 60–90 minutes, you should get a 70–90% yield. In other words, if 10 defects existed, you'd find 7 to 9 of them.

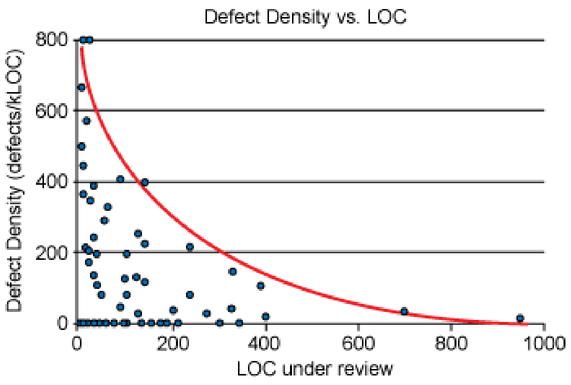
More about the Cisco study

After 10 months of monitoring, the study crystallized our theory: done properly, lightweight code reviews are just as effective as formal ones, yet they're substantially faster (and less annoying) to conduct. Our lightweight reviews took an average of 6.5 hours less than formal reviews, but found just as many bugs.
Besides confirming some theories, the study uncovered some new rules, many of which are outlined in this article.

The graph in Figure 1, which plots defect density against the number of lines of code under review, supports this rule. *Defect density* is the number of defects found per 1000 lines of code. As the number of lines of code under review grows beyond 200, defect density drops off considerably.

In this case, defect density is a measure of "review effectiveness." If two reviewers review the same code and one finds more bugs, we would consider that reviewer more effective. Figure 1 shows how, as we put more code in front of a reviewer, her effectiveness at finding defects drops. This result makes sense, because she probably doesn't have a lot of time to spend on the review, so, inevitably, she won't do the job as well on each file.

Figure 1. Defect density dramatically decreases when the number of lines of inspection goes above 200, and it is almost zero after 400



2. Aim for an inspection rate of fewer than 300–500 LOC per hour

Definitions

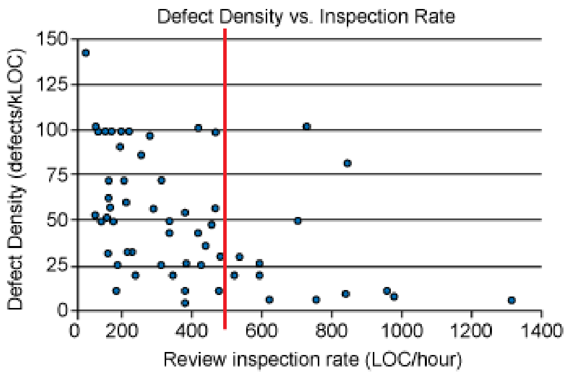
- **Inspection rate:** How fast are we able to review code? Normally measured in kLOC (thousand Lines of Code) per work hour.
- **Defect rate:** How fast are we able to find defects? Normally measured in number of defects found per work hour.
- **Defect density:** How many defects do we find in a given amount of code (not how many there are)? Normally measured in number of defects found per kLOC.

Take your time with code review. Faster is not better. Our research shows that you'll achieve optimal results at an inspection rate of less than 300–500 LOC per hour. Left to their own devices, reviewers' inspection rates will vary widely, even with similar authors, reviewers, files, and review sizes.

To find the optimal inspection rate, we compared *defect density* with *how fast* the reviewer went through the code. Again, the general result is not surprising: if you don't spend enough time on the review, you won't find many defects. If the reviewer is overwhelmed by a large quantity of code, he won't give the same attention to every line as he might with a small change. He won't be able to explore all ramifications of the change in a single sitting.

So, how fast is too fast? Figure 2 shows the answer: reviewing faster than 400 LOC per hour results in a severe drop-off in effectiveness. And at rates above 1000 LOC per hour, you can probably conclude that the reviewer isn't actually looking at the code at all.

Figure 2. Inspection effectiveness falls off when more than 500 lines of code are under review



3. Take enough time for a proper, slow review, but not more than 60–90 minutes

Never review code for more than 90 minutes at a stretch.

