Code Review #2

Majid Babaei

Code Reviews facilitate knowledge sharing across the code base and across the team.

Improving Quality with Code Reviews

Informal reviews are the more naïve way to check the code for defects. This kind of review usually involves no particular preparation nor planned metrics to measure the effectiveness of the review. They are usually performed by one or more peers, typically for brainstorming ideas.

Walkthrough reviews are slightly more formal than informal reviews. They can be performed by a single person or by multiple participants. Defects are usually pointed out and discussed. This type of review is more about querying for feedback and comments from participants rather than actually correcting defects.

Inspection reviews are well planned and structured. It aims at finding and logging defects, gathering insights, and communicating them within the team. The process is supported by a checklist and metrics to gather the effectiveness of the review process. It is usually not performed by the author.

Design Smells

Architectures that are not well designed and/or not properly maintained over time make new functionalities more difficult to develop. Furthermore, technical debt can quickly pile up in such scenario.

Cyclic Dependenci

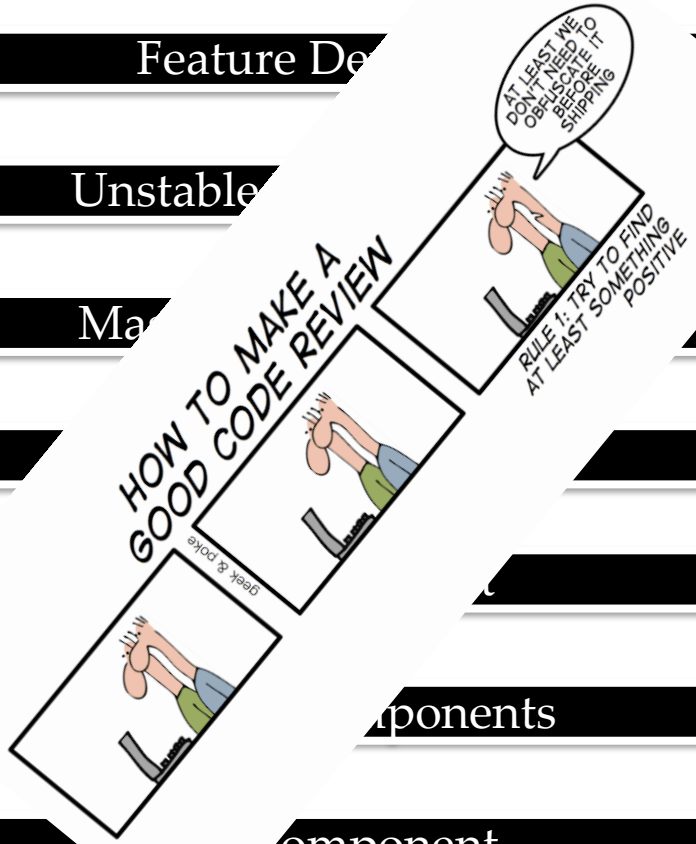
Feature De

Unstable

Ma

ponents

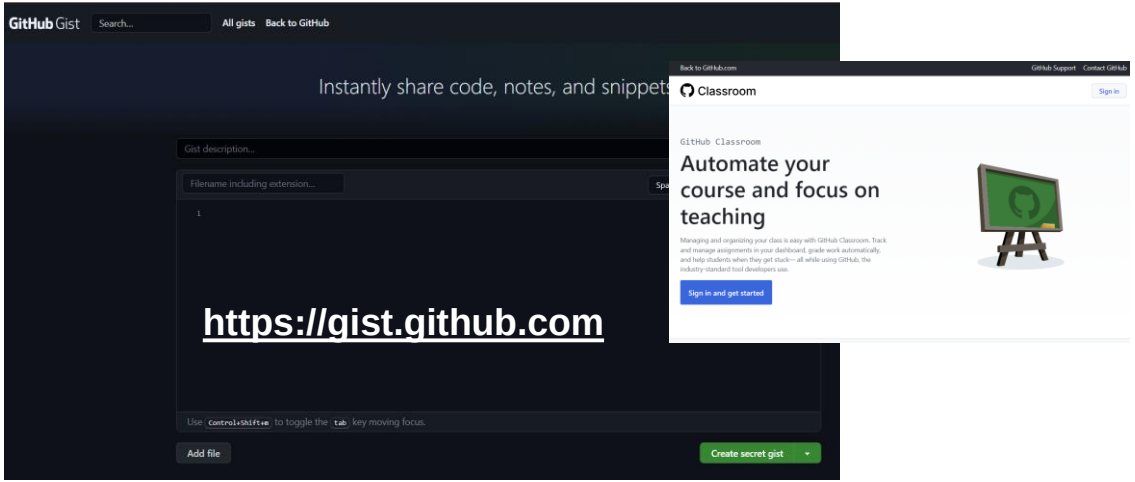
omponent



Gerrit

Gerrit is a ***Git server*** that provides

- Code Review
- Access Control on the Git repositories



Code Review:

- Gerrit allows to ***review commits before*** they are integrated into a *target branch*.
- Code review is ***optional***, but required by default (*bypassing code review can be allowed* by granting access rights for direct push)

Access Rights:

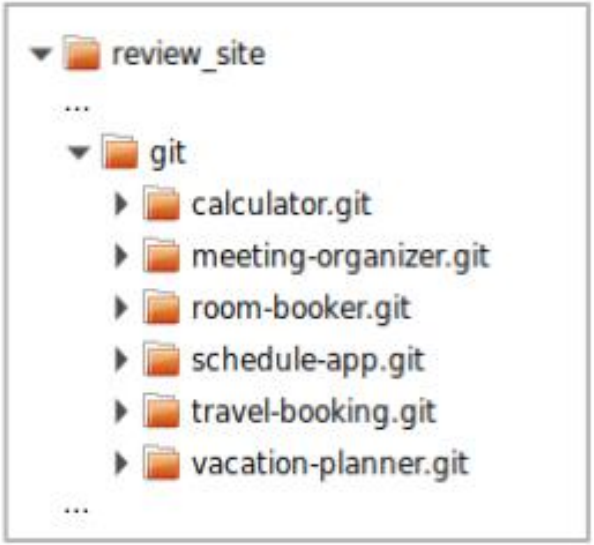
- Gerrit provides ***fine-grained read and write permissions on branch level*** (with *Git only you have access to everything* once you can access a repository)
- This presentation concentrates on the code review aspect, access controls are not covered.

Gerrit

Gerrit is built on top of Git

- It manages standard Git repositories and
- It controls access and updates to them

Gerrit



In the *Gerrit installation folder* on the server (`review_site`) you can find a `git` folder that *contains all Git repositories* that are managed by Gerrit:

- these are ***bare repositories*** (this means they don't have a working dir)
- the repositories may be hierarchically structured in subfolders

Build stability

- When adopting Continuous Integration, you will get a RAG (red-amber-green) status of your changes in the project repository.

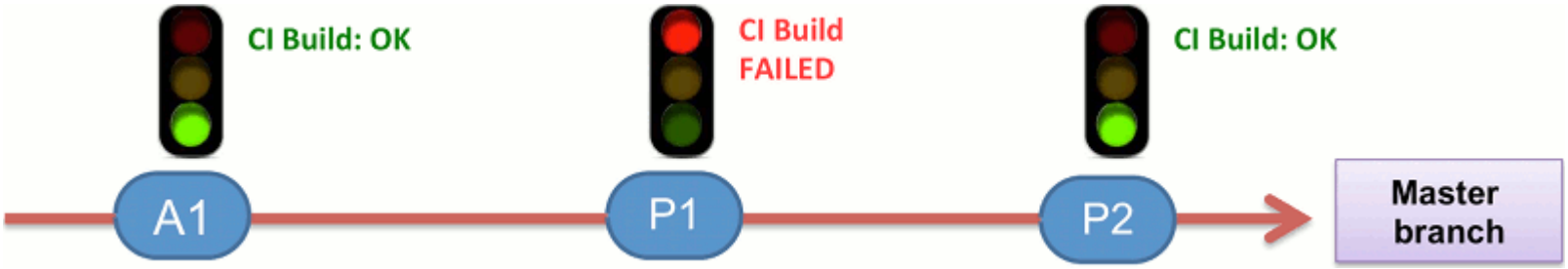
- Whenever a build is not green, it is defined as "broken" .

- It is the team's priority to **re-establish a healthy head version** of the branch by fixing the build.

- Code Review helps by **reducing broken builds**, thanks to the ability to **link the review status to a validation step**, resulting in less time wasted by the development team.

