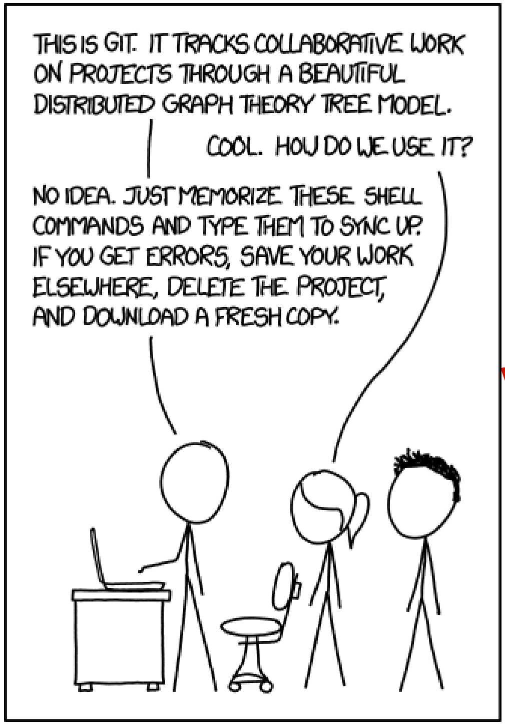

THIS IS GIT. IT TRACKS COLLABORATIVE WORK ON PROJECTS THROUGH A BEAUTIFUL DISTRIBUTED GRAPH THEORY TREE MODEL.
COOL. HOW DO WE USE IT?
NO IDEA. JUST MEMORIZE THESE SHELL COMMANDS AND TYPE THEM TO SYNC UP. IF YOU GET ERRORS, SAVE YOUR WORK ELSEWHERE, DELETE THE PROJECT, AND DOWNLOAD A FRESH COPY.

Versioning systems & Git

Maura Cerioli

Dipartimento di Informatica, Bioingegneria, Robotica e Ingegneria dei Sistemi

University of Genova

Dipartimento di Informatica, Bioingegneria, Robotica e Ingegneria dei Sistemi

Disclaimer

- What we discuss in this presentation applies to any kind of digital content, and to any process for its evolution
- However, we will keep to source code as content and software development as process
  - this is our intended goal
  - version control systems were introduced to manage software development

Agenda

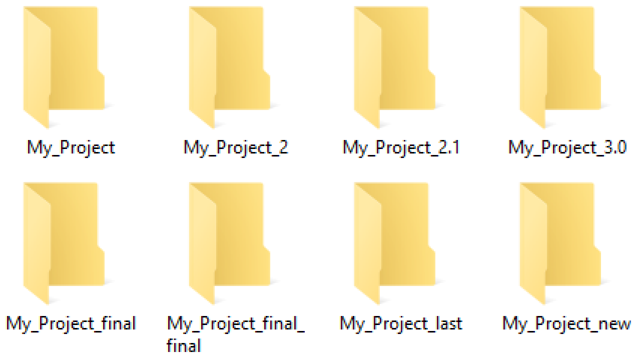
- What are versioning systems for ⟵
- Different approaches to versioning
- Git - Introduction

Working on a project
Problem 1: the Basis

Developing SW *naïve style*

- many versions
  - work progress
  - experimenting new ideas
  - changing technology
  - …
- requires
  - good documentation/ eidetic memory
  - strong rigour
  - dedication

to keep things working



Working on a project
Problem 2: Evolution

- *Real world* software has many coexisting versions
  - users who do not buy the new version
  - users who *cannot* install the new version…
- What should we do upon receiving from our best client a bug report for version 1.4.2.7 when we are working on 3.0.0.1?
- We need to
  - get back version 1.4.2.7
  - write a unit test capturing the bug report
  - find the error/bug **cause**
  - if the error is still present in any supported version fix it for those versions

- BTW: if 1.4.2.7 is now unsupported, and the bug does not appear in the current version can we just ignore it/mark it as solved?

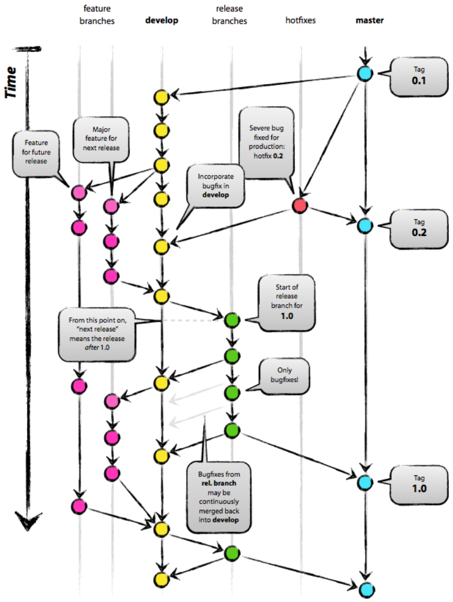
Working on a project
Problem 3: Cooperation

- Real projects are not one-man show
- How can we work independently on the same project?
  - distribute responsibility (*I work on subsystem A, you work on subsystem B…*)
  - lock (*do not touch class C till I tell you so*)
  - spend time harmonizing contributions
- It can be done by a well-knit team
  - error prone
  - trust among members

same problems (and solutions) for single users with many machines …but in case of error you can only be mad at yourself

