CS6

Practical System Skills

Fall 2019 edition

Leonhard Spiegelberg
lspiegel@cs.brown.edu

Recap

Last lecture: Version Control Systems ⇒ git

⇒ 3 areas

⇒ creating commits, checking out old commits

⇒ working with remotes

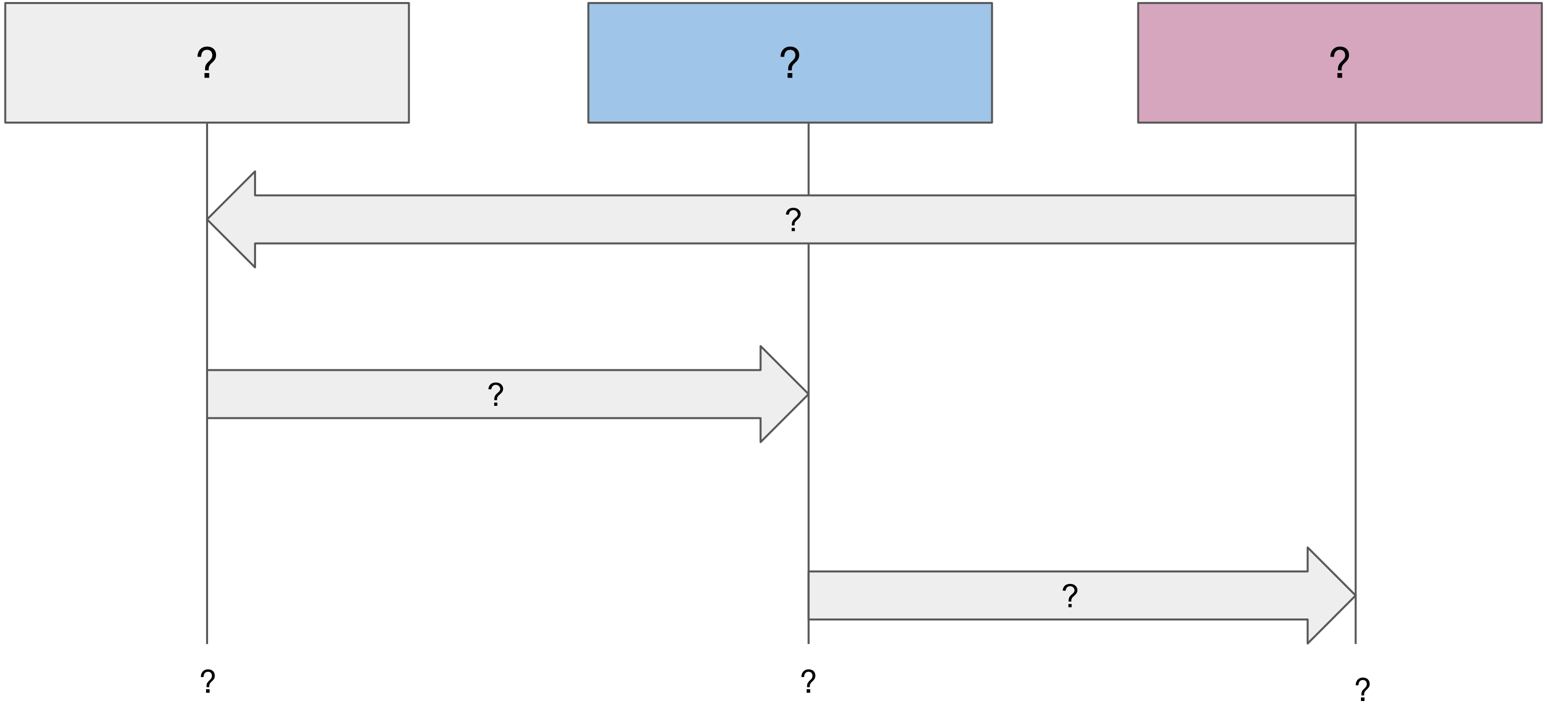
⇒ pull/push

⇒ working with branches

⇒ merging branches via git merge

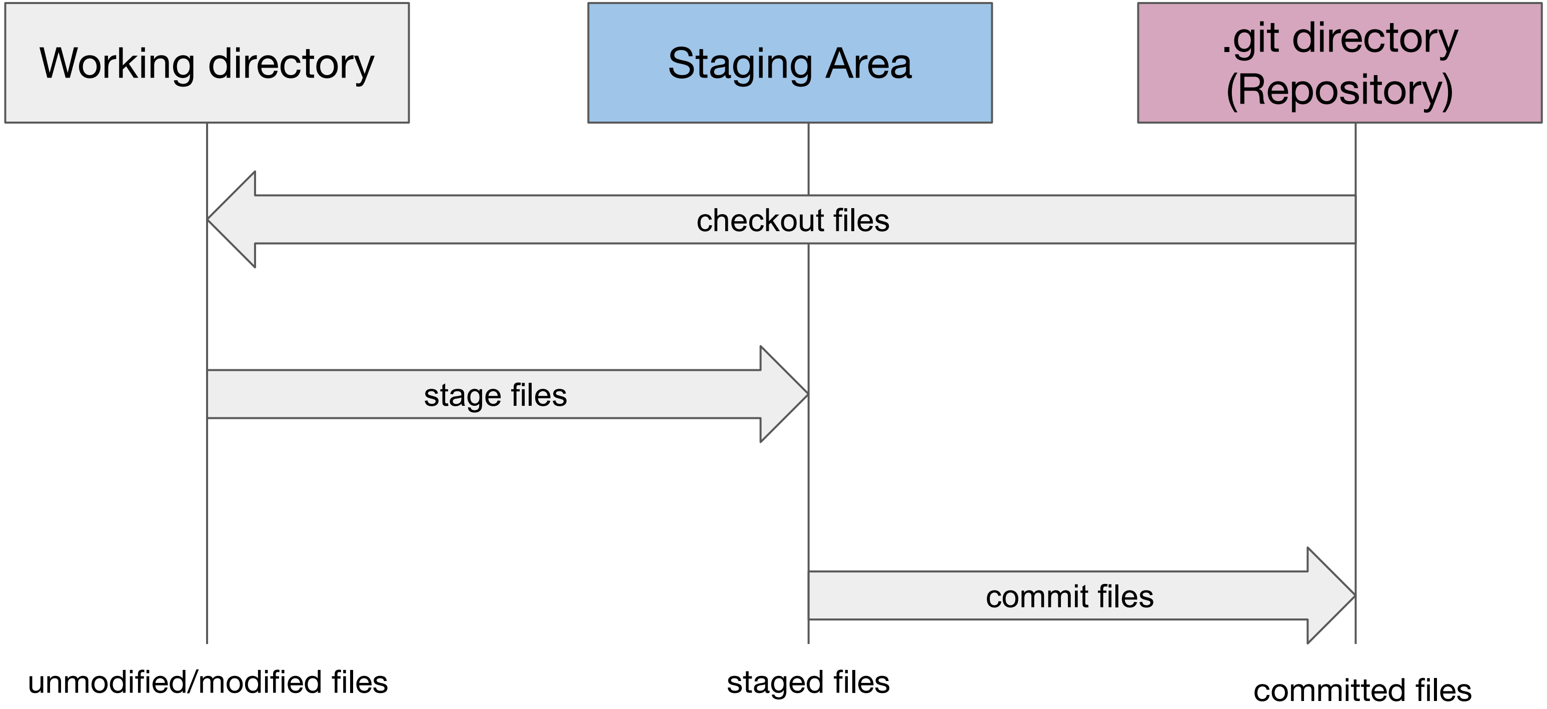
Recap quiz

Fill out the following graphic:



Recap quiz

Fill out the following graphic:



15 More on Git

CS6 Practical System Skills

Fall 2019

Leonhard Spiegelberg *lspiegel@cs.brown.edu*

Master and feature branches

⇒ Typically, there's a master branch in the repo

→ don't use it for development, rather store "releasable" version of your code/assets on it

→ content on the master branch should work, i.e. no errors.

