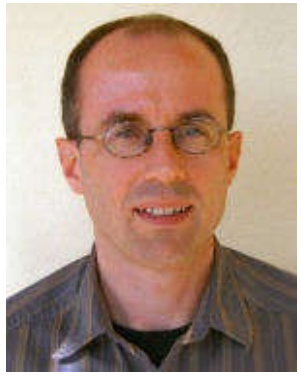
Effective Git

Chris Aniszczyk(Redhat)



Stefan Lay (SAP)

Shawn Pearce (Google)

Matthias Sohn (SAP)

http://eclipse.org/egit
http://code.google.com/p/gerrit

Git

… a distributed revision control system built by the Linux project to facilitate code review

Distributed means no central repository

- No central authority!
- Easy offline usage
- Easy to branch a project
- Protected against manipulation by cryptographic hashes

Really good at merging

- Coordination only needed "after the fact”
- Easier to rejoin (or refresh) branches

Structured around commits (i.e. patches)

- Tools for identifying problem commits (git bisect)
- Tools for restructuring branches w/ specific commits

Git at Eclipse

Eclipse defined a roadmap to move to Git
CVS has been deprecated

EGit is an Eclipse Team provider for Git

- http://www.eclipse.org/egit/

JGit is a lightweight Java library implementing Git

- http://www.eclipse.org/jgit/

The goal is to build an Eclipse community around Git.

EGit/JGit are still beta and we want to establish a feedback loop to improve the tooling.

Modern Code Review – What is it ? Guido van Rossum [1]

When one developer writes code, another developer is asked to review that code

A careful line-by-line critique

Happens in a non-threatening context

Goal is cooperation, not fault-finding

Integral part of coding process

Otherwise this will happen:
Debugging someone else's broken code
– **Involuntary code review:** Not so good; emotions may flare

[1] http://code.google.com/p/rietveld/downloads/detail?name=Mondrian2006.pdf

Code Review – Benefits Guido van Rossum [1]

Four eyes catch more bugs

- Catch bugs early to save hours of debugging

Mentoring of new developers / contributors

- Learn from mistakes without breaking stuff

Establish trust relationships

- Prepare for more delegation

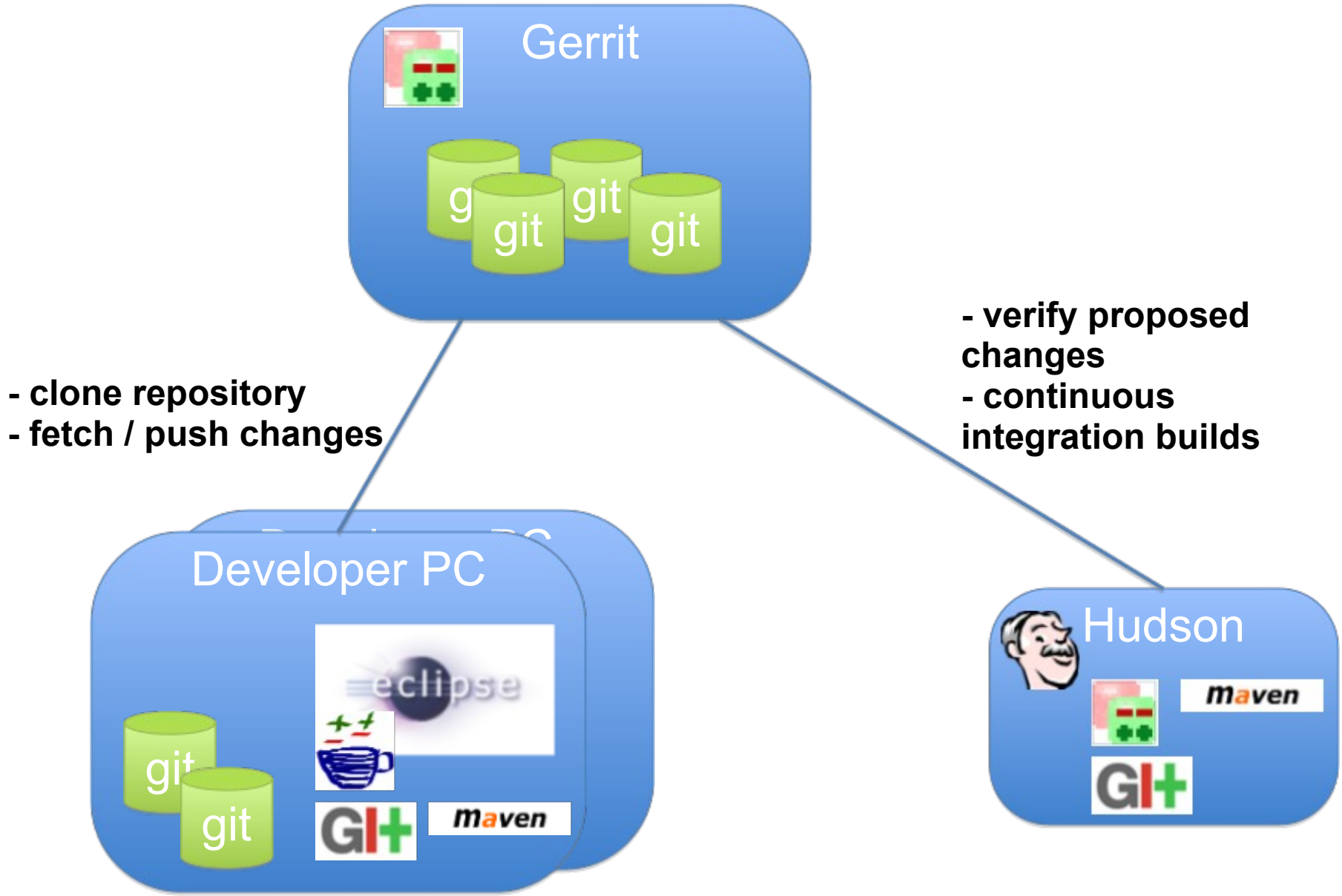
Good alternative to pair programming

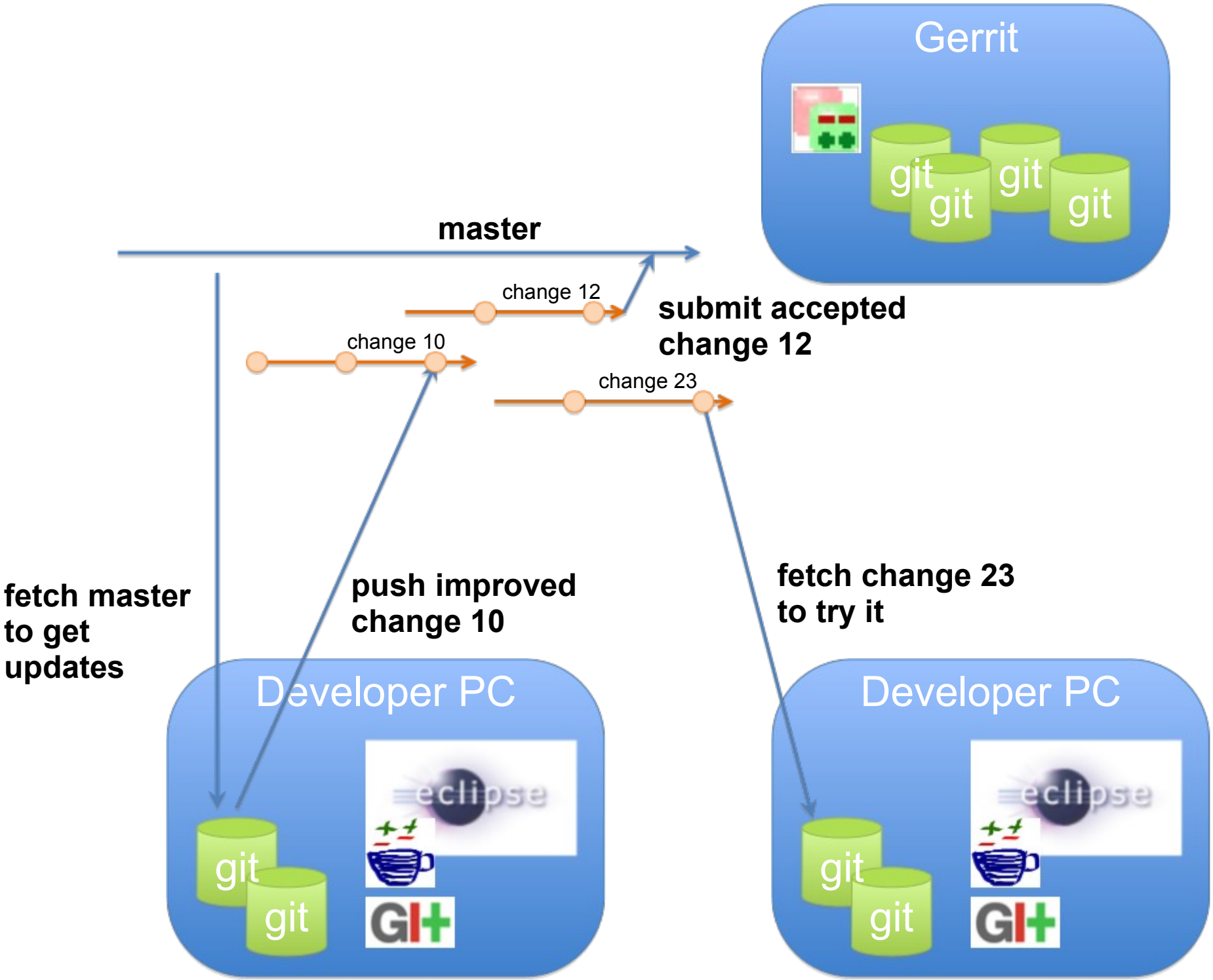
- asynchronous and across locations

Coding standards

- Keep overall readability & code quality high

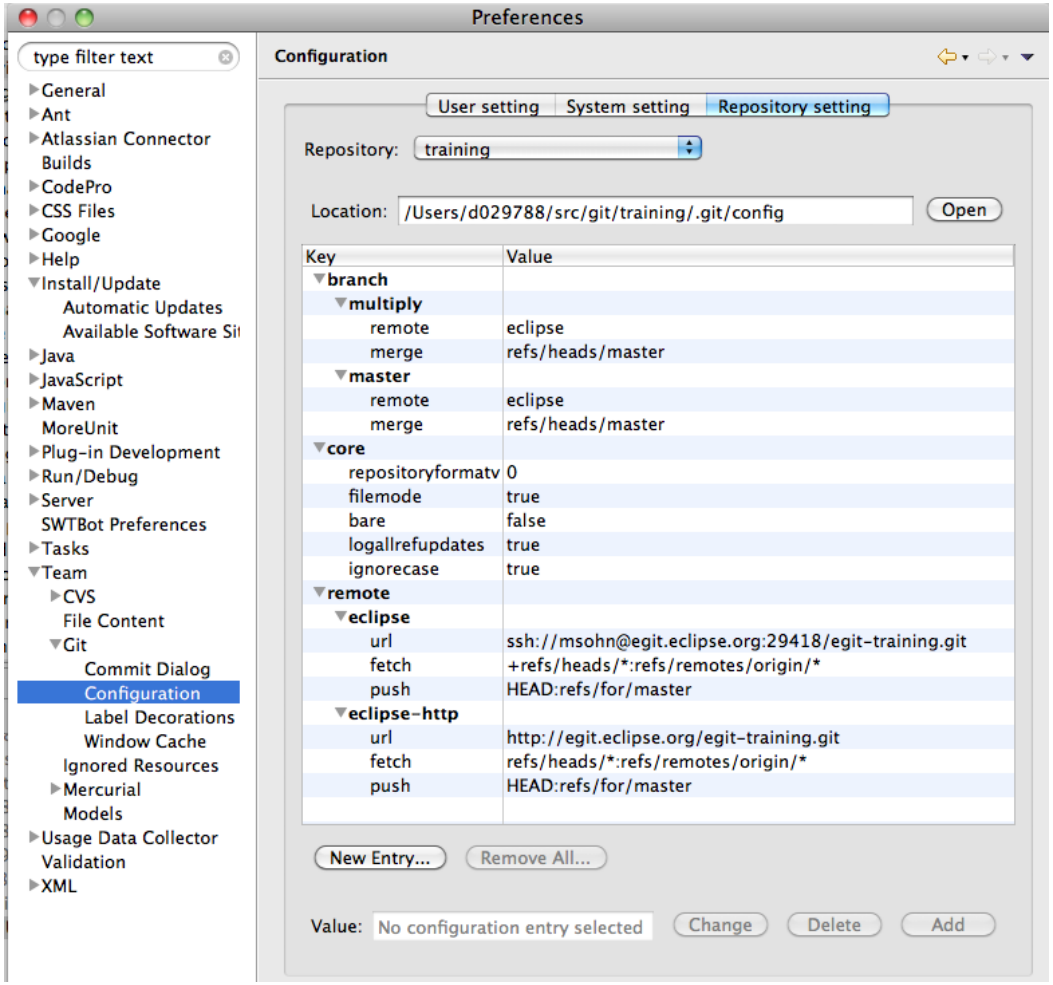
[1] http://code.google.com/p/rietveld/downloads/detail?name=Mondrian2006.pdf