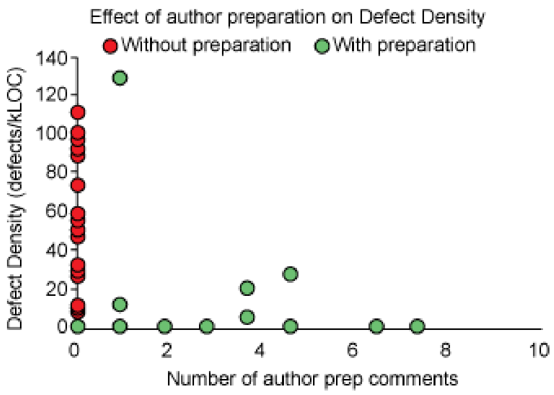
We've talked about how, for best results, you shouldn't review code too fast. But you also shouldn't review too long in one sitting. After about 60 minutes, reviewers simply get tired and stop finding additional defects. This conclusion is well-supported by evidence from many other studies besides our own. In fact, it's generally known that when people engage in any activity requiring concentrated effort, performance starts dropping off after 60–90 minutes. Given these human limitations, a reviewer will probably not be able to review more than 300–600 lines of code before performance drops.

On the flip side, you should always spend at least five minutes reviewing code, even if it's just one line. Often, a single line can have consequences throughout the system, and it's worth the five minutes to think through the possible effects that a change could have.

4. Be sure that authors annotate source code before the review begins

It occurred to us that authors might be able to eliminate most defects before a review even begins. If we required developers to double-check their work, maybe reviews could be completed faster without compromising code quality. As far as we could tell, this specific idea had not been studied before, so we tested it during the study at Cisco.

Figure 3. The striking effect of author preparation on defect density



The idea of *author preparation* is that authors annotate their source code before the review begins. We invented the term to describe a certain behavior pattern that we measured during the study, which was exhibited by authors in about 15% of the reviews. Annotations guide the reviewer through the changes, showing which files to look at first and defending the reason and methods for each code modification. These notes are not comments in the code, but rather comments given to other reviewers.

Our theory was that, because the author has to re-think and explain the changes during the annotation process, the author will uncover many of the defects before the review even begins, thus making the review itself more efficient. As such, the review process should yield a lower density of defects, because fewer bugs remain. Sure enough, reviews preceded by author preparation had barely any defects, compared to reviews without author preparation.

We also considered a pessimistic theory to explain the lower bug findings. What if, when the author makes a comment, the reviewer becomes biased or complacent so just doesn't find as many bugs? We took a random sample of 300 reviews to investigate, and the evidence showed that the reviewers were, indeed, carefully reviewing the code and there were simply fewer bugs.

5. Establish quantifiable goals for code review, and capture metrics so you can improve your processes

As with any project, decide in advance on the goals of the code review process and how you will measure its effectiveness. When you've defined specific goals, you will be able to judge whether peer review is truly achieving the results that you require.

It's best to start with external metrics, such as "reduce support calls by 20%" or "halve the percentage of defects injected by development." This information gives you a clear picture of how your code is doing from the outside perspective, and it needs to be a quantifiable measurement, not just a vague goal to "fix more bugs."

However, it can take a while before external metrics show results. Support calls, for example, won't be affected until new versions are released and in customers' hands. So it's also useful to watch internal process metrics to get an idea of how many defects are found, where your problems lie, and how long your developers are spending on reviews. The most common internal metrics for code review are *inspection rate*, *defect rate*, and *defect density*.

Consider that only automated or tightly controlled processes can give you repeatable metrics; humans aren't good at remembering to stop and start stopwatches. For best results, use a code review tool that gathers metrics *automatically* so that your critical metrics for process improvement are accurate.

To improve and refine your processes, collect your metrics and tweak your processes to see how changes affect your results. Pretty soon, you'll know exactly what works best for your team.

6. Use checklists, because they substantially improve results for both authors and reviewers

Checklists are especially important for reviewers because, if the author forgets a task, the reviewer is likely to miss it also.

