# GIT

## Distributed Version Control and Source Code Management

Info one

# Agenda

- Version Control Systems
- GIT
  - Basics
  - Branching
  - Advanced use

Info one

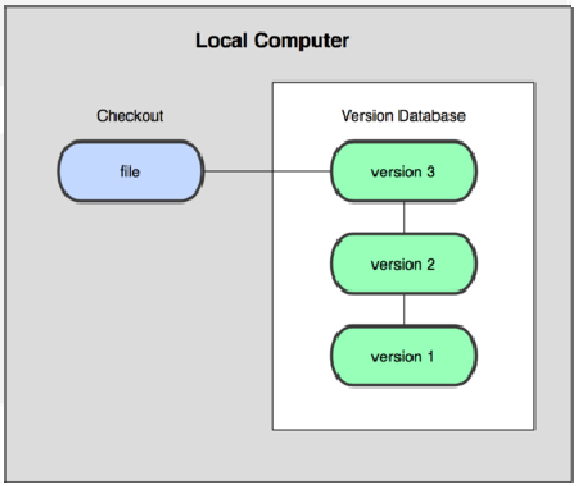
# Version Control

- A system that records changes to a file or set of files over time
- A Version Control System (VCS) allows to:
  - revert files back to a previous state
  - revert the entire project back to a previous state
  - review changes made over time
  - see who last modified a file
  - see who introduced an issue and when

Info one

# Local Version Control

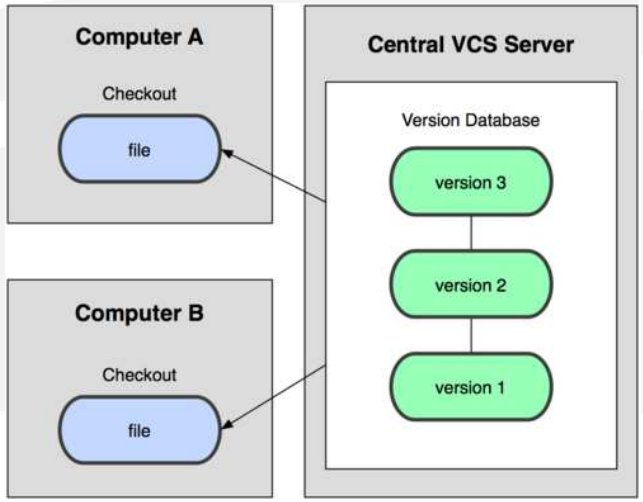
- Many people's version-control method of choice is to copy files into another directory (perhaps a time-stamped directory)
  - Problem: error prone
    - It is easy to write to the wrong file or copy over files you do not mean to
  - Solution: local VCSs with a simple database that keeps all the changes to files under revision control
    - Example: rcs
      - It keeps patch sets (i.e., the differences between files) in a special format on disk; it can then recreate what any file looked like at any point in time by adding up all the patches.




Info one

# Centralized Version Control

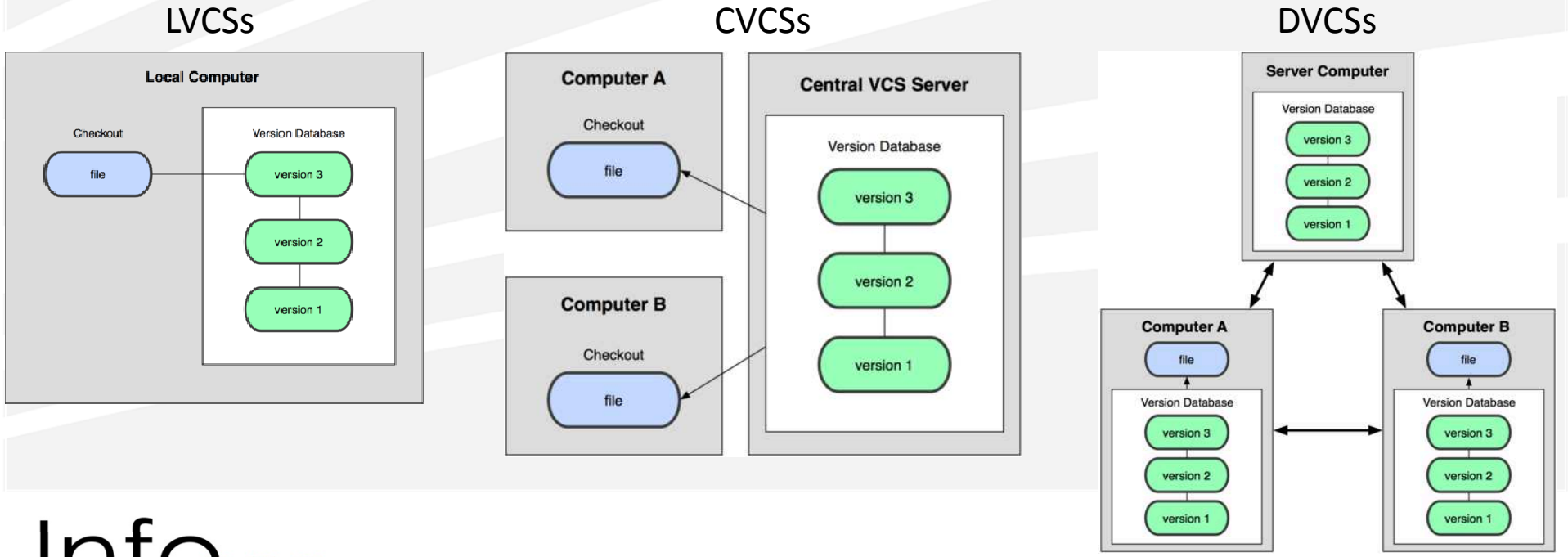
- Problem of local version control: collaboration with other developers
- Solution: deploy of Centralized Version Control Systems (CVCs)
  - Single server that contains all versioned files
  - Access via clients
  - Fine-grained access rights control
  - Examples: CVS, Subversion, Perforce




Info one

# Distributed Version Control

- Problem of CVC: single point of failure
- Solution: distribute the repository to every client
  - Examples: GIT, Mercurial, Bazaar, Darcs



# GIT BASICS

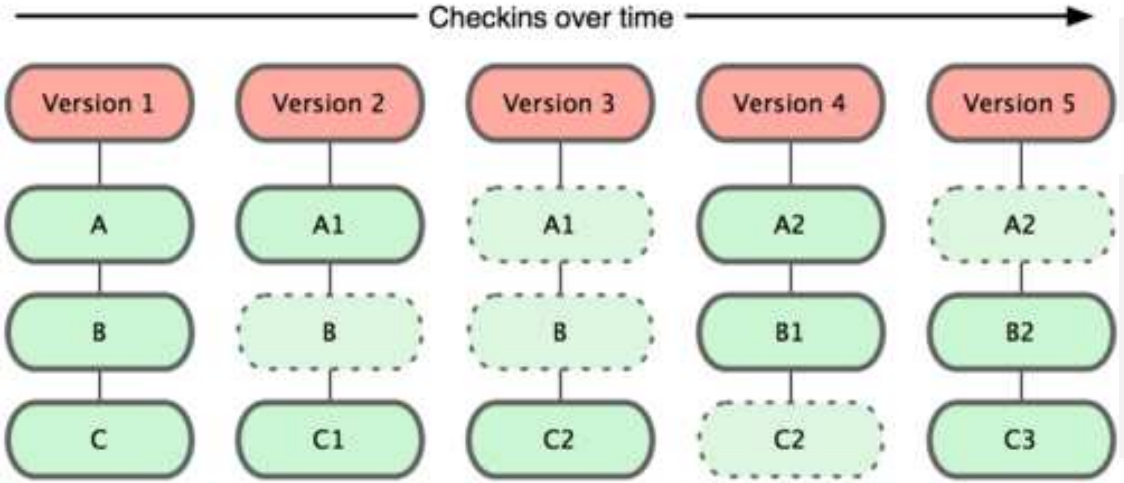
# GIT: a Distributed Version Control System

- History of Linux kernel source change management
  - 1991-2002: changes distributed as patches and archive files
  - 2002-2005: BitKeeper, a DVCS by BitMover
    - Bankrupt of the company
    - Creation of GIT by Linux community (headed by Linus Torvalds)
  - 2005-today: GIT
- Focus: simple design, support for parallel development and performance in terms of speed for big projects