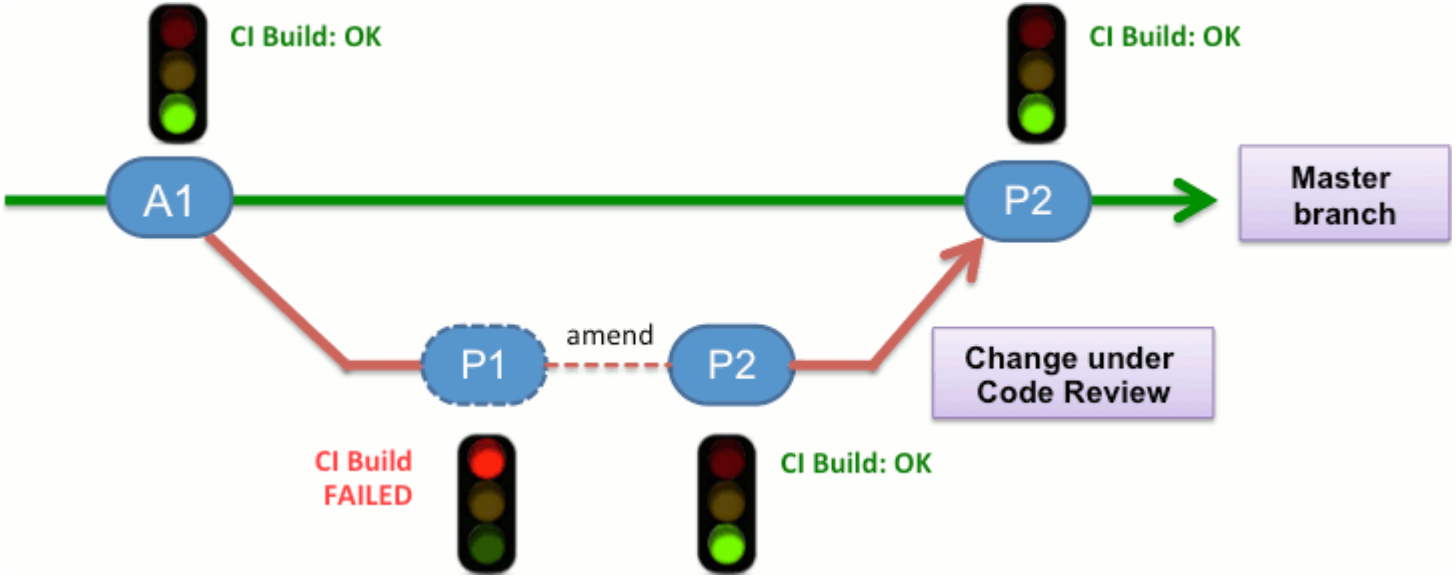
Build stability

- Continuous Integration without Code Review
- When every commit is directly pushed to the master branch.
- An incorrect patch (P1) is pushed, the build fails, which results in them pushing a new patch (P2) to re-establish a green build again.
- The master branch will *keep a record of the failure* by having both P1 and P2 in the history of the master branch.



Build stability

- Continuous Integration with Code Review
- It allows you to *check the sanity of the change in isolation*, without interfering with the normal integration flow of the functionalities and keeping a green build.
- It causes a build failure only in the review branch



Build stability

- The benefits are twofold:

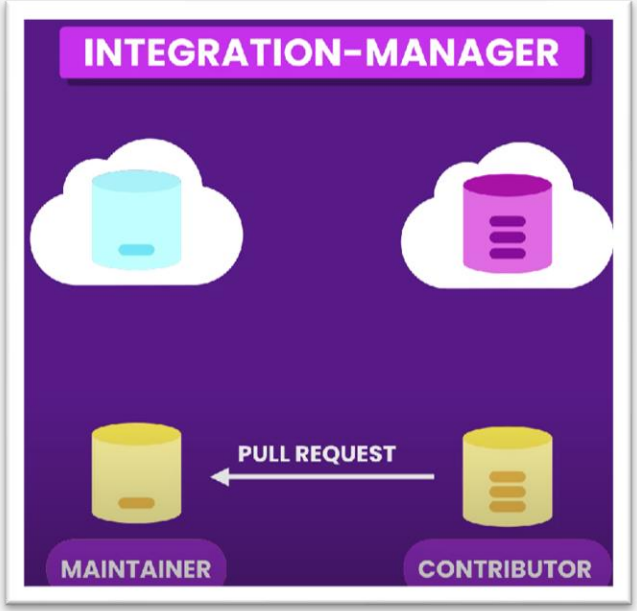
- Firstly*, the normal branch stability has not been impacted*

- Secondly, the Code Review branch has allowed the *pre-validation and clean-up of the code for allowing a consistent history* without losing track of the review activity and the continuous integration feedback.

Gerrit Concepts

- Gerrit “speaks” the ***Git protocol***
  ⇒ users only need a ***Git client***
  (there is no need to install a “Gerrit client”)
  ⇒ this means *Gerrit must somehow map its concepts onto Git*
- Gerrit allows to ***review commits before*** they are integrated into the target branch, but ***code review is optional***
- commits are pushed to Gerrit by using the `git push` command
- Git is a toolbox (“Swiss army knife”) which allows many workflows, *Gerrit defines one workflow for working with Git*

GitHub ***Pull Requests*** is another workflow for working with Git (not supported by Gerrit).



Q: Since code review is optional, how does Gerrit know if you push directly to Git or for code review?

Push for Code Review

Push for code review:

- Same command as pushing to Git with one Gerrit speciality: The target branch is prefixed with *refs/for/*
- *git push origin HEAD:refs/for/<branch-name>*
- Example:
  git push origin HEAD:refs/for/master

Push directly to Git (bypassing code review):

- *git push origin HEAD:<branch-name>*
- Example:
  git push origin HEAD:master
  same as
  git push origin HEAD:refs/heads/master

Whether pushing directly to Git, and hence bypassing code review, is allowed can be controlled by access rights.

Using *HEAD* in the push command means that the current commit/branch is pushed. Instead you can also specify a branch or SHA1.

In Git, `origin` is a shorthand name for the remote repository that a project was originally cloned from. More precisely, it is used instead of that original repository's URL.

Push for Code Review

The command git push works like the following:

```
git push REPOSITORY SOURCE_REFSPEC:DESTINATION_REFSPEC
```

With Gerrit, when you use:

```
git push origin HEAD:refs/for/branch
```

You're asking Git to Push to the origin repository (by default, the repository you have cloned from)

Push the commit to "branch" to be reviewed on Gerrit. The "refs/for/" prefix is a "magical" branch which ***instructs Gerrit that a code review must be created***

Gerrit - Overview

Gerrit CHANGES YOUR DOCUMENTATION BROWSE

Active 323619 Configure new All-Users repository to use jgit ref cache ABANDON EDIT

Change Info SHOW ALL

Owner: Matthias Sohn
Author: Matthias Sohn
Committer: Matthias Sohn
Reviewers: Jacek Centko... Luca Patrick +1 GerritCI
CC: Compton Banks
Repo | Branch: gerrit | stable-3.2

Submit Requirements

- Code-Review +1
- No-Unresolved-Comments Unsatisfied
- Code-Style +1
- Release-Notes Unsatisfied
- Verified -1

Change Metadata

REPLY

```
Configure new All-Users repository to use jgit ref cache

Issue:
When AccountCache got serialized (see [1]) it was decided to
get rid of eviction problems by direct access to the All-Users
repository. Back then it seemed like a pretty cheap and viable option.

The problem is that JGit so far didn't cache refs and gets the
corresponding object id (which is used as cache key) in roughly 3 steps:
* lock
* lookup ref in question (I/O operation)
* unlock

That doesn't seem bad however single call to AccountCacheImpl.get(Id)
results already in 3 calls to lookup refs in order to get ids:
* refs/users/default (to get ObjectId for preferences)
```

Commit Message

SHOW ALL EDIT

Comments 2 unresolved 1 resolved
Checks No results

Review Summary

Relation chain

- Configure new All-Users repository to use jgit ref cache
- Update jgit to 7c0b3e1d61b3e9a61481ca3a45dc94c7df152d75
- uploadPackAuditEventLog: Avoid commit timestamp mismatch
- InitJgitConfig: Git protocol v2 is enabled per default
- Remove the redundant HttpPushForReviewIT
- Auto-enable git wire protocol v2 in jgit
- Fix warnings that repository with useCnt=0 is closed again
- ~~Update jgit to dc766feffce1edc32d156520960eae374e06bef0~~ (Abandoned)

Submitted together SHOW ALL (8)

- Configure new All-Users repository to use jgit ref cache gerrit | stable-3.2
- Update jgit to 7c0b3e1d61b3e9a61481ca3a45dc94c7df152d75 gerrit | stable-3.2
- uploadPackAuditEventLog: Avoid commit timestamp mismatch gerrit | stable-3.2
- InitJgitConfig: Git protocol v2 is enabled per default gerrit | stable-3.2
- Remove the redundant HttpPushForReviewIT gerrit | stable-3.2
- Auto-enable git wire protocol v2 in jgit gerrit | stable-3.2

Related Changes

Files Comments Checks

Base → Patchset 1 bc880b1 File List DOWNLOAD EXPAND ALL

File	Comments	Size	Delta
Commit message			
M java/com/google/gerrit/server/schema/AllUsersCreator.java	2 comments (2 unresolved)		+9 -0
			+9 -0

Change Log

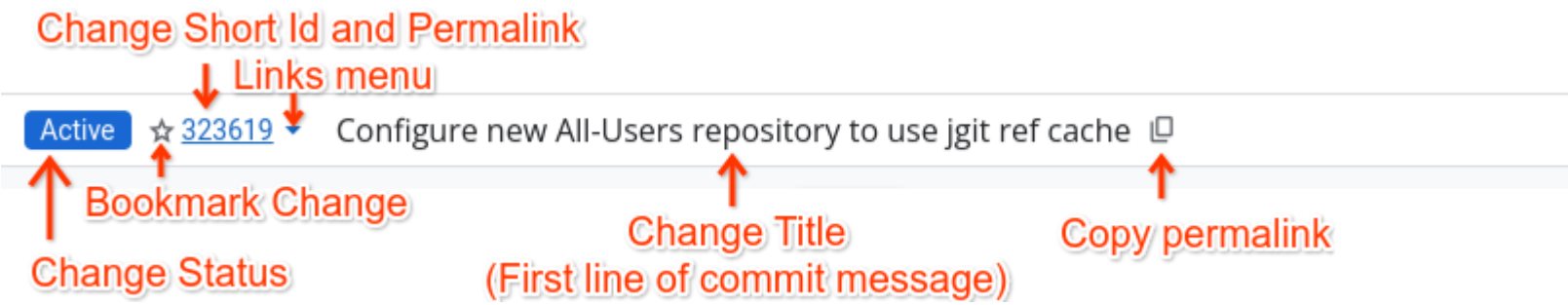
Chronological log of change state

EXPAND ALL

Matthias Sohn	Uploaded patch set 1.	VIEW DIFF	Patchset 1 | Nov 13, 2021 10:27 PM
Matthias Sohn	Added to reviewer: Luca Jacek Centkowski Patrick		Patchset 1 | Nov 13, 2021 10:30 PM
GerritCI	Added to reviewer: GerritCI		Patchset 1 | Nov 13, 2021 11:49 PM
GerritCI	Code-Style +1		Patchset 1 | Nov 13, 2021 11:49 PM
GerritCI	Verified -1		Patchset 1 | Nov 13, 2021 11:49 PM
Patrick	Code-Review +1 2		Patchset 1 | Nov 23, 2021 9:36 AM
Compton Banks	Added to cc: Compton Banks		Patchset 1 | Feb 05, 2022 5:12 AM
Compton Banks	1 I don't get how to read this		Patchset 1 | Feb 05, 2022 5:12 AM

