

Code Collaborator

Screen-by-screen walk-through for the most popular peer code review tool on the market.

“Although many new software design techniques have emerged in the past 15 years, there have been few changes to the procedures for reviewing the designs produced using these techniques.” This observation was made in 1985¹. Have we learned nothing since Michael Fagan’s seminal 1976 paper? Don’t changes in development techniques, languages, personnel, and remote sites require us to revisit review practices? Haven’t we developed any technology in the past 30 years that might remove drudgery and waste while preserving the proven benefits of code review?

¹ Parnas, D. L. and Weiss, D. M. 1985. Active design reviews: principles and practices. In *Proceedings of the 8th international Conference on Software Engineering* (London, England, August 28 - 30, 1985). International Conference on Software Engineering. IEEE Computer Society Press, Los Alamitos, CA, 132-136.

Code Collaborator™ enables peer review by enforcing workflows, integrating with incumbent development tools, and automating audit trails, metrics, and reporting. It is the culmination of many years of contributions from hundreds of software organizations, all of whom use the tool to improve their code review process.

Code Collaborator replaces traditional formal inspections and over-the-shoulder or email-pass-around reviews with a system that enforces inspection rules, handles diverse workflows, automates audit trails and metrics-gathering, generates reports, and saves developers time through 3rd-party tool integration. Users who are at their keyboards simultaneously can review code with a chat-like interface; users separated by many time zones interact with a newsgroup-like interface. Code Collaborator integrates with SCM systems, bug/issue trackers, and custom scripts through tool plugins. Reviewers can verify that fixes are properly made by developers before allowing code to be checked into version control.

Solving common peer review problems

There are five specific problems with typical peer code review that Code Collaborator addresses².

1. Developers waste time packaging and delivering source code for inspection.

No matter what type of code review you wish to implement, developers will bear the burden of packaging files for delivery. For example, say a developer is preparing for a review of changes that he has not yet checked into version control. After determining which files were added, removed, and modified, he needs to copy them somewhere for transport. He'll also need to extract the previous versions of those files from version control; otherwise the

² Other forms of peer code review have most if not all of these problems. Please see our essay, "Five Types of Review," for a detailed treatment of the pros and cons of many classes of review and inspection.

reviewers will not know what they're supposed to be looking at. (Imagine a change to five lines in the middle of a 2000-line file!) These previous versions must be organized such that reviewers can easily see which current file goes with which previous and can easily run some sort of file-difference program to assist their viewing. And it's easy for the developer to accidentally miss a file he changed.

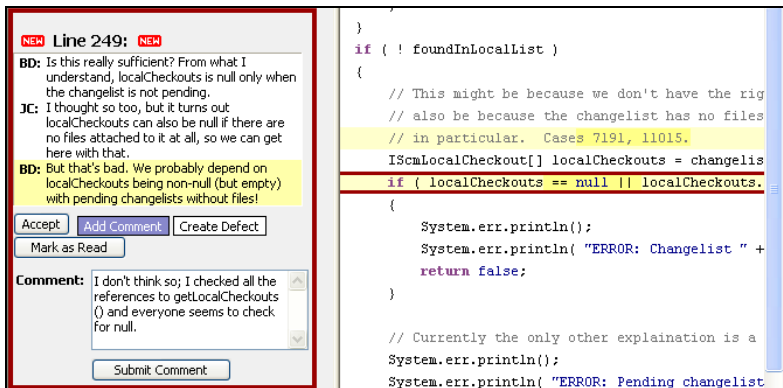


Figure 31: Code Collaborator screenshot showing threaded comments next to Java code under inspection. The author is defending a design decision.

With Code Collaborator, client-side version control integration removes this burden completely. With a single mouse-click or a command-line utility, developers can upload sets of files for review. This can be changes not yet checked into version control as in our example above, or this can be any sets of already-

checked-in changes such as differences between branches, labels, dates, or atomic check-in ID's³.

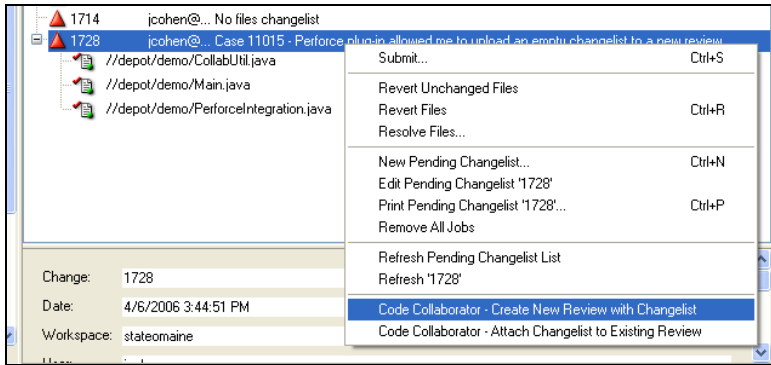


Figure 32: Integration with version control lets Code Collaborator package and deliver source code for review with a single mouse-click. Pictured here is integration with the Perforce® visual client P4V.