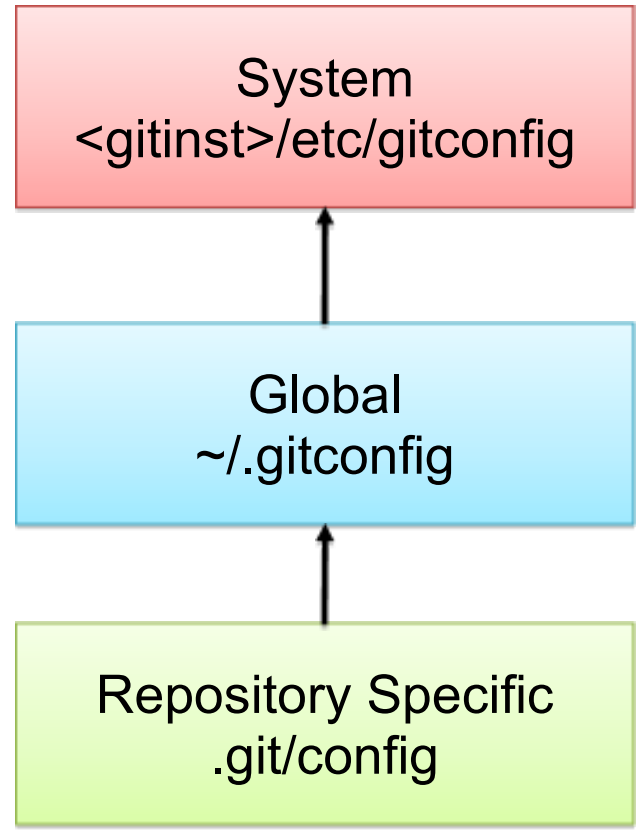

Gerrit
git
git
git
git
- clone repository
- fetch / push changes
- verify proposed changes
- continuous integration builds
Developer PC
eclipse
git
git
maven
Hudson
maven


Gerrit
git
git
git
git
master
change 12
change 10
change 23
submit accepted
change 12
fetch master
to get
updates
push improved
change 10
fetch change 23
to try it
Developer PC
git
git
eclipse
Developer PC
git
git
eclipse

Git Configuration

Git Concepts – config files



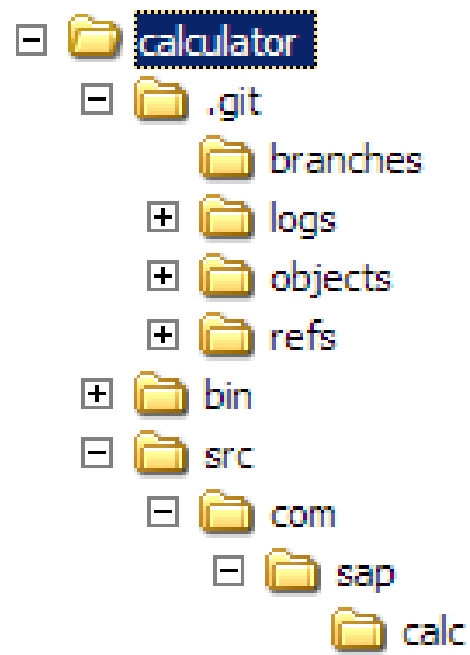


- `git config -l`
- `git config -e`
- `--system`
- `--global`

Basic Concepts

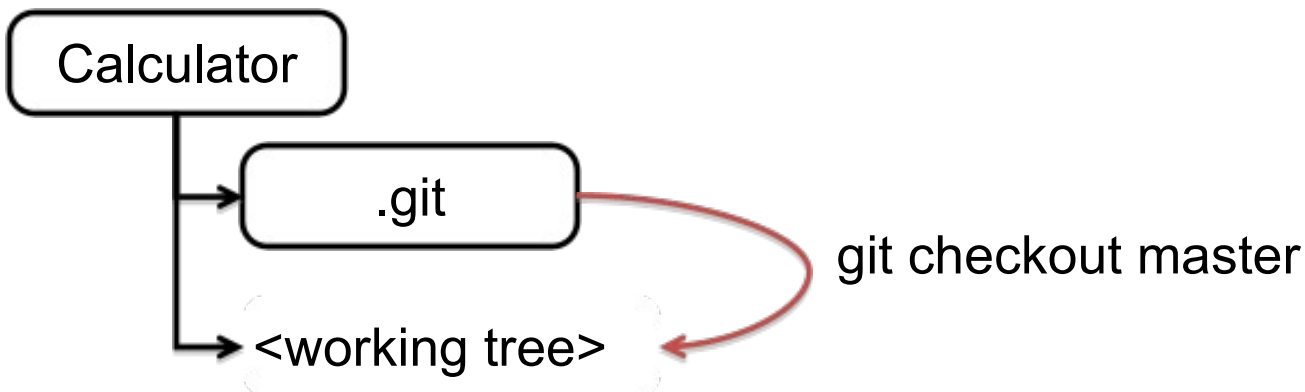
Making Changes

- Structure in the file system
- One working tree per repo
- `.git` folder is the Git repo
- Files/folders under the parent of `.git` are the working tree



Making Changes

- **Checkout:** populate working tree with the commit you want to start working from
  - most of the time you will checkout a branch
  - ➔ checkout the commit pointed to by the branch (aka: tip of the branch)
  - per file checkout means revert !



Making Changes

- **Just start doing your changes**
  - modify, add, delete files
- **Tell Git which changes you intend to commit**
  - **`git add`**
  - EGit helps by auto-detecting what you changed

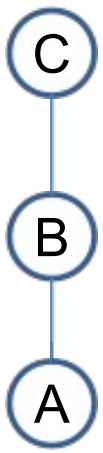
Commiting Changes

`git commit`

- **Provide a commit message**
  - First line is header
  - separated by a blank line follows the body
  - last paragraph is for meta data in **key: value** format
- **commit represents a version of the complete repository**
- **commits are identified by a globally unique ID (SHA1)**
- If two Git repos both contain a commit with the same ID then the content in these two commits is identical