Info one

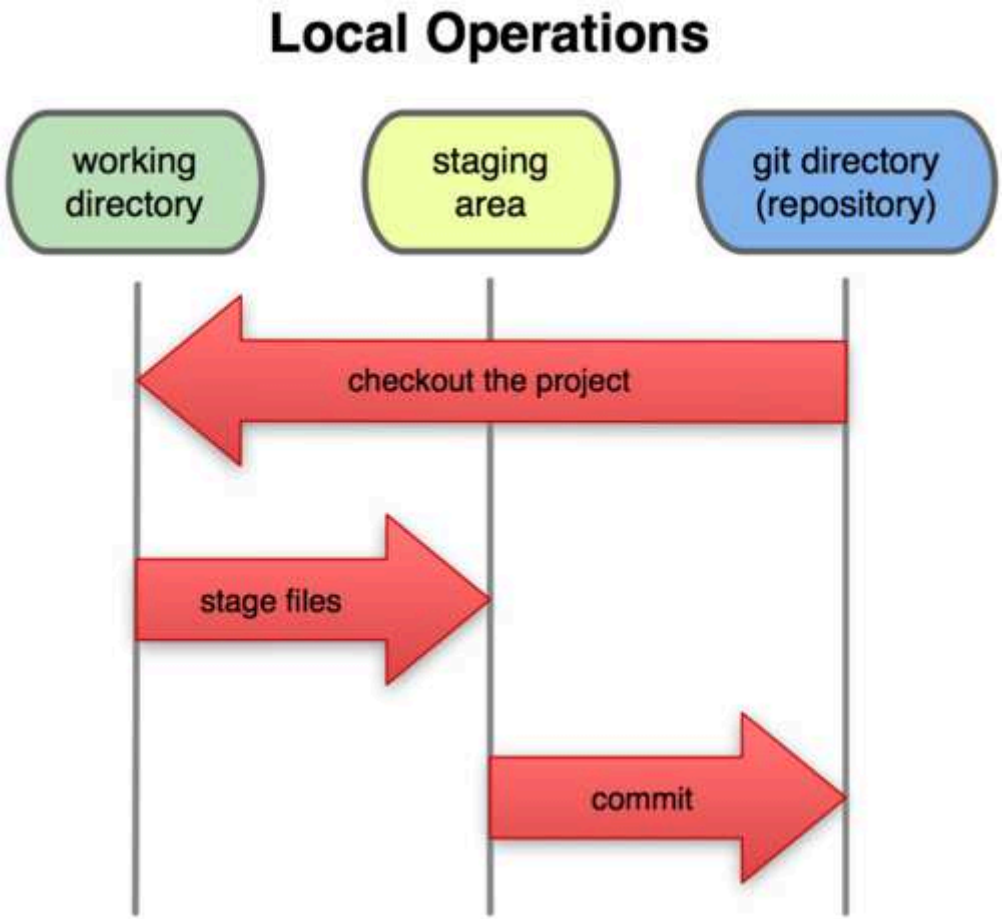
# GIT's Data Management

- Snapshots of the project file system
  - At every commit (i.e. the operation you make at a given time to create a "restore point"), GIT takes a picture of all project files and stores a reference to it
    - If files have not changed, they are not stored again
    - Every file is check-summed (SHA-1 hash)



# GIT Data States

- States:
  - Committed: Data are stored in the local database
  - Modified: Data are changed, but not stored in the local database
  - Staged: Data are marked to go in the next commit
- Directories
  - Git Directory: directory for storing metadata and object database (this is copied when a repository is cloned)
  - Working Directory: one checked-out version of the project
  - Staging area: file containing staged data information (index)




Info one

# Installation and First Configuration

- Download and run the installer
- Set username and email (commits will use them)
  - *git config --global user.name <your username>*
  - *git config --global <your email>*
- Set default editor, diff tool
  - *git config --global core.editor <your editor>*
  - *git config --global merge.tool <your diff tool>*
- Check configuration or get help
  - *git config { --list | <key> }*
  - *git help config*
- *Create command aliases*
  - *git config –global alias.<alias> <command>*

Info one

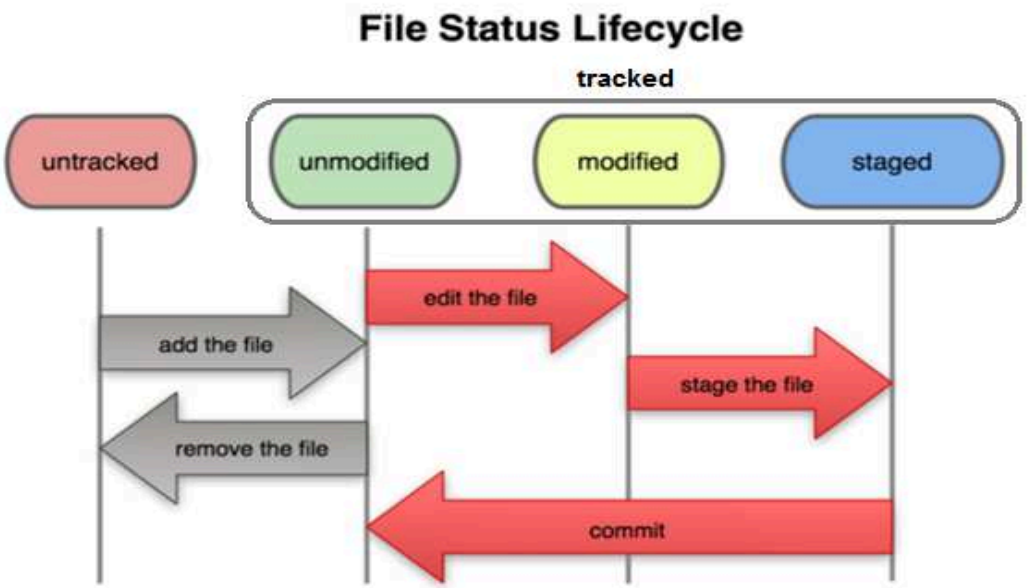
# Create or Clone a Repository

- Create a new repository
  - Point at the directory
  - Initialize the directory: *git init*
    - It will create a subdirectory named .git containing all repository files
- Clone a repository
  - Download all files required to have a local
    - git clone <url>
      - *{ http | https | git }://<domain>/<project>/<repository name>.git*
        - » SSH or local protocols can be used
    - It will create and initialize a .git directory inside the project folder named <repository name>
    - Project files are inside the folder <repository name>

Info one

# File Status Lifecycle

- Files in the working directory of the repository can be in two states
  - Tracked
  - Untracked
- Tracked files
  - Were in the last snapshot
  - Can be
    - Unmodified
    - Modified
    - Staged
- Untracked files
  - Were not in the last snapshot nor staging
- Right after cloning, all files are tracked and unmodified
- Check the status of files (list untracked, modified and staged files)
  - *git status*



# Staging Files

- *add <file>* stages a file (i.e. plan file for next commit)
- Notice: adding a staged file means that in the next commit it will be added as it was at the moment you added it
  - If a staged file is modified, the committed file will not incorporate such changes
    - After modifying, the git status command will show the file both as staged and unstaged (original and modified version, respectively)
    - In order to commit the modified version, the file has to be added again
    - *git diff* shows changes not yet staged (but not all changes from last commit)
      - *git diff --cached* or *git diff --staged* shows what is staged and is going to be committed
  - *git reset HEAD* unstages staged files

Info one

# Ignoring Files

- The *.gitignore* file contains a list of files and folders that should be not committed
  - E.g. automatically generated log files, temporary files, object or binary files
- Files and folders are specified via rules
  - Glob patterns
    - Simplified regular expression
      - * for zero or more character matching
      - [<characters>] for any character inside the brackets
      - ? For any single character
      - [<char>-<char> for any character in the interval
      - **/ for any directory
  - / at the end indicates a directory
  - ! at the beginning indicates a negation
  - # for comments

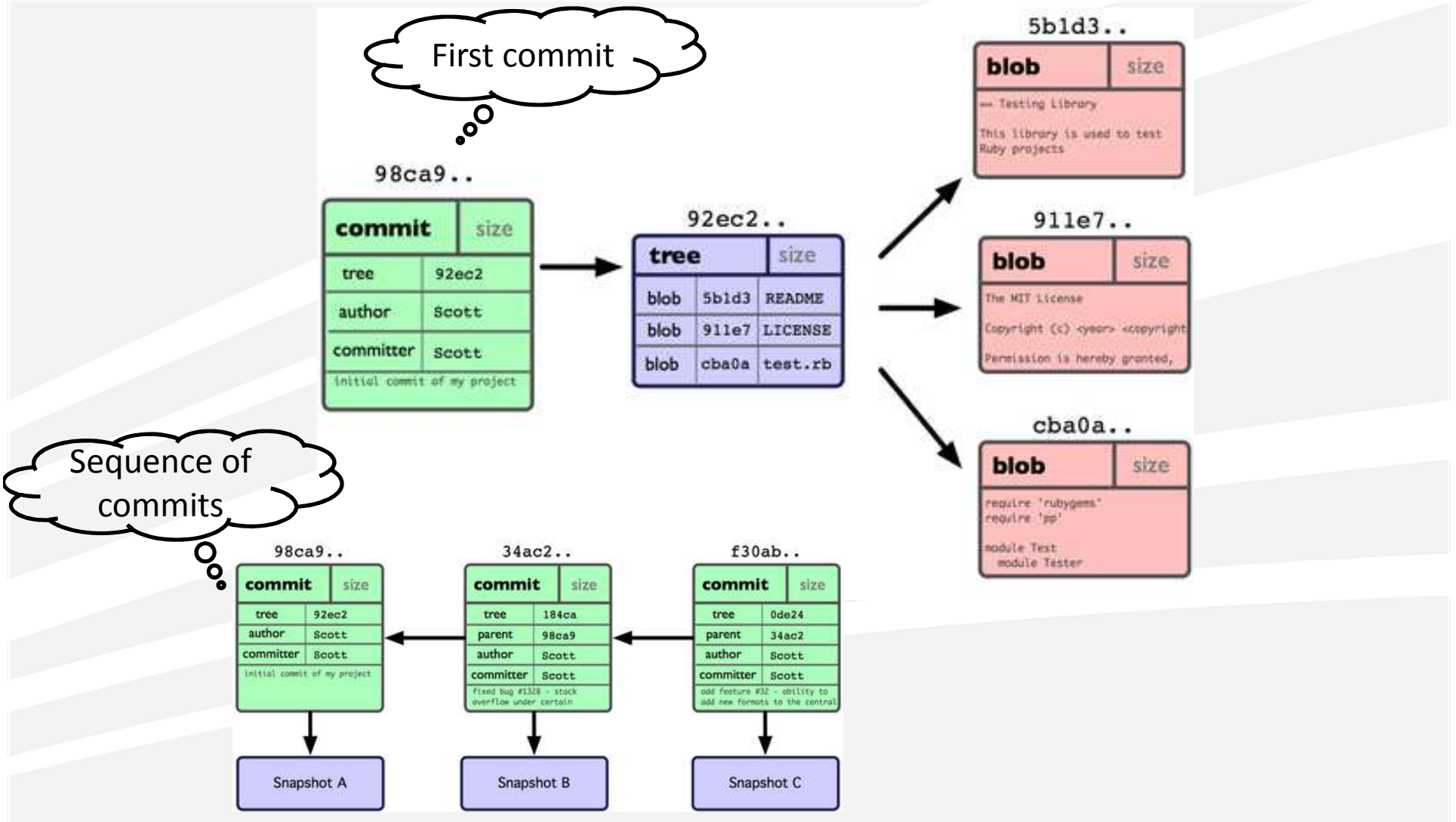
```
# a comment - this is ignored
# no .a files
*.a
# but do track lib.a
!lib.a
# only ignore the root TODO file, not subdir/TODO
/TODO
# ignore all files in the build/ directory
build/
# ignore doc/notes.txt, but not doc/server/arch.txt
doc/*.txt
# ignore all .txt files in the doc/ directory
doc/**/*.txt
```