Checklists are a highly recommended way to check for the things that you might forget to do, and they are useful for both authors and reviewers. Omissions are the hardest defects to find; after all, it's hard to review something that's not there. A checklist is the single best way to combat the problem, because it reminds the reviewer or author to take the time to look for something that might be missing. A checklist will remind authors and reviewers to confirm that all errors are handled, that function arguments are tested for invalid values, and that unit tests have been created.

Another useful concept is the *personal checklist*. Each person typically makes the same 15–20 mistakes. If you notice what your typical errors are, you can develop your own personal checklist (Personal Software Process, the Software Engineering Institute, and the Capability Maturity Model Integrated recommend this practice, too). Reviewers will do the work of determining your common mistakes. All you have to do is keep a short checklist of the common flaws in your work, particularly the things that you most often forget to do.

As soon as you start recording your defects in a checklist, you will start making fewer of these errors. The rules will be fresh in your mind, and your error rate will drop. We've seen this happen over and over.

Tip:
For more detailed information on checklists, plus a sample checklist, you can get a free copy of the book by the author of this article, *Best Kept Secrets of Peer Code Review*, at www.CodeReviewBook.com.

7. Verify that the defects are actually fixed

OK, this "best practice" seems like a no-brainer. If you're going to all of the trouble of reviewing code to find bugs, it certainly makes sense to fix them! Yet many teams that review code don't have a good way of tracking defects found during review and ensuring that bugs are actually fixed before the review is complete. It's especially difficult to verify results in email or over-the-shoulder reviews.

Keep in mind that these bugs aren't usually entered in Rational Team Concert logs, because they are found before code is released to QA. So, what's a good way to ensure that defects are fixed before the code is given the "All Clear" sign? We suggest using good collaborative review software, integrated with Rational Team Concert, to track defects found in reviews. With the right tool, reviewers can log bugs and discuss them with the author as necessary. Authors then fix the problems and notify reviewers, and reviewers must then verify that each issue is resolved. The tool should track bugs found during a review and prohibit review completion until all bugs are verified as *fixed* by the reviewer (or tracked as a separate work item to be resolved later). A work item should be approved only when the review is complete.

> **If you are going to go to the trouble of finding the bugs, make sure that you've fixed all of them!**

Now that you've learned these best practices for the *process* of code review, we'll discuss some social effects and how you can manage them for best results.

8. Foster a good code review culture in which finding defects is viewed positively

Code review can do more for true team building than almost any other technique we've seen, but only if managers promote it at a means for learning, growing, and communication. It's easy to see defects as a bad thing (after all, they are mistakes in the code), but fostering a negative attitude toward defects found can sour a whole team, not to mention sabotage the bug-finding process.

> **The point of software code review is to eliminate as many defects as possible, regardless of who "caused" the error.**

Managers must promote the viewpoint that defects are positive. After all, each one is an opportunity to improve the code, and the goal of the bug review process is to make the code as good as possible. Every defect found and fixed in peer review is a defect that a customer never sees and another problem that QA doesn't have to spend time tracking down.

Teams need to maintain the attitude that finding defects means that the author and reviewer have successfully worked *as a team* to improve the product. It's not a case of "the author created a defect and the reviewer found it." It's more like a very efficient form of pair-programming.

Reviews present opportunities for all developers to correct bad habits, learn new tricks, and expand their capabilities. Developers can learn from their mistakes, but only if they know what their issues are. And if developers are afraid of the review process, the positive results disappear.

Especially if you're a junior developer or are new to a team, defects found by others are a good sign that your more experienced peers are doing a good job in helping you become a better developer. You'll progress far faster than if you were programming in a vacuum, without detailed feedback.

To maintain a consistent message that finding bugs is good, management must promise that defect densities will never be used in performance reports. It's effective to make these kinds of promises in the open. Then developers know what to expect and can call out any manager that violates a rule made so publicly.

Managers should also never use buggy code as a basis for negative performance reviews. They must tread carefully and be sensitive to hurt feelings and negative responses to criticism and continue to remind the team that finding defects is good.