What We Need

- Capability to go back in time to work on the system as it was
  - DAG ~~series~~ of snapshots
  - navigation functionalities
- Support for cooperation
  - independent work
  - merge of code individually done
    - minimal fuss
    - human control on *real* conflicts


feature branches
develop
release branches
hotfixes
master
Time
Tag 0.1
Feature for future release
Major feature for next release
Severe bug fixed for production: hotfix 0.2
Incorporate bugfix in develop
Tag 0.2
Start of release branch for 1.0
From this point on, "next release" means the release after 1.0
Only bugfixes!
Bugfixes from rel. branch may be continuously merged back into develop
Tag 1.0

- Software providing these functionalities is a version control system
  - somehow it adds a time dimension to a part of the file system

Agenda

- What are versioning systems for
- Different approaches to versioning
- Git - Introduction

Versioning Systems: 3 Generations

- Prehistory: eigthies
  - centralized on one server
  - check-out/check-in of files with **lock**
- Since nineties: client/server
  - Server = one remote repository aka *The Truth*
  - Clients = working copies to be synchronized with the server
  - Concurrent access, multi-file
- Since 2K: distributed
  - **local** repositories + peer-to-peer synchronization...
    - In practice, in most cases one peer plays the role of server

Versioning Systems: Generation 2

Basic ideas

- One Repository contains all versions
  - true history of the system
- All users (many of them)
  - work on their own local copy
    - no conflicts during development
    - more efficient
  - commit to the repository their sharable work
    - only commits based on current version are allowed
  - standard workflow
    - update current version from server
    - compare with work and solve conflicts
    - commit

sharable
⇓
it compiles + passes all tests

Otherwise...

Versioning Systems: Generation 2

- First major implementation CVS, current market dominator SVN (subversion)
- Limits
  - of the approach: most operations need connection to the server
  - of subversion: versions are memorized as *delta*
    - many frequently used operations are slow
    - *branching* easily becomes a nightmare

Versioning Systems: Generation 3

Basic ideas

- Many local repositories
  - each has a part of the truth
  - when they are all synchronized the truth is shared
- Each user
  - works on his/her own local repository
    - possibly on many different branches
  - when work on a branch is sharable
    - clean up local history of that branch (if needed)
    - push branch to others
  - push allowed only for commit based on current status
  - standard workflow
    - fetch current version from other’s repository
    - compare(, solve, locally commit)
    - push new commit(s)
- Main limit: much more complex to grasp than previous approaches
- Many examples: TeamWare (discontinued), Bazaar, BitKeeper, Mercurial, **Git**

Agenda

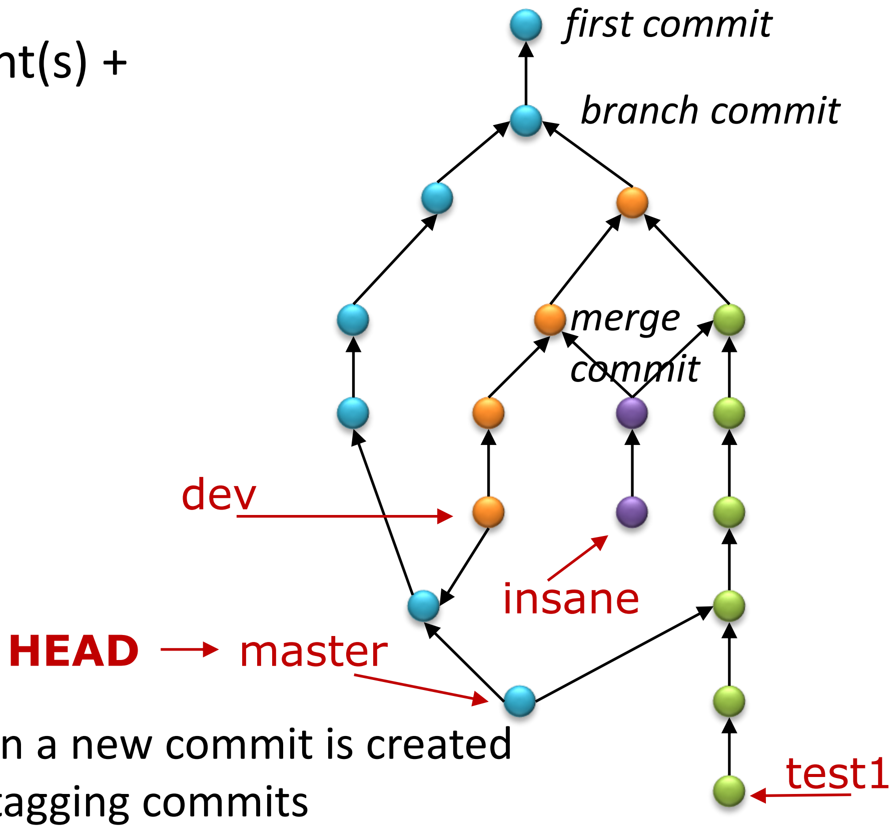
- What are versioning systems for
- Different approaches to versioning ⟵
- Git - Introduction

Git - History

- April 2005 free use of BitKeeper for Linux development is withdrawn
- No existing versioning system satisfies the need of Linux developers
- In less than one month Linus Torvalds and his group have Git up and running
- Major requirements
  - support for distributed BitKeeper-like workflow
  - very strong safeguards against corruption, either accidental or malicious
  - high performance