# Committing Files

- *git commit*
  - Staged files are committed, but unstaged files stay on the hard disk (not in the repository)
  - Runs the editor and opens a file containing the output of *git status*
    - Adding *-v* to the command will add the output of *git diff*
    - Comments can be added in the text file or inline
      - *git commit -m "<comment>"*
  - After committing, the impacted branch and its checksum are shown
  - The committed records constitute a snapshot that can be reverted or compared to other snapshots
- *git commit --amend* merges into the last commit the changes happened after that commit (e.g. for a forgotten file)
- *git log* lists the commits made in that repository in reverse chronological order and has a number of options for different formats and information

Info one

# An Example of Commit Results



# Removing, Renaming and Reverting Files

- To remove a file it has to be untracked
  - If the file is removed from the working directory, it becomes unstaged
  - *git rm* stages the removal
    - Next commit will produce a snapshot without the removed files
      - If you previously modified and added a file to remove, use *--f* to force removal
      - If you want to remove a file from the staged area without removing it from the working directory, use *--cached*
- Renaming a file is not an explicit command
  - *git mv <file_source> <file_destination>*
    - It adds <file_destination> and removes <file_source>
- *Reverting a file to the last committed version*
  - *git checkout <file>*

Info one

# Remote Repositories

- Remotely stored versions of the project
  - *git remote* shows all remote repository names
    - *-v* adds URLs
    - *git remote show <name>* presents additional information
  - *git remote add <name> <url>* creates a new remote repository
  - *git fetch <name>* pulls all data not yet pulled
    - *git fetch origin* pulls any new work that has been pushed to the server from which the repository was cloned
    - No merging is performed
  - *git push <name> <branch>* pushes the project to the remote repository
    - *It works only when writing is allowed*
    - *It works only when nobody else pushed after last local pull*
  - *git remote rename <original name> <new name>* renames the repository
  - *git remote rm <name>* remove the repository

Info one

# Tagging

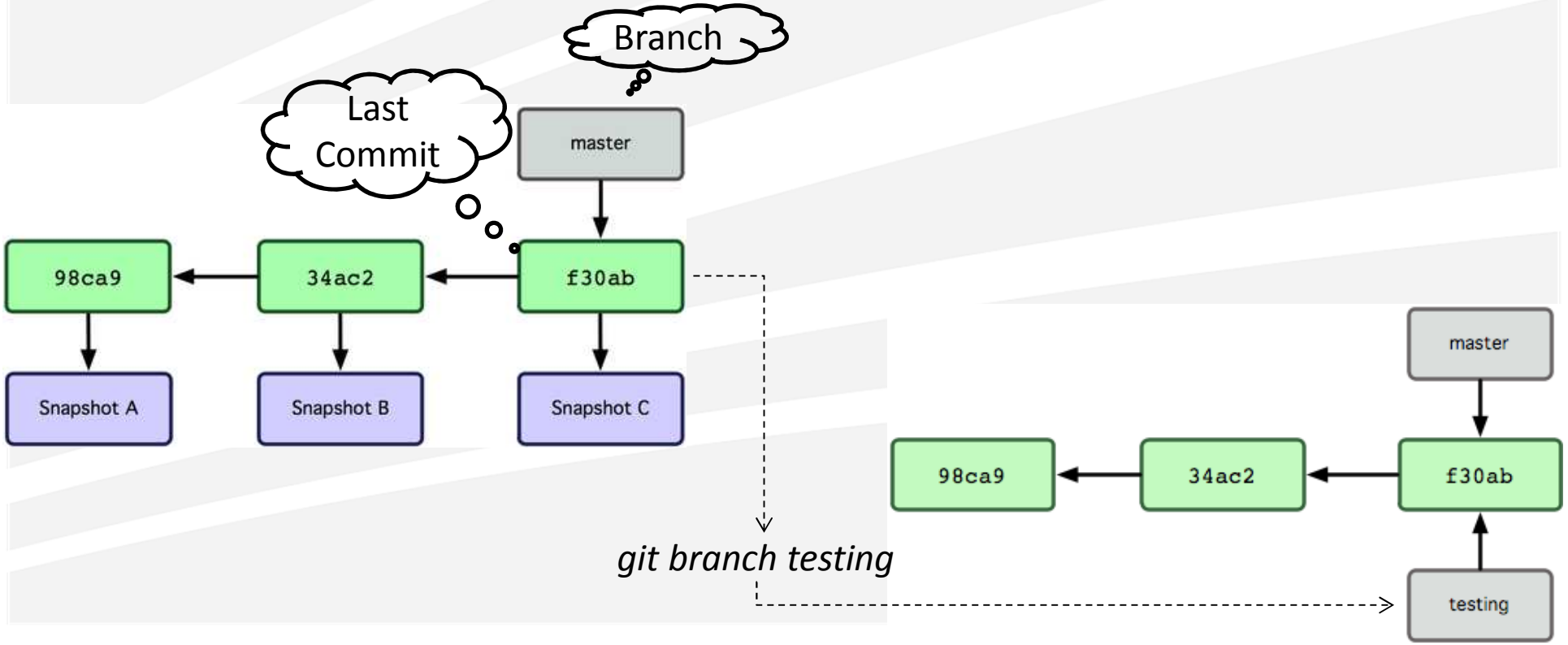
- Tags are labels to associate to commits, e.g. to mark release points
- *git tag* shows all available tags alphabetically
  - *git tag –l <pattern>* shows only tags matching the given pattern
- *git tag -a <tagname> -m '<message>'* creates a new tag named <tagname> and stores it in the git database, together with tagger name, email address, date and a message
  - It also runs the editor
  - Tag data are shown along with tagged commit info when running *git show*
  - Tags can be signed by replacing *-a* with *-s*
    - *git tag –v <tagname> verifies the signature*
  - Adding a checksum option at the end of the command tags the corresponding commit
  - Omitting all options (*-a*, *-s* and *-m*) leads to creating a lightweight tag (no tag data)
- Tags have to be pushed one at a time (*git push <repository> <tagname>*) or all at once (*git push <repository> --tags*)

