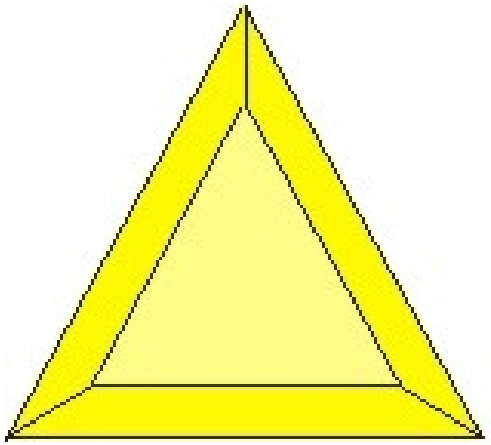


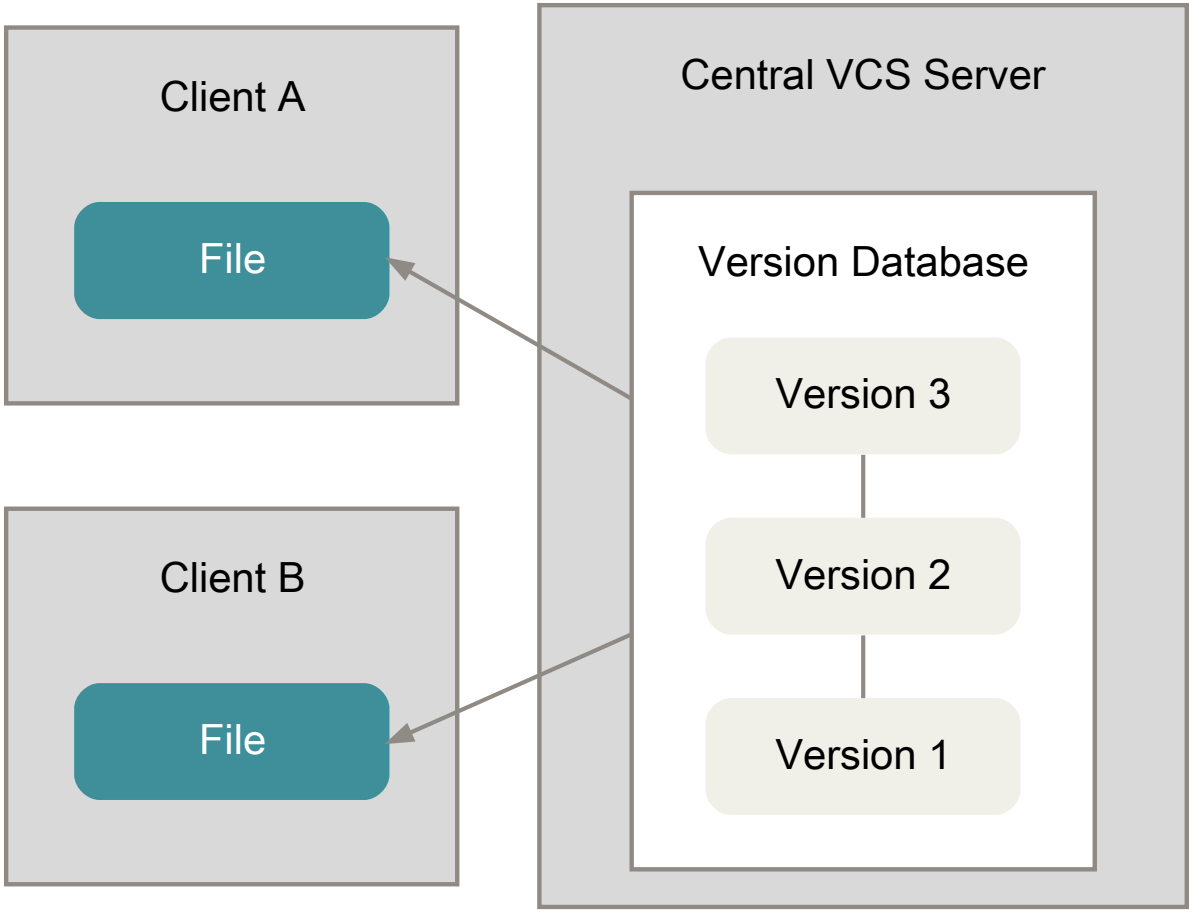
CSA

CSA Computer & Antriebstechnik GmbH

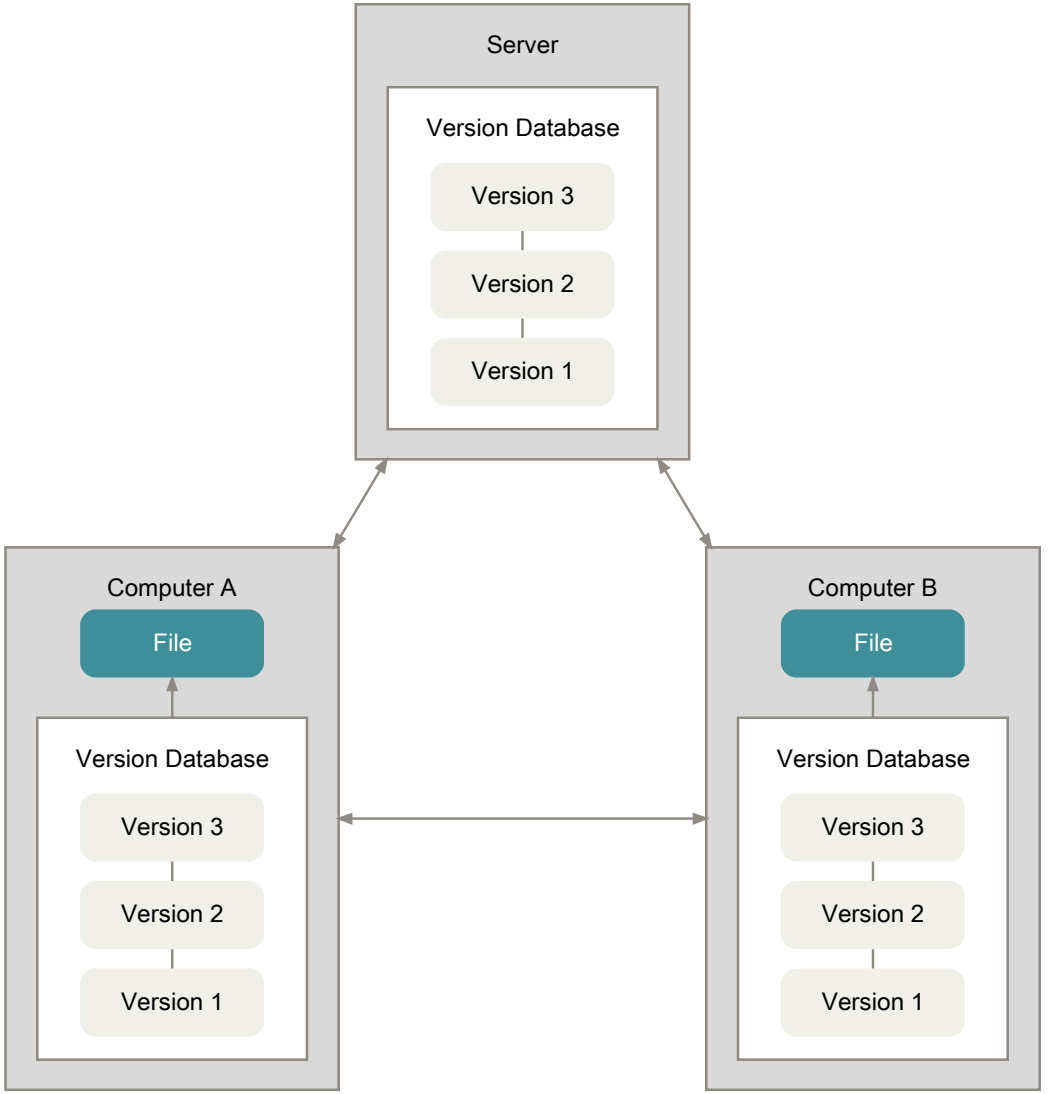

git

Comparison to other version control systems

Central VCS


Client A
File
Client B
File
Central VCS Server
Version Database
Version 3
Version 2
Version 1

Distributed VCS



Advantages of DVCSs

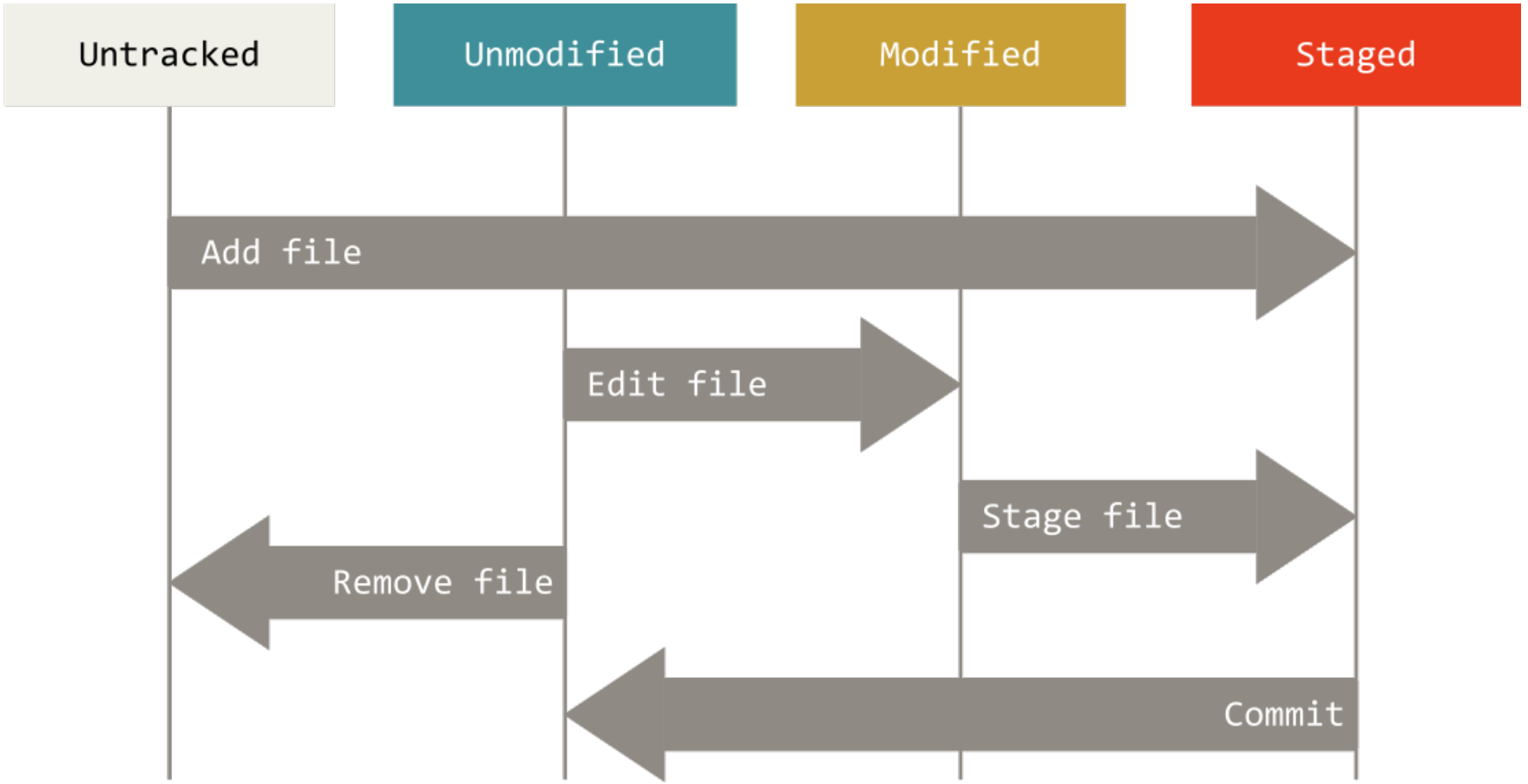
- No need to be connected to some central repository
- Local changes to repository do not affect other participants
- Every participant has a backup automatically
- Better performance without need of network