9. Beware of the Big Brother effect

As a developer, you automatically assume that it's true that "Big Brother is watching you," especially if your review metrics are measured automatically by review-supporting tools. Did you take too long to review some changes? Are your peers finding too many bugs in your code? How will this affect your next performance evaluation?

> **Metrics should never be used to single out developers, particularly in front of their peers. This practice can seriously damage morale.**

Metrics are vital for process measurement, which, in turn, provides the basis for process improvement. But metrics can be used for good or evil. If developers believe that metrics will be used against them, not only will they be hostile to the process, but they will probably focus on improving their metrics rather than truly writing better code and being more productive.

Managers can do a lot to improve the problem. First and foremost, they must be aware of it and keep watching to make sure that they're not propagating the impression that Big Brother is indeed scrutinizing every move.

Metrics should be used to measure the efficiency of the process or the effect of a process change. Remember that, often, the most difficult code is handled by your most experienced developers. This code, in turn, is more likely to be more prone to error, thus to be reviewed heavily, with more defects found. Therefore, large numbers of defects are often more attributable to the complexity and risk of a piece of code than to the author's abilities.

If metrics do help a manager uncover an issue, singling someone out is likely to cause more problems than it solves. We recommend that managers deal with any issues by addressing the group as a whole, instead. It's best not to call a special meeting for this purpose, because developers might feel uneasy if it looks like there's a problem. Instead, just roll it into a weekly status meeting or other normal procedure.

Managers must continue to foster the idea that finding defects is good, not bad, and that defect density is not correlated with a developer's ability. Remember to make sure that it's clear to the team that defects, particularly the number of defects introduced by a team member, shouldn't be shunned and will never be used for performance evaluations.

10. Review at least part of the code, even if you can't do all of it, to benefit from The Ego Effect

Imagine yourself sitting in front of a compiler, tasked with fixing a small bug. But you know that as soon as you say "I'm finished," your peers — or worse, your boss — will be examining your work. Won't this change your development style? As you work, and certainly before you declare code finished, you'll be a little more conscientious. You'll be a better developer immediately because you want the general timbre of the "behind your back" conversations to be, "His stuff is pretty tight. He's a good developer;" not "He makes a lot of silly mistakes. When he says he's done, he's not."

The Ego Effect drives developers to write better code because they know that others will be looking at their code and their metrics. No one wants to be known as the guy who makes all those junior-level mistakes. The Ego Effect drives developers to review their own work carefully before passing it on to others.

A nice characteristic of The Ego Effect is that it works equally well whether reviews are mandatory for all code changes or used just as "spot checks," like a random drug test. If your code has a 1 in 3 chance of being called out for review, that's still enough of an incentive to make you do a great job. However, spot checks must be frequent enough to maintain the Ego Effect. If you had just a 1 in 10 chance of getting reviewed, you might not be as diligent. You know you can always say, "Yeah, I don't usually make that mistake."

Reviewing 20–33% of the code will probably give you maximal Ego Effect benefit with minimal time expenditure and reviewing 20% of your code is certainly better than reviewing none.