Info one

# BRANCHING

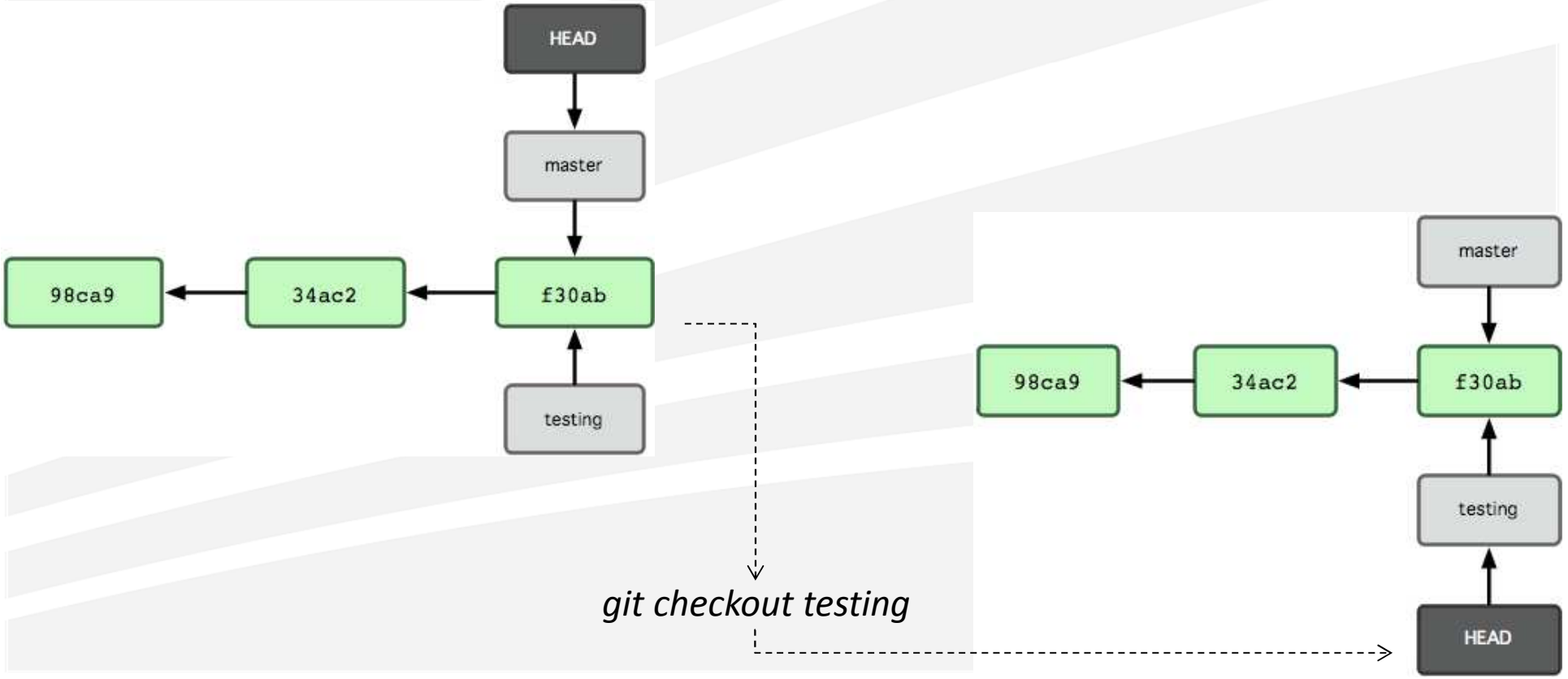
# Branching: Create a Branch

- Creating a deviation in the main line of development
  - *git branch <branch name>*
- A branch is a pointer to a commit

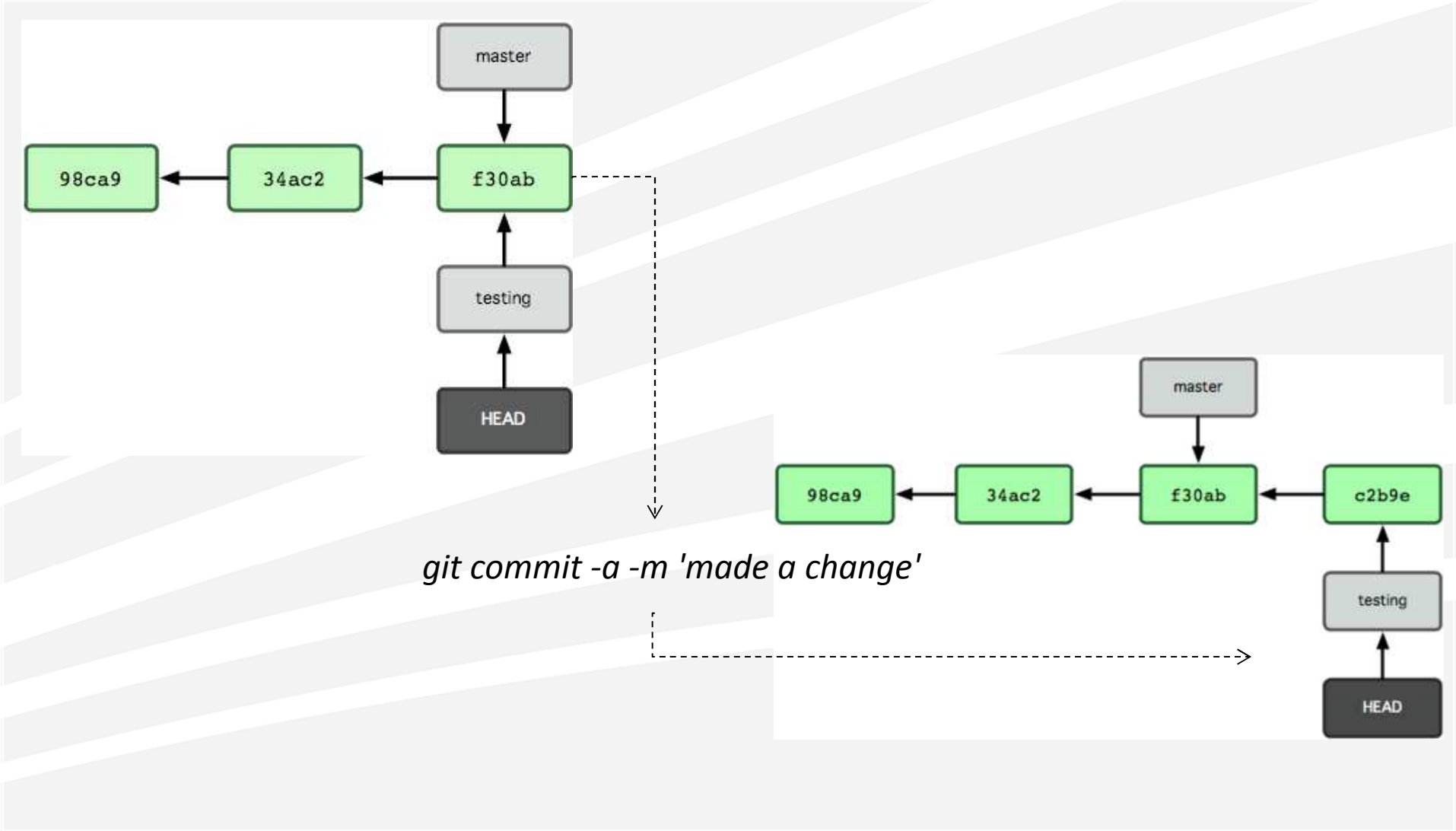


# Branching: Switch to a Branch

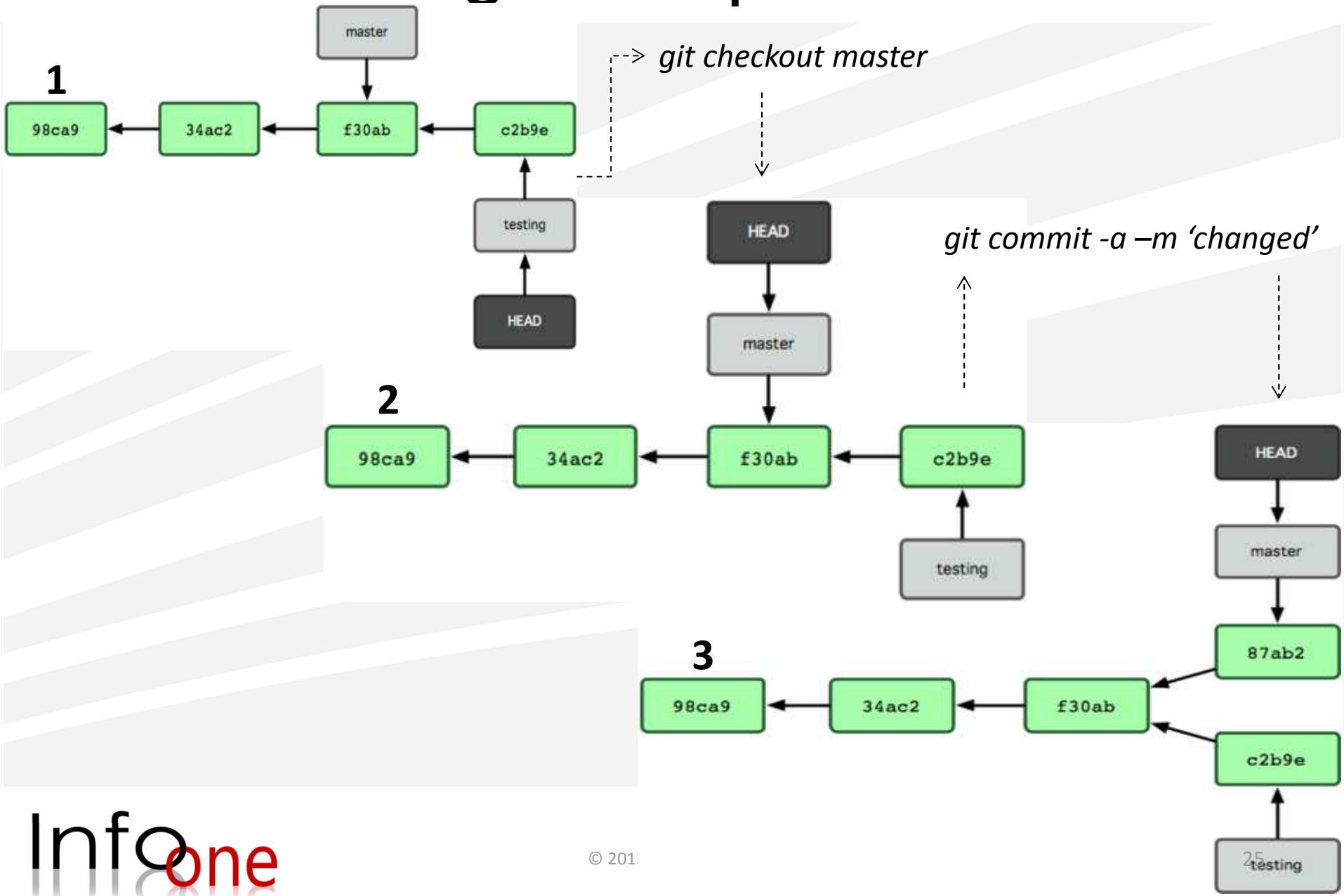
- HEAD is a pointer to the current branch



