git basics

Learning-Catalogue code: 067689

Sébastien SNAIDERO

Software product owner
& senior developer
@ TDS / PDK & Design Flows

life.augmented

Agenda

DAY 1

- Introduction
  - About version control
  - Why Git ?
- Install & configure Git
- Work with local repo
- Branching
  - Branches
  - Merge
- Work with distant repo
- Rebasing

DAY 2

- Day 1 debrief
- GUI tools
- Additional commands
  - Tagging
  - Patching
  - Debugging
  - Stashing
  - Ignoring files
  - Delivering
- Workflows
- Best practices

Introduction

About Version Control

Why ?

- Source code tracking and backup
  - Version control software records text files changes over time
  - Change history is saved
    - It can recall each specific version
    - It compares changes over time

➔ No mistake penalty, the recover is easy

➔ Encourage trials, rollback is easy

➔ Enforce consistency, changes overview available

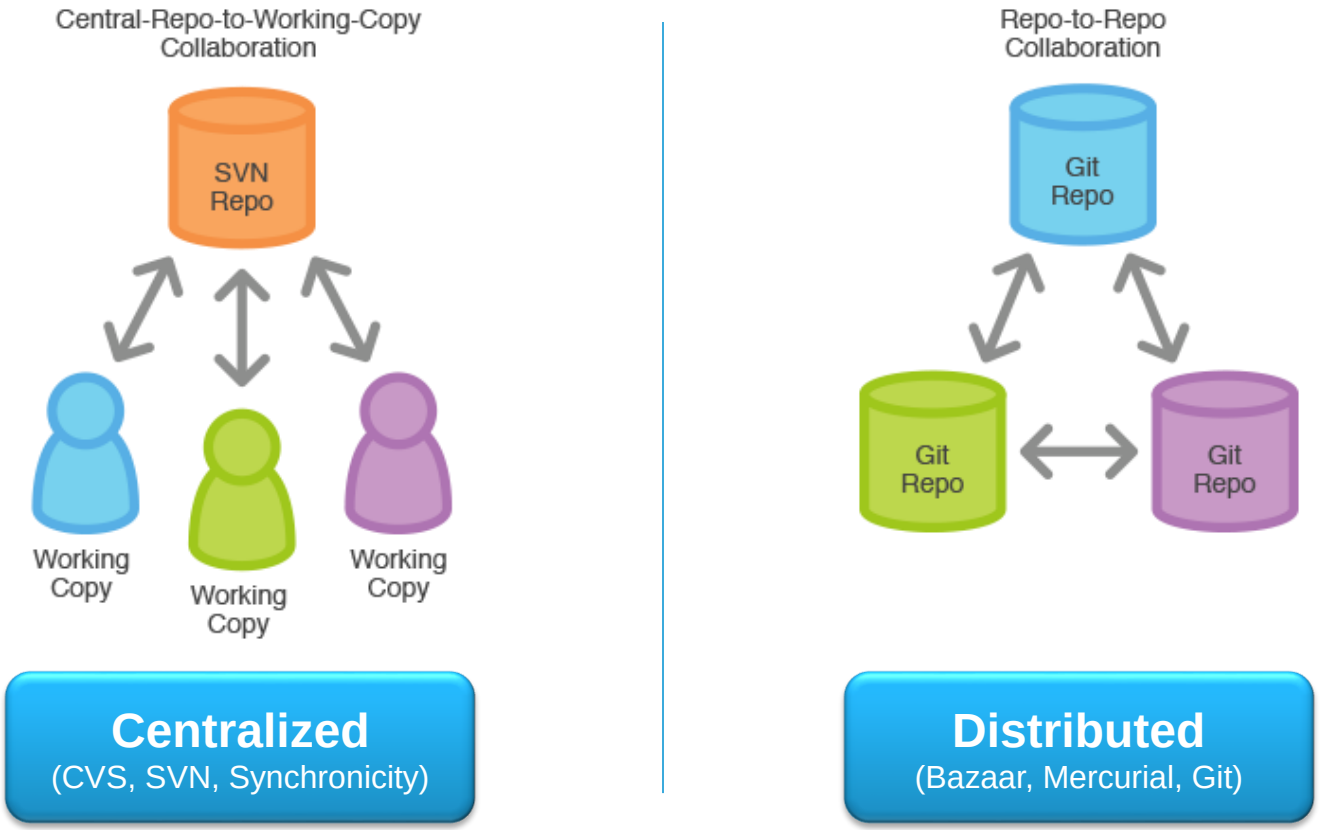
- Helps collaboration
  - Allows the merge of all changes in a common version