What’s a Git Repository

- A DAG of *Commits* + a bunch of pointers
  - arcs go from children to parent(s)
- Commit = snapshots + pointer(s) to parent(s) + metadata
  - Denoted by their hashing
    - No tampering with contents
- Snapshot = set of files
  - file content (*blobs*)
  - file name to hash of file content + hash of file content to content
    - same file content in different commits not duplicated
      - space saving
    - no tampering with contents
- Pointer = branch
  - branches are automatically updated when a new commit is created
  - it’s possible to create *frozen* pointers by tagging commits
- HEAD one pointer of different type
  - (attached) HEAD points to the branch we are working on
    - its files are available in the working directory



Entities Involved in the Process

- The Repository (all of it on each machine)
  - data are saved in efficient format
  - not easy to work on as they are
- Working directory
  - the sandbox you are playing within
  - may contain both tracked and untracked files
  - no automatic saving
  - when checking-out a commit tracked files are replaced by their versions from the commit
- HEAD
  - last checked-out commit (usually a branch)
- Index/Staging area
  - initialized with HEAD version of files
  - you can add new files, remove files, change files to save new versions
  - at the next commit its contents will be added to the Repo (and saved permanently as they are)

Git – Solo Repository
1. Repository Creation

- Any directory can be used as root of a repository
- Command

```
> git init
```

creates an empty repo in the directory the command was issued in

- no blobs, no pointers
- empty index
  - no associations name-hash
  - no associations hash-blobs

  - any file existing in the directory is
    - still there safe
    - not tracked into the repository
- You can check it using command

```
> git status
```

to get information on the repository

Git – Solo Repository
2. Add File(s)

- Files can be
  - tracked = are part of the repo
  - untracked = not part of the repo, but in the working directory

  immediately after creation all files are untracked
- To start tracking files command
  ```
  > git add <filename>
  > git add .  adds all files
  ```
  - never add generated files (binary code…)
  - add/edit the `.gitignore` file to avoid adding files you don't want to track
- Added files will be part of the next commit
- Files will be committed as they are when staged
  - file content is *staged* for commit
  - if they are furtherly changed, the new changes will not be committed

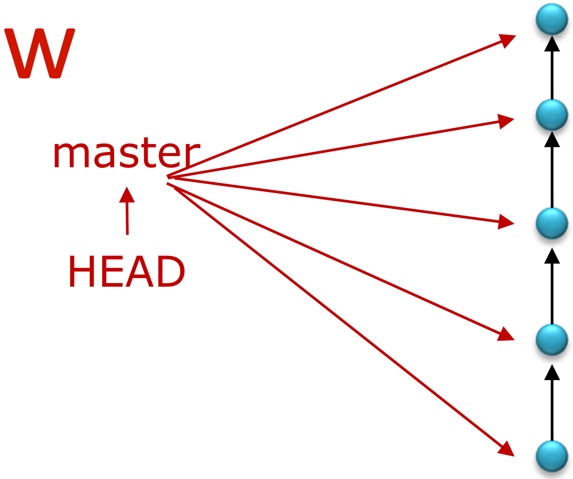
Git – Solo Repository
3. Staging and committing

- A tracked file can be
  - unchanged = same status as in the last commit
  - staged = changed and the current status added to the next commit
  - changed w.r.t. last commit (and its staged copy, if any)
- To stage a file (content), the command is `add`
  - `add` = add *something* to staging area
  - to *unstage* a file the command is `git reset <filename>`
    - file in the working directory is unaffected
- Staged files can be committed by command

  ```
  > git commit…
  ```

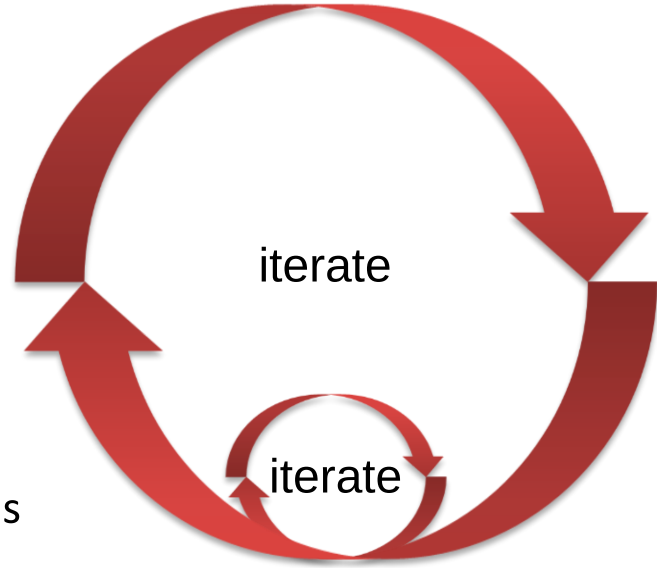
  commit has several parameters/options
  - Many of them *can* be inferred from global settings
    - name and email of the committer…
  - Some have to be provided
    - message (compulsory)…