# Branching: Impact of a Commit

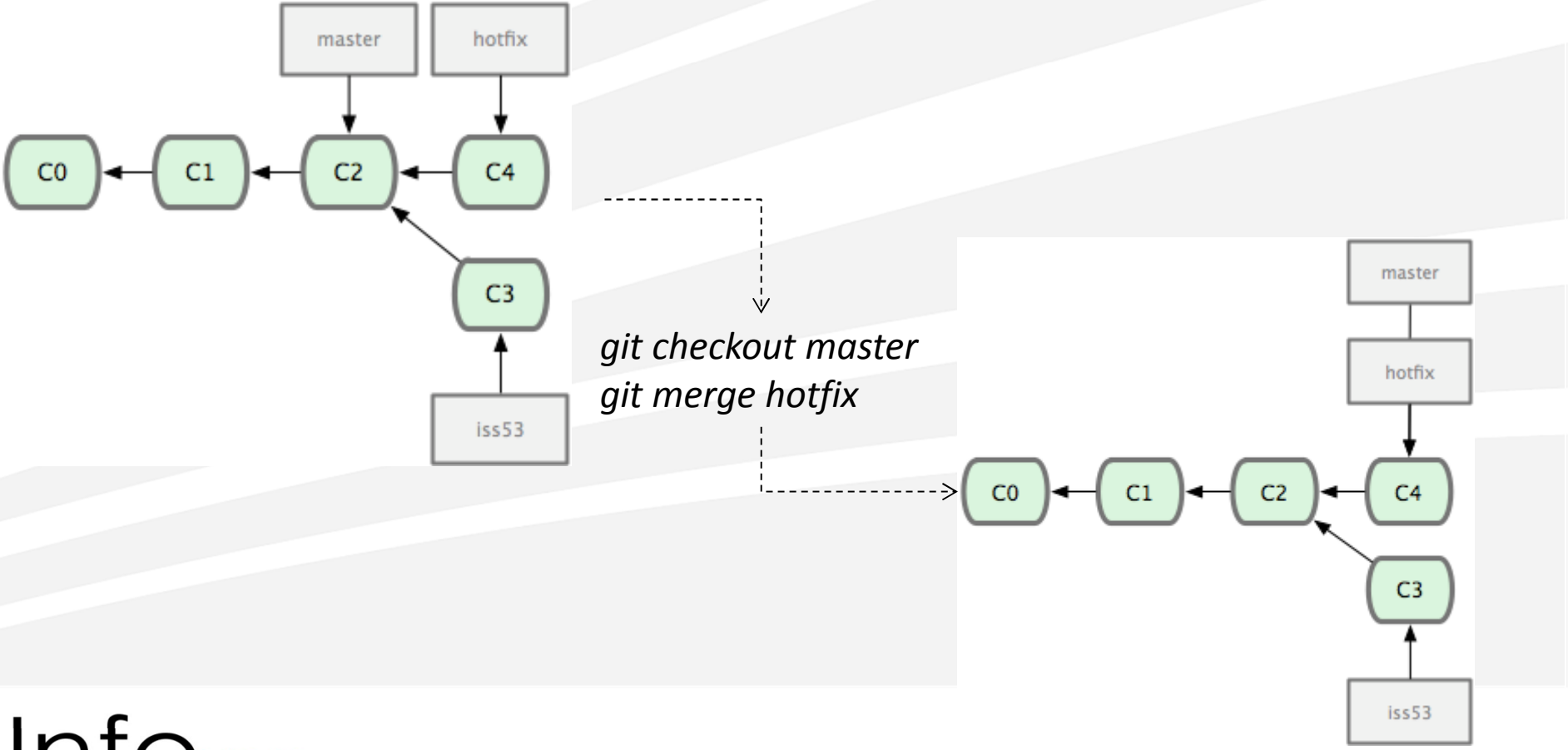


# Branching: Multiple Branches

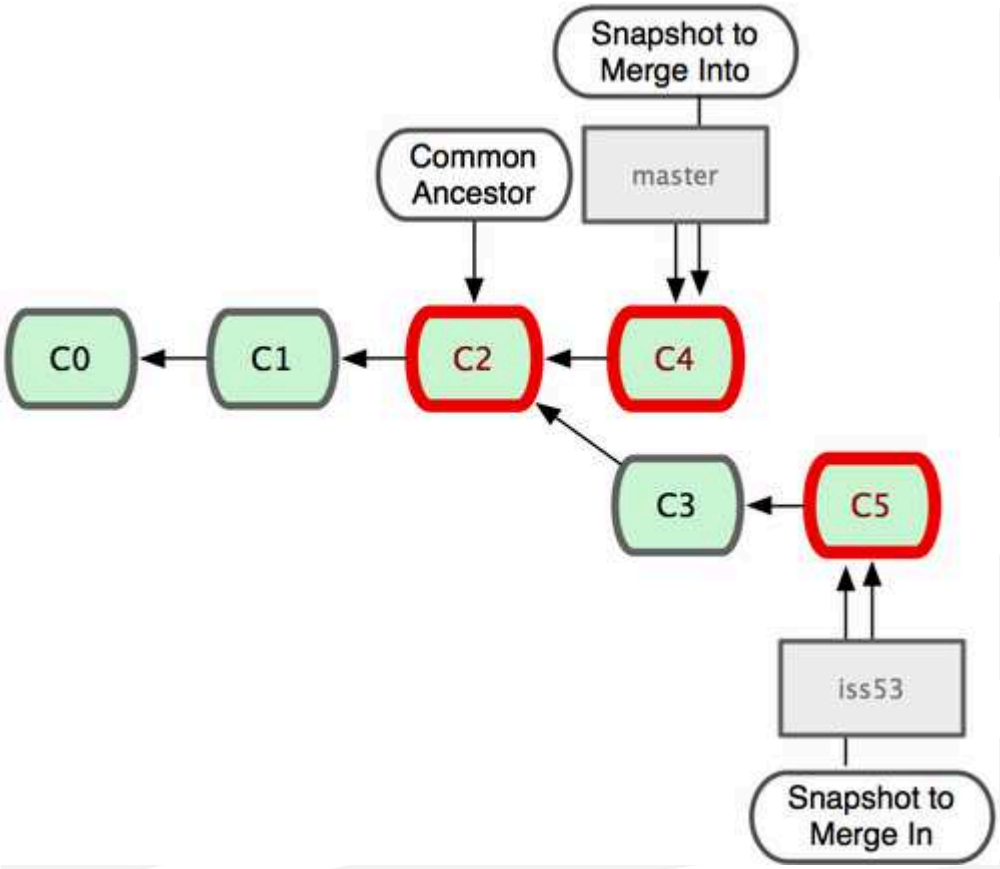


# Merging Branches: Fast-Forward

- Merge a commit with another commit that can be reached by following the first commit's history

