GERRIT GUIDE   
(2016/07)

This guide expected that readers know about [GIT](http://git-scm.com/) and that they are familiar with basic GIT commands and workflows.

History

|  |  |  |
| --- | --- | --- |
| **Date** | **Author** | **Description** |
| 2015/06 | Manu | Create first version |
| 2016/07 | Khu Vu | Some changes |
|  |  |  |

Contents

[I. Quick introduction 4](#_Toc457396237)

[1. Overview 4](#_Toc457396238)

[2. Where does Gerrit fit in 4](#_Toc457396239)

[II. Development Guide 5](#_Toc457396240)

[1. Tool 5](#_Toc457396241)

[2. Clone the project 5](#_Toc457396242)

[3. Code Review Workflow 5](#_Toc457396243)

[4. Upload a Change 5](#_Toc457396244)

[5. Review Change 6](#_Toc457396245)

[6. Update the Change 7](#_Toc457396246)

[7. Developing multiple features in parallel ( or take more task one time ) 7](#_Toc457396247)

[8. Adding Reviewers 7](#_Toc457396248)

[9. Submit a Change 7](#_Toc457396249)

[10. Rebase a Change 8](#_Toc457396250)

[11. Abandon/Restore a Change 8](#_Toc457396251)

[12. Using Topics 9](#_Toc457396252)

[13. Working with Drafts 9](#_Toc457396253)

[14. Inline Edit 9](#_Toc457396254)

[15. Working without Code Review 9](#_Toc457396255)

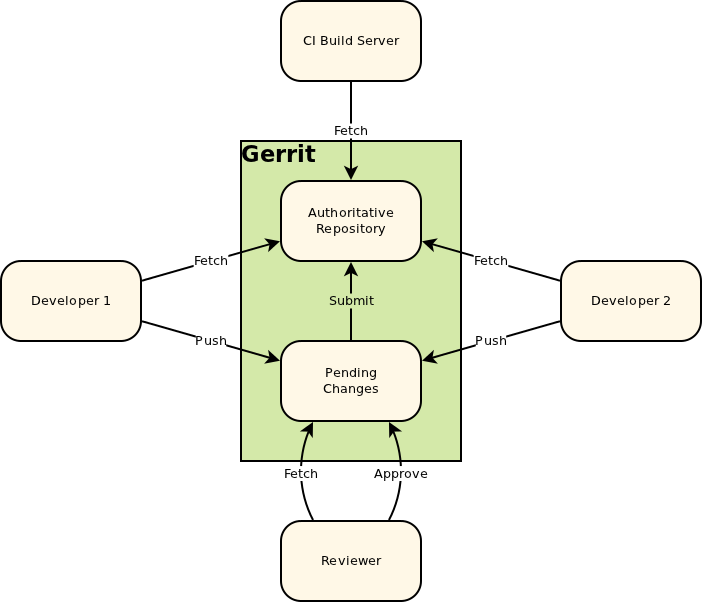
# I. Quick introduction

## Overview

Gerrit is a web-based code review tool built on top of the GIT version control system. Gerrit is intended to provide a lightweight framework for reviewing every commit before it is accepted into the code base. Changes ( The commit in GIT ) are uploaded to Gerrit but don’t actually become a part of the project until they’ve been reviewed and accepted.

## Where does Gerrit fit in

Gerrit is deployed in place of this central repository and adds an additional concept, a store of pending changes. Everyone still fetches from the authoritative repository but instead of pushing back to it, they push to this pending changes location. A change can only be submitted into the authoritative repository and become an accepted part of the project once the change has been reviewed and approved.



# II. Development Guide

## Tool

Gerrit speaks the git protocol. This means in order to work with Gerrit you do not need to install any Gerrit client, but having a regular git client, such as the git command line or EGit in Eclipse, is sufficient. This guide prefer the git command.

## Clone the project

Cloning a Gerrit project is done the same way as cloning any other git repository by using the git clone command. The URL for cloning the project can be found in the Gerrit web UI under *Projects* > *List* > *<project-name>* > *General*. The example as below:

$ git clone http://android-review.fsoft.com.vn/projectname

Cloning into projectname...

Gerrit supports the SSH and the HTTP/HTTPS protocols.

## Code Review Workflow

With Gerrit Code Review means to review every commit before it is accepted into the code base. The author of a code modification uploads a commit as a change to Gerrit. In Gerrit each change is stored in a staging area where it can be checked and reviewed. Only when it is approved and submitted it gets applied to the code base. If there is feedback on a change, the author can improve the code modification by amending the commit and uploading the new commit as a new patch set. This way a change is improved iteratively and it is applied to the code base only when is ready.

## Upload a Change

Uploading a change to Gerrit is done by pushing a commit to Gerrit. The commit must be pushed to a ref in the *refs/for/* namespace which defines the target branch: *refs/for/<target-branch>.*

The following example with target branch as *master* branch.

$ git commit

$ git push origin HEAD:refs/for/master

// this is the same as:

$ git commit

$ git push origin HEAD:refs/for/refs/heads/master

In the some case, you’d like to push directly ( bypassing code review) to the Git repository, flow as command.

$ git commit

$ git push origin HEAD:master

// this is the same as:

$ git commit

$ git push origin HEAD:refs/heads/master

**Note:** If pushing to Gerrit fails consult the Gerrit documentation that explains the *II. Appendix 2. Error Messages*

When a commit is pushed for review, Gerrit stores it in a staging area which is a branch in the special refs/changes/ namespace. A change ref has the format refs/changes/XX/YYYY/ZZ where YYYY is the numeric change number, ZZ is the patch set number and XX is the last two digits of the numeric change number, e.g. refs/changes/20/884120/1. Understanding the format of this ref is not required for working with Gerrit.

A change consists of a Change-Id, meta data (owner, project, target branch etc.), one or more patch sets, comments and votes. A patch set is a git commit. Each patch set in a change represents a new version of the change and replaces the previous patch set. Only the latest patch set is relevant. This means all failed iterations of a change will never be applied to the target branch, but only the last patch set that is approved is integrated.

The Change-Id is important for Gerrit to know whether a commit that is pushed for code review should create a new change or whether it should create a new patch set for an existing change.

The Change-Id is a SHA-1 that is prefixed with an uppercase I. It is specified as footer in the commit message (last paragraph):

[Modify] Some changes

[Objective] Bug #JIRA\_ID

[Test] OK

Change-Id: Ie5b8a9e01af3c680d9049d20fc795f944ea6fdcd

If a commit that has a Change-Id in its commit message is pushed for review, Gerrit checks if a change with this Change-Id already exists for this project and target branch, and if yes, Gerrit creates a new patch set for this change. If not, a new change with the given Change-Id is created.

If a commit without Change-Id is pushed for review, Gerrit rejects the change with the error message: *Missing Change-Id in commit message footer*. See more at the *II. Appendix 2. Error Messages*

Change-Ids are unique for a branch of a project. E.g. commits that fix the same issue in different branches should have the same Change-Id, which happens automatically if a commit is cherry-picked to another branch. This way you can search by the Change-Id in the Gerrit web UI to find a fix in all branches.

Change-Ids will be created automatically by installing the commit-msg hook in the git repository.

## Review Change

After uploading a change for review reviewers can inspect it via the Gerrit web UI. Reviewers can see the code delta and comment directly in the code on code blocks or lines. They can also post summary comments and vote on review labels. The documentation of the review UI explains the screens and controls for doing code reviews, checkout it at *II. Appendix 1. REVIEW UI*

## Update the Change ( Rework )

If there is feedback from code review and a change should be improved a new patch set with the reworked code should be uploaded. This is done by amending the commit of the last patch set and push again.

It is important that the commit message contains the Change-Id of the change that should be updated as a footer (last paragraph). Normally the commit message already contains the correct Change-Id and the Change-Id is preserved when the commit is amended.

// fetch and checkout the change

// (checkout command copied from change screen)

$ git fetch https://gerrithost/myProject refs/changes/74/67374/2 && git checkout FETCH\_HEAD

// rework the change

$ git add <path-of-reworked-file>

...

// amend commit

$ git commit --amend

// push patch set

$ git push origin HEAD:refs/for/master

Note: Never amend a commit that is already part of a central branch.

## Developing multiple features in parallel ( or take more task one time )

Code review takes time, which can be used by the change author to implement other features. Each feature should be implemented in its own local feature branch that is based on the current HEAD of the target branch. This way there is no dependency to open changes and new features can be reviewed and applied independently. If wanted, it is also possible to base a new feature on an open change. This will create a dependency between the changes in Gerrit and each change can only be applied if all its predecessor are applied as well. Dependencies between changes can be seen from the Related Changes tab on the change screen.

## Adding Reviewers

In the change screen reviewers can be added explicitly to a change. The added reviewer will then be notified by email about the review request.

Mainly this functionality is used to request the review of specific person who is known to be an expert in the modified code or who is a stakeholder of the implemented feature. Normally it is not needed to explicitly add reviewers on every change, but you rather rely on the project team to watch their project and to process the incoming changes by importance, interest, time etc.

## Submit a Change

Submitting a change means that the code modifications of the current patch set are applied to the target branch. Submit requires the Submit access right and is done on the change screen by clicking on the Submit button.

In order to be submittable changes must first be approved by voting on the review labels. By default a change can only be submitted if it has a vote with the highest value on each review label and no vote with the lowest value (veto vote).

**Note:** Submitting a change may fail with conflicts. In this case you need to rebase the change locally, resolve the conflicts and upload the commit with the conflict resolution as new patch set. If a change cannot be merged due to path conflicts this is highlighted on the change screen by a bold red *Cannot Merge* label.

## Rebase a Change

While a change is in review the HEAD of the target branch can evolve. In this case the change can be rebased onto the new HEAD of the target branch. When there are no conflicts the rebase can be done directly from the change screen, otherwise it must be done locally.

Rebase on local:

// update the remote tracking branches

$ git fetch

// fetch and checkout the change

// (checkout command copied from change screen)

$ git fetch https://gerrithost/myProject refs/changes/74/67374/2 && git checkout FETCH\_HEAD

// do the rebase

$ git rebase origin/master

// resolve conflicts if needed and stage the conflict resolution

...

$ git add <path-of-file-with-conflicts-resolved>

// continue the rebase

$ git rebase --continue

// push the commit with the conflict resolution as new patch set

$ git push origin HEAD:refs/for/master

**Note:** Never rebase commits that are already part of a central branch.

## Abandon/Restore a Change

Sometimes during code review a change is found to be bad and it should be given up. In this case the change can be abandoned so that it doesn’t appear in list of open changes anymore.

Abandoned changes can be restored if later they are needed again.

## Using Topics

Changes can be grouped by topics. This is useful because it allows you to easily find related changes by using the topic search operator. Also on the change screen changes with the same topic are displayed so that you can easily navigate between them.

Often changes that together implement a feature or a user story are group by a topic. Assigning a topic to a change can be done in the change screen. It is also possible to set a topic on push.

git push origin HEAD:refs/for/master%topic=FEATURE\_ONE

## Working with Drafts

Changes can be uploaded as drafts. By default draft changes are only visible to the change owner. This gives you the possibility to have some staging before making your changes visible to the reviewers. Draft changes can also be used to backup unfinished changes.

A draft change is created by pushing to the magic *refs/drafts/<target-branch>* ref.

$ git commit

$ git push origin HEAD:refs/drafts/master

Draft changes have the state Draft and can be published or deleted from the change screen. By adding reviewers to a draft change the change is made visible to these users. This way you can collaborate with other users in privacy.

By pushing to *refs/drafts/<target-branch>* you can also upload draft patch sets to non-draft changes. Draft patch sets are immediately visible to all reviewers of the change, but other users cannot see the draft patch set. A draft patch set can be published and deleted in the same way as a draft change.

## Inline Edit

It is possible to edit changes inline directly in the web UI. This is useful to make small corrections immediately and publish them as a new patch set.

## Working without Code Review

Doing code reviews with Gerrit is optional and you can use Gerrit without code review as a pure Git server.

**Note:** Bypassing code review must be enabled in the project access rights. The project owner must allow it by assigning the *Push* access right on the target branch (*refs/heads/<branch-name>).*

$ git commit

$ git push origin HEAD:master

// this is the same as:

$ git commit

$ git push origin HEAD:refs/heads/master

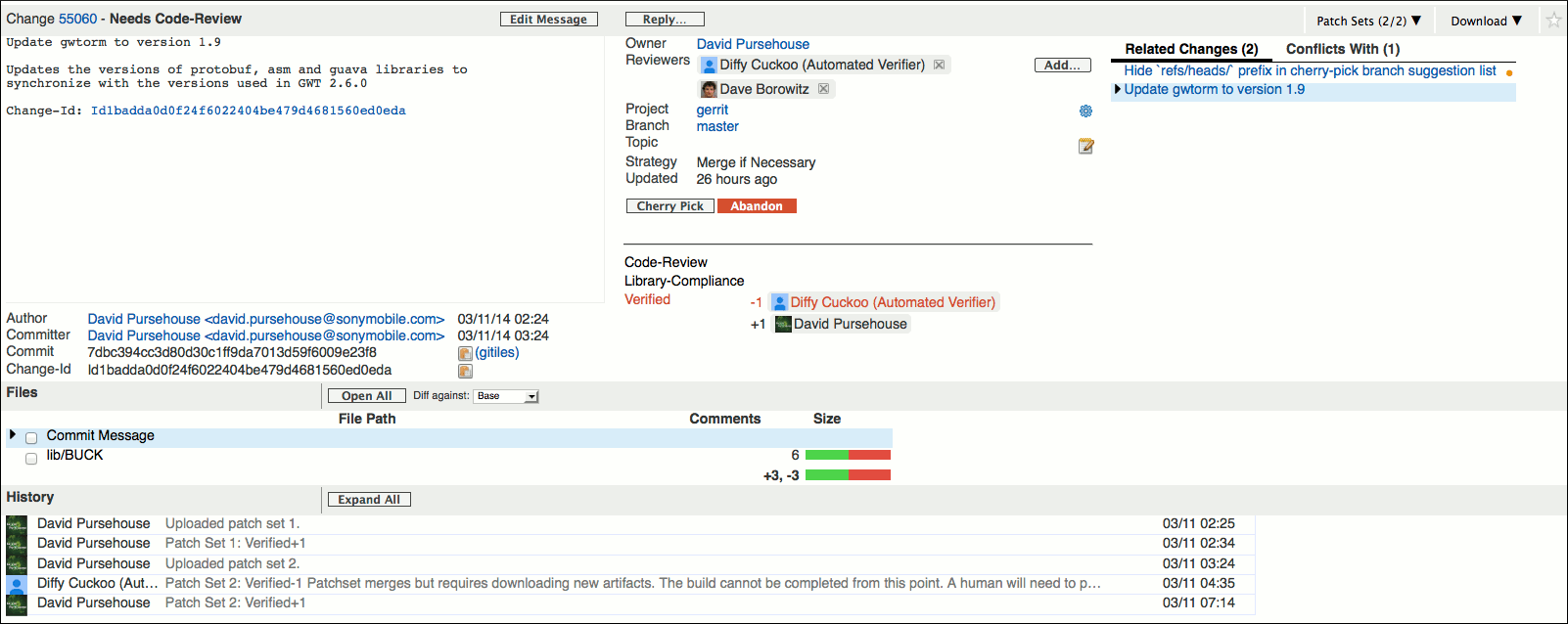
# II. Appendix

## Review UI

Reviewing changes is an important task and the Gerrit Web UI provides many functionalities to make the review process comfortable and efficient. This is a guide through the review UI that explains the different functions and UI elements.

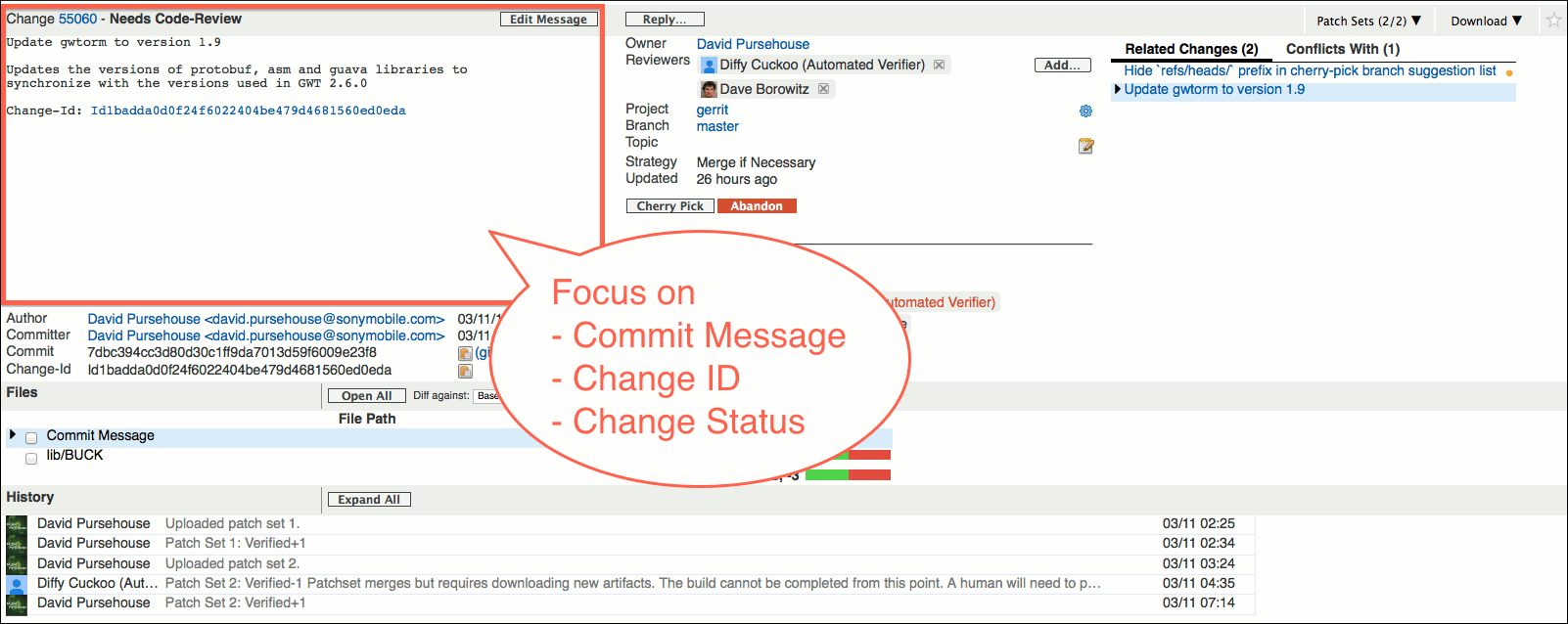
### Change Screen

The change screen shows the details of a single change and provides various actions on it.

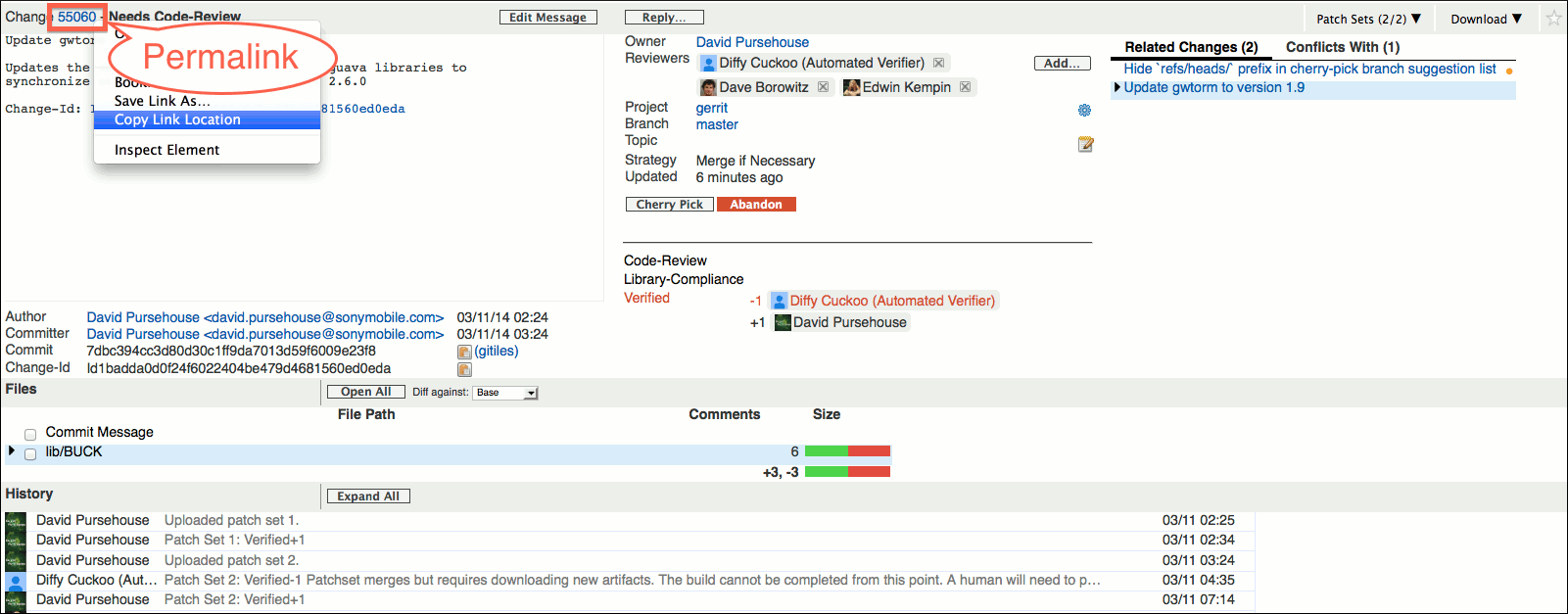


### Commit Message Block

The focus of the change screen is on the commit message since this is the most important information about a change. The numeric change ID and the change status are displayed right above the commit message.



The numeric change ID is a link to the change and clicking on it refreshes the change screen. By copying the link location you can get the permalink of the change.



The change status shows the state of the change:

*Needs <label>:*

The change is in review and an approval on the shown label is still required to make the change submittable.

*Not <label>:*

The change is in review and a veto vote on the shown label is preventing the submit.

*Not Current:*

The currently viewed patch set is outdated.

Please note that some operations, like voting, are not available on outdated patch sets, but only on the current patch set.

*Ready to Submit:*

The change has all necessary approvals and may be submitted.

*Merged:*

The change was successfully merged into the destination branch.

*Abandoned:*

The change was abandoned.

*Draft:*

The change is a draft that is only visible to the change owner, the reviewers that were explicitly added to the change, and users who have the View Drafts global capability assigned.

