Git introduction, course requirements and workflow

Kharkiv, 2019

Agenda

1. Introduction and motivation
2. About git
3. Main rules
4. How to implement these rules
5. Project workflow

What's a VCS?

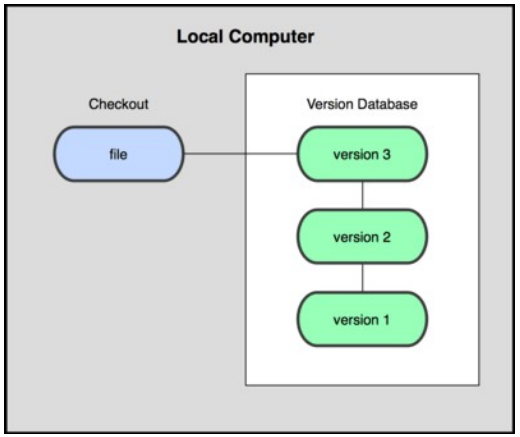
Version control is a system that records changes to a set of files over time so that you can recall specific versions later

Also usually it is used for

- Collaboration
- CI/CD interactions
- Analysis
- Backups and release management
- Primary source for effort tracking

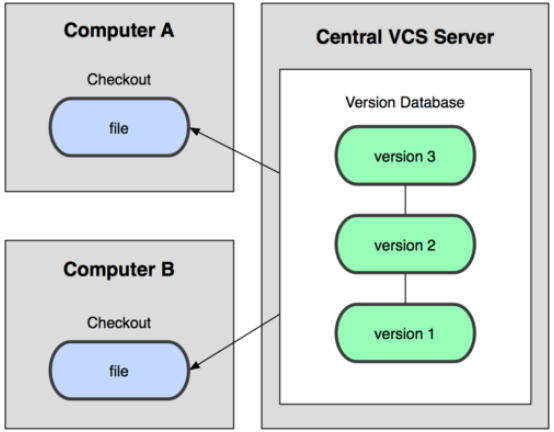
VCS Types

Local VCS



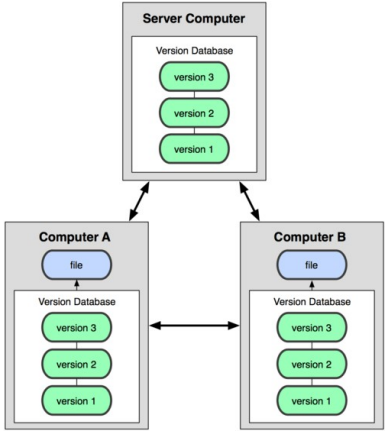
rcs, dropbox?

Centralized VCS



CVS, Subversion и Perforce

Distributed VCS



Git, Mercurial, Bazaar

Distributed VS Centralized VCS

Advantages of distributed VCS

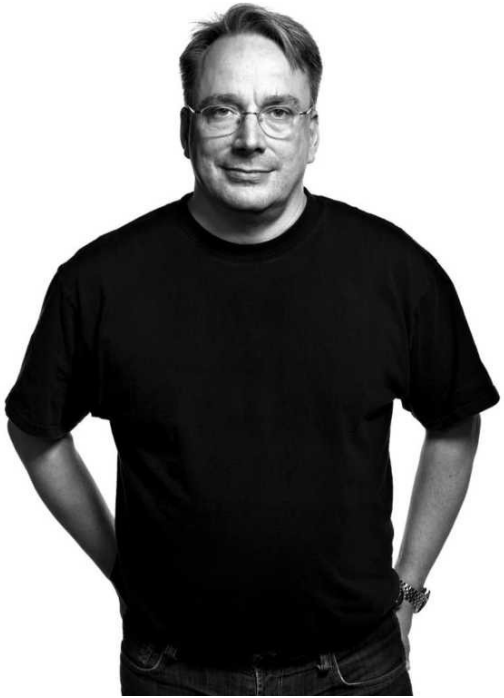
- Most of operations are local.
- Repository data and history available on each local copy, so you could do a lot of operation without internet connection.
- If central copy of data will be lost, any local copy could be used to restore central.
- Lightweight branching.
- Possibility of working with several remotes in one time.

Distributed VS Centralized VCS

Advantages of centralized VCS

- Storing only current copy of data in a local repository could be an advantage.
- Easier workflow for novice users.

LINUS TORVALDS



Creator of Linux and Git

- Nobody actually creates perfect code the first time around, except me. But there's only one of me.

- I'm an egotistical bastard, and I name all my projects after myself. First Linux, now git

What is git

- Git is a distributed version control system.
- Git is used in many commercial and open source projects like Linux Kernel, OpenGL Mesa library, OpenCV etc.
- There are many web-based hosting services for git like github, gitlab, bitbucket.