Besides saving time on the developer's side, Code Collaborator also removes the burden on reviewers by displaying file differences and allowing reviewers to efficiently comment on individual lines of code without having to write down file names and line numbers.

³ Not all version control systems have the concept of an atomic check-in ID. This is a single identifier that represents a set of files checked in at one time by a single author. In Perforce® this is called a “changelist;” in Source Integrity® it is a “change package;” and in Subversion this is a “revision” number associated with a tree snapshot.

2. Don't know if reviews are really happening.

Are developers really reviewing code? Are they perfunctory? With traditional review processes it's impossible to know whether all source code changes have been reviewed.

Code Collaborator provides two mechanisms for enforcing the review process. Review coverage reports allow teams to see how much they've reviewed; with that information they can either fill in the gap or make an informed decision to not review those portions. Or, through server-side version control triggers, Code Collaborator can strictly enforce rules like "All changes made to this branch must be reviewed."




JC	BD	File Relative to: //depot/demo/	Line Number	# of Lines Added (+) Changed (*) Removed (-)
		CollabUtil.java	[Overall]	+0 *2 -0
			247	
			249	
		Main.java	[Overall]	+14 *0 -12
		PerforceIntegration.java	[Overall]	+8 *0 -0

Figure 33: The file summary list makes it easy for reviewers to see which files were changed, examine prior conversations, and mark defects "fixed" only when verified. New defects can be opened if necessary.

3. Reviewers not verifying that developers are actually fixing defects.

It's one thing to send an e-mail saying that something needs to be fixed. It's another to verify that those defects have actually been fixed and that no new defects have been opened in the process. Typical code review processes leave this step to the author, but if no one knows whether reviewers are verifying fixes, the value of pointing out those defects is greatly diminished.

Code Collaborator maintains a defect log for each review, making it easy for an author to see what needs fixing and easy for a reviewer to verify once the author re-submits the fixes. Comments and defects are kept threaded like a newsgroup so it's easy for all parties to refer back to previous conversations.

4. No metrics for process measurement or feedback for improvement.

Developers hate coming up with metrics. No one wants to do a code review holding a stopwatch and wielding line-counting tools. And our field experience is that developers often make up numbers because they forget to start (or stop!) the stopwatch and they tend to invent numbers that they believe make them look good.

Only an automated process can give you repeatable metrics. Code Collaborator collects metrics quietly and automatically so developers aren't burdened. Data include kLOC/hour, defects/hour, and defects/kLOC.

5. Review process doesn't work with remote and/or time-delayed reviews.

Most review processes cannot handle one of the participants being twelve time zones away or even away from the keyboard for a few hours.

Formal inspections, over-the-shoulder reviews, and other in-person reviews don't work unless everyone is in the same place.

E-mail reviews can send data around, but the plethora of conversations and separate reviews can quickly bury a reviewer in e-mail.

Code Collaborator keeps conversations and defects threaded and attached to individual lines of code so it's easy to juggle multiple reviews. It acts like instant messaging when all participants are at the keyboard simultaneously but also like message boards for time-delayed communication.

Anatomy of a review

Most customers follow the standard review workflow described here; system administrators can change this to fit more exactly a desired review process.

Phase 1: Planning

The review begins in the “Planning” phase where the author uploads files for the review and invites the other participants.

Files are either not yet checked in to version control (i.e. review before check-in) or files that have already been checked in (to review after check-in or to review a set of branch changes). SCM integration plug-ins provided by Smart Bear make this typically a one-click or one-command process.

However, any set of files can be uploaded, whether under version control or not. Design and requirements documents can be uploaded, as can arbitrary file-differences generated by comparing directories or from other development tools. For example, you can review the set of changes on a certain branch, or all changes between two dates or labels, or all changes since the last branch integration, just by generating those differences using your version control system and piping them into the Code Collaborator command-line.