Powered by Gerrit Code Review (3.7.0-461-g6def0e41b5) | Privacy

Press "?" for keyboard shortcuts



The change status shows the state of the change:

Active: The change is under active review.
Merge Conflict: The change can’t be merged into the destination branch due to conflicts.
Ready to Submit: The change has all necessary approvals and fulfils all other submit requirements. It can be submitted.
Merged: The change was successfully merged into the destination branch.
Abandoned: The change was abandoned. It is not intended to be updated, reviewed or submitted anymore.
WIP: The change was marked as "Work in Progress“ to indicate to reviewers that they shouldn’t review the change yet.

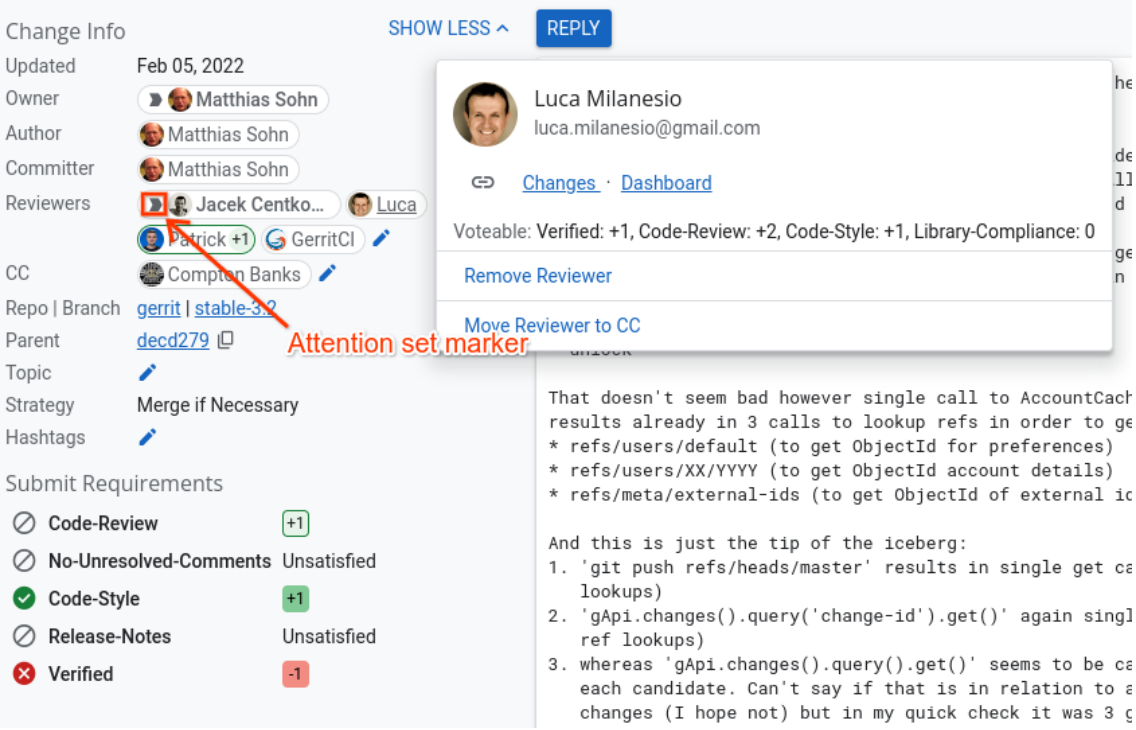
Owner is the person who created the change

Uploader is the person who uploaded the latest patchset (the patchset that will be merged if the change is submitted)

Author/Committer are concepts from Git and are retrieved from the commit when it's sent for review.

Accounts in CC receive notifications for the updates on the change, but don't need to vote/review

Attention set, Something updated/changed since last review, their vote is required

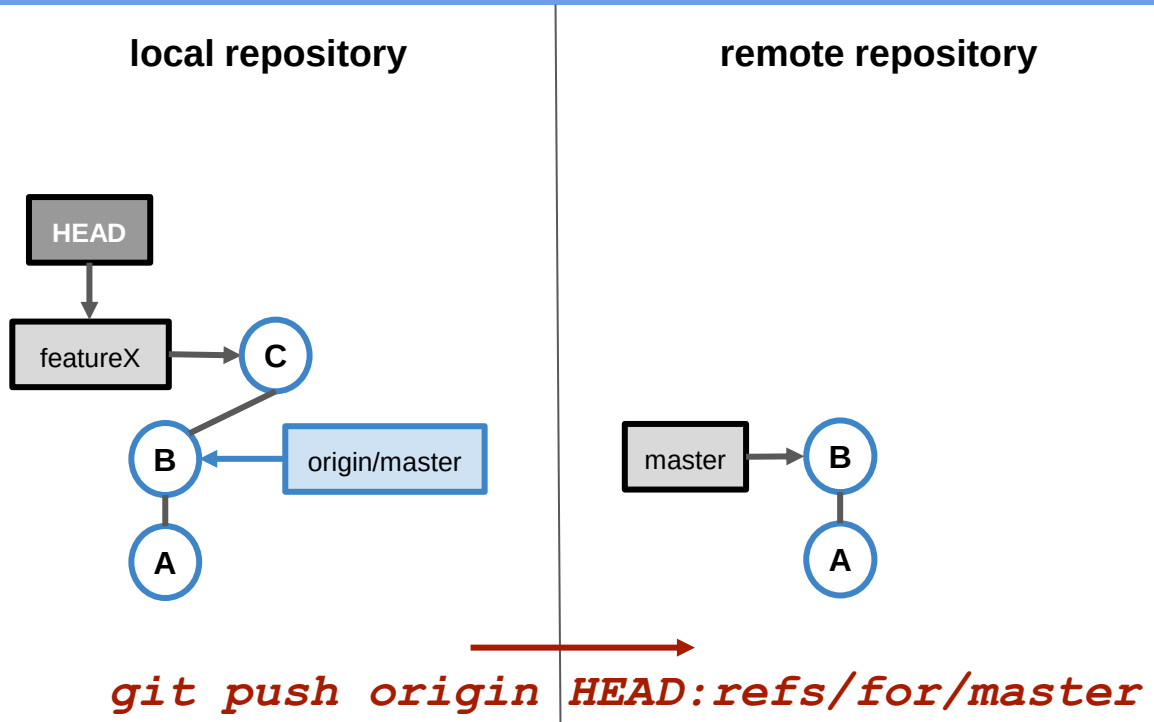


Push for Code Review

```
git push origin HEAD:refs/for/master
```

- From the Git clients perspective it looks like every push for code review goes to the same branch:
  `refs/for/master`
- However Gerrit tricks the Git client:
  - it creates a new ref for the commit(s) that are pushed
  - it creates or updates an open Gerrit change for each pushed commit

Push for Code Review - Case 1



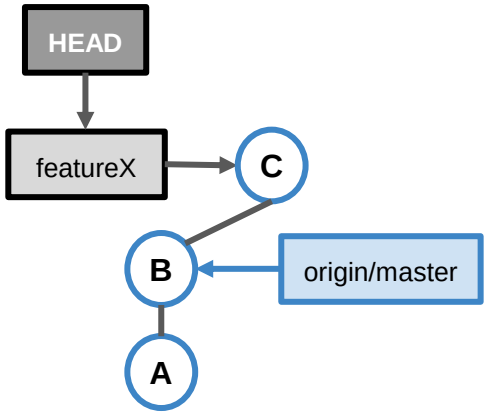
Situation:

- The remote repository was cloned, a local `featureX` branch was created and in this branch a commit **C** was created.

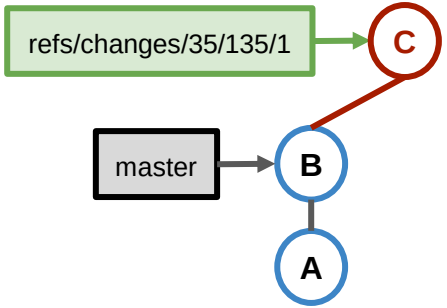
Q: What happens on push for code review?