Working with Git

Basic principles

- All operations are done on the "current" branch
- Most operations should be done with a clean working directory
- As long as you did not publish your commits to a central one, you can change almost all of them
  - Change the order of the commits
  - Split or combine commits

Basic principles
Lifecycle of a file



First steps

- Basic command syntax:

```
$ git <verb>
```

- Get help:

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

First steps

- Add file to repository

```
$ git add {file name}
```

- Make a commit:

```
$ git commit -m "{commit message}"
```

First steps

- Modify a file

```
$ git add {file name}
$ git commit -m "{commit message}"
```

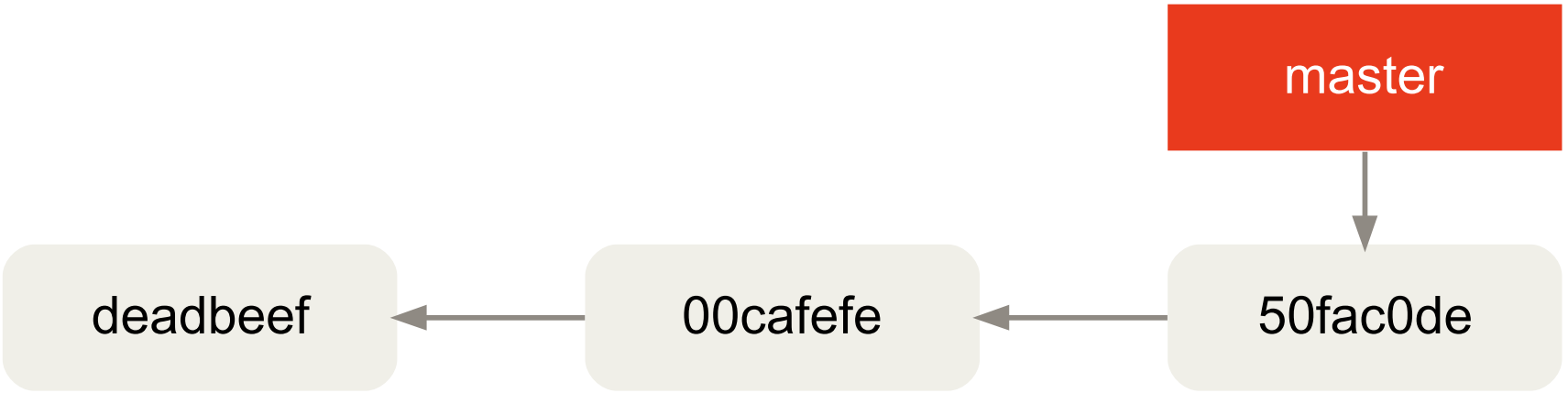
First steps
.gitignore

- Should be added at the very beginning
  - Ideally with the first commit
- Large collection of templates on GitHub
- Notes:
  - Affects only "untracked" files
  - No effect to files that are already in the repository (and modified)

Branching

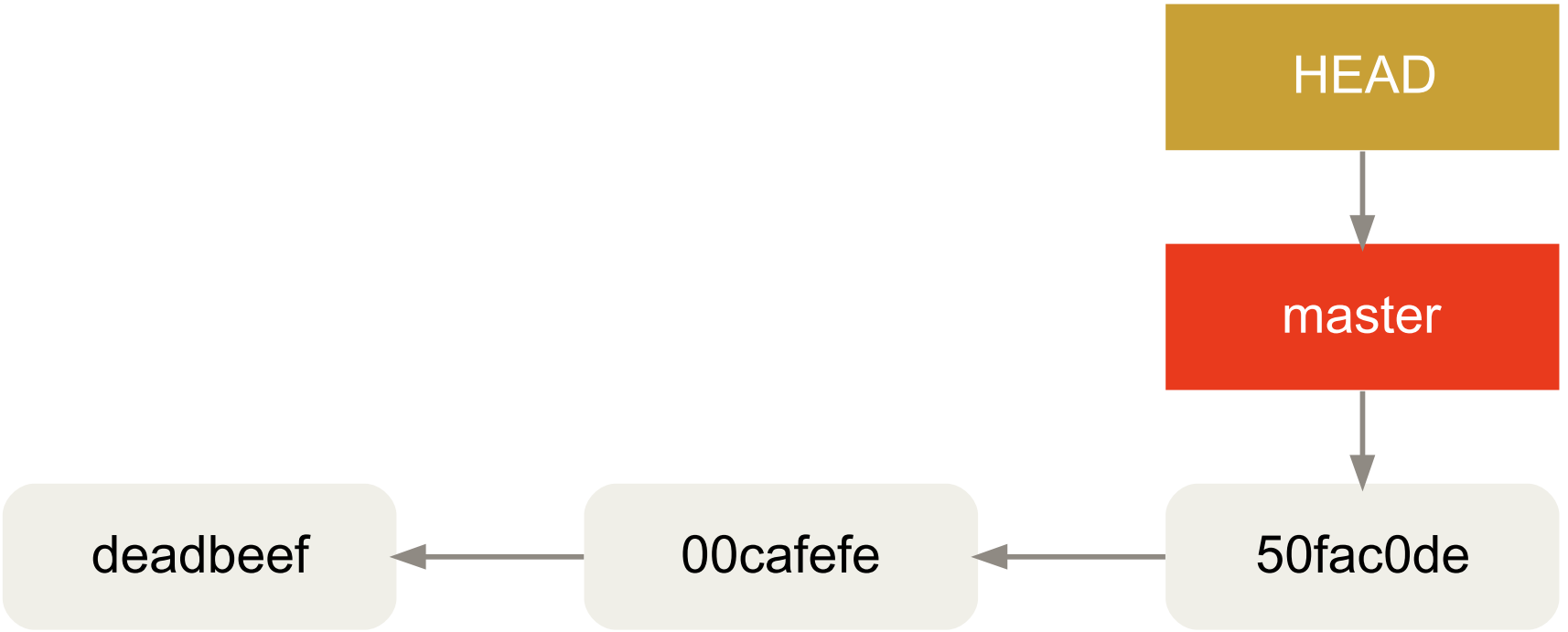
Introduction

Branches are lightwheight – they are just pointers



Introduction

HEAD pointer represents "current position"

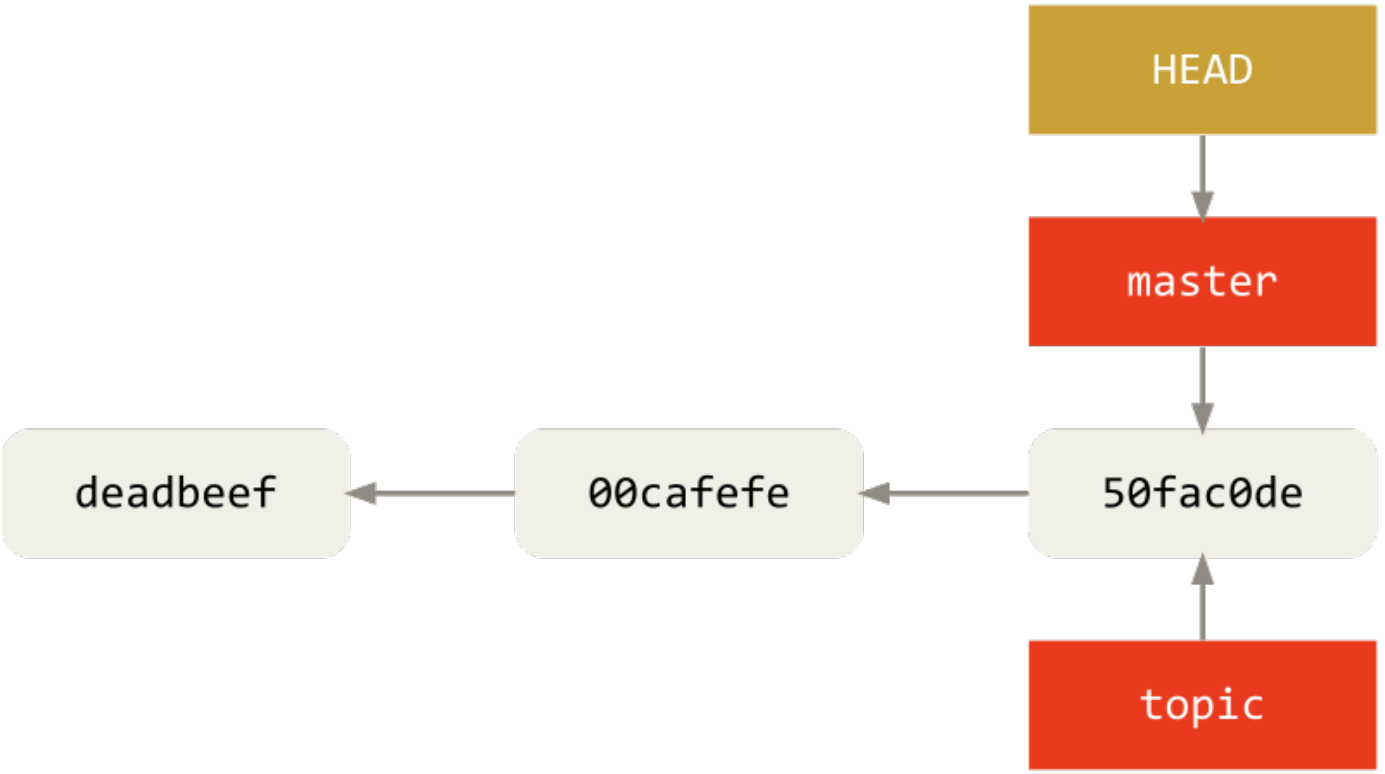


Introduction

- "master" branch
  - Default branch
  - Created with first commit automatically
  - Nothing more special about "master"