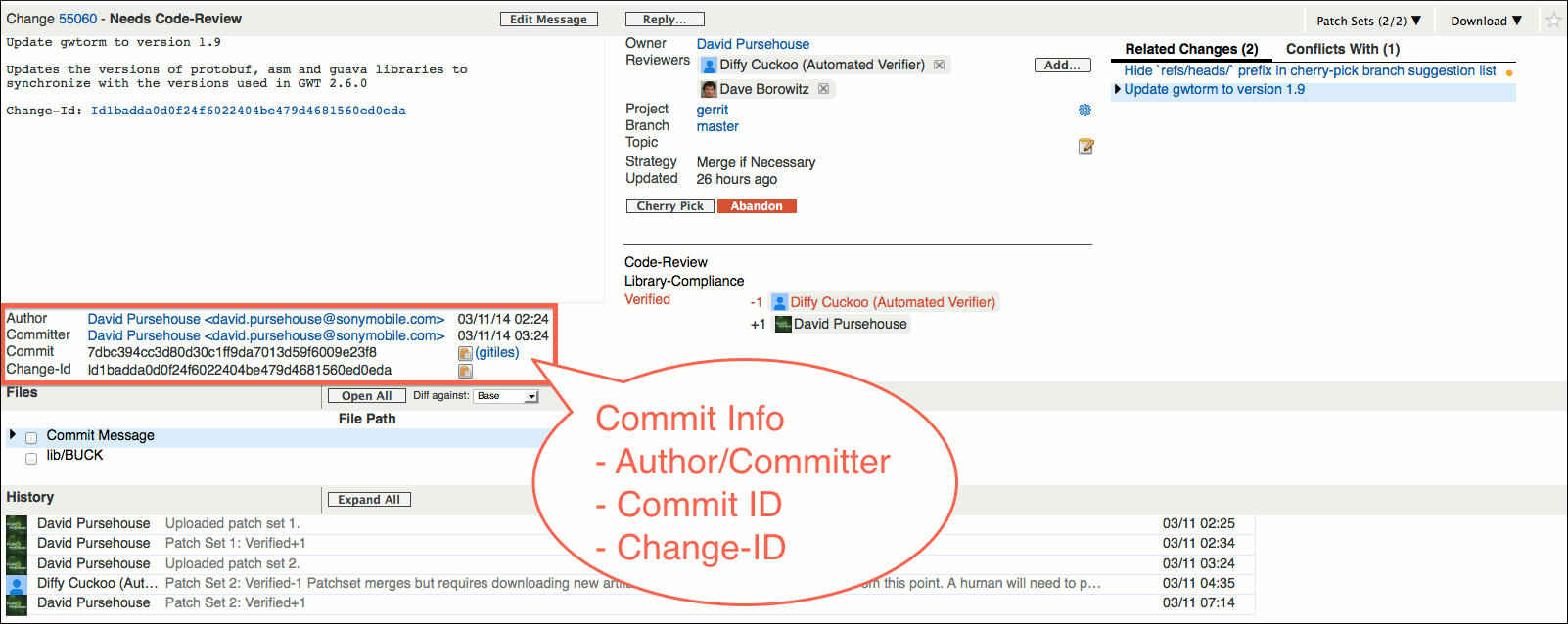
### Commit Info Block

The commit info block shows information about the commit of the currently viewed patch set.

It displays the author and the committer as links to a list of this person’s changes that have the same status as the currently viewed change.

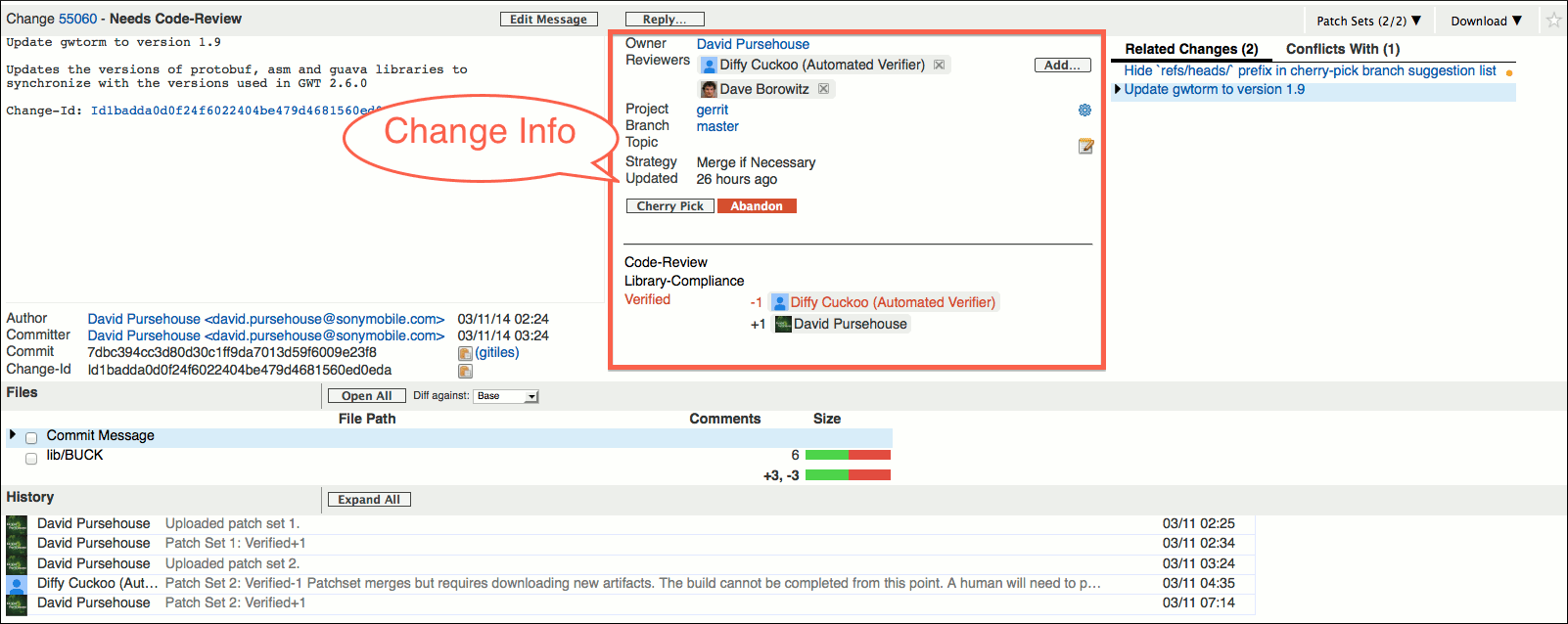
The commit ID, the parent commit(s) and the Change-Id are displayed with a copy-to-clipboard icon that allows the ID to be copied into the clipboard.

If a Git web browser, such as gitweb or Gitiles, is configured, there is also a link to the commit in the Git web browser.

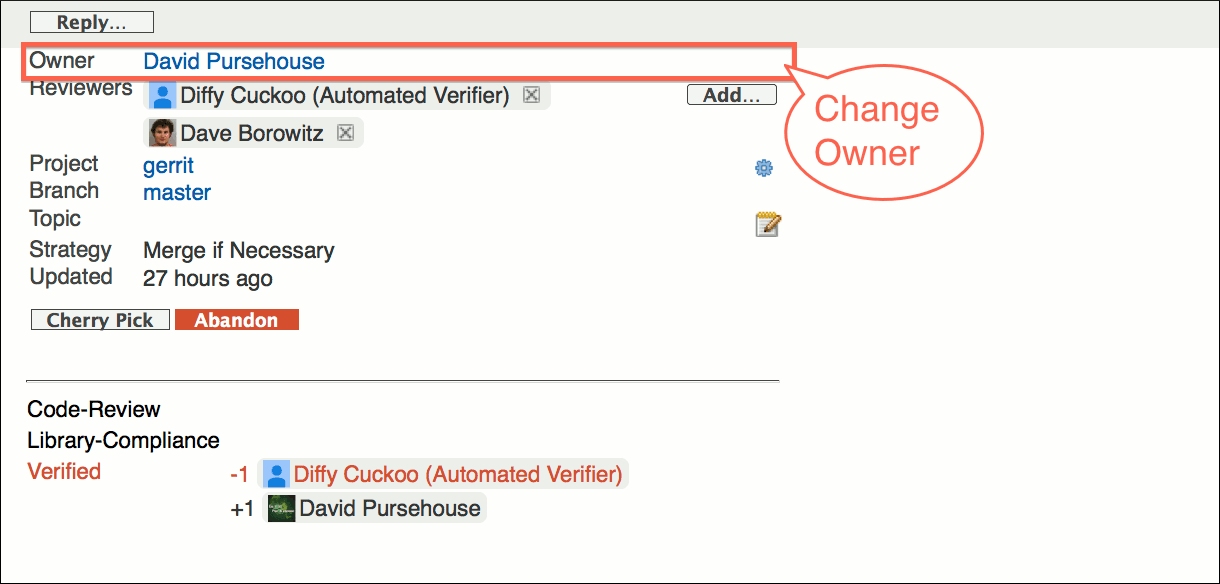


### Change Info Block

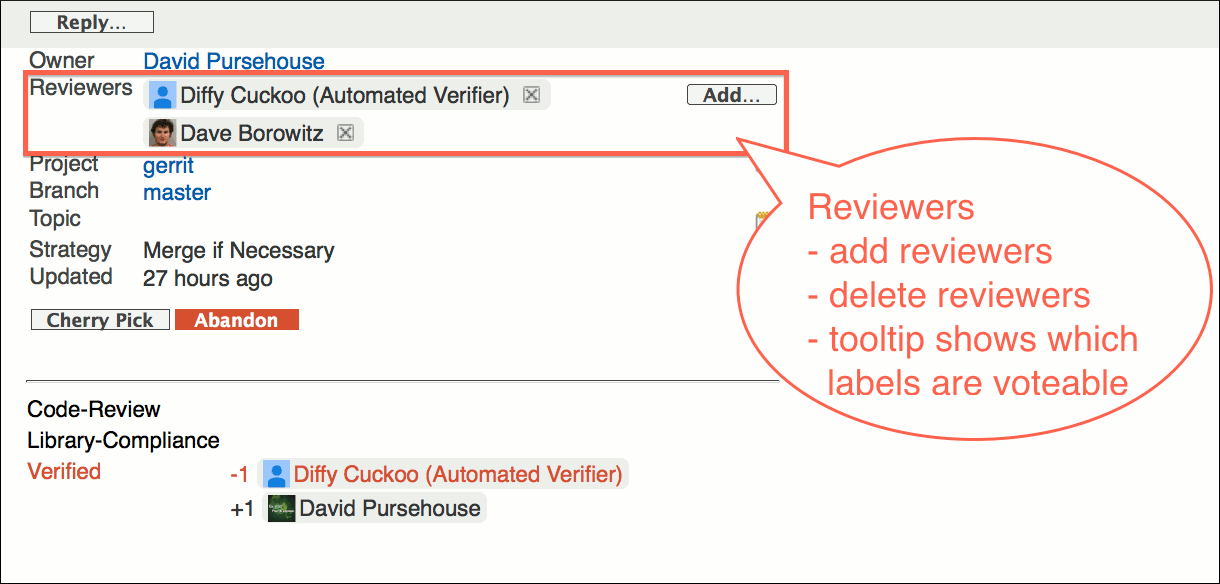
The change info block contains detailed information about the change and offers actions on the change.



* *Change Owner*: The owner of the change is displayed as a link to a list of the owner’s changes that have the same status as the currently viewed change.



* *Reviewers*: The reviewers of the change are displayed as chip tokens. For each reviewer there is a tooltip that shows on which labels the reviewer is allowed to vote. New reviewers can be added by clicking on the *Add*…​ button. Typing into the pop-up text field activates auto completion of user and group names.

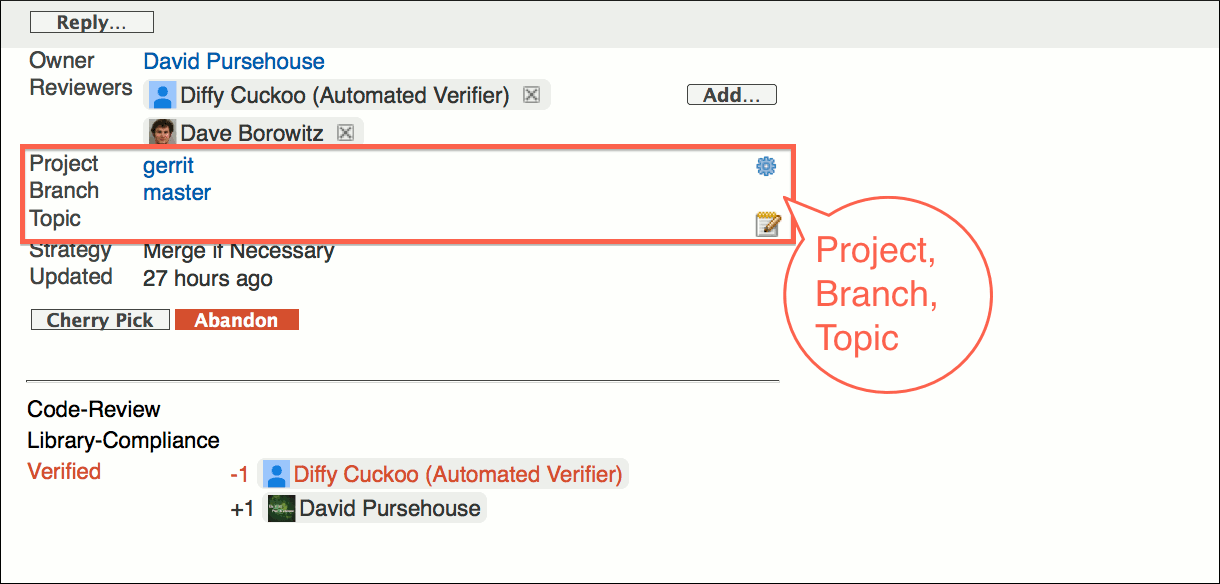


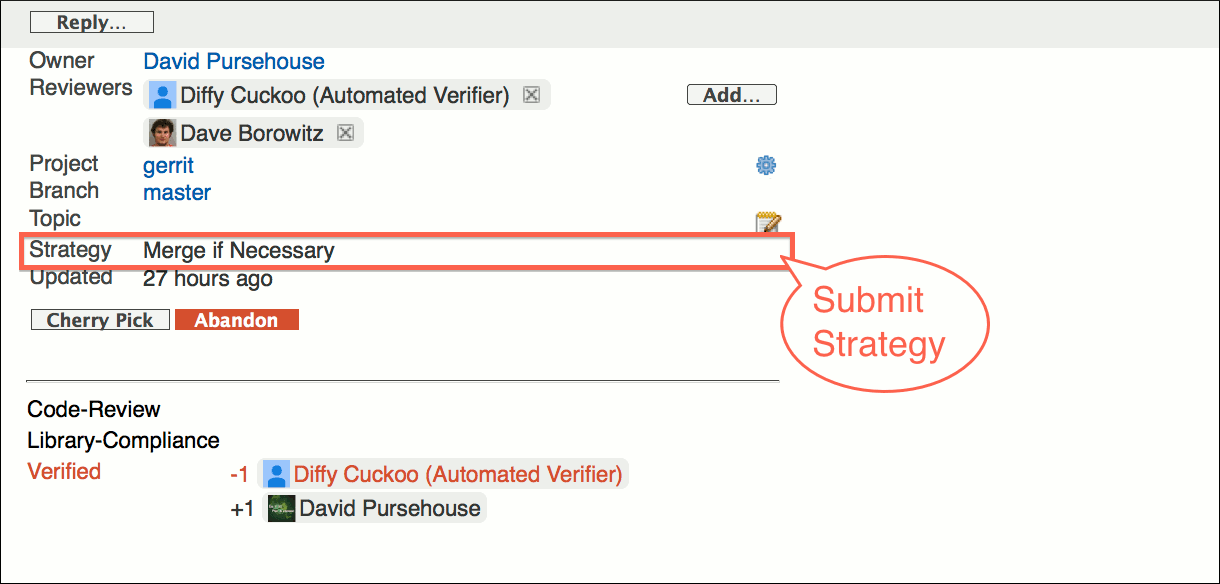
* *Project / Branch / Topic:* The name of the project for which the change was done is displayed as a link to the default dashboard of the project. If no default dashboard is defined, the link opens a list of open changes on the project.

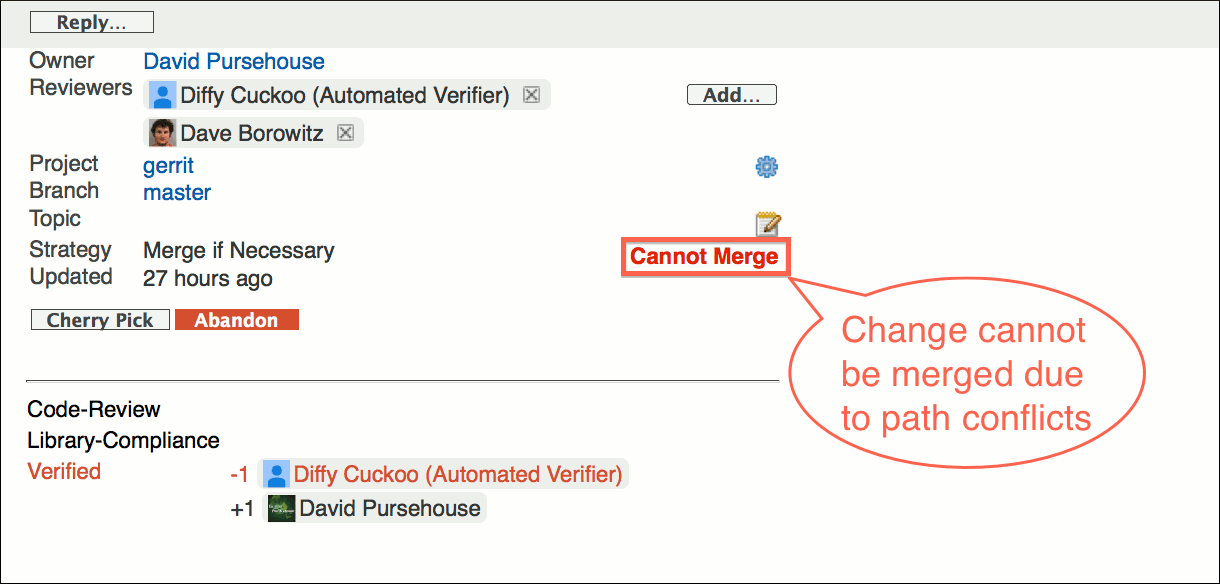
Clicking on the settings icon on the right side navigates to the project administration screen.

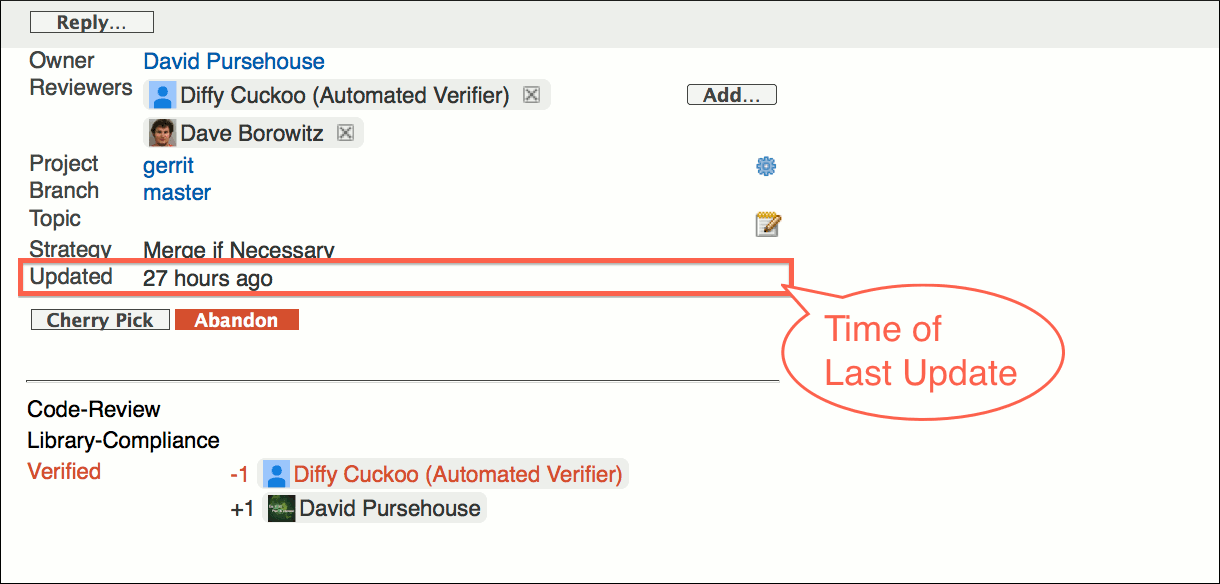
The name of the destination branch is displayed as a link to a list with all changes on this branch that have the same status as the currently viewed change.

If a topic was assigned to the change it is displayed below the branch. By clicking on the edit icon the topic can be set. This requires the Edit Topic Name access right. To be able to set a topic on a closed change, the Edit Topic Name must be assigned with the force flag.



* *Submit Strategy*: The submit strategy that will be used to submit the change. The submit strategy is only displayed for open changes.  
    
  

If a change cannot be merged due to path conflicts this is highlighted by a bold red Cannot Merge label.   


* *Time of Last Update*:   
  
* *Action*: Depending on the change state and the permissions of the user, different actions are available on the change:

*Submit*:

Submits the change and adds it to the merge queue. If possible the change is merged into the destination branch.

The Submit button is available if the change is submittable and the Submit access right is assigned.

*Abandon*:

Abandons the change.

The Abandon button is only available if the change is open and the Abandon access right is assigned.

When a change is abandoned, a panel appears that allows one to type a comment message to explain why the change is being abandoned.

*Restore*:

Restores the change.

The Restore button is only available if the change is abandoned and the Abandon and the Push access right is assigned.

When a change is restored, a panel appears that allows one to type a comment message to explain why the change is being restored.

*Rebase*:

Rebases the change. The rebase is always done with content merge enabled. If the rebase is successful a new patch set with the rebased commit is created. If the rebase fails, there are conflicts that have to be resolved manually.

If the change does not depend on another open change, it is rebased onto the tip of the destination branch.

If the change depends on another open change, it is rebased onto the current patch set of that other change.

It is possible to change parent revision of a change. The new parent revision can be another change towards the same target branch, or the tip of the target branch.

The Rebase button is only available if the Rebase access right is assigned. Rebasing merge commits is not supported.

*Cherry-Pick*:

Allows to cherry-pick the change to another branch. The destination branch can be selected from a dialog. Cherry-picking a change creates a new open change on the selected destination branch.

It is also possible to cherry-pick a change to the same branch. This is effectively the same as rebasing it to the current tip of the destination branch. This can be used to remove dependencies on other open changes.

Users can only cherry-pick changes to branches for which they are allowed to upload changes for review.

*Publish*:

Publishes the currently viewed draft patch set. If this is the first patch set of a change that is published, the change will be published as well.

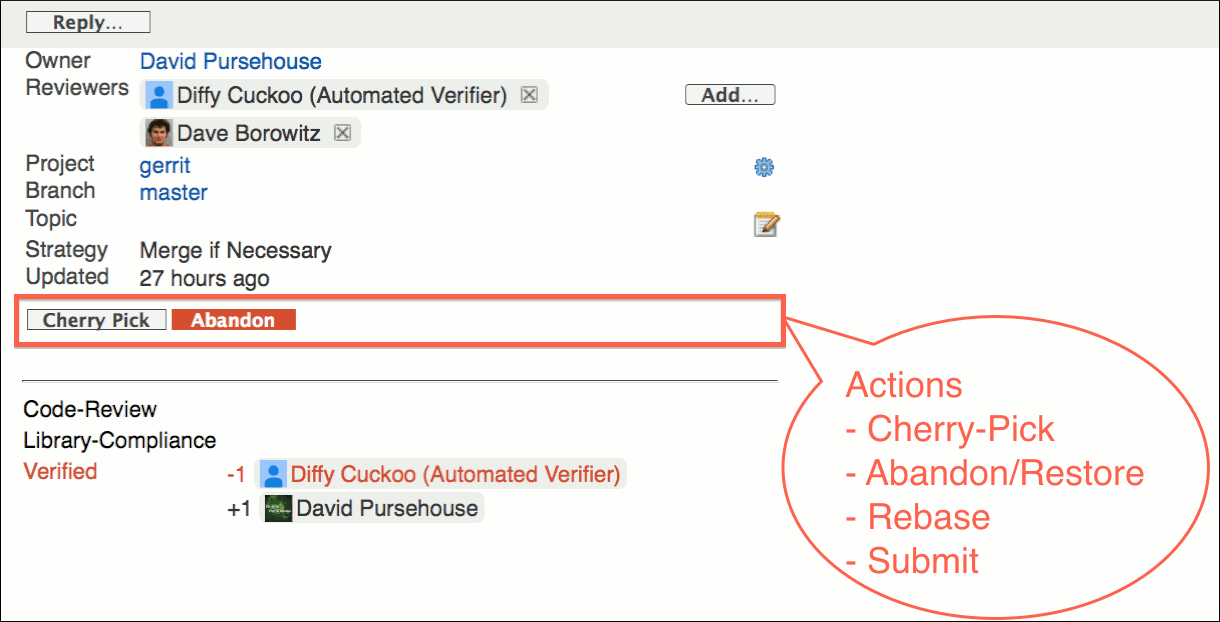
The Publish button is only available if a draft patch set is viewed and the user is the change owner or has the Publish Drafts access right assigned.

