GIT

Distributed Version Control and Source Code Management

Agenda

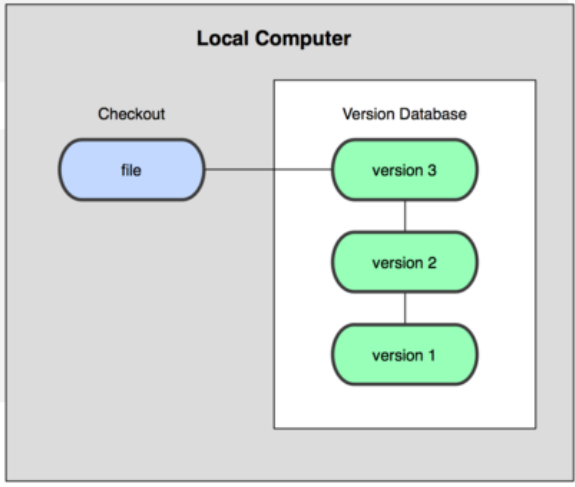
- Version Control Systems
- GIT
  - Basics
  - Branching

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 and compare changes made over time
  - see who last modified a file and when

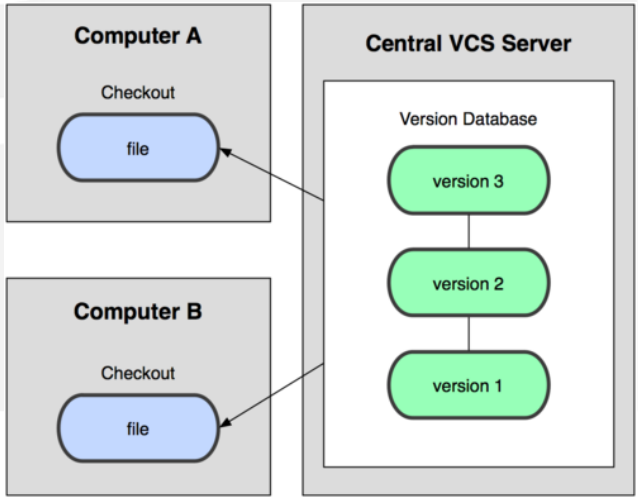
Local Version Control

- Many people's version-control method of choice is to copy files into another directory (perhaps a time-stamped directory)
  - Main 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.



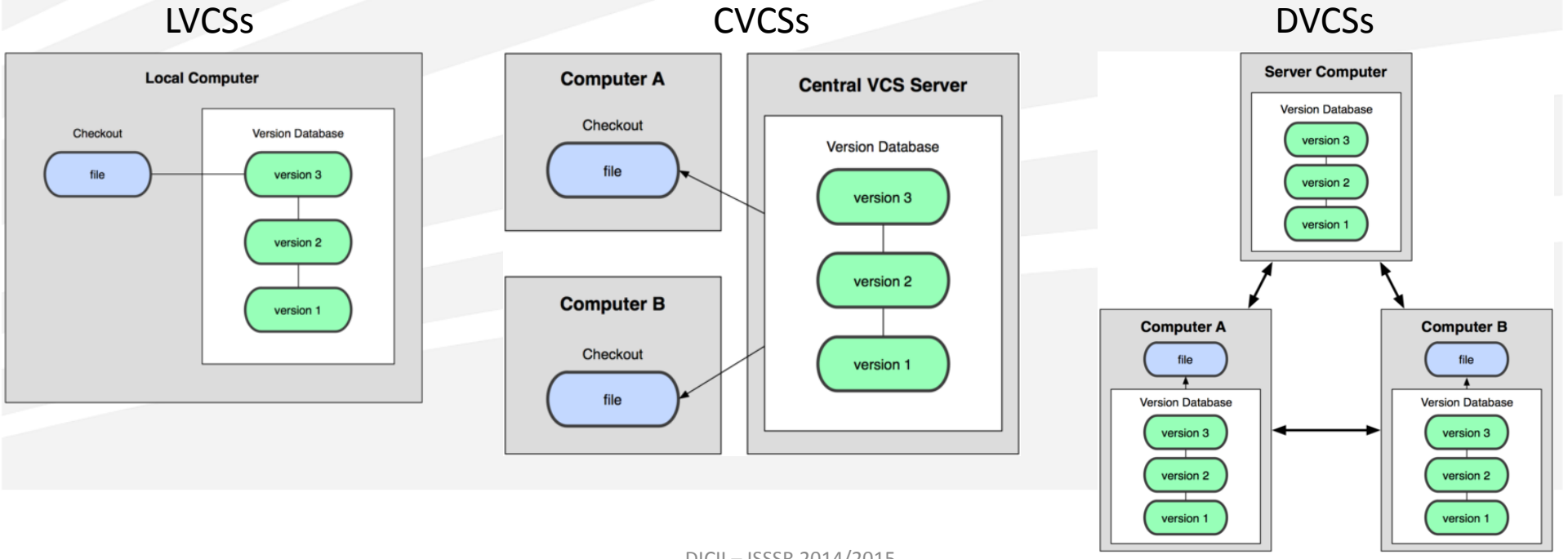
Centralized Version Control

- Main 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



Distributed Version Control

- Main 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

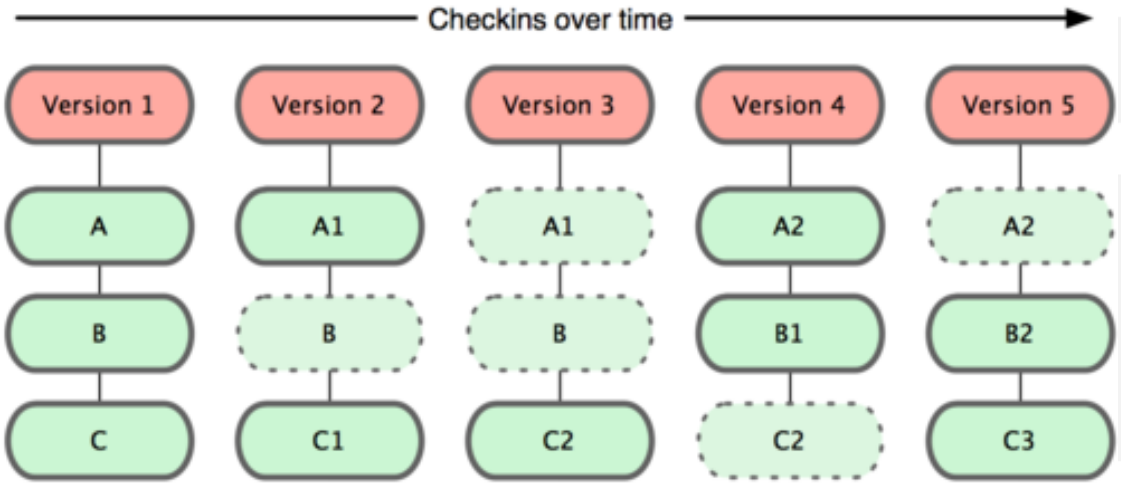
- History of Linux kernel source change management
  - 1991-2002: changes distributed as patches and archive files
  - 2002-2005: BitKeeper, a DVCS by BitMover
    - Bankrupt of the company
    - Creation of GIT by Linux community (headed by Linus Torvalds)
  - 2005-today: GIT
- Focus
  - Support for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

GIT: a Distributed Version Control System

- Most operations are just local
  - Browse history
  - Commit
  - Compare versions
- All changes are check-summed and can be referred to via such check sum (SHA-1)
- Almost all changes only add information to the database
- So, changes are tracked and can be reverted

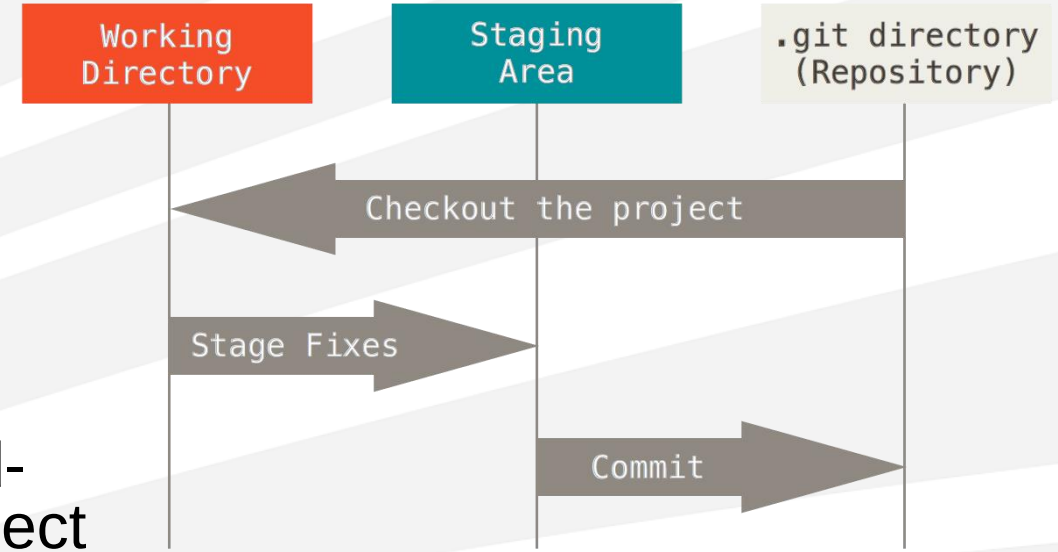
GIT Data Management

- Project history represented as a stream of project snapshots
- At every commit (i.e. the operation to create a “restore point”), GIT saves a snapshot
  - If files have not changed, they are not stored again



GIT States

- Committed
  - .git directory stores metadata and object database
- Modified
  - Working directory contains one checked-out version of the project currently being worked
- Staged
  - Staging area contains the index of staged files and their snapshots



Typical workflow:
1. Modify files in working directory.
2. Stage files, adding snapshots of them to the staging area
3. Do a commit, which takes the files from the staging area and stores them permanently to your Git directory

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 user.email <your email>*
- Set default editor
  - *git config --global core.editor <your editor>*
- Check configuration or get help
  - *git config { --list | <key> }*
  - *git help <key>*

Create a Repository

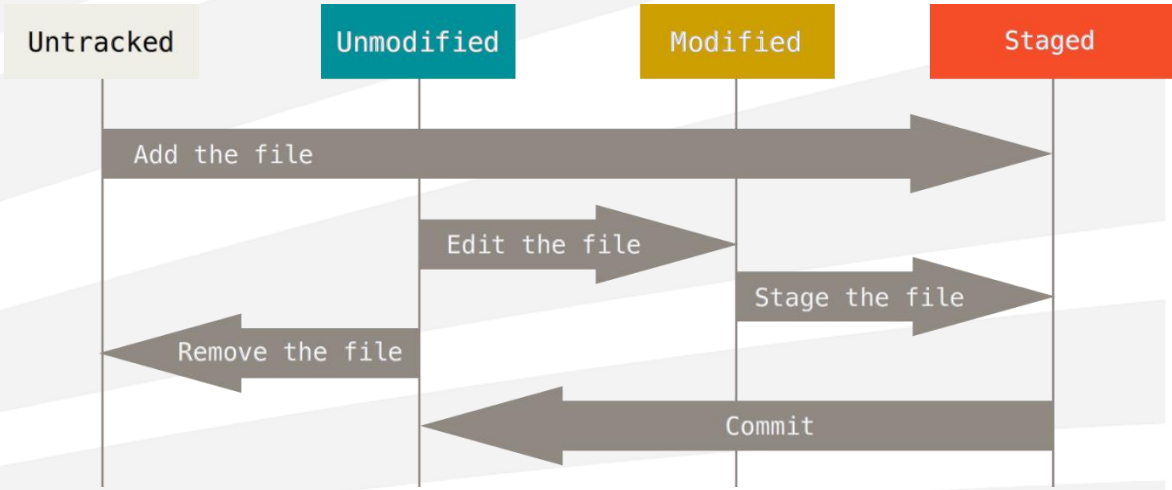
- Point at the directory where to create the repository in
- Initialize the repository: *git init*
  - It will create the .git subdirectory
  - Also a working directory is created and pointed at: **master**
- No files are being tracked

Clone a Repository

- Download all files required to have a local copy of the entire repository
  - *git clone <url> [repository name]*
    - *{ 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*> and they are all tracked