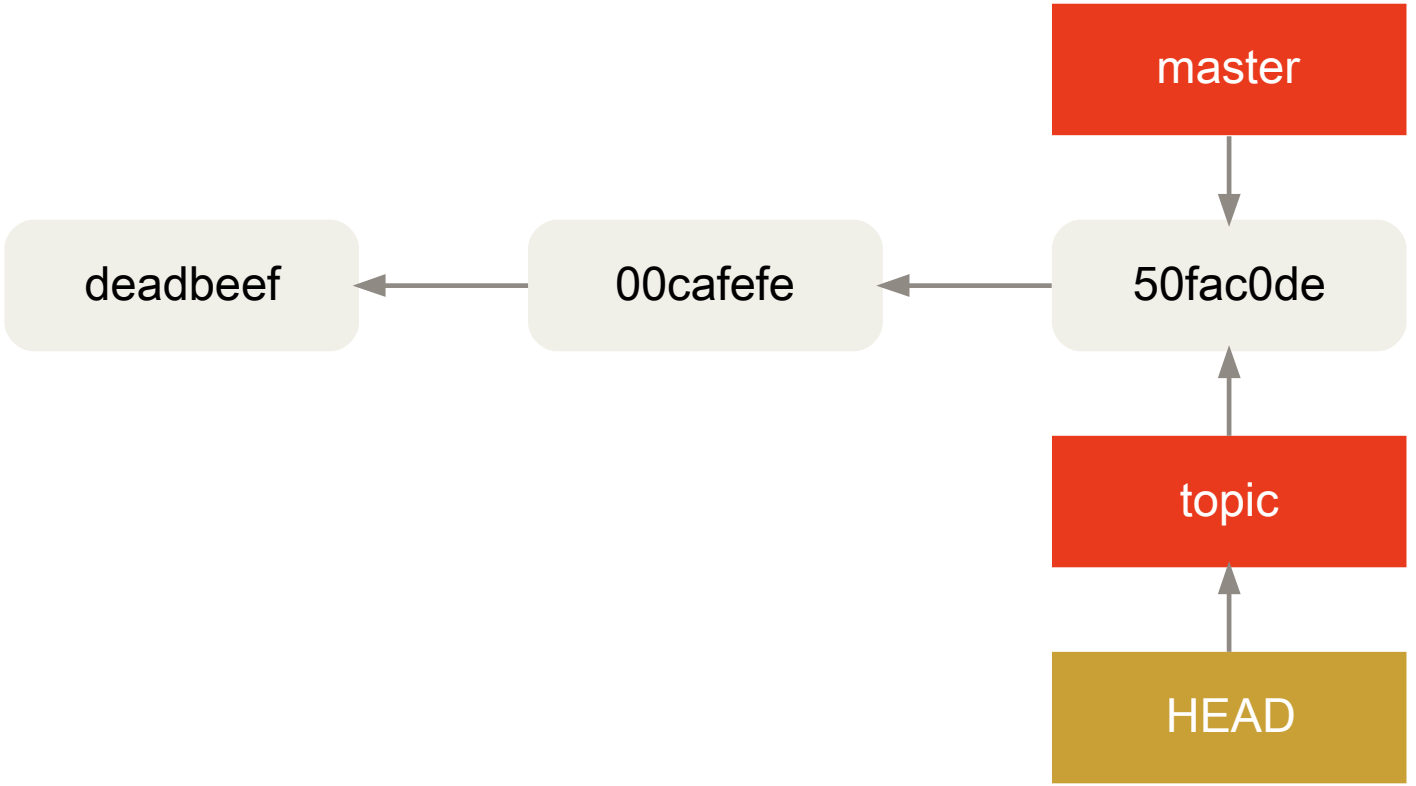
Create a branch

```
$ git branch topic
```