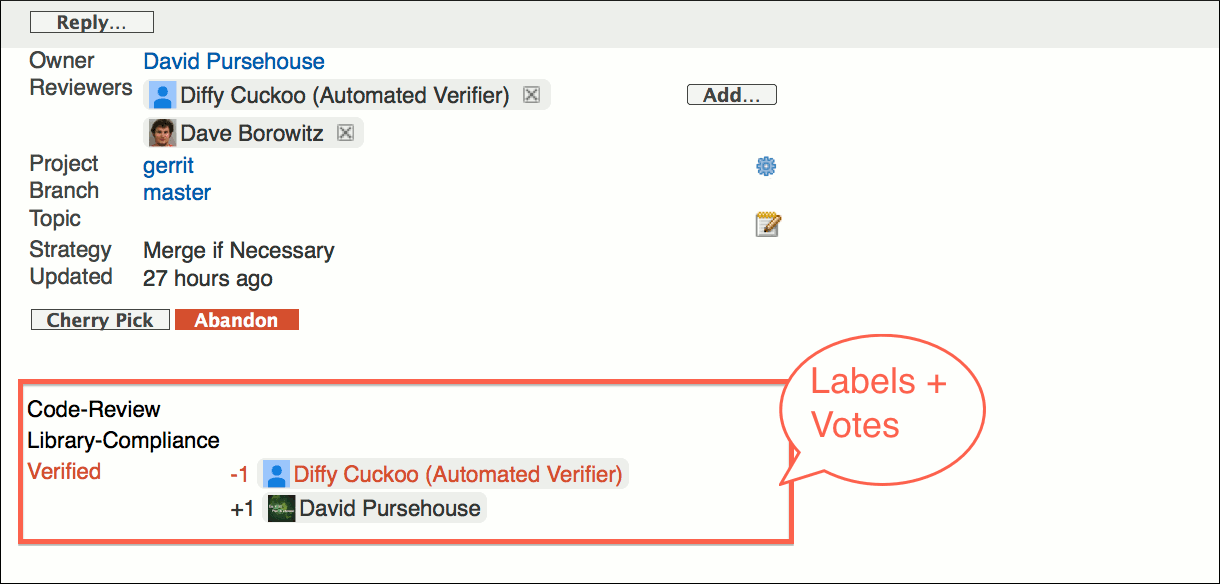
*Delete Change / Delete Revision*:

Deletes the draft change / the currently viewed draft patch set.

The Delete Change / Delete Revision buttons are only available if a draft patch set is viewed and the user is the change owner or has the Delete Drafts access right assigned.

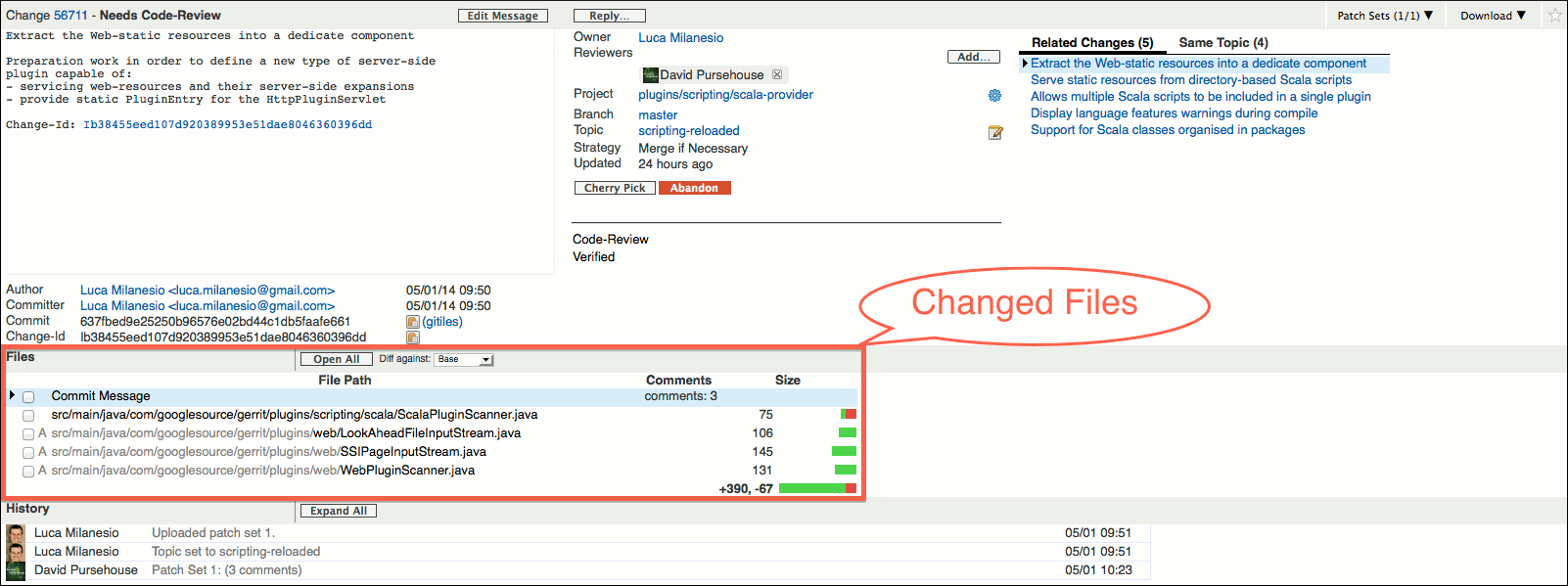
Further actions may be available if plugins are installed.



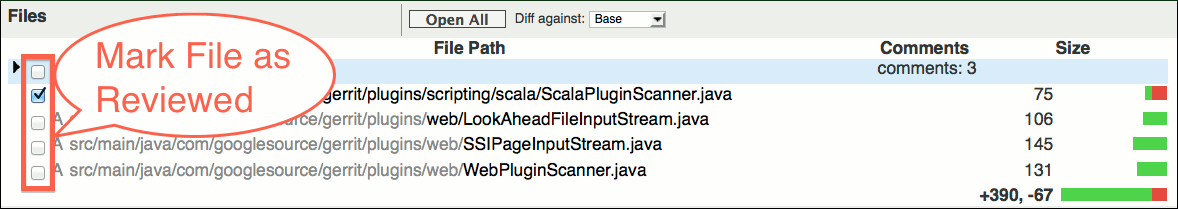
* Labels & Votes: Approving votes are colored green; veto votes are colored red. 

### File List

The file list shows the files that are modified in the currently viewed patch set.



The checkboxes in front of the file names allow files to be marked as reviewed.



The type of a file modification is indicated by the character in front of the file name:

*'no character' (Modified):*

The file existed before this change and is modified.

*A (Added):*

The file is newly added.

*D (Deleted):*

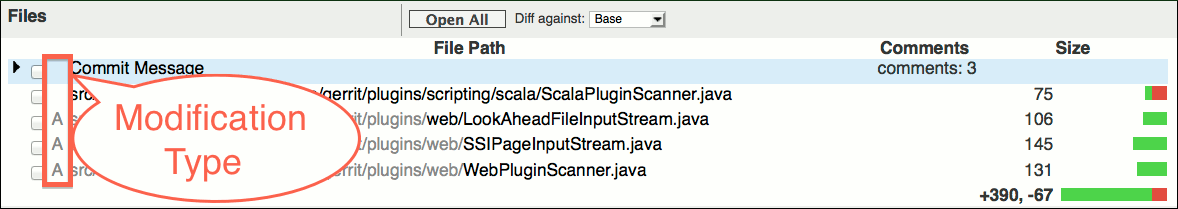
The file is deleted.

*R (Renamed):*

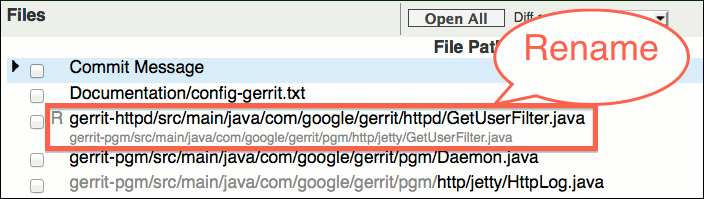
The file is renamed.

*C (Copied):*

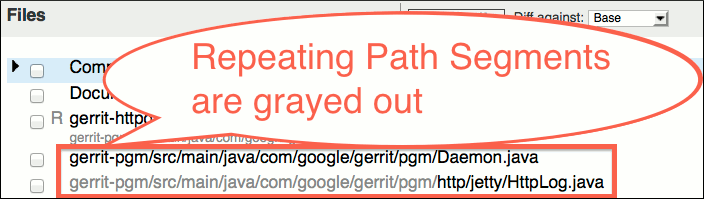
The file is new and is copied from an existing file.



If a file is renamed or copied, the name of the original file is displayed in gray below the file name.



Repeating path segments are grayed out.



Inline comments on a file are shown in the Comments column.

Draft comments, i.e. comments that have been written by the current user but not yet published, are highlighted in red.

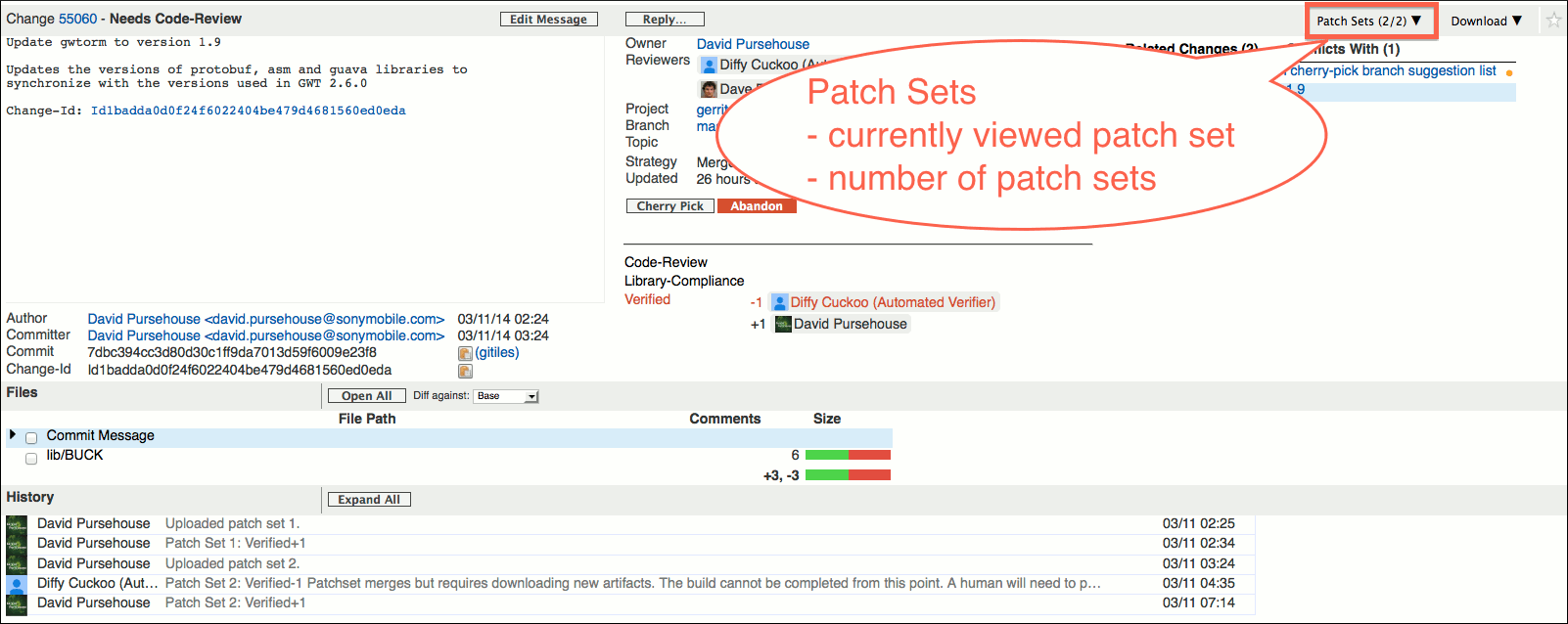
New comments from other users, that were published after the current user last reviewed this change, are highlighted in bold.



### Patch Sets

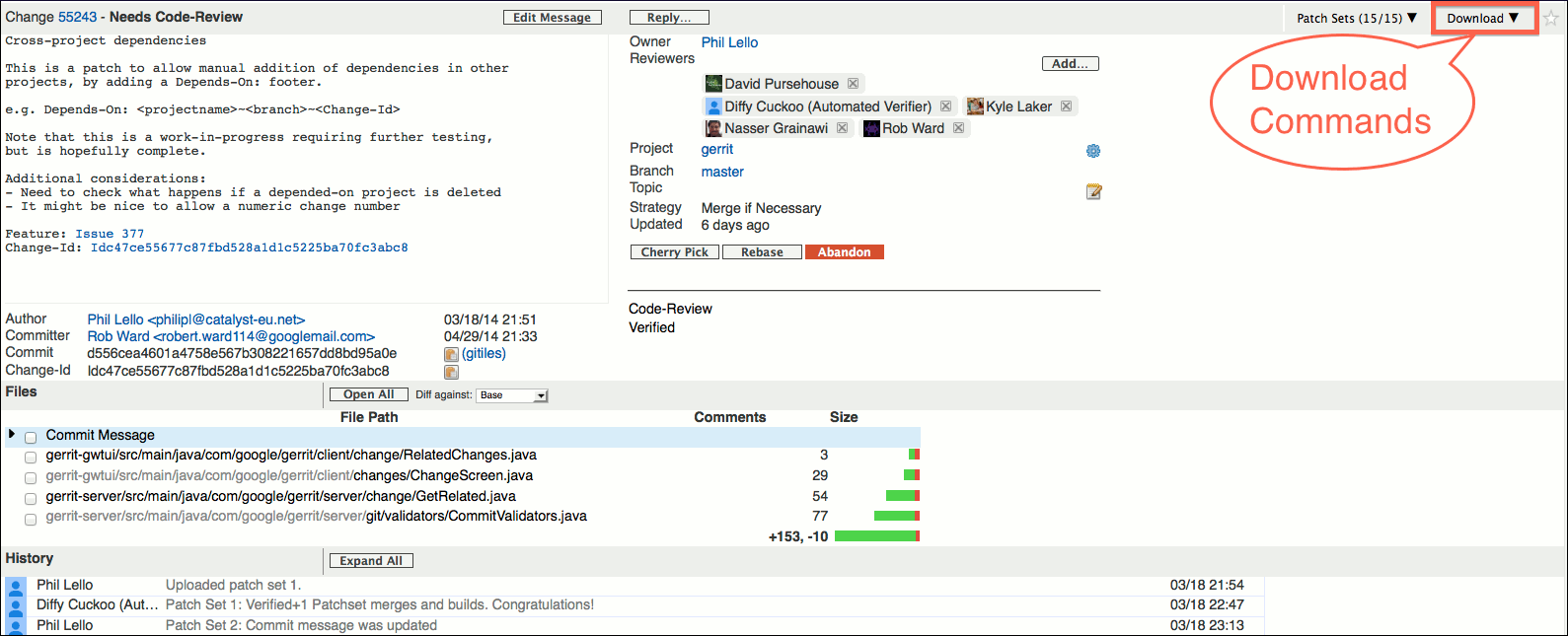
The change screen only presents one patch set at a time. Which patch set is currently viewed can be seen from the Patch Sets drop-down panel in the change header. It shows the number of the currently viewed patch set and the total number of patch sets, in the form: "current patch set/number of patch sets".

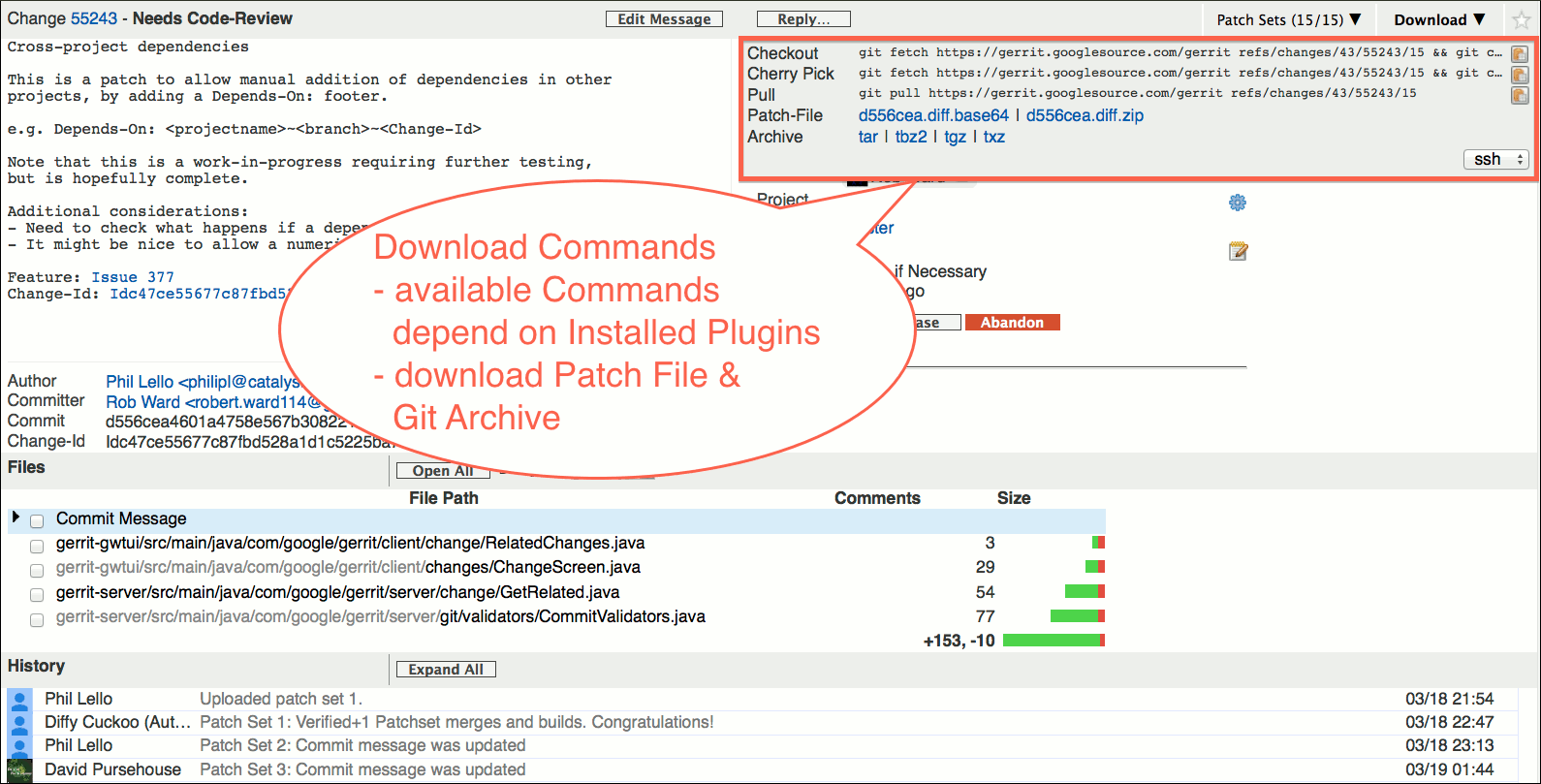
If a non-current patch set is viewed this is indicated by the Not Current change state. Please note that some operations are only available on the current patch set.



### Download

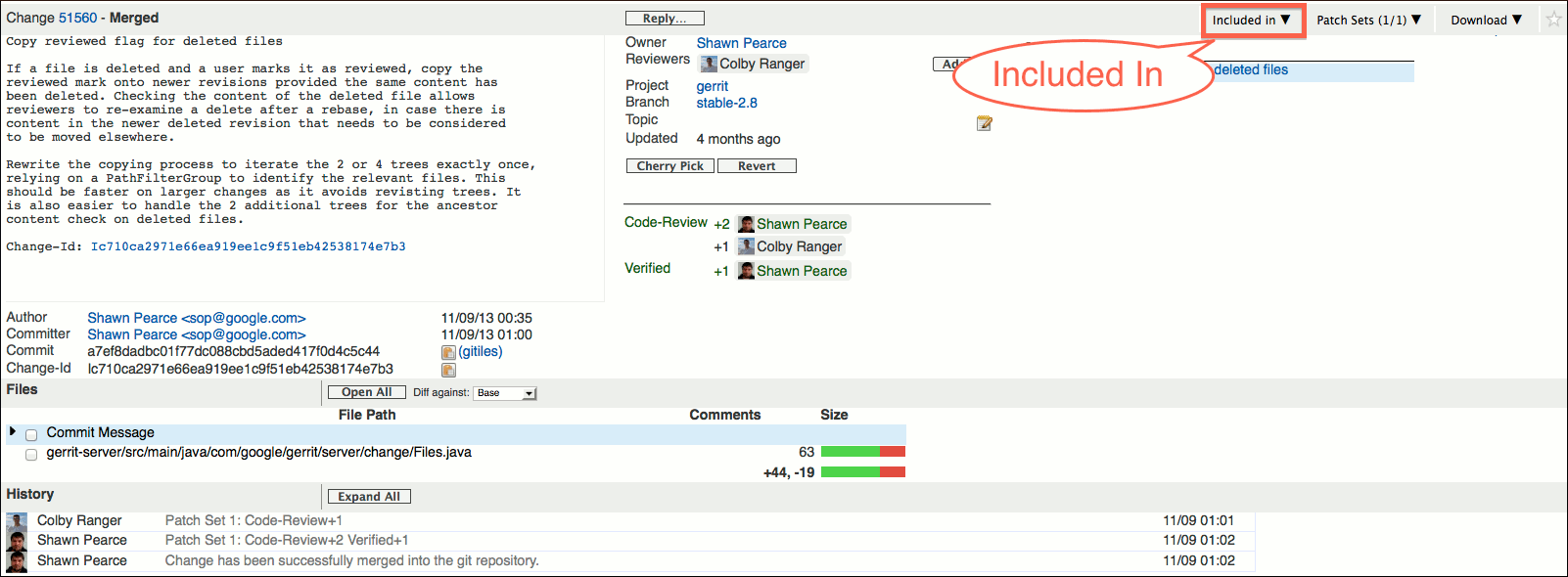
The Download drop-down panel in the change header offers commands and links for downloading the currently viewed patch set.