11. Adopt lightweight, tool-assisted code reviews

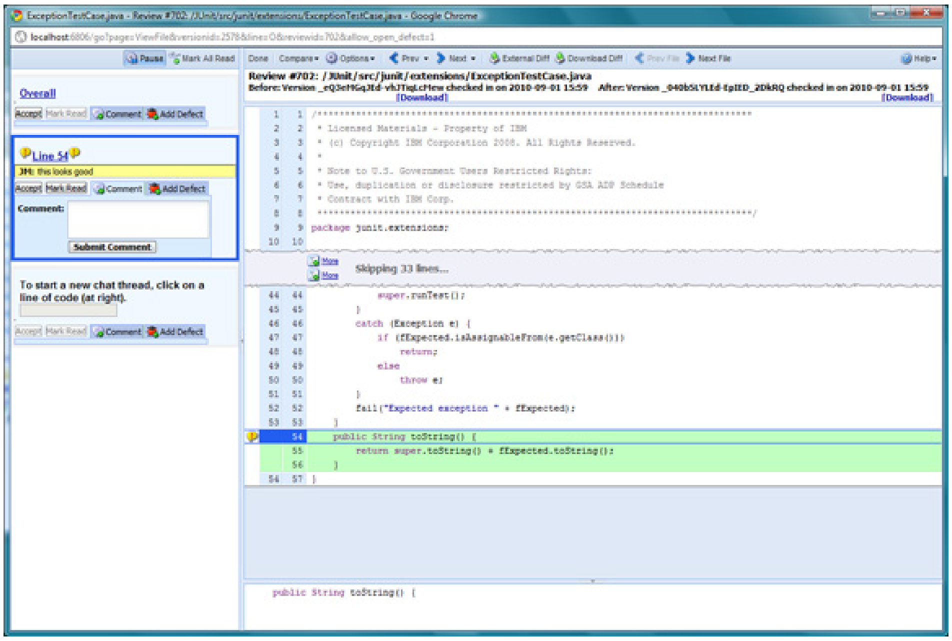
There are several main types and countless variations of code review, and these guidelines will work with any of them. However, to fully optimize the time that your team spends in review, we got optimum results with a tool-assisted lightweight review process. It's efficient, practical, and effective at finding bugs.

Formal, or *heavyweight*, inspections have been around for 30 years. They are no longer the most efficient way to review code. The average heavyweight inspection takes nine hours per 200 lines of code. Although it's effective, this rigid process requires three to six participants and hours of painful meetings paging through code printouts in exquisite detail. Unfortunately, most organizations cannot afford to tie up people for that long, and most programmers despise the tedious process required. In recent years, many development organizations have shrugged off the yoke of meeting schedules, paper-based code readings, and tedious metrics-gathering in favor of new *lightweight* processes that eschew formal meetings and lack the overhead of the older, heavyweight processes.

We used our case study at Cisco to determine how the lightweight techniques compare to the formal processes. The results showed that lightweight code reviews take one-fifth the time (or less) of formal reviews, yet they find just as many bugs.

Although several methods exist for lightweight code review, such as over-the-shoulder reviews and reviews by email, the most effective reviews are conducted using a collaborative software tool to facilitate the review, such as SmartBear's CodeCollaborator (see Figure 4).

Figure 4. CodeCollaborator, the lightweight code review tool used in the Cisco study



Review #702: /JUnit/src/junit/extensions/ExceptionTestCase.java
Before: Version _eQ3eHGqJtd-vhJTlqLcHew checked in on 2010-09-01 15:59 After: Version _040b5LYLEd-EpIED_2DkRQ checked in on 2010-09-01 15:59

JM: this looks good

To start a new chat thread, click on a line of code (at right).

```
/*******************************************************************************
 * Licensed Materials - Property of IBM
 * (c) Copyright IBM Corporation 2008. All Rights Reserved.
 *
 * Note to U.S. Government Users Restricted Rights:
 * Use, duplication or disclosure restricted by GSA ADP Schedule
 * Contract with IBM Corp.
 *******************************************************************************/
package junit.extensions;

            super.runTest();
        }
        catch (Exception e) {
            if (fExpected.isAssignableFrom(e.getClass()))
                return;
            else
                throw e;
        }
        fail("Expected exception " + fExpected);
    }
    public String toString() {
        return super.toString() + fExpected.toString();
    }
}
```

```
public String toString() {
```

CodeCollaborator is the only code review tool that integrates with IBM® Rational Team Concert workflows. It integrates source code viewing with chat-style collaboration to free the developer from the tedium of associating comments with individual lines of code. When programmers add change sets to a work item for review, the review gets automatically created in CodeCollaborator with the appropriate approvers assigned. Team members can comment directly on the code, chat with the author and each other to work through issues, and track bugs and verify fixes. No meetings, printouts, stopwatches, or scheduling required.

With a lightweight review process based on Rational Team Concert and CodeCollaborator, your team can conduct more efficient reviews and fully realize the substantial benefits of code review.