⇒ DON'T BREAK THE MASTER!

⇒ In a couple slides: Typical git workflows.

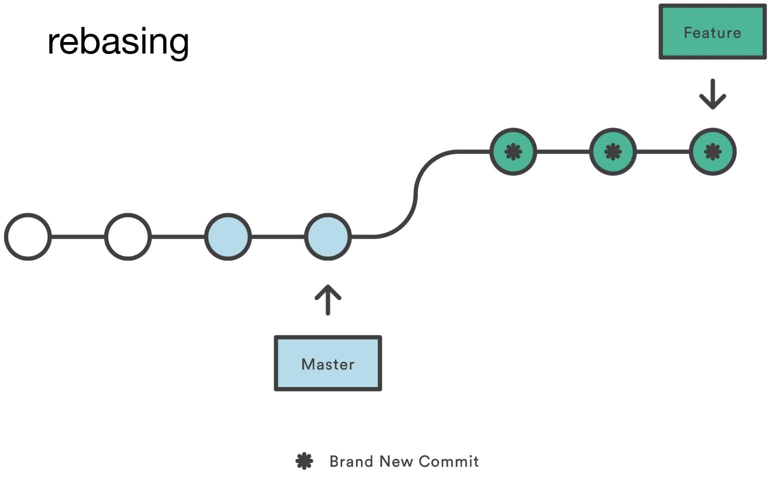
Rebasing

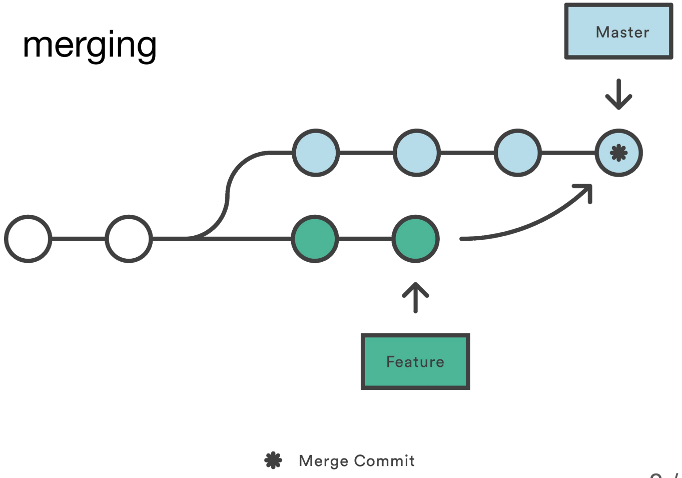
15.01 git merge vs. git rebase

⇒ To join branches, `git rebase` is an alternative to `git merge`

⇒ If you don't know how to rebase properly, things can go wrong badly

Recap:



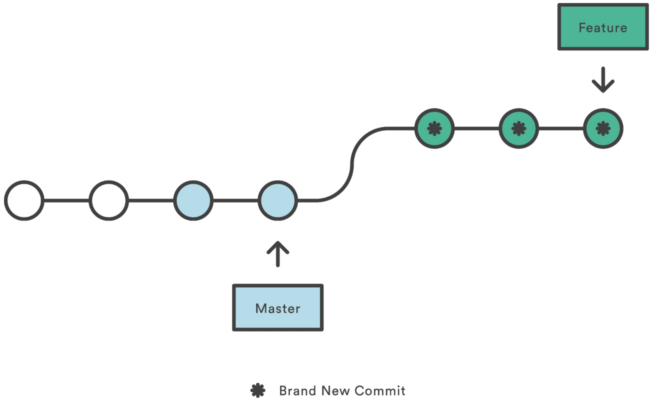


15.02 git rebase

⇒ to rebase a branch on another, run `git rebase`. Assuming you're on branch feature, then you can rebase onto the master via:

```
git rebase master
```

branch on which to base on



rebasing feature on master general syntax:

```
git rebase master feature
```

(will checkout feature)

15.03 Golden rule of rebasing

⇒ **NEVER, NEVER, NEVER** rebase the master onto a feature branch.