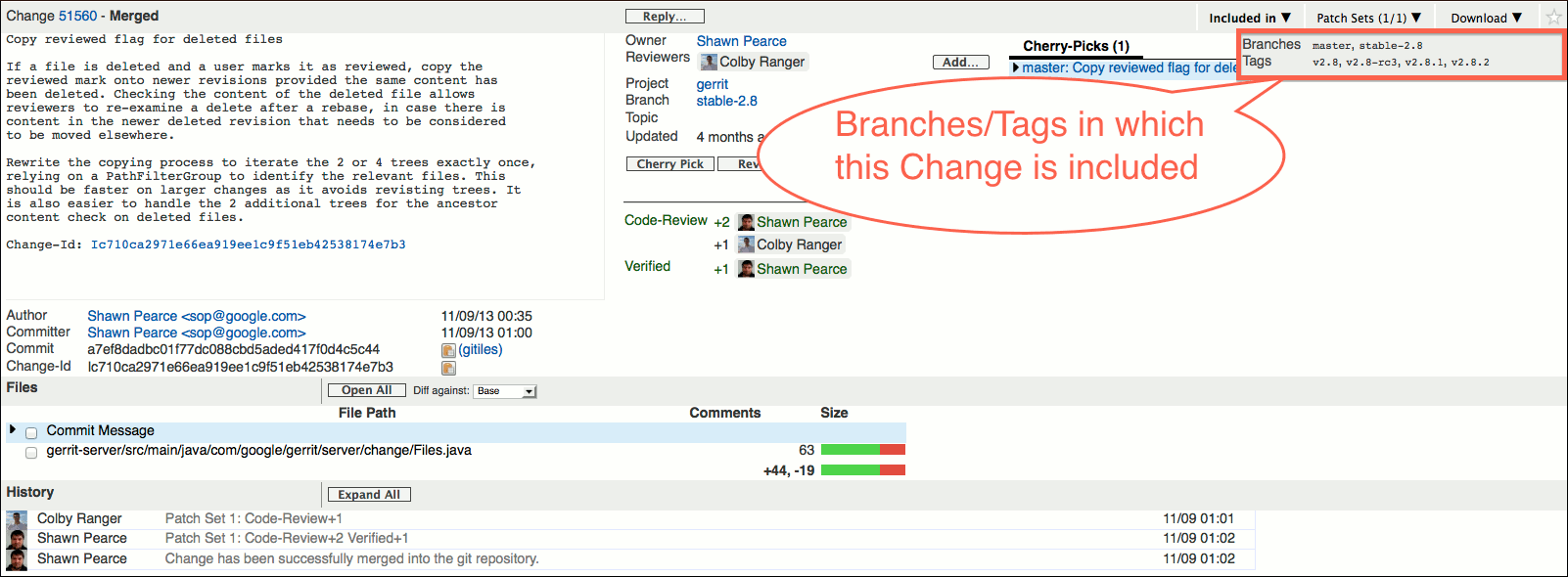


### Included In

For merged changes the Included In drop-down panel is available in the change header.

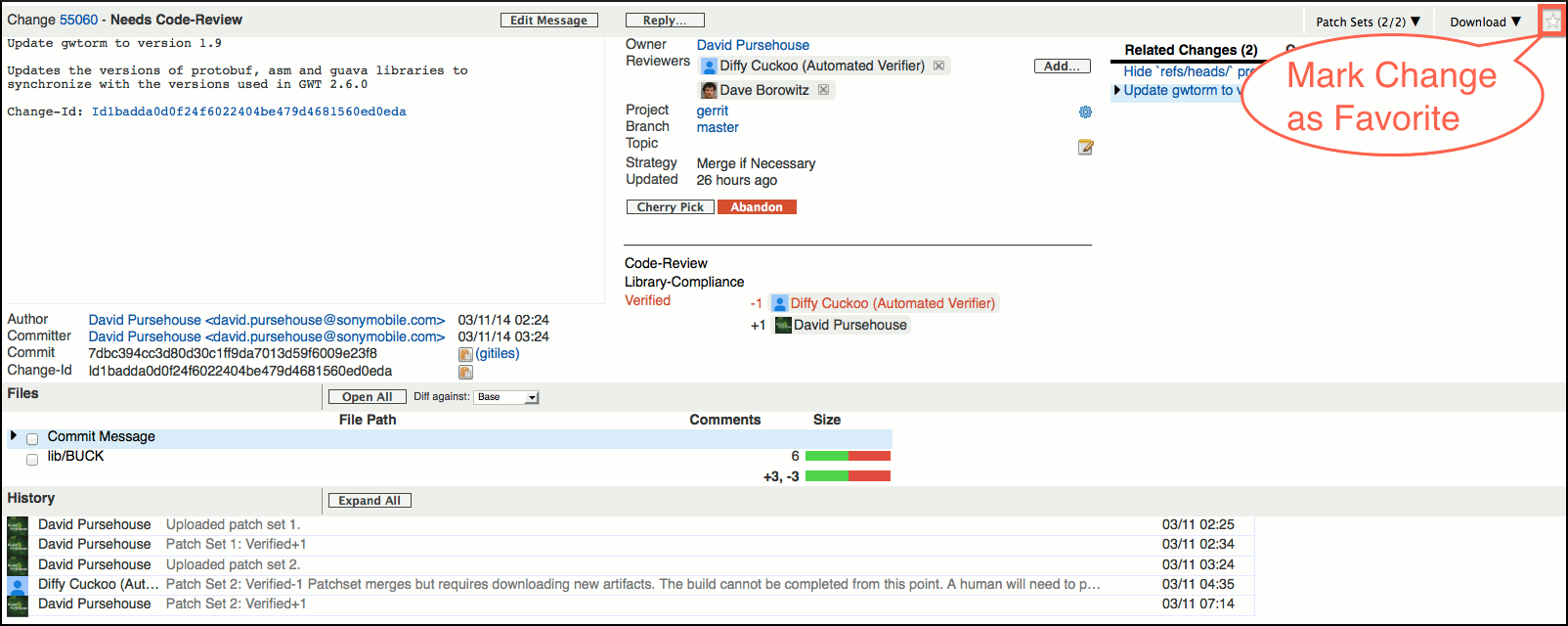


The Included In drop-down panel shows the branches and tags in which the change is included. E.g. if a change fixes a bug, this allows to quickly see in which released versions the bug-fix is contained (assuming that every release is tagged).



### Star Change

The star icon in the change header allows to mark the change as a favorite. Clicking on the star icon again, unstars the change.



Starring a change turns on email notifications for this change. Starred changed are listed under *My > Starred Changes*. and can be queried by the *is:starred* search operator.

### Related Changes

If there are changes that are related to the currently viewed change they are displayed in the third column of the change screen. There are several lists of related changes and a tab control is used to display each list of related changes in its own tab.

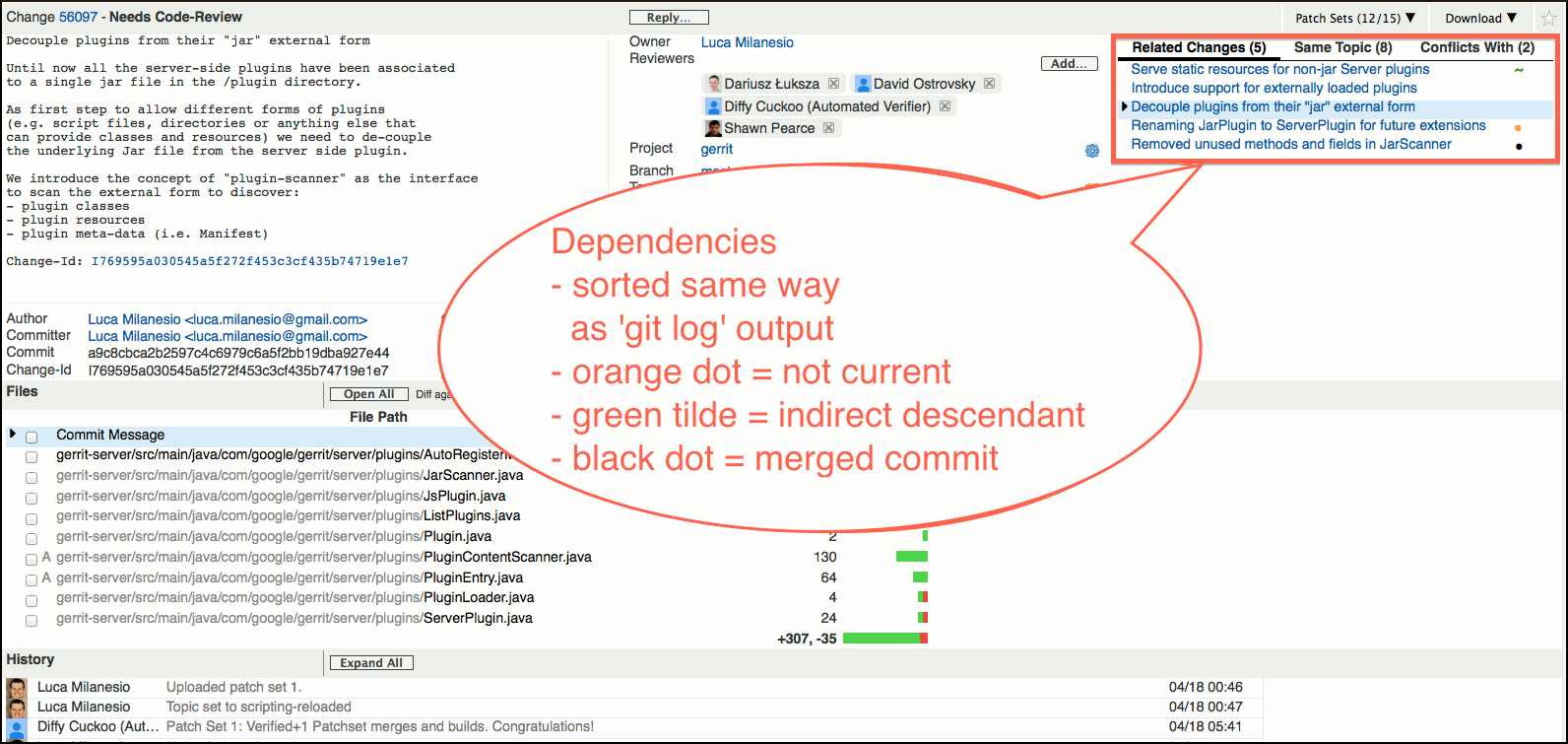
The following tabs may be displayed:

*Related Changes:*

This tab page shows changes on which the current change depends (ancestors) and open changes that depend on the current change (descendants). For merge commits it also shows the closed changes that will be merged into the destination branch by submitting the merge commit.

The changes are sorted in the same way as the output of 'git log'. This means the relationship between the changes can be inferred from the position of the changes in the list. Changes listed above the current change are descendants; changes below the current change are ancestors.

For merged changes this tab is only shown if there are open descendants.



Related changes may be decorated with an icon to signify dependencies on outdated patch sets, or commits that are not associated to changes under review:

*Orange Dot:*

The selected patch set of the change is outdated; it is not the current patch set of the change.

If an ancestor change is marked with an orange dot it means that the currently viewed patch set depends on a outdated patch set of the ancestor change. This is because a new patch set for the ancestor change was uploaded in the meantime and as result the currently viewed patch set now needs to be rebased.

If a descendant change is marked with an orange dot it means that an old patch set of the descendant change depends on the currently viewed patch set. It may be that the descendant was rebased in the meantime and with the new patch set this dependency was removed.

*Green Tilde:*

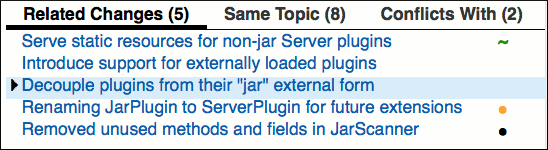
The selected patch set of the change is an indirect descendant of the currently viewed patch set; it has a dependency to another patch set of this change. E.g. this could mean that a new patch set was uploaded for this change and the descendant change now needs to be rebased. Please note that following the link to an indirect descendant change may result in a completely different related changes listing.

*Black Dot:*

Indicates a closed ancestor, e.g. the commit was directly pushed into the repository bypassing code review, or the ancestor change was reviewed and submitted on another branch. The latter may indicate that the user has accidentally pushed the commit to the wrong branch, e.g. the commit was done on branch-a, but was then pushed to refs/for/branch-b. A black dot is also present if the change was abandoned.

*Strikethrough Subject:*

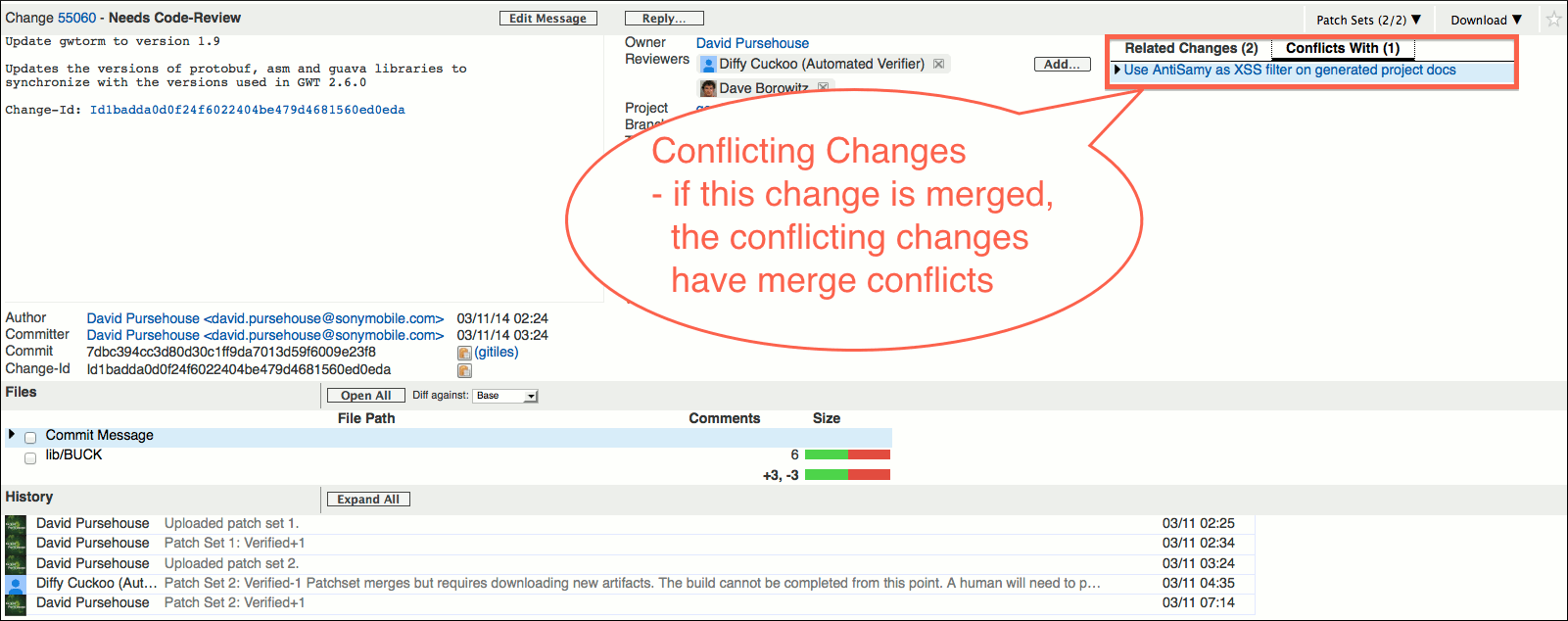
When the commit is abandoned, its subject line will be striked through.



*Conflicts With:*

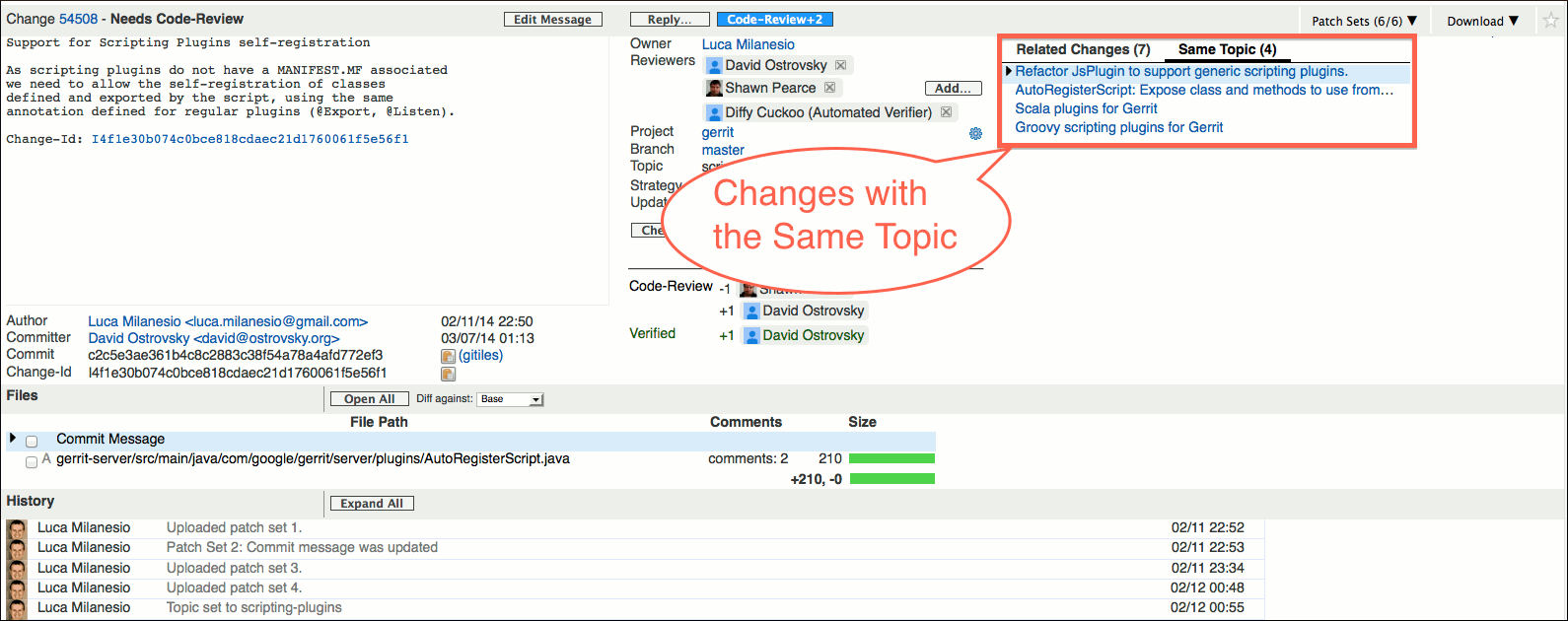
This tab page shows changes that conflict with the current change. Non-mergeable changes are filtered out; only conflicting changes that are mergeable are shown.

If this change is merged, its conflicting changes will have merge conflicts and must be rebased. The rebase of the other changes with the conflict resolution must then be done manually.



*Same Topic:*

This tab page shows changes that have the same topic as the current change. Only open changes are included in the list.



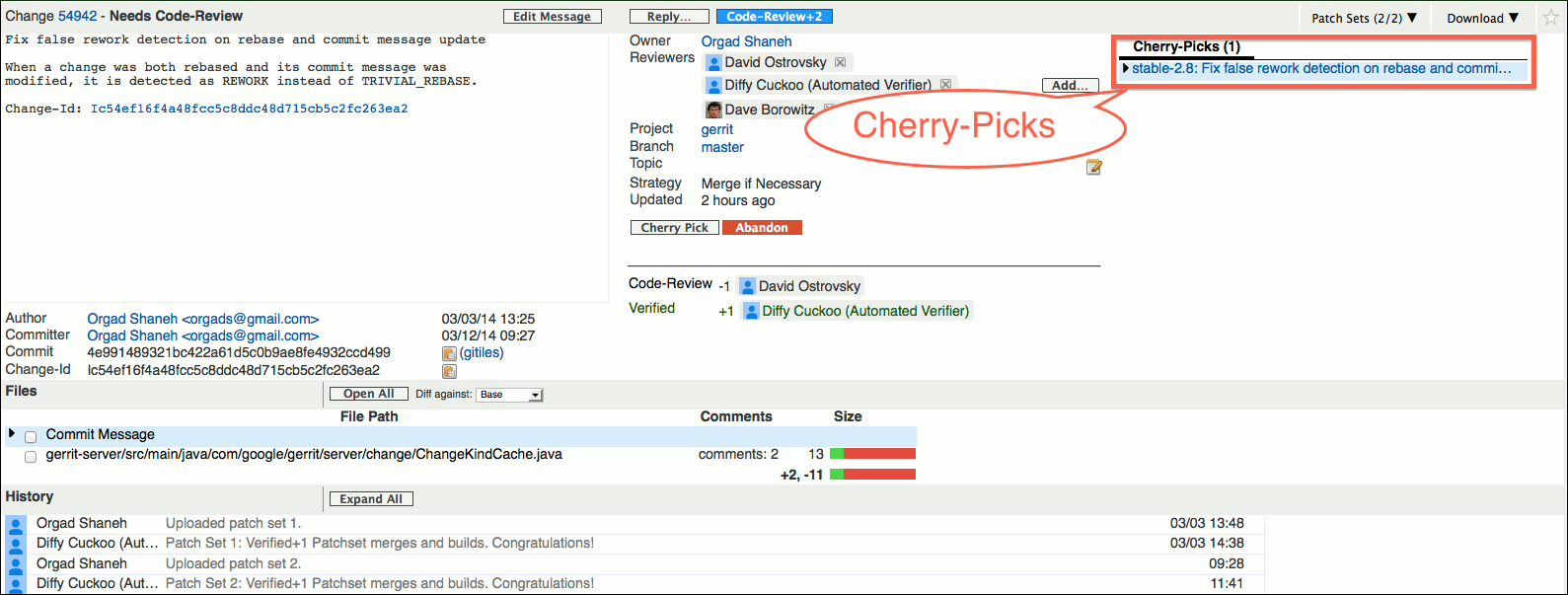
*Submitted Together:*

This tab page shows changes that will be submitted together with the currently viewed change, when clicking the submit button. It includes ancestors of the current patch set.

This may include changes and its ancestors with the same topic if change.submitWholeTopic is enabled. Only open changes with the same topic are included in the list.

*Cherry-Picks:*

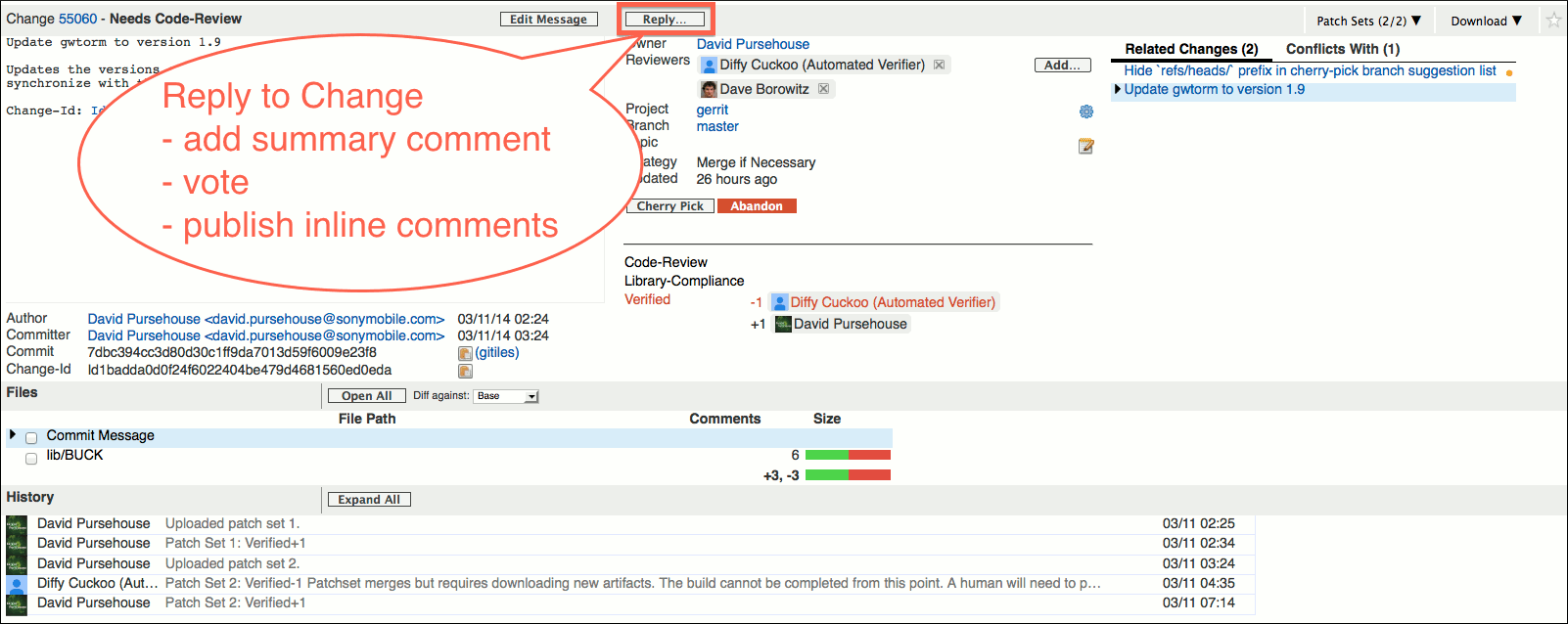
This tab page shows changes with the same Change-Id for the current project. Abandoned changes are filtered out. For each change in this list the destination branch is shown as a prefix in front of the change subject.