How to install git

Via binary installer (RPM based distro):

```
$ sudo yum install git-all
```

If you 're on a Debian-based distribution like Ubuntu, Debian

```
$ sudo apt-get install git-all
```

Windows

```
http://git-scm.com/download/win
```

Mac OS using brew

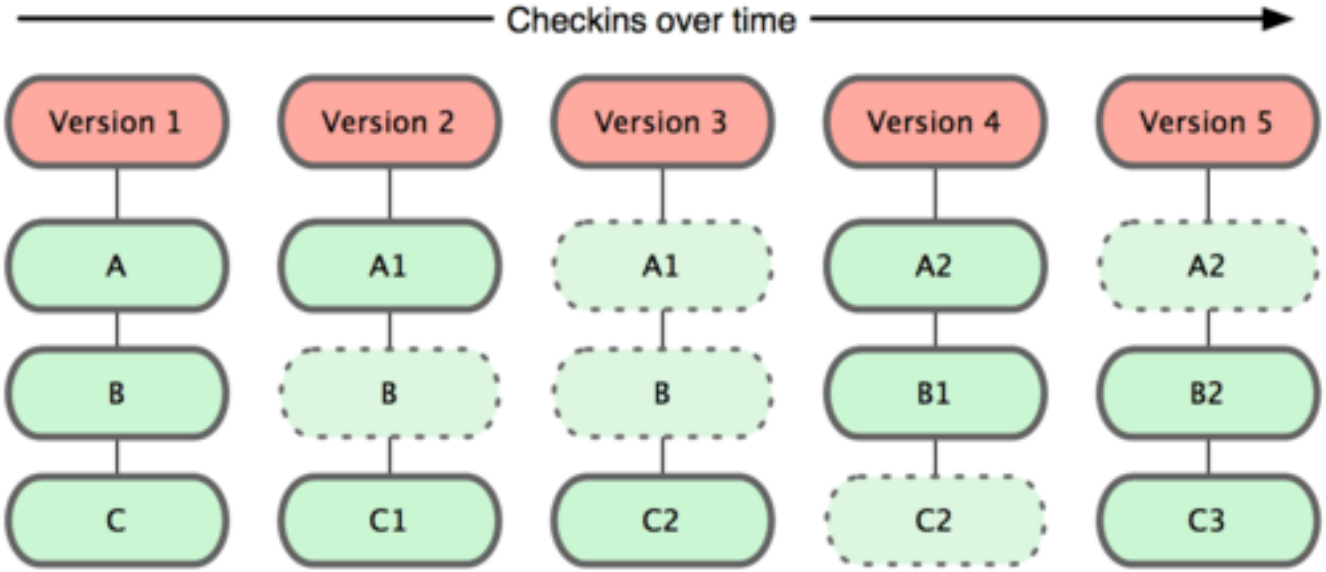
```
$ brew install git
```

Or download package from

```
https://git-scm.com/download/mac
```

What's inside?

- Git thinks of its data more like a series of snapshots of a miniature filesystem. Git
- To be efficient, if files have not changed**, Git doesn’t store the file again**, just a link to the previous identical file it has already stored.
- Git thinks about its data more like a stream of snapshots