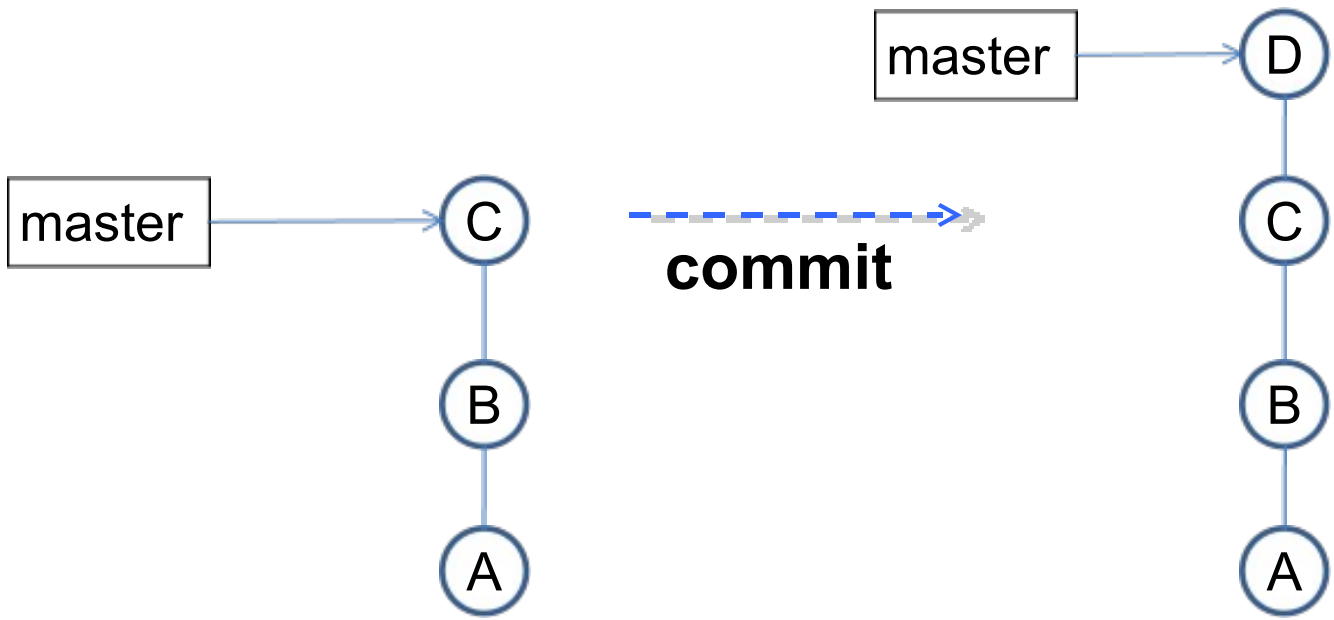
Commits

- **Commit history**
  - B is successor of A
  - C is successor of B



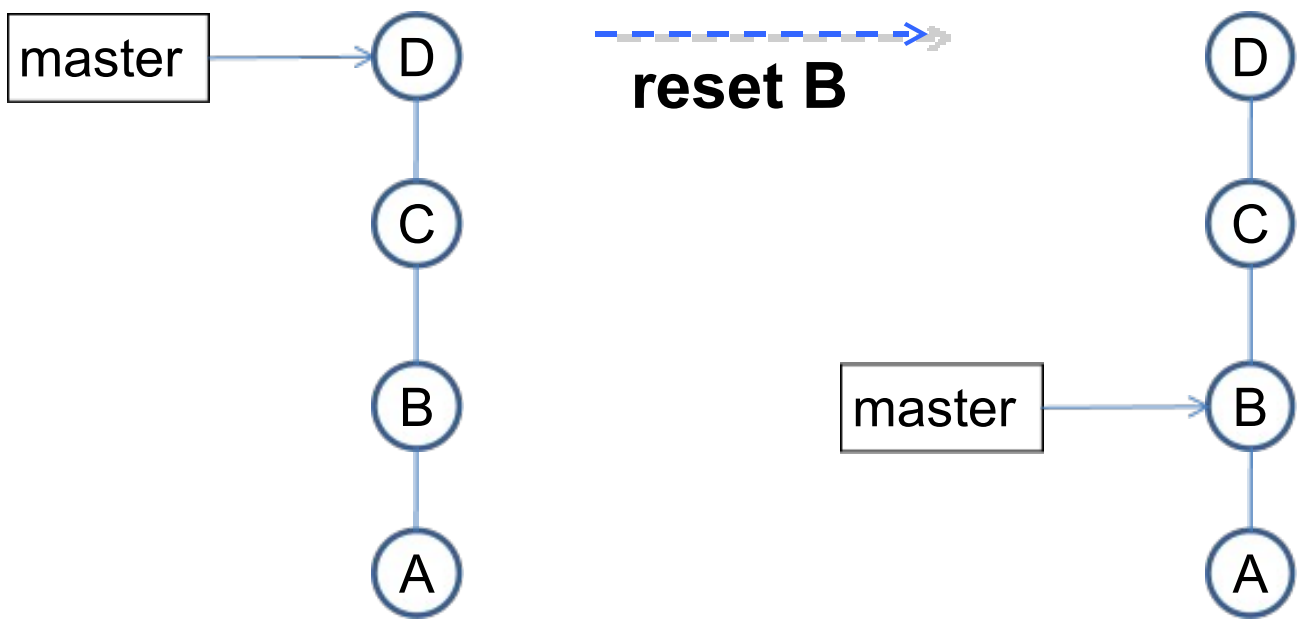
Branches

- Branch is a named pointer to a commit
- Commit command moves the pointer



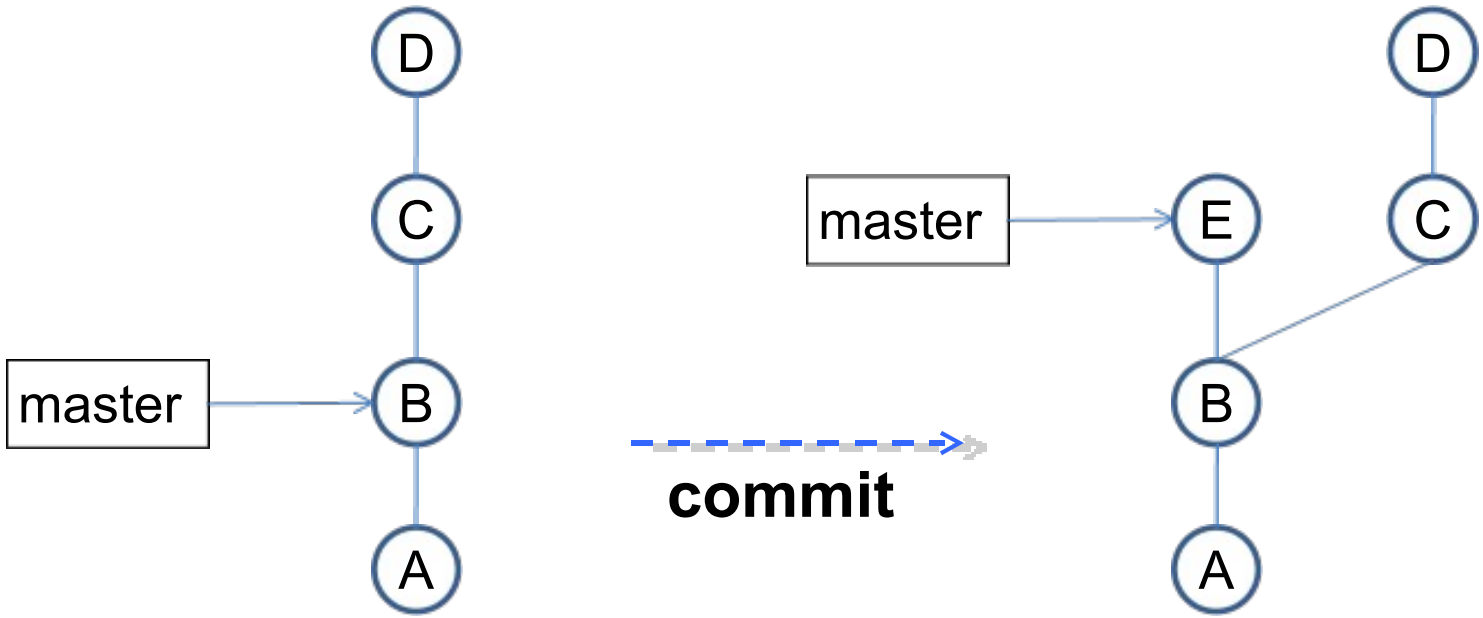
Branches

- **The (branch) pointer can also be moved "manually" to any commit**
  - **`git reset`**



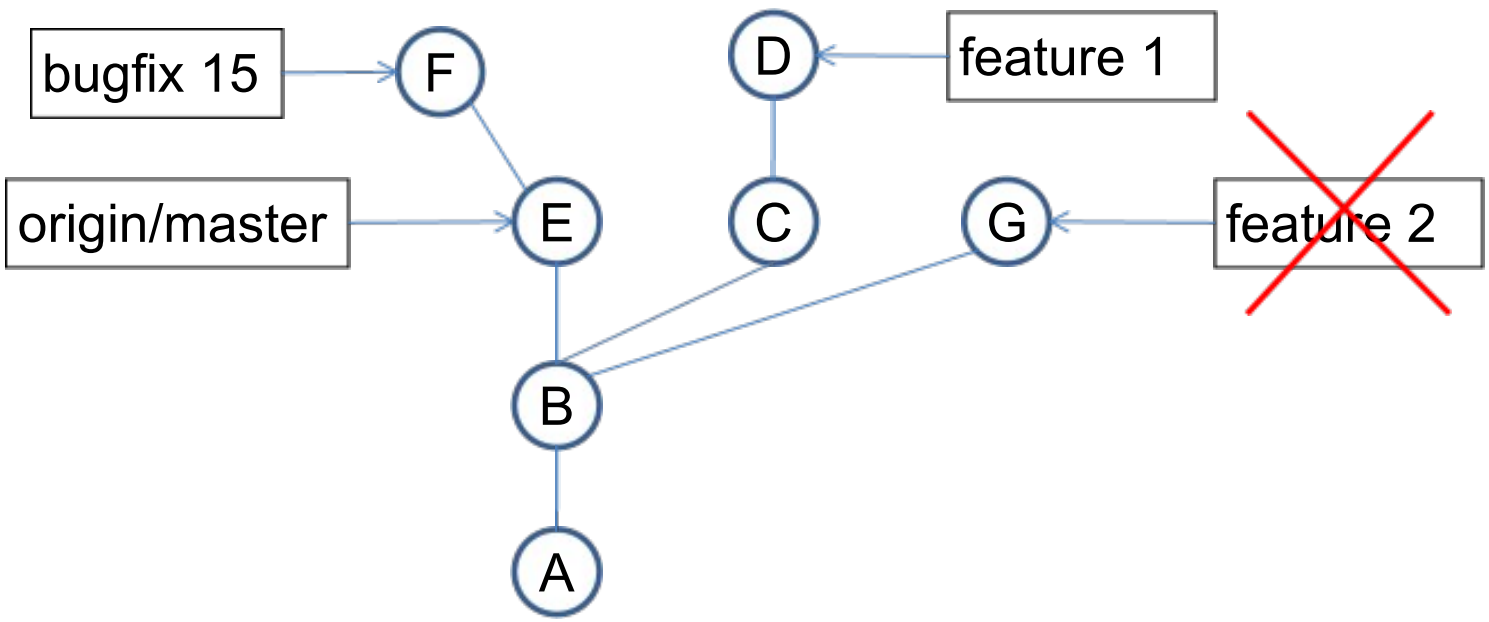
Branches

- **What happens on next `git commit`?**
- **C and D continue to exist but they are not in the history or the master branch**



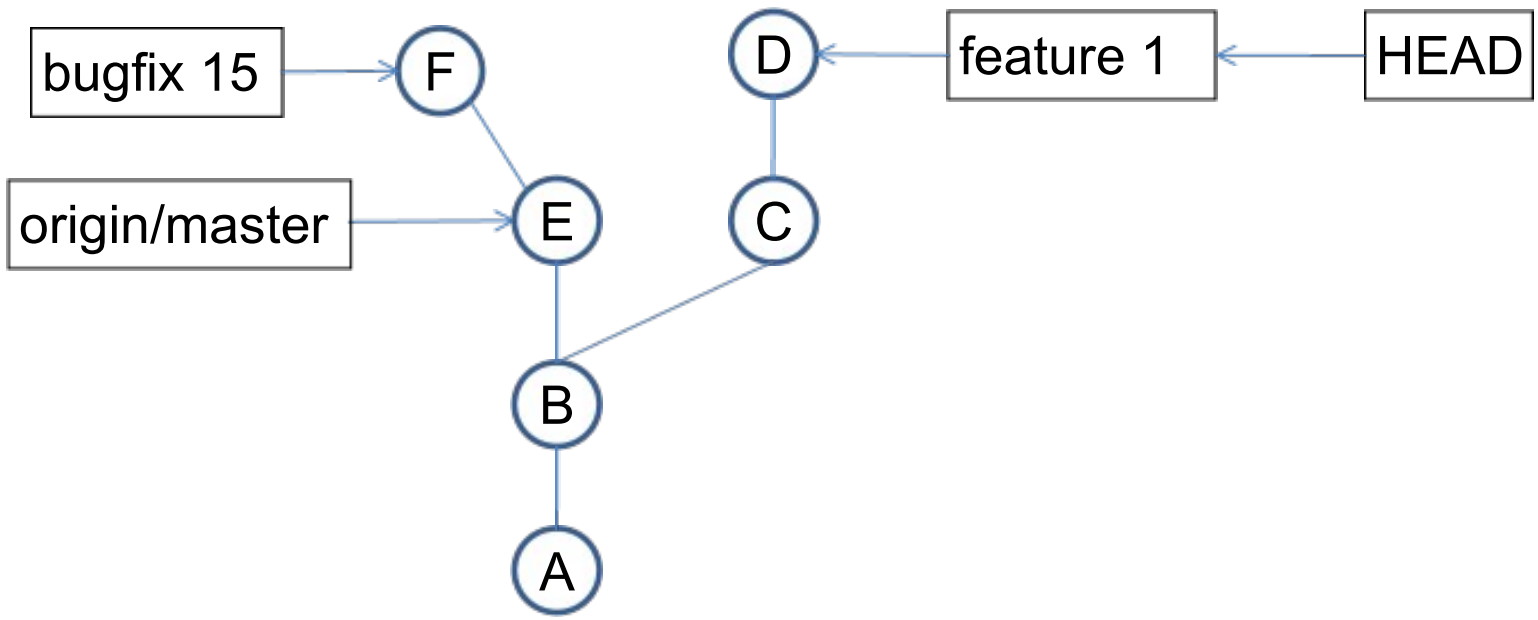
Branches

- **Usually there are many branches in a Git repository**
- **Branches can also be deleted**



HEAD

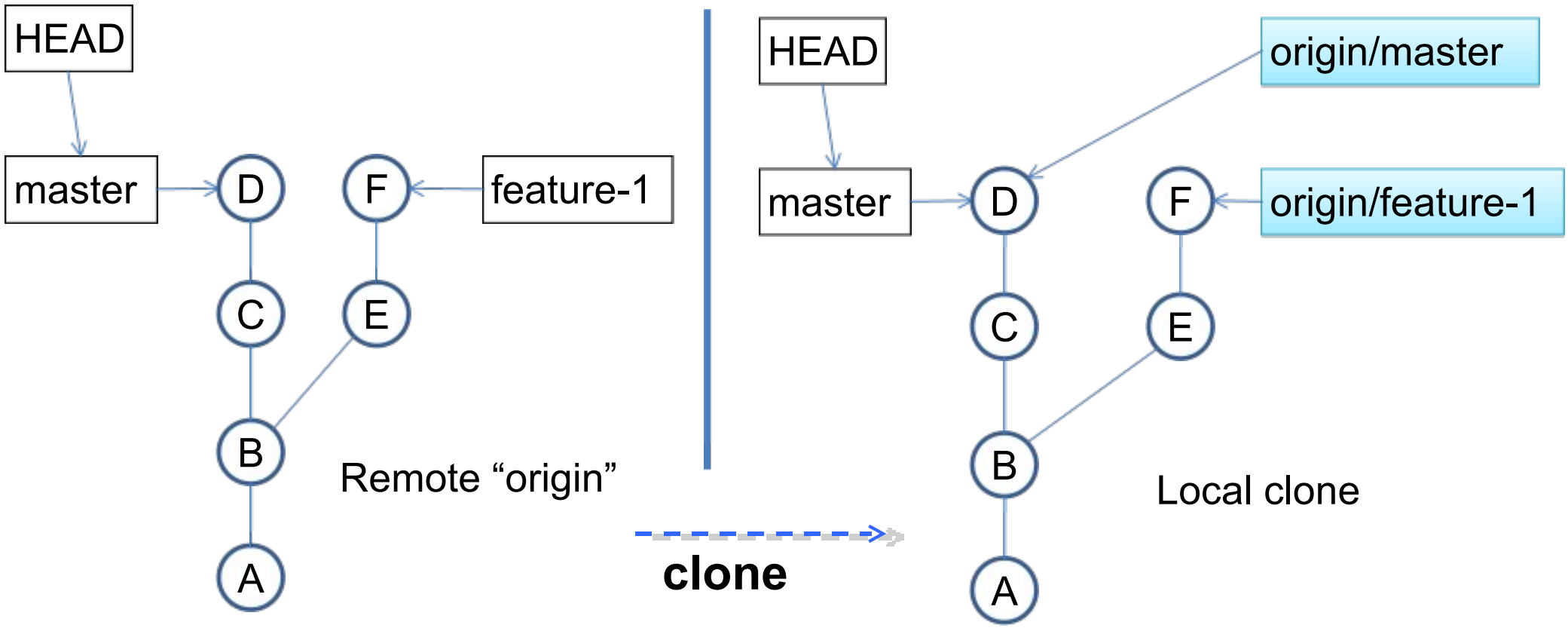
- **HEAD – pointer to a branch**
- **Means:**
  “Current Branch” – the branch which you have checked out