If there are no related changes for a tab, the tab is not displayed.

### Reply

The Reply…​ button in the change header allows to reply to the currently viewed patch set; one can add a summary comment, publish inline draft comments, and vote on the labels.

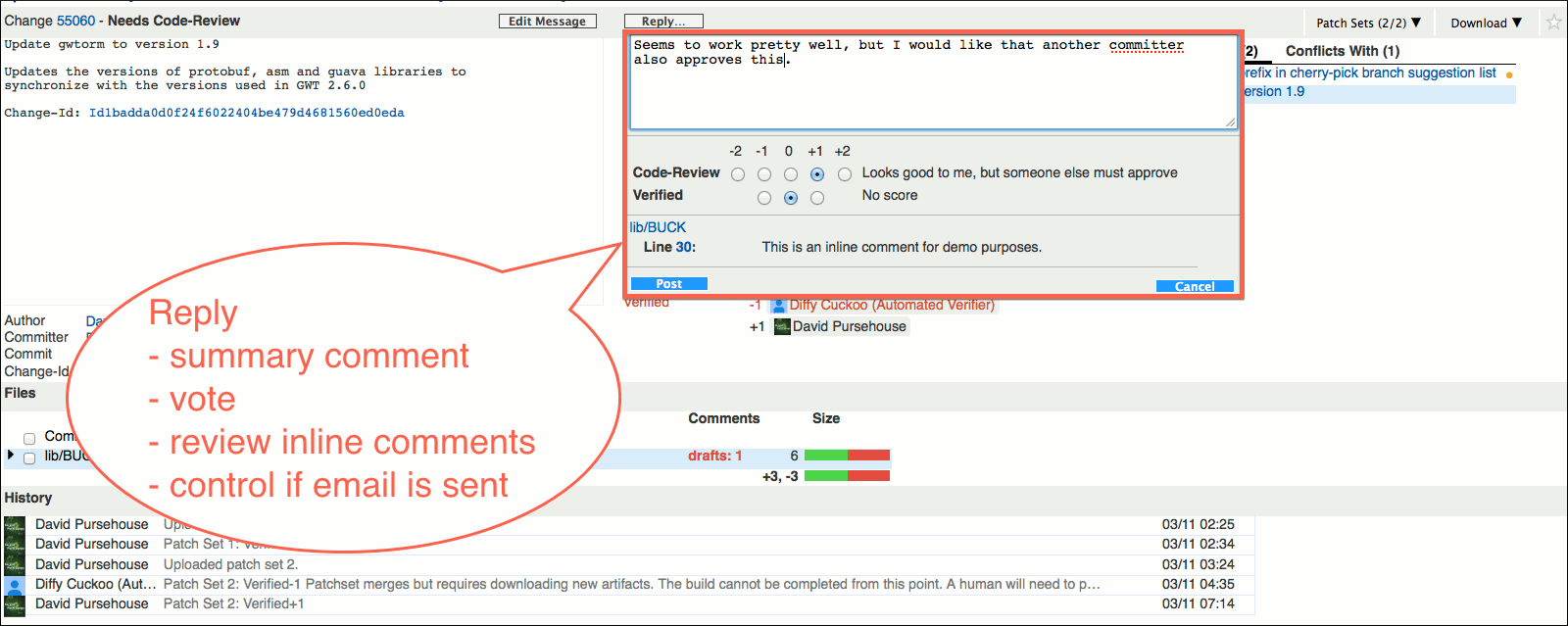


Clicking on the Reply…​ button opens a popup panel. A text box allows to type a summary comment for the currently viewed patch set.

If the current patch set is viewed, radio buttons are displayed for each label on which the user is allowed to vote. Voting on non-current patch sets is not possible.

The inline draft comments that will be published are displayed in a separate section so that they can be reviewed before publishing. There are links to navigate to the inline comments which can be used if a comment needs to be edited.

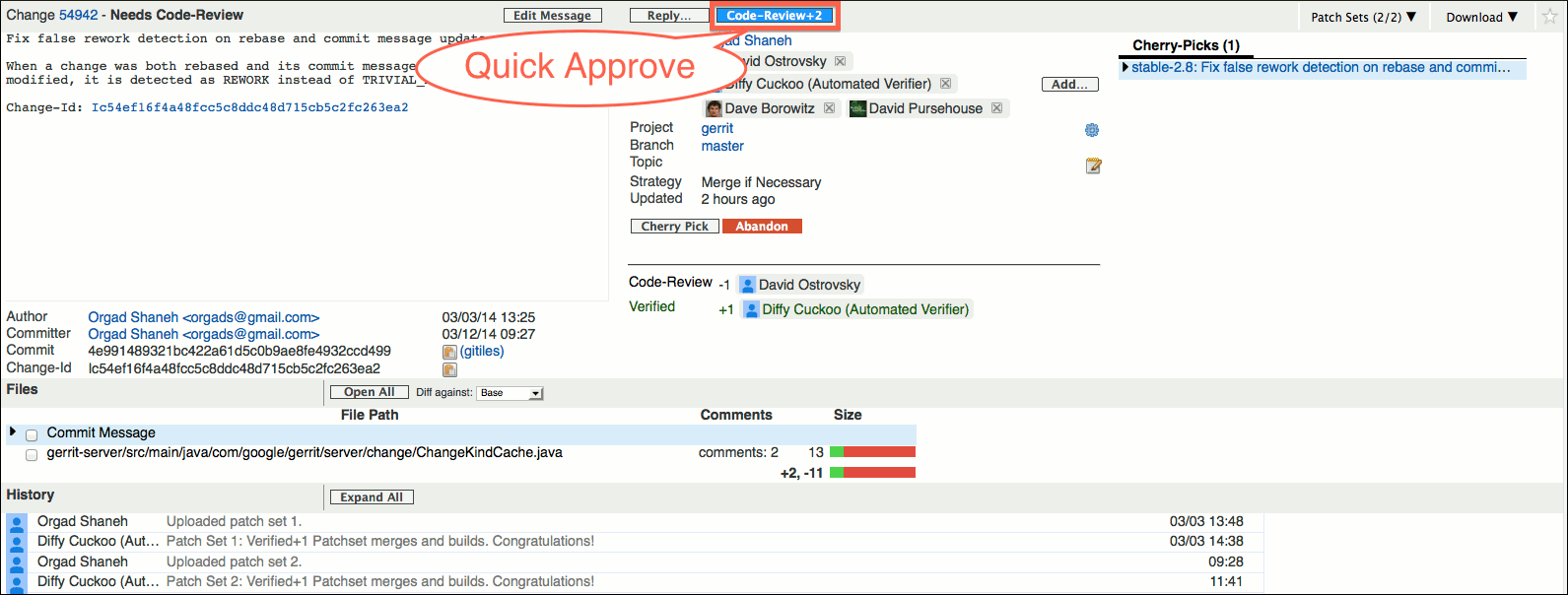
The Post button publishes the comments and the votes.



If a user can approve a label that is still required, a quick approve button appears in the change header that allows to add this missing approval by a single click. The quick approve button only appears if there is a single label that is still required and can be approved by the user.

E.g. if a change requires approvals on the 'Code-Review' and the 'Verified' labels, and there is already a '+1 Verified' vote, then if the user is allowed to vote the max score on 'Code-Review', a Code-Review+2 quick approve button appears that approves the 'Code-Review' label if clicked.

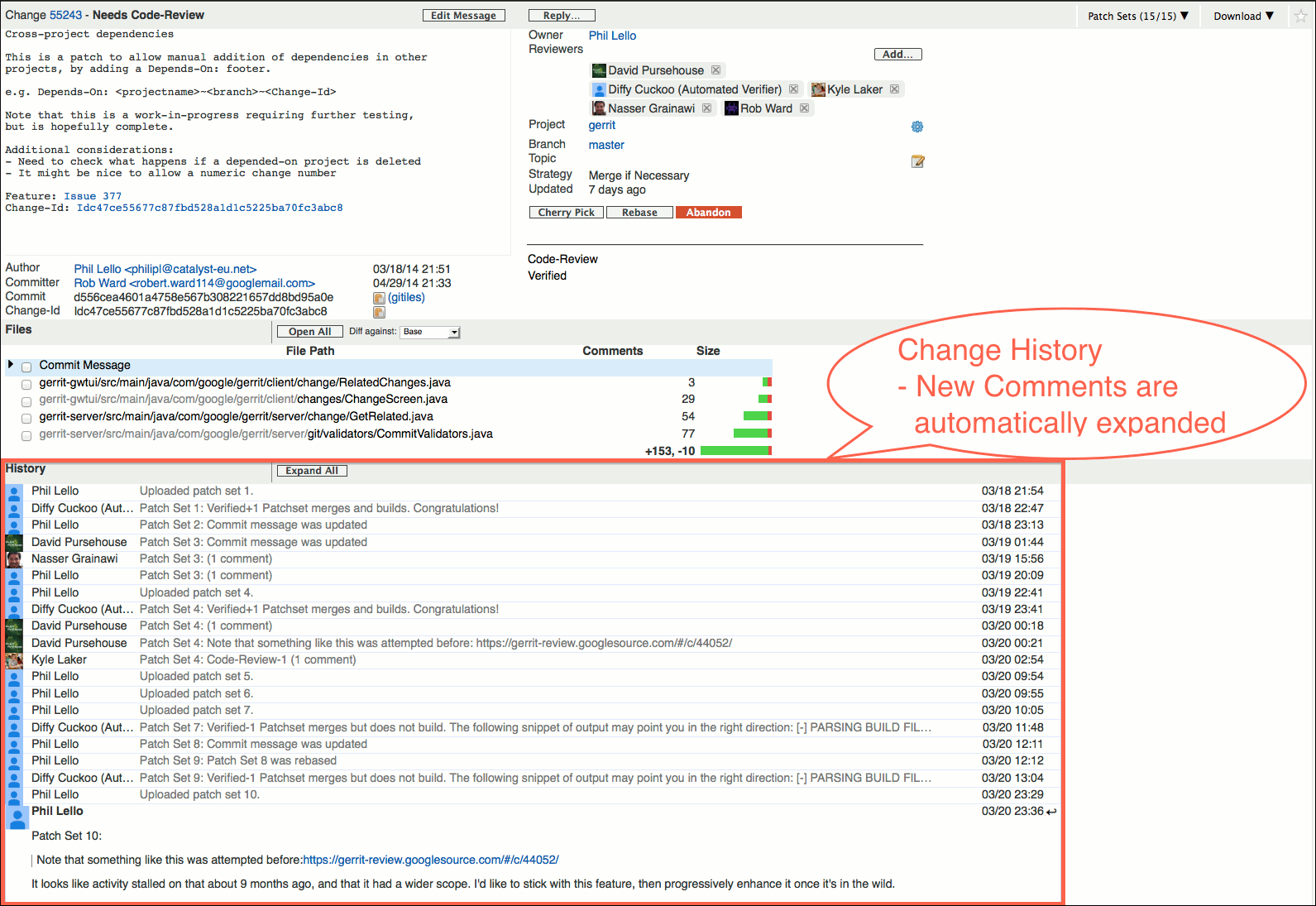
Using the quick approve button also publishes all inline draft comments; a summary comment is only added if the reply popup panel is open when the quick approve button is clicked.



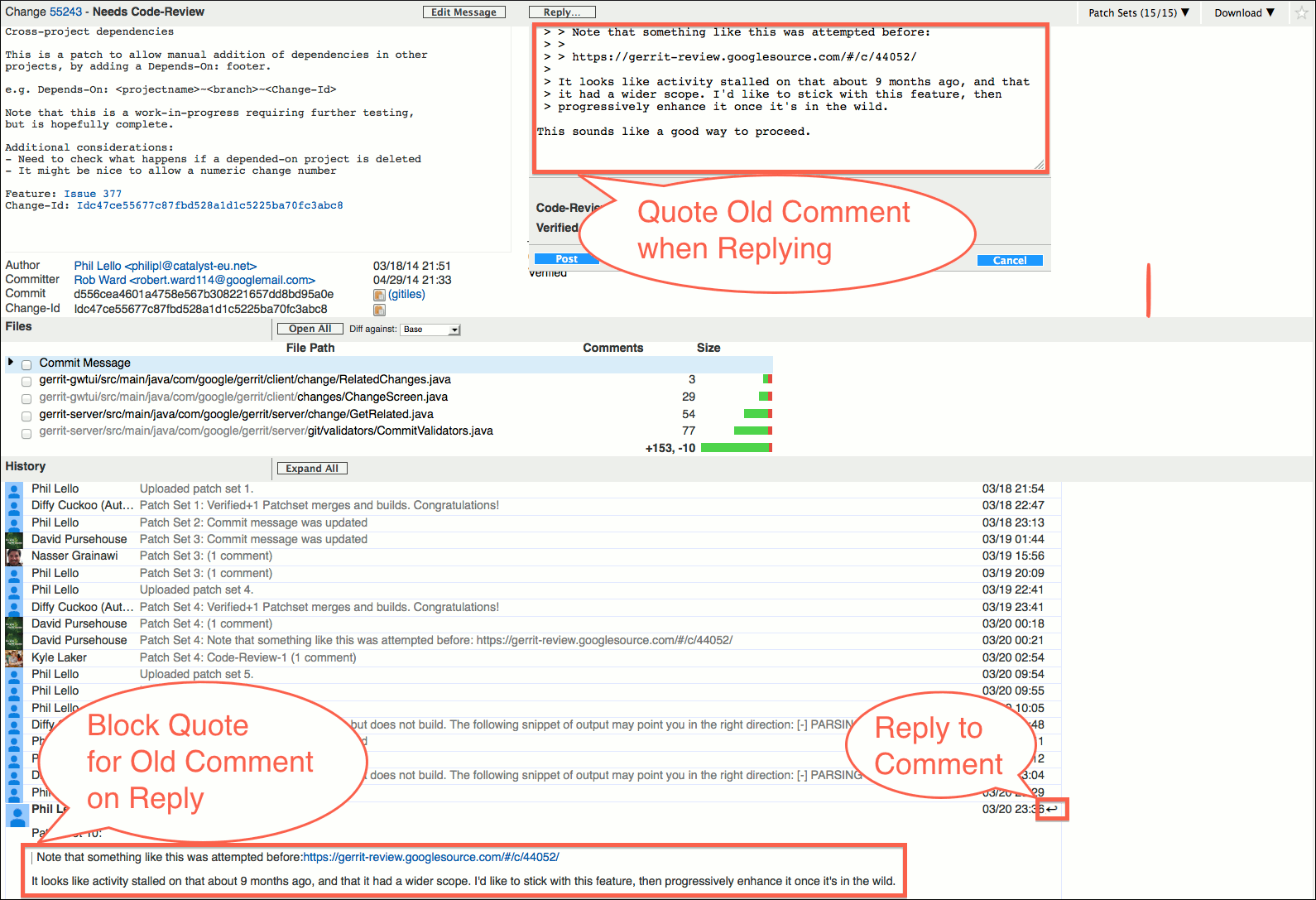
### History

The history of the change can be seen in the lower part of the screen. The history contains messages for all kinds of change updates, e.g. a message is added when a new patch set is uploaded or when a review was done.

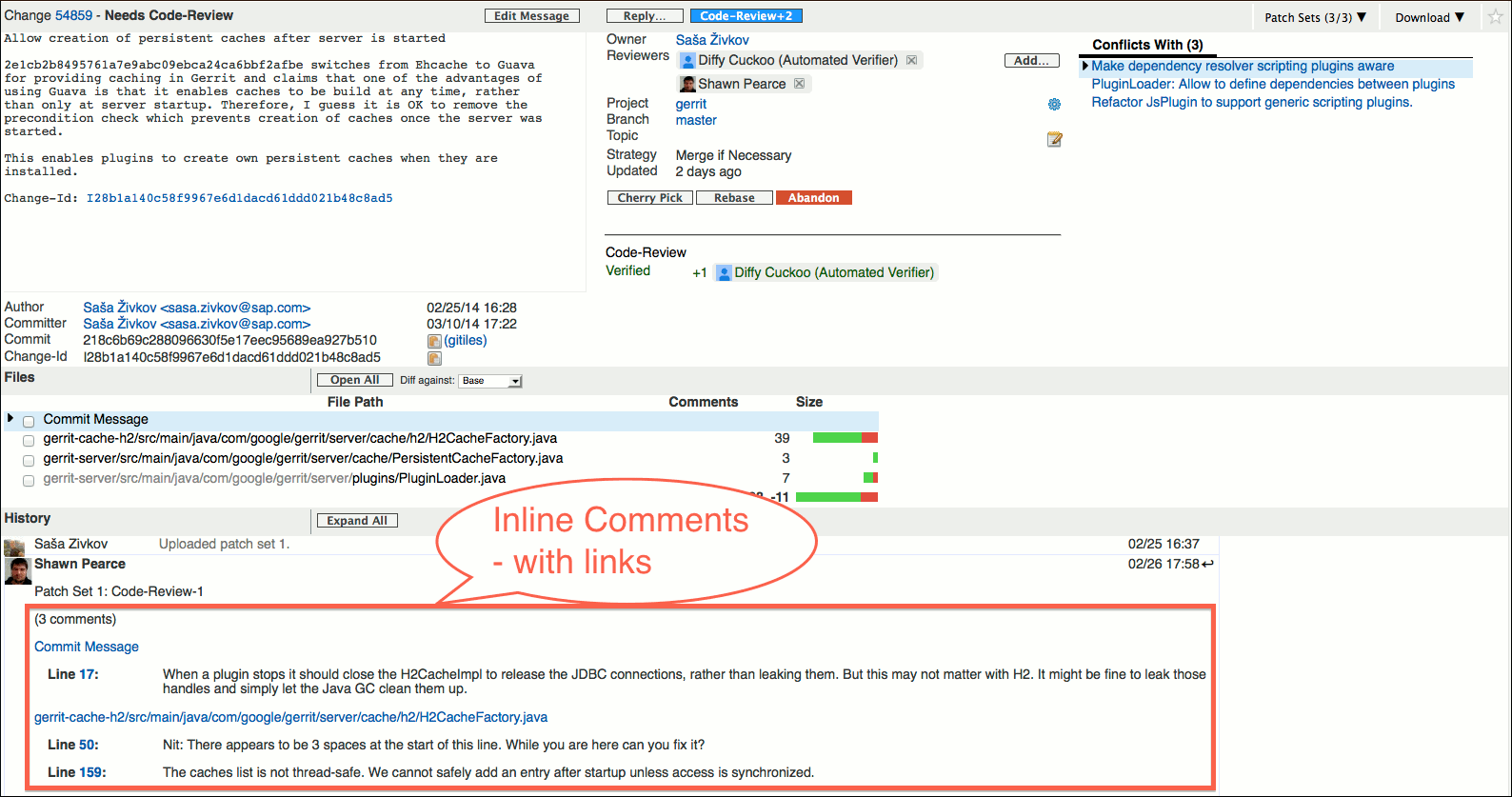
Messages with new comments from other users, that were published after the current user last reviewed this change, are automatically expanded.



It is possible to directly reply to a change message by clicking on the reply icon in the right upper corner of a change message. This opens the reply popup panel and prefills the text box with the quoted comment. Then the reply can be written below the quoted comment or inserted inline. Lines starting with " > " will be rendered as a block quote. Please note that for a correct rendering it is important to leave a blank line between a quoted block and the reply to it.



Inline comments are directly displayed in the change history and there are links to navigate to the inline comments.



The Expand All button expands all messages; the Collapse All button collapses all messages.

### Update Notification

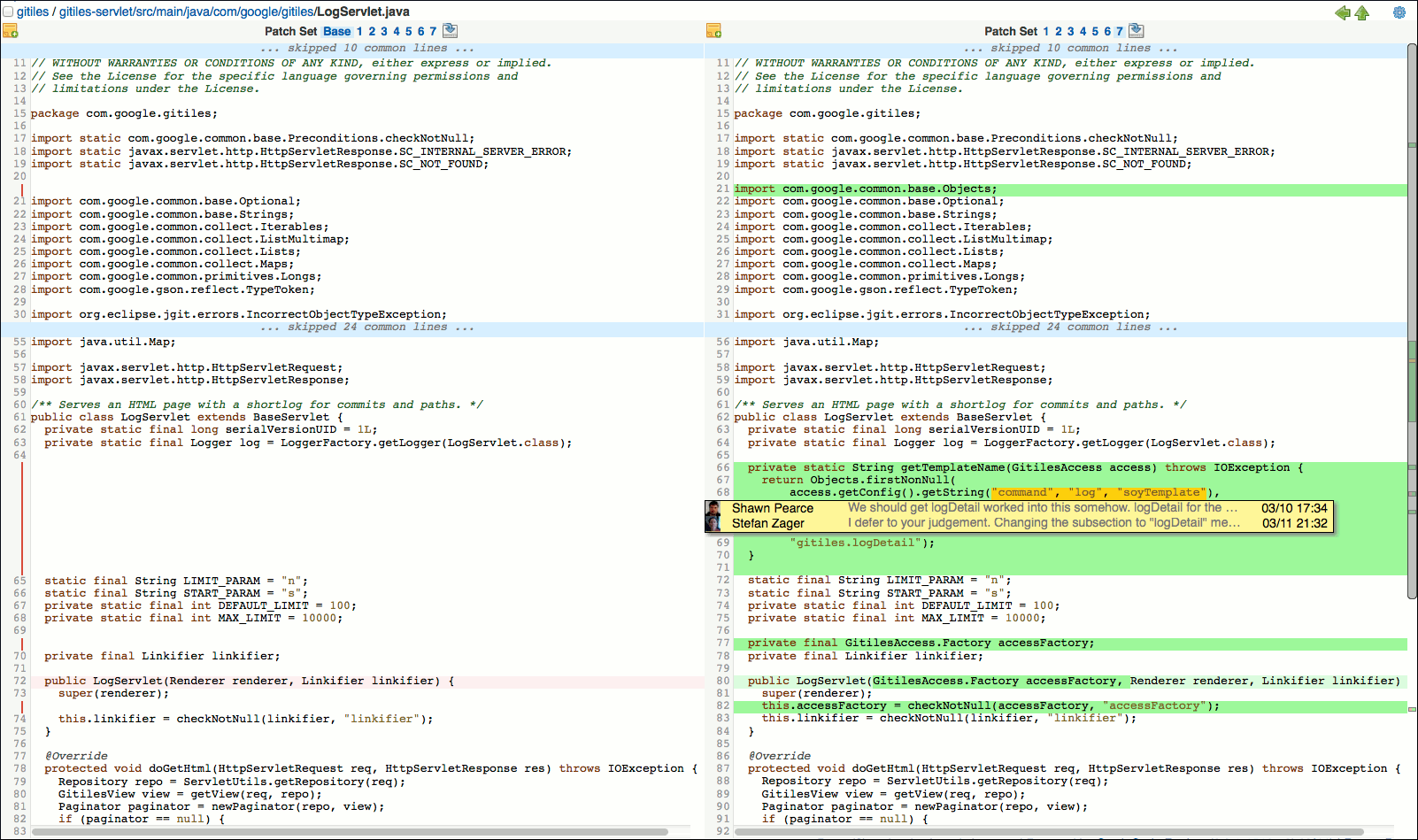
The change screen automatically polls for updates to the currently viewed change. If there is an update the user is informed by a popup panel in the bottom right corner. The polling frequency depends on the server configuration; by default it is 30 seconds. Polling may also be completely disabled by the administrator.