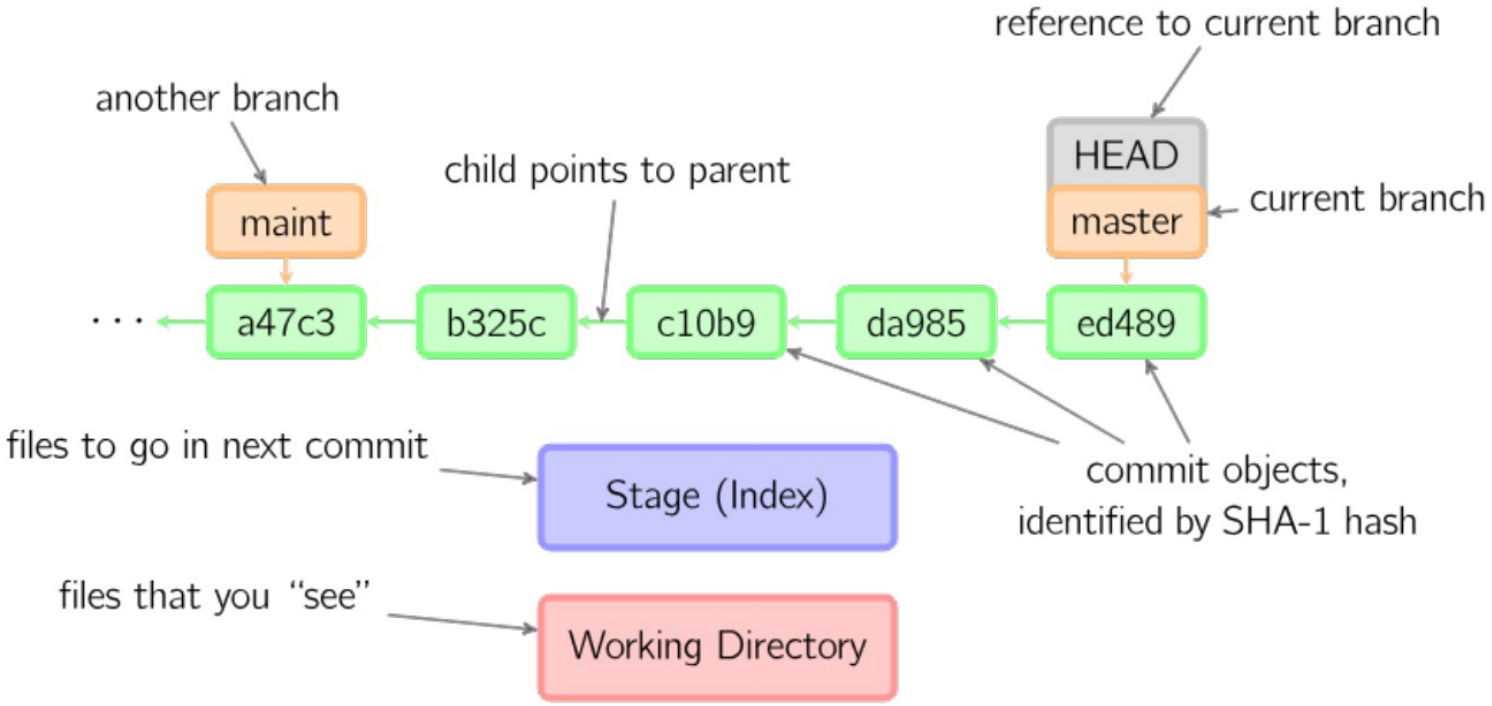
Git Terminology

- Repository / Repo: the .git filesystem which contains the project history and settings
- Staging area / index / cache: files (or 'hunks' of code) which will be committed
- Working Copy: the directory you're working in, may not be staged or committed yet
- Hash: A checksum which acts as a unique identifier for your commit. Also guarantees file integrity.
- Commit: create a snapshot or restore point, to which you can return in future (n: the snapshot itself)
- Checkout: sync your Working Copy with the selected commit
- Clone: to download a copy of a repository

Git Terminology

- Branch: an active line of development
- Head: a reference to the branch you're working on
- Master: the default development branch
- Merge: to integrate changes from another branch into the current
- Tag: a label which acts like a 'bookmark', generally used for tagging release versions.
- Rebase: funky merge... think cherry picks & hard reset: Using 'git cherry-pick' to Simulate 'git rebase'
- Fork: to make modifications to someone else's project
- Push: send changes to a remote server
- Fetch: receive changes from a remote server
- Pull: Fetch & merge in one operation

Git Terminology



Git hosting services

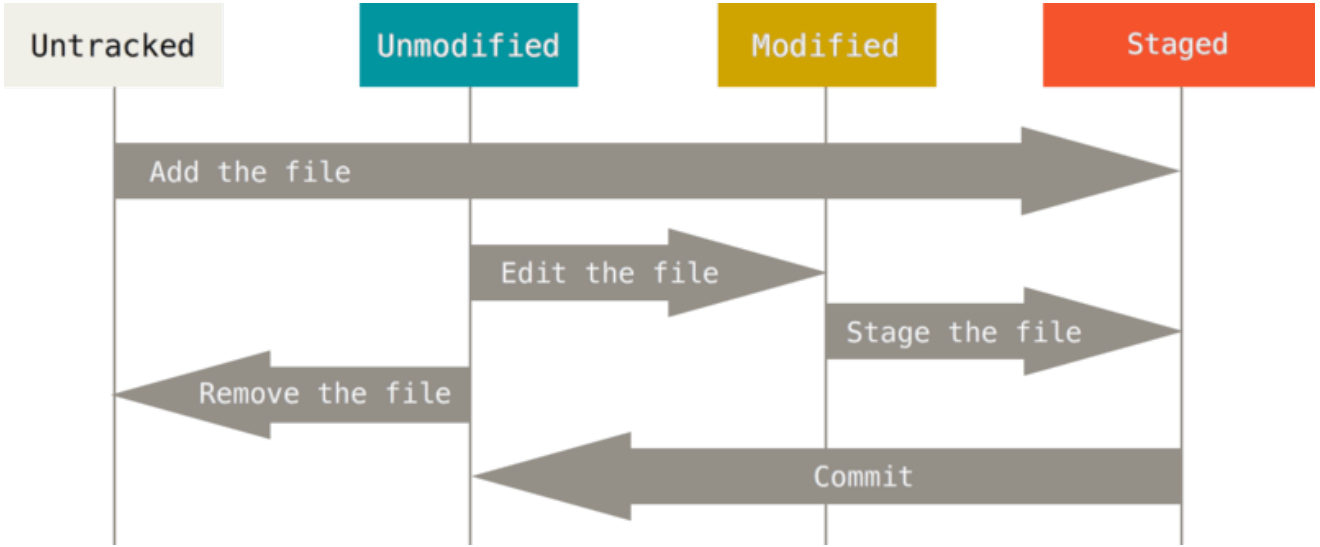
GitHub

GitLab

- Web GUI tools for team collaboration
- Code review tools Integrations with bug tracking systems
- Useful views for analytics