Push for Code Review - Case 1

local repository



remote repository

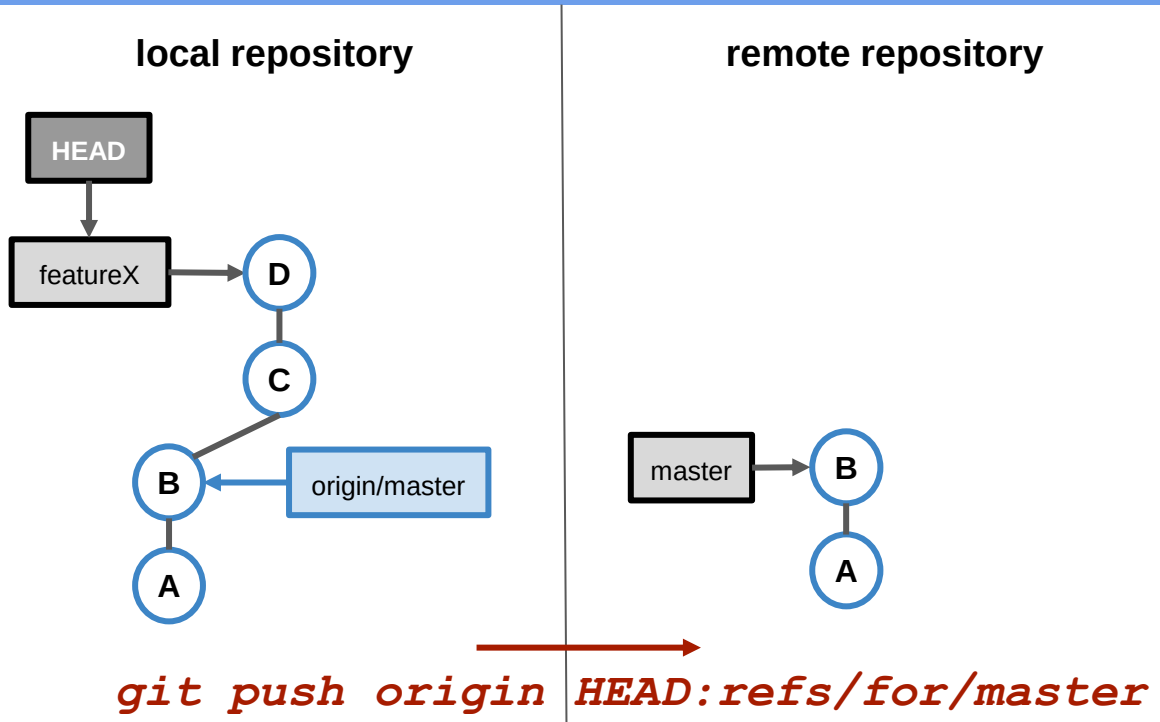



```
git push origin HEAD:refs/for/master
```

Push for Code Review:

- It pushes commit ***C*** to the remote repository
- Gerrit creates a new ***change ref*** that points to the new commit (`refs/changes/35/135/1`)
- Gerrit creates a new ***change*** object in its database
- Gerrit does **not** update the `master` branch in the remote repository
- The target branch is only updated once code review was done and the change is approved and submitted

Push for Code Review - Case 2



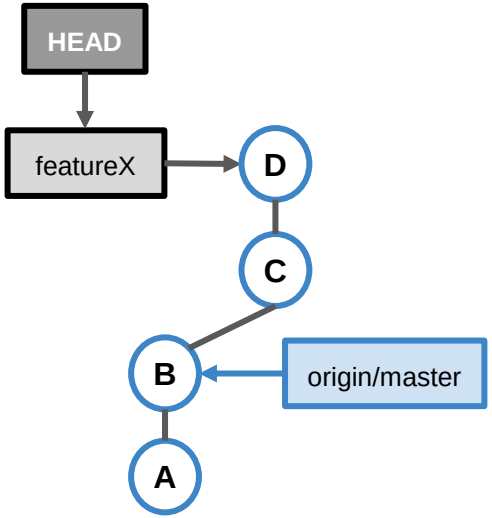
Situation:

- The remote repository was cloned, a local `featureX` branch was created and in this branch two commits, ***C*** and ***D***, were created.

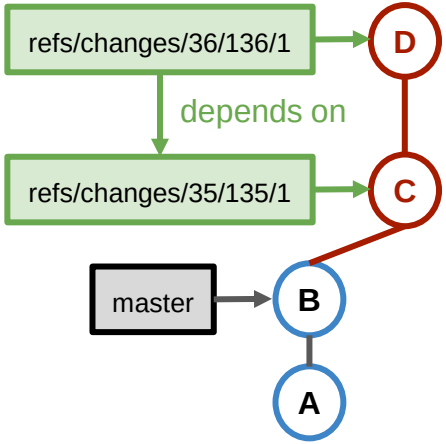
Q: Which commits get pushed?

Push for Code Review - Case 2

local repository



remote repository

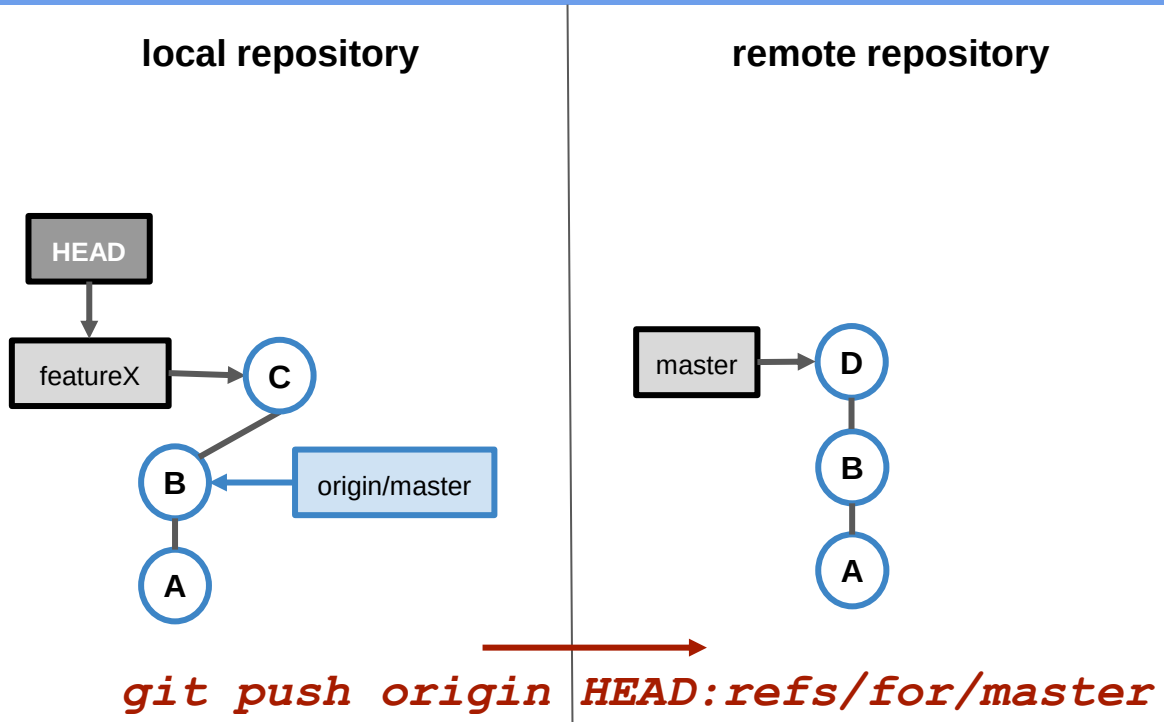



```
git push origin HEAD:refs/for/master
```

Push for Code Review:

- pushes **all** commits which are reachable from the pushed commit and which are not available in the remote repository
- for each pushed commit Gerrit creates a change ref and a Gerrit change in its database
- The change for commit ***D*** depends on the change for commit ***C*** (since commit ***D*** depends on commit ***C***).

Push for Code Review - Case 3



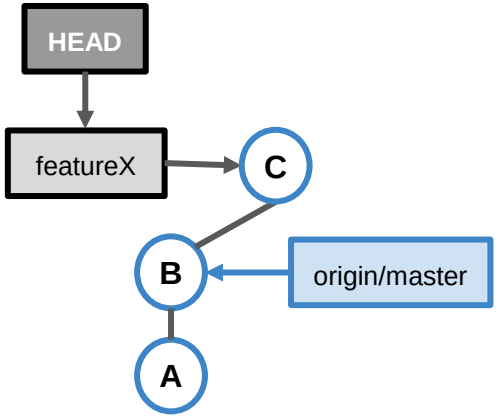
Situation:

- The remote repository was cloned, a local `featureX` branch was created and in this branch a commit ***C*** was created. In the meantime the remote branch `master` was updated to a commit ***D***.

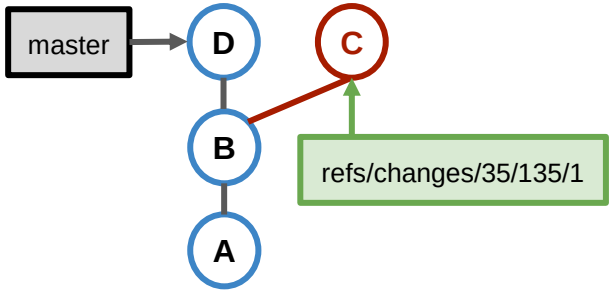
Q: What happens on push for code review?