### Side-by-Side Diff Screen

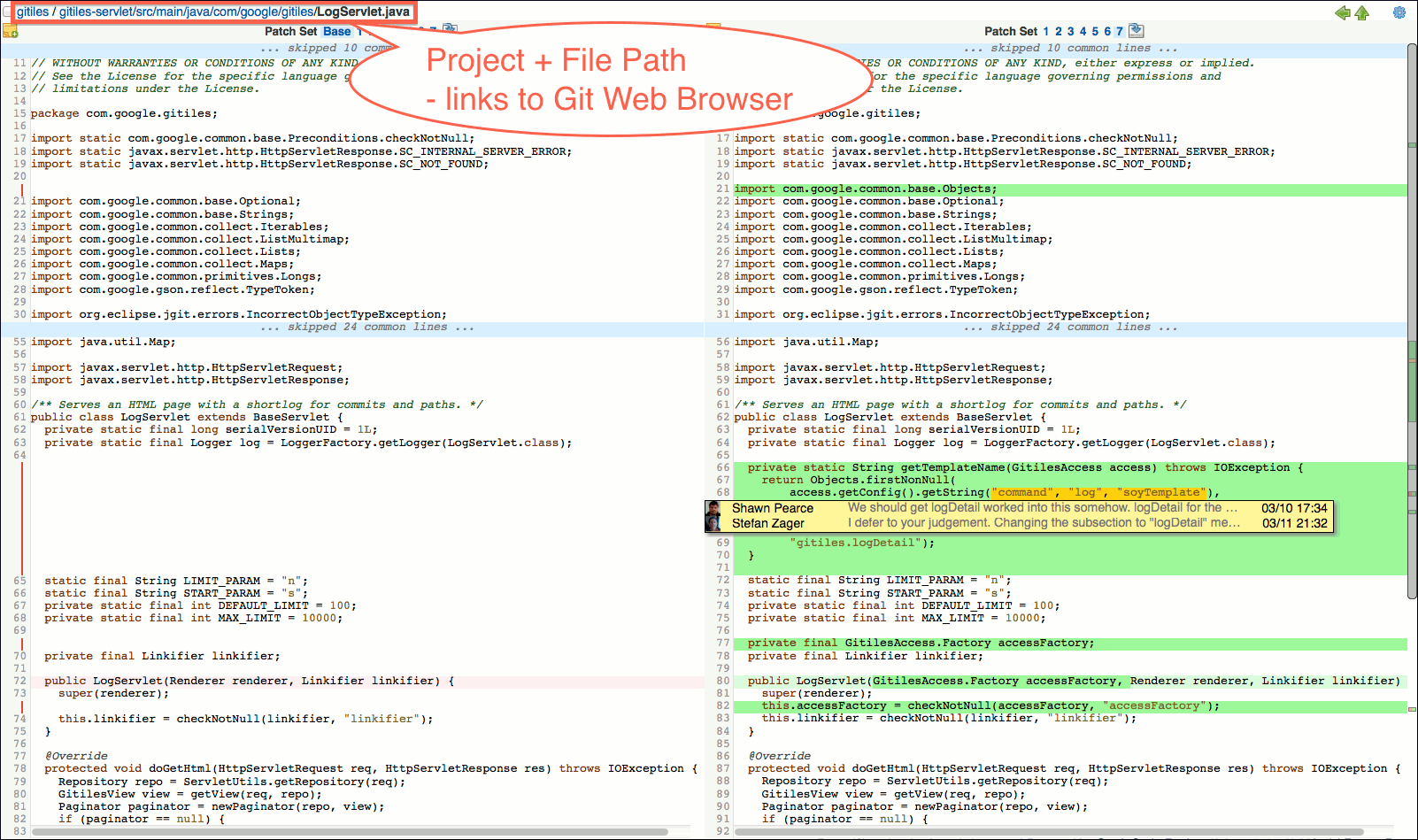
The side-by-side diff screen shows a single patch; the old file version is displayed on the left side of the screen; the new file version is displayed on the right side of the screen.

This screen allows to review a patch and to comment on it.

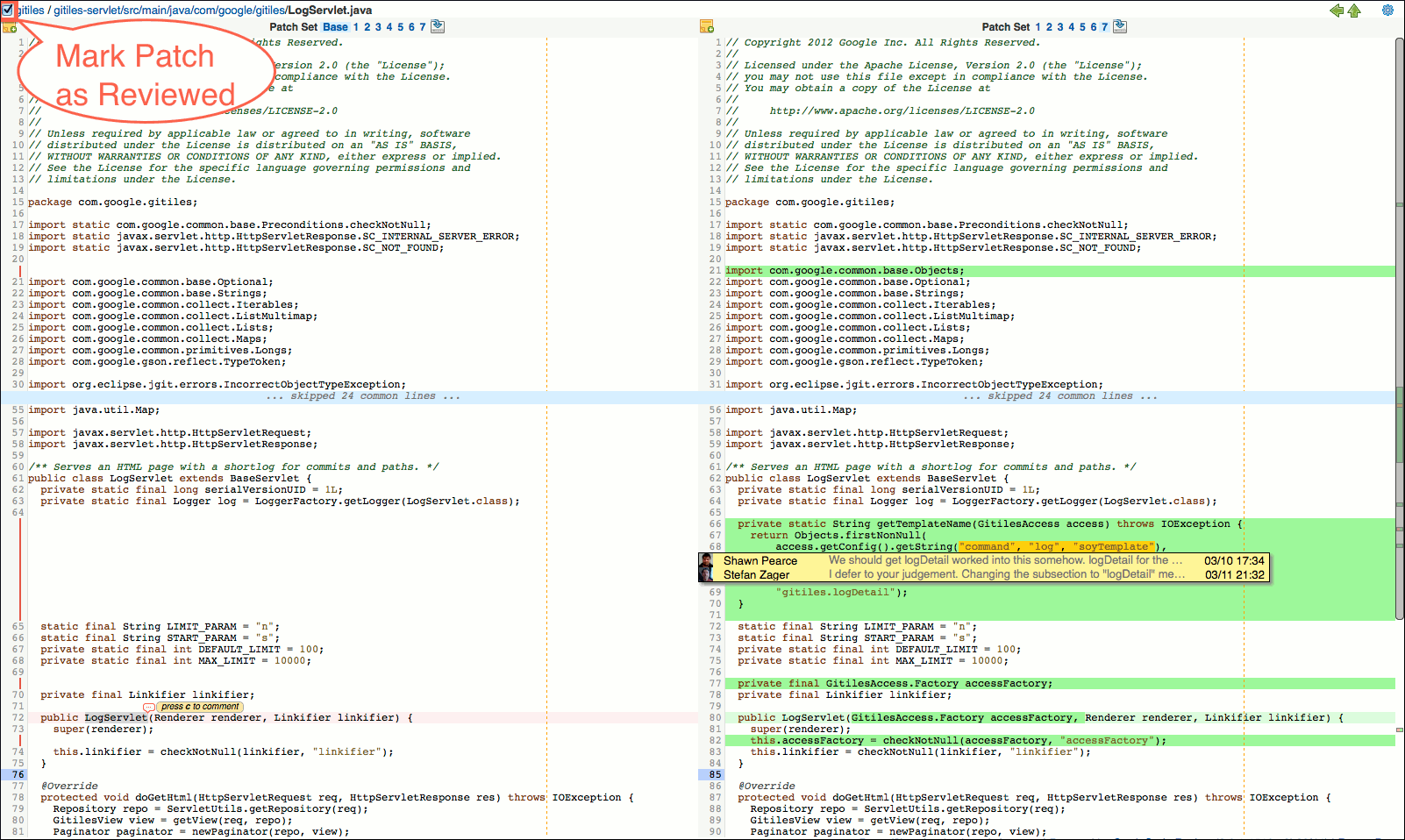


In the screen header the project name and the name of the viewed patch file are shown.

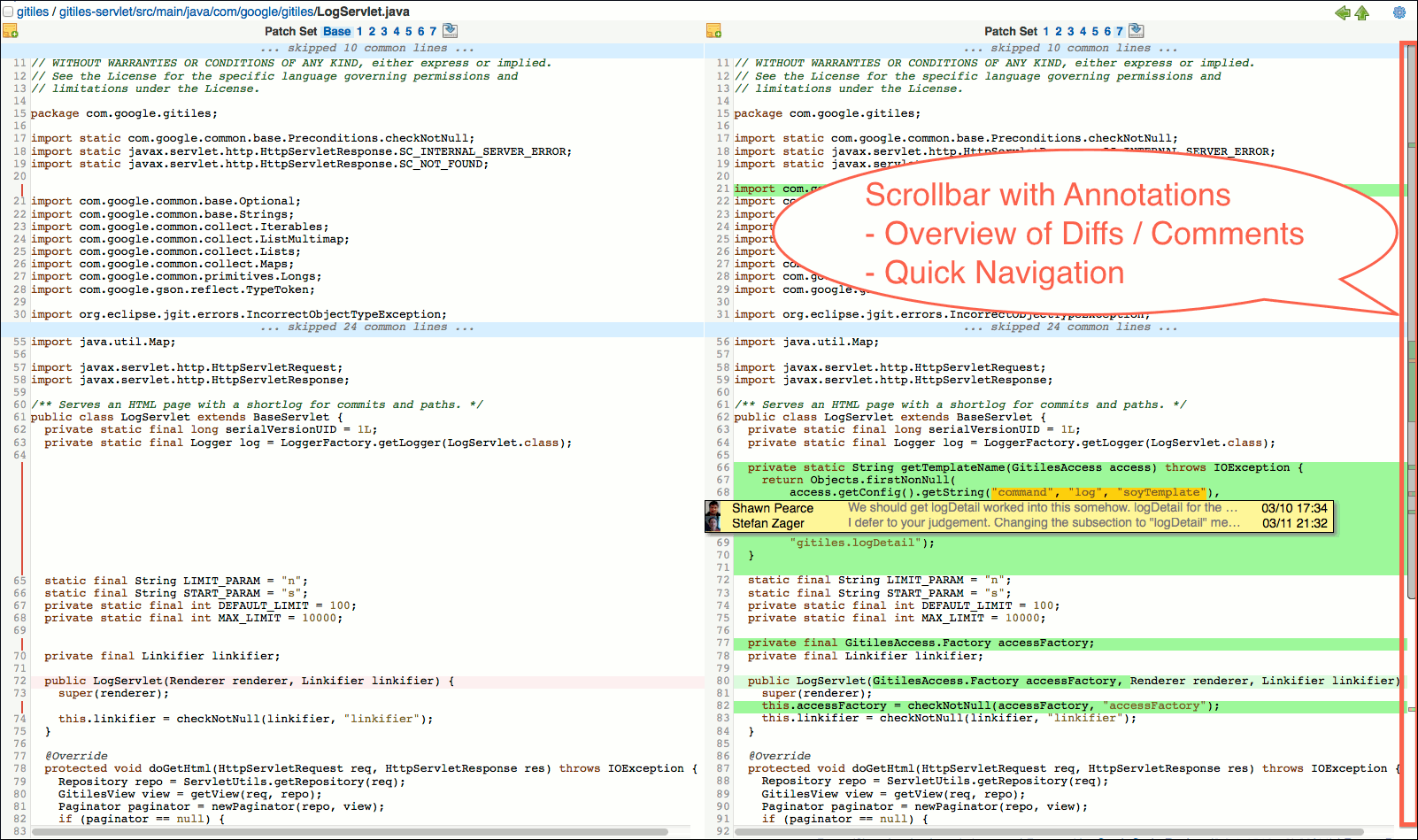
If a Git web browser is configured on the server, the project name and the file path are displayed as links to the project and the folder in the Git web browser.

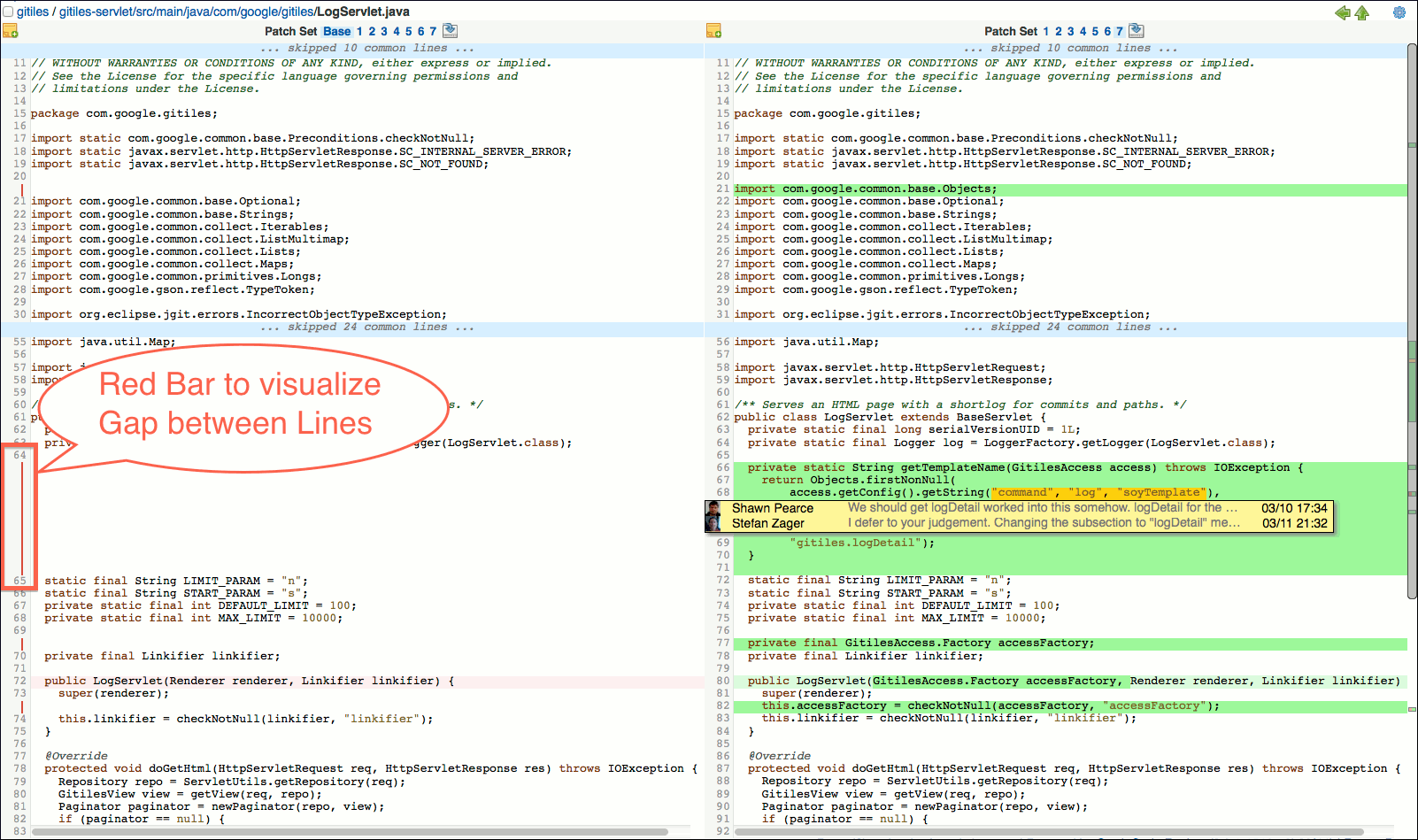


The checkbox in front of the project name and the file name allows the patch to be marked as reviewed. The Mark Reviewed diff preference allows to control whether the files should be automatically marked as reviewed when they are viewed.



The scrollbar shows patch diffs and inline comments as annotations. This provides a good overview of the lines in the patch that are relevant for reviewing. By clicking on an annotation one can quickly navigate to the corresponding line in the patch.

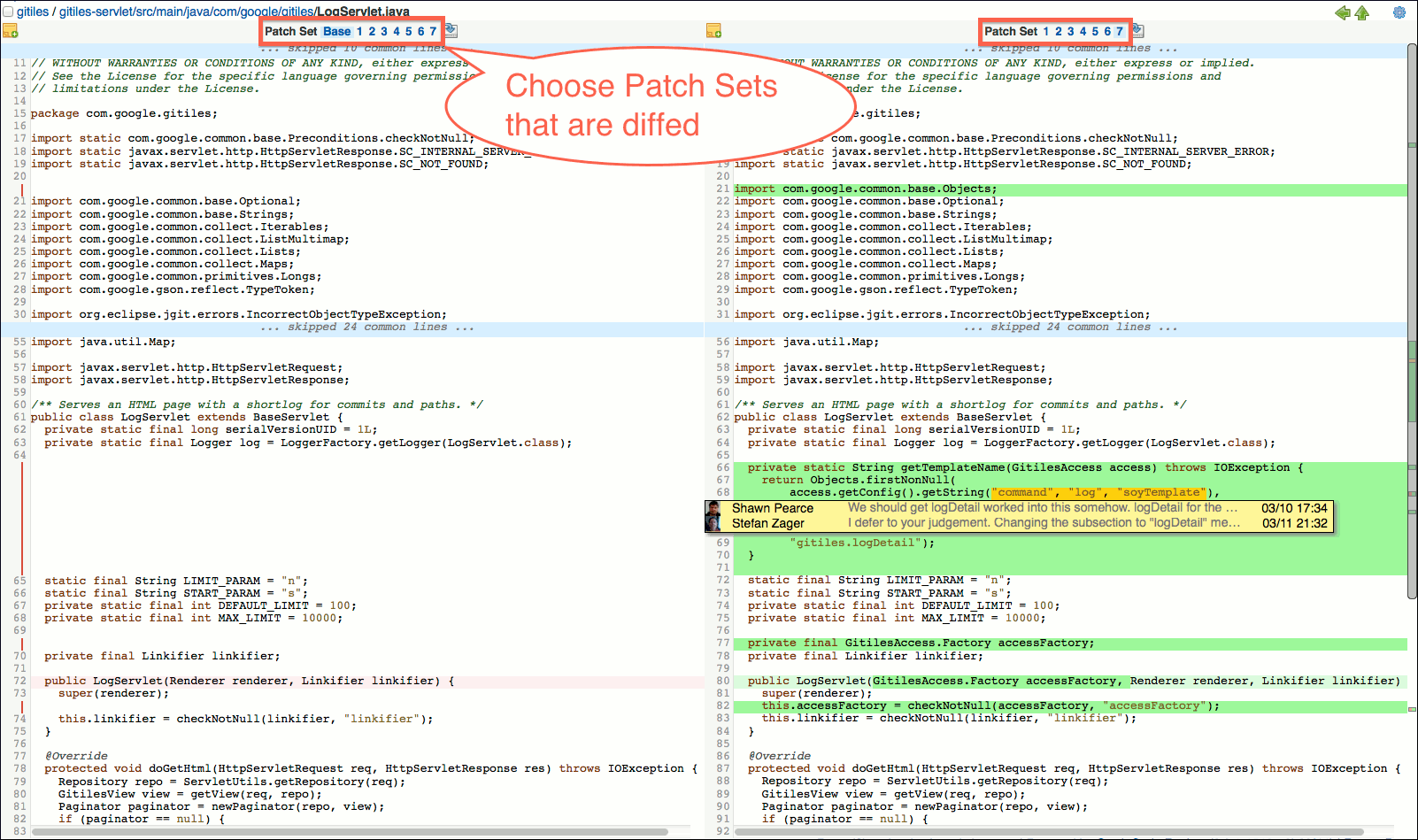


A gap between lines in the file content that is caused by aligning the left and right side or by displaying inline comments is shown as a vertical red bar in the line number column. This prevents a gap from being mistaken for blank lines in the file 

In the header, on each side, the list of patch sets is shown. Clicking on a patch set changes the selection for the patch set comparison and the screen is refreshed to show the diff between the selected patch sets. The currently selected patch set is highlighted by a light blue background.

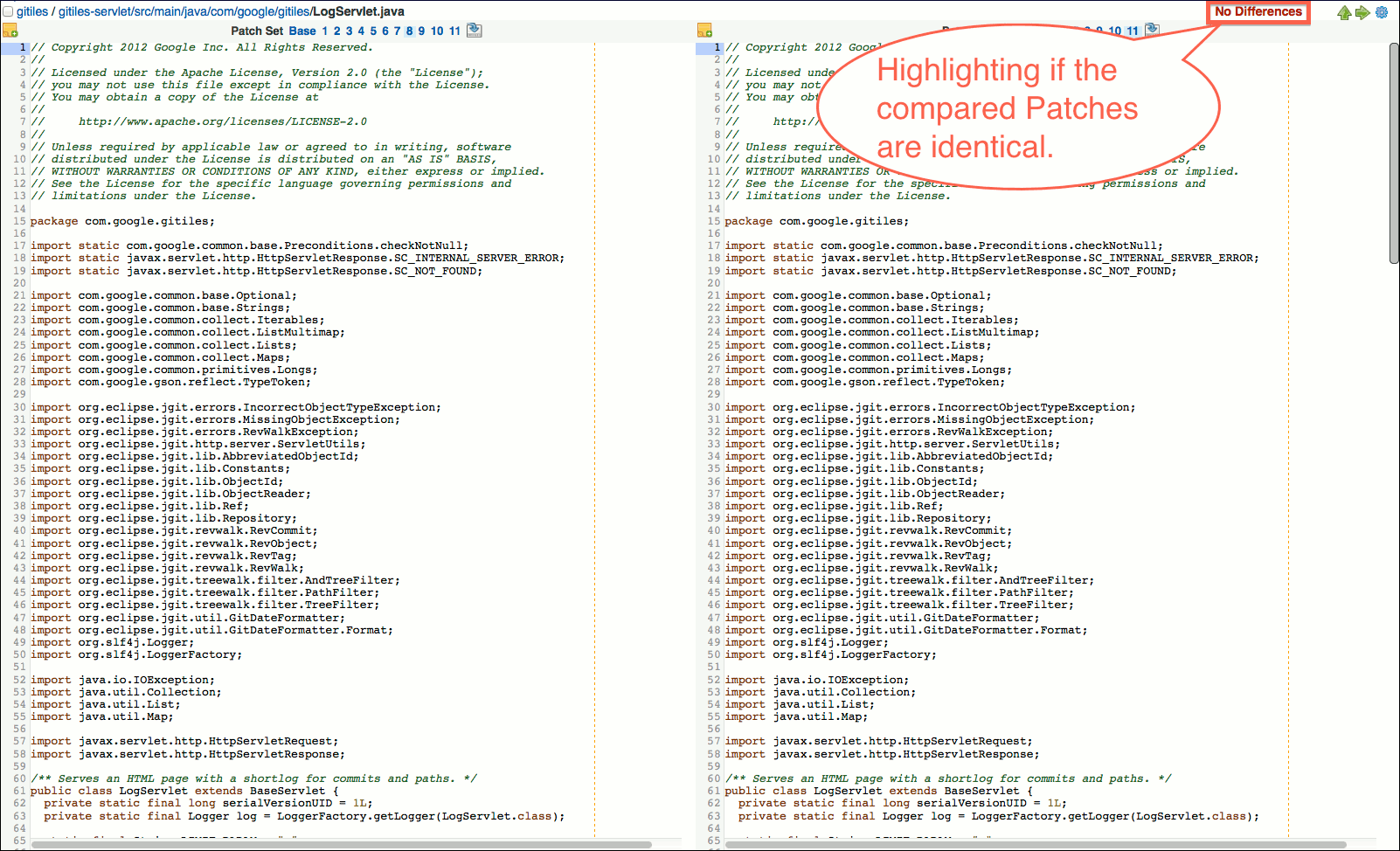
On the left side Base can be selected to compare a patch set against its base. For merge commits Auto Merge is available instead which allows to compare the patch against the result of the auto merge. The auto merge version may contain Git conflict markers and is useful for reviewing how conflicts are resolved by a patch.