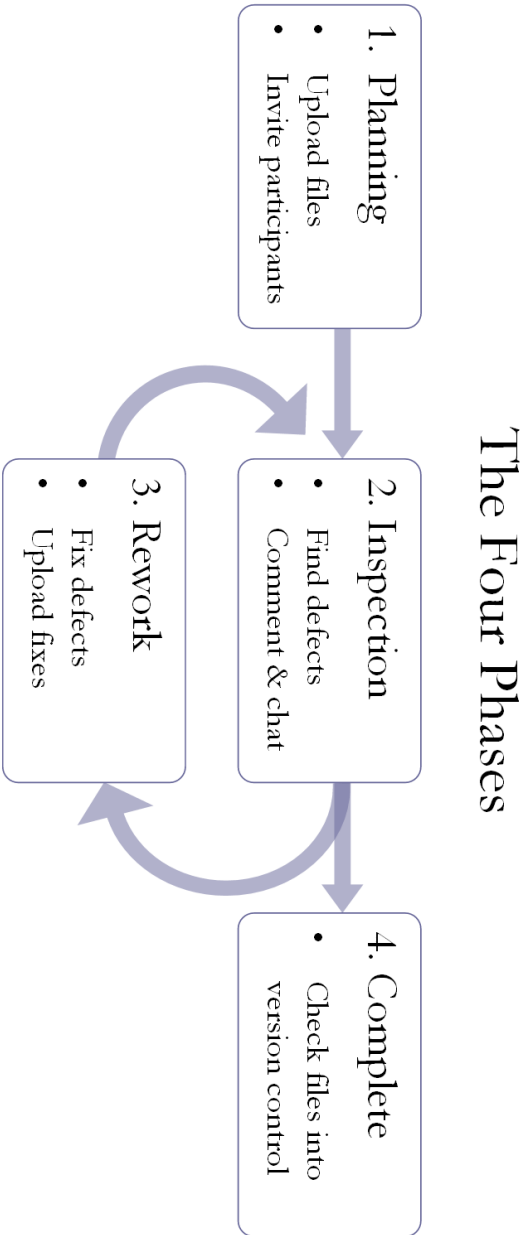


Figure 34: Lifecycle of a review.

There may be other participants, including one or more reviewers and zero or more observers. The former are responsible for careful review, and subsequent review activity will depend on their consensus; observers are invited and can participate but their consensus is never required.

The "Planning" phase is complete when the author decides the correct people are invited and all necessary files are uploaded. The review then enters the "Inspection" phase.

Phase 2: Inspection

When the review switches to "Inspection," e-mails are sent to all participants to alert them the review is starting. For those running the Windows GUI, the taskbar icon will change to indicate that a review is now in progress.

Reviewers are presented with the uploaded files with a side-by-side before/after highlighted difference view (if the files were under version control).

Several options are available for the difference view including: ignore white-space, skip unchanged lines, and whether or not to word-wrap long lines of code. Many common source code files are displayed with syntax-coloring just like an IDE.

Anyone can begin a conversation by clicking on a line of code and typing⁴. Any number of conversations can be going on at once; the associated line of code and an overall review summary view keep the conversation threads distinct and manageable.

⁴ An important attribute of any developer tool is conservation of movement. It was an important design consideration that starting or responding to a conversation must be as simple as: 1. Click the line of code or conversation, 2. Begin typing. No clicking around or tabbing between GUI elements. If operations are not fast and easy (and optionally keyboard-centric), developers will not use them as much.

```

if ( ! foundInLocalList )
{
  // This might be because we don't have the ri
  // also be because the changelist has no file
  // in particular. Cases 7191, 11015.
  IScmLocalCheckout[] localCheckouts = changelis
  if { localCheckouts == null || localCheckouts
  {
    System.err.println();
    System.err.println( "ERROR: Changelist "
    return false;
  }

  // Currently the only other explanation is a
  System.err.println();
  System.err.println( "ERROR: Pending changelis
  System.err.println( "      Current: " + scmf
-----
ES...
* static boolean haveInitiedBrowser = false;

```

```

if ( ! foundInLocalList )
{
  // This might be because we don't have the rig
  // also be because the changelist has no files
  // in particular. Case 7191.
  IScmLocalCheckout[] localCheckouts = changelis
  if { localCheckouts != null && localCheckouts.
  {
    System.err.println();
    System.err.println( "ERROR: Changelist " +
    return false;
  }

  // Currently the only other explanation is a
  System.err.println();
  System.err.println( "ERROR: Pending changelist
  System.err.println( "      Current: " + scmfCl
-----
ES...
* static boolean haveInitiedBrowser = false;

```

Figure 35: The side-by-side view displays the same file before and after the proposed changes.