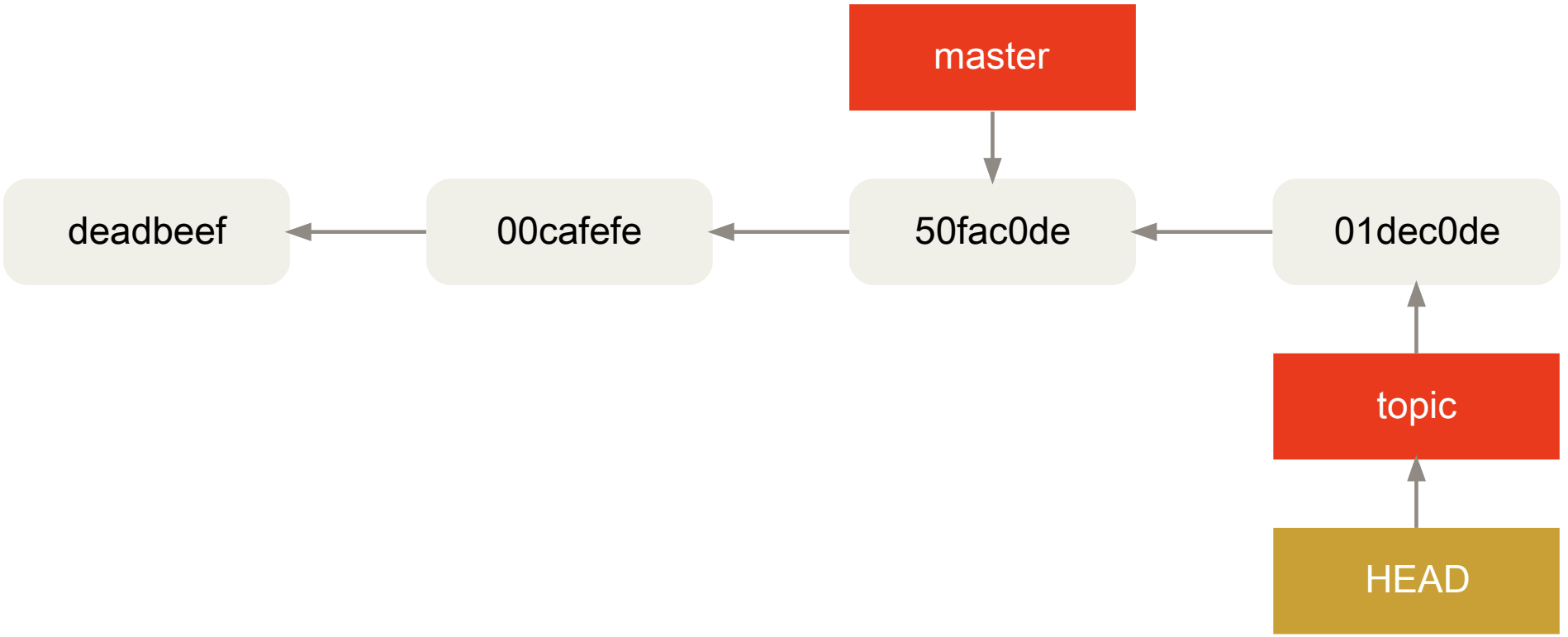


Switch branch

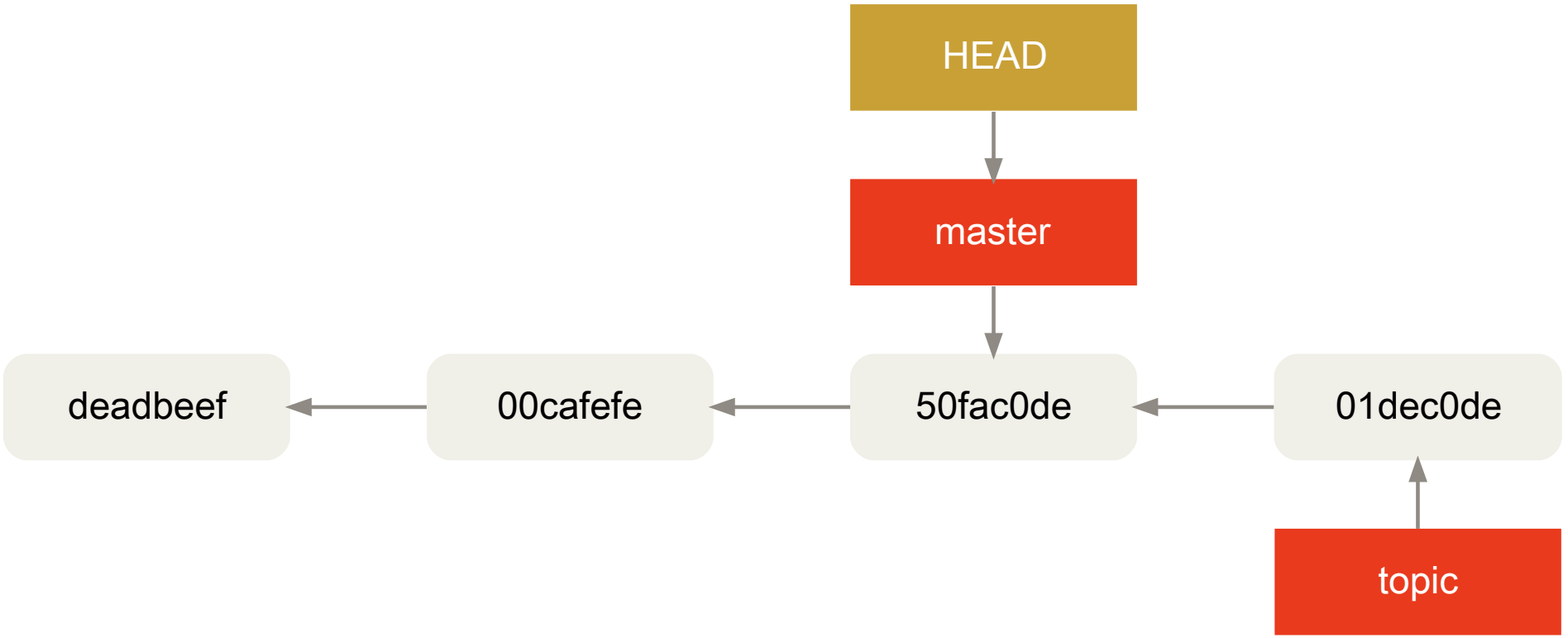
```
$ git checkout topic
```



Committing on branches

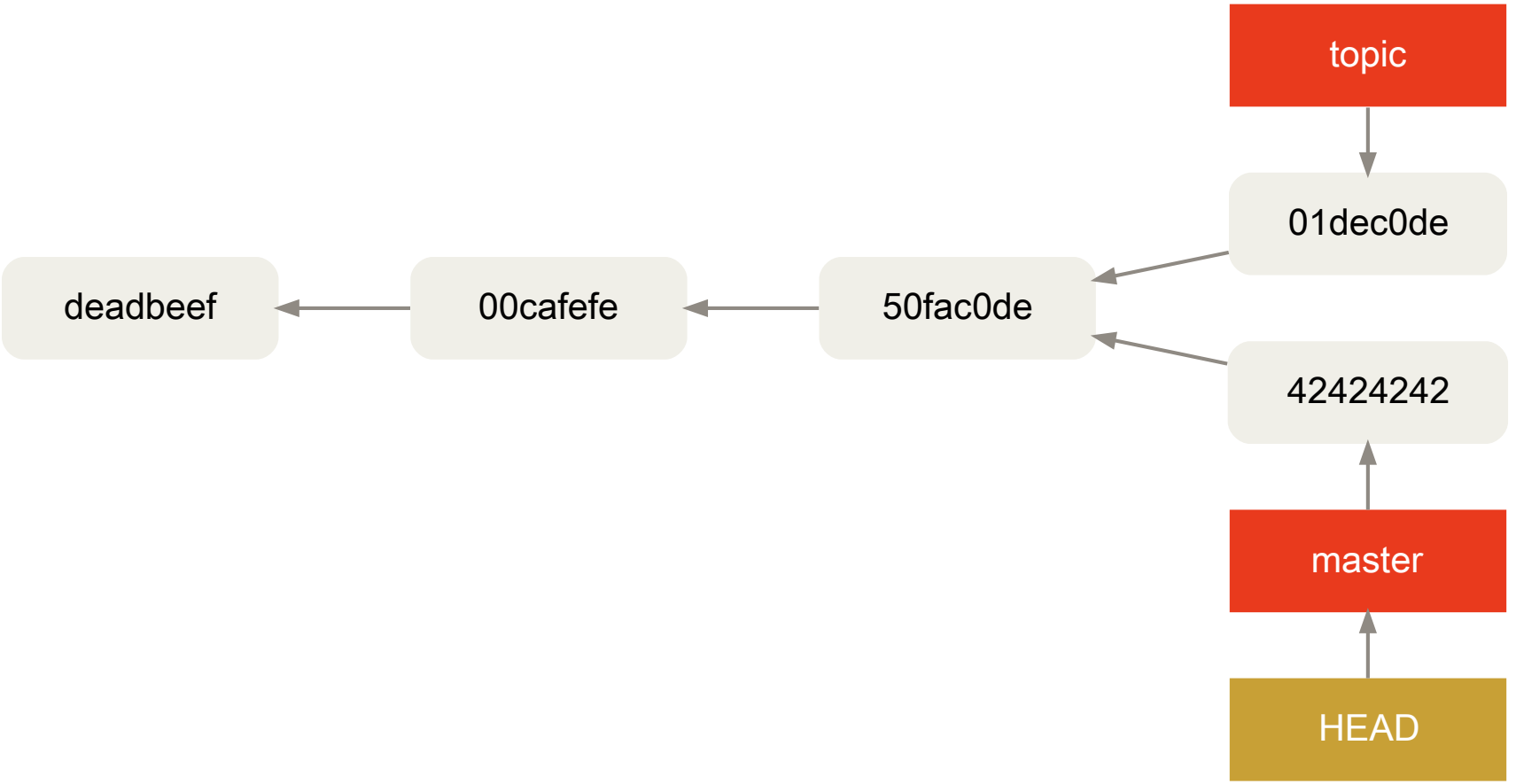


Committing on branches
Switch back to "master"



Committing on branches

Commit on "master"



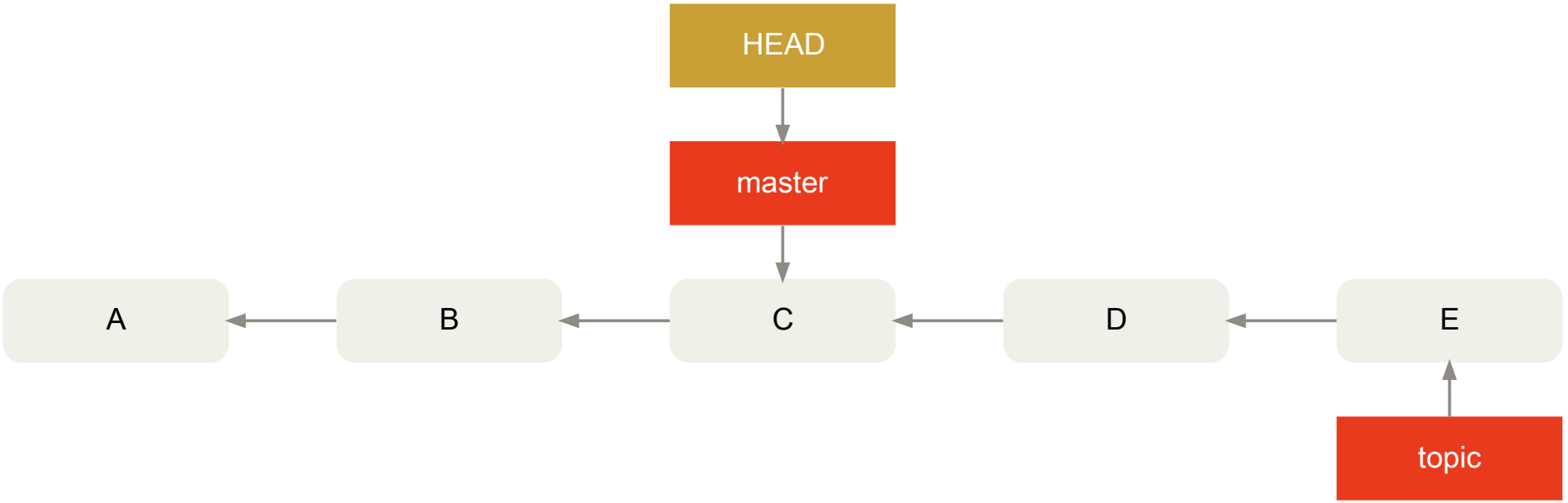
Integration (merge and rebase)

Integration

- 3 merge strategies:
  - Fast-Forward
  - Recursive
  - Rebase (use with caution)
- Default is fast-foward if possibly
  - Recursive if fast-forward is not possible