Push for Code Review - Case 3

local repository



remote repository

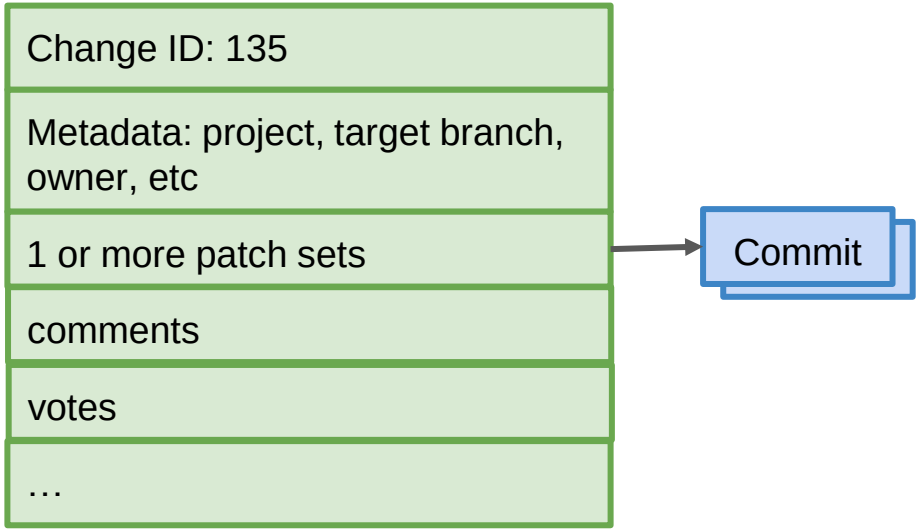



```
git push origin HEAD:refs/for/master
```

The push succeeds:

- Gerrit accepts commit ***C*** and creates a new change for it.
- The push succeeds even if commit ***C*** and ***D*** would be conflicting.
- If the push would have been done directly to Git this push would have failed since `master` cannot be fast-forwarded to the pushed commit.
- Submitting the change may or may not succeed (depends on the submit strategy).

Change



- the ***numeric ID*** uniquely identifies a change on a Gerrit server
- the ***change owner*** is the user that uploaded the change (can differ from committer and commit author)
- ***patch sets*** correspond to Git commits

Commit author vs. Committer

The **FREE** online **Pro Git** book explains it like this

- You may be wondering what the difference is between author and committer.
- The author is the person who originally wrote the code or made the changes, while the committer is the person who added the changes to the repository
- So, if you send in a patch to a project and one of the core members applies the patch, both of you get credit — you as the author and the core member as the committer.

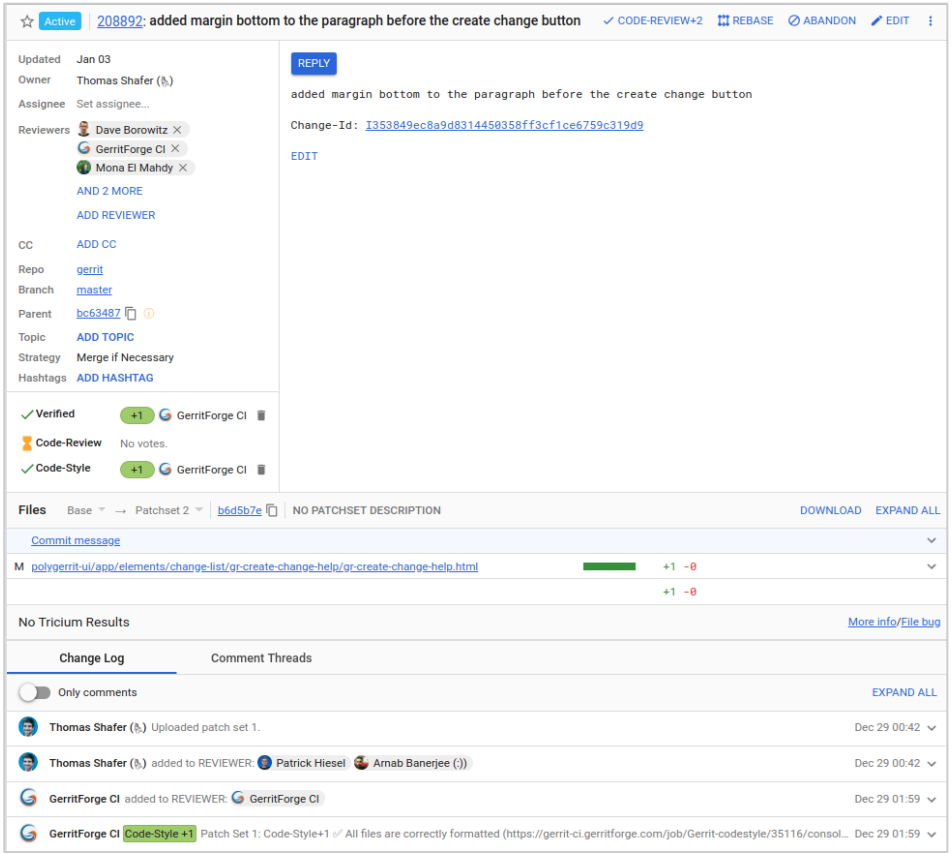
Commit author vs. Committer

- When you amend a commit you're *updating the committer aspects of that commit* and the *author aspects remain unchanged*. You can see both in Git log by using the ***--format=fuller*** argument:

```
$ git log --format=fuller
commit 9324ea7390b5c411c5cc050cf80965ce7425887a (HEAD -> foobar)
Author:     Adam Parkin <obfuscated@gmail.com>
AuthorDate: Fri Aug 6 11:37:15 2021 -0700
Commit:     Adam Parkin <obfuscated@gmail.com>
CommitDate: Fri Aug 6 11:37:15 2021 -0700

    Test commit
```

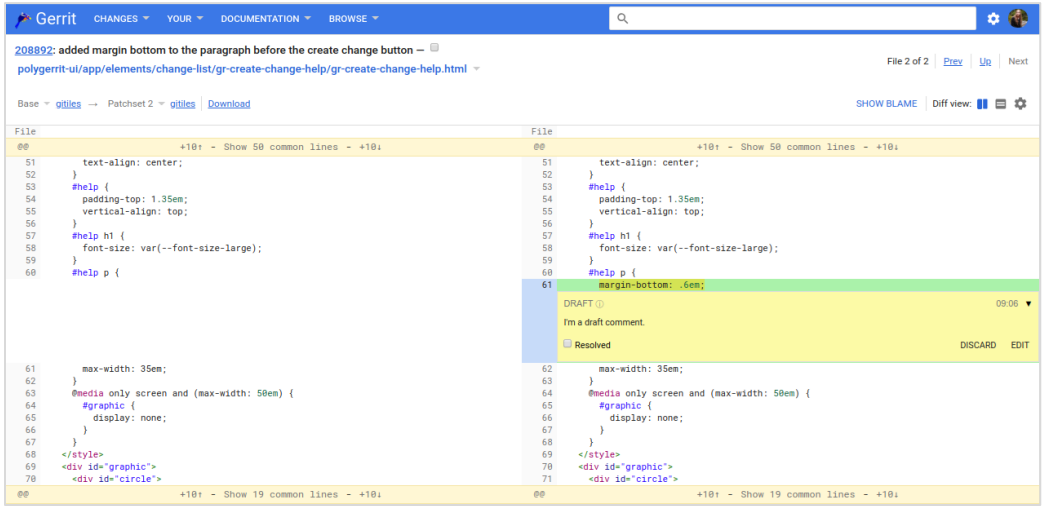
Review and Vote

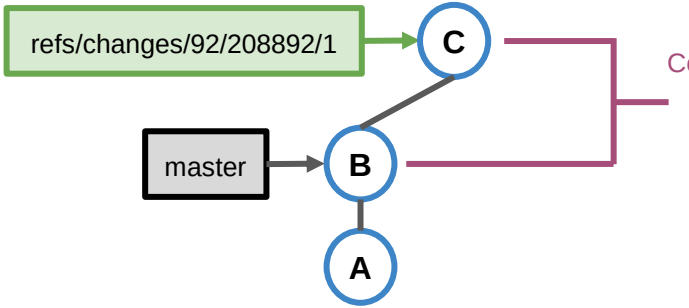


Changes can be inspected in the Gerrit WebUI:

- The ***change screen*** shows you all information about a change, including which files have been changed.

Review and Vote





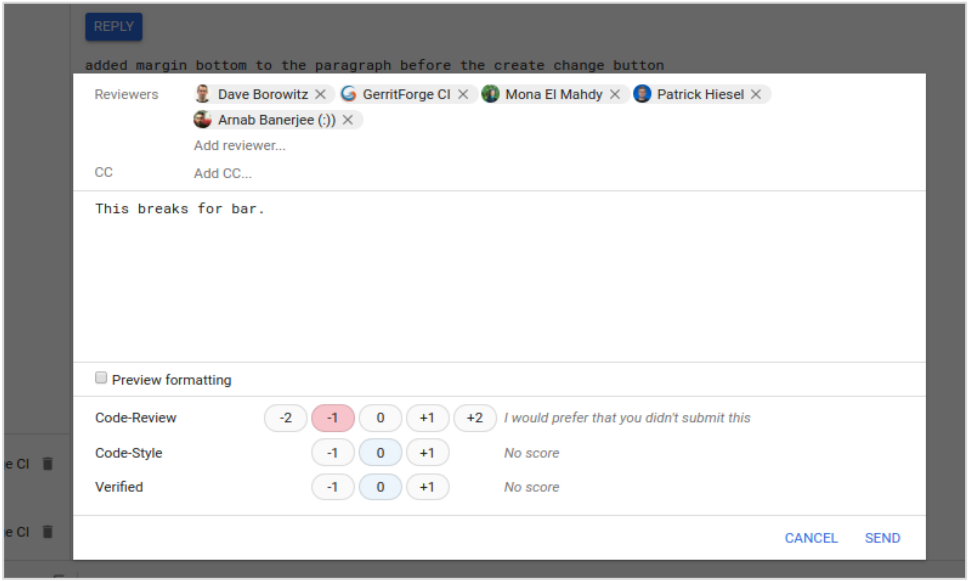
Compared versions:

- left side: commit ***B*** (***base/parent version***)
- right side: commit ***C***

Changes can be inspected in the Gerrit WebUI.