Reviewers that are reviewing a patch for the first time look at its diff against its base; reviewers that have reviewed an old patch version before, may see what has changed since that version by comparing the old patch against the current patch.

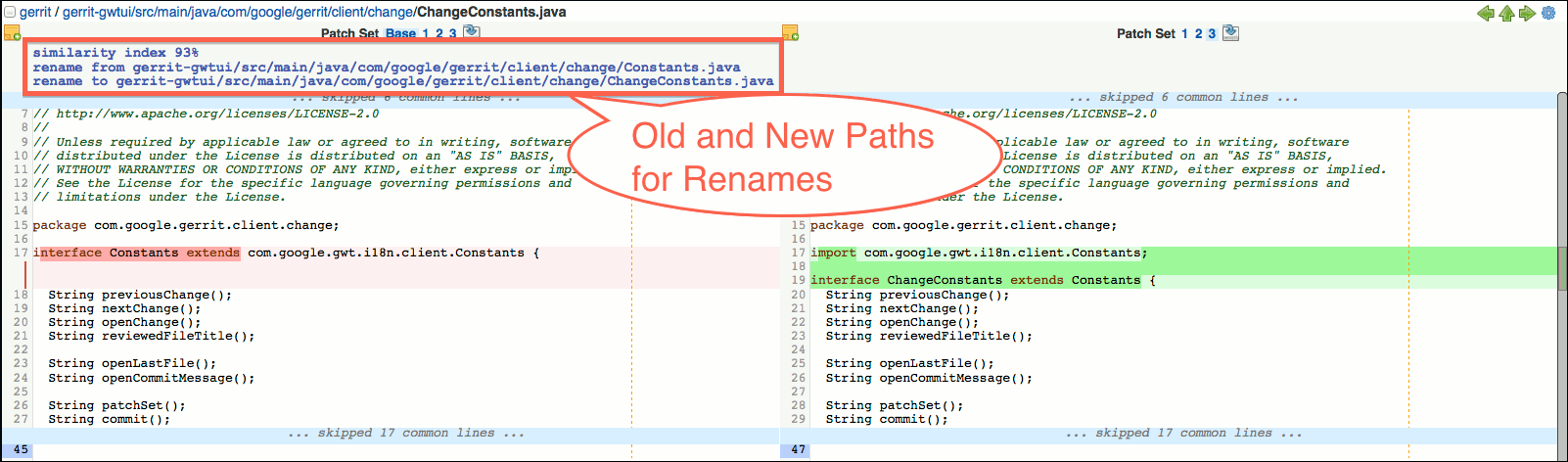


The download icon next to the patch set list allows to download the patch. Unless the mime type of the file is configured as safe, the download file is a zip archive that contains the patch file.

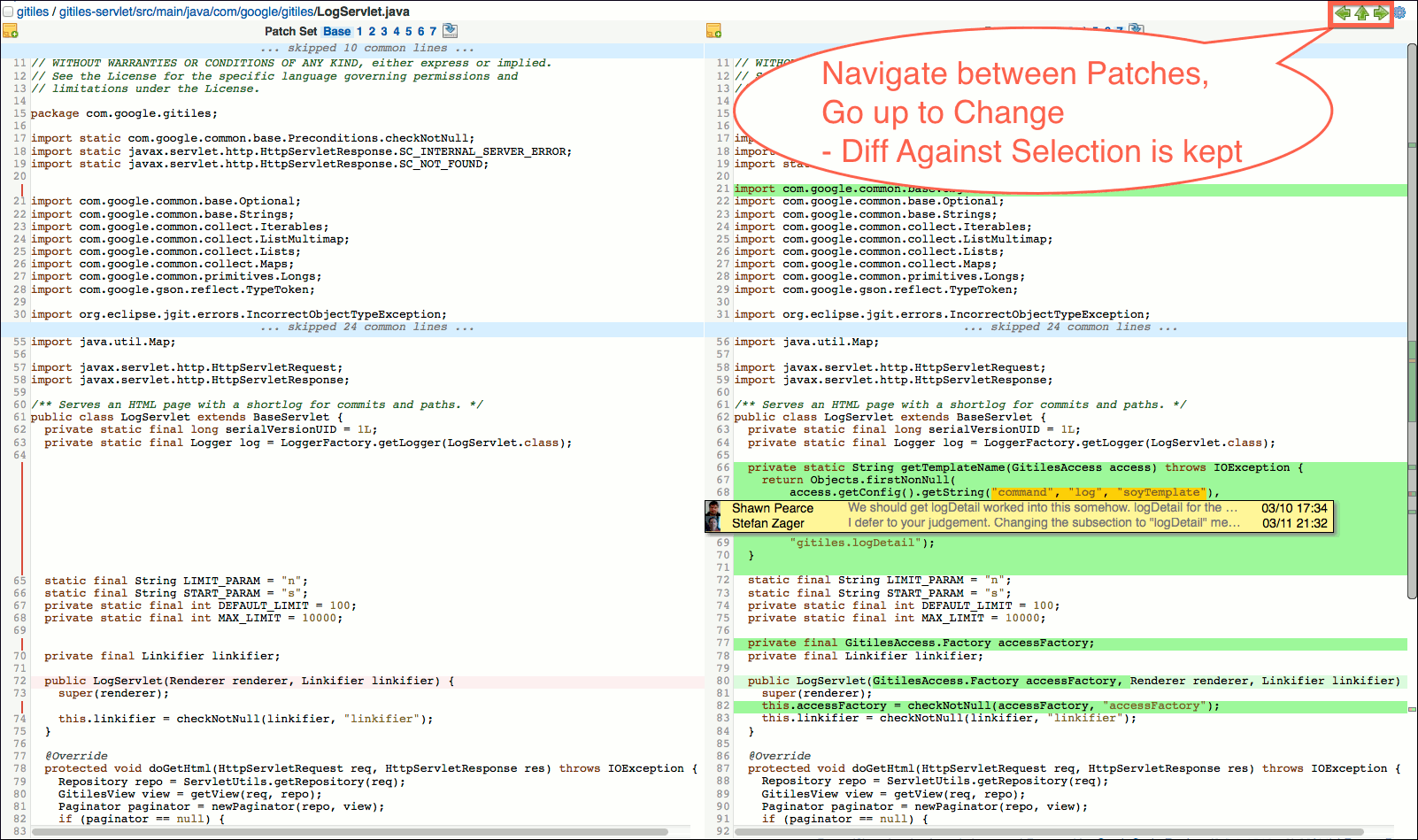
If the compared patches are identical, this is highlighted by a red No Differences label in the screen header.



If a file was renamed, the old and new file paths are shown in the header together with a similarity index that shows how much of the file content is unmodified.



For navigating between the patches in a patch set there are navigation buttons on the right side of the screen header. The left arrow button navigates to the previous patch; the right arrow button navigates to the next patch. The arrow up button leads back to the change screen. In all cases the selection for the patch set comparison is kept.

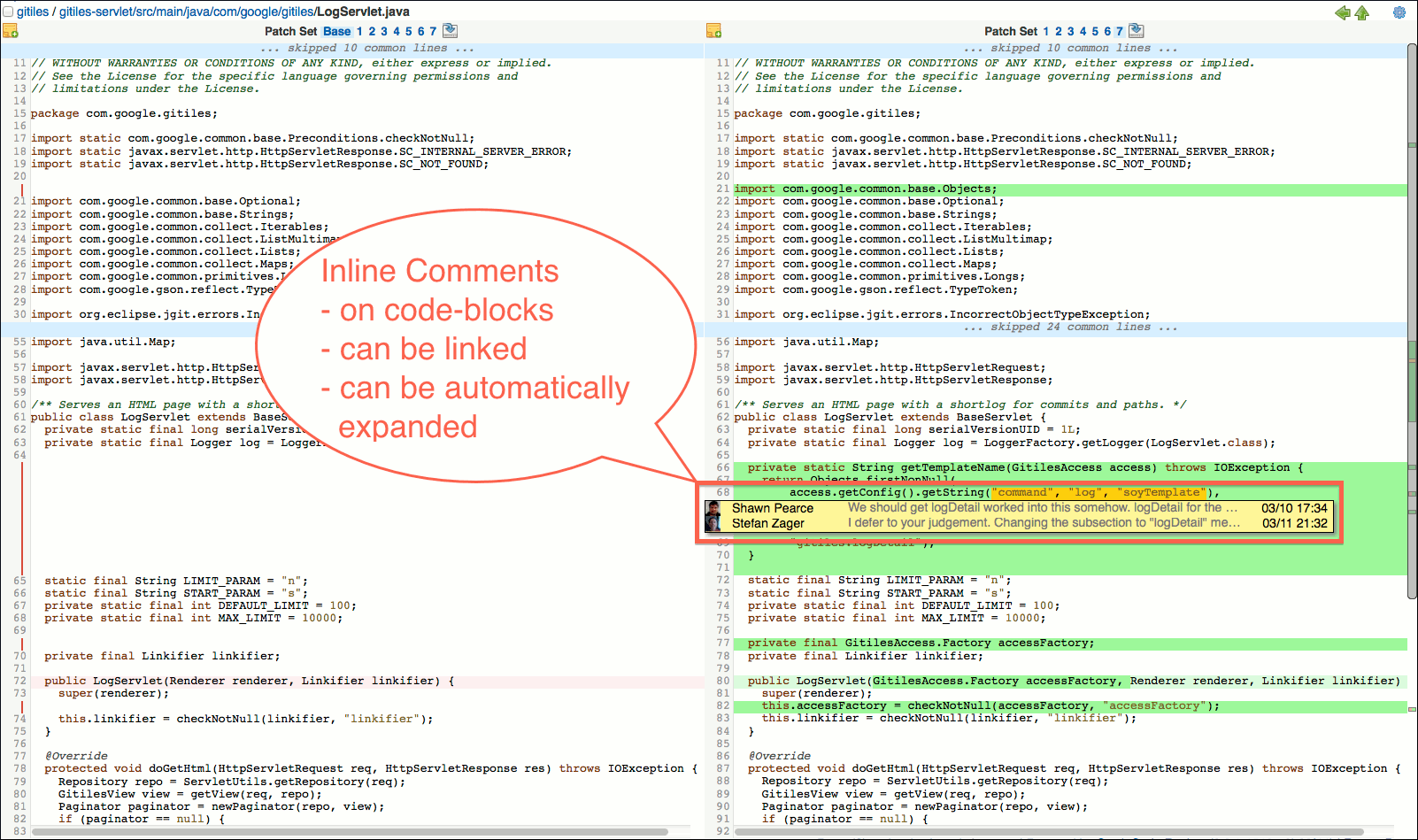


### Inline Comments

Inline comments are displayed directly in the patch file under the code that is commented. Inline comments can be placed on lines or on code blocks.

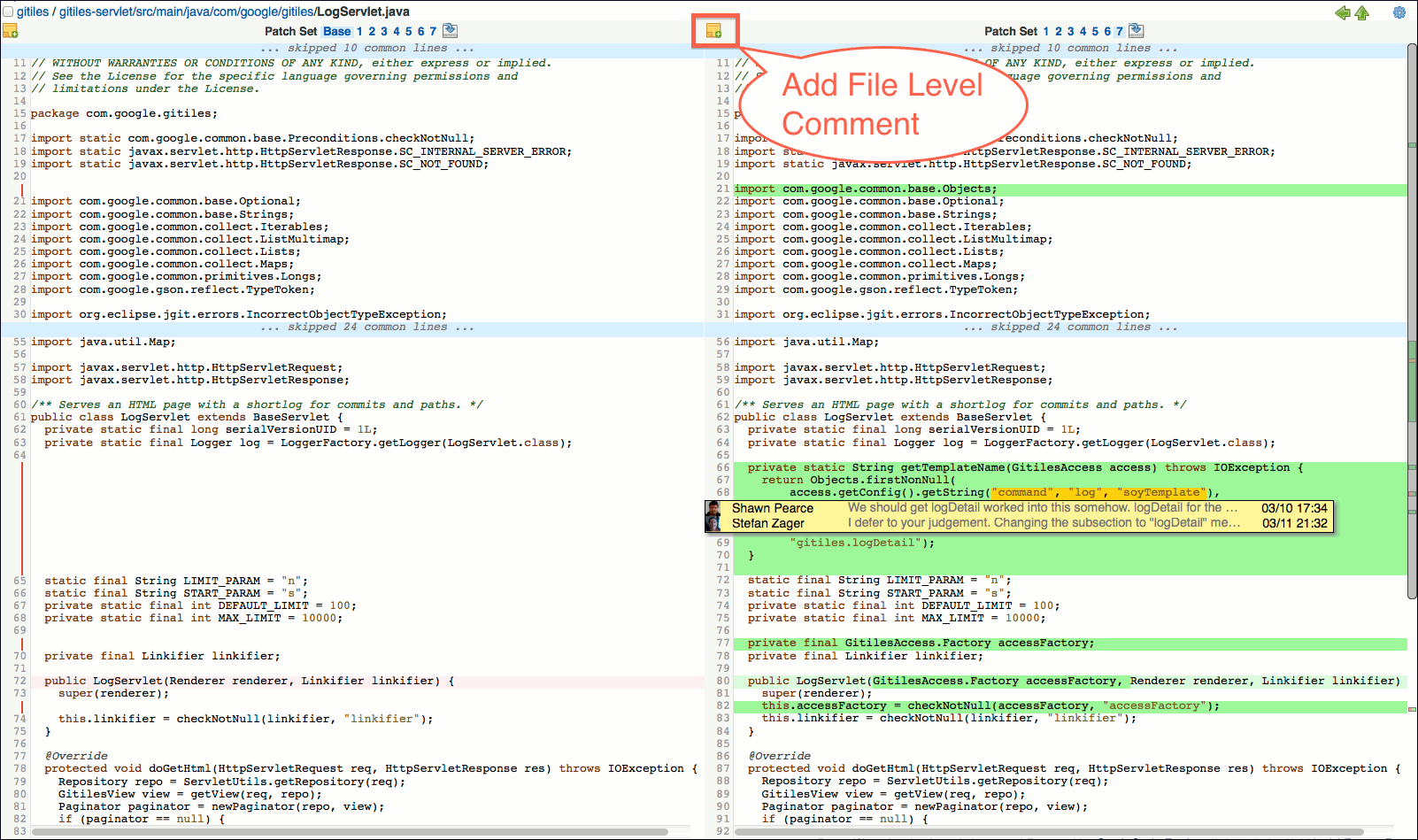
If an inline comment relates to a code block, this code block is highlighted by a yellow background.

Code blocks with comments may overlap. This means it is possible to attach several comments to the same code.

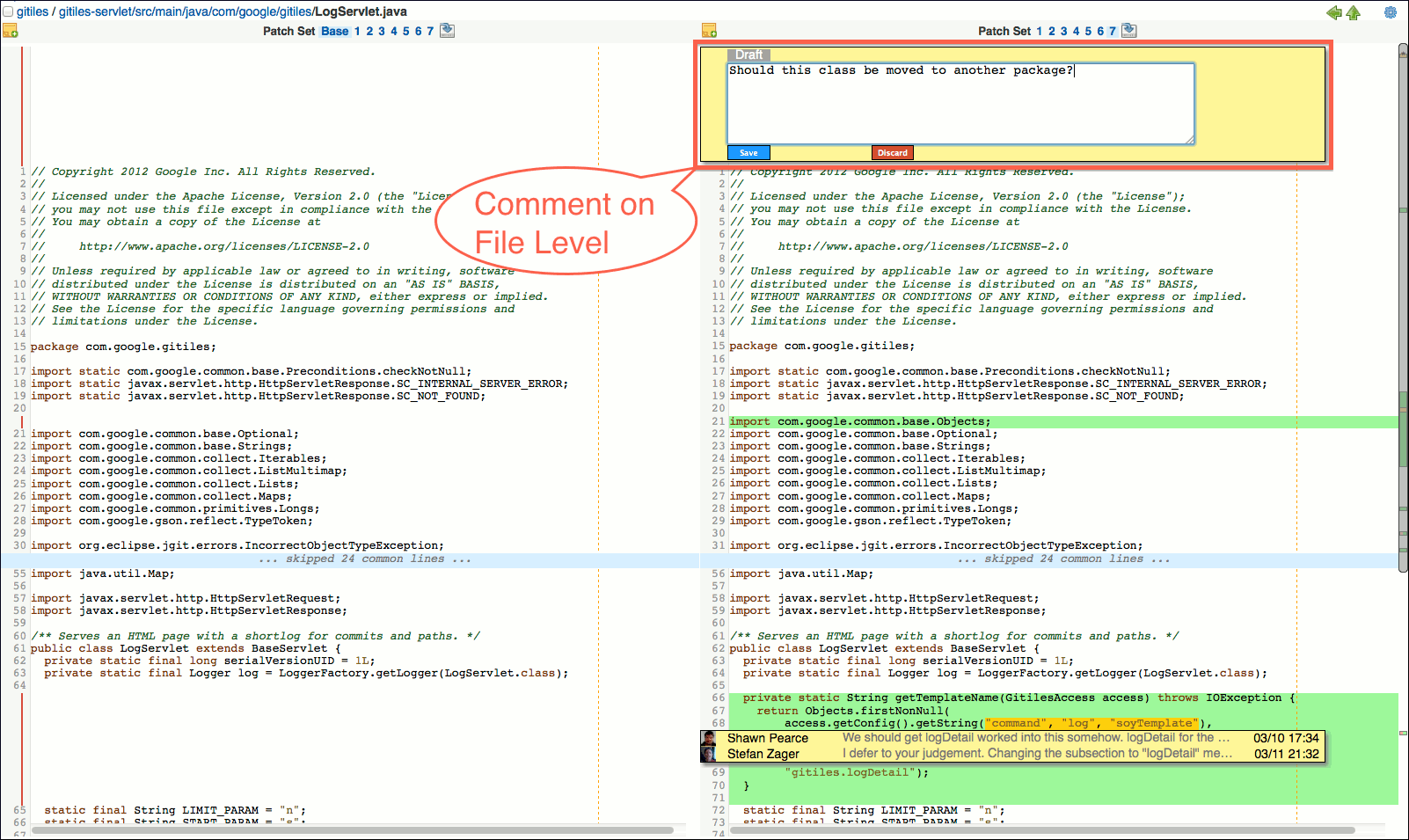


### File Level Comments

Comments that apply to a whole file can be added on file level. File level comments are added by clicking on the comment icon in the header above the file.

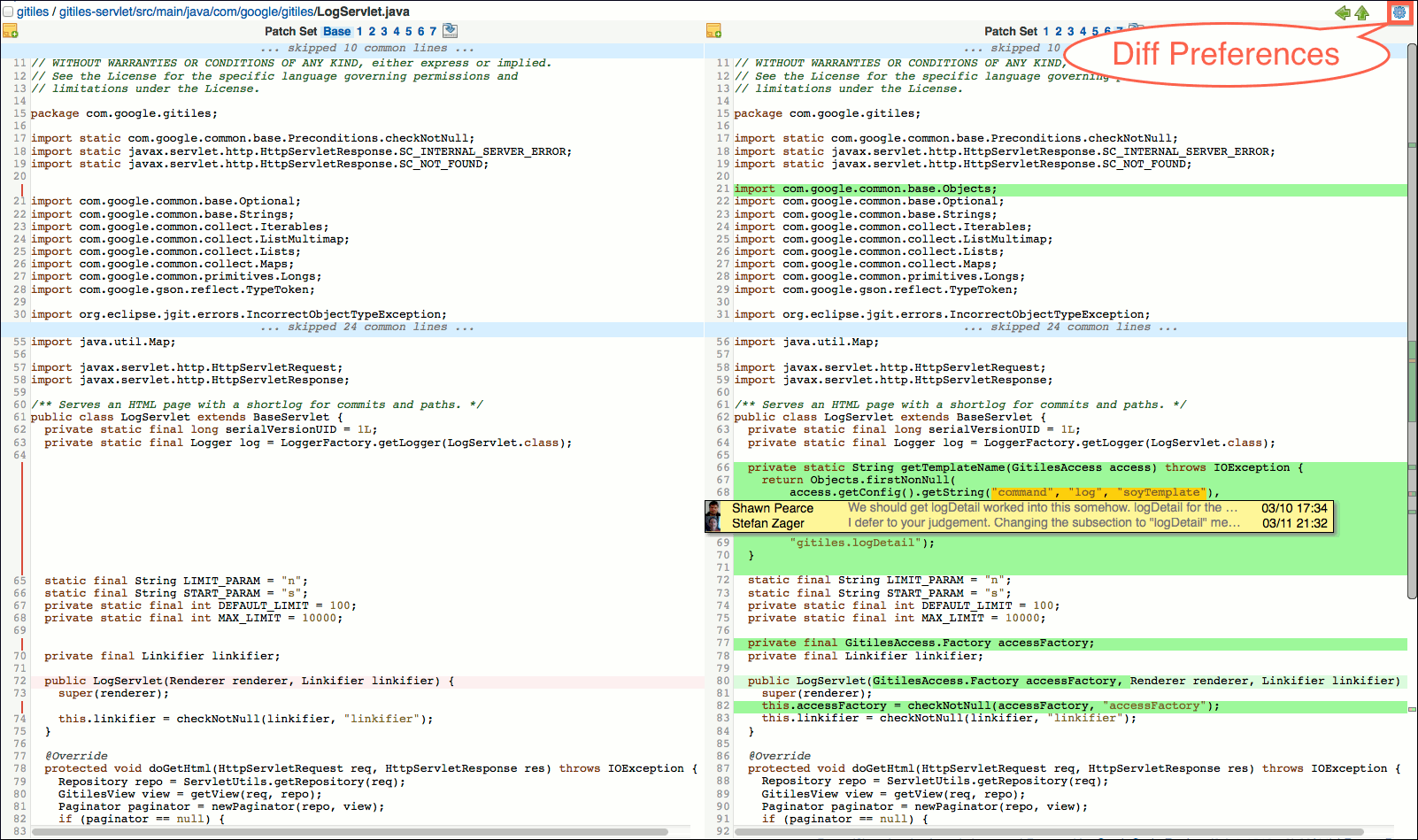


Clicking on the comment icon opens a comment box for typing the file level comment.

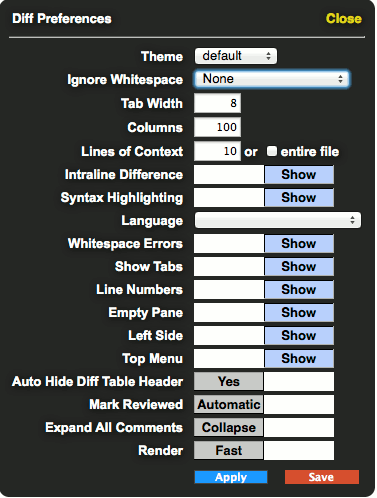


### Diff Preferences

There are several options to control how patch diffs should be rendered. Users can configure their preferences in the diff preferences. The diff preferences can be accessed by clicking on the settings icon in the screen header.