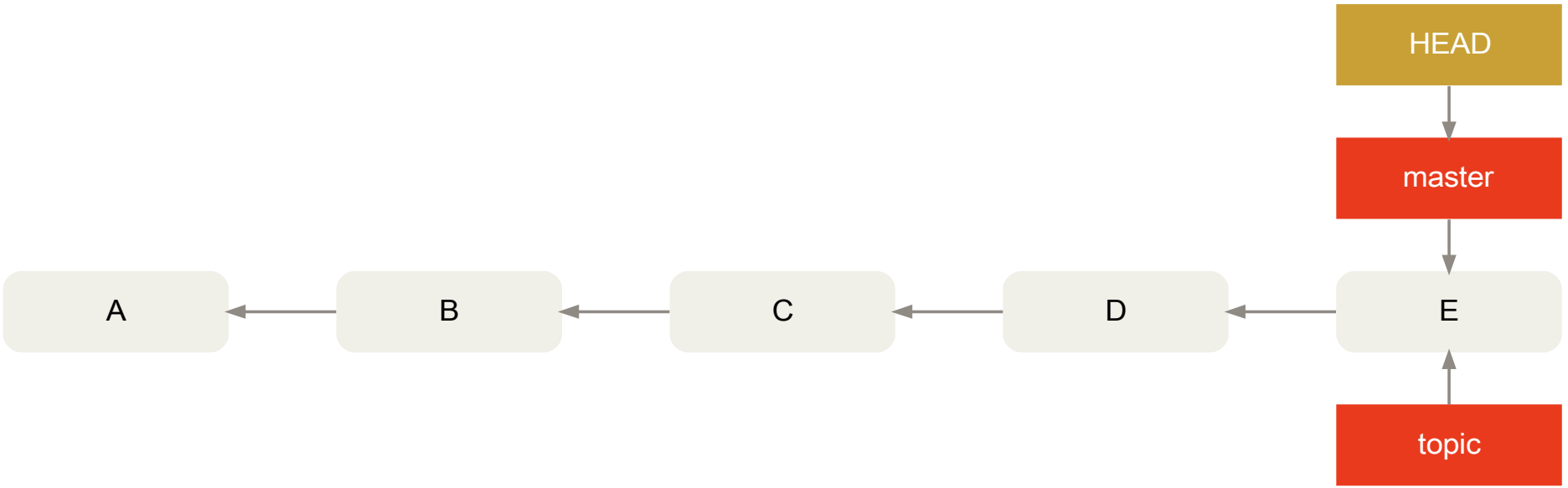
Fast-forward merge

Possible if history did not diverge



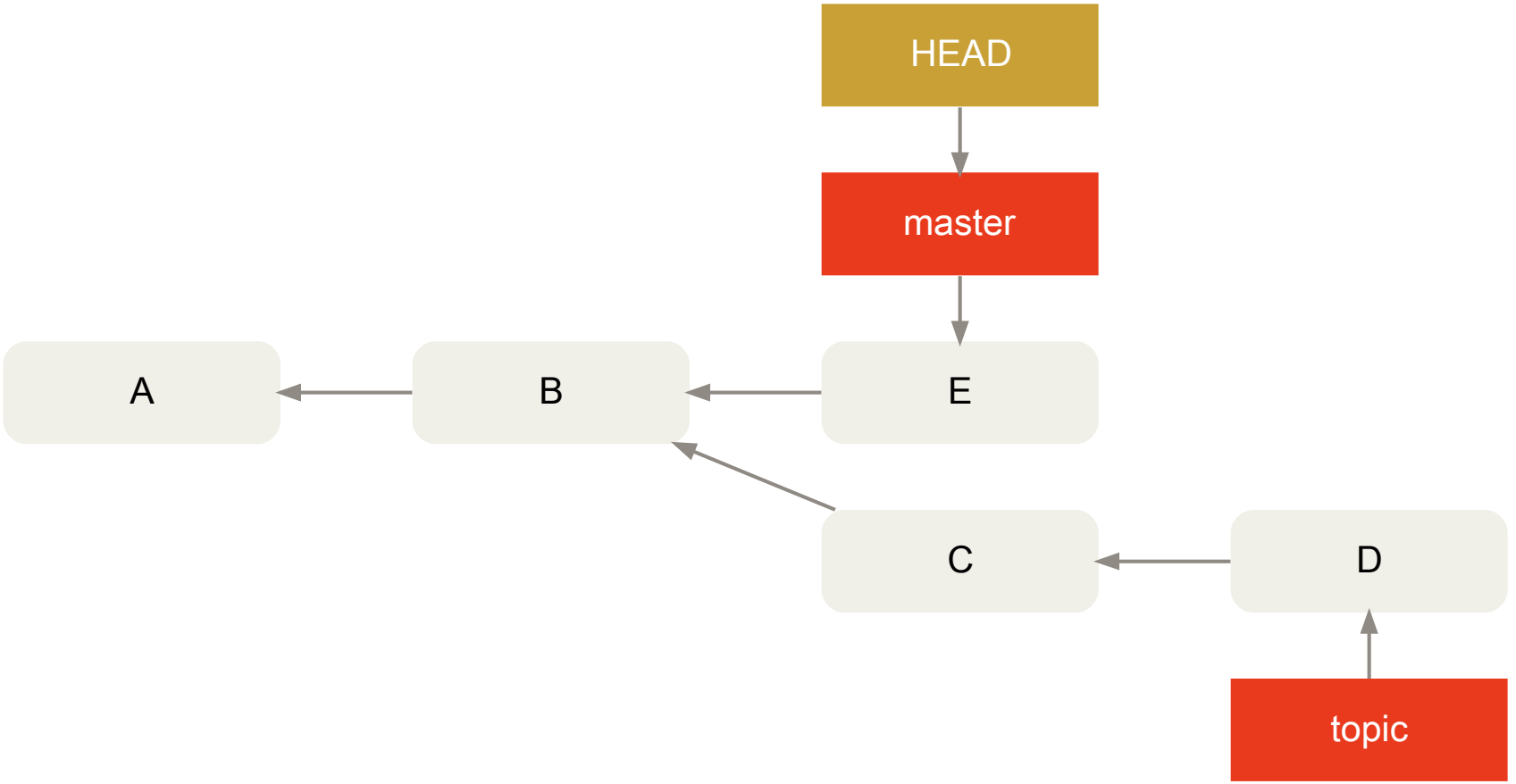
Fast-forward merge

After merge:



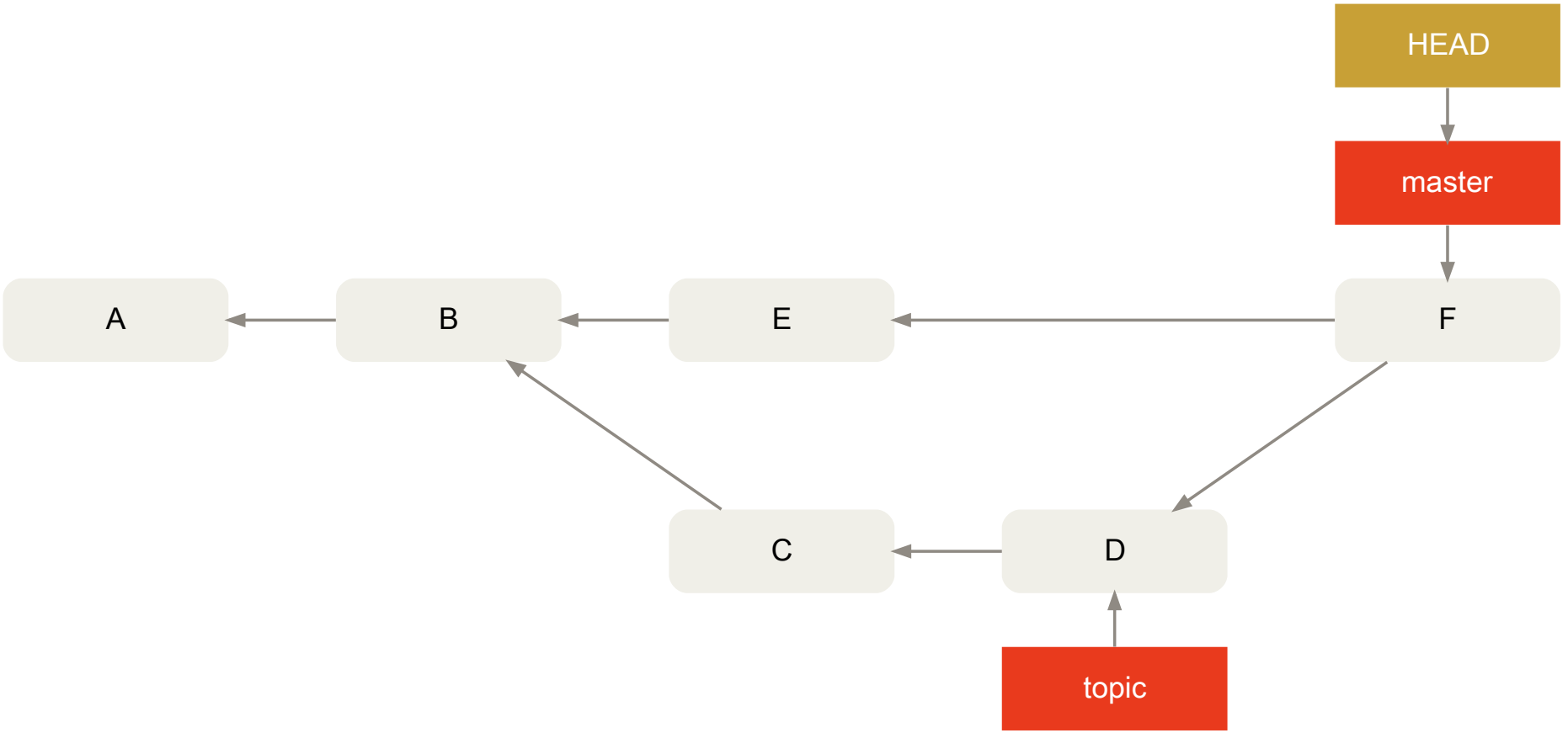
Recursive merge

History is diverged:



Recursive merge

New commit for merge:



Rebase

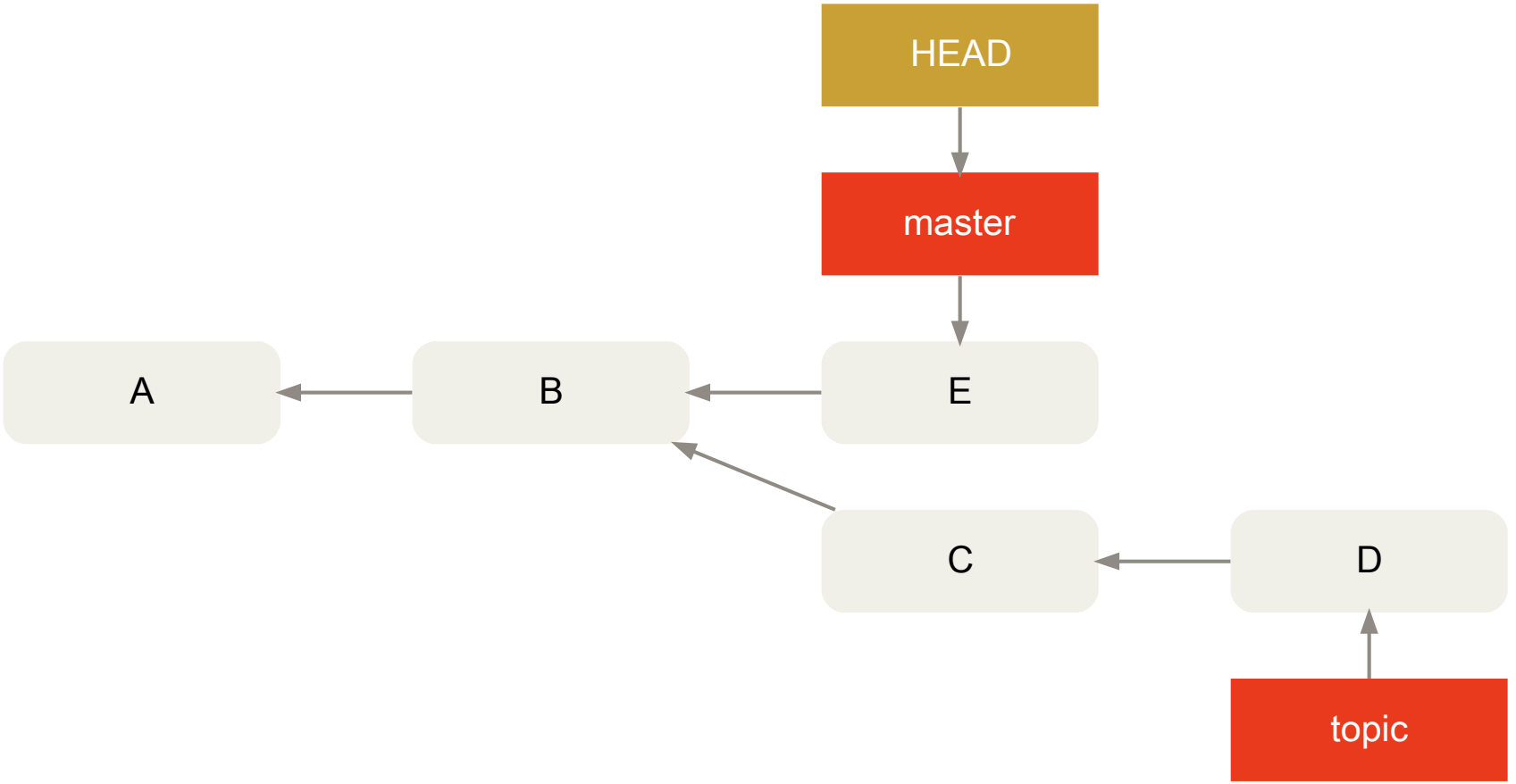
- Change commit history!
- All the things possible!
  - Change the order of commits
  - Split commits
  - Merge multiple commits to one
  - Make a diverged history linear

Rebase

- But use with caution!
- No big problem with unpublished repositories
- At published repositories:
  - Other people have to reproduce your rebase
  - If not:
    - Repositories get screwed up
    - People will hate you