→ Only rebase `feature` on `master` OR

→ `featureA` on `featureB`

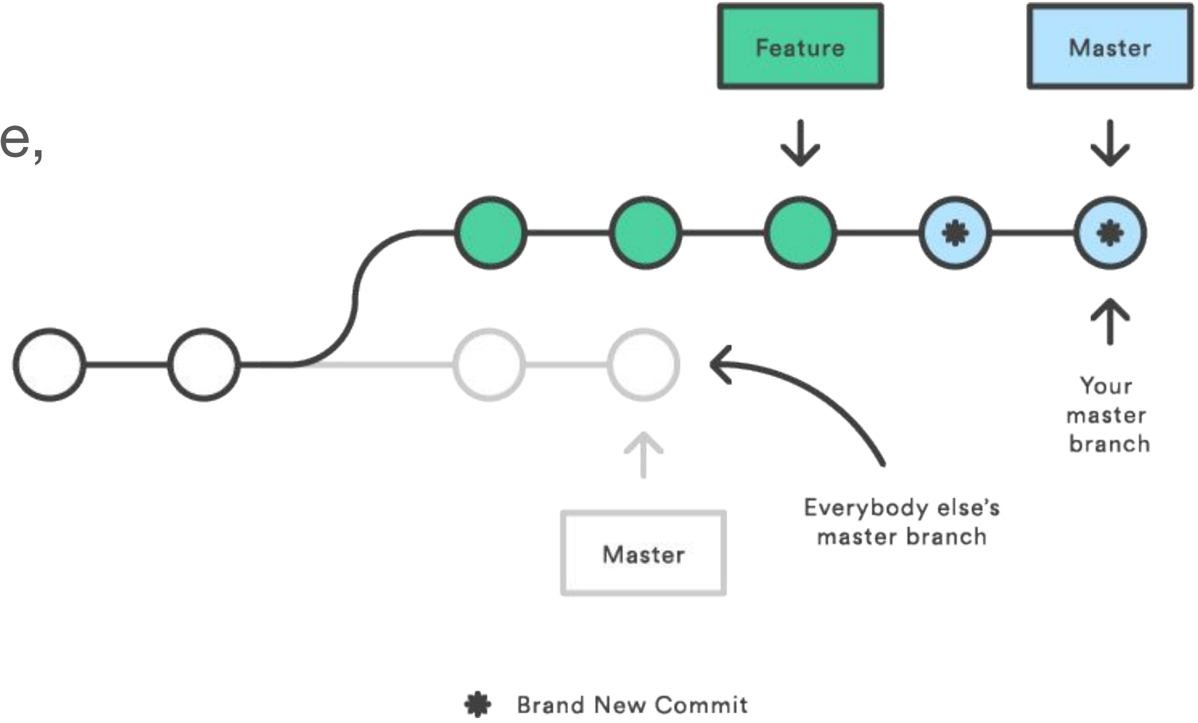
I.e., don't run the following commands:

~~`git checkout master && git rebase feature`~~

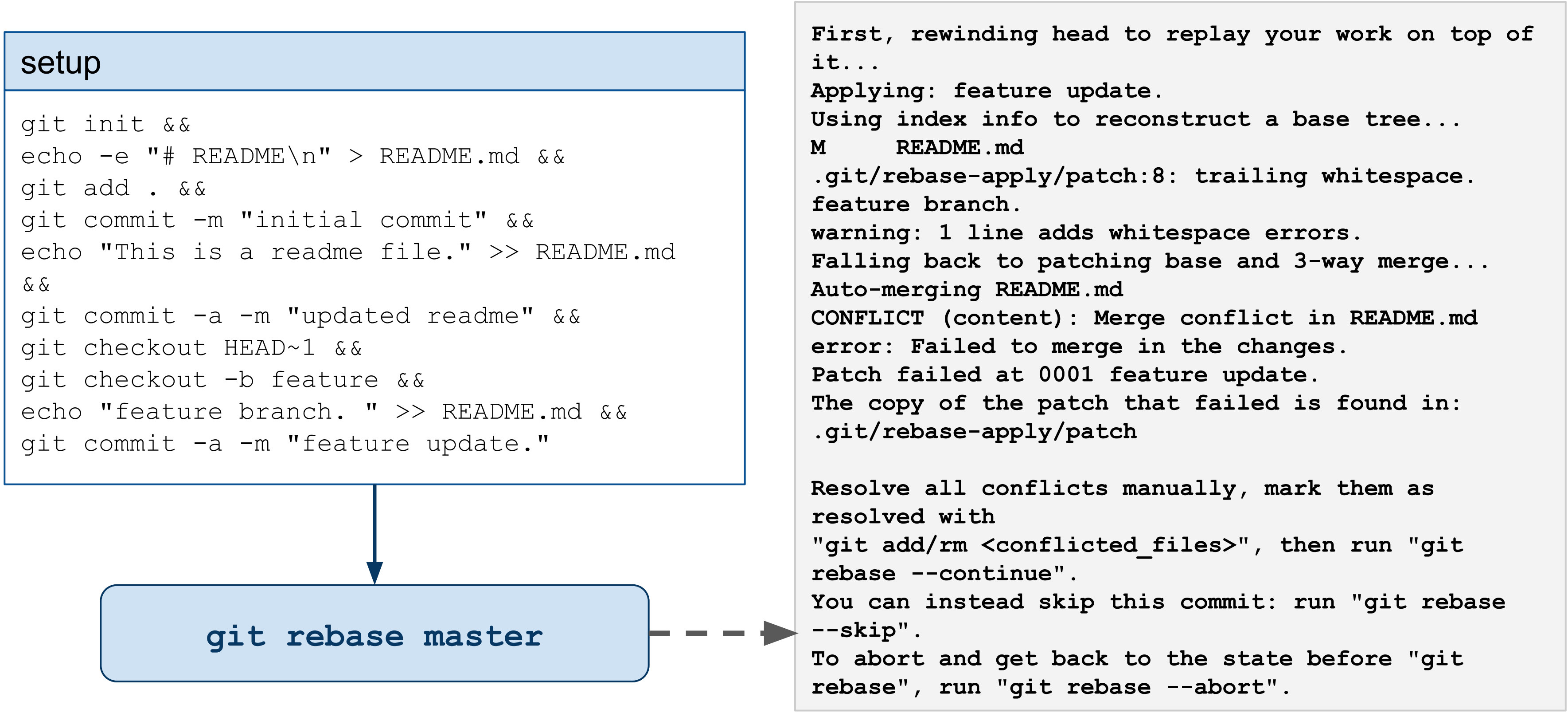
~~`git rebase feature master`~~

15.03 Golden rule of rebasing

⇒ If you rebased the master on your feature, you would create a confusing history.



15.04 Rebase example



setup

```
git init &&
echo -e "# README\n" > README.md &&
git add . &&
git commit -m "initial commit" &&
echo "This is a readme file." >> README.md
&&
git commit -a -m "updated readme" &&
git checkout HEAD~1 &&
git checkout -b feature &&
echo "feature branch. " >> README.md &&
git commit -a -m "feature update."
```

```
git rebase master
```

```
First, rewinding head to replay your work on top of
it...
Applying: feature update.
Using index info to reconstruct a base tree...
M       README.md
.git/rebase-apply/patch:8: trailing whitespace.
feature branch.
warning: 1 line adds whitespace errors.
Falling back to patching base and 3-way merge...
Auto-merging README.md
CONFLICT (content): Merge conflict in README.md
error: Failed to merge in the changes.
Patch failed at 0001 feature update.
The copy of the patch that failed is found in:
.git/rebase-apply/patch

Resolve all conflicts manually, mark them as
resolved with
"git add/rm <conflicted_files>", then run "git
rebase --continue".
You can instead skip this commit: run "git rebase
--skip".
To abort and get back to the state before "git
rebase", run "git rebase --abort".
```

15.05 Resolving conflicts in rebase

⇒ resolve conflicts on individual files using the 3 options:

1.) `git checkout --ours`
2.) `git checkout --theirs`
3.) manual merge

⇒ add resolved files via git add. (you can also edit the commit message via `git commit --amend`)

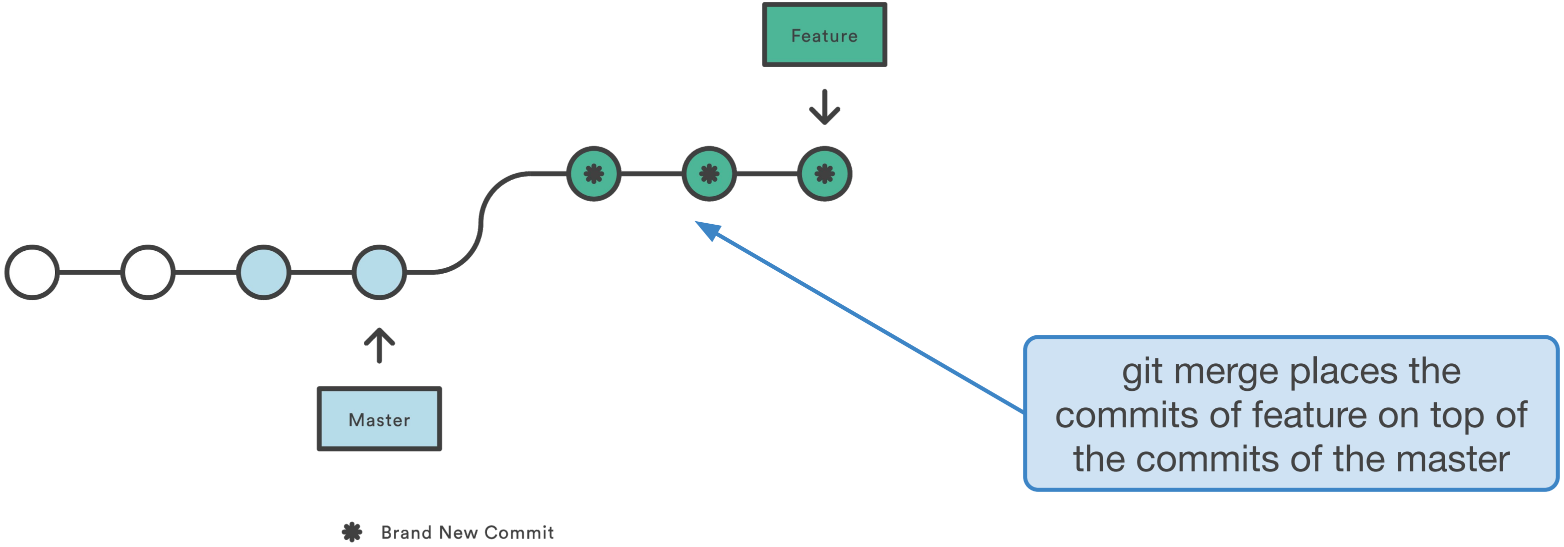
⇒ after conflict resolution run `git rebase --continue`

⇒ `git rebase --abort` stops the rebase!