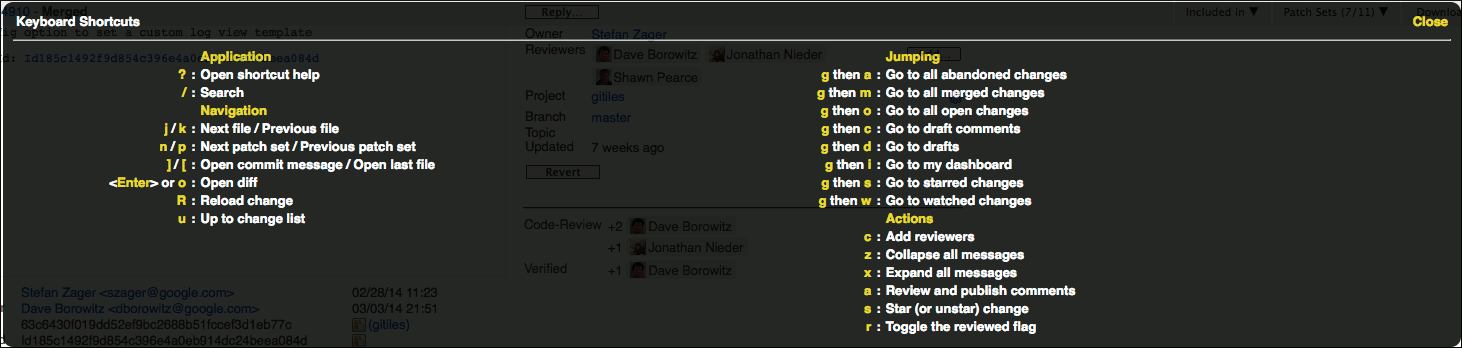
The diff preferences popup allows to change the diff preferences. By clicking on the Save button changes to the diff preferences are saved permanently. Clicking on the Apply button applies the new diff preferences to the current screen, but they are discarded when the screen is refreshed. The Save button is only available if the user is signed in.



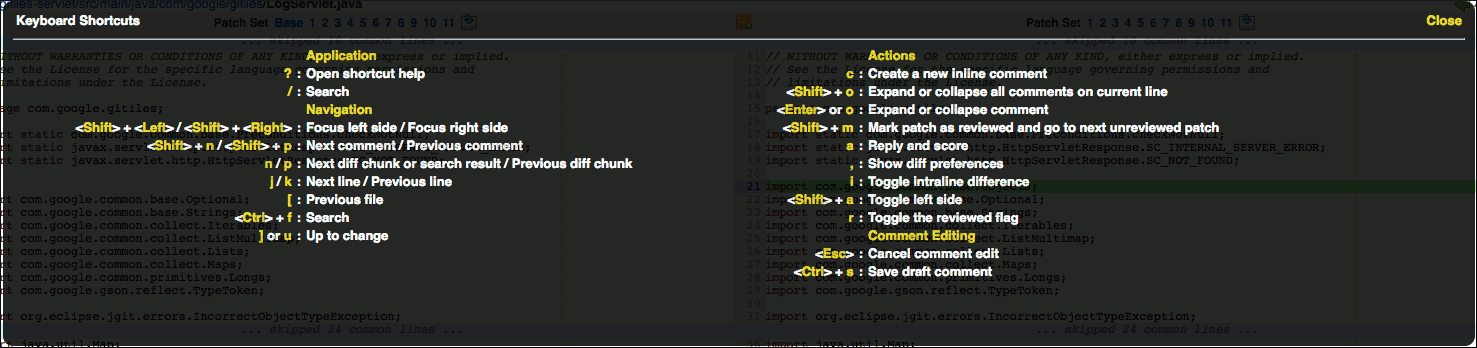
### Keyboard Shortcuts

Navigation within the review UI can be completely done by keys, and most actions can be controlled by keyboard shortcuts. Typing ? opens a popup that shows a list of available keyboard shortcuts:

*Change Screen*



*Side-by-Side Diff Screen*



## Error Messages

This section provides access to detailed explanations of Gerrit error messages. For each error message it is explained why the error is occurring and what can be done to solve it.

### branch …​ not found

With this error message Gerrit rejects to push a commit for code review if the specified target branch does not exist. To push a change for code review the commit has to be pushed to the project’s magical *refs/for/'branch'* ref (for details have a look at Create Changes). If you specify a non-existing branch in the *refs/for/'branch'* ref the push fails with the error message 'branch …​ not found'.

To fix this problem verify:

* that the branch name in the push specification is typed correctly (case sensitive) and
* that the branch really exists for this project

### change …​ closed

With this error message Gerrit rejects to push a commit or submit a review label (approval) to a change that is already closed.

When Pushing a Commit: This error occurs if you are trying to push a commit that contains the Change-Id of a closed change in its commit message. A change can be closed either because it was already submitted and merged or because it was abandoned.

If the change for which you wanted to upload a new patch set was already submitted and merged you may want to push your commit as a new change. To do this you have to remove the Change-Id from the commit message as explained and ideally generate a new Change-Id using the commit hook or EGit. Before pushing again it is also recommended to do a git rebase to base your commit on the submitted change. Pushing again should now create a new change in Gerrit.

If the change for which you wanted to upload a new patch set was abandoned and your new changes overcome the reasons for abandoning this change you may want to restore the change in the Gerrit Web UI (browse the abandoned change in the Gerrit Web UI and click on the 'Restore Change' button). Afterwards the push should succeed and a new patch set for this change will be created.

### change …​ does not belong to project …​

With this error message Gerrit rejects to push a commit to a change that belongs to another project.

This error message means that the user explicitly pushed a commit to a change that belongs to another project by specifying it as target ref. This way of adding a new patch set to a change is deprecated as explained. It is recommended to only rely on Change-Ids for replacing changes.

### change …​ not found

With this error message Gerrit rejects to push a commit to a change that cannot be found.

This error message means that the user explicitly pushed a commit to a non-existing change by specifying it as target ref. This way of adding a new patch set to a change is deprecated as explained. It is recommended to only rely on Change-Ids for replacing changes.

### commit already exists

With "commit already exists (as current patchset)" or "commit already exists (in the change)" error message Gerrit rejects to push a commit to an existing change via refs/changes/n if the commit was already successfully pushed to the change.

With "commit already exists (in the project)" error message Gerrit rejects to push a commit to an existing change via refs/changes/n if the commit was already successfully pushed to a change in project scope.

In any above case there is no new commit and consequently there is nothing for Gerrit to do. For further information about how to resolve this error, please refer to no new changes.

### contains banned commit …​

With this error message Gerrit rejects to push a commit that is banned or that would merge in an ancestor that is banned.

If a commit was identified as a bad commit (e.g. because it contains coding that violates intellectual property) and because of this it was removed from the central git repository it can be marked as banned. Gerrit will then prevent that this commit ever enters the repository again by rejecting every push of such a commit with the error message "contains banned commit …​".

If you have commits that you want to push that are based on a banned commit you may want to cherry-pick them onto a clean base and push them again.

### ... has duplicates

With this error message Gerrit rejects to push a commit if its commit message contains a Change-Id for which multiple changes can be found in the project.

This error means that there is an inconsistency in Gerrit since for one project there are multiple changes that have the same Change-Id. Every change is expected to have an unique Change-Id.

### invalid author

For every pushed commit Gerrit verifies that the e-mail address of the author matches one of the registered e-mail addresses of the pushing user. If this is not the case pushing the commit fails with the error message "invalid author". This policy can be bypassed by having the access right 'Forge Author'.

This error may happen for two reasons:

* incorrect configuration of the e-mail address on client or server side
* missing privileges to push commits of other authors

### no new changes

With this error message Gerrit rejects to push a commit if the pushed commit was already successfully pushed to Gerrit in project scope. In this case there is no new change and consequently there is nothing for Gerrit to do.

### missing Change-Id in commit message footer

With this error message Gerrit rejects to push a commit to a project which is configured to always require a Change-Id in the commit message if the commit message of the pushed commit does not contain a Change-Id in the footer (the last paragraph).

This error may happen for different reasons:

* missing Change-Id in the commit message
* Change-Id is contained in the commit message but not in the last paragraph

### not valid ref

With this error message Gerrit rejects to push a commit if the target ref in the push specification has an incorrect format (for example: '/refs/for/master', 'refs/for//master').To solve the problem you have to correct the target ref in the push specification.

### Permission denied (publickey)

With this error message an SSH command to Gerrit is rejected if the SSH authentication is not successful.

The SSH protocol uses Public-key Cryptography for authentication. This means for a successful SSH authentication you need your private SSH key and the corresponding public SSH key must be known to Gerrit.

If you are facing this problem, do the following:

* Verify that you are using the correct username for the SSH command and that it is typed correctly (case sensitive). You can look up your username in the Gerrit Web UI under 'Settings' → 'Profile'.
* Verify that you have uploaded your public SSH key for your Gerrit account. To do this go in the Gerrit Web UI to 'Settings' → 'SSH Public Keys' and check that your public SSH key is there. If your public SSH key is not there you have to upload it.
* Verify that you are using the correct private SSH key. To find out which private SSH key is used test the SSH authentication as described below. From the trace you should see which private SSH key is used.

### prohibited by Gerrit

This is a general error message that is returned by Gerrit if a push is not allowed, e.g. because the pushing user has no sufficient privileges.

* if you push a commit for code review to a branch for which you don’t have upload permissions (access right 'Push' on refs/for/refs/heads/\*)
* if you bypass code review without 'Push' access right on refs/heads/\*
* if you bypass code review pushing to a non-existing branch without 'Create Reference' access right on refs/heads/\*
* if you push an annotated tag without 'Push Annotated Tag' access right on refs/tags/\*
* if you push a signed tag without 'Push Signed Tag' access right on refs/tags/\*
* if you push a lightweight tag without the access right 'Create Reference' for the reference name refs/tags/\*
* if you push a tag with somebody else as tagger and you don’t have the 'Forge Committer' access right for the reference name refs/tags/\*
* if you push to a project that is in state 'Read Only'

For new users it often happens that they accidentally try to bypass code review. The push then fails with the error message 'prohibited by Gerrit' because the project didn’t allow to bypass code review. Bypassing the code review is done by pushing directly to refs/heads/\* (e.g. refs/heads/master) instead of pushing to refs/for/\* (e.g. refs/for/master).

### Project not found: …​

With this error message Gerrit rejects to push a commit if the git repository to which the push is done does not exist as a project in the Gerrit server or if the pushing user has no read access for this project.

The name of the project in Gerrit has the same name as the path of its git repository (excluding the '.git' extension).

### squash commits first

With this error message Gerrit rejects to push a commit if it contains the same Change-Id as a predecessor commit.

The reason for rejecting such a commit is that it would introduce, for the corresponding change in Gerrit, a dependency upon itself. Gerrit prevents such dependencies between patch sets within the same change to keep the review process simple. Otherwise reviewers would not only have to review the latest patch set but also all the patch sets the latest one depends on.

This error is quite common, it appears when a user tries to address review comments and creates a new commit instead of amending the existing commit. Another possibility for this error, although less likely, is that the user tried to create a patch series with multiple changes to be reviewed and accidentally included the same Change-Id into the different commit messages.

### you are not allowed to upload merges

With this error message Gerrit rejects to push a merge commit if the pushing user has no permission to upload merge commits for the project to which the push is done.

If you need to upload merge commits, you can contact one of the project owners and request permission to upload merge commits (access right 'Push Merge Commit') for this project.

If one of your changes could not be merged in Gerrit due to conflicts and you created the merge commit to resolve the conflicts, you might want to revert the merge and instead of this do a rebase.

### Upload denied for project '…​'

With this error message Gerrit rejects to push a commit if the pushing user has no upload permissions for the project to which the push was done.

## Searching Changes

### Default Searches

Most basic searches can be viewed by clicking on a link along the top menu bar. The link will prefill the search box with a common search query, execute it, and present the results. If exactly one change matches the search, the change will be presented instead of a list.

| Description | Default Query |
| --- | --- |
| All > Open | status:open '(or is:open)' |
| All > Merged | status:merged |
| All > Abandoned | status:abandoned |
| My > Drafts | owner:self is:draft |
| My > Watched Changes | status:open is:watched |
| My > Starred Changes | is:starred |
| My > Draft Comments | has:draft |
| Open changes in Foo | status:open project:Foo |

### Basic Change Search

Similar to many popular search engines on the web, just enter some text and let Gerrit figure out the meaning:

| Description | Examples |
| --- | --- |
| Legacy numerical id | 15183 |
| Full or abbreviated Change-Id | Ic0ff33 |
| Full or abbreviated commit SHA-1 | d81b32ef |
| Email address | [user@example.com](mailto:user@example.com) |
| Approval requirement | Code-Review>=+2, Verified=1 |

### Search Operators

Operators act as restrictions on the search. As more operators are added to the same query string, they further restrict the returned results. Search can also be performed by typing only a text with no operator, which will match against a variety of fields.

age:'AGE'

Amount of time that has expired since the change was last updated with a review comment or new patch set. The age must be specified to include a unit suffix, for example age:2d:

s, sec, second, seconds

m, min, minute, minutes

h, hr, hour, hours

d, day, days

w, week, weeks (1 week is treated as 7 days)

mon, month, months (1 month is treated as 30 days)

y, year, years (1 year is treated as 365 days)

before:'TIME'/until:'TIME'

Changes modified before the given 'TIME', inclusive. Must be in the format 2006-01-02[ 15:04:05[.890][ -0700]]; omitting the time defaults to 00:00:00 and omitting the timezone defaults to UTC.

after:'TIME'/since:'TIME'

Changes modified after the given 'TIME', inclusive. Must be in the format 2006-01-02[ 15:04:05[.890][ -0700]]; omitting the time defaults to 00:00:00 and omitting the timezone defaults to UTC.

change:'ID'

Either a legacy numerical 'ID' such as 15183, or a newer style Change-Id that was scraped out of the commit message.

conflicts:'ID'

Changes that conflict with change 'ID'. Change 'ID' can be specified as a legacy numerical 'ID' such as 15183, or a newer style Change-Id that was scraped out of the commit message.

destination:'NAME'

Changes which match the current user’s destination named 'NAME'. (see Named Destinations).

owner:'USER', o:'USER'

Changes originally submitted by 'USER'. The special case of owner:self will find changes owned by the caller.

ownerin:'GROUP'

Changes originally submitted by a user in 'GROUP'.

query:'NAME'

Changes which match the current user’s query named 'NAME' (see Named Queries).

reviewer:'USER', r:'USER'

Changes that have been, or need to be, reviewed by 'USER'. The special case of reviewer:self will find changes where the caller has been added as a reviewer.

reviewerin:'GROUP'

Changes that have been, or need to be, reviewed by a user in 'GROUP'.

commit:'SHA1'

Changes where 'SHA1' is one of the patch sets of the change.

project:'PROJECT', p:'PROJECT'

Changes occurring in 'PROJECT'. If 'PROJECT' starts with ^ it matches project names by regular expression. The dk.brics.automaton library is used for evaluation of such patterns.

projects:'PREFIX'

Changes occurring in projects starting with 'PREFIX'.

parentproject:'PROJECT'

Changes occurring in 'PROJECT' or in one of the child projects of 'PROJECT'.

branch:'BRANCH'

Changes for 'BRANCH'. The branch name is either the short name shown in the web interface or the full name of the destination branch with the traditional 'refs/heads/' prefix.

If 'BRANCH' starts with ^ it matches branch names by regular expression patterns. The dk.brics.automaton library is used for evaluation of such patterns.

intopic:'TOPIC'

Changes whose designated topic contains 'TOPIC', using a full-text search.

If 'TOPIC' starts with ^ it matches topic names by regular expression patterns. The dk.brics.automaton library is used for evaluation of such patterns.

topic:'TOPIC'

Changes whose designated topic matches 'TOPIC' exactly. This is often combined with 'branch:' and 'project:' operators to select all related changes in a series.

ref:'REF'

Changes where the destination branch is exactly the given 'REF' name. Since 'REF' is absolute from the top of the repository it must start with 'refs/'.

If 'REF' starts with ^ it matches reference names by regular expression patterns. The dk.brics.automaton library is used for evaluation of such patterns.

tr:'ID', bug:'ID'

Search for changes whose commit message contains 'ID' and matches one or more of the trackingid sections in the server’s configuration file. This is typically used to search for changes that fix a bug or defect by the issue tracking system’s issue identifier.

label:'VALUE'

Matches changes where the approval score 'VALUE' has been set during a review. See labels below for more detail on the format of the argument.

message:'MESSAGE'

Changes that match 'MESSAGE' arbitrary string in the commit message body.

comment:'TEXT'

Changes that match 'TEXT' string in any comment left by a reviewer.

path:'PATH'

Matches any change touching file at 'PATH'. By default exact path matching is used, but regular expressions can be enabled by starting with ^. For example, to match all XML files use file:^.\*\.xml$. The dk.brics.automaton library is used for the evaluation of such patterns.

The ^ required at the beginning of the regular expression not only denotes a regular expression, but it also has the usual meaning of anchoring the match to the start of the string. To match all Java files, use file:^.\*\.java.

The entire regular expression pattern, including the ^ character, should be double quoted when using more complex construction (like ones using a bracket expression). For example, to match all XML files named like 'name1.xml', 'name2.xml', and 'name3.xml' use file:"^name[1-3].xml".

file:'NAME', f:'NAME'

Matches any change touching a file containing the path component 'NAME'. For example a file:src will match changes that modify files named gerrit-server/src/main/java/Foo.java. Name matching is exact match, file:Foo.java finds any change touching a file named exactly Foo.java and does not match AbstractFoo.java.

Regular expression matching can be enabled by starting the string with ^. In this mode file: is an alias of path: (see above).

has:draft

True if there is a draft comment saved by the current user.

has:star

Same as 'is:starred', true if the change has been starred by the current user.

has:edit

True if the change has inline edit created by the current user.

is:starred

Same as 'has:star', true if the change has been starred by the current user.

is:watched

True if this change matches one of the current user’s watch filters, and thus is likely to notify the user when it updates.

is:reviewed

True if any user has commented on the change more recently than the last update (comment or patch set) from the change owner.

is:owner

True on any change where the current user is the change owner. Same as owner:self.

is:reviewer

True on any change where the current user is a reviewer. Same as reviewer:self.

is:open, is:pending

True if the change is open.

is:draft

True if the change is a draft.

is:closed

True if the change is either merged or abandoned.

is:merged, is:abandoned

Same as status:'STATE'.

is:mergeable

True if the change has no merge conflicts and could be merged into its destination branch.

status:open, status:pending

True if the change state is 'review in progress'.

status:reviewed

Same as 'is:reviewed', matches if any user has commented on the change more recently than the last update (comment or patch set) from the change owner.

status:closed

True if the change is either 'merged' or 'abandoned'.

status:merged

Change has been merged into the branch.

status:abandoned

Change has been abandoned.

added:'RELATION''LINES', deleted:'RELATION''LINES', delta/size:'RELATION''LINES'

True if the number of lines added/deleted/changed satisfies the given relation for the given number of lines.

For example, added:>50 will be true for any change which adds at least 50 lines.

Valid relations are >=, >, ⇐, <, or no relation, which will match if the number of lines is exactly equal.

commentby:'USER'

Changes containing a top-level or inline comment by 'USER'. The special case of commentby:self will find changes where the caller has commented.

from:'USER'

Changes containing a top-level or inline comment by 'USER', or owned by 'USER'. Equivalent to (owner:USER OR commentby:USER).

reviewedby:'USER'

Changes where 'USER' has commented on the change more recently than the last update (comment or patch set) from the change owner.

author:'AUTHOR'

Changes where 'AUTHOR' is the author of the current patch set. 'AUTHOR' may be the author’s exact email address, or part of the name or email address.

committer:'COMMITTER'

Changes where 'COMMITTER' is the committer of the current patch set. 'COMMITTER' may be the committer’s exact email address, or part of the name or email address.

### Argument Quoting

Operator values that are not bare words (roughly A-Z, a-z, 0-9, @, hyphen, dot and underscore) must be quoted for the query parser.

Quoting is accepted as either double quotes (e.g. message:"the value") or as matched curly braces (e.g. message:{the value}).

### Boolean Operators

*Negation*

Any operator can be negated by prefixing it with -, for example -is:starred is the exact opposite of is:starred and will therefore return changes that are not starred by the current user.

The operator NOT (in all caps) is a synonym.

*AND*

The boolean operator AND (in all caps) can be used to join two other operators together. This results in a restriction of the results, returning only changes that match both operators.

*OR*

The boolean operator OR (in all caps) can be used to find changes that match either operator. This increases the number of results that are returned, as more changes are considered.

### Labels

Label operators can be used to match approval scores given during a code review. The specific set of supported labels depends on the server configuration, however the Code-Review label is provided out of the box.

A label name is any of the following:

The label name. Example: label:Code-Review.

The label name followed by a ',' followed by a reviewer id or a group id. To make it clear whether a user or group is being looked for, precede the value by a user or group argument identifier ('user=' or 'group='). If an LDAP group is being referenced make sure to use 'ldap/<groupname>'.

A label name must be followed by a score, or an operator and a score. The easiest way to explain this is by example.

*label:Code-Review=2*

*label:Code-Review=+2*

*label:Code-Review+2*

Matches changes where there is at least one +2 score for Code-Review. The + prefix is optional for positive score values. If the + is used, the = operator is optional.

*label:Code-Review=-2*

*label:Code-Review-2*

Matches changes where there is at least one -2 score for Code-Review. Because the negative sign is required, the = operator is optional.

*label:Code-Review=1*

Matches changes where there is at least one +1 score for Code-Review. Scores of +2 are not matched, even though they are higher.

*label:Code-Review>=1*

Matches changes with either a +1, +2, or any higher score.

*label:Code-Review=+2,aname*

Matches changes with a +2 code review where the reviewer or group is aname.

*label:Code-Review=2,user=jsmith*

Matches changes with a +2 code review where the reviewer is jsmith.

*label:Code-Review=+1,group=ldap/linux.workflow*

Matches changes with a +1 code review where the reviewer is in the ldap/linux.workflow group.

*label:Code-Review⇐-1*

Matches changes with either a -1, -2, or any lower score.

*is:open label:Code-Review+2 label:Verified+1 NOT label:Verified-1 NOT label:Code-Review-2*

Matches changes that are ready to be submitted.

*is:open (label:Verified-1 OR label:Code-Review-2)*

Changes that are blocked from submission due to a blocking score.

### Magical Operators

Most of these operators exist to support features of Gerrit Code Review, and are not meant to be accessed by the average end-user. However, they are recognized by the query parser, and may prove useful in limited contexts to administrators or power-users.

visibleto:'USER-or-GROUP'

Matches changes that are visible to 'USER' or to anyone who is a member of 'GROUP'. Here group names may be specified as either an internal group name, or if LDAP is being used, an external LDAP group name. The value may be wrapped in double quotes to include spaces or other special characters. For example, to match an LDAP group: visibleto:"CN=Developers, DC=example, DC=com".

This operator may be useful to test access control rules, however a change can only be matched if both the current user and the supplied user or group can see it. This is due to the implicit 'is:visible' clause that is always added by the server.

is:visible

Magical internal flag to prove the current user has access to read the change. This flag is always added to any query.

starredby:'USER'

Matches changes that have been starred by 'USER'. The special case starredby:self applies to the caller.

watchedby:'USER'

Matches changes that 'USER' has configured watch filters for. The special case watchedby:self applies to the caller.

draftby:'USER'

Matches changes that 'USER' has left unpublished draft comments on. Since the drafts are unpublished, it is not possible to see the draft text, or even how many drafts there are. The special case of draftby:self will find changes where the caller has created a draft comment.

limit:'CNT'

Limit the returned results to no more than 'CNT' records. This is automatically set to the page size configured in the current user’s preferences. Including it in a web query may lead to unpredictable results with regards to pagination.