Cloning & Fetching

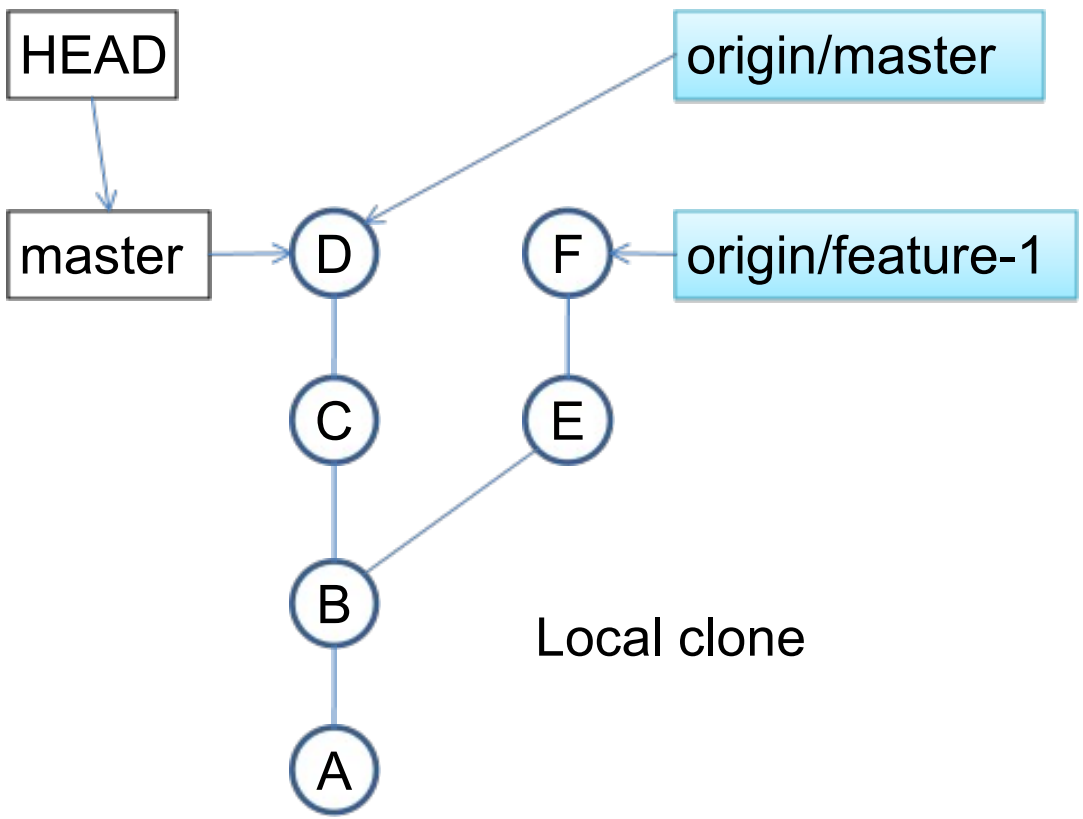
Clone Remote Repository

- **Git is a distributed versioning system**
  - **`git clone <remote-repo>`**
  - **cloned repo gets local name “origin” (by default)**



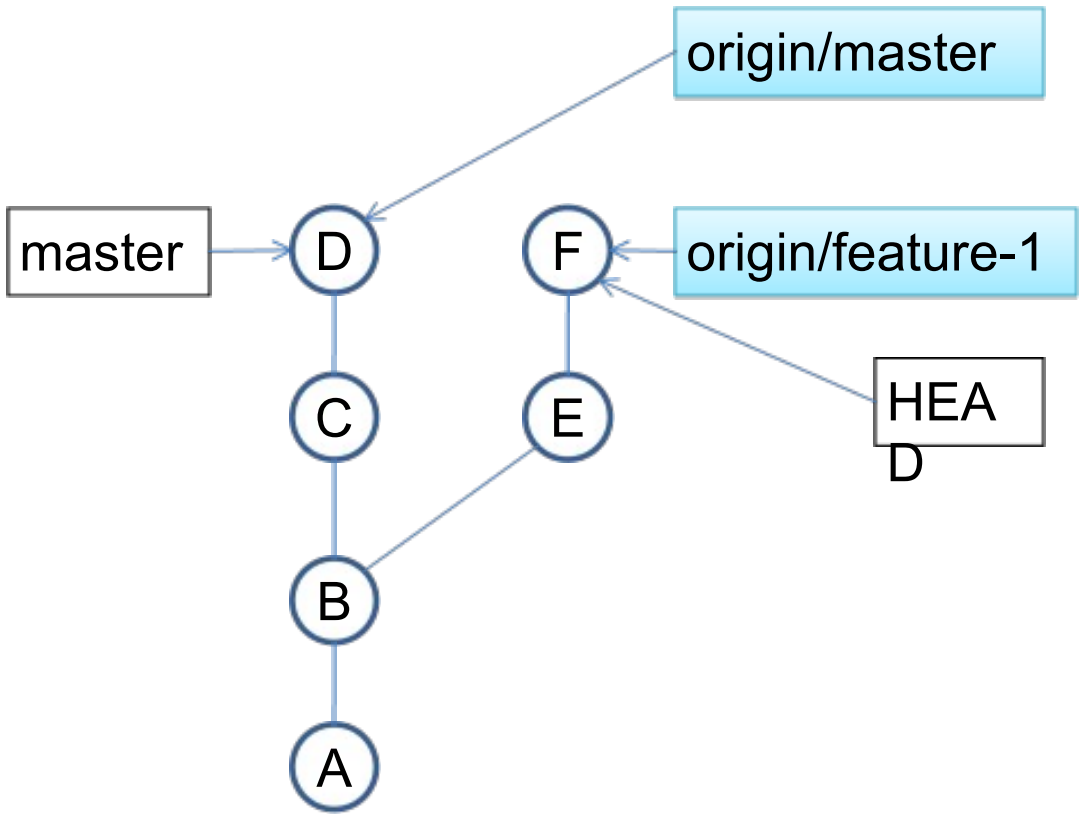
Clone Remote Repository

- **Remote tracking branches, full names:**
  - **remotes/origin/master**
  - **remotes/origin/feature-1**



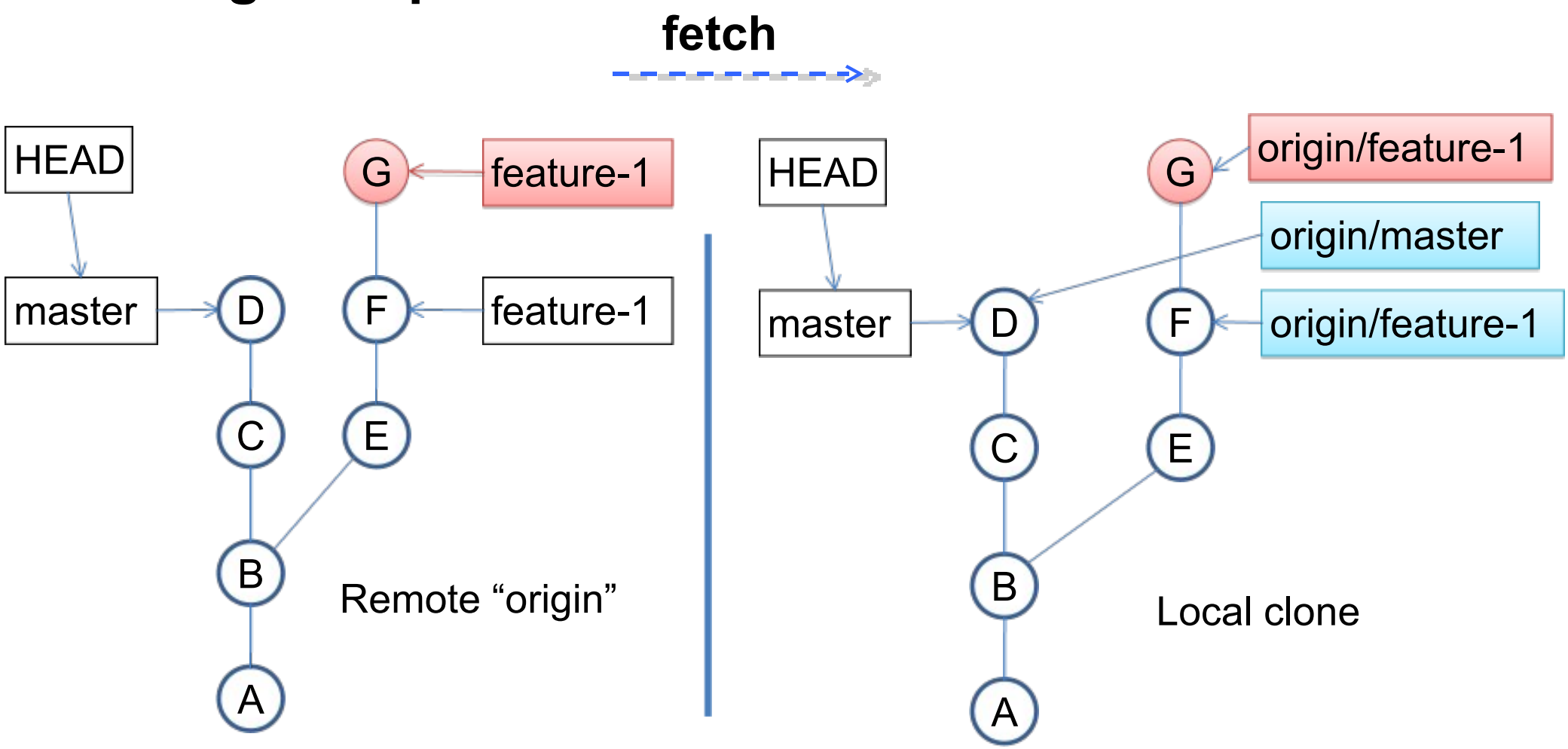
Remote Tracking Branches

- **Just like any other branch, but read-only**
  - **possible: git checkout origin/feature1**
  - **However, HEAD gets detached!**



Fetch

- **`git fetch` will update all remote tracking branches to reflect the changes done in the "origin" repo**