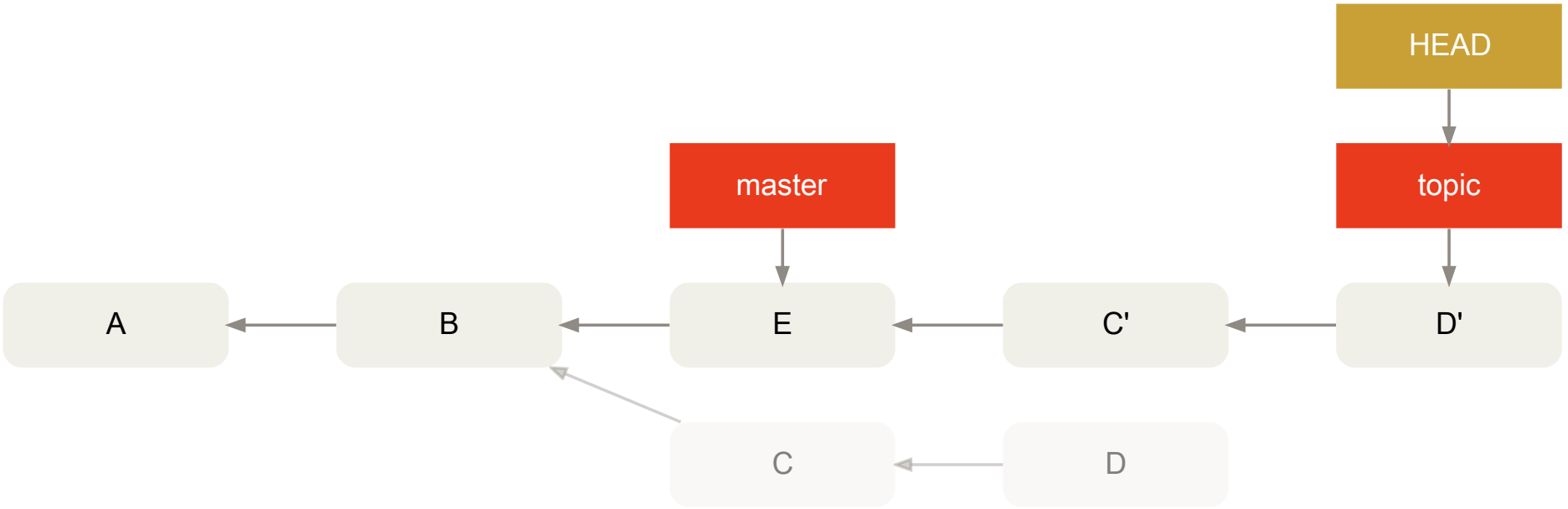
Rebase

Diverged history again:



Rebase

Diverged history made linear:



Rebase
Lightweight

```
$ git commit --amend
```

- Correct last commit
  - Change / add / delete files as intended
  - Change last commit message
- Better alternative to "correction commits"
  - If commit is not published yet!

Rebase
Nuclear option

```
$ git filter-branch
```

- Change whole history
  - That is reachable by the HEAD pointer
- A command or script is executed at every commit
- E.g. for deleting build artefacts out of history

Rebase
Notes

- Backup your repository if you are not sure
- Participants have to prepare:
  1. Publish the current state of work
  2. Backup the repositories
  3. Rebase the private repositories to the public rebase
     - Or clone a fresh repository

Merge conflicts

- Same as other VCSs
- Git will not perform automatic commit
- User must resolve conflict
  - Edit conflicted file
  - Mark as resolved:
    ```
    $ git add {conflicted file}
    ```

Remote repositories

Remote repositories
Basics

- Needed when multiple participants work on the same prject
- No need for any special server
- Nothing more than directory with certain data
  - Much like a personal repository
  - But without working copy

Remote repositories
Basics

- Any place is legit for a "remote" repository, e.g.:
  - HTTP server
  - Network file share
  - Even local hard drive
- Remote has to be registered in personal repository
  - Multiple remotes are possible
  - Default is named "origin"

Remote repositories
Basic commands

- Initialize new remote:
  ```
  $ git init --bare
  ```
- Clone from remote:
  ```
  $ git clone {URL} {directory}
  ```

Remote repositories
Basic commands

- "Push" to remote:
  `$ git push {remote name} {branch name}`
- Get changes from remote:
  - Without touching your local changes:
    `$ git fetch {remote name}`
  - With merge to the current branch:
    `$ git pull {remote name}`

Workflows

Workflows
Branching

- Centralized VCSs
  - Unusual
  - If made, then with big caution
- Decetralized VCSs
  - Regulary
  - Usual to have many branches
  - In fact, every participant has his own branch

Workflows
Branching

- 2 extremes:
  - "Master only workflow"
    - Everything is made on a single branch line
  - GitFlow
    - Multiple eternal living branches
    - Short living feature branches

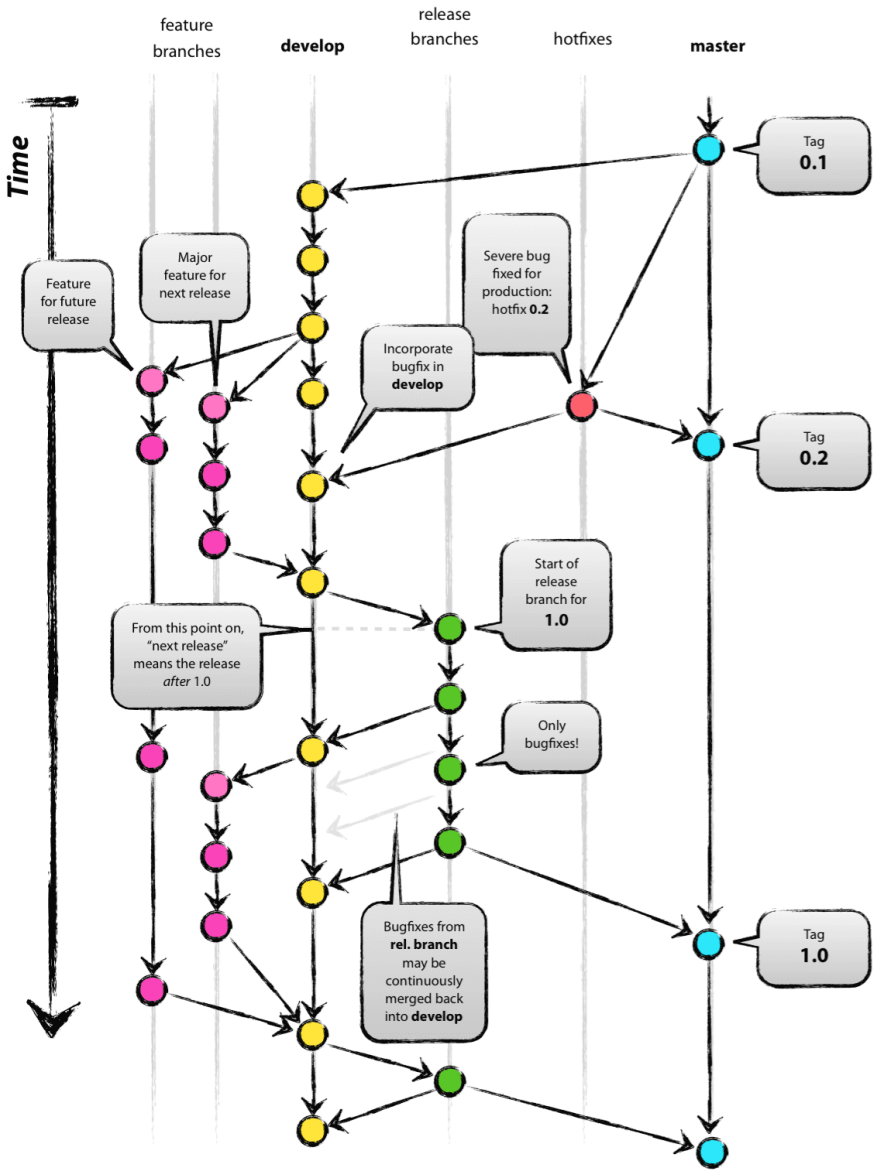
Workflows
Branching

- "Master only workflow"
  - Appropriate for individuals or very small teams
  - If either:
    - Breaking changes can be accepted
    - A QA mechanism exists

Workflows
Branching

- GitFlow
  - Eternal branches
    - "master": For release
    - "develop": Current state
  - Temporary branches
    - Release branches: For stabilization
    - Feature branches: For features that are implemented currently

Workflows
Branching

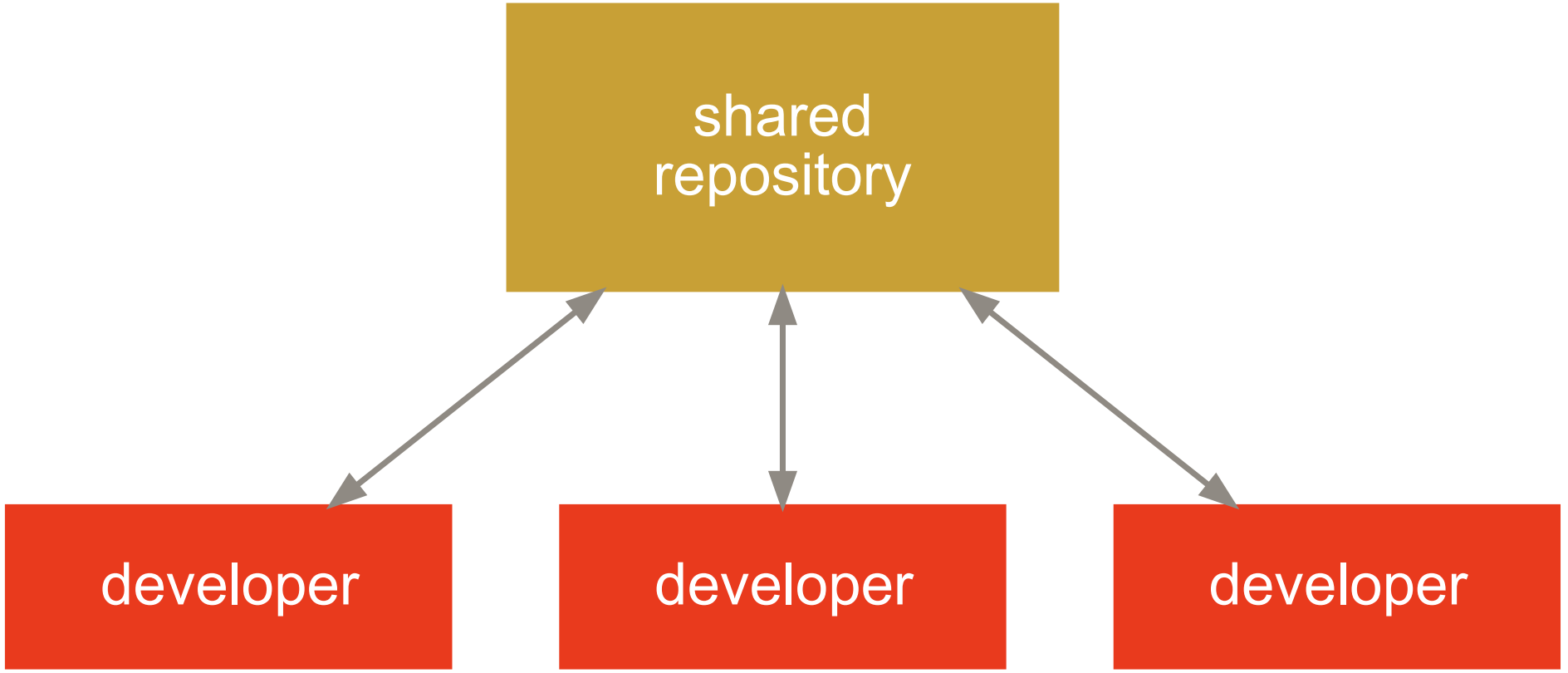


Workflows
Remote repositories in teams

- "Blessed repository" for every project
- 3 common basic models:
  - 1 public repository that is used by everyone
  - Integration manager
  - Benelovent dictator