Differences are highlighted. Sub-line modifications are highlighted specially. Source code syntax is color-coded like an IDE.

Choose between word-wrapped or ganged-scrollbar views. Options include ignore changes in white-space and skip unchanged lines.

Comments work a bit like instant message chat and a bit like newsgroups. If everyone is at the keyboard simultaneously, you have a real-time “instant message” environment so the review can progress swiftly. If one or more participants are separated by many time zones or just aren't currently at the computer, the chat looks like a newsgroup where you post comments and receive e-mails when someone responds.

All this implies that Code Collaborator works equally well no matter where your developers or reviewers are located. In fact, both methods work at the same time inside a single review! This

means everyone can participate in the manner in which he is most comfortable, and the system will adapt accordingly.

The screenshot shows a code editor interface with a chat window on the left and code on the right. The chat window contains a conversation about a code issue. The chat text is highlighted in yellow, and the corresponding code lines are highlighted in yellow. The chat text includes:

NEW Line 249; NEW

BD: Is this really sufficient? From what I understand, localCheckouts is null only when the changelist is not pending.

JC: I thought so too, but it turns out localCheckouts can also be null if there are no files attached to it at all, so we can get here with that.

BD: But that's bad. We probably depend on localCheckouts being non-null (but empty) with pending changelists without files!

Buttons: Accept, Add Comment, Create Defect, Mark as Read

Comment: I don't think so; I checked all the references to getLocalCheckouts () and everyone seems to check for null.

Submit Comment

The code on the right is:

```

}
}
if ( ! foundInLocalList )
{
    // This might be because we don't have the rig
    // also be because the changelist has no files
    // in particular. Cases 7191, 11015.
    IScmLocalCheckout[] localCheckouts = changelist
    if ( localCheckouts == null || localCheckouts
    {
        System.err.println();
        System.err.println( "ERROR: Changelist " +
        return false;
    }

    // Currently the only other explanation is a
    System.err.println();
    System.err.println( "ERROR: Pending changelist

```

Figure 36: Conversations are threaded by each line of code. To start or join a conversation, just click a line of code and start typing.

The highlighted chat text indicates that the viewer hasn't read that comment yet. By responding or clicking "Mark as Read" he can clear the highlight.

Reviewers open a “defect” for every change the developer will need to make before the review can be deemed complete⁵. Like comments, defects are associated with files and lines of code and show up in the conversations (although you can also create a defect for the review as a whole).

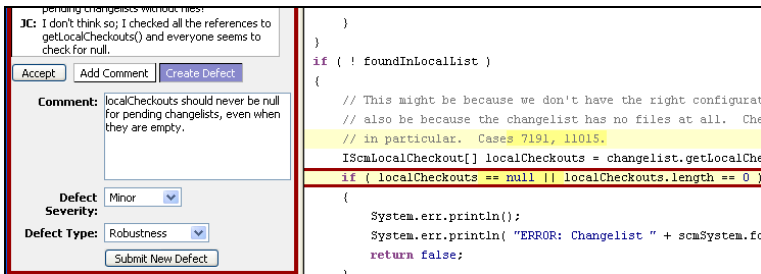


Figure 37: Opening a defect is similar to conversations and is also associated with individual lines of code.

Defects are logged and summarized for the entire review and are also available as part of the metrics and reporting system.

⁵ Some people object to the word “defect.” Some review philosophies insist that words that carry negative connotations should not be used during review. However this is the word used most commonly in the literature and was understood readily by our test subjects in the field.

Are these defects the same as issues from an external issue-tracker used by QA and front-line support? Should Code Collaborator defects be mirrored into the external issue-tracker? Although it is possible to map the two systems, almost none of our customers do. Typically an issue logged into the external system is verified by QA and/or front-level support; however Code Collaborator defects by definition never made it out of development. Often it's not clear what exactly a QA associate would do with a defect. For example, how would he test a defect like "This method needs better documentation" or even something like a null-pointer exception? The exception here is with defects identified in code review but for which you've decided the fix should be deferred. In this case, you need to be able to transform the Collaborator defect into an external issue so you can track it along with other known issues. Collaborator has a facility for this specific use-case.


The "Inspection" phase is complete when all reviewers say it's complete. If defects were opened, the review proceeds to the "Rework" phase; otherwise the review moves to the "Complete" phase.

Phase 3: Rework

In the "Rework" phase the author is responsible for fixing the defects found in the "Inspection" phase.