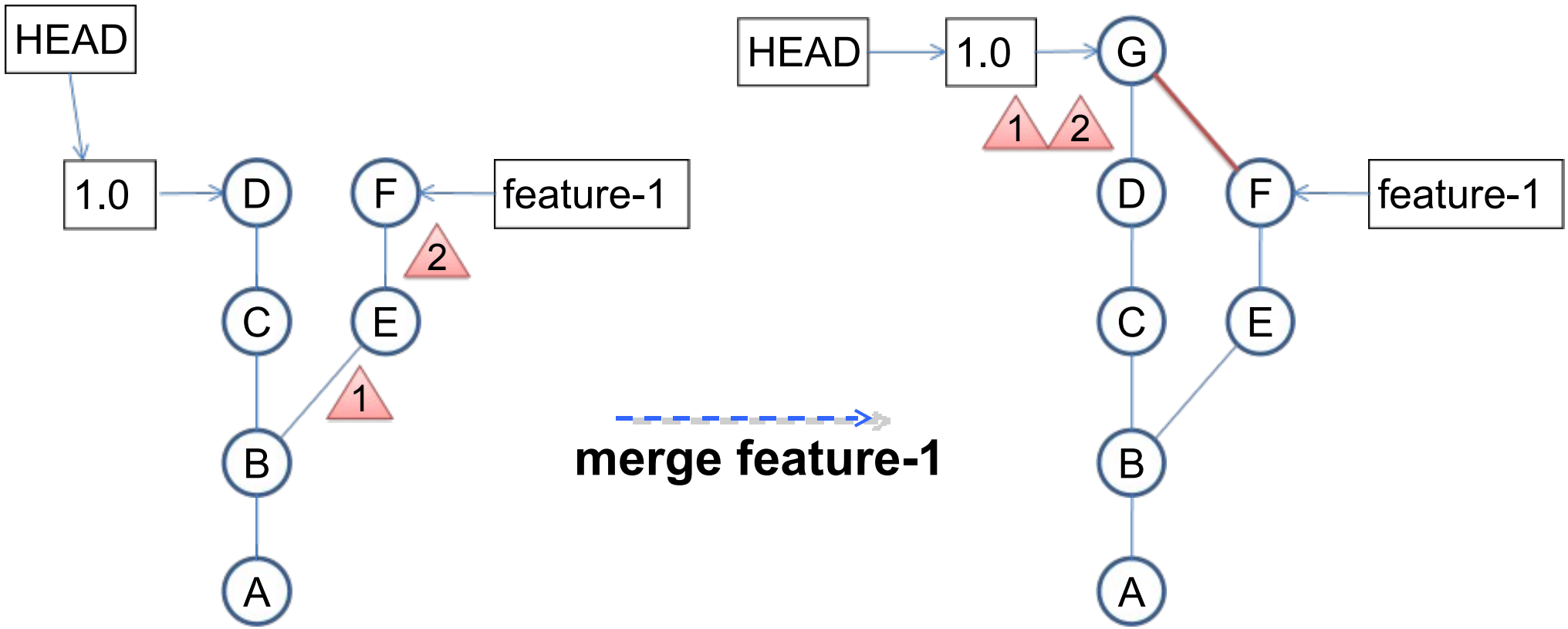
Fetch

- Always safe to do
- Updates only remote tracking branches
- Does never change local branches

Merge & Rebase

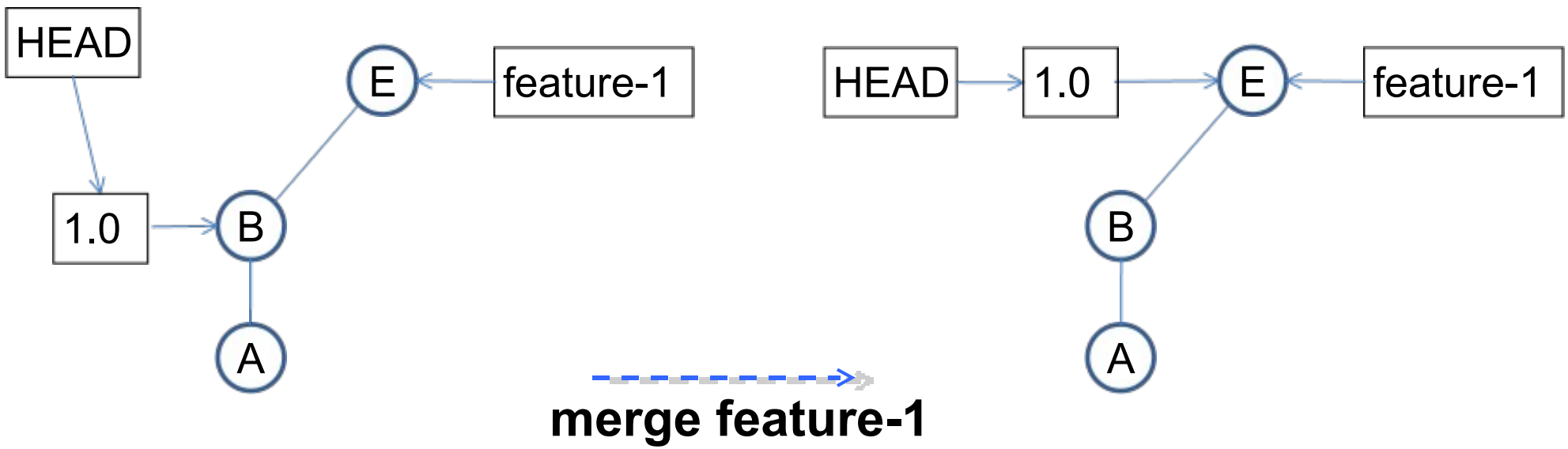
Merge

- **git merge feature-1**
  - **will replay all changes done in feature-1 since it diverged from 1.0 (E and F)**



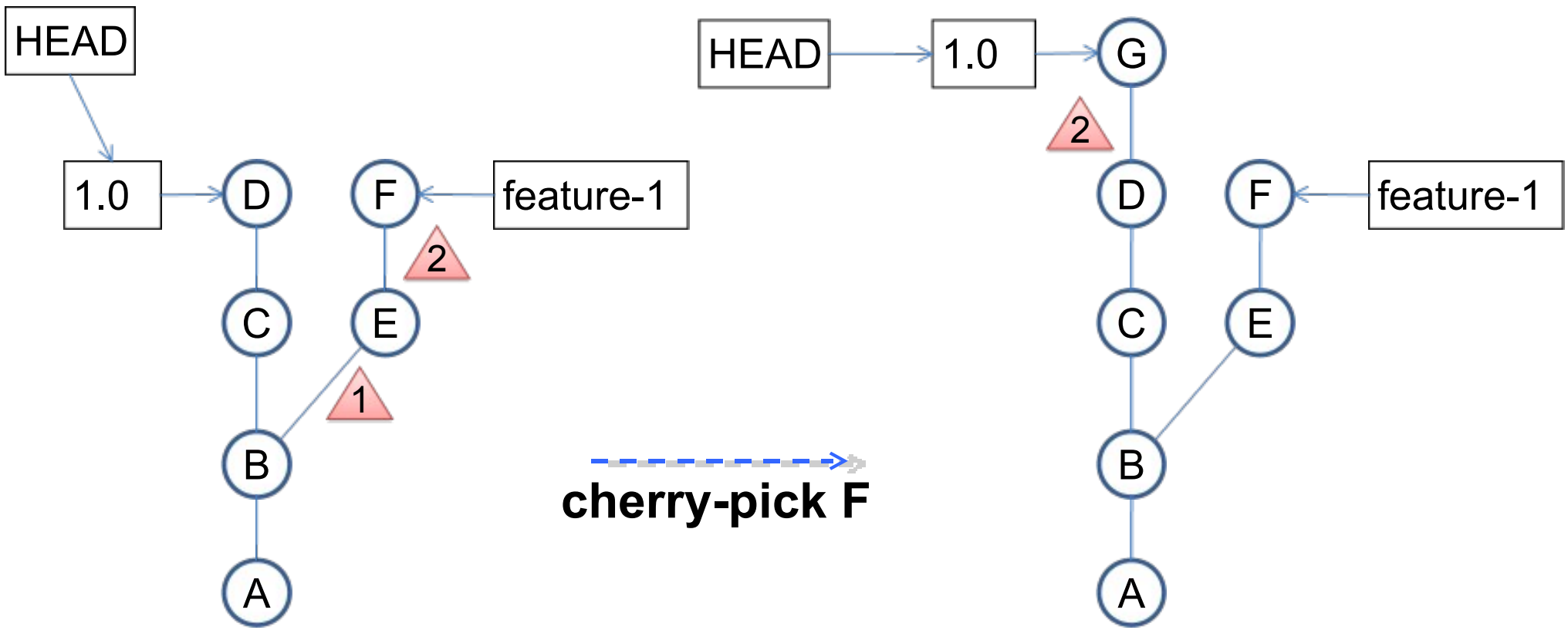
Merge

- **Easy Case - Fast Forward**
  - `git merge feature-1`
  - **no new commit, just move the pointer**



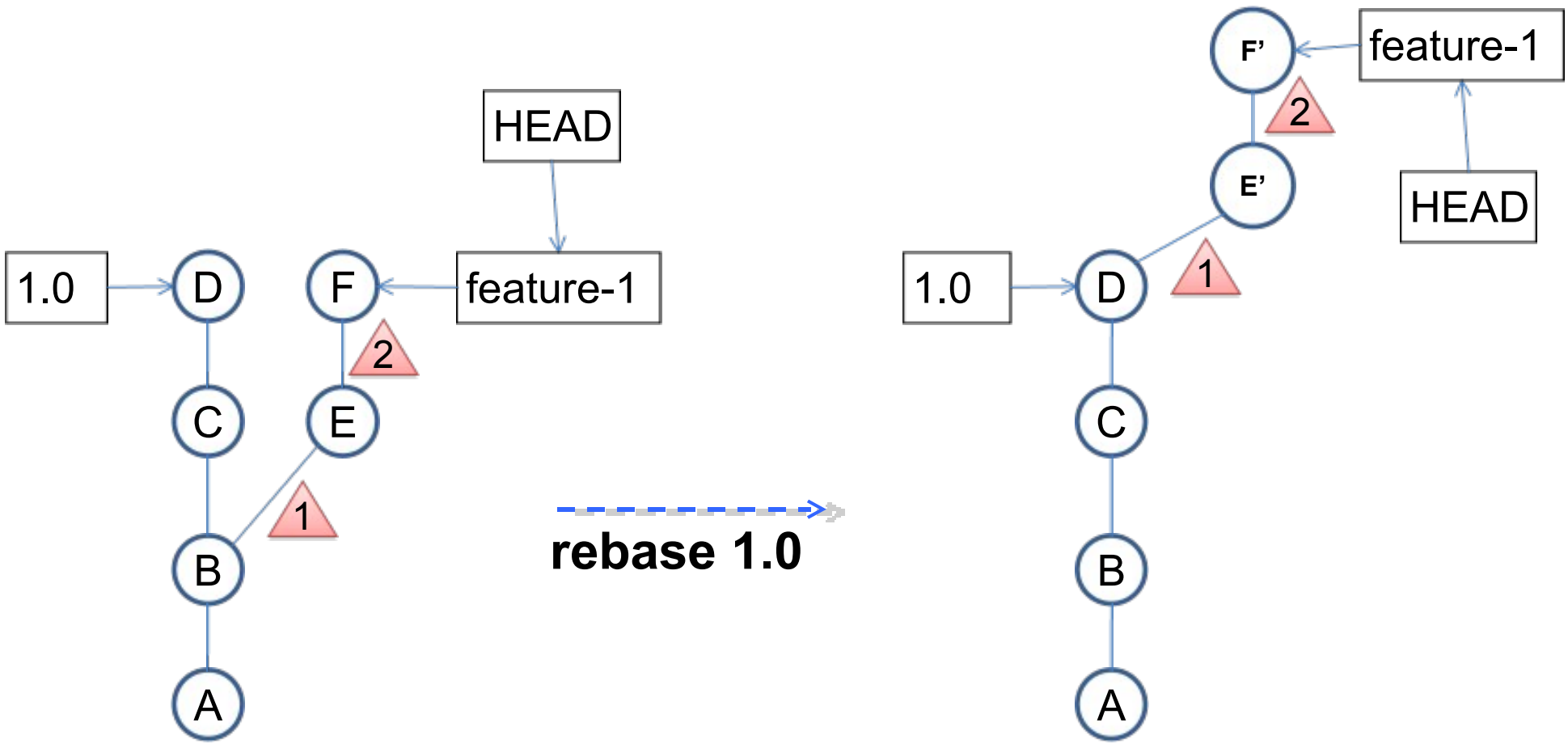
Git Concepts – Cherry Pick

- **git cherry-pick F**
  - **applies changes introduced by F, means delta-2**
  - **no merge relation**



Git Concepts - Rebase

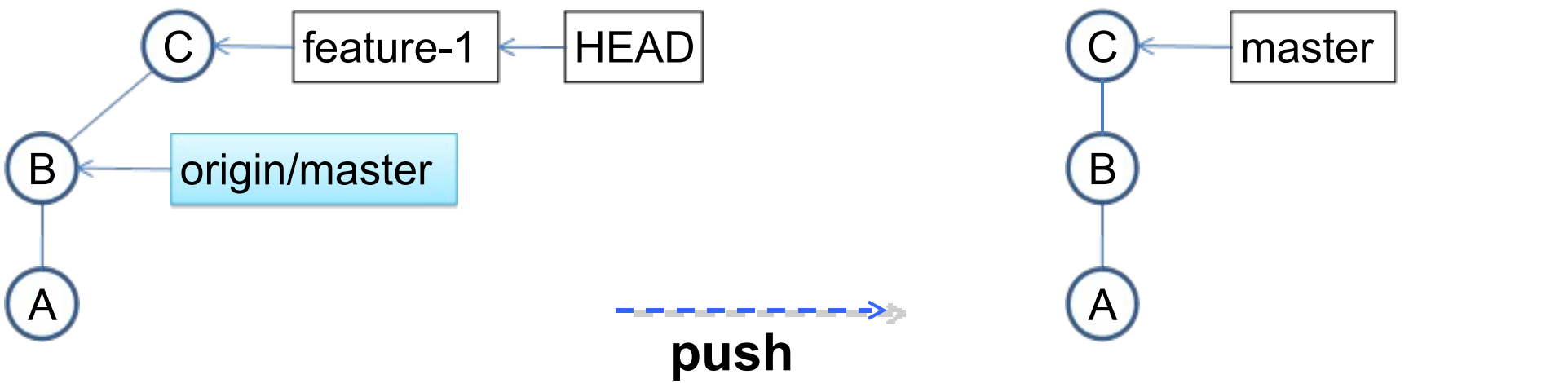
- Alternative to Merge - Keeping history linear
- git rebase 1.0
- after rebase fast-forward possible!