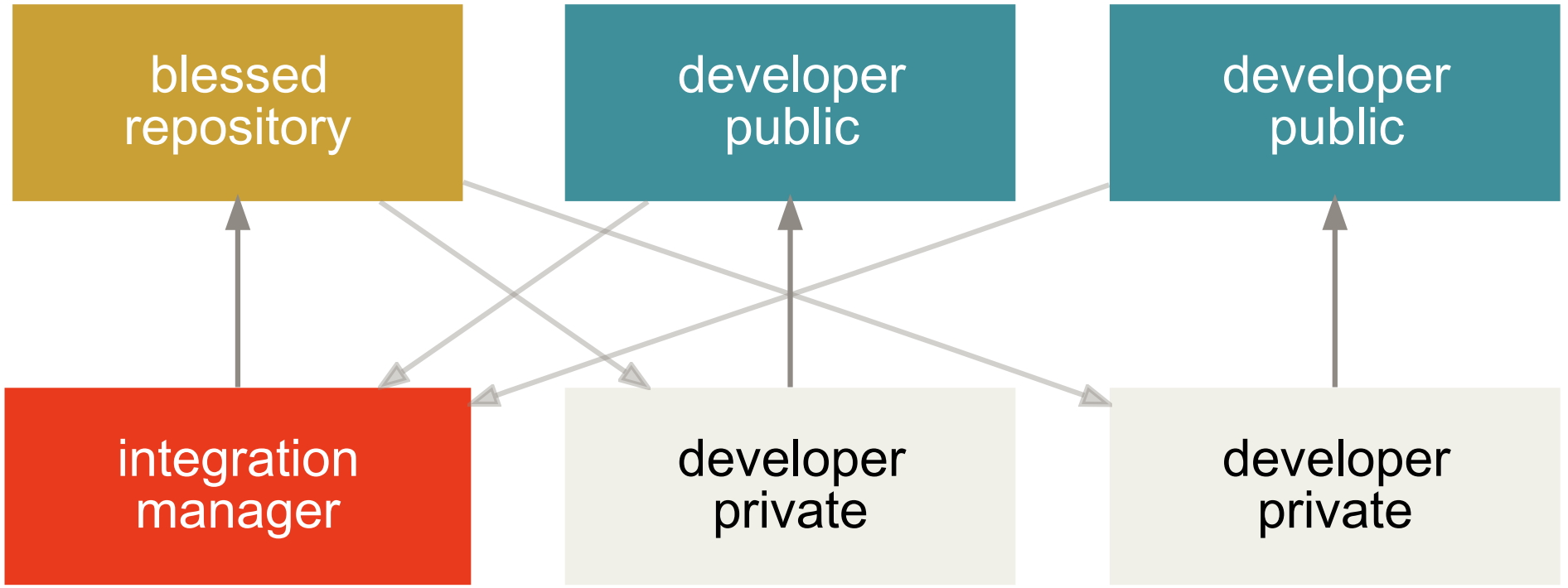
Workflows
Remote repositories in teams

Centralized:



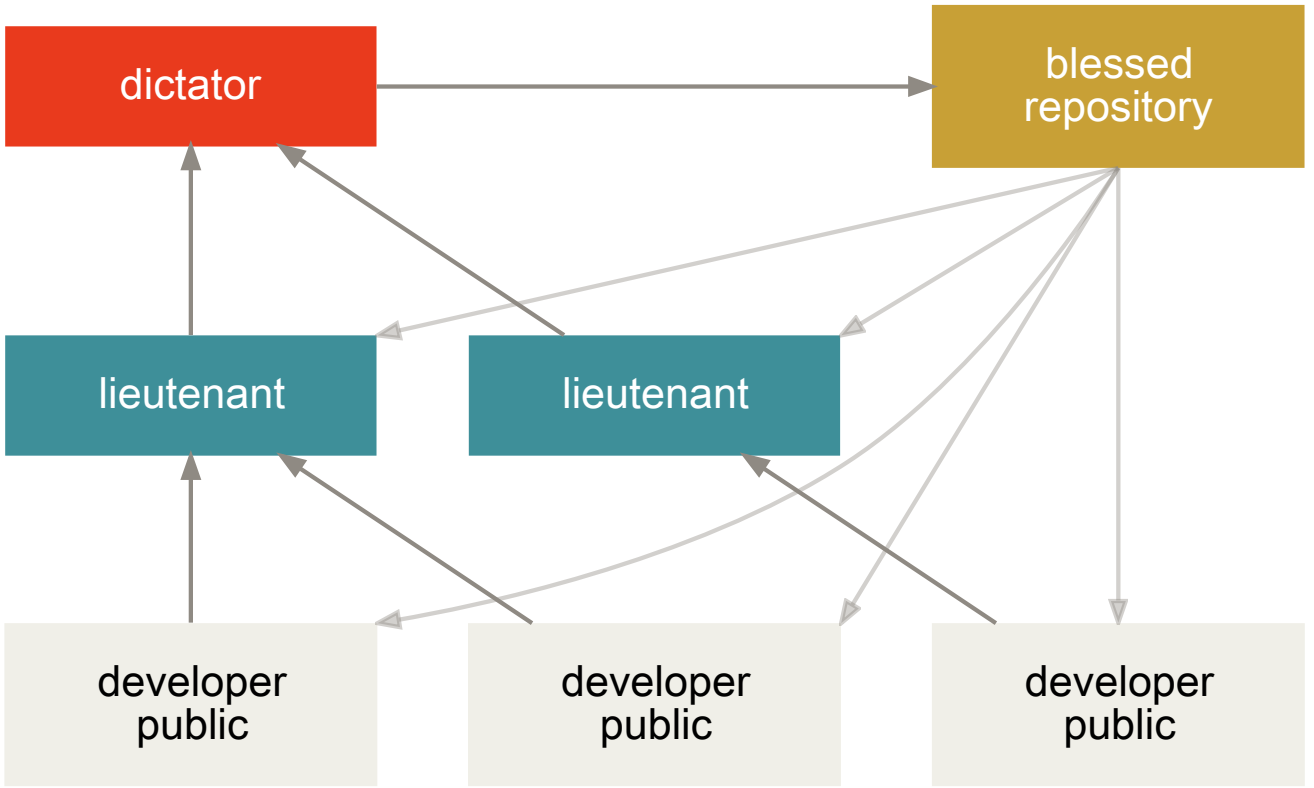
Workflows
Remote repositories in teams

Integration Manager:



Workflows
Remote repositories in teams

Benevolent Dictator:



Technical background

Technical background

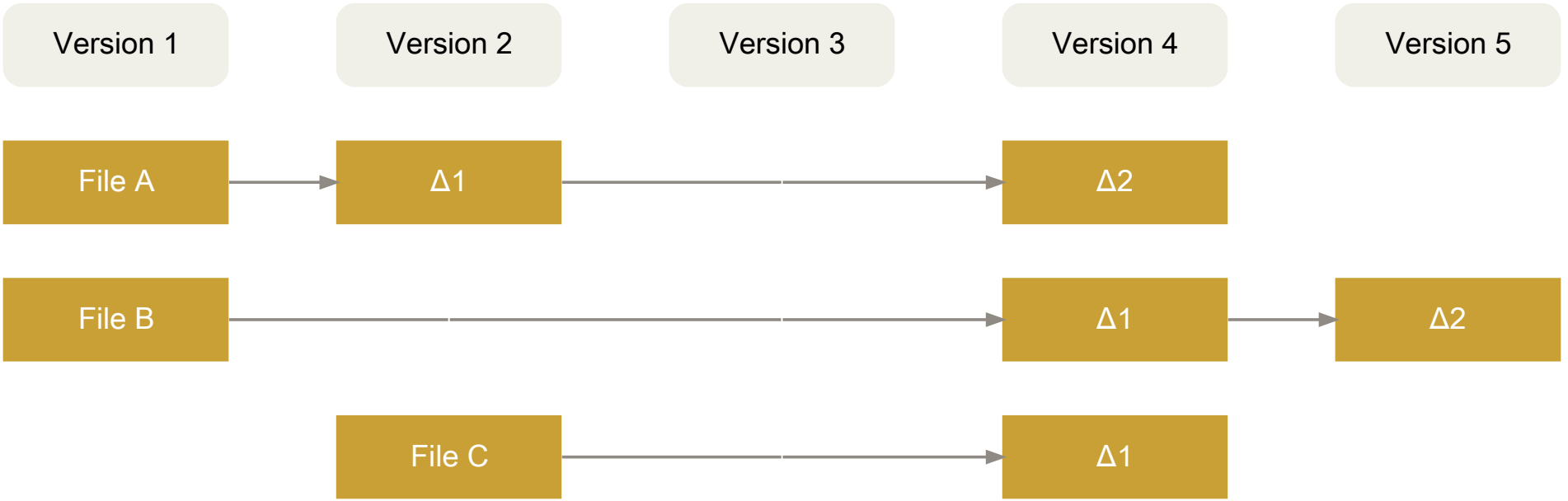
- Subdirectory ".git" contains whole repository
- If ".git" is deleted, the repository is deleted
- Remote is ".git" in project root

Technical background
Plumbing and porcelain

- Git has many low level commands
  - To manipulate the underlying data model
  - Because of history
  - Enables:
    - Messing up repository completely
    - Repairing a messed up repository
- User friendly "porcelain" commands came later