File Status Lifecycle

- Files in the working directory of the repository can be tracked or untracked
- Untracked: not in last snapshot nor in staging area
- Tracked
  - In last snapshot
  - Can be
    - Unmodified
    - Modified
    - Staged



- 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 --staged* shows what is staged and is going to be committed
  - *git reset HEAD <file>* unstages staged files

Ignoring Files

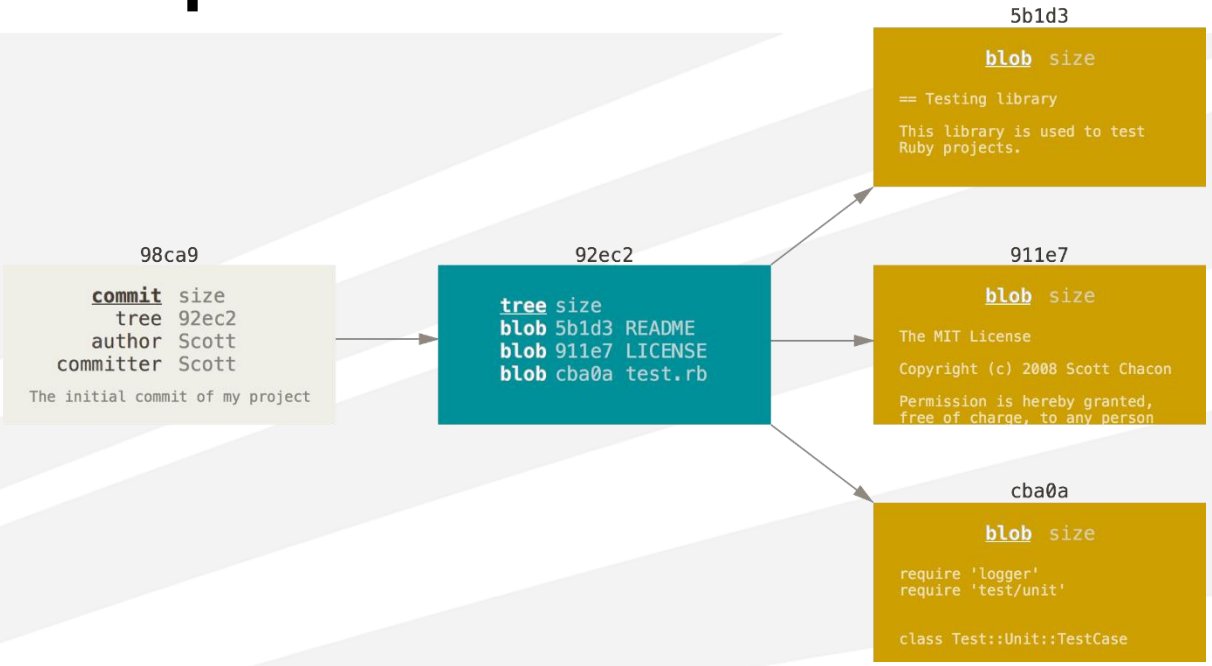
- The .gitignore file contains a list of files and folders that should be not committed
  - E.g. automatically generated log files, temporary files, object or binary files
- Files and folders are specified via rules
  - Glob patterns
  - / at the end indicates a directory
  - ! at the beginning indicates a negation
  - # for comments
- Details: http://git-scm.com/book/en/v2/Git-Basics-Recording-Changes-to-the-Repository#Ignoring-Files

```
#Example
.settings
.springBeans
bin
build.sh
/build
target/
.classpath
.project
```

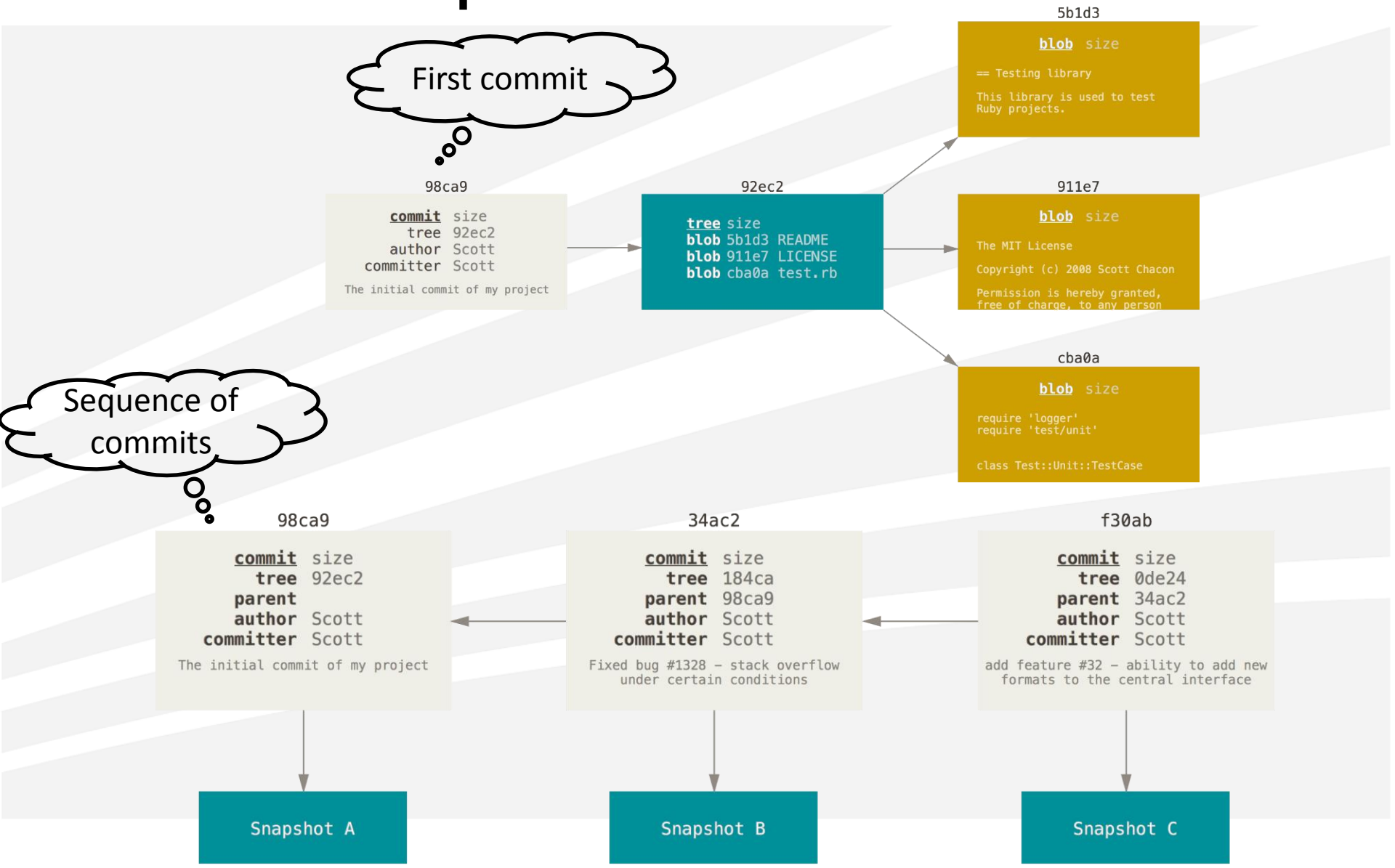
Committing Files

- *git commit*
  - Staged files are committed
  - Runs the editor and opens a file containing the output of *git status*
    - *git* commit *-m "<comment>"* allows to add a comment and skit the editor
  - After committing, the impacted branch and its checksum are shown
- *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
  - http://git-scm.com/book/en/v2/Git-Basics-Viewing-the-Commit-History

An Example of Commit Results



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 <file>* 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 (safety feature to avoid data loss)
      - If you want to remove a file from the staging area and untrack it without removing it from the working directory, use *git rm –cached <file>*
- 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>*

Tagging

- Tags are labels to associate to commits, e.g. to mark release points
- *git tag* shows all available tags alphabetically
- *git tag -a <tagname> -m '<message>'* creates a new tag named <tagname> and stores it
  - Tagger info, date and message are also stored
  - 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
- Tags have to be pushed
  - One at a time: *git push <repository> <tagname>*
  - All at once: *git push <repository> --tags*

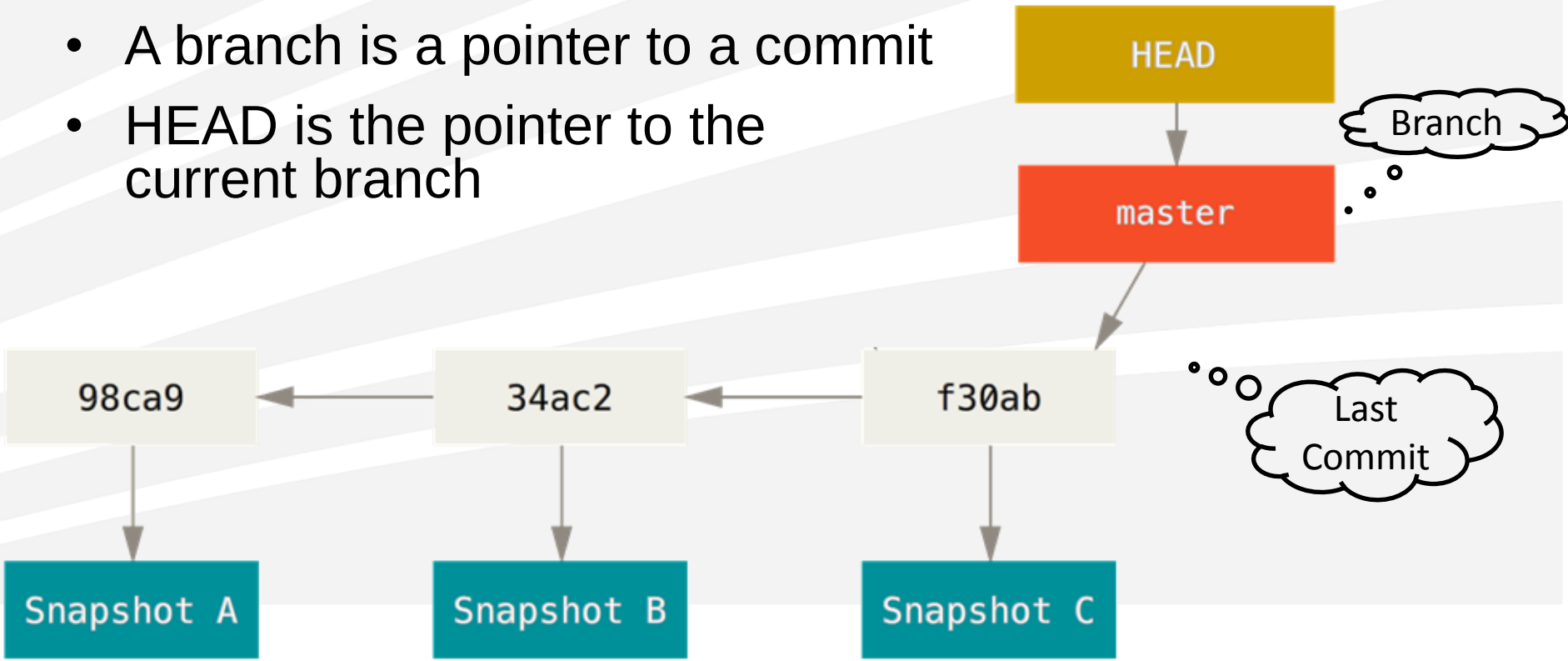
Stashing

- *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

BRANCHING

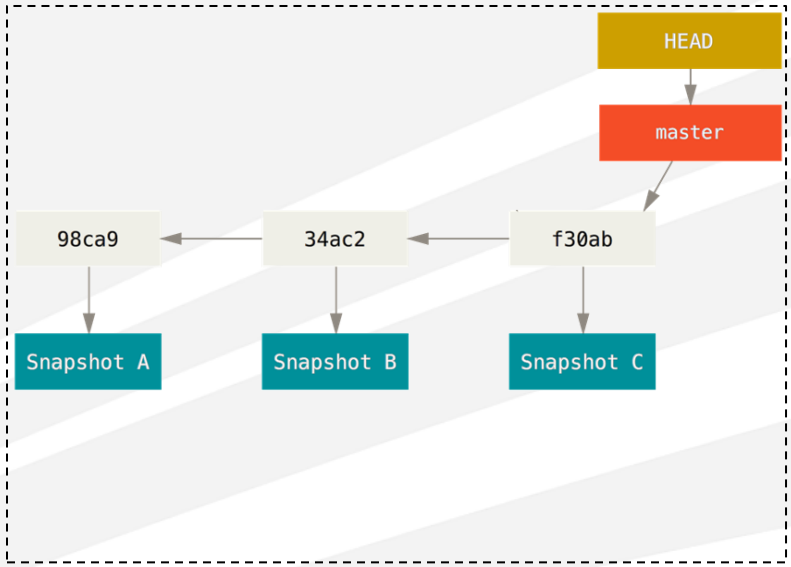
Branching: Create a Branch

- After the first commit, the repository has at least one commit (generally named **master**)
- Usually, a deviation in the main line of development
  - *git branch <branch name>*
- A branch is a pointer to a commit
- HEAD is the pointer to the current branch