This might be as simple as fixing a typo in a comment, or as complex as a complete rework of the task involving different files than in the original review.

Defect Log

	ID	Severity	Type	Problem Description
	D22	Minor	Robustness	localCheckouts should never be null for pending changelists, even when they are empty. //depot/demo/CollabUtil.java

--Select--
--Select--

Review Materials

Perforce Changelists

@1728 by jcohen on 2006-04-06 15:58

Case 11015 - Perforce plug-in allowed me to upload an empty changelist to a new review.




	JC	BD	File Relative to:	Line Number	# of Lines Added (+) Changed (*) Removed (-)
			CollabUtil.java	[Overall] 247 249	+0 *2 -0
			Main.java	[Overall]	+14 *0 -12
			PerforceIntegration.java	[Overall]	+8 *0 -0

Figure 38: The defect-log and file-summary views guide the developer during the "Rework" phase and reviewers during the subsequent follow-up verification inspection phase.

When the author believes all defects are fixed, she uploads the fixed files to the server and thereby causes the review to re-enter the "Inspection" phase so reviewers can verify the fixes and ensure no new defects have been opened in the process.

If the author (or any other participant) needs to ask a question or otherwise re-open the review, she can do that prior to uploading fixes. The review will also re-enter the "Inspection" phase.

<p>JC: I don't think so; I checked all the references to <code>getLocalCheckouts()</code> and everyone seems to check for null.</p> <p>BD: Created Defect D22: <code>localCheckouts</code> should never be null for pending changelists, even when they are empty.</p> <p>BD: Marked defect fixed: D22</p>	<pre> } } if (! foundInLocalList) { // This might be because we // also be because the chang // in particular. Cases 719 IScmLocalCheckout[] localChe if { localCheckouts == null { System.err.println(); } } } </pre>
<p>D22 Minor / Robustness-</p> <p>[Mark Open] Added 2006-04-06 17:43 by BD:</p> <p>[Edit] <code>localCheckouts</code> should never be null for</p> <p>[Delete] pending changelists, even when they are empty.</p> <p><input type="button" value="Accept"/> <input type="button" value="Add Comment"/> <input type="button" value="Create Defect"/></p>	

Figure 39: Reviewers examine fixes using the same side-by-side view as the original code inspection.

If the fix is acceptable, the reviewer marks the defect “fixed” by clicking the link next to the defect log located under the conversation history. Displayed here is a defect that was just marked “fixed.”

The review is not complete until all defects are marked “fixed” in this manner. New defects can be opened if necessary.

This “round two” of the “Inspection” phase is similar to the first in that conversations can continue, new conversations can start, new defects can be opened, and so on. But this time the reviewers are verifying fixes rather than inspecting fresh code⁶.

To “verify,” reviewers examine the new changes. If the fix is complete, the reviewer clicks a link next to the defect listing to

⁶ It is possible for reviewers to cause the review to enter “Rework” before all files have been inspected. This is typical when defects are found that indicate many files will have to be changed, especially with architectural reworks. In this case, “round two” might include much more new-code inspection. Code Collaborator supports this workflow automatically.

indicate that the defect is “fixed.” Reviewers can also edit or even delete defects if necessary⁷.

This cycle of inspect-rework-inspect is finished when all defects are marked fixed and all reviewers indicate they are finished with the review. At this point the review is officially complete.

Phase 4: Complete

When the review is complete, that's it!

If the author had uploaded some version control changes that weren't yet checked in, the author is sent an e-mail reminding her that the changes may now be committed.

Metrics: Collection, and what to do with it

Developers hate collecting metrics, and they're not good at it. Code Collaborator collects metrics automatically and invisibly, giving product managers the data they need for process improvement.

The three most important metrics collected are: inspection rate, defect rate, and defect density.

The first two are “rates.” That is, they answer the question “How long does it take us to do this?” For kLOC⁸/man-hour, often called the “inspection rate,” how much code are we able to review per hour of work? For defects/man-hour, often called the “defect rate,” how many defects do we find when we spend an hour looking at source code?

⁷ When should a defect be deleted instead of marked fixed? Deleted means there was never a defect at all – the reviewer was at fault for marking it so. Fixed means there was a problem, but now it's fixed. The distinction is important for metrics, reports, and general communication during the review.

⁸ The use of kLOC (thousand lines of code) has been standard practice with metrics analysis for thirty years. It typically refers to physical lines in a file (regardless of white-space, comments, etc.), but some groups ignore white-space and still others ignore comments as well. Typically in code review you need to include comments because you are reviewing comments as well as code.