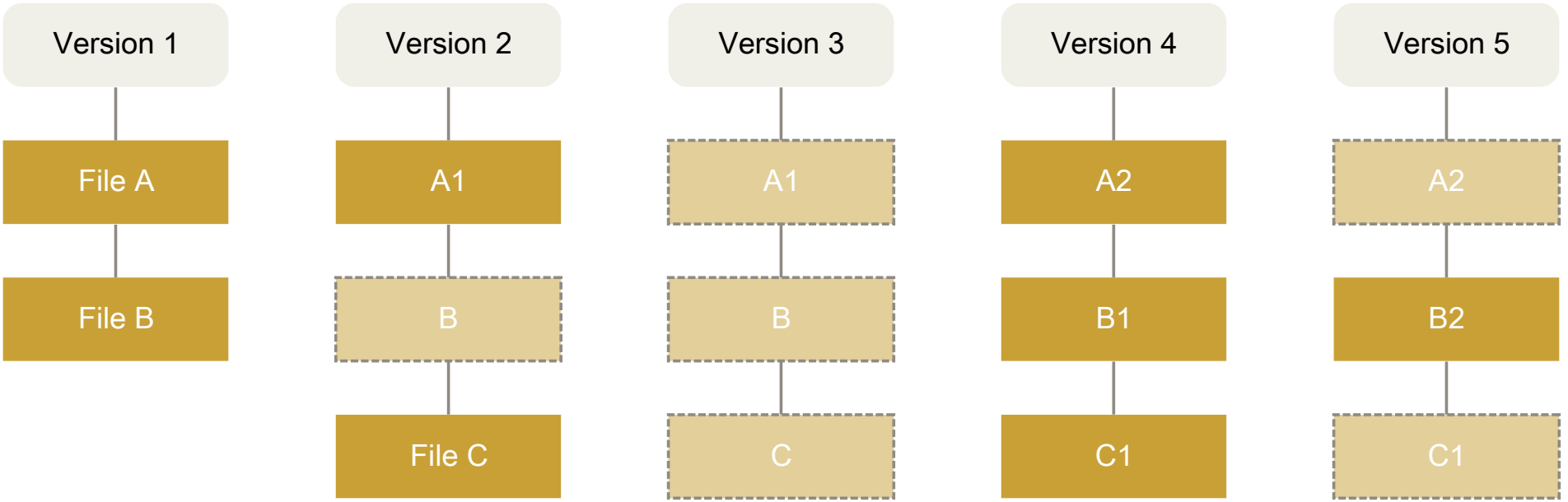
Technical background
Storing revision history

Possible:



Technical background
Storing revision history

Used by Git:



Technical background
Objects

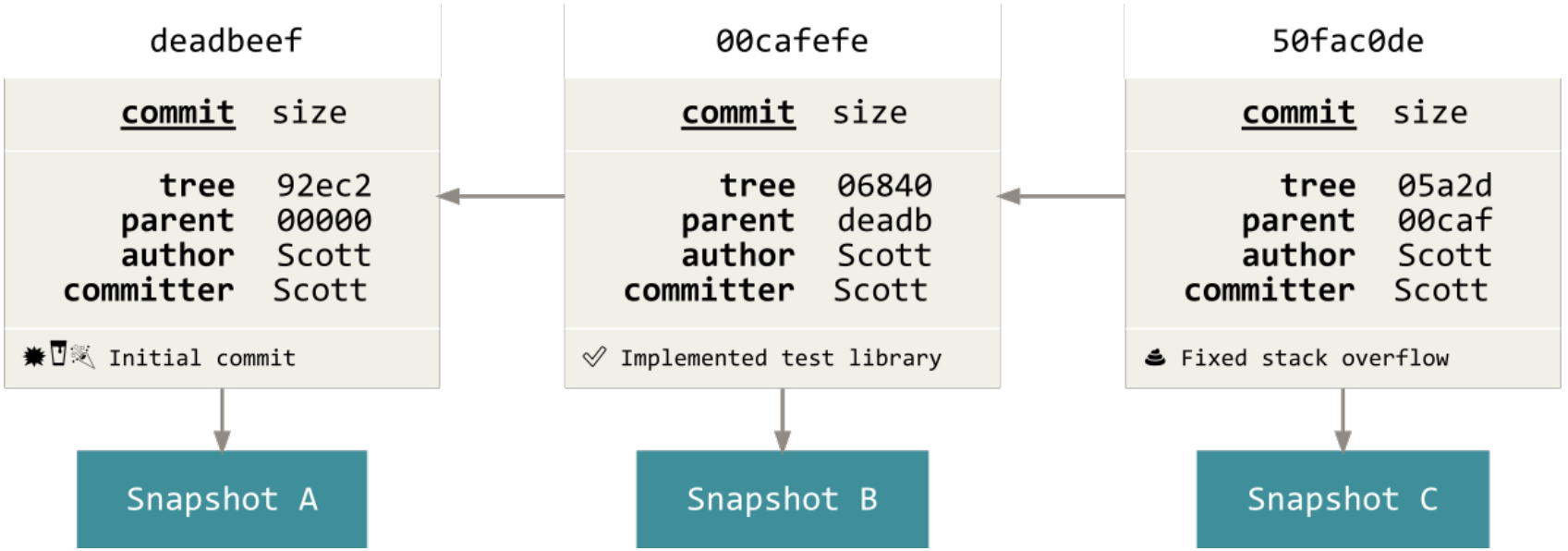
- Repository consists of "objects" and "pointers"
- Objects contain data addressed by SHA1 hash
- 4 object types:
  - Blob
  - Tree
  - Commit
  - Tag

Technical background
Commit objects

- Commits contain:
  - Pointers
    - Corresponding Tree object
    - Parent commit
  - Author
  - Committer
  - Message

Technical background
Commit objects

Commits are linked to a directed acyclic graph



Technical background
Tree objects

- Represents contents of a directory
- Mapping between file names and object hashes
- Contains pointers to:
  - Tree objects (sub trees)
  - Blob objects

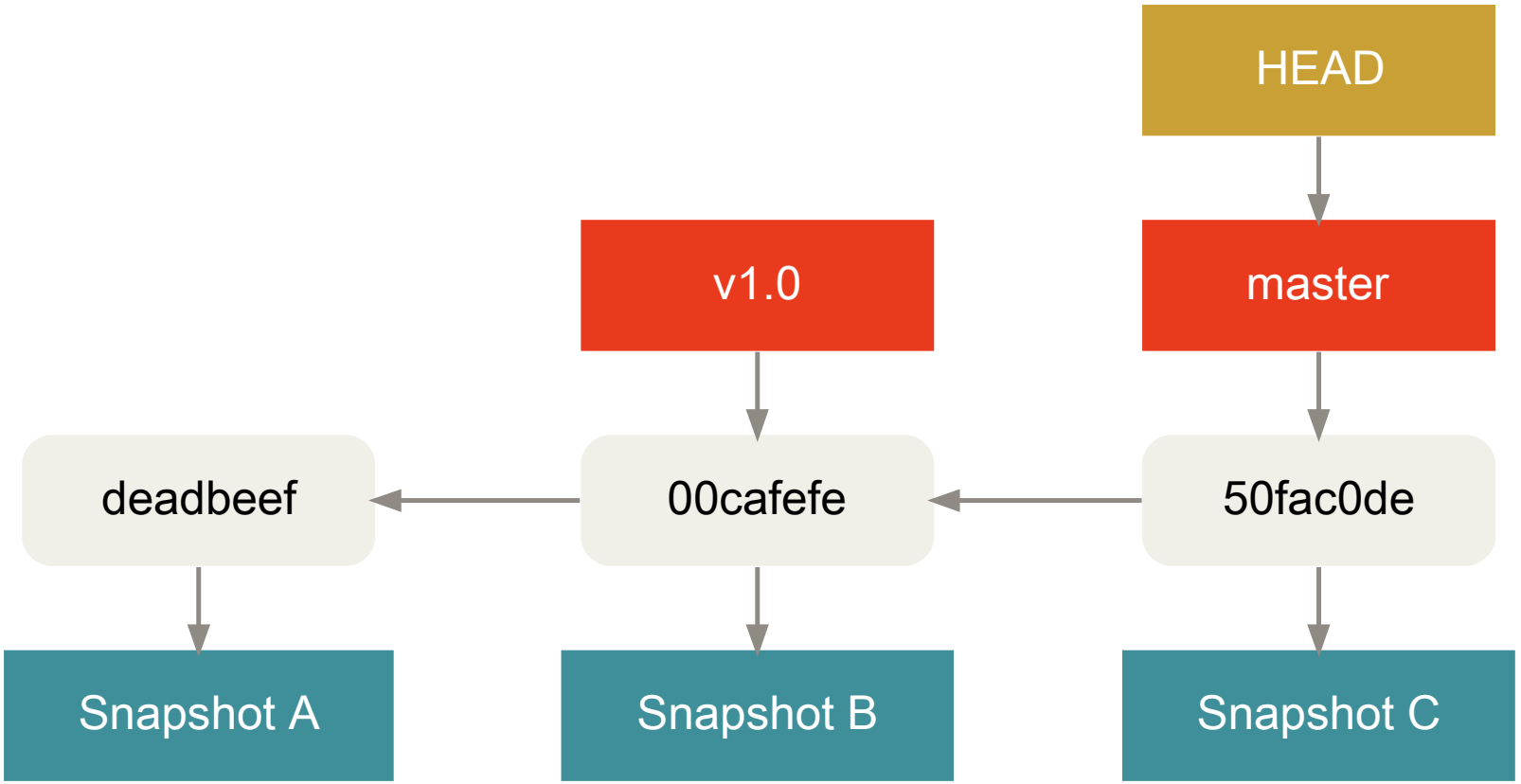
Technical background
Tag objects

- To mark an object as trusted by Tag creator
- Contains:
  - Pointer to another object
  - Type of pointed object
  - Tag creator
  - Tag message
  - Optionally GPG signature

Technical background
Pointers, pointers, pointers

- Cumbersome to operate with SHA1 hashes
- Multiple pointers in every repository
- Pointers are manged automatically ususally

Technical background
Pointers, pointers, pointers



Technical background
Pointers, pointers, pointers

- HEAD pointer
  - "Current" object in the repository
    - Directly or (usually) indirectly
  - Working copy is compared against "current" object
  - "git checkout" manipulates HEAD pointer

Technical background
Pointers, pointers, pointers

- Detached HEAD pointer
  - HEAD points to a branch pointer usually
  - Possible to "checkout" everything, e.g. commits, tags
  - All things possibly:
    - Create commits
    - Create branches
  - Beware of making branchless commits

Technical background
Pointers, pointers, pointers

- Branch pointer
  - Points always to a commit
    - Considered last commit of branch
  - As many branch pointers possible as desired
  - Deleting a banch pointer is trivial
    - If commits are still reachable via other branch pointers
    - If not:
      - Deletion has to be forced
      - Commits can be lost

Technical background
Pointers, pointers, pointers

- Tag
  - Object with some properties of a pointer
  - Points to considered trustworthy object
  - HEAD can point to it
    - It is detached

Addtional Tools

- Git Extensions
  - GUI for Git
- posh-git
  - Extension for PowerShell
- BFG Repo-Cleaner
  - Clean up repository history