So now you're armed with an arsenal of proven practices to ensure that you get the most of out of the time that your team spends on code reviews, both from a process and a social perspective. Of course, you must actually *do* code reviews to realize the benefits. Formal methods of review are simply impractical to implement for 100% of your code (or any percent, some would argue). Tool-assisted, lightweight code review integrated into the Rational Team Concert environment provides the most "bang for the buck," because it offers an efficient and effective method to locate defects without requiring painstaking tasks that developers hate to do. With the right tools and these practices, your team can peer-review all of your code and find costly bugs before your software reaches even QA stage, so that your customers get top-quality products every time.

For your convenience, here are the 11 practices in a simple list that's easy to keep on file:

1. Review fewer than 200–400 lines of code at a time.
2. Aim for an inspection rate of fewer than 300–500 LOC per hour.
3. Take enough time for a proper, slow review, but not more than 60–90 minutes.

4. Be sure that authors annotate source code before the review begins.
5. Establish quantifiable goals for code review and capture metrics so you can improve your processes.
6. Use checklists, because they substantially improve results for both authors and reviewers.
7. Verify that the defects are actually fixed.
8. Foster a good code review culture in which finding defects is viewed positively.
9. Beware of the Big Brother effect.
10. Review at least part of the code, even if you can't do all of it, to benefit from The Ego Effect.
11. Adopt lightweight, tool-assisted code reviews.

CodeCollaborator has received "Ready for IBM Rational Software" validation for Rational Team Concert Versions 2 and 3, as well as for IBM® Rational® ClearCase® and IBM® Rational® Synergy® software.

Resources

Learn

- More information on these best practices, the case study, and other topics are chronicled in Jason Cohen's book, *Best Kept Secrets for Peer Code Review*, which is available free at www.CodeReviewBook.com.
- Find out more about CodeCollaborator, SmartBear Software's code review tool, including more about CodeCollaborator integration with IBM Rational Team Concert, and watch the four-minute video demo, Code review within IBM Rational Team Concert.
- Get details on IBM Rational Team Concert:
  - **Documentation:**Rational Team Concert v2: Getting started and the Rational Team Concert Information Center
  - **Articles and other resources:**IBM developerWorks page for Rational Team Concert
  - **Webcast:** Using Rational Team Concert in a globally distributed team
  - **Demo:**Dashboards and reports
  - **Podcast:**IBM Rational Team Concert and Jazz
- Visit the Rational software area on developerWorks for technical resources and best practices for Rational Software Delivery Platform products.
- Stay current with developerWorks technical events and webcasts focused on a variety of IBM products and IT industry topics.
- Attend a free developerWorks Live! briefing to get up-to-speed quickly on IBM products and tools, as well as IT industry trends.
- Follow developerWorks on Twitter.
- Watch developerWorks on-demand demos, ranging from product installation and setup demos for beginners to advanced functionality for experienced developers.
- Check the Rational training and certification catalog, which includes many types of courses on a wide range of topics. You can take some of them anywhere, any time, and many of the "Getting Started" ones are free.

Get products and technologies

- Rational Team Concert trial downloads (free):
  - Enterprise Edition
  - Express Edition
  - Standard Edition
- Try building and deploying your next project on the IBM Bluemix cloud platform, where you can take advantage of pre-built services, runtimes, frameworks, application lifecycle management, and continuous integration.
- Evaluate IBM software in the way that suits you best: Download it for a trial, try it online, use it in a cloud environment, or spend a few hours in the SOA Sandbox learning how to implement service-oriented architecture efficiently.

Discuss

- Join the Rational Team Concert discussions or ask questions in the Jazz.net forums.

- Get involved in the developerWorks community. Connect with other developerWorks users while exploring the developer-driven blogs, forums, groups such as the Rational Café and Rational wikis.

About the author

Jason Cohen

Jason Cohen is the original architect of CodeCollaborator and founder of SmartBear Software. After the acquisition of his company, he continues to be involved in a number of strategic initiatives of the new SmartBear Software and frequently speaks about the benefits of peer code review. He is also the founder of three other companies, including WPEngine and ITWatchDogs. For more information about SmartBear Software, visit www.smartbear.com.