State lifecycle

- New (Untracked): file was recently created
- Modified: you have changed the file but have not committed it to your local database
- Staged: you have marked a modified file in its current version to go into your next commit snapshot.
- Committed (Unmodified): the data is safely stored in your local database



Creating git repository

Init

```
$ git init
$ git init <folder>
```

Clone

```
$ git clone <remote repository url>
```

Git configuration and help

```
$ git config name "value" # set property
$ git config --global # for current user
$ git config --system # for all users
$ git config --global user.name "Ivan Ivanov"
$ git config --list # config for current repo
$ git config --global core.editor
"'C:/Program Files (x86)/Notepad++/notepad+
+.exe' -multilnst -notabbar -nosession -
noPlugin“
```

For other setting and editor configuration loot at
https://swcarpentry.github.io/git-novice/02-setup/

Git help

```
$ git help <verb>
$ git <verb> --help
$ man git-<verb>
```

.gitignore

This is a file, which you could create in the root of your repository. All files, which are match patterns from gitignore, would be untracked by default. This could be binary files; files, which are generated by IDE, logs, ect. So all of this files exist in you project directory, but you will never want to commit them to repository. The rules for the patterns you can put in the .gitignore file are as follows:

- Blank lines or lines starting with # are ignored.
- Standard glob patterns work.
- You can start patterns with a forward slash (/) to avoid recursivity.
- You can end patterns with a forward slash (/) to specify a directory.
- You can negate a pattern by starting it with an exclamation point (!).

.gitignore

```
# no .a files
*.a
# but do track lib.a,
# even though you're ignoring .a files above
!lib.a
# ignore all files in the build/ directory
build/
# ignore all .pdf files in the doc/ directory
doc/**/*.pdf
```

Commit

Commit is basic unit which includes the change itself and its description. Another common name is patch.

Each commit is recognized using its **commit id**, it is a SHA256 checksum of the diff and the message.

Note that commit message has timestamp, so even applying the same change on the same base, but in different time, will result in different commit ids.

To create a commit you need to:

< make changes >

```
$ git diff

$ git add < files >

$ git commit

$ git show
```

Diff and show commands and needed to ensure that changes are valid. Commit rules will be described in next chapter.

Branch

A branch in Git is simply a movable pointer to one of the commits. Every time you commit, it moves forward automatically.

To create a branch you need to:

$ git checkout -b new_name <commit id>

* commit id may be either hash, local or remote branch name, HEAD, HEAD~1 etc.

Branches are usually created to implement some feature or fix. After done, branch needs to be combined with the main (master) branch.

This can be done using merge or rebase methods.

```
$ git checkout master

$ git merge feature


$ git rebase feature --onto master
```

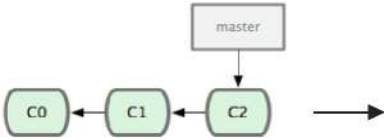
Useful resources

https://git-scm.com/doc

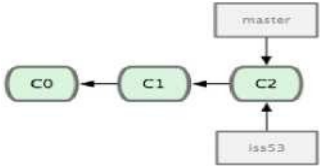
https://githowto.com/

https://kinsta.com/knowledgebase/what-is-github/

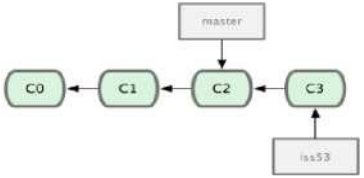
Branching & merging workflow



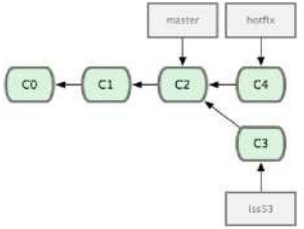

```
$ git checkout -b iss53
Switched to a new branch ’iss53’
```



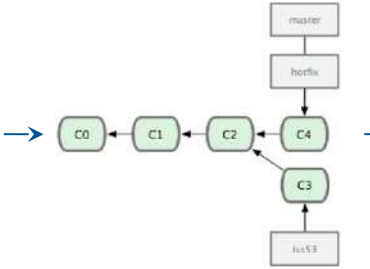

```
[working on iss53]
$ git commit -a -m “issue53 add footer”
```



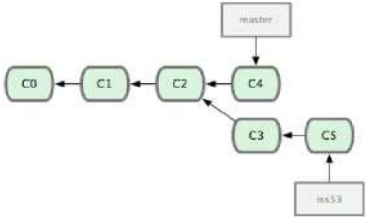

```
$ git checkout master
Switched to branch ’master
```



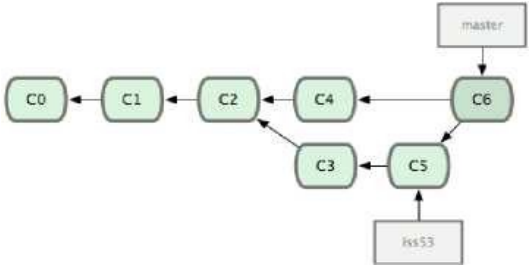

```
$ git checkout -b hotfix
Switched to a new branch ’hotfix‘
[do some fixes]
$ git commit -a -m “fix something”
```