- The ***change screen*** shows you all information about a change, including which files have been changed.
- For each modified file you can review the ***file diff*** and comment inline on it, which creates ***unpublished draft comments***. You can also reply to existing comments.

Review and Vote



Changes can be inspected in the Gerrit WebUI.

- The ***change screen*** shows you all information about a change, including which files have been changed.
- For each modified file you can review the ***file diff*** and comment inline on it, which creates ***unpublished draft comments***. You can also reply to existing comments.
- The comments are published by ***replying*** on the change. The reply can include a general ***change message*** and you can give a ***voting*** on the change.

Voting

Voting is done on ***review labels***:

- Which ***review labels*** and ***voting values*** are available can be configured per repository, by default there is only the `Code-Review` label.
- Usually a change requires an ***approval*** (highest possible vote) for each label in order to become submittable.
- ***Veto votes*** (lowest possible value) block the submit of a change.
- ***Access rights*** control which user is allowed to vote on which review label
- Votings on some labels may be done automatically by bots.

Q: What can the change owner do if a negative vote is received?

Voting

- Rework the change and upload a new version of the change.
  OR
- ***Abandon*** the change.

Abandoning a change means that the modifications are discarded and the change doesn't get submitted.

Abandoned changes are still accessible and can be ***restored*** if needed.

Q: When you push a commit for code review how does Gerrit know if you push a new change or a new version of an existing change?

Change-Id

- ***Change-Id***:
  ID of a ***change*** that is set as footer in the commit message.

- Automatically generated and inserted on commit by a commit hook.

- If a commit is pushed that contains a `Change-Id` in the commit message Gerrit checks if a change with this `Change-Id` already exists.
  - If yes, this change is updated.
  - If not, a new change with that `Change-Id` is created.

The ***Gerrit commit-msg*** *hook* that generates and inserts ***Change-Ids*** on `git commit` must be installed once in a repository after it was cloned:

- The clone command that is offered by Gerrit in the WebUI includes the command to install the `commit-msg` hook.

Change-Id

```
First line is the subject, should be shorter than 70 chars

Separate the body from the subject by an empty line. The
commit message should describe why you are doing the
change. That's what typically helps best to understand
what the change is about. The details of what you changed
are visible from the file diffs.

The body can have as many paragraphs as you want. Lines
shouldn't exceed 80 chars. This helps command line tools
to render it nicely. Paragraphs are separated by empty
lines.

Change-Id: I351351fa6661010058d3684b2983f5b38bf3233d0f7
Bug: Issue 123
```

`Change-Id`:

- Format: ‘I’ + SHA1
- To be recognized by Gerrit the *`Change-Id`* must be contained in the **last paragraph** of the commit message (as all Git footers)

Q: What is a patch set?

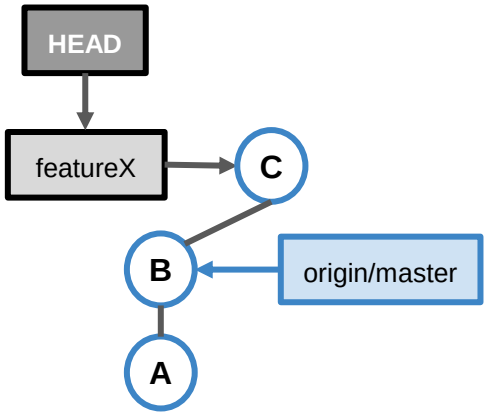
Patch Set

- ***Patch Set***: A version of a ***change***
- Correlates to a ***Git commit***.
- A change *contains one or more patch sets*.
- Each new patch set **replaces** the previous patch set, only **the latest patch set is relevant**.

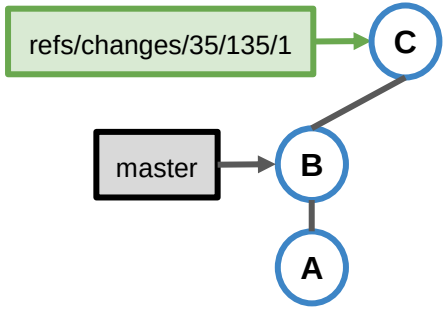
Often the term ***revision*** is used as synonym for ***patch set,*** or the Git commit of a patch set.

Push new Patch Set

local repository



remote repository



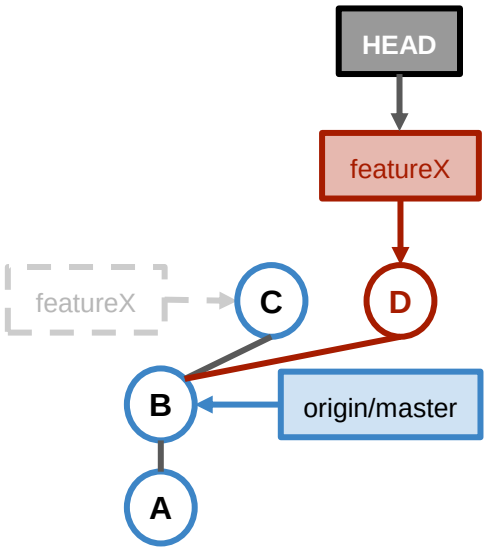
Situation:

- In the local `featureX` branch a commit ***C*** was done that was pushed for code review.
- During code review an issue was detected and the change should be reworked. The user has already checked out the `featureX` branch.

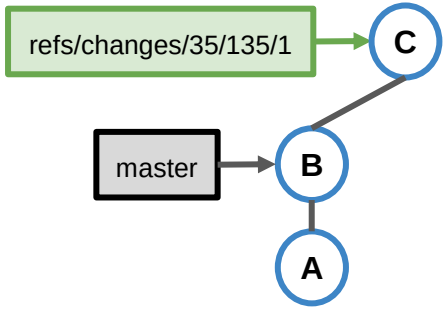
Q: How is a new a patch set created?

Push new Patch Set

local repository



remote repository

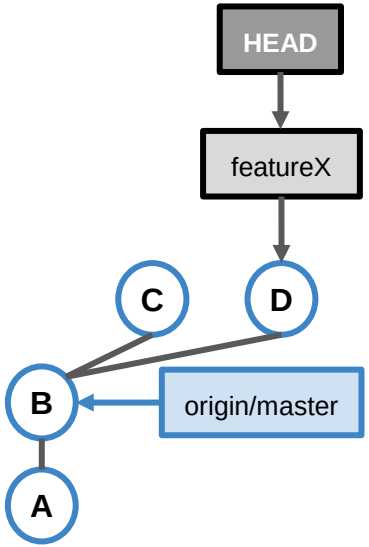


The user fixes the code and create a new commit by using `git commit --amend`:

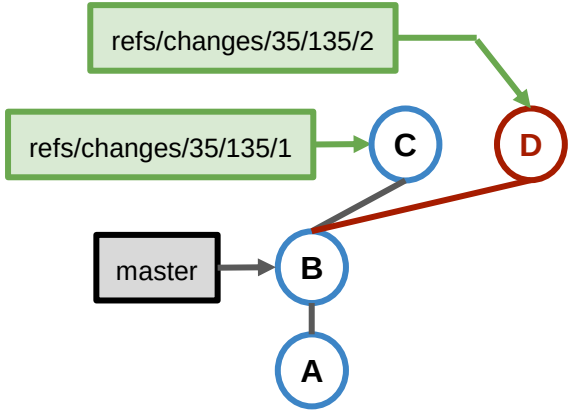
- A new commit ***D*** is created that is a sibling of the old commit ***C***.
- The commit message, including the `Change-Id`, is preserved.

Push new Patch Set

local repository



remote repository

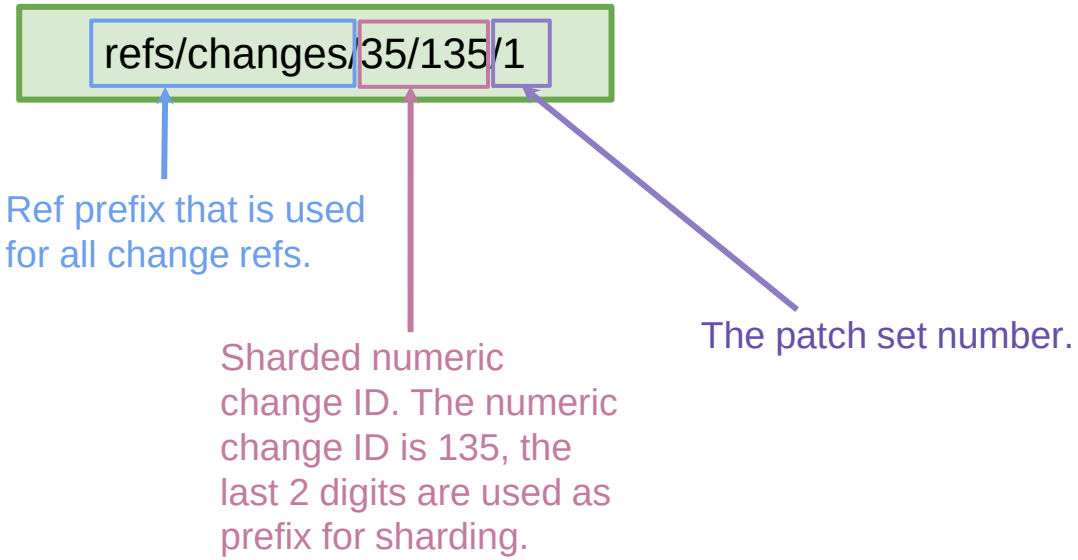



```
git push origin HEAD:refs/for/master
```

The new commit ***D*** is pushed for code review:

- Gerrit inspects commit ***D*** and finds the `Change-Id` in its commit message.
- Gerrit checks if for the target branch a change with that `Change-Id` already exists.
- Since a change with this `Change-Id` already exist Gerrit accepts commit ***D*** as a new patch set for this change and creates a new change ref for the second patch.
- The new patch set **replaces** the old patch set.
- The latest patch set on a change is called ***current patch set***.