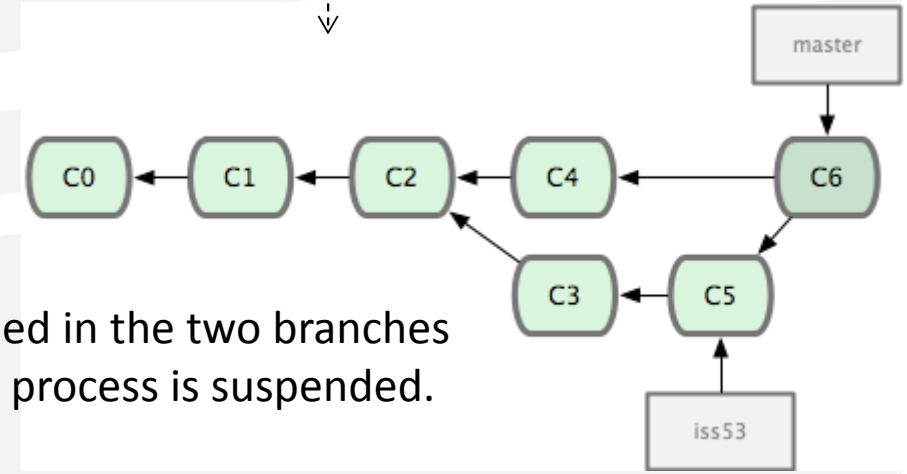


# Merging Branches: 3-Way Merge




Notice: Hotfix branch is not contained in the files in the iss53 branch

*git checkout master*
*git merge iss53*



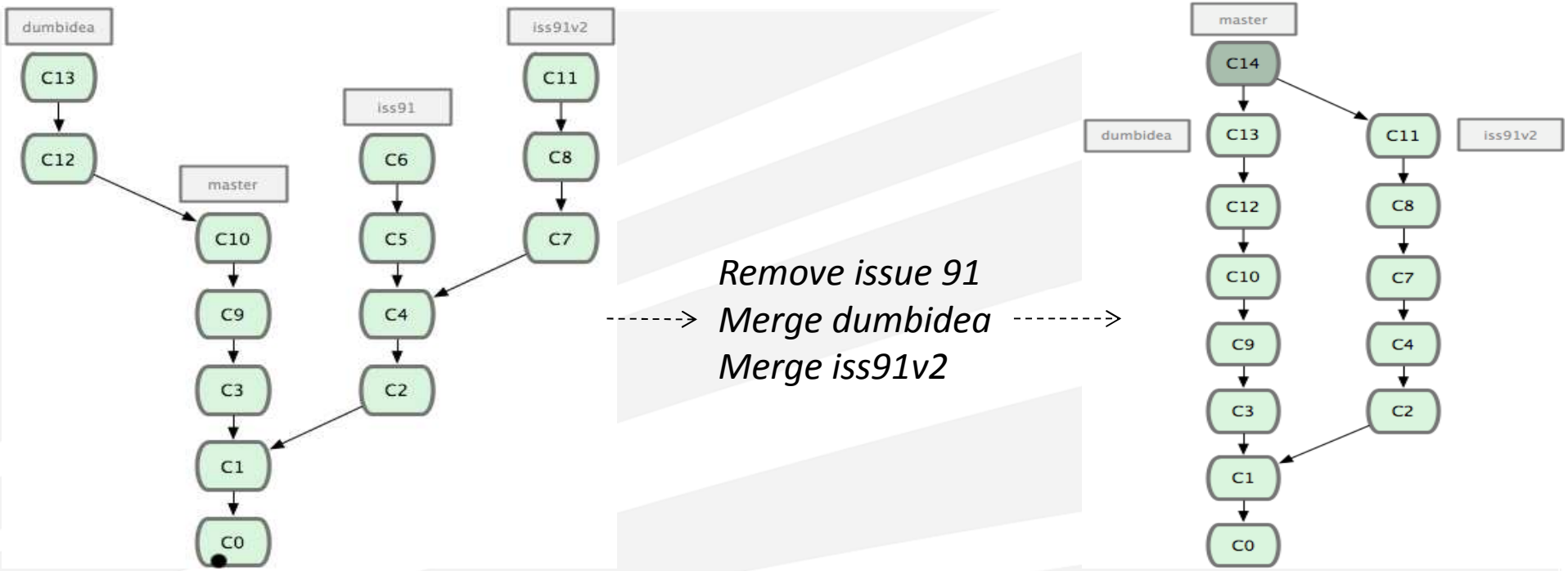

Notice: If the same part of the same file was modified in the two branches to merge, manual merge is required and the merge process is suspended. Command *git status* shows unmerged files.

# Branch Management

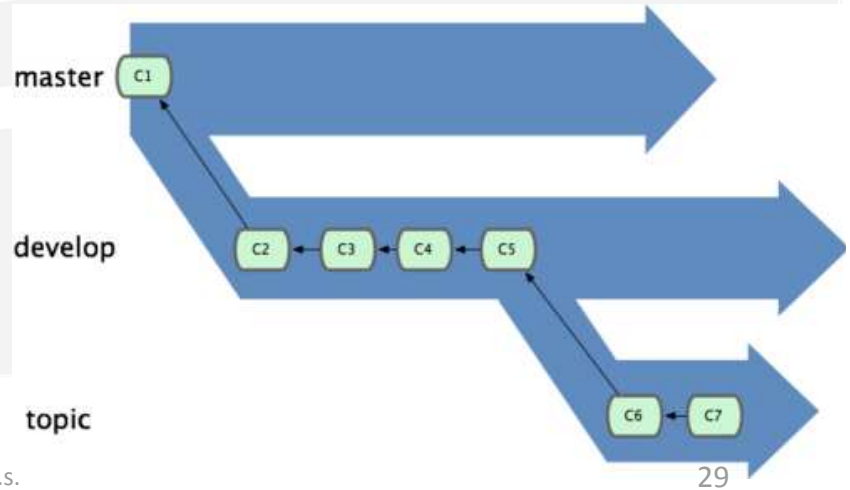
- *git branch* shows existing branches
- *git branch -v* shows last commit on all existing branches
- *git branch --merged* shows all branches merged into the current branch
- *git branch --no-merged* shows all branches not merged into the current branch
- *git branch -d <branch-name>* deletes <branch-name>
  - It succeeds if everything has been merged into another branch
  - In order to force removal, use -D in place of -d

Info one

# Examples of Branching

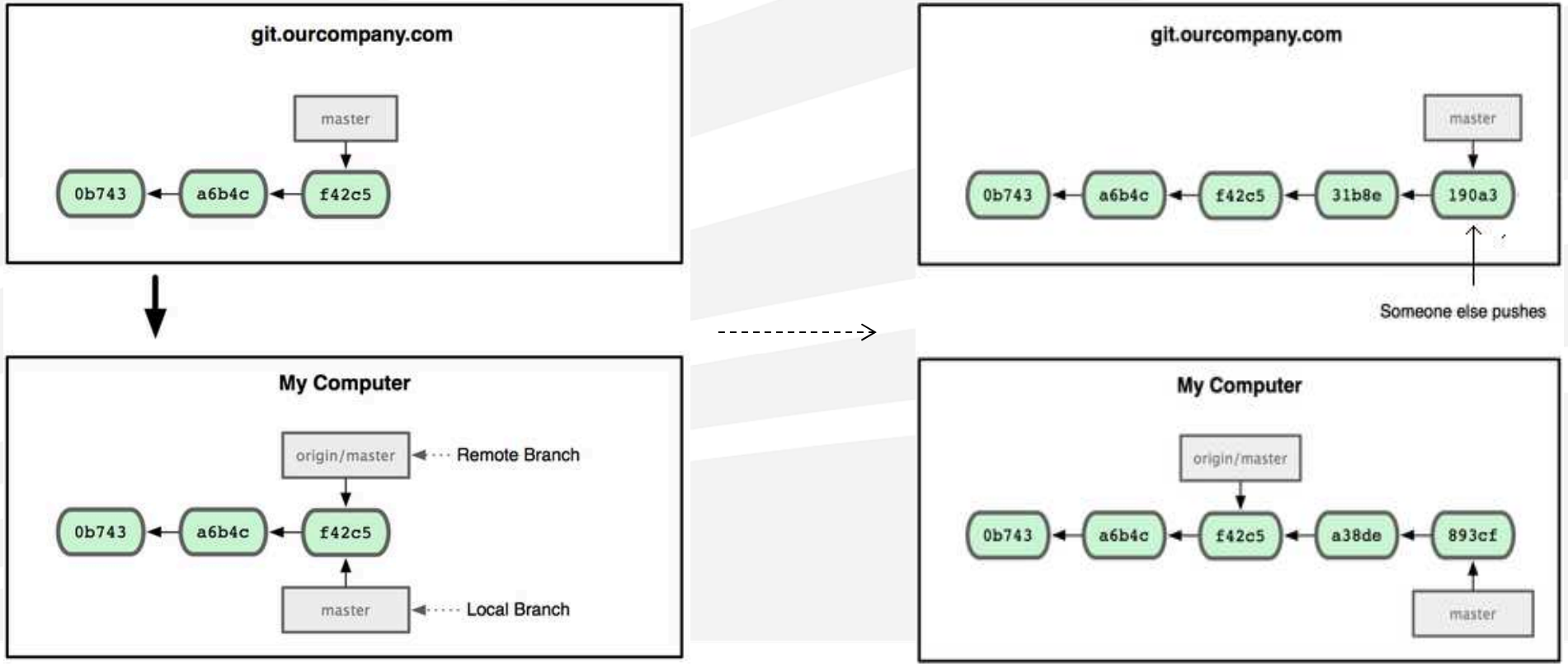



Gradually more stable code bottom-up, e.g.:
Master: stable code
Develop/Next: not necessarily stable, but under test
Topic: currently working on, short-life branches