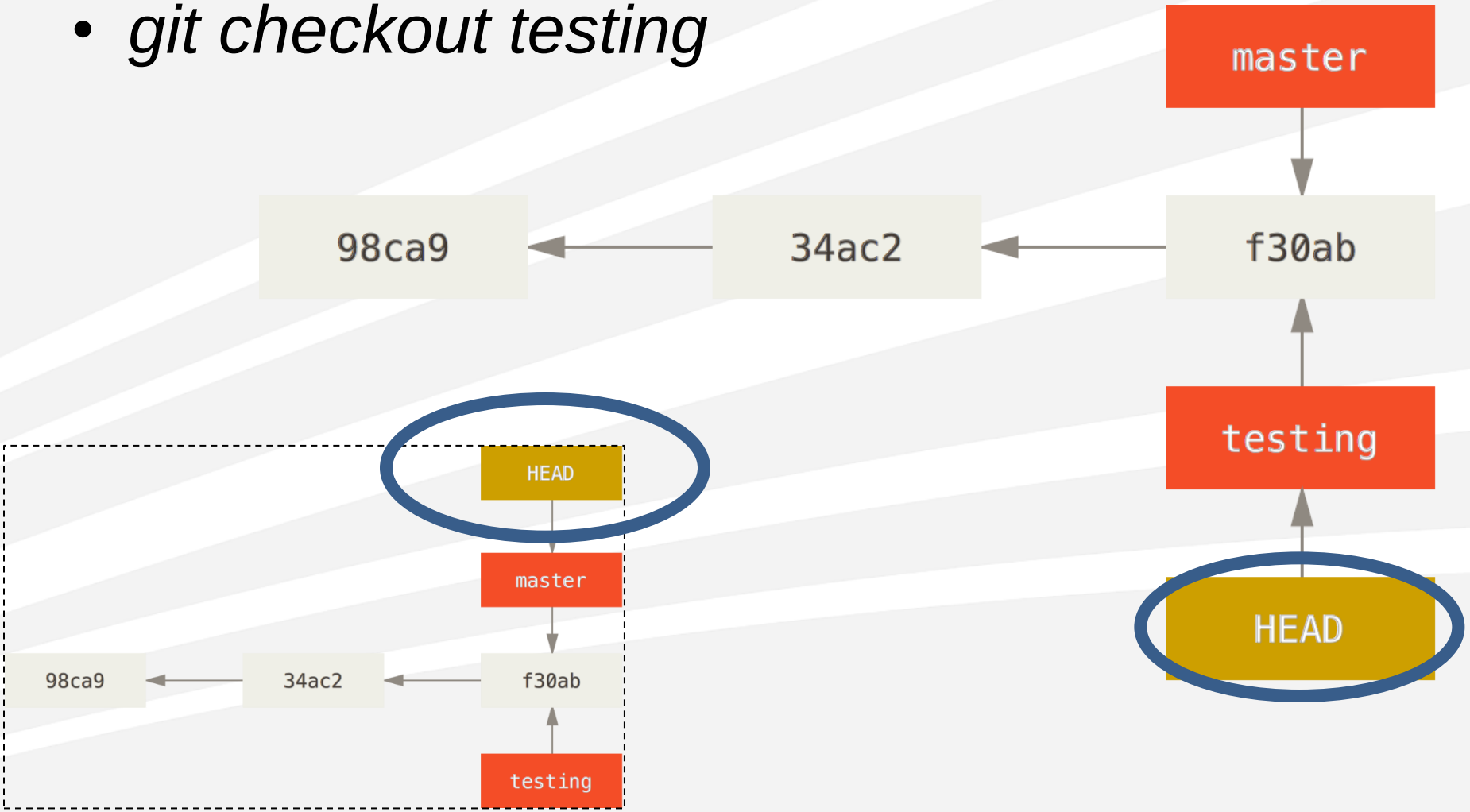
Branching: Create a Branch

- *git branch testing*



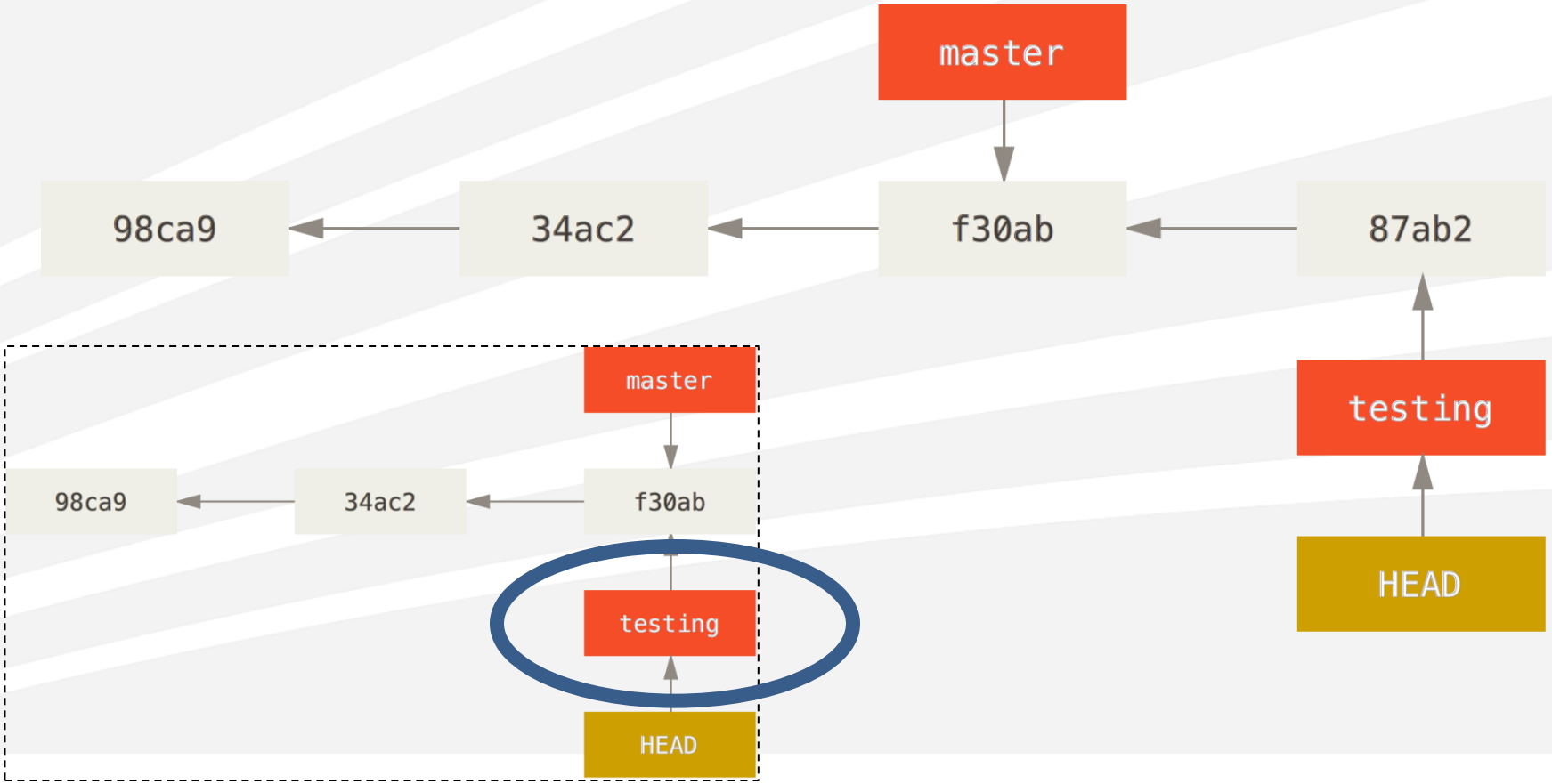
Branching: Switch Branch

- *git checkout testing*



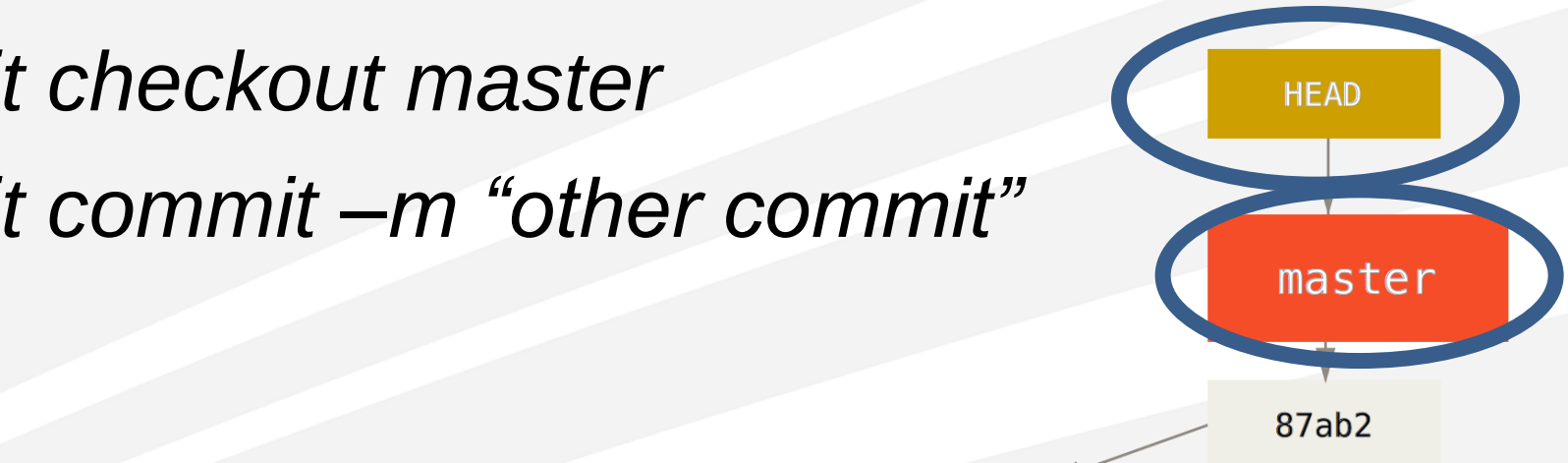
Branching: Impact of a Commit

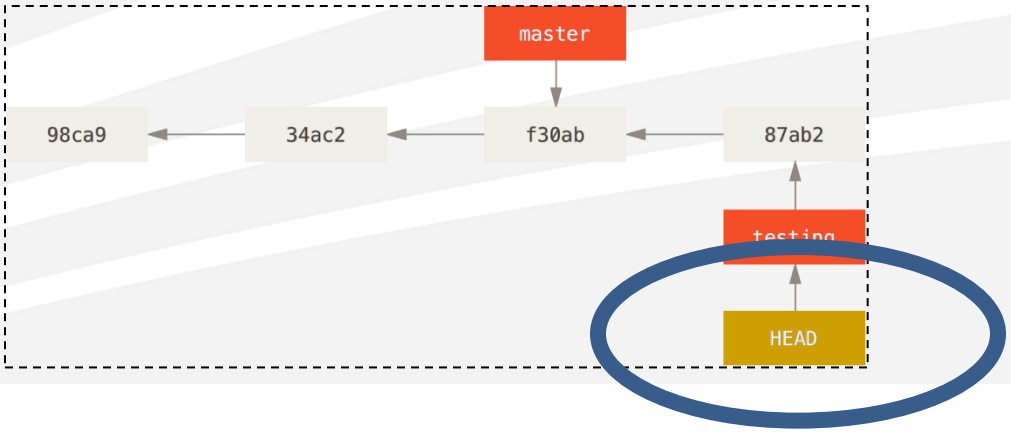
- *git commit –m “committed change”*



Branching: Multiple Branches

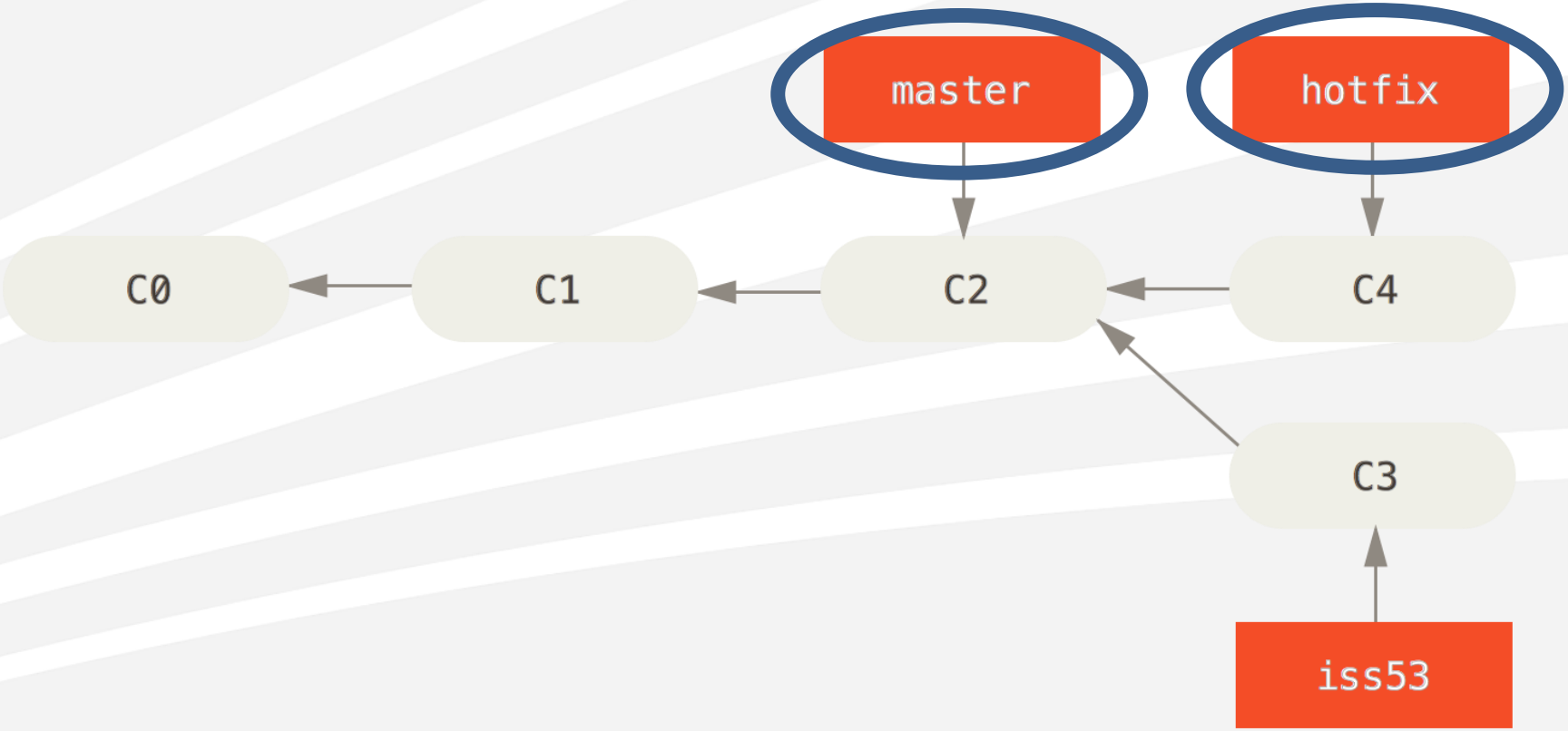
- *git checkout master*
- *git commit –m “other commit”*