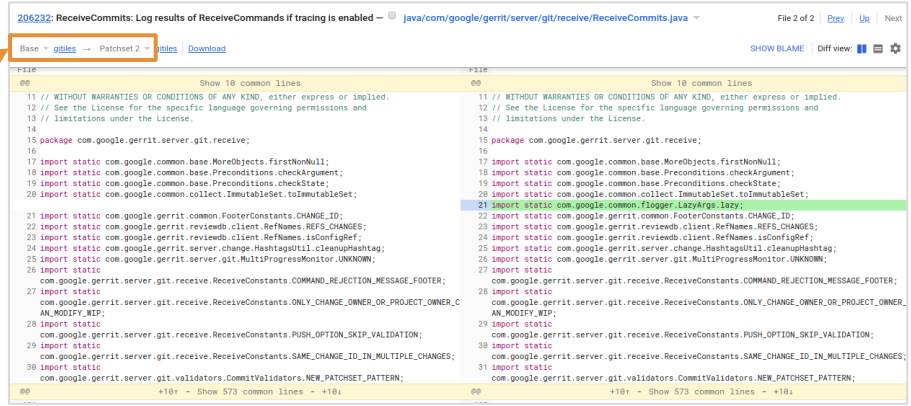
Change ref

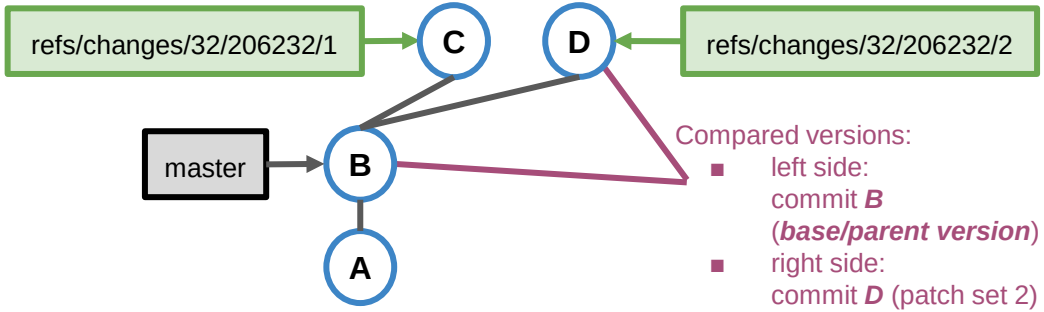


- All ***change refs*** share the same `refs/changes/` namespace. Refs in this namespace are not automatically fetched on `git clone` and `git fetch`.
- Can be fetched on need. Gerrit offers the fetch command on the change screen so that patch sets can be easily downloaded (e.g. to amend them and create a new patch set).

Review of new Patch Set

Patch Set Selector:





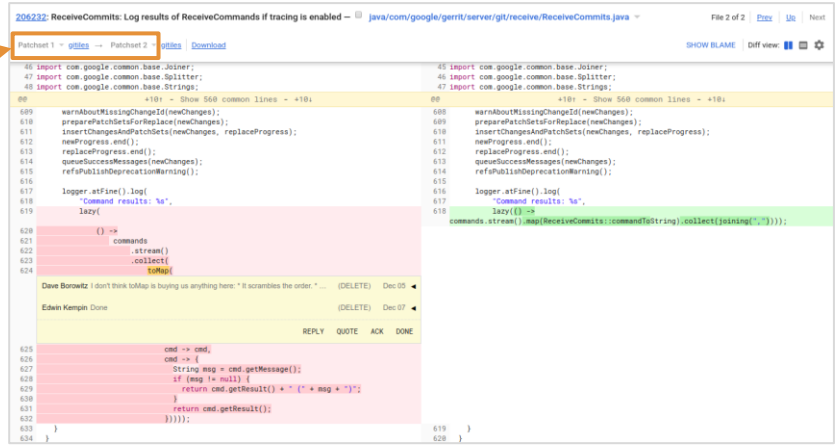
Compared versions:

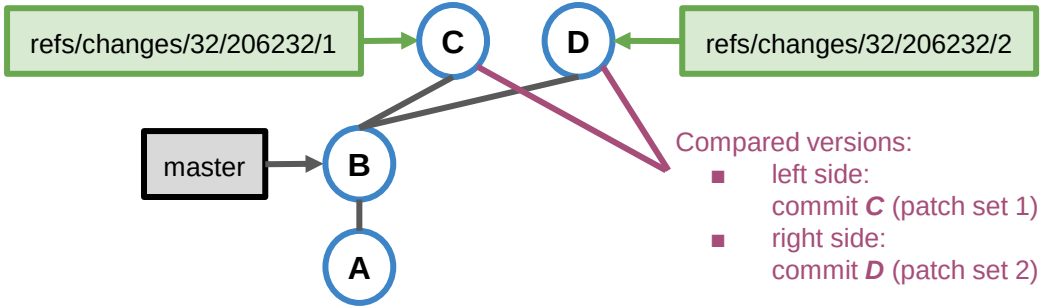
- left side: commit ***B*** (***base/parent version***)
- right side: commit ***D*** (patch set 2)

- Reviewers can see the full diff by comparing the new patch set against base version.

Comparing Patch Sets

Patch Set Selector:





Compared versions:

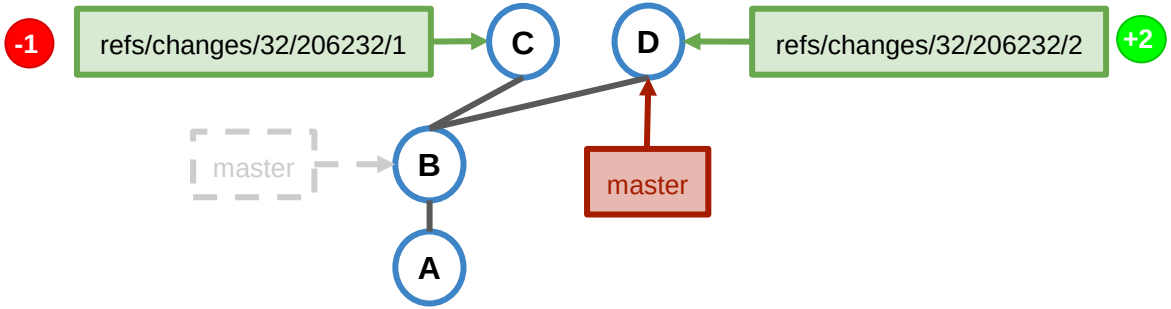
- left side:
  commit ***C*** (patch set 1)
- right side:
  commit ***D*** (patch set 2)

- Users that have previously reviewed the old patch set likely want to see what has changed with the new patch set. They can do so by comparing the old and new patch set.

Submit

Pre-conditions for submit:

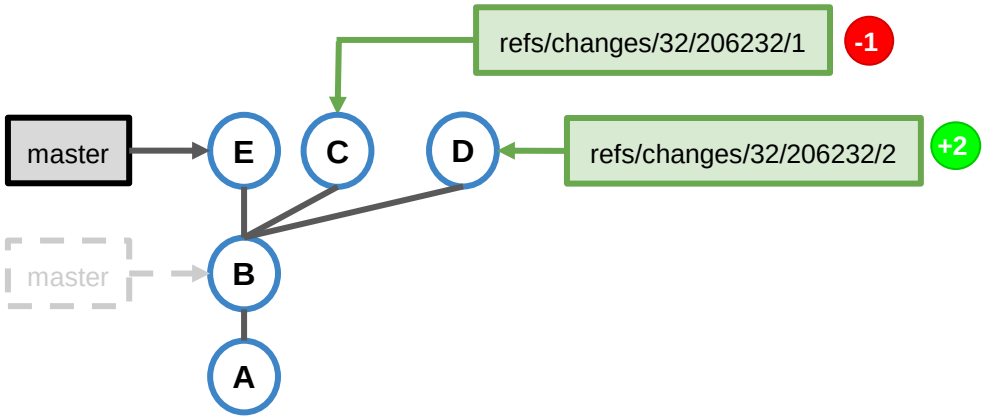
- The change has an ***approval*** (highest possible vote) for each ***review label***.
- None of the ***review labels*** has a ***veto vote*** (lowest possible vote)
- The change doesn't depend on other changes that are non-submittable.
- The user is allowed to submit.



Submit integrates a change into its target branch (more precisely integrates the current patch set into the target branch):

- The `master` branch is ***fast-forwarded*** to the commit that represents the current patch set.

Submit



Situation:

- A change has two patch sets (commit ***C*** and commit ***D***) which are both based on commit ***B***. The current patch set (commit ***D***) was approved and is submittable. In the meantime the `master` branch was updated to commit ***E***. Fast-forwarding `master` to the current patch set is not possible.

Q: What happens on submit if fast-forwarding the target branch is not possible?