Pushing

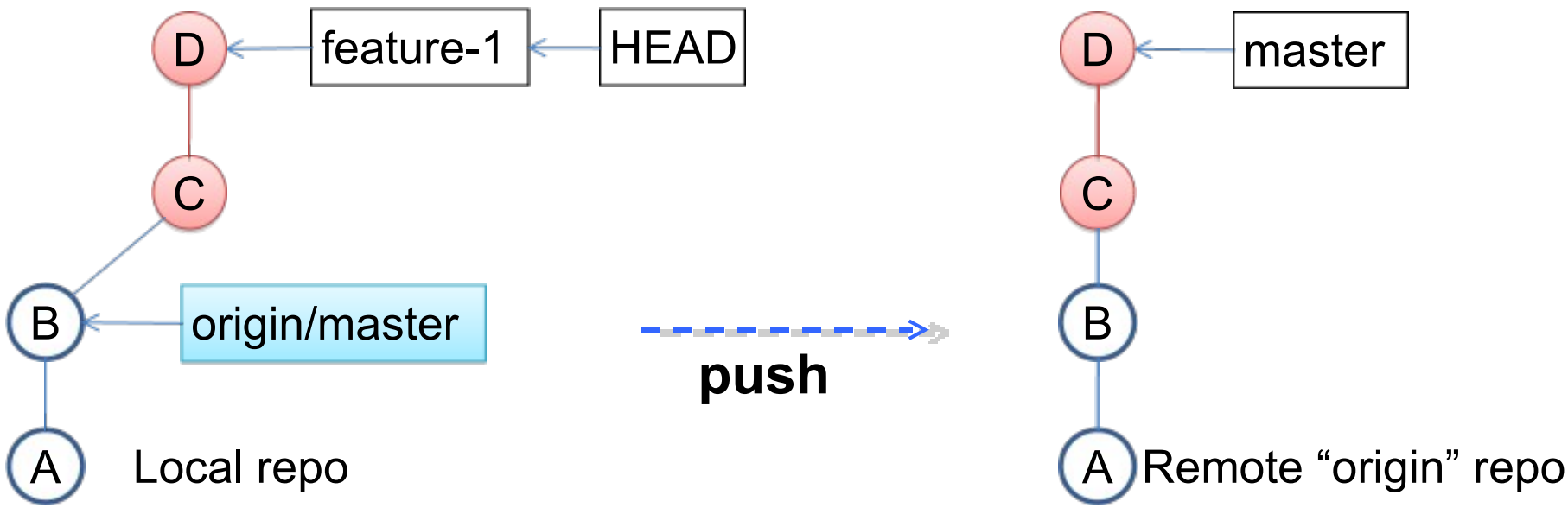
Push

- `git push origin HEAD:master`
- **From local to remote repository**
  - **more precisely: from a local to a remote branch**



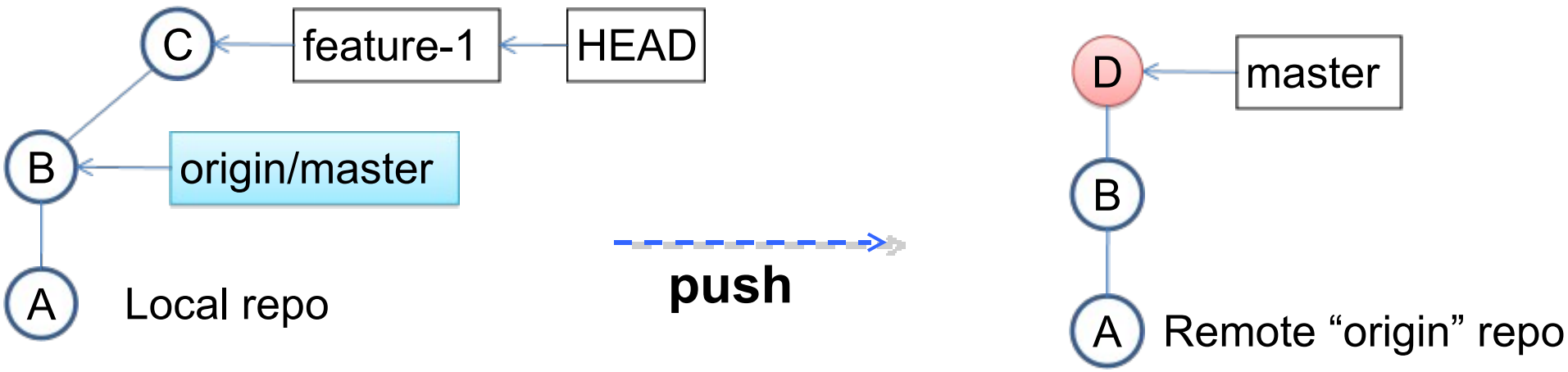
Push

- **Which commits get pushed?**
- **ALL commits from the local branch not available in the remote branch**



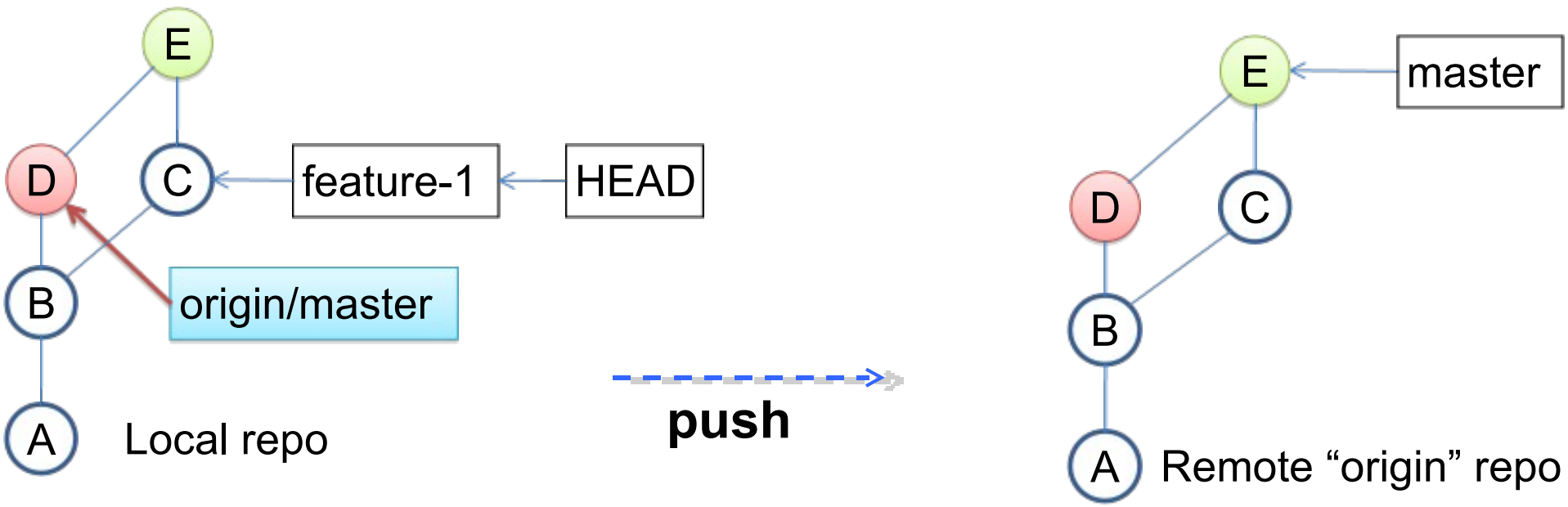
Push

- Remote branch has changed
  - git push will fail because fast forward is not possible



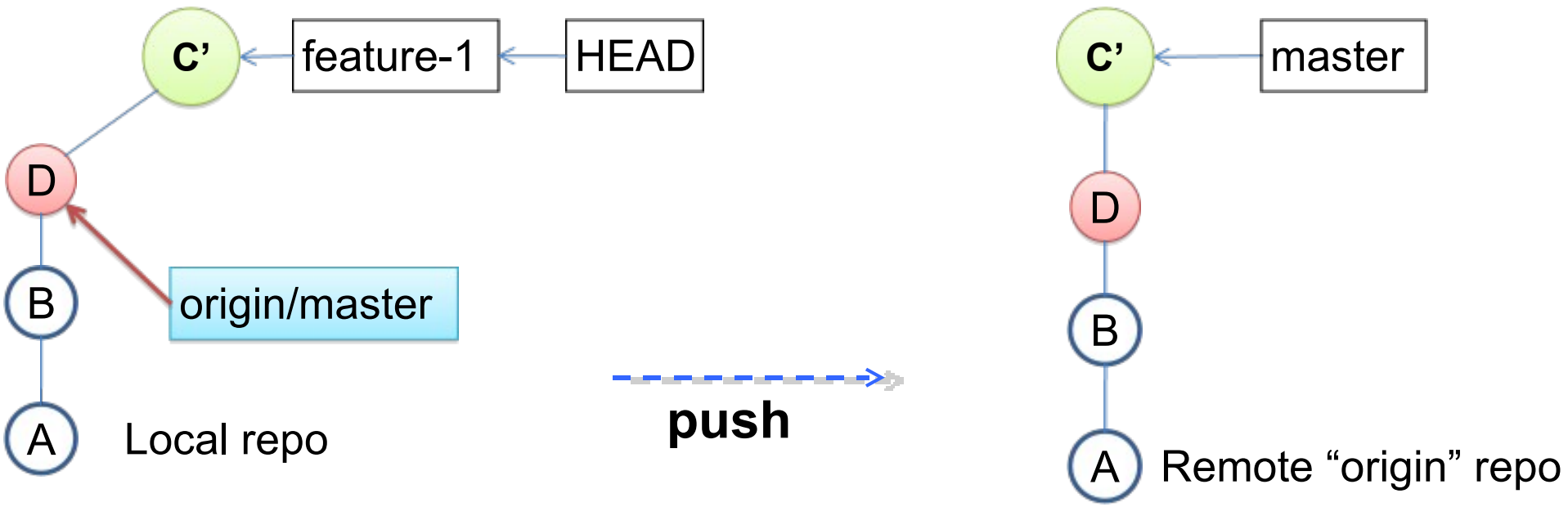
Git Concepts - Push

- Possibility One
  - pull (fetch + merge), push



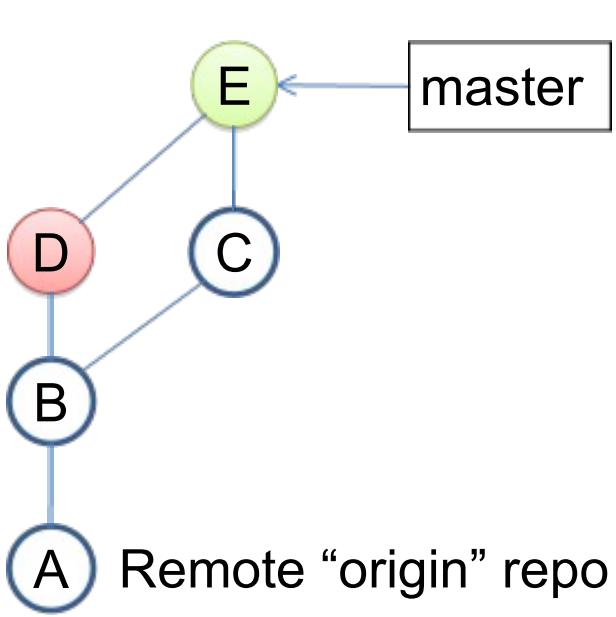
Push

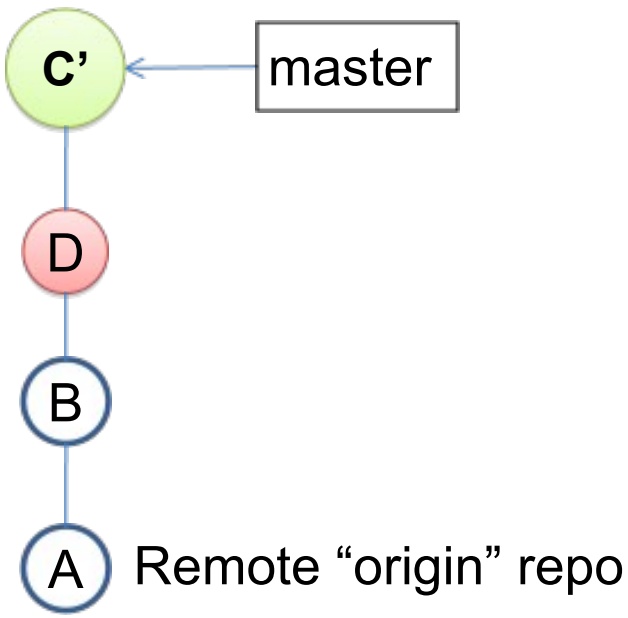
- Possibility Two
  - fetch, rebase, push



Push

- Which graph do you like more ?