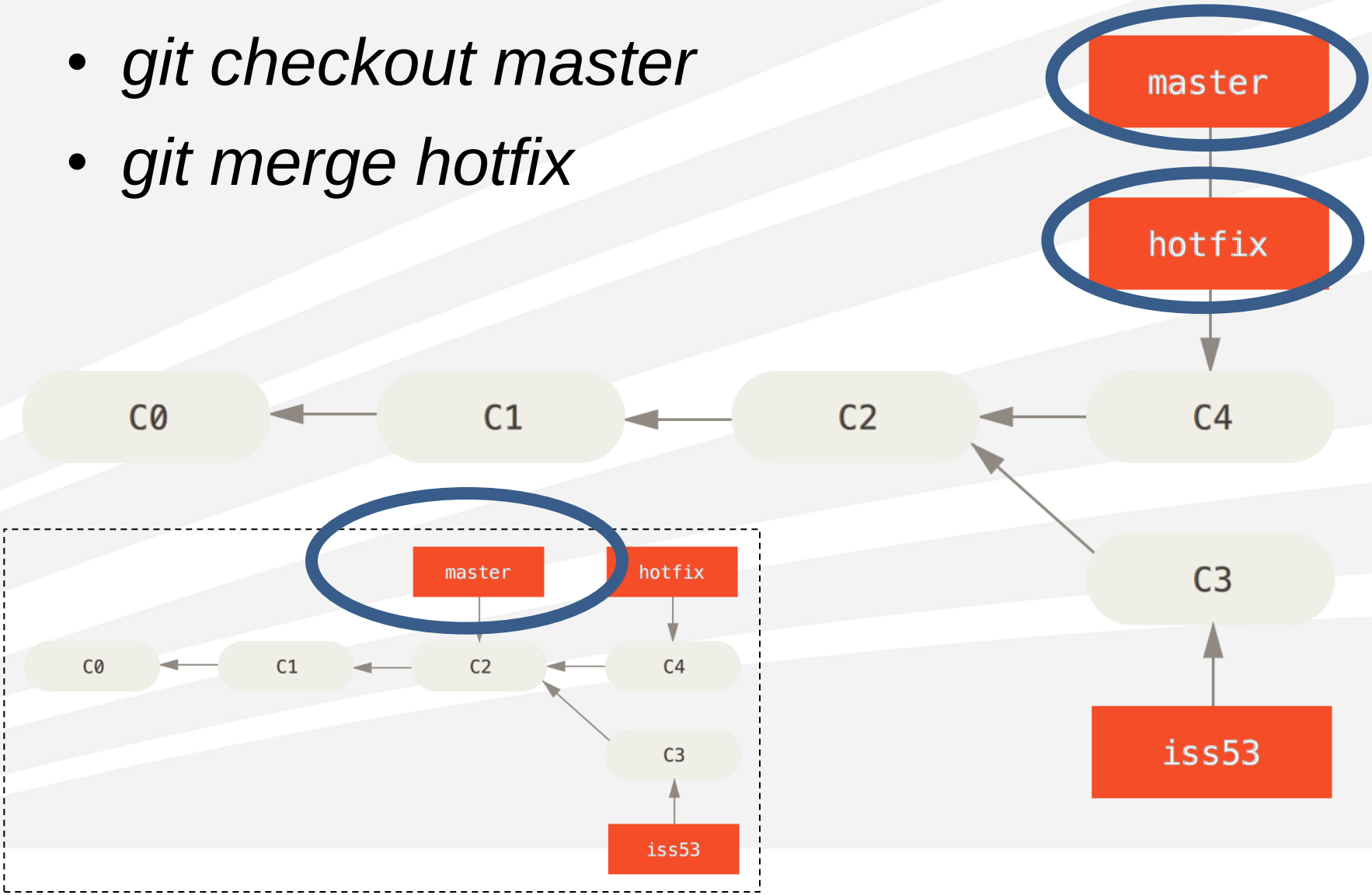
Merging Branches: Fast-Forward

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



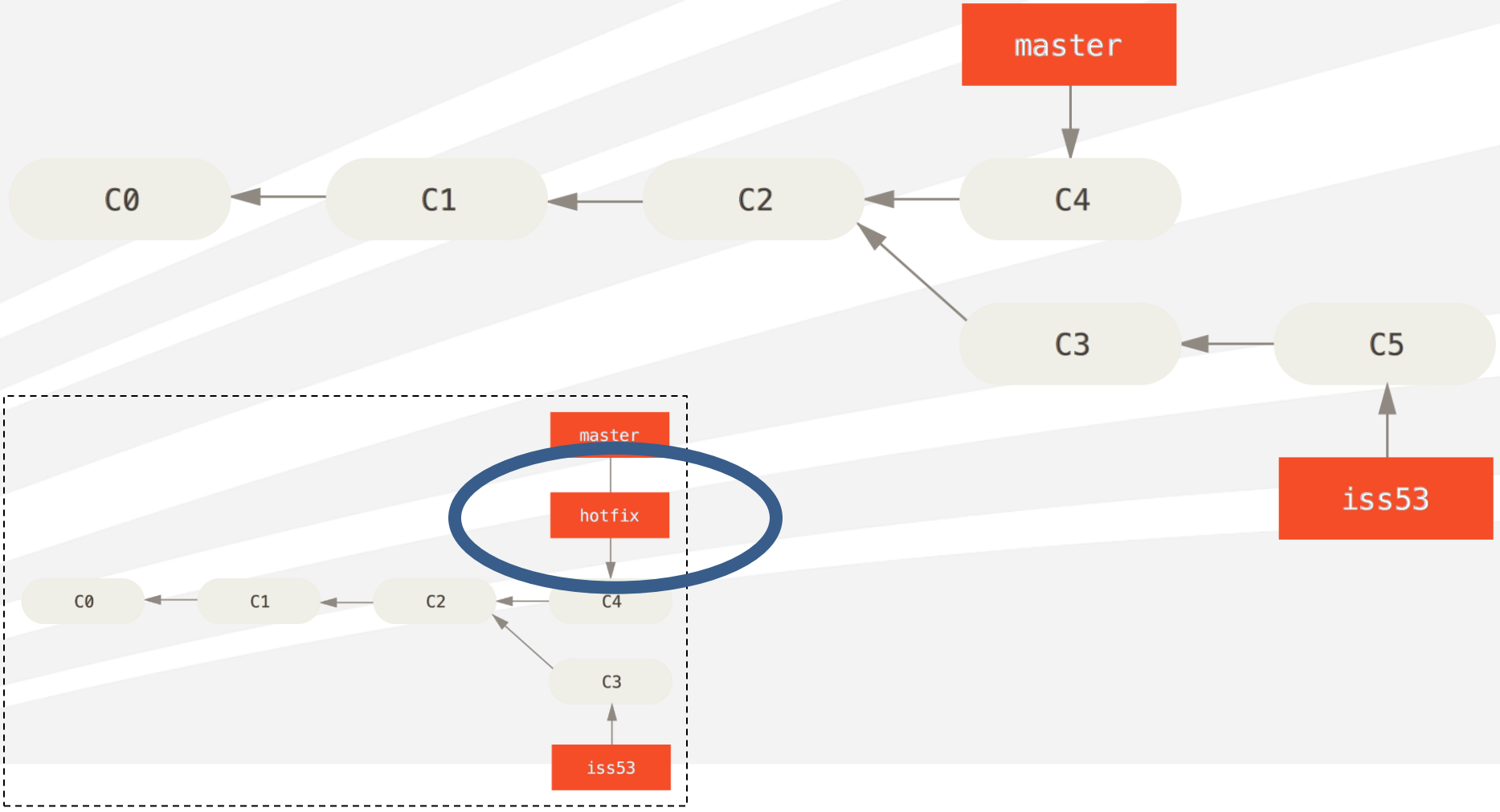
Merging Branches: Fast-Forward

- *git checkout master*
- *git merge hotfix*



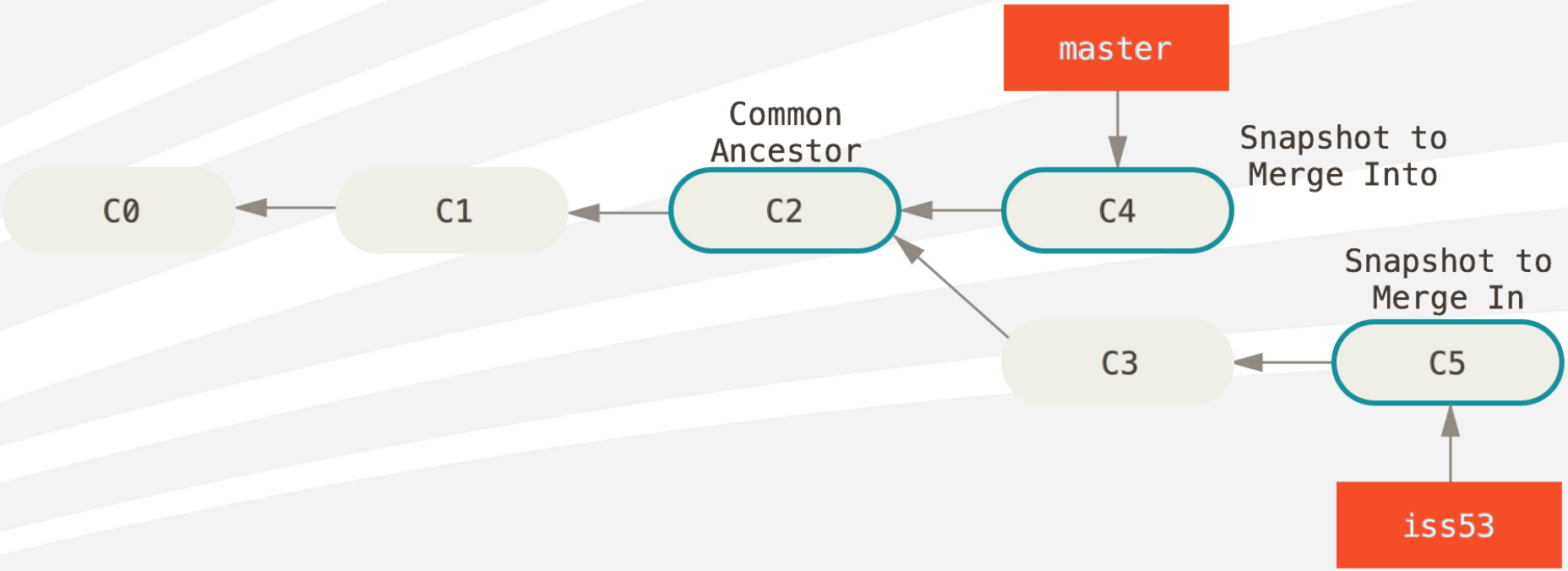
Merging Branches: Remove Branch

- *git branch -d hotfix*



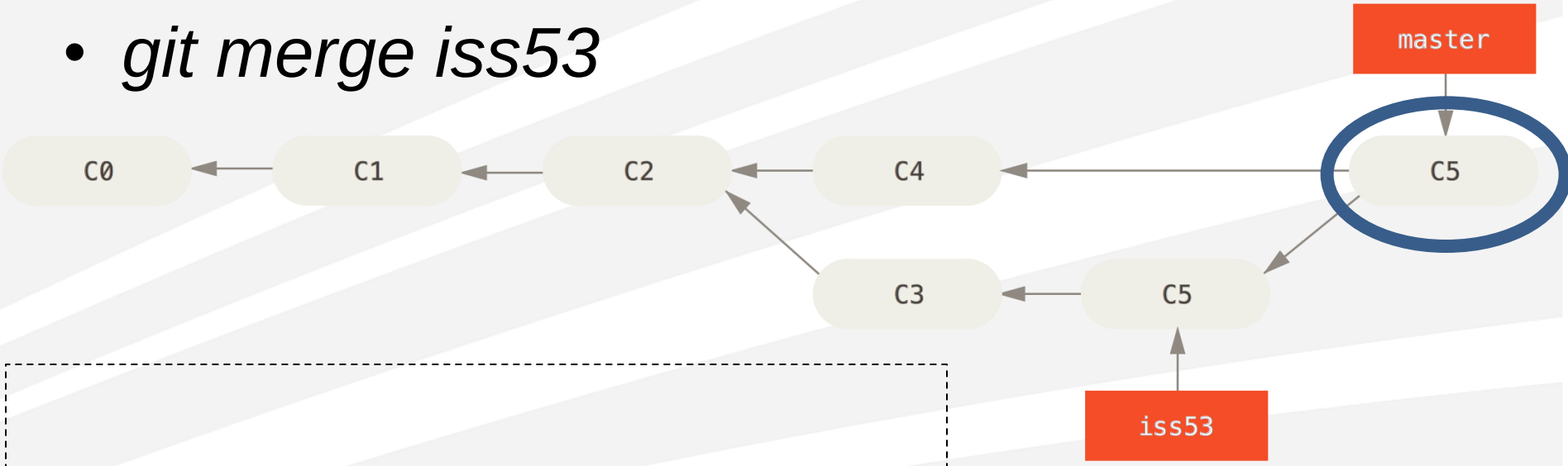
Merging Branches: 3-Way Merge

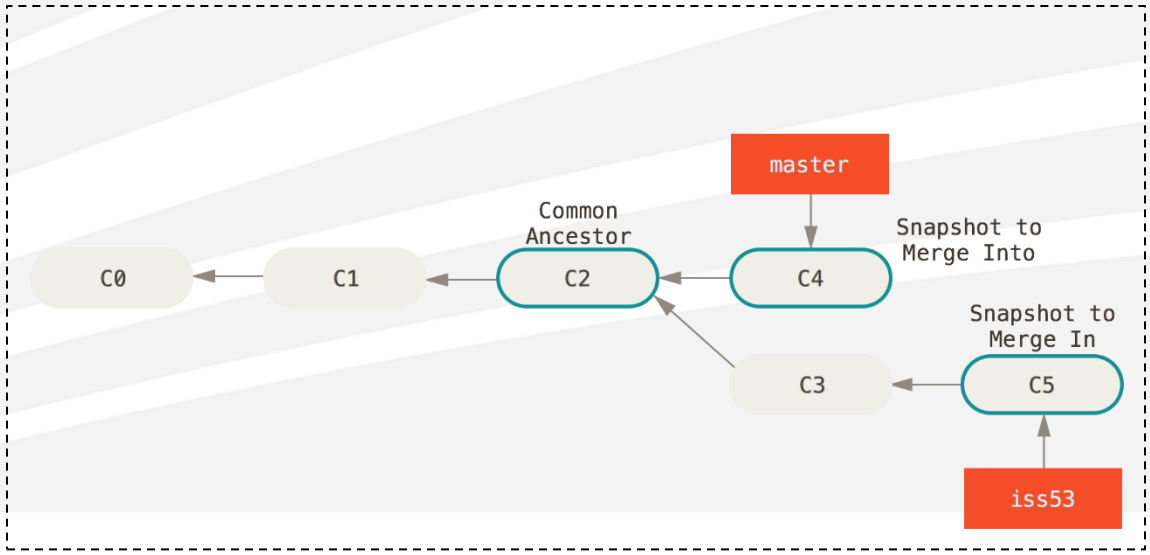
- *git checkout master*
- *git merge iss53*



Merging Branches: 3-Way Merge

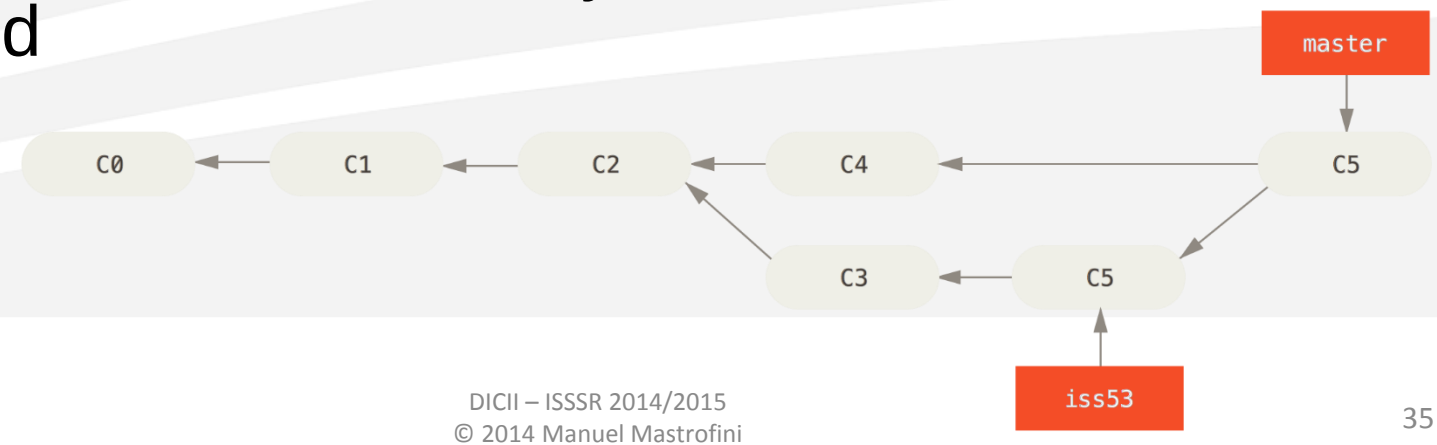
- *git checkout master*
- *git merge iss53*





Merging Branches: Conflicts

- If the same part of the same file was modified in the two branches to merge, there is a conflict and merging is suspended
  - E.g. Hotfix branch included changes on the same files as isss53