Git – Solo Repository
4. Linear Workflow



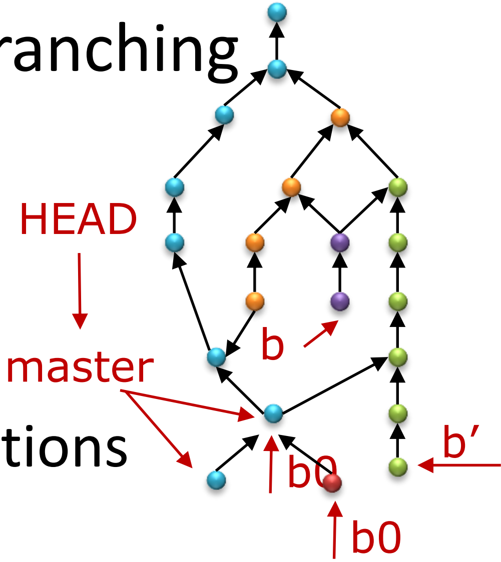
Simplest workflow

- Create empty repository with git init
  - no files, no branches…nothing
  - HEAD points to (unexisting) *master*
- First Commit
  - creates first branch (master)
  - (HEAD points to *now existing* master)
  - if no staged files, the commit has empty indexes and no blobs
- Stage some previously untracked files by git add
  - files are added to staging area, ready for commit
- Commit
  - a new commit is added to the current branch
  - branch is moved to point to the new commit
  - staging area is initialized with files in the commit
- Work on tracked files
- Stage modified tracked files
  - staging area is updated with the new versions of the files



Git – Solo Repository
5. Branching

- Many development processes are based on branching
- Branch = parallel development
  - *same* project different platforms
  - independent features
  - bug fixing
  - same feature different (alternative) implementations
  - …



- Linux development makes strong use of branching ⇒ Git has efficient support for it
- Branching = creating a new pointer
  - instantaneous and tiny cost
- Further commits on new and old branches cause divergent evolution

Git – Solo Repository
5.1. Branching

- To create a new branch
  `> git branch <new branch name>`
  - new branch created, you still working on the original one
- To switch to an existing branch
  `> git checkout <destination>`
  - move HEAD to `<destination>`
  - change (tracked) files in working directory to their version in `<destination>`
  - if work can be lost, switch is not allowed
    - unless you force it
- Shortcut: to create a new branch and switch to it
  `> git checkout –b <new branch name>`

Git – Solo Repository
5.2. Branching

```
git branch b0
...some work...
```

changes to working directory

```
git add .
```

changes to staging area

```
git commit
git checkout b0
```

changes to working directory

```
...some work...
```

changes to working directory

```
git add .
```

changes to staging area

```
git commit
...some work...
```

changes to working directory

```
git add .
```

changes to staging area

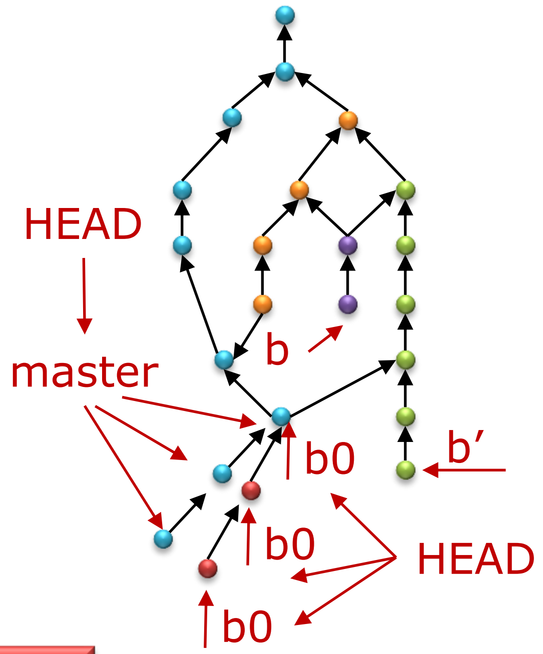
```
git commit
git checkout master
```

changes to working directory

```
...some work...
```

changes to working directory

```
git add .
git commit
```



Git – Solo Repository
5.3. Branching – useful commands

- To list all existing branches

```
> git branch
```

  - a * denotes current branch (HEAD)
- To list all branches yet to be merged into current branch

```
> git branch --no-merged
```

- To list all branches already merged into current branch

```
> git branch --merged
```

  - those are safe to delete
- To delete an existing branch

```
> git branch –d <branch name>
```

  - only for branches already merged into current branch
  - it can be forced by –D option (instead of `-d`)