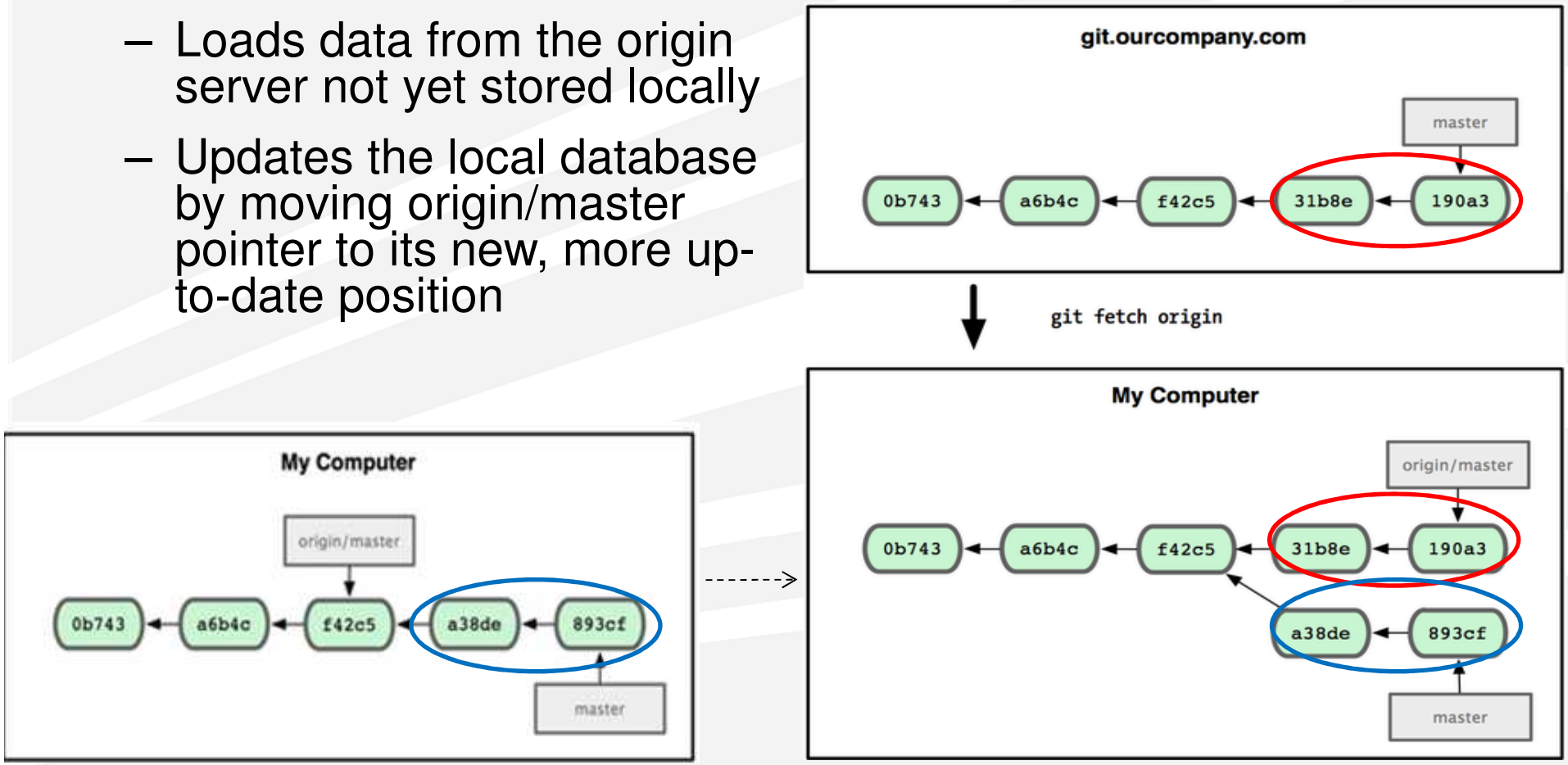
# Remote Branches

- References to the state of branches on the remote repository
  - E.g. clone the master branch from ”ourcompany” server
    - When doing some work on local branch while someone else is pushing to git.ourcompany.com and updates its master branch, then histories move forward differently



# Remote Branches: Synchronizing (1)

- *git fetch origin*
  - Loads data from the origin server not yet stored locally
  - Updates the local database by moving origin/master pointer to its new, more up-to-date position



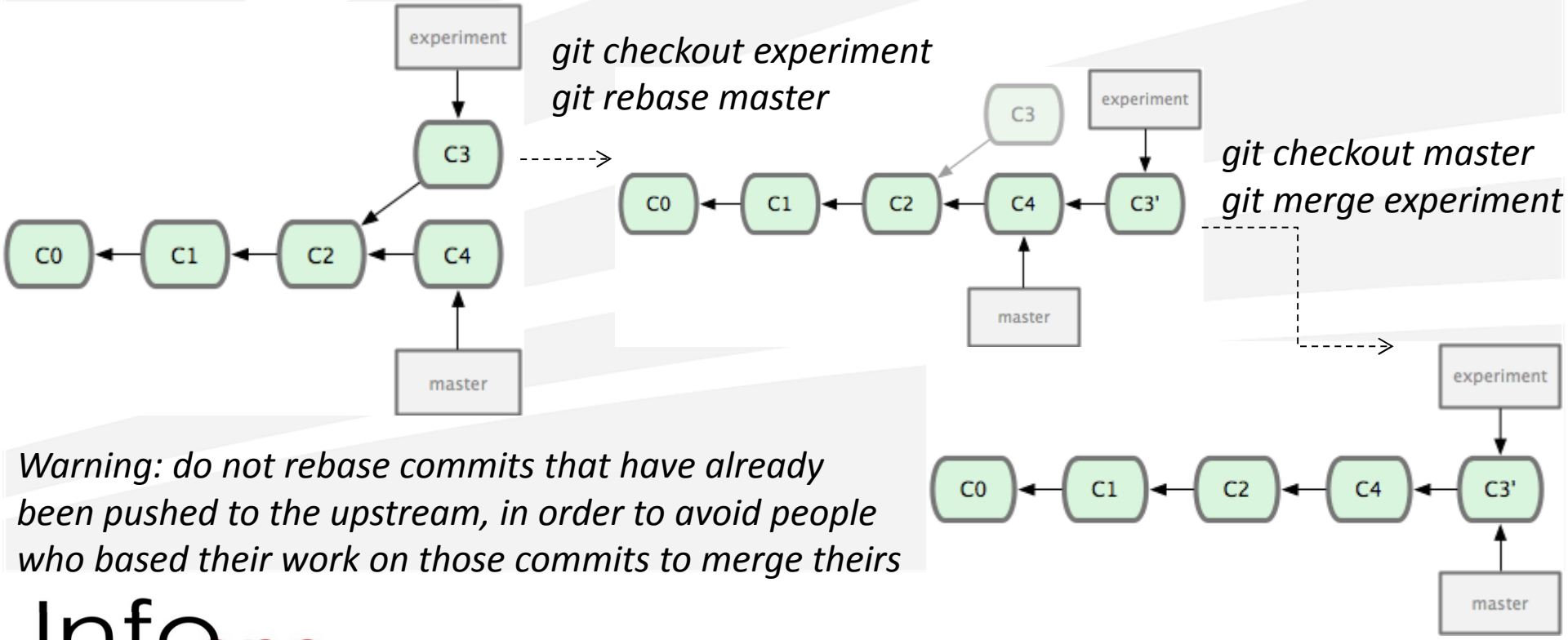
# Remote Branches: Synchronizing (2)

- Fetching does not automatically create a local, editable copy of a fetched branch
  - *git merge <server>/<branch-name>* has to be run to merge it into the local branch
  - *git merge checkout -b <branch-name> <server>/<branch-name>* has to be run in order to have a local copy of that branch, i.e. to create a tracking branch
    - *git merge checkout --track <server>/<branch-name>* if local and remote branch names are the same
- *git push <server> <branch-name>* synchronizes local branches with remote repository
  - Add *:[remote branch-name]* if local and remote names differ
- *git push <server> :<branch-name>* deletes the remote branch

Info one

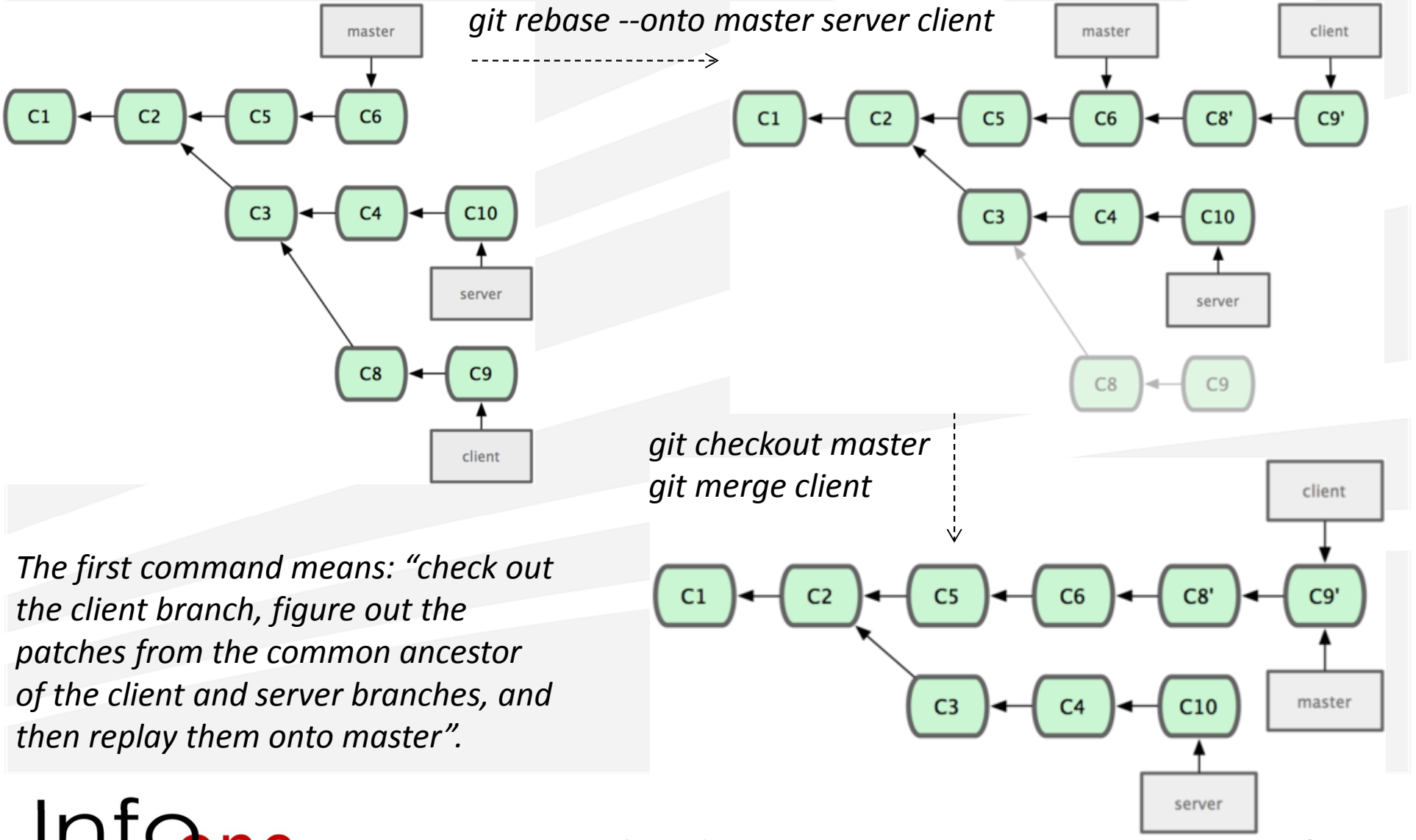
# Branch Rebase (1)