- Command *git status* shows unmerged files
- File markers are inserted into the conflicting files
  - E.g. “<<<<<<< HEAD”, “=======“ and “>>>>>>> isss53”
- Conflict has to be manually solved, then files must be re-added



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 it has been merged into the current branch
  - In order to force removal, use -D in place of -d

Remote Repositories

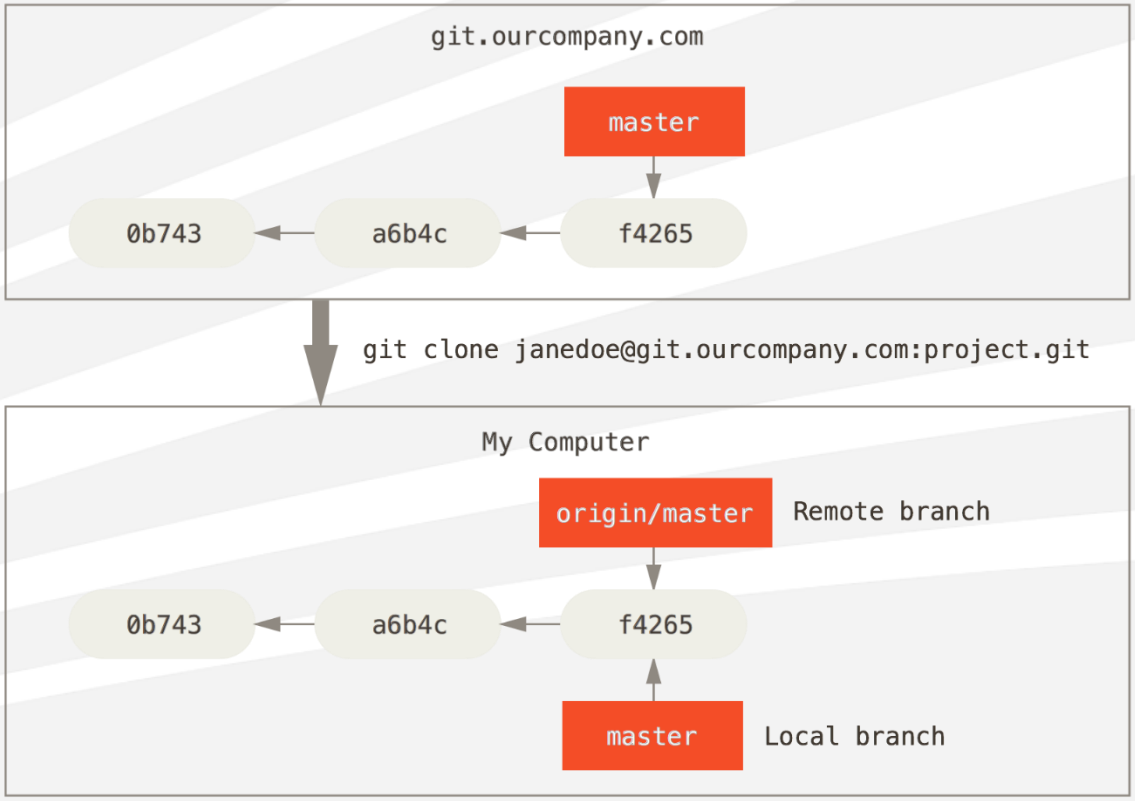
- Remotely stored versions of the project
  - *git remote [-v]* shows all remote repository names (and URLs)
    - *git remote show* <*name*> presents additional information on <name>
  - *git remote add* <*repo*> <*url*> creates a new remote repository
  - *git remote rename* <*original name*> <*new name*> renames the repository
  - *git remote rm* <*name*> removes the repository