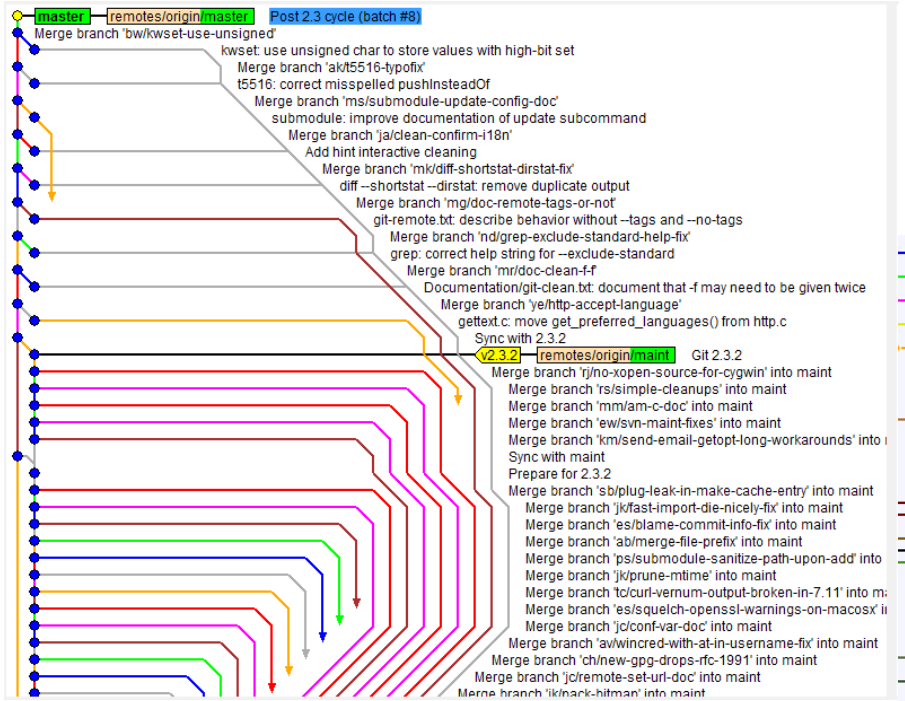

```
$ git branch -d hotfix
Deleted branch hotfix (was 3a0874c).
$ git checkout iss53 Switched to
branch ’iss53’
[Finish working on iss53]
$ git commit -a -m ’finish [issue
53]’
```

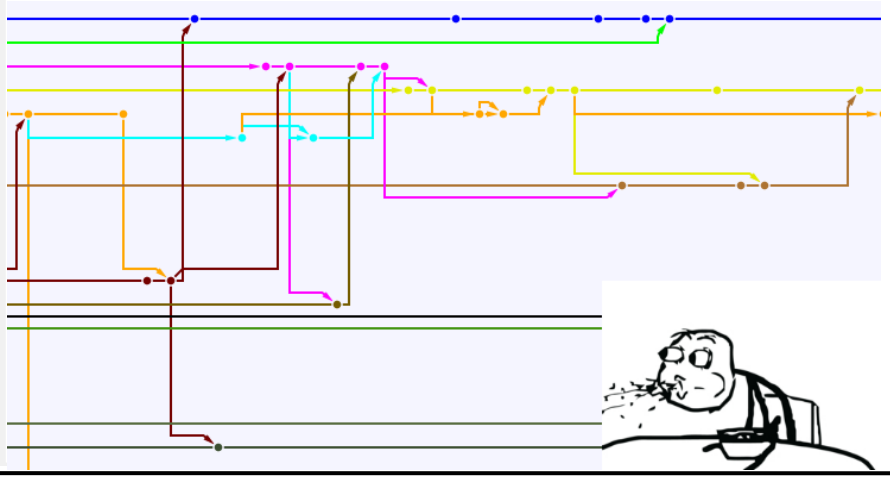



```
$ git checkout master
$ git merge hotfix
Updating f42c576..3a0874c
Fast-forward
```