15.06 Completing the rebase of feature on master

⇒ After rebasing on the master, feature branch can be "cleanly" merged to master (i.e. a fast-forward merge)

⇒ `git checkout master && git merge feature`



git merge places the commits of feature on top of the commits of the master

More on rebasing

15.07 interactive rebase & squashing commits

⇒ often you work on a separate branch but don't want to put all commits on the master or only a subset

⇒ `git rebase -i` starts rebase in interactive mode, which allows for more efficient history manipulation.

⇒ interactive mode allows to squash multiple commits into one (first commit must be pick / p though)

Tip:
Use `git config rebase.abbreviateCommands true` to force git to use shortcuts only instead of pick, squash, ...

More on conflict resolution

15.08 Visual merge tools

⇒ manual conflict resolution can be done in the console or via an IDE (most IDEs provide built-in support for merging)

⇒ there are many visual merge tools available, e.g.

- vimdiff
- meld
- GitKraken

⇒ to start merging via a tool, run `git mergetool`

→ configure it via `git config merge.tool meld`

→ per default, git creates .orig backup files. Disable via `git config mergetool.keepBackup false`

Stashing

15.09 Stashing

⇒ sometimes you work on a branch and have to switch to another one, but you don't feel like committing yet.

→ `git stash` saves changes away onto a temporary stack and reverts your local working copy

⇒ use `git stash` to save local changes

⇒ `git stash pop` to apply previously stashed changes

15.09 Stashing

⇒ to keep changes in stash and to apply them, use `git stash apply`

- → Can be used to apply changes to multiple branches

⇒ `git stash list` displays overview of stashed changes

- → pop n-th stash via `git stash pop stash@{n}`
- → remove n-th stash via `git stash drop stash@{n}`
- → clear stash via `git stash clear`

⇒ you can add a note to a stash, by using `git stash save "note"`

15.10 Cleaning the repo

⇒ sometimes you accumulate a lot of temporary / ignored files in your repository.

→ `git clean -n` to list what files would be removed

→ `git clean -f` to remove untracked files

→ `git clean -xf` to remove untracked and ignored files

15.11 Discarding local changes

⇒ to discard ALL local changes (no undo for this), you can use

```
git reset --hard
```

⇒ You can also use `git reset` to reset the HEAD to a specific commit, DO THIS ONLY if you haven't pushed to a remote yet.

→ don't screw up the remote
→ if you use `git reset` frequently, there's something wrong.

Tags

15.12 Tags

⇒ Last lecture: checkout specific commits via their SHA-1 hash

→ creates detached head

⇒ Often you want to release your software to the public at specific commits

→ tags provide a option to "tag" or mark a commit

⇒ List available tags via `git tag`

→ you can search for tags using a regex via `git tag -l "<regex>"`

15.12 Creating tags

⇒ There are two types of tags:

1) lightweight 2) annotated

⇒ lightweight tags are just a reference to a commit (i.e., the checksum)

→ use `git tag <tagname>` to create a lightweight tag

→ you need to explicitly push a tag to a remote via

```
git push origin <tagname>
```

→ checkout a tag via `git checkout <tagname>`

15.12 Creating tags

⇒ to create an annotated tag (with a message) use

```
git tag -a <tagname> -m "tag message"
```

⇒ to retrieve tag info, use `git show <tagname>`

⇒ push tag via `git push origin <tagname>`

more on tags: https://git-scm.com/book/en/v2/Git-Basics-Tagging

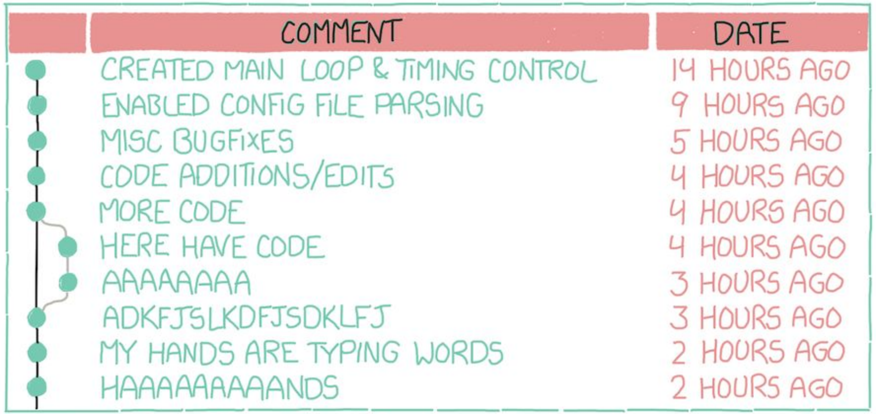
Commit messages

15.13 How to write good commit messages

⇒ writing good commit messages is an art for itself

⇒ Try to make them informative and to keep track of changes

⇒ If you make a pull request or push onto a public branch, have **clean & clear commit messages**



COMMENT	DATE
CREATED MAIN LOOP & TIMING CONTROL	14 HOURS AGO
ENABLED CONFIG FILE PARSING	9 HOURS AGO
MISC BUGFIXES	5 HOURS AGO
CODE ADDITIONS/EDITS	4 HOURS AGO
MORE CODE	4 HOURS AGO
HERE HAVE CODE	4 HOURS AGO
AAAAAAAA	3 HOURS AGO
ADKFJSLKDFJSDKLFJ	3 HOURS AGO
MY HANDS ARE TYPING WORDS	2 HOURS AGO
HAAAAAAAANDS	2 HOURS AGO

AS A PROJECT DRAGS ON, MY GIT COMMIT MESSAGES GET LESS AND LESS INFORMATIVE.