Git – Solo Repository
6. Merging

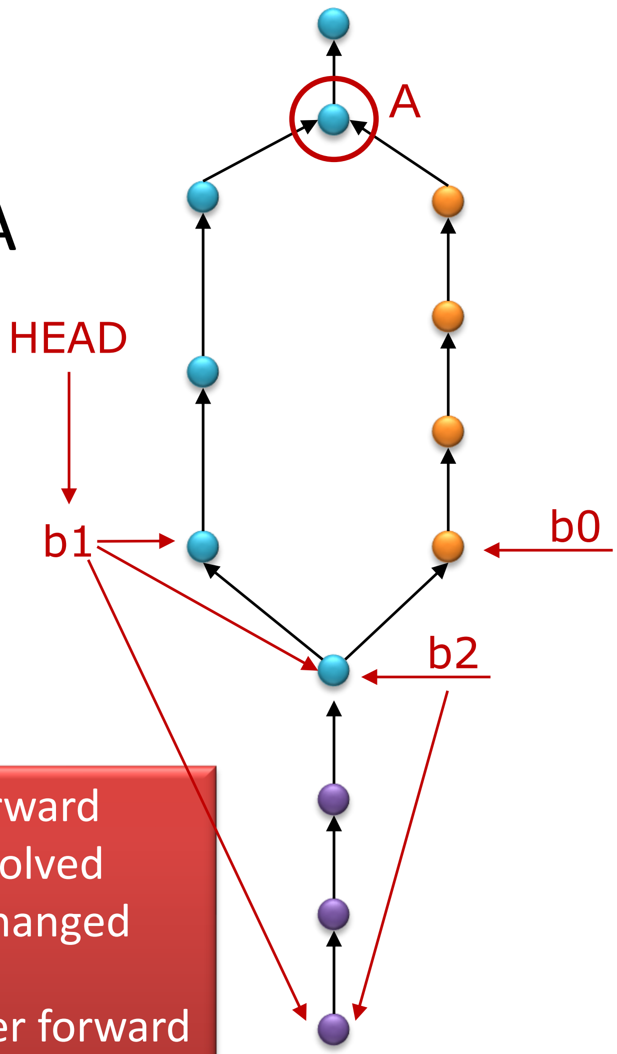
- In many cases we want to merge parallel work in one branch
  - development of independent features/subsystems
  - cooperation of team members
  - development and bug fixing
  - …
- In Git you can merge work done in another branch into the current one
  - also many branches at a time
- After merging branch b0 into b1
  - further commits on b0 could be merged in (possibly evolved) b1 in the future
  - if b0 will not have further development, it can be safely deleted

Git – Solo Repository

6.1. Merging – Ideal Situation(s)

```
> git merge b0
```

- find common ancestor A
- compare versions in b1 and b0 with A and creates Deltas
    - delta = patches to apply to A-version to get b0/b1 version
- if no conflicts (deltas for b0 and b1 affect different file areas)
    - merge files (versions from A+deltas from both b0 and b1)
    - add merged files
    - commit



Special case: fast forward

- branch b2 has evolved
- branch b1 is unchanged
- merge b2 into b1
    - move pointer forward
    - no new commit
    - no check for conflicts

Git – Solo Repository
6.2. Merging - Conflicts

- What if the same file has been modified in the same area on both branches?
- Git cannot automatically solve the conflict
  - non-conflicting files are staged
  - conflicting files are made ready in the working directory
    - non conflicting modifications already merged
    - both version of conflicting modifications are included, clearly marked
  - user has to solve the conflict
  - when all conflicts are solved, user has to explicitly commit
- Till the merge is completed no other commit can be done
  - `git merge --abort`
    reverts to commit before merging attempt

Git – Solo Repository
7. Rebasing

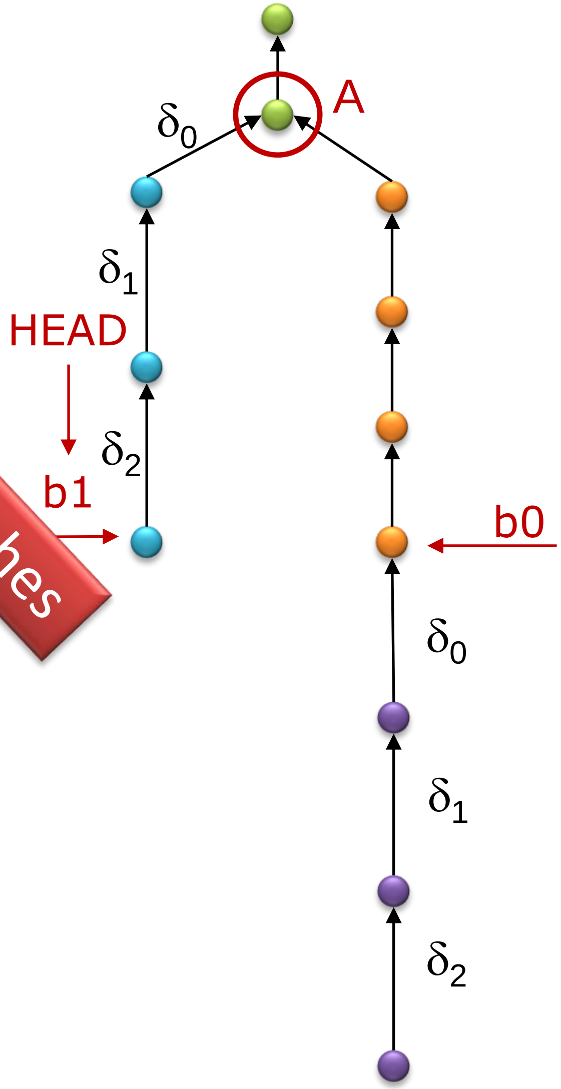
Alternative to merge

```
> git rebase b0
```

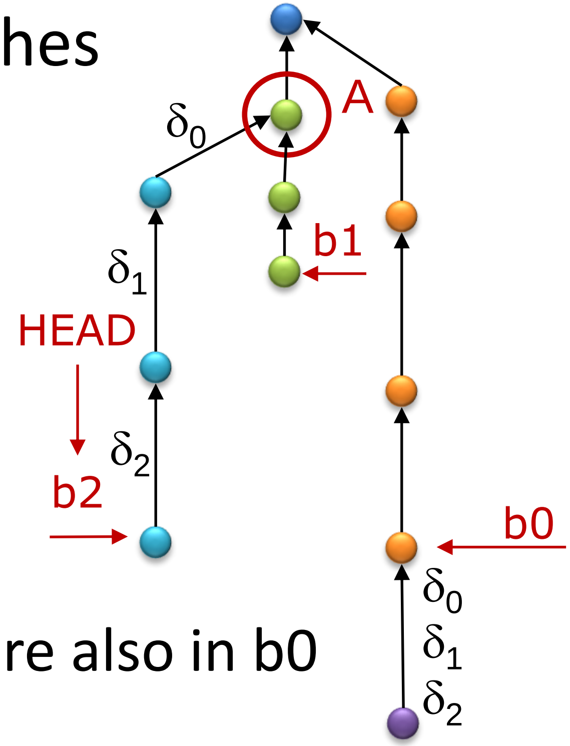
- find common ancestor A
- compute changes from A to HEA which are not in b0
- move b1 to b0
- apply changes and fast forward b1