Remote Repositories

- *git push <repo> <branch>* pushes the project to the remote repository, specifically on the named branch
  - It works only when writing is allowed
  - It works only when nobody else pushed after last local fetch
  - Add :[remote branch-name] if local and remote names differ
- *git fetch <repo>* pulls all data not yet pulled
  - *git fetch origin* pulls any new work pushed to the server
  - No merging is performed
  - Fetching does not automatically create a local, editable copy of a fetched branch
    - *git merge <server>/<branch-name>* merges it into the local branch
    - *git merge checkout -b <branch-name> <server>/<branch-name>* creates a tracking branch
      - *--track* in place of <*branch-name*> to use the remote name

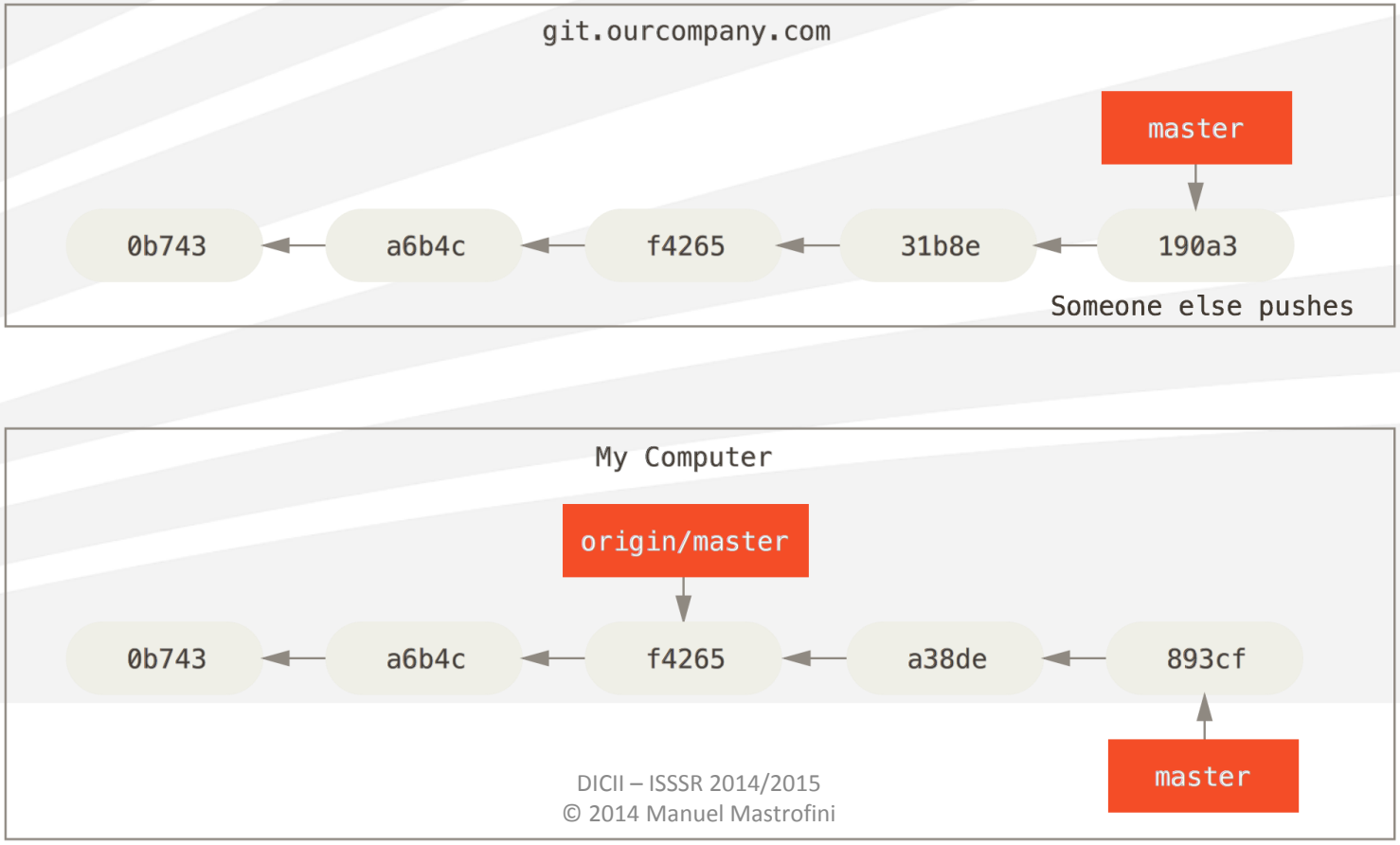
Remote Branches

- References to the state of branches on the remote repository
  - E.g. clone the master branch from ”ourcompany” server