15.13 How to write good commit messages

bad examples	good examples

`add cli new`
`fixes`
`fix code review comments`
`no message`
`description`
`wip`
`hackz`
`little edit`
	`Fix error when the URL is not reachable`
	`Add error message for file not found`
	`Add server fingerprint check`
	`Fix shadow box closing problem`

15.13 How to write good commit messages

⇒ write in imperative mode: If commit is applied, <your message>

⇒ write a short subject line of a maximum of 50-72 chars and capitalize first word e.g.

```
Fix float casting bug in compiler
```

⇒ add whitespace line, then details of your commit.

⇒ Don't explain how it was done, but instead **what** and **why**

More info: https://medium.com/@cvortmann/what-makes-a-good-commit-message-995d23687ad

15.13 How to write good commit messages

```
# 50-character subject line
#
# 72-character wrapped longer description. This should answer:
#
# * Why was this change necessary?
# * How does it address the problem?
# * Are there any side effects?
#
# Include a link to the ticket, if any.
#
# Add co-authors if you worked on this code with others:
#
# Co-authored-by: Full Name <email@example.com>
# Co-authored-by: Full Name <email@example.com>
```

template from https://thoughtbot.com/blog/write-good-commit-messages-by-blaming-others

Git workflows

15.14 Common git workflows

⇒ There are several collaboration models or workflows used in (software) engineering teams:

→ central part of them is a pull request

→ most repository management systems like github/bitbucket/gitlab/… provide support for pull requests/reviews/...

Following slides are based on: https://www.slideshare.net/psquy/git-collaboration

15.14 Pull request

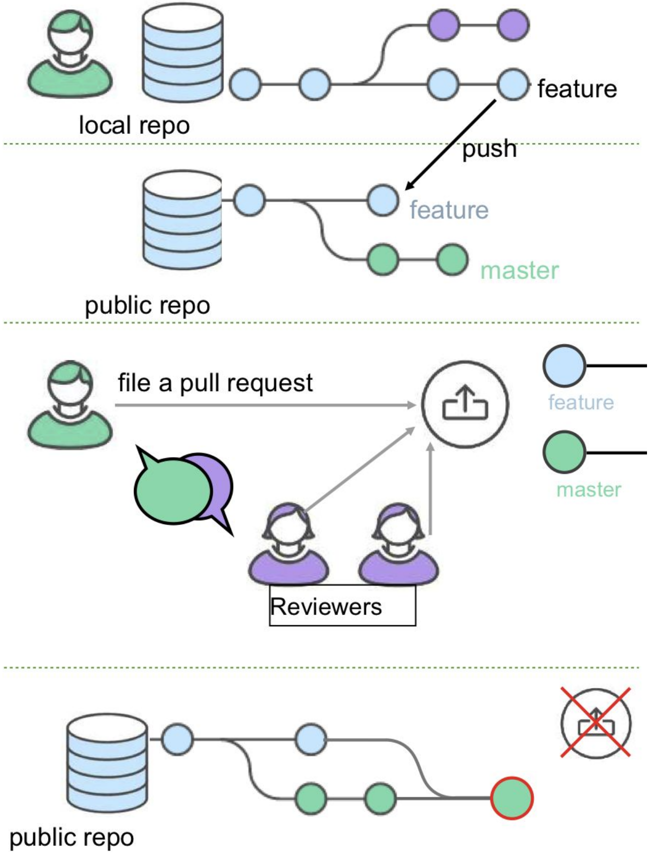
(1) Create feature on dedicated branch in local repo

(2) Push branch to public repository/remote

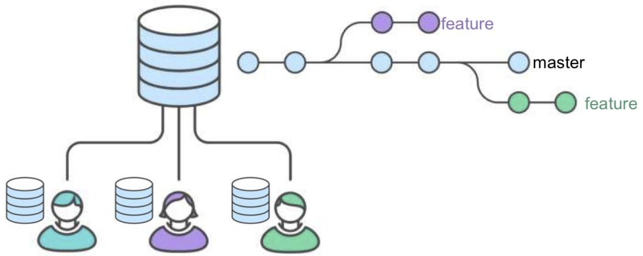
(3) file pull request to official repository

(4) other developers review code, discuss it, update it

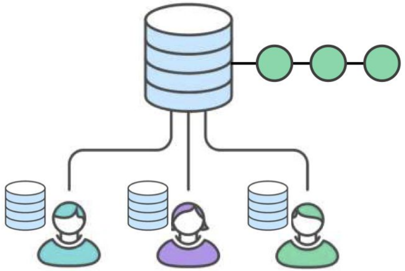
(5) project maintainer merges feature into official repository and closes the pull request



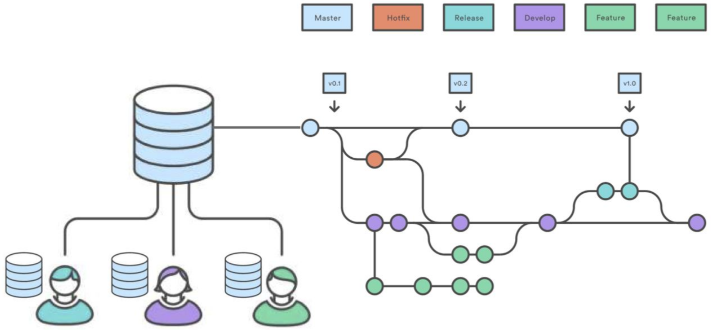
15.15 Four standard git workflows



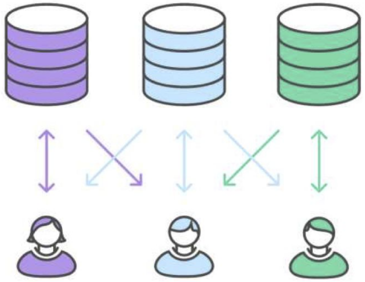
Feature Branch Workflow



Centralized Workflow



Gitflow Workflow

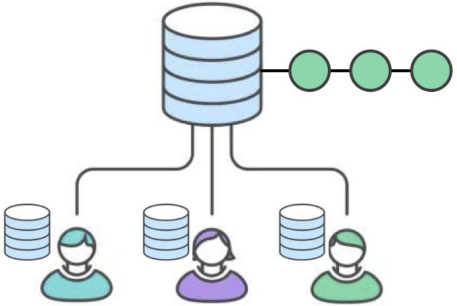


Forking Workflow

15.16 Centralized workflow

⇒ one master branch on which everybody works

Pro	Con
+ simple flow + good for not-frequently updated/changed projects	- more conflicts when many developers work together - no review or feature pull requests allowed - no branching - everybody works on the same branch - high chance for dirty master/problems