Gerrit Concepts

Push

- **Push to Gerrit is the same like push to Git**
  - **with one Gerrit speciality:**
    `refs/for`
    in the target branch name
- **Compare:**
  - **Push to Git:**
    `git push origin HEAD:master`
  - **Push to Gerrit:**
    `git push origin HEAD:refs/for/master`

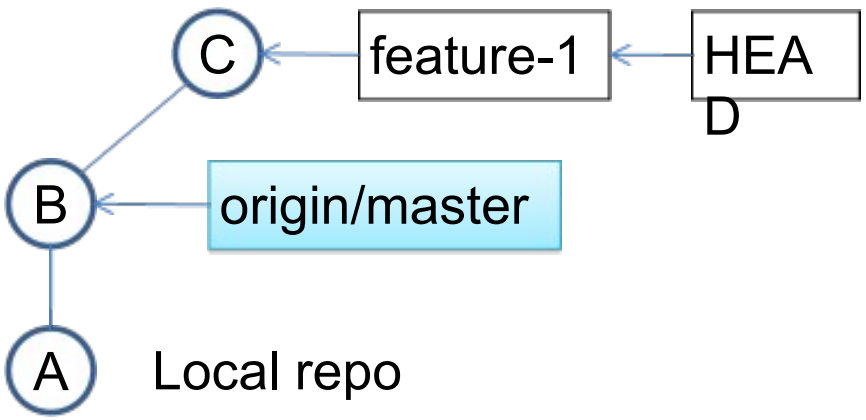
Push

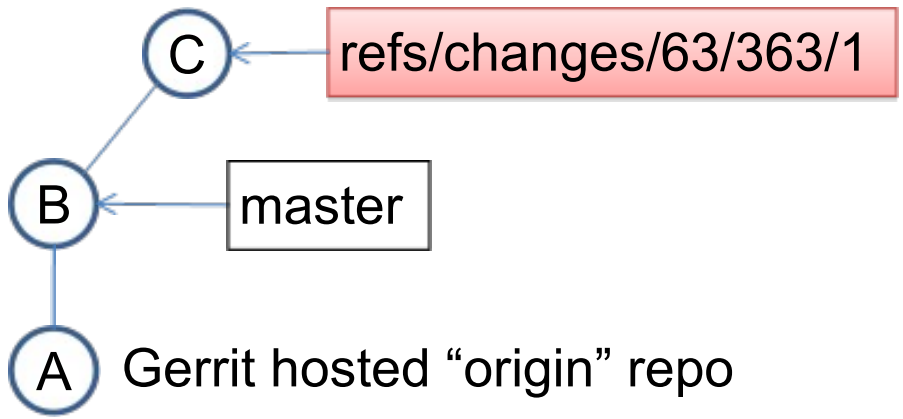
- **It seems like every push to Gerrit goes to the same branch `refs/for/master`**
- **However, Gerrit tricks your git client:**
  - **it creates a new branch for the commit(s) you push**
  - **and creates a new open Gerrit change containing the pushed commit**

Push

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

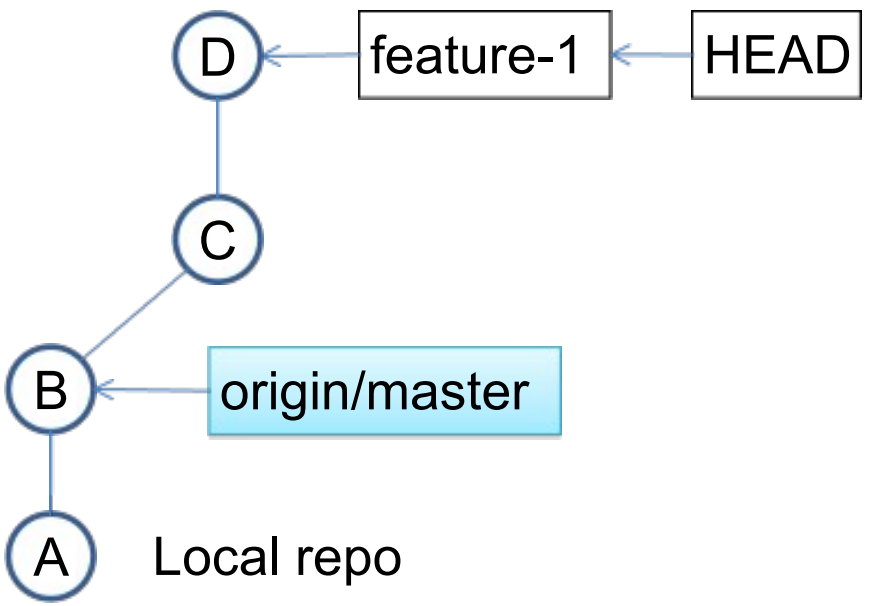
Gerrit DB - Open Changes:
…
{Change-ID = 1234,
Patch-Set-1 = refs/changes/63/363/1
}
…

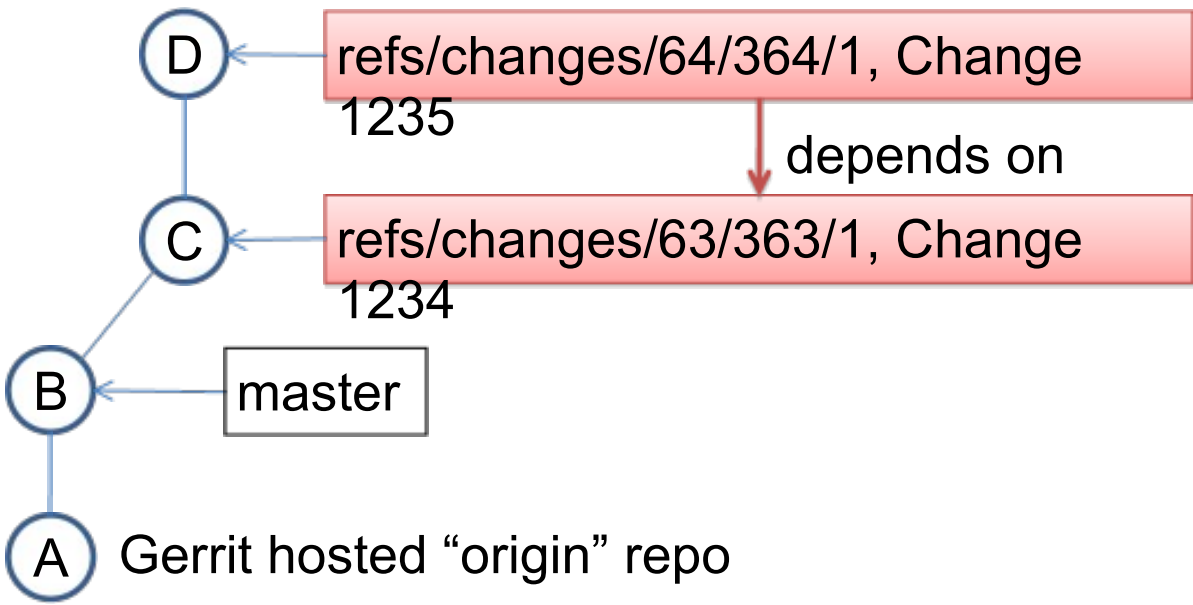




Push

- **What if feature branch has 2 commits ?**
- **Remember Git semantics for Push?**
  - **ALL commits from the local branch not available in the remote branch**
    ➔ 2 changes in Gerrit !





Changes

- Change consists of
  - Change ID (important!)
  - metadata (owner, project, etc..)
  - one or more patch-sets
  - comments
  - votes

- Patch Set represents a Git Commit

New Change vs New Patch Set

- How does Gerrit know whether to create a new Change or a new Patch Set for an existing change?

- It looks at the Commit Message and searches for String “Change-Id: <ISHA1>” in the last paragraph of the commit message
- If found it will create a new patchset for this changes, otherwise it will create a new change

New Change vs New Patch Set

- **Example Commit Message with Change-Id:**

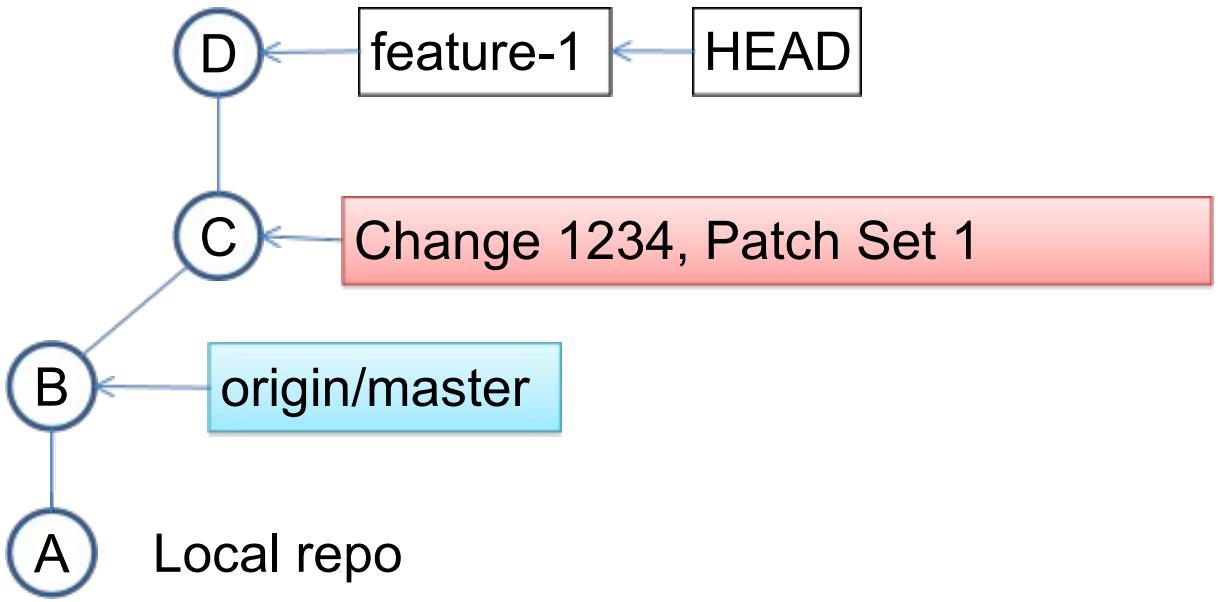
Make lib.Repository abstract and lib.FileRepository its implementation

To support other storage models other than just the local filesystem,
…
will rename it into storage.file.FileRepository, but to do that we
need to also move a number of other related class, which we aren't
quite ready to do.