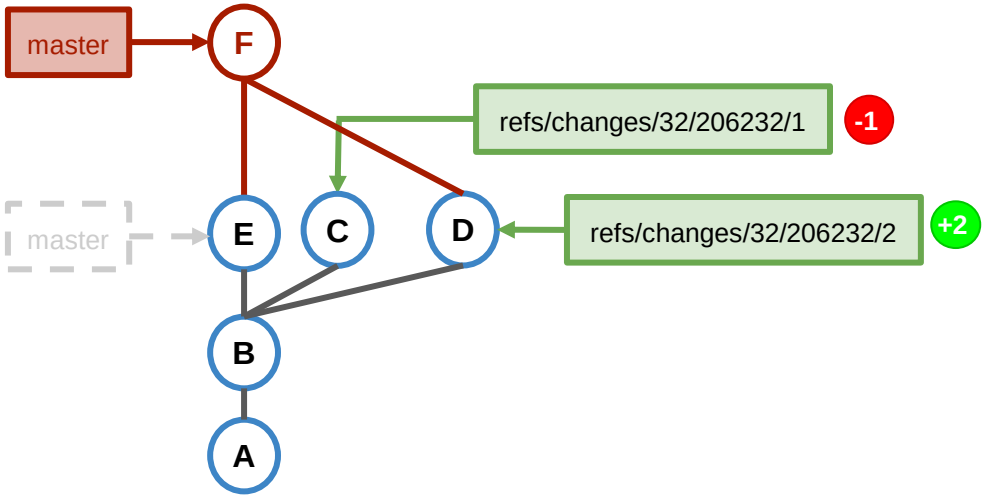
Submit Type / Submit Strategy

The behaviour on submit is configurable per repository:

- ***Submit Type*** / ***Submit Strategy***:
  - *`FAST_FORWARD_ONLY`*:
    Submit fails if fast-forward is not possible.
  - *`MERGE_IF_NECESSARY`*:
    If fast-forward is not possible, a merge commit is created.
  - *`REBASE_IF_NECESSARY`*:
    If fast-forward is not possible, the current patch set is automatically rebased (creates a new patch set which is submitted).
  - *`MERGE_ALWAYS`*:
    A merge commit is always created, even if fast-forward is possible.
  - *`REBASE_ALWAYS`*:
    The current patch set is always rebased, even if fast-forward is possible. For all rebased commits some additional footers will be added (*`Reviewed-On`*, *`Reviewed-By`*, *`Tested-By`*).
  - *`CHERRY_PICK`*:
    The change is cherry-picked. This ignores change dependencies. For all cherry-picked commits some additional footers will be added (*`Reviewed-On`*, *`Reviewed-By`*, *`Tested-By`*).

- Recommended setting:
  - Submit type:
    `MERGE_IF_NECESSARY`
    or
    `REBASE_IF_NECESSARY`
  - *`Allow content merges: true`*

Submit - MERGE_IF_NECESSARY

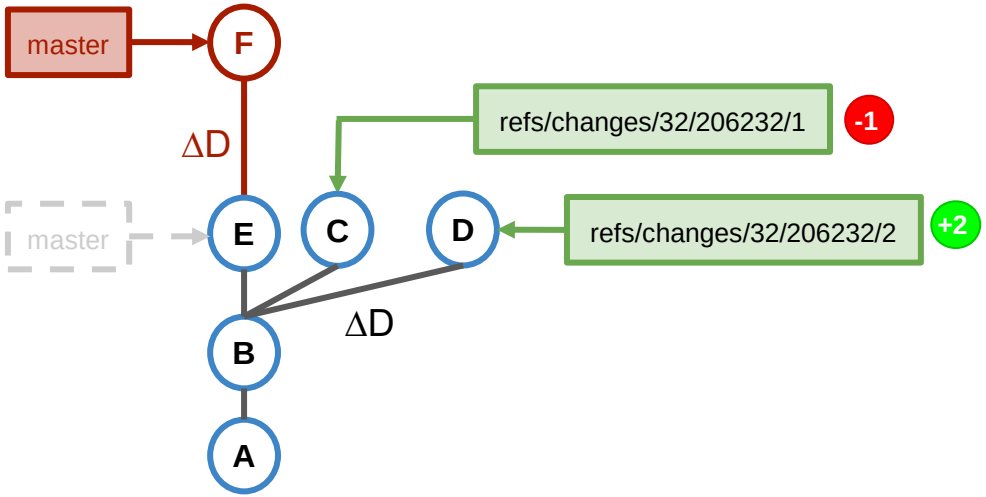


Since a fast-forward of the target branch is not possible a merge commit is created :

- The target branch is then fast-forwarded to the merge commit.
- The merge may fail due to conflicts.

Q: How would the result look like with REBASE_IF_NECESSARY?

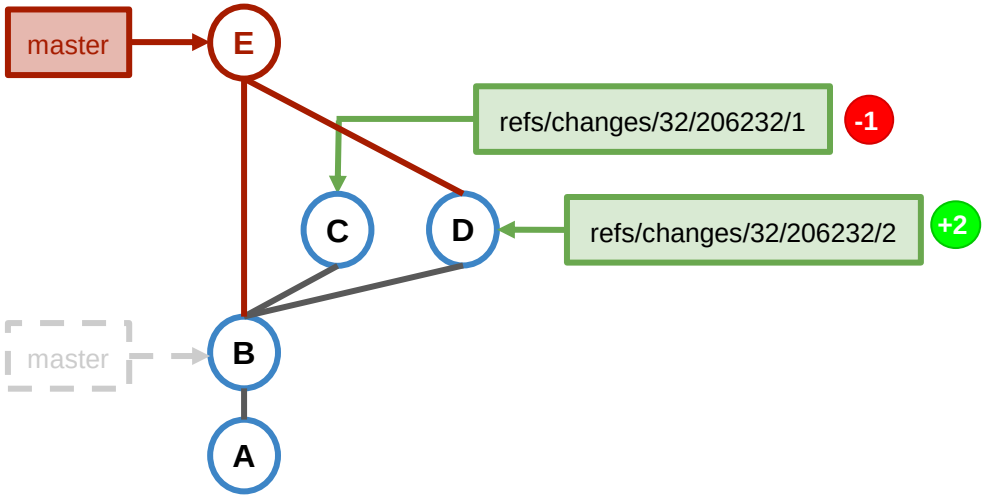
Submit - REBASE_IF_NECESSARY



Since a fast-forward of the target branch is not possible the current patch set, commit ***D***, is rebased which creates patch set ***F***:

- The target branch is then fast-forwarded to the merge commit.
- The rebase may fail due to conflicts.
- Results in linear history.

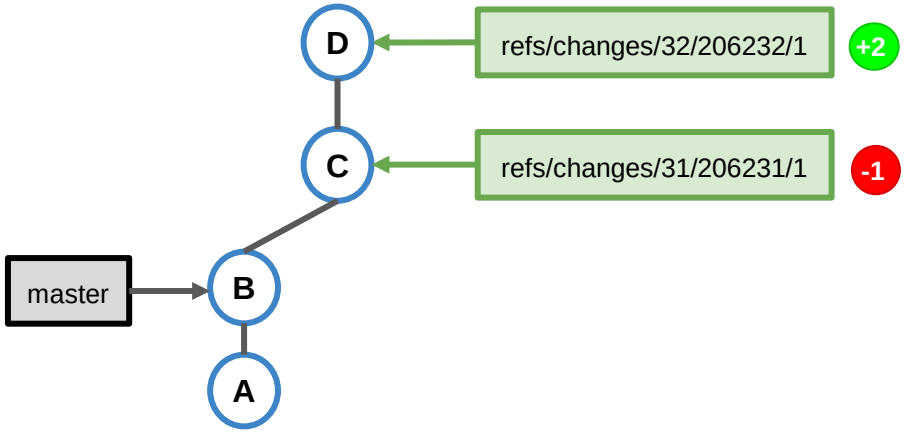
Submit - MERGE_ALWAYS



Although a fast-forward of the target branch to the current patch set, commit ***D***, is possible a merge commit ***F*** is created:

- The target branch is then fast-forwarded to the merge commit.
- Since a merge commit is always created you can always see from the version graph when a change got submitted (commit timestamp of the merge commit).

Submit - CHERRY_PICK

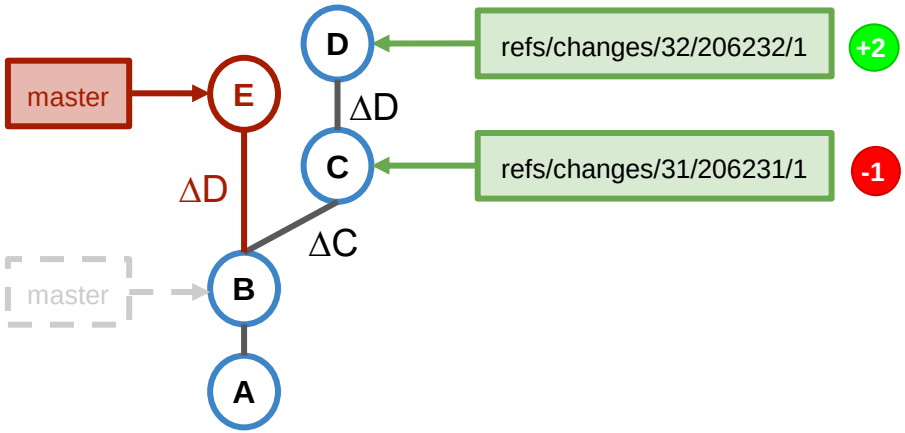


Situation:

- There is a change series with two changes, the change for commit ***D*** depends on the change for commit ***C***.
- The change for commit ***D*** was approved, the change for commit ***C*** got a negative review and needs to be reworked. Hence the change for commit ***C*** is not submittable.

Q: What happens on submit of the change for commit D?

Submit - CHERRY_PICK



- The change for commit ***D*** is submittable since the `CHERRY_PICK` submit strategy **ignores change dependencies** (with all other submit strategies the change would be non-submittable because it depends on a non-submittable change).
- The current patch set of the submitted change is cherry-picked and the target branch is fast-forwarded to it.
- The cherry-pick may fail with conflicts.