- Blue commits become *unreachebale*
  - could and would be garbage collected
- b0 can fast forward to b1 if needed
- Cleaner history
  - not as happened

Never rebase shared branches



Git – Solo Repository
7.1. Rebasing++



Rebasing may be applied onto different branches

```
> git rebase b1 --onto b0
```

- as before computes changes in b2 not in b1
- but move b2 to, and apply changes onto b0

- Development dependencies change
  - branch b2 becomes child of b0 instead of b1
  - Makes sense if changes in b1 relevant to b2 are also in b0
    - special case b1 is independent from b2
- More general form

b1

```
– > git rebase <upstream branch> [<branch>]
  --onto <destination branch>
```

b0

If present checkout <branch> before starting
Convenient if you are on master and want to rebase some other branch onto it

Git – Solo Repository
8. Amending Commits

- Last commit can be *amended* if it had errors
  - correct the staging area if/as needed
  - use `git commit --amend [-m <new message>....]`
  - last commit is **replaced** by this
- History is changed/cleaned up
  - never again the "Ops I forgot a file" commit messages

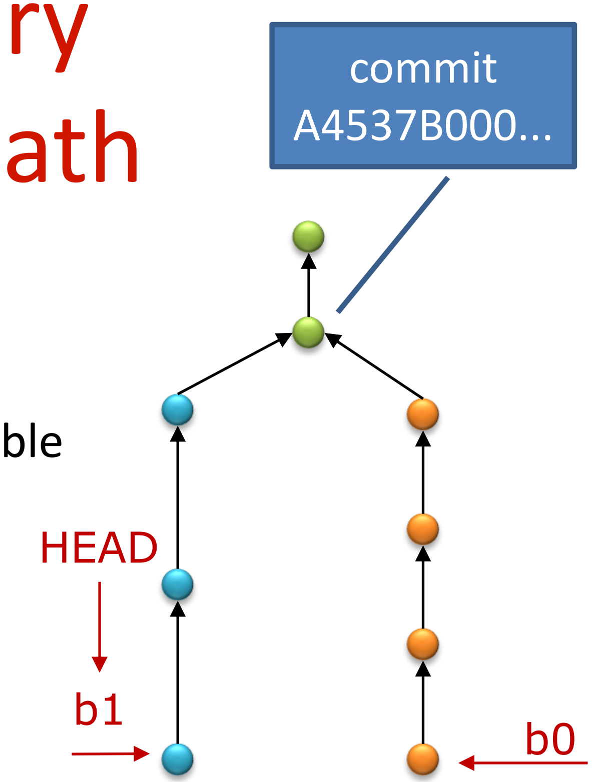
Git – Solo Repository

9.1. Reset without path

```
> git reset <commit>
```

changes

- current branch to <commit>
  - beware of losing data as commits become unreachable
  - with `--soft` nothing more occurs
- staging area to <commit>
  - default behaviour, or `--mixed` option
- with `--hard` option also changes working directory to commit
  - any uncommited work will be lost forever



```
> git reset A4537B000
```

- mostly used to change history
  - use with default option to go back to some previous commit
  - change the staging area with files from working directory
  - commit again
  - unreachable commits disappear from history
  - similar to/more general than amending

Git – Solo Repository
9.2. Reset with path

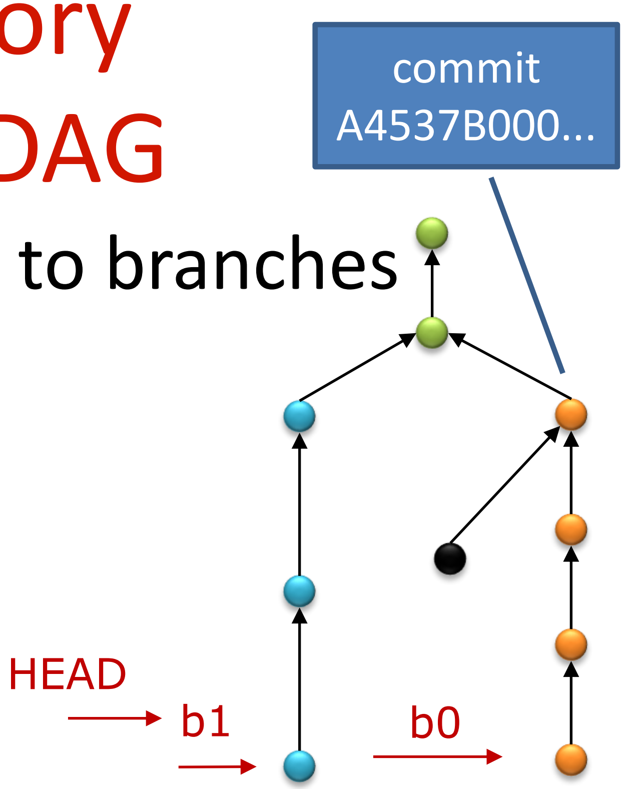
`> git reset [<commit>] <path>`

- path represents a tracked file
- file is changed in staging area to version in given commit
  - default behaviour, or `--mixed` option