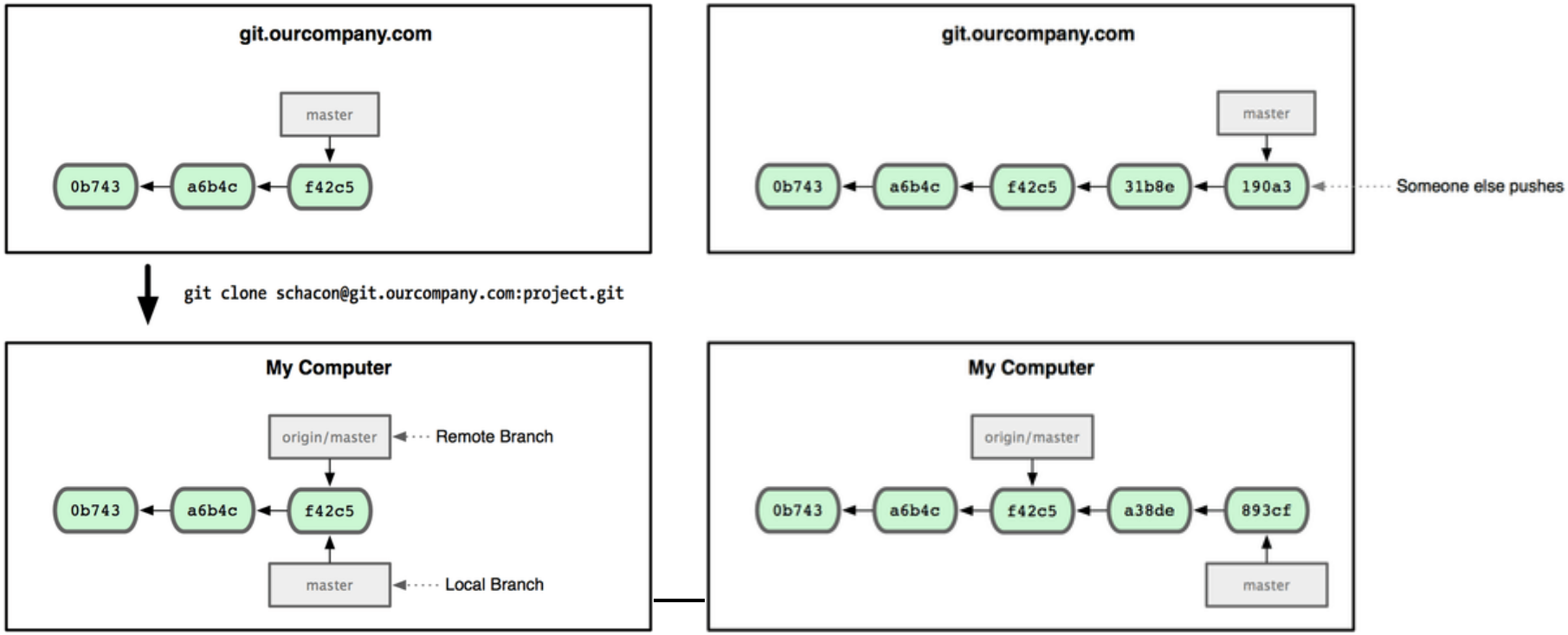


Merge hell

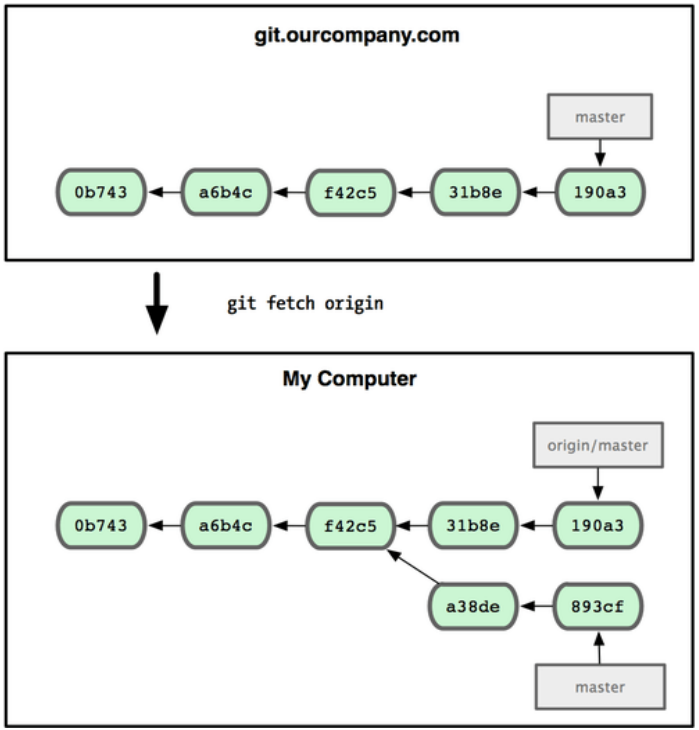

master
remotes/origin/master
Post 2.3 cycle (batch #8)
Merge branch 'bw/kwset-use-unsigned'
kwset: use unsigned char to store values with high-bit set
Merge branch 'ak/t5516-typofix'
t5516: correct misspelled pushInsteadOf
Merge branch 'ms/submodule-update-config-doc'
submodule: improve documentation of update subcommand
Merge branch 'ja/clean-confirm-i18n'
Add hint interactive cleaning
Merge branch 'mk/diff-shortstat-dirstat-fix'
diff --shortstat --dirstat: remove duplicate output
Merge branch 'mg/doc-remote-tags-or-not'
git-remote.txt: describe behavior without --tags and --no-tags
Merge branch 'nd/grep-exclude-standard-help-fix'
grep: correct help string for --exclude-standard
Merge branch 'mr/doc-clean-f-f'
Documentation/git-clean.txt: document that -f may need to be given twice
Merge branch 'ye/http-accept-language'
gettext.c: move get_preferred_languages() from http.c
Sync with 2.3.2
v2.3.2
remotes/origin/maint
Git 2.3.2
Merge branch 'rj/no-xopen-source-for-cygwin' into maint
Merge branch 'rs/simple-cleanups' into maint
Merge branch 'mm/am-c-doc' into maint
Merge branch 'ew/svn-maint-fixes' into maint
Merge branch 'km/send-email-getopt-long-workarounds' into
Sync with maint
Prepare for 2.3.2
Merge branch 'sb/plug-leak-in-make-cache-entry' into maint
Merge branch 'jk/fast-import-die-nicely-fix' into maint
Merge branch 'es/blame-commit-info-fix' into maint
Merge branch 'ab/merge-file-prefix' into maint
Merge branch 'ps/submodule-sanitize-path-upon-add' into
Merge branch 'jk/prune-mtime' into maint
Merge branch 'tc/curl-vernum-output-broken-in-7.11' into m
Merge branch 'es/squelch-openssl-warnings-on-macosx' i
Merge branch 'jc/conf-var-doc' into maint
Merge branch 'av/wincred-with-at-in-username-fix' into maint
Merge branch 'ch/new-gpg-drops-rfc-1991' into maint
Merge branch 'jc/remote-set-url-doc' into maint
Merge branch 'jk/pack-bitmap' into maint