Change-Id: I1bd54ea0500337799a8e792874c272eb14d555f7
Signed-off-by: Shawn O. Pearce <spearce@spearce.org>

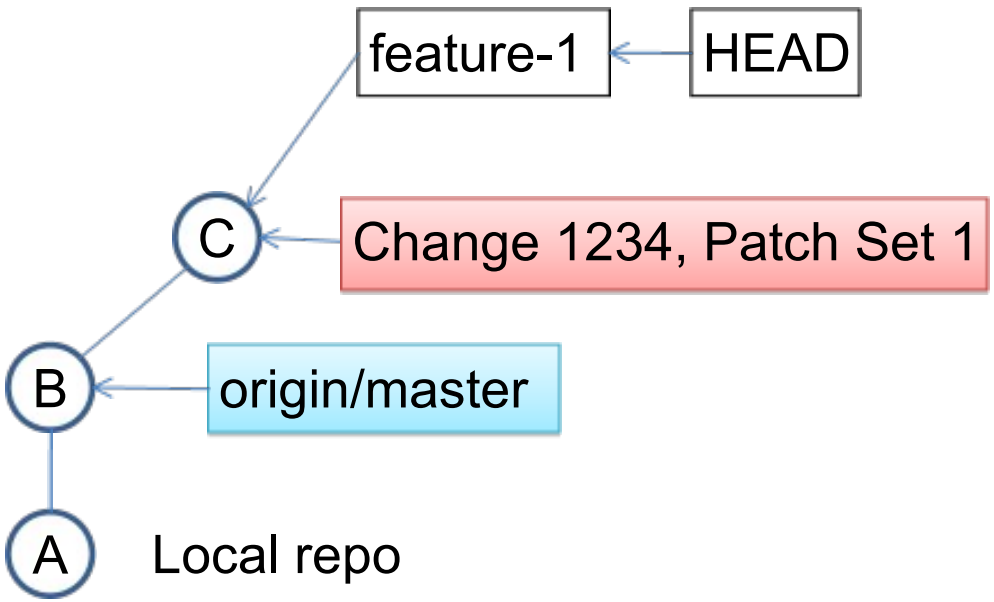
Push New Patchset

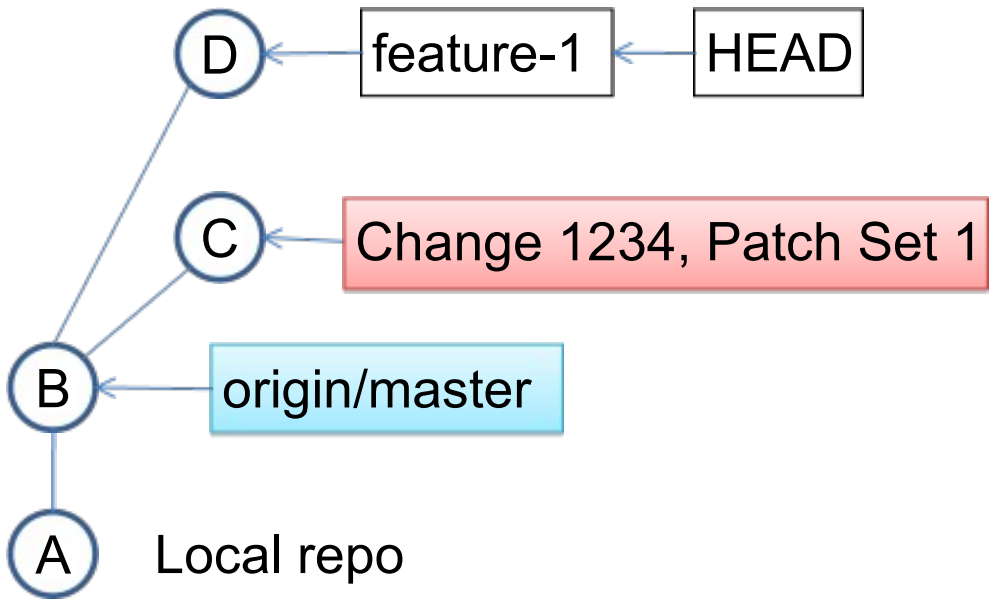
- **No dependencies between Patchsets**
  - **can’t push a successor commit as the next patchset**
  - **commit D can’t be Patch Set 2 of change 1234**



Push New Patchset

- **If you pushed C and need to replace it by a new commit as a new patchset use**
  - **`git commit –amend`**
  - **with –amend option the new commit replaces the current instead of becoming successor**



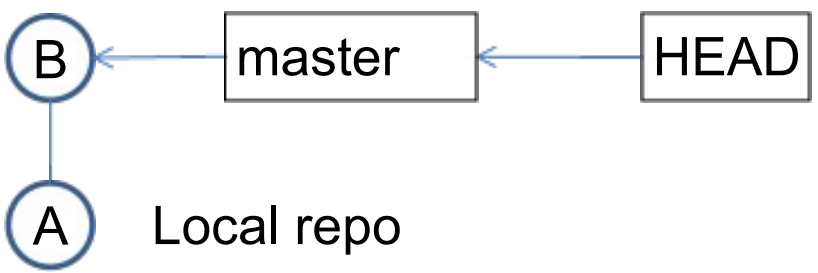


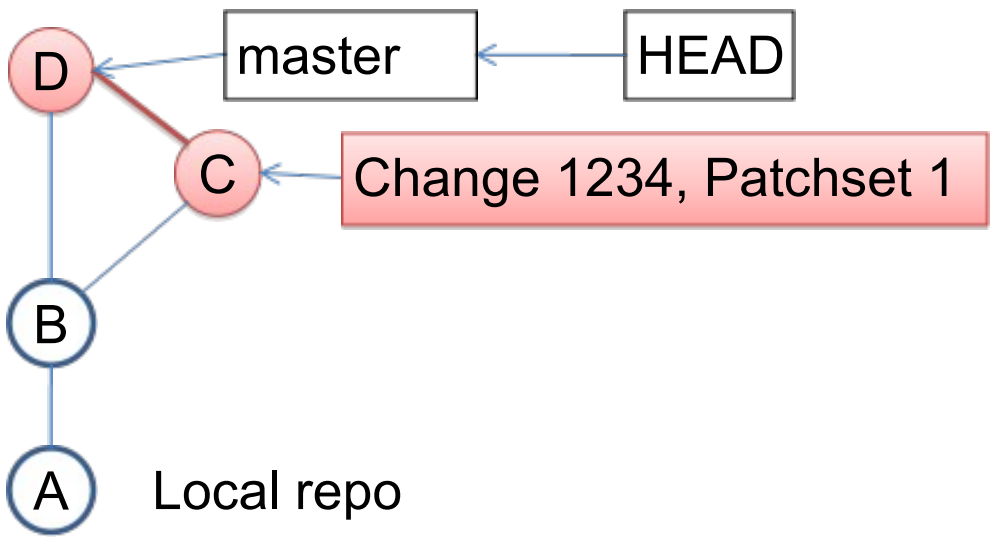
Push New Patch Set

- **A Common Mistake**
  - **author of the Patch Set 1 is not available and somebody else needs to continue and provide Patch Set 2**
  - **use `git pull` to get the Patch Set 1 into a local branch**
  - **Fix issues in commit**
  - **Push (including the same Change-Id)**
  - **Gerrit rejects!**

Push New Patchset

- **A Common Mistake**
  - `git pull origin refs/changes/66/366/1`
  - **D is successor of C and cannot be Patch Set 2**



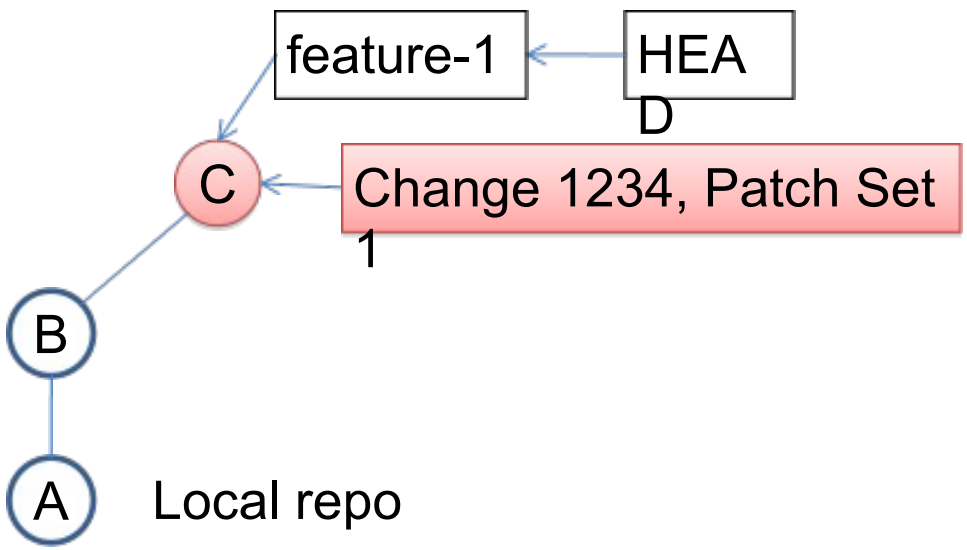


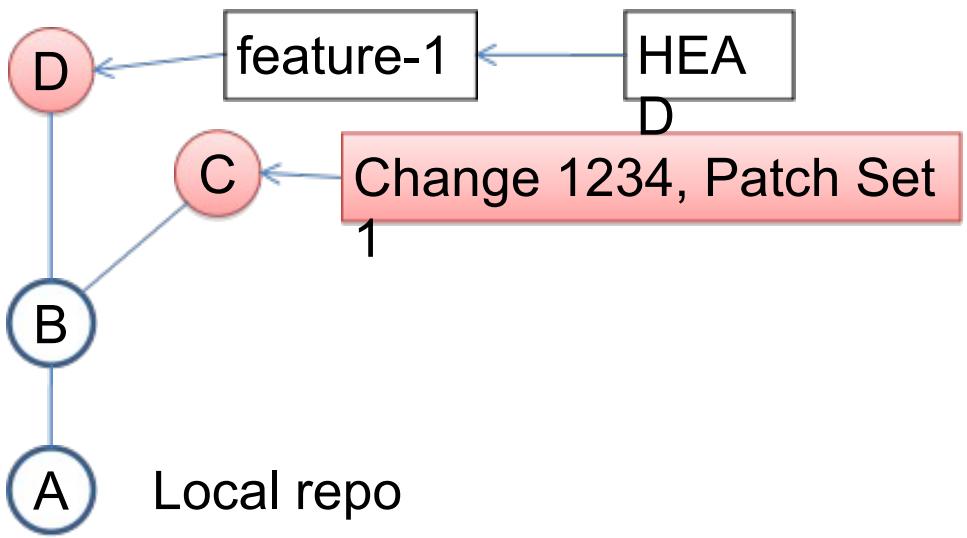
Push New Patch Set

- **The right way:**
  - **`fetch` (don't pull)**
  - **create a new branch based on the fetched patchset 1**
  - **fix the issue**
  - **`commit –amend`**
  - **`push`**

Push New Patchset

- **The right way:**
  - **commit D is not successor of C**
  - **D can become patchset 2**





Review and Vote

- Show in Gerrit Web UI
- Voting the lowest mark means Veto
- Highest marks in all voting categories needed for change to be merged

Fetch Open Change Locally

- Don’t forget that every patchset is a commit
- Use `git fetch` to fetch it locally if you want to play with it
- Hudson Gerrit plugin does just that !
- Gerrit creates fetch command for you
  - show in Web UI

Best Practices

- **Create local branch for each feature or bugfix you work on**
  - **branch name tells you what your intention was**
  - **less likely you will mix two features in the same branch**
  - **you can have many feature branches at a time**
  - **and switch between them**

Best Practices

- **Push finished features only**
  - **who wants to review non finished feature ?**
  - **who wants non-finished features in history ?**
- **Push complete feature as one commit**
  - **even if you created many commits squash them into one before push**
  - **otherwise you create one change in Gerrit per each commit !**
  - **use multiple changes if feature can be split into smaller logical units and use last change to switch on the new feature**

Best Practices

- **Write good commit message**
  - **First line is summary**
  - **Empty line between paragraphs**
  - **Explain WHY you did the change not WHAT you did (this is visible from the commit)**
- **Prefer many small changes to one big**
  - **still each small change must be logically complete**