- An alternative to the three-way merge
  - It consists in applying the patch of the branch to merge on top of the branch to merge into




*Warning: do not rebase commits that have already been pushed to the upstream, in order to avoid people who based their work on those commits to merge theirs*

# Branch Rebase (2)

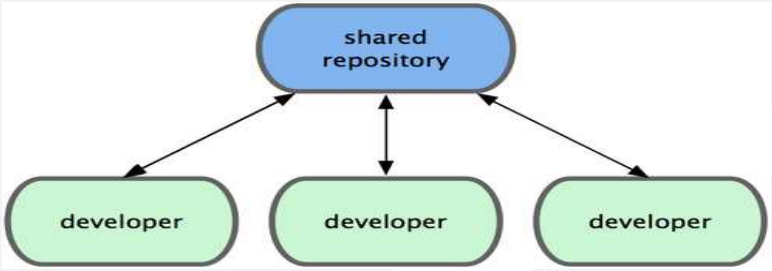



*The first command means: "check out the client branch, figure out the patches from the common ancestor of the client and server branches, and then replay them onto master".*

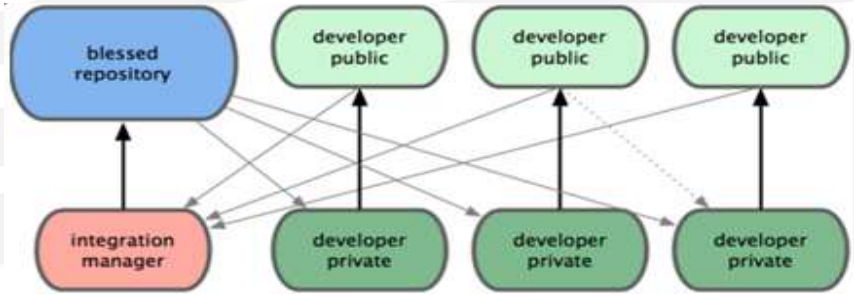
# ADVANCED USE

# Distributed Workflows

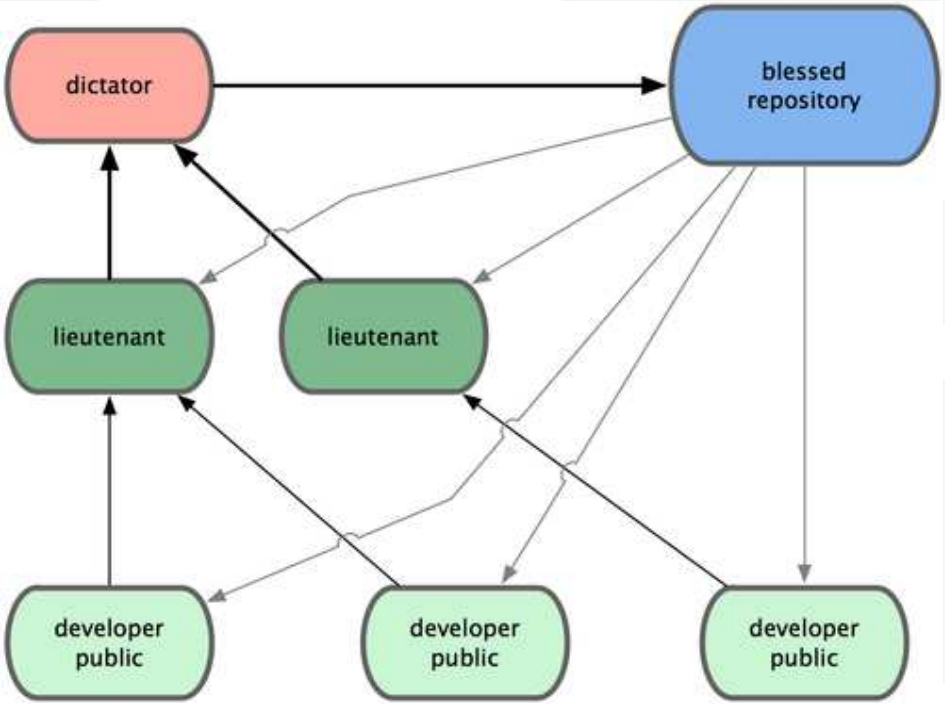
- Centralized workflow




- Integration-manager workflow
  - Developers clone the main repository and have their own public repository
  - Ask the manager to merge their repository into the main repository
  - The manager adds it as public and merge into the main repository




- Dictator and Lieutenants Workflow
  - Developers rebase the main repository
  - They ask the lieutenants to merge into their masters
  - They ask the dictator to merge into the main repository



# Stashing

- *git stash* stores the current “dirty” status of the commit (modifications and staging information) in a stack for future revert, but it does not commit anything
  - *git stash list* lists all stashes
  - *git stash apply <name>* applies the *<name>* stash (or the most recent if no name is specified)
    - If reverting to that stash is impossible due to changes to modified files into the stash, merge conflicts are generated
  - *git stash drop* removes the stash from the stack
  - *git stash pop* applies and drops the stash
  - *git stash show -p <name> | git apply -r* unapplies an applied stash
  - *git stash branch <branchname>* creates a new branch based on the commit current when stashing and applies the stash

Info one

# Changing Local History

- *git commit –amend* allows to modify last commit message
  - If files are added or removed, it updates the commit according to changes in the staging area
  - To change older commits, interactive rebase command is required
    - E.g. *git rebase -i HEAD~3* to change last 3 commits, then follow the interactive tool instructions
    - It allows to reording commits
    - It allows merging commits
    - It allows splitting commits
- *filter-branch* command for changes affecting all commits
  - *git filter-branch --tree-filter <command> HEAD*
    - *--tree-filter* applies *<command>* for all checked-out commits
      - E.g. *<command> = rm –f <file>*
  - Project root can be changes
  - Metadata, e.g. email address, can be updated

Info one

# Debugging with GIT

- *git blame [-L <range>] <file>* shows the list of commits that modified the lines in *<range>* in *<file>*
- *git blame -C [-L <range>] <file>* shows the list of all files where the lines in *<range>* of *<file>* were in their history
- *git bisect start, git bisect bad, and git bisect good <goodcommit>*
  - It marks the current commit as bad, <goodcommit> as good and checkouts the one half way, so to enable testing it
    - Using *git bisect bad/good* will keep on the binary search of the commit which introduced an issue
    - *git bisect reset* ends the search

Info one

# Submodules

- A project or library referenced by the current tracked project that has to be dealt with independently
  - *git submodule add <url> <name>* clones the repository at *<url>* in the subdirectory *<name>* of the current project
    - Information on the submodule are in the *.gitmodules* file
    - Notice that the current project has a snapshot of that repository and cannot have a symbolic reference (e.g. master)
    - When browsing to the directory of the submodule, command scope changes accordingly
    - When cloning a project with submodules, directories of submodules are cloned, but they are empty
      - *git submodule init* and *git submodule update* initialize and populate it
    - *git submodule update* has to be run whenever a change is made on the submodule and the reference in *.gitmodules* changes
      - Changes to the submodule should be always performed on a branch

Info one

# Subtrees

- Alternative to Submodules
- *git read-tree --prefix=<subdirectory> -u <branch>*
  - It pulls *<branch>* into *<subdirectory>* of the currently checked out branch (main branch)
  - The branch can be checked out and updated
  - Changes can be merged into the main branch
    - *git merge -s subtree <subtreebranch>*
      - *--squash pre-populates the comment*
    - git diff-tree -p <subtreebranch> must be used in place of diff to compare unstaged changes against <subtreebranch>
    - git diff-tree -p < remotesubtreebranch > must be used to compare unstaged changes against <remotesubtreebranch>

Info one

# Miscellaneous

- Many configuration can be via git --config
  - Formatting, colors, external tools, server action rights
- Configuration can be for all Git projects, for one project or for a path
  - Git Attributes
    - Binary and binary-like files
    - Keyword expansions-like behavior
    - Repository export
- Hooks
  - Pre-commit, prepare-commit-msg, commit-msg
  - Post-commit
  - Applypatch-msg, pre-applypatch, post-applypatch
  - Pre-rebase
  - Post-checkout
  - Post-merge
  - Pre-receive, post-receive
  - Update

Info one

# References

- http://git-scm.com/book/en/