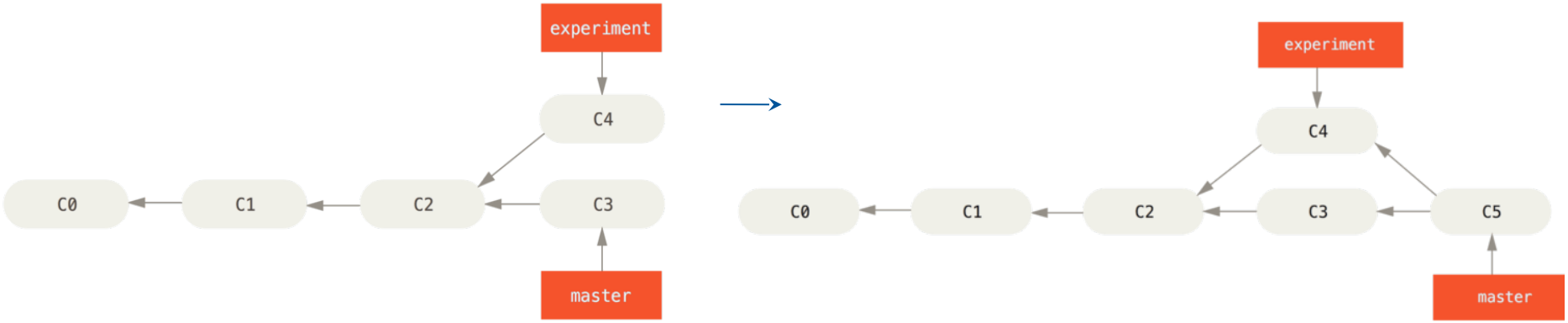
Remote and local branches



Remote and local branches

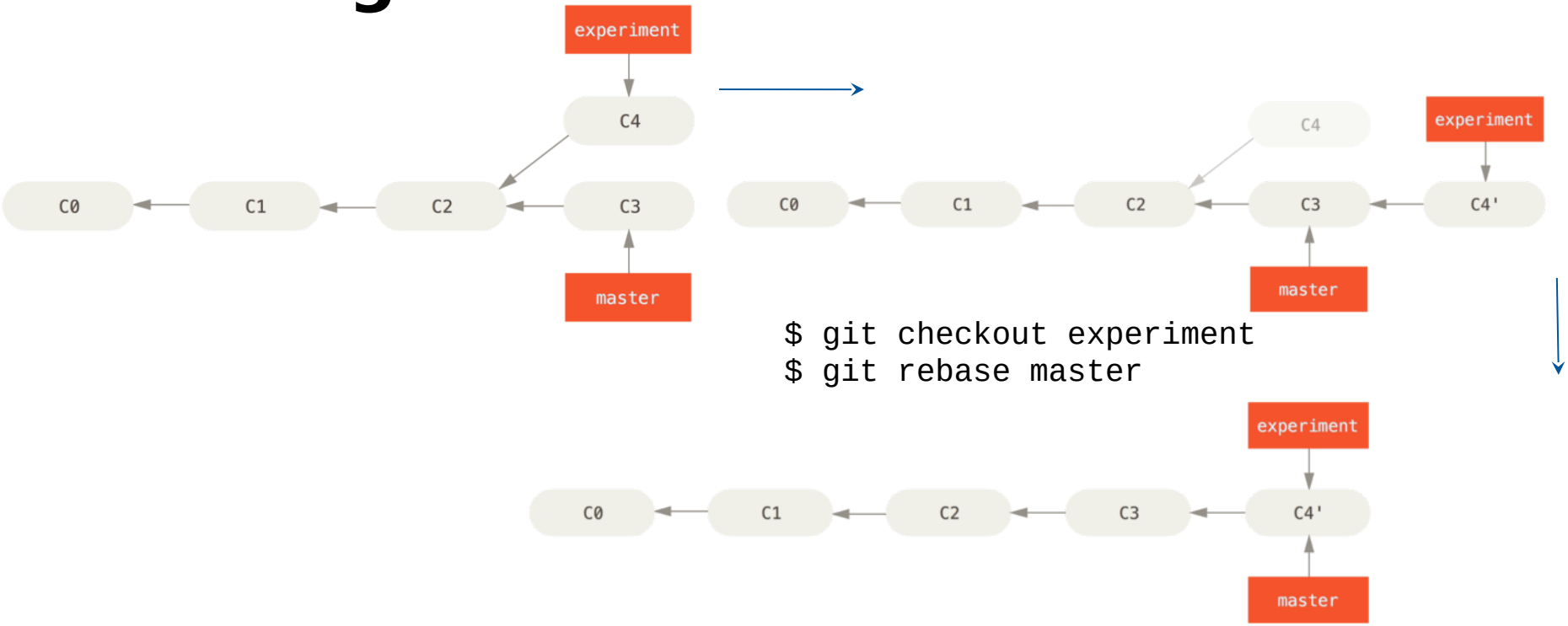


Merge vs Rebase




```
$ git checkout master
$ git merge experiment
```

Rebasing


experiment
C4
C0
C1
C2
C3
master
C4
experiment
C0
C1
C2
C3
C4'
master
$ git checkout experiment
$ git rebase master
experiment
C0
C1
C2
C3
C4'
master

2. Main rules

Every step (commit, branch, pull request etc) in the work process should follow the code style and processual rules.

This is mandatory for the fast and smooth workflow.

Commit requirements

- Use signature when committing, this will be automatically done by the $ git commit -s (name needs to be set in ~/.gitconfig).
- No more than 72 characters on line (use your signature string as a basic ruler).
- Empty line between title and description and between description and signature.

Commit title should be like:

Topic: <Action> <the rest title text>

- Topic: describes the area of changes, e.g. Lesson1: or LCD:
- Action: Add, Fix, Implement … (not added, implemented). Should start from capital letter.
- The whole title should give enough info about the commit when using $ git log --oneline.
- No dot at the end of the title.

Commit description should describe why and how it was done, and not what was done (this is visible in diff).

For example, a bad description: Changed short to int.

The correct one should be: Moved to a bigger data type because of overflow in some corner cases.

Description is not needed in the “title says it all” cases.

- Do not mix several different actions in one commit: split them in several. For example, code style fixes and the change itself.
- Always check your code for code style compliance. In this project we use a **kernel style**.
- Branch at each commit should be buildable and working. Do not add constants (header) and their usage (code) in different commits.
- **Do not create patches to unmerged patches**, fix the original ones.

Always check your code before publishing!

The possible ways:

1. git show
2. Setup formatting in IDE/editor