- if <commit> is missing, the default value is HEAD
  - `git reset file1` = `git reset HEAD file1`
    copy last committed version of file1 to staging area
    = revert any add file1
    = is an *unstaging*

Git – Solo Repository
10. Moving in the DAG

- Checkout applies to commit not only to branches
- checking out a commit creates a detached HEAD
- next commit(s if any) will not belong to any branch
  - they will be unreachable at the next checkout



```
> git checkout A4537B000
```

- Better strategy
  - `git checkout <commit> -b <new branch name>`
  - `work on new branch`
  - `merge/rebase`
  - `delete`

There is world out there!

- Git is for shared work, so far we have considered just one repository
- Let’s move to the real world with many
- Two main points
  - connections between Repos
  - synchronization of related Repos

Cloning a Repo

- Somebody in the team has initialized the project Repo
- We want to have a local copy to work on it

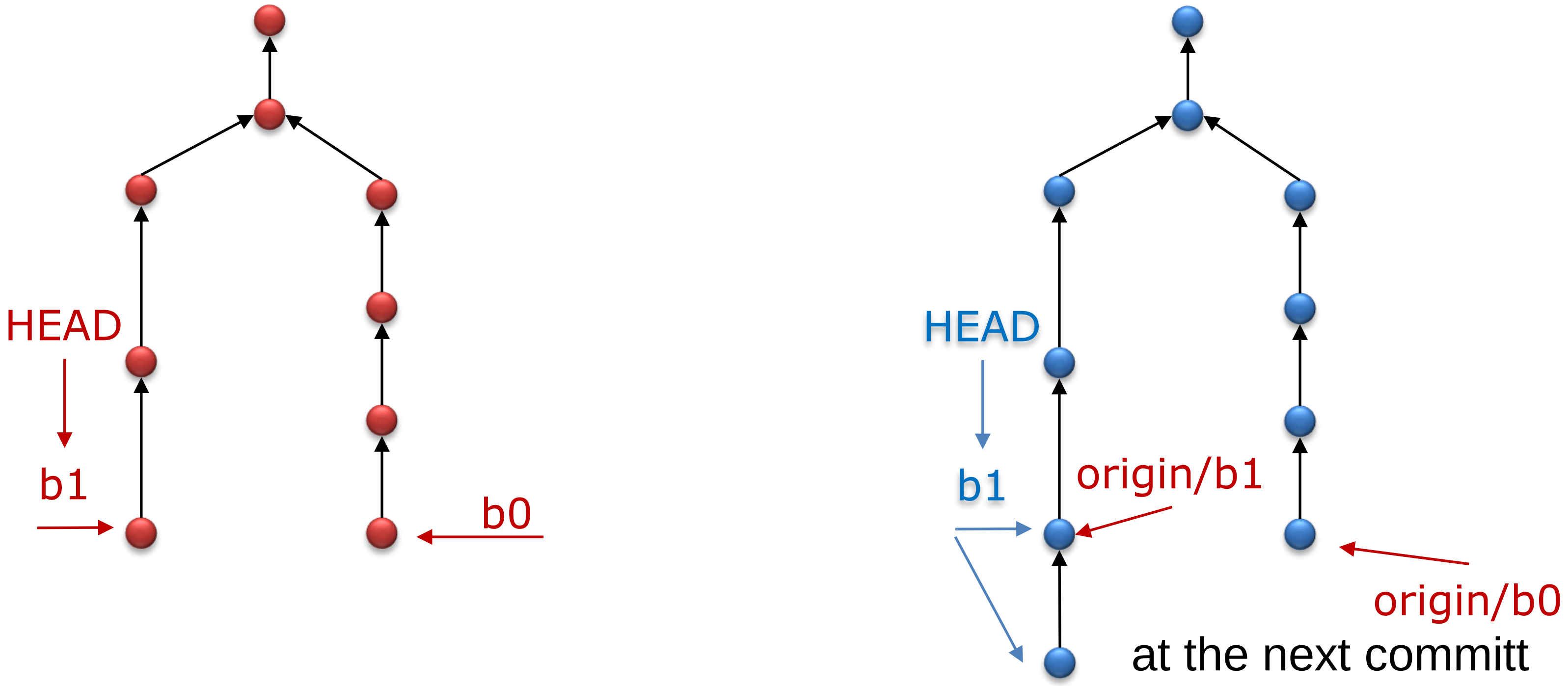
```
> git clone <url>
```

- Fetches the current status of the remote Repo
  - for each remote branch b a local *remote tracking* branch `origin:b`
- Creates local copy of the branch pointed to by remote HEAD (if any) + make local HEAD point to it
  - defines the remote branch as upstream of its local copy
    - default for merging/rebasing
  - no local copies of other remote branches are automatically created
    - if needed: `git checkout origin/<branch>` creates local copy (if not ambiguous)
    - ..with `origin/<branch>` as upstream
    - to list remote tracking branches `git branch -r`
- Defines the <url> as origin of the Repo
  - default remote repo for synchronizing

remote tracking branches are updated only by interaction with remote origin (not by new commit)