The third – defects/kLOC – is often called “defect density” in the literature. This indicates the quantity of defects found in a given amount of code.

The most natural question is: What are “good” values for these metrics? And the follow-up: If our metrics are “bad,” what can we do to correct the problem?

The answers to these questions are complicated. This subject is taken up in several other essays in this book. See specifically the two essays on case studies (metrics in the real world), the essay on metrics and measurement, and the essay on questions for a review process.

Moving parts

Code Collaborator follows the normal client/server pattern of enterprise software. See Figure 40.

The Server

As with most enterprise-class software systems, a server process acts as the hub, manager, and controller of information. The server has a web-based user interface where users and administrators can do everything – create and perform reviews, configure personal and system-wide settings and run reports.

The server uses an external database to store all data and configuration. This database can be shared across multiple instances of the server for load-balancing or real-time fail-over.

Besides the web-based user interface, the server also hosts a Web Services server. This server is integrated into the same web server as the web-based user interface, so no additional configuration is necessary. You can use Web Services to integrate Code Collaborator into any external systems (SCM server triggers, issue-tracking systems, reporting scripts, intranet portals, data-mining tools, etc.).

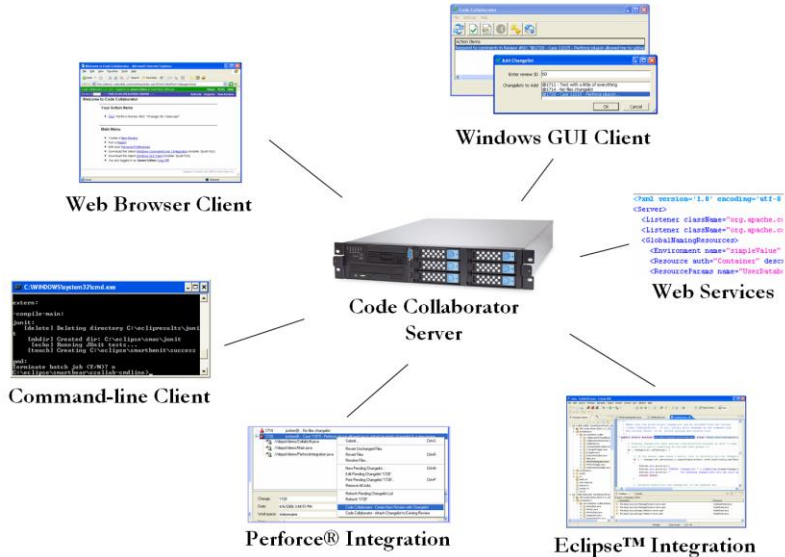


Figure 40: The parts that make up the Code Collaborator system. A server provides central services; many options are available for accessing and manipulating the server data or integrating with 3rd-party tools.

Other Code Collaborator software (such as the command-line and GUI clients) also uses this Web Services interface. We even supply Java- and .NET-based libraries that provide an object-oriented interface to this system.

Command-line Client

Developers will typically install the cross-platform command-line client. This tool lets you upload local files (and file-changes) into new and existing reviews.

Report-generators will also install the command-line client. Although all information in the system can be retrieved through

Web Services, we also support a rich set of reports in a variety of data formats (HTML, XML, CSV) with appropriate filtering options. You can access these reports through the command-line (as well as through the web-based user interface).

Windows GUI Client

Windows users have the option of installing a graphical client to complement the web-based user interface already provided by the server.

The Windows GUI Client provides all the functionality of the command-line client, but in a graphical interface.

In addition, the Windows GUI Client gives you a taskbar icon that updates to show you whether you have any pending tasks in Code Collaborator.

Perforce® Integration

Perforce users will probably want to install the Perforce Client Integration tools. These are included in the command-line installer.

Integration with P4V and P4Win lets developers upload changelists into new or existing reviews just by right-clicking on the changelist. This works on both "pending" and "submitted" changelists.

We also supply a special tool for use as a Perforce server trigger. For example, you can use this to enforce a rule like "Every commit on this branch requires a review." You can also use this to automatically upload all submitted changelists into Code Collaborator so that you can review code *after* it has been checked in. This can be especially useful with off-shore development groups.

Eclipse™ Integration

Eclipse plug-in integration allows users of Eclipse-based development tools to perform code reviews without leaving their IDE. Code Collaborator provides both inside-Eclipse integration

as well as a general-purpose cross-platform GUI client that doesn't depend on Eclipse.