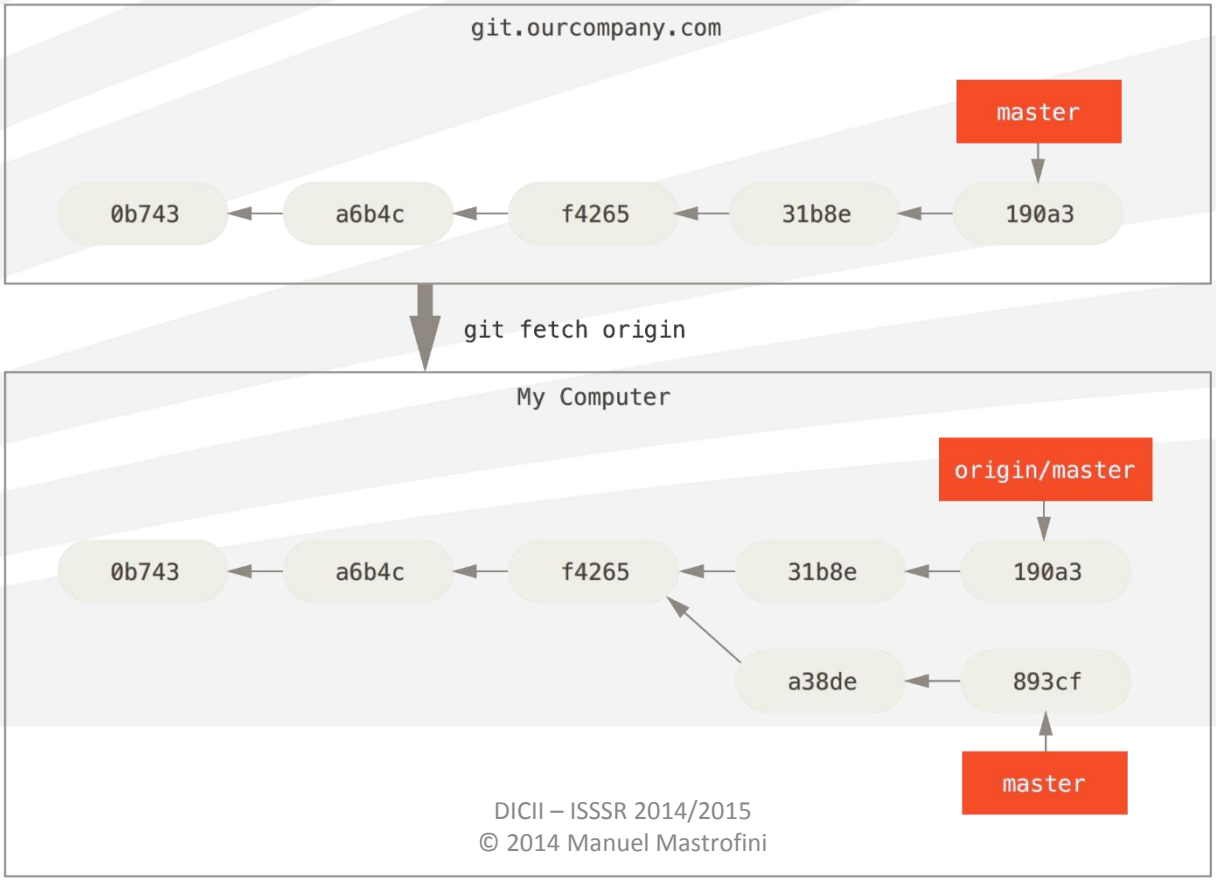
Remote Branches

- 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



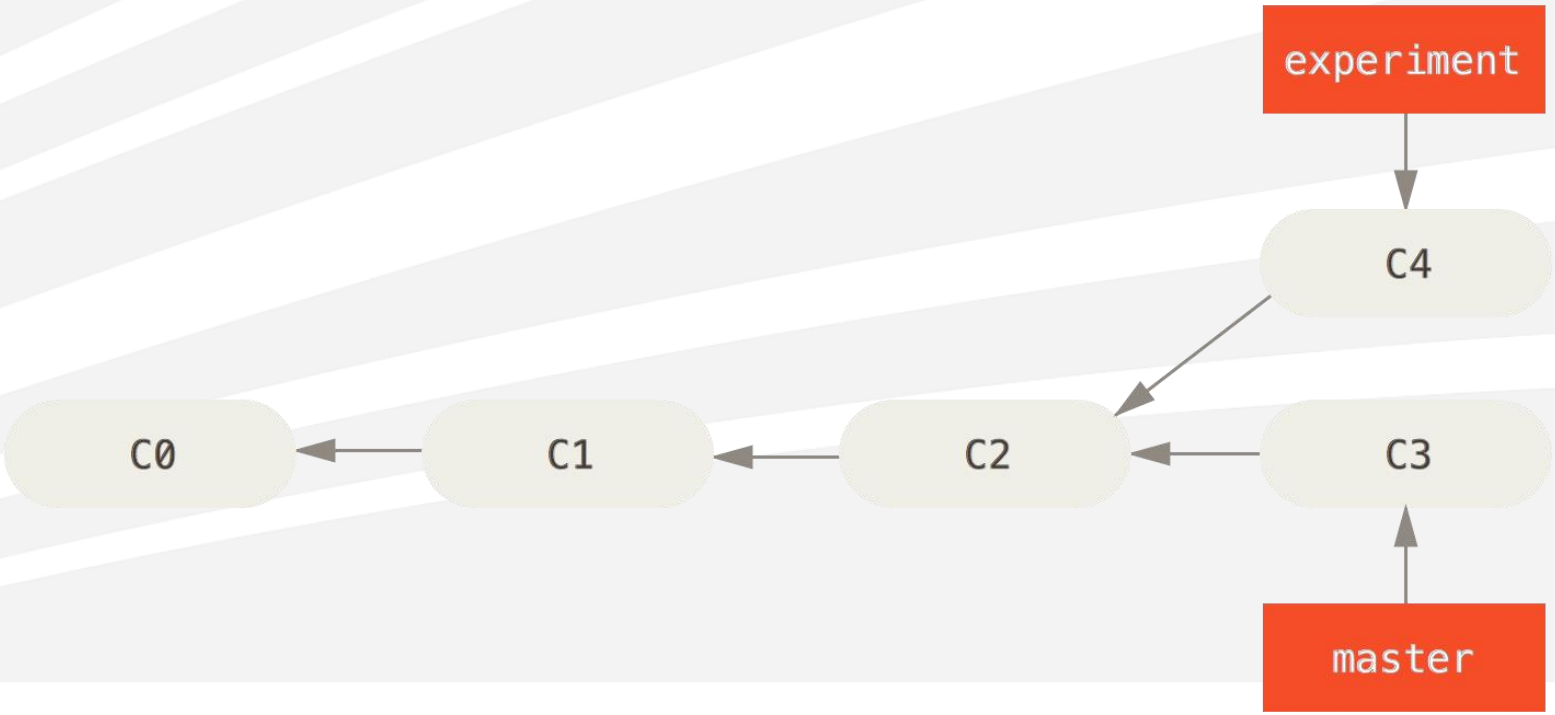
Remote Branches: Synchronizing

- *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



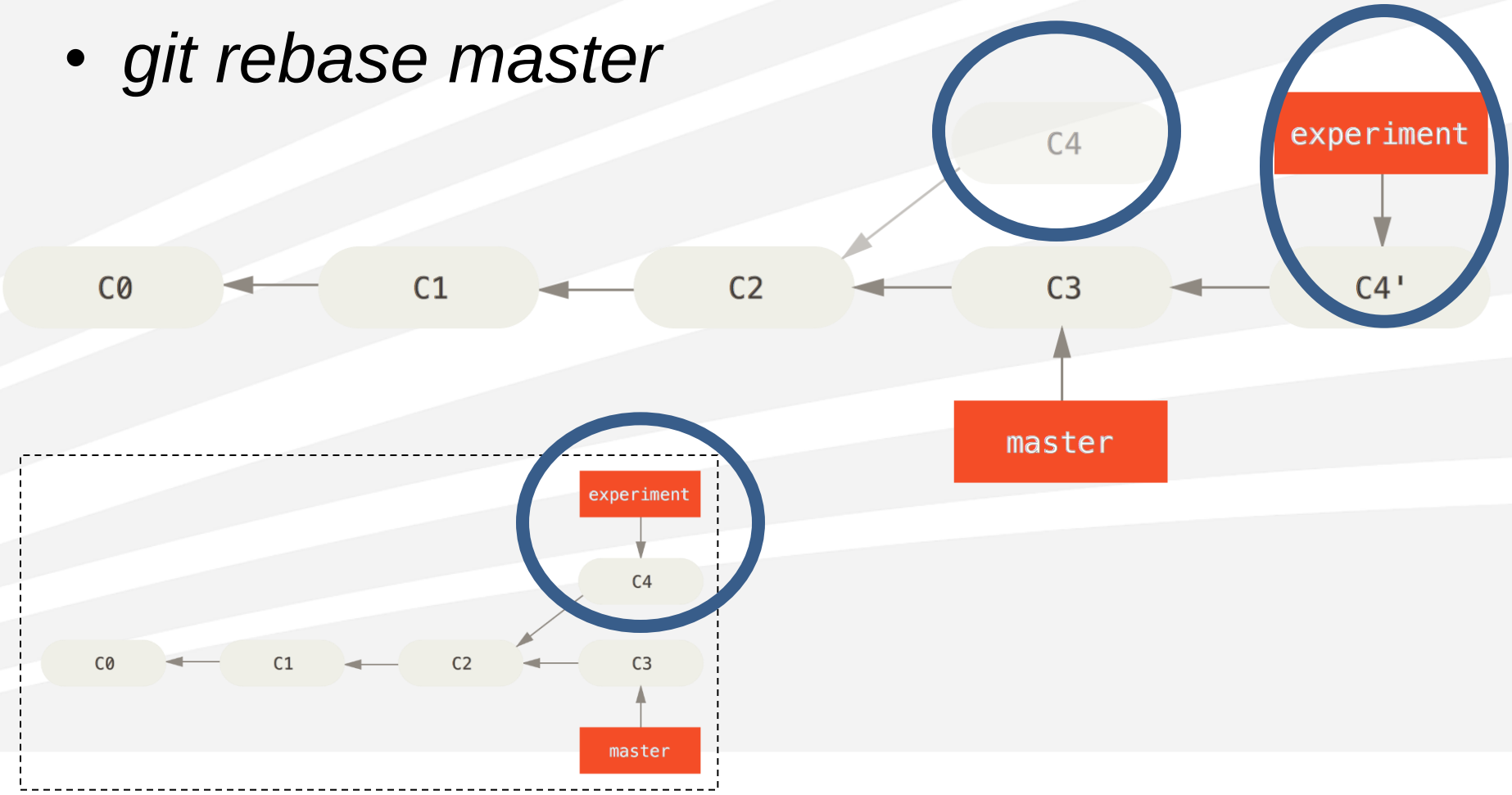
Branch Rebase

- 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



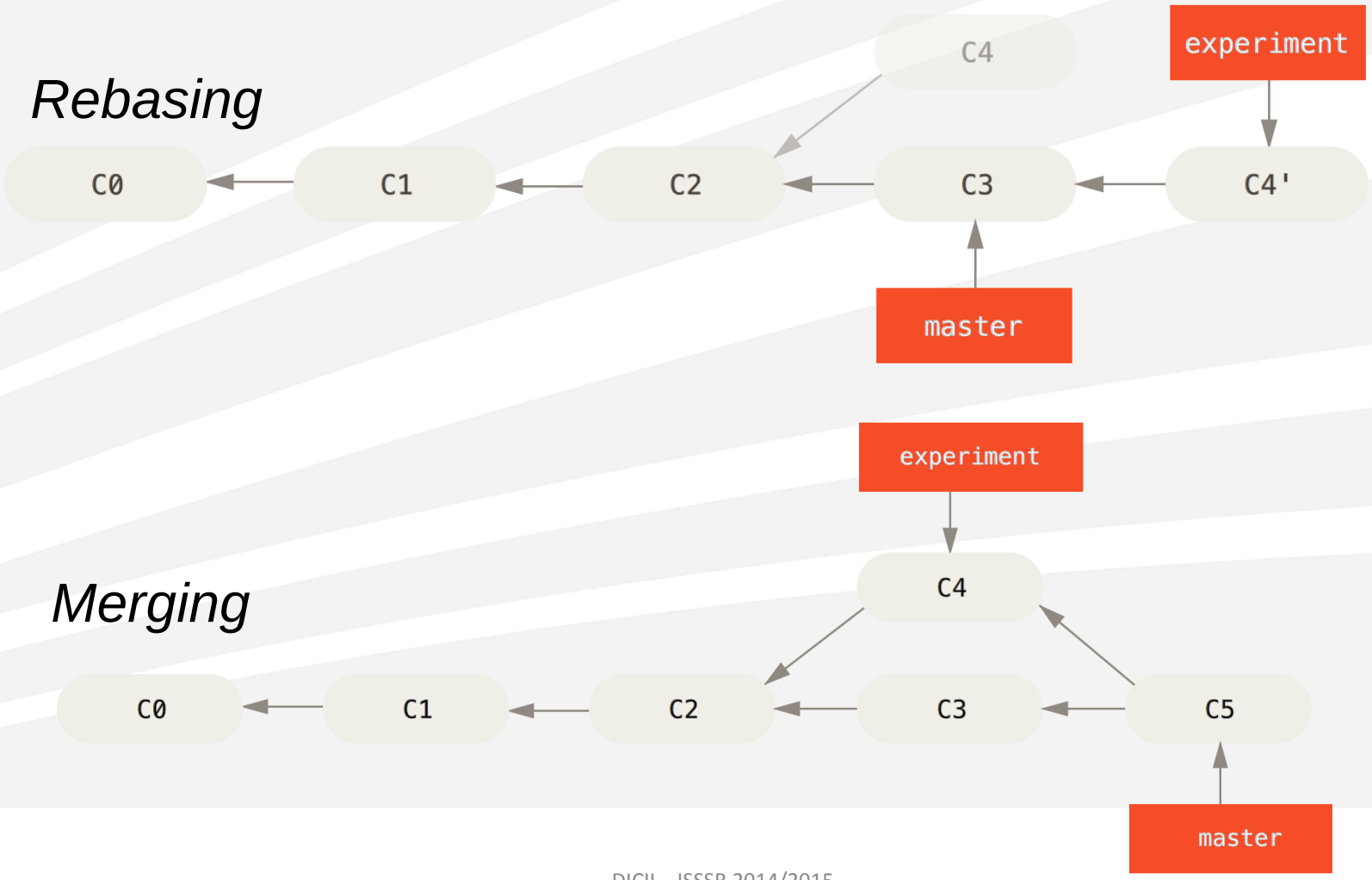
Branch Rebase

- *git checkout experiment*
- *git rebase master*



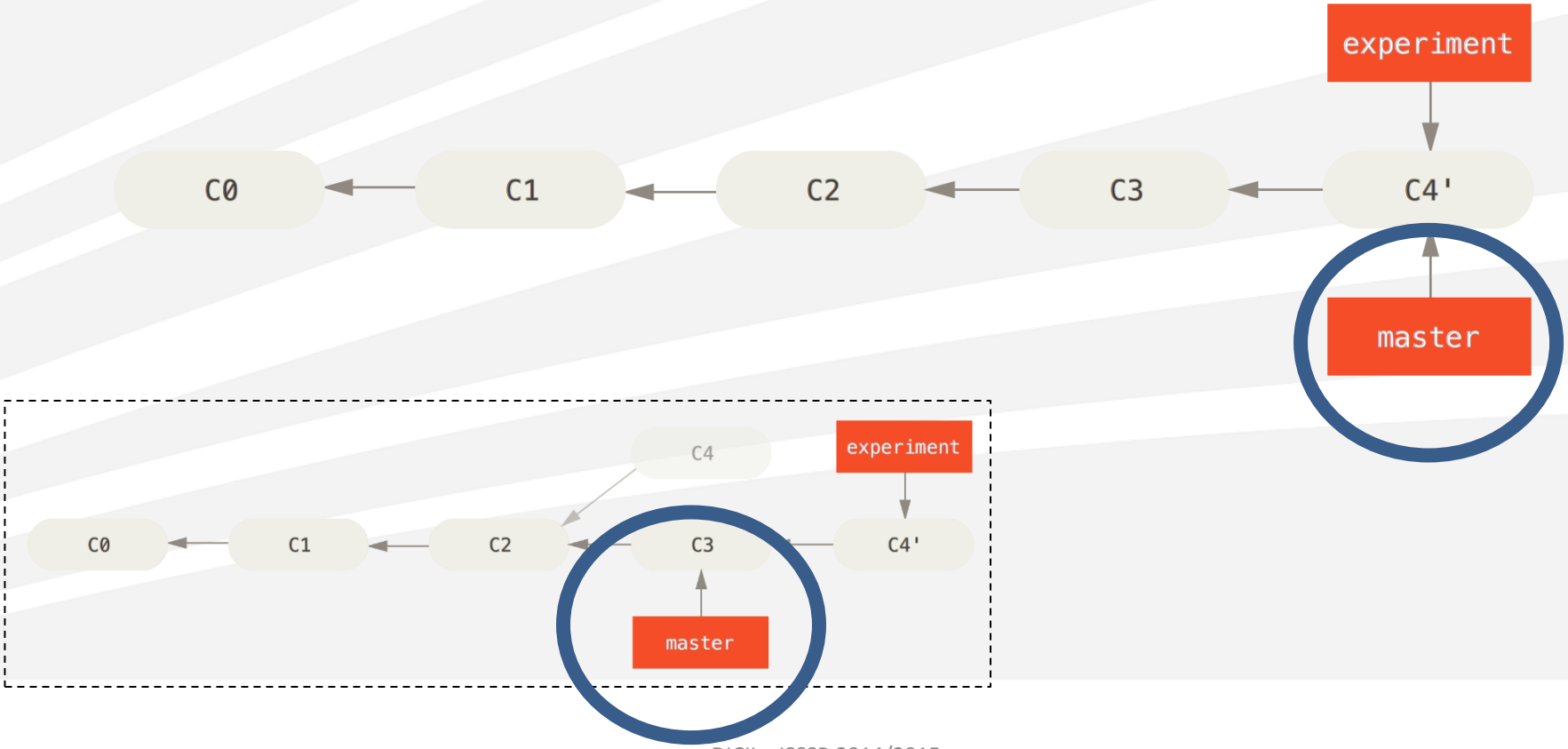
Branch Rebase

- Different from merging



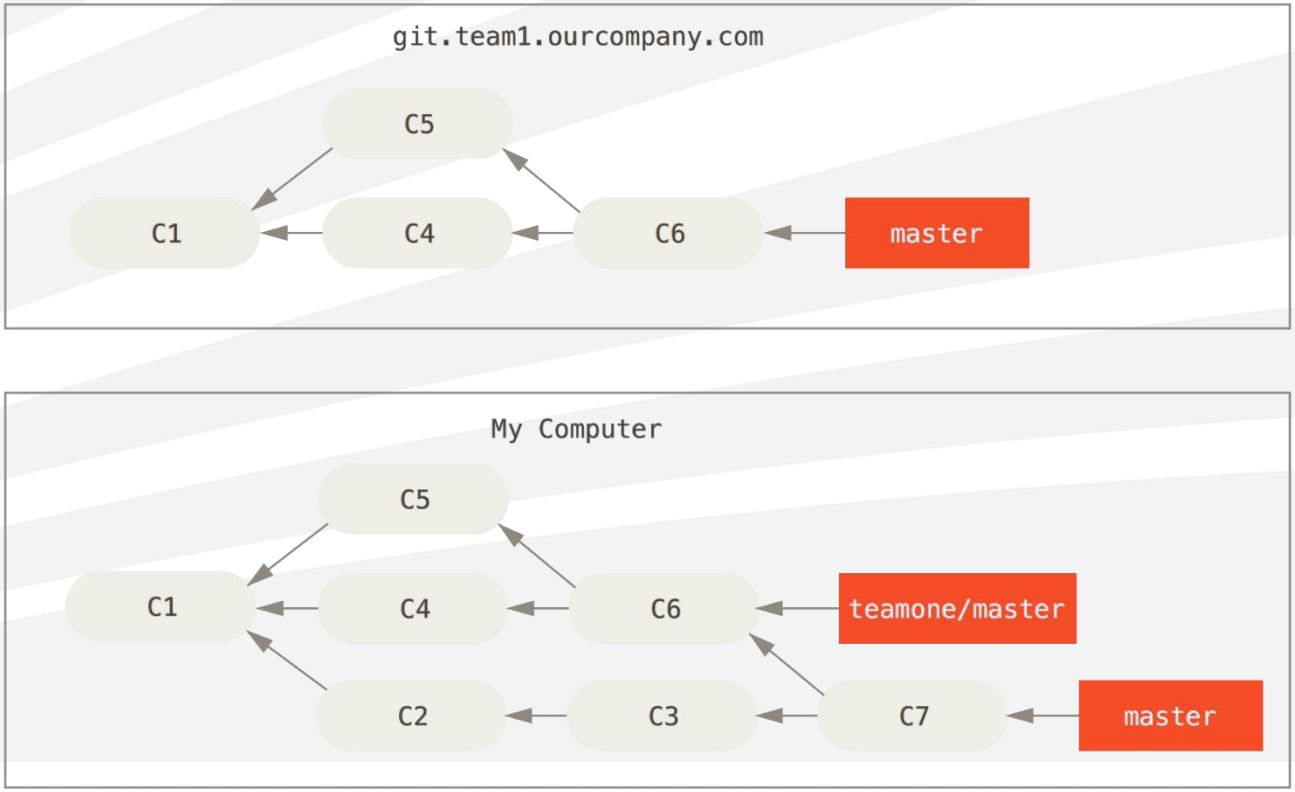
Branch Rebase

- *git checkout master*
- *git merge experiment*



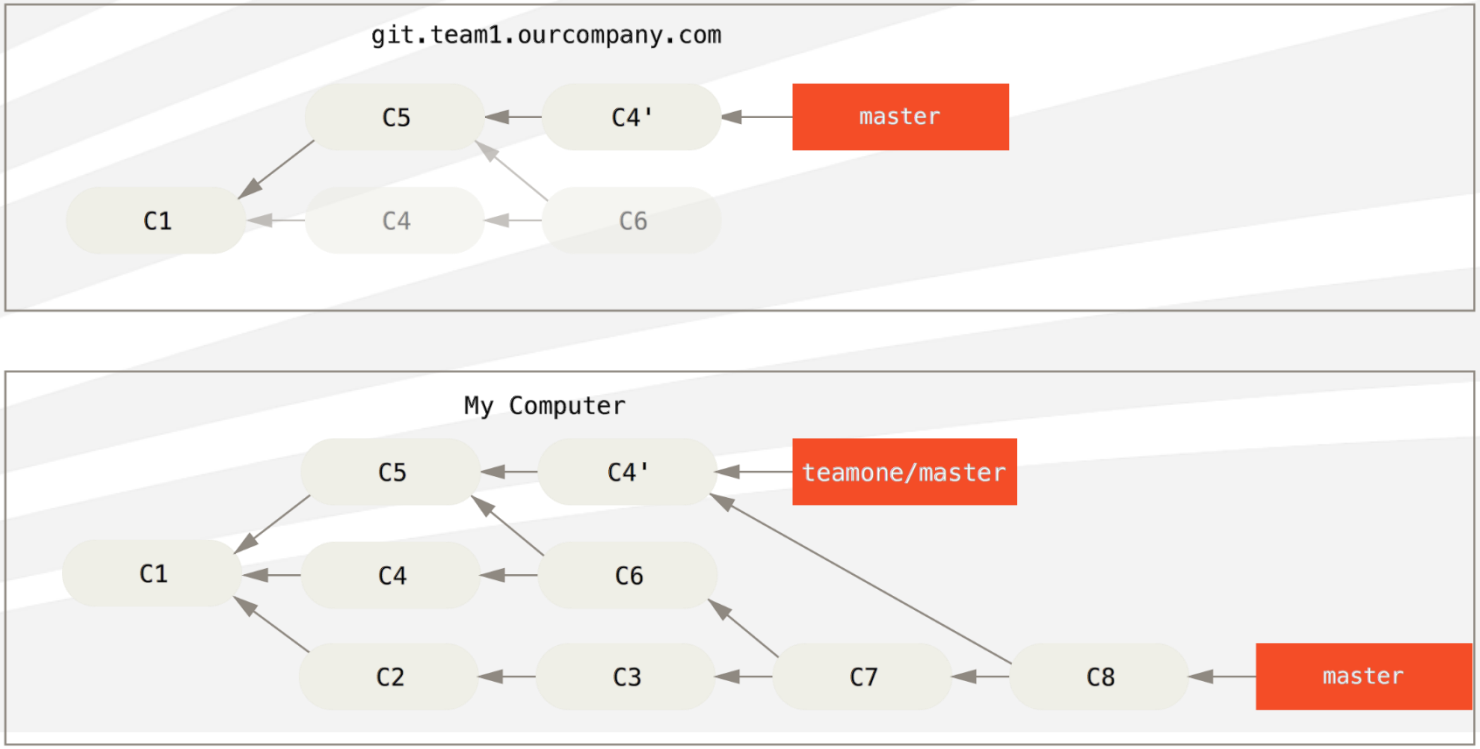