Cloning a Repo Example

- Remote Repo at
  `https://github.com/aaa/bbb`
- Local Repo



```
git clone https://github.com/aaa/bbb
```

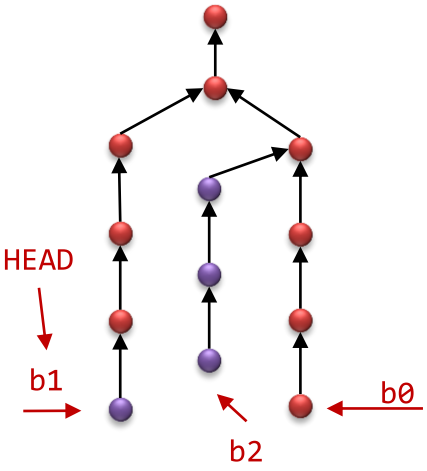
Fetching a Repo

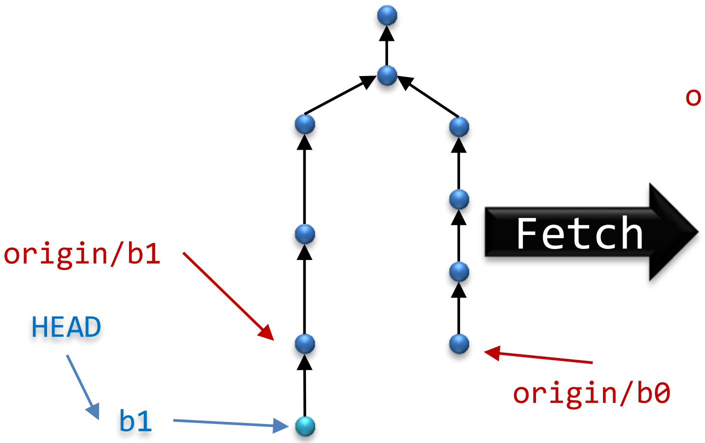
- We want to update our local copy w.r.t. what's happened on the original Repo

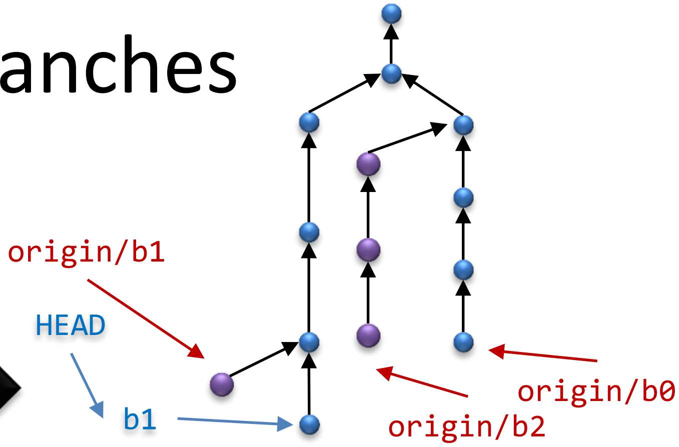
```
> git fetch [remote]
```

- Fetches the current status of the remote Repo
- Updates the remote tracking branches

https://github.com/aaa/bbb







NO merging
NO local downstream version of remote tracking branches

Pulling a Repo

- After fecthing you could (and should) merge remote tracking branches into your local versions
- A single line command can do all

```
git pull == git fetch + git merge
```

- Merges into current branch its upstream
  - origin/<branch> (unless you changed default names)
- If merge has conflicts pull remains pending
  - you can solve conflicts/apply same options as for merge
- Option -r changes behaviour to

```
git pull == git fetch + git rebase
```

Pushing to a Repo

- When you are ready to share

```
git push [<remote>] [<branch>]
```

- Works only if local branch is an evolution of the remote
  - for the remote must be fastforward
  - to push work you need to be aware of all the work committed so far
- Execution of scripts can be automatically required after each push ⇒*Hooks*
  - running unit tests
  - publishing summary somewhere...
- To push write rights are needed
  - otherwise make a *pull-request*
    - `git request-pull` generates a summary of pending changes
    - in GitHub extremely sophisticated process

GitHub

- Hosting service for Git.
  Possibly the most successful/used
- Many reasons
  - Free for a large community: public repo + private for education
  - So mainstream no programmer can live and prosper without a presence on it
    - part of standard curriculum
  - Many extra services, e.g. integration with unit tests, bug report systems…
  - Derived functions to make some common process easy, e.g. forking and pull request

GitHub – Forking

- Solves a common problem in open source development: project Repo readable by anybody, with write rights reserved to main contributors
- If somebody wants to contribute
  - clones the repo
  - works on it (e.g., solve some bugs)
  - makes a commit deserving sharing, but
    - cannot push (no writing rights)
    - it's unfeasible to make a pull request (working on own private machine, not server)
- Solution
  - fork the Repo = clone the Repo on GitHub
  - clone the forked Repo to own machine, work, keep forked and local Repos synchronized
  - keep forked Repo updated, by frequently pulling from original
  - when done, make a pull request on the original Repo for the forked one
    - as the forked one is on GitHub, owners of original Repo can easily pull from it
- Same technique also supports using a project as starting point for a new independent project
  - as in *LibreOffice is a fork of OpenOffice*