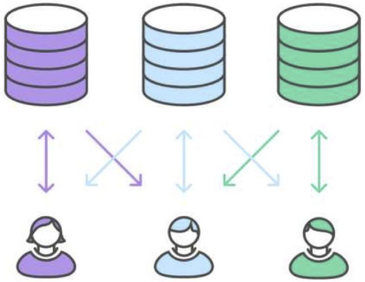


Centralized Workflow

15.17 Forking workflow

⇒ everybody forks the official repo, changes are added using pull request to the official repo

Pro	Con
+ standard used for open-source projects + allows to incorporate changes into "read" only repos, i.e. not everybody needs push access + less "code conflict" friction	- slower, because they require maintainer to incorporate changes



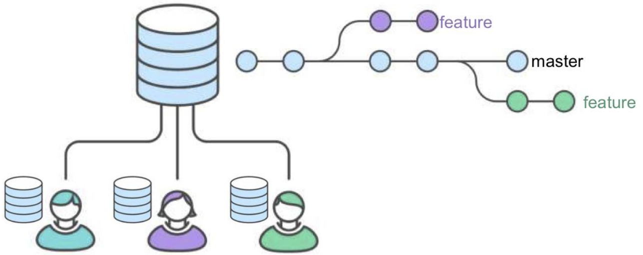
Forking Workflow

Note: a forked repo is a "server-side" cloned repo

15.18 Feature branch workflow

⇒ best for small teams. Have 1-2 senior engineers who merge in pull requests

⇒ Each developer creates for a feature a separate branch and makes a pull request



Feature Branch Workflow

Pro	Con
+ master branch not disturbed by development + pull requests/reviews + easy to manage + good for internal projects	- develop vs. production? - feature vs. hotfix? - release tracking? - dirty master branch?