3. linux/scripts/checkpatch.pl
4. cppcheck

2.2 Branch requirements

- Always create branches even for single-commit changes.
- Local branch names are up to you, but global names should be meaningful.

These both are needed for pull requests.

2.3. Pull request requirements

Pull requests are used to notify others that you pushed some changes (a branch) to your github repository.

Each pull request has topic and description, it should follow the same rules as for commit (except 72 characters of cause, the markdown syntax gives much more features).

3. How to implement these rules

Of cause by following them initially, but...

The main problem comes from the need to change a commit after testing or review (remember, no patches to unmerged patches).

There are two cases there:

- Modify only last commit;
- Modify commit (s) in the middle.

How to change the last commit

1. Do the changes
2. $ git add files (Don’t forget!)
3. $ git commit --amend

To only change the title or message, use only

$ git commit --amend

How to change commit in the middle (straightforward way)

Let’s assume that branch “featureX” contains the following commits:

55555 xxx: Commit 5
44444 xxx: Commit 4
33333 xxx: Commit 3
22222 xxx: Commit 2
11111 xxx: Commit 1
And it is needed to make changes to the commit 33333

$ git checkout -b temp1 33333

Do the changes
$ git add
$ git commit --amend

$ git checkout featureX

$ git rebase 33333 --onto temp1

Why 33333 and not 44444: we rebasing state after 33333 so the first commit applied will be 44444.

The same operations are used for inserting commits.

Interactive rebase is a very powerful tool

git rebase -i HEAD~10

- You will see the table with the order how commits will be applied.
- You may change the order by simply moving the lines.
- You can change the action with each commit: reword (change the description without touching changes), edit or squash (combine several commits into one).
- Add --root to include first initial commit.


In case of fire
1. git commit
2. git push
3. leave building

Thank you!