Branch Rebase

- *Warning: do not rebase commits that have already been pushed to the upstream*



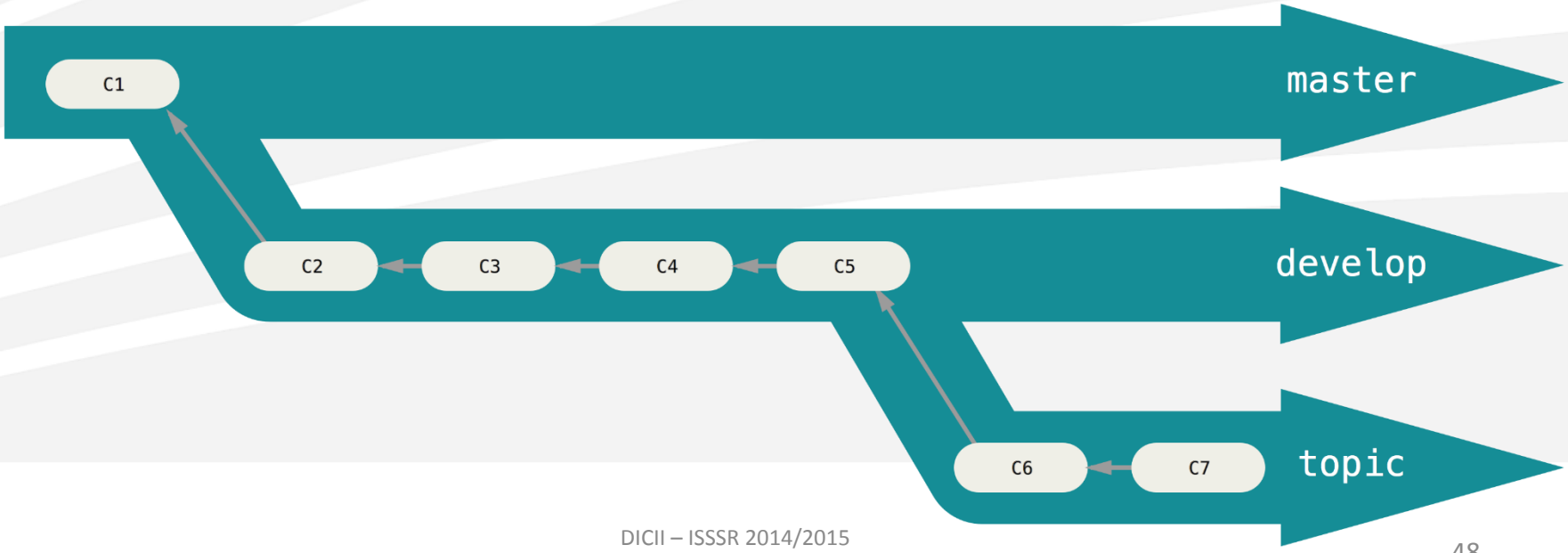
Branch Rebase

- *Warning: do not rebase commits that have already been pushed to the upstream*



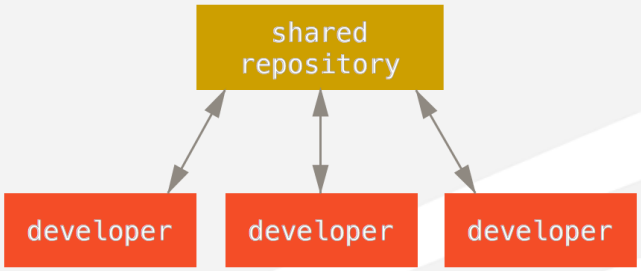
Examples of Branching Use

- 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

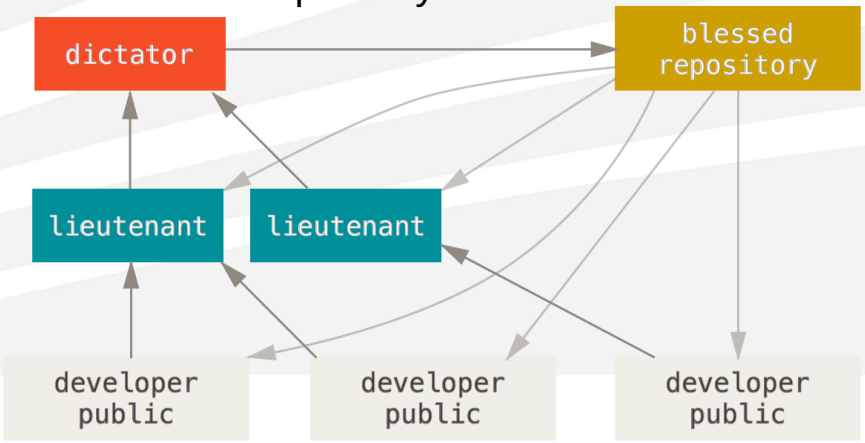


Distributed Workflows

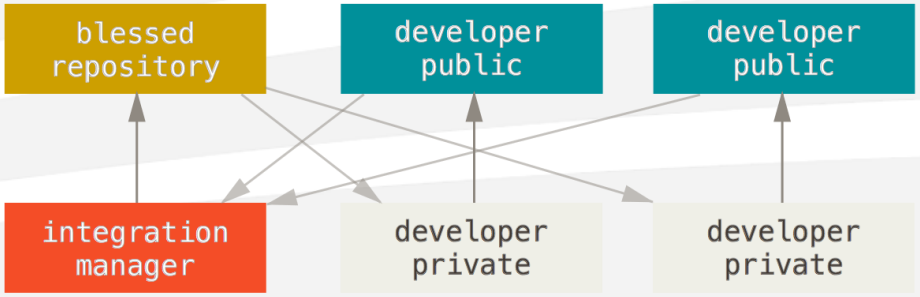
- Centralized workflow



- 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



- 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



References

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