15.19 Gitflow workflow

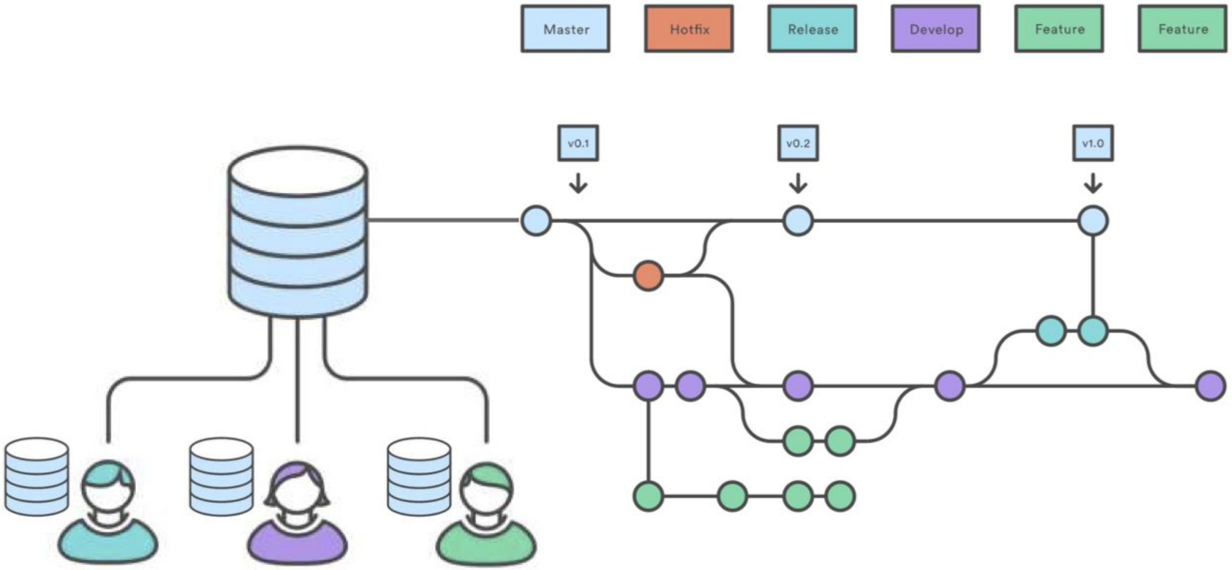
One master branch

One develop branch

One temporary branch for each release

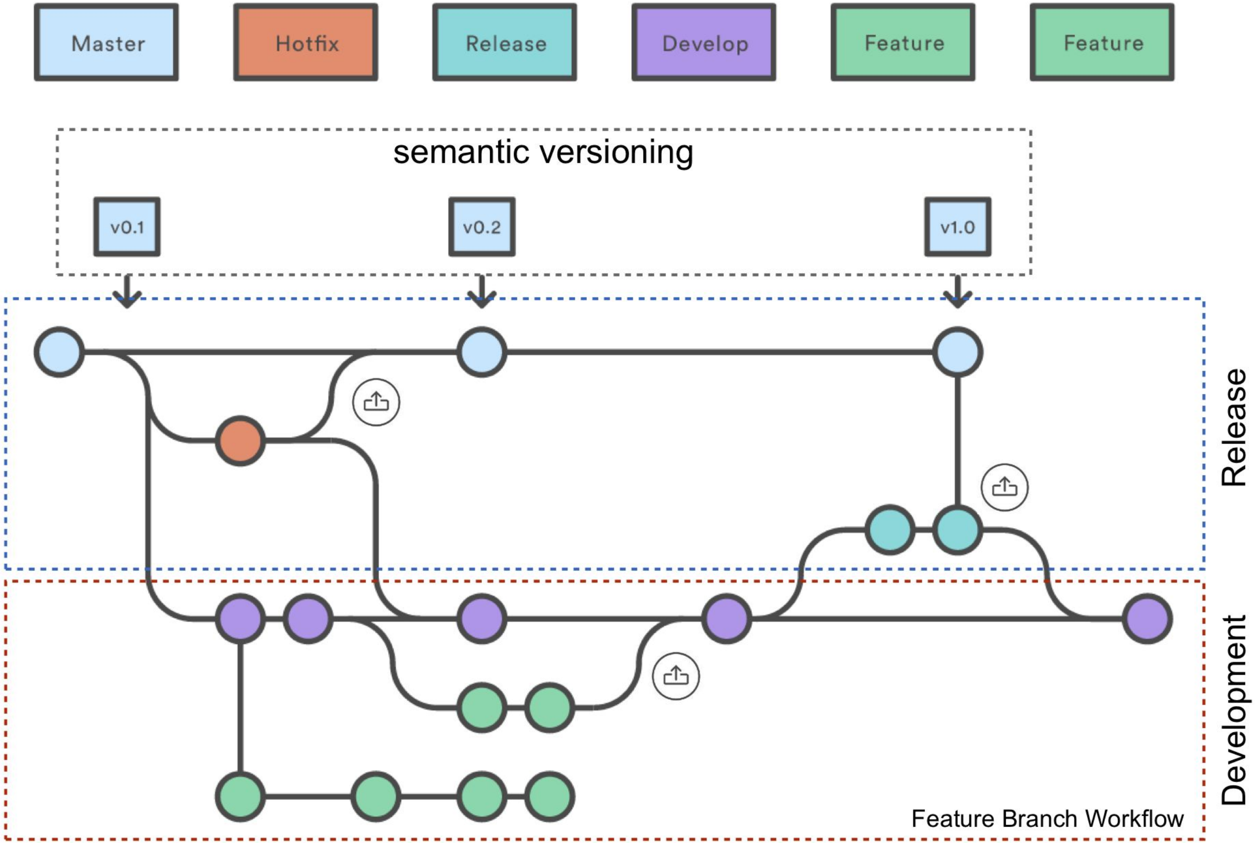
One feature branch for each feature

One temporary hotfix branch for each hotfix



Gitflow Workflow

15.19 Gitflow workflow



15.19 Gitflow workflow

⇒ naming conventions:

feature branch	`feature/<name>`
hotfix branch	`hotfix/<name>`
release branch	`release/v1.0`

⇒ for practical management, there are plugins to support this workflow explicitly in git https://github.com/nvie/gitflow

Pro	Con
+ separate release and dev + no dirty branch history + good for product with release base	- need to follow conventions to work smoothly - many branches, overkill for small projects

Merging or rebasing?

Merging vs. rebasing

⇒ Some persons argue you should always rebase on the master before you file a pull request.

⇒ this is more about faith than arguments, both solve the same problem

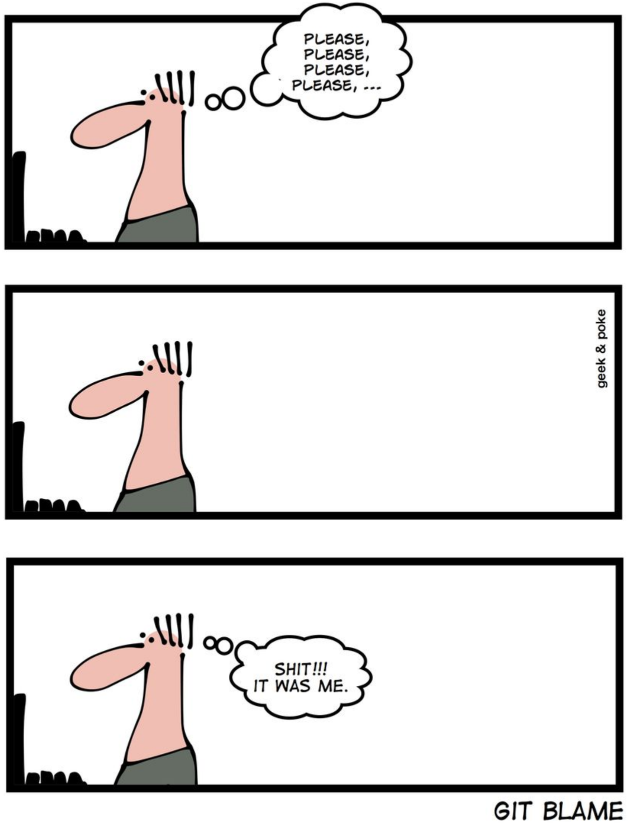
⇒ squash your commits when you make a pull request!

Rebase	Merge
+ clean, linear history + scales well with many developers/branches + no extra merge commit - more difficult, many developers make mistakes - reverting commits is difficult - destructive operation	+ clear history, shows exactly what happened + non destructive - leads to polluted and difficult to understand history when many branches/developers are involved in a project - extra merge commit

Final words

⇒ don't push blindly to a remote, always examine first what you did.

→ fixing branches on a remote is difficult and may screw up your team member's working copies.



End of lecture.

Next class: Tue, 4pm-5:20pm @ CIT 477