➔ Every body is able to work on any file at any time

About Version Control

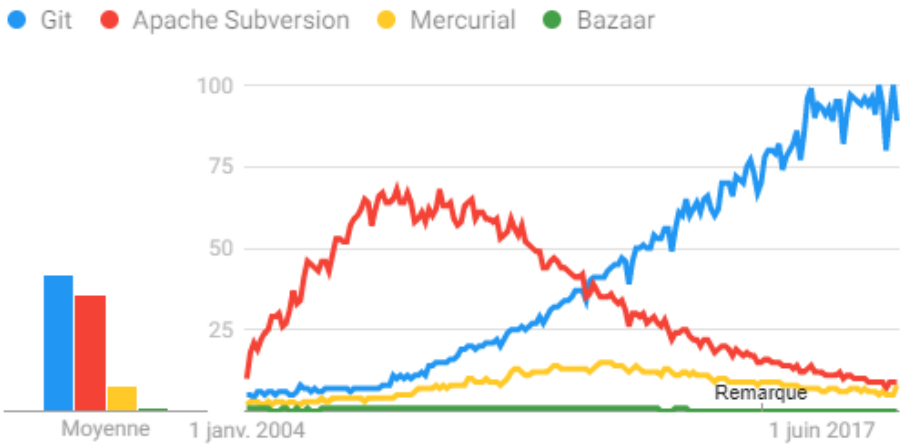
How ?

- There are two types of **V**ersion **C**ontrol **S**ystems



Why Git

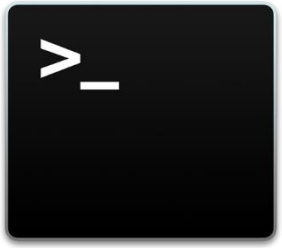
- Open source
- Git is designed for
  - Speed
  - Simple design
  - Massive branching usage
  - Fully distributed
  - Able to handle large projects



- Git is distributed version control system
  - You work locally on the complete copy with the complete history of the project
    ➔ Every operation is done locally : fast & can be done offline
- Many major open source projects use Git :
  - Linux Kernel, Fedora, Android, VLC, Twitter, NodeJS, ...
- Integrated in many IDEs
  - Eclipse, VSCode, PyCharm, Atom, ...

The command line

- Git is originally provided as a ***command line interface***
- There are many ***graphical user interfaces*** of varying capabilities
- Only in ***command line*** you can run **all** Git commands
  - If you know how to run the command line
    ➔ you can figure how to run the GUI
  - The opposite is not necessary true
- Graphical client is a matter of ***personal taste*** and ***environment***



`We will train on terminal`

- Everything in Git is subcommand of **`git`**
  - `git init, git clone, git add, git commit, git push, ...`

Basics

Objective

At the end of this module, you will be able to

- Configure Git
- Start a local project
- Start from a remote project
- Commit your changes

Install & configure Git

- Git install
  - VNC / LSF: sw git 2.18.0
  - Windows: http://git-scm.com/download/win
  - Ubuntu: sudo apt-get install Git
- Git configuration
  - Single command `git config` to get and set configuration setup
  - Configuration setup stored in giconfig files that can be stored in three places

OS	System (--system)	User (--gloal)	Project (--local)
Linux	<install>/etc/gitconfig	~/.gitconfig ~/.config/git/config	.git/config
Windows	<install dependent>	%USERPROFILE%\.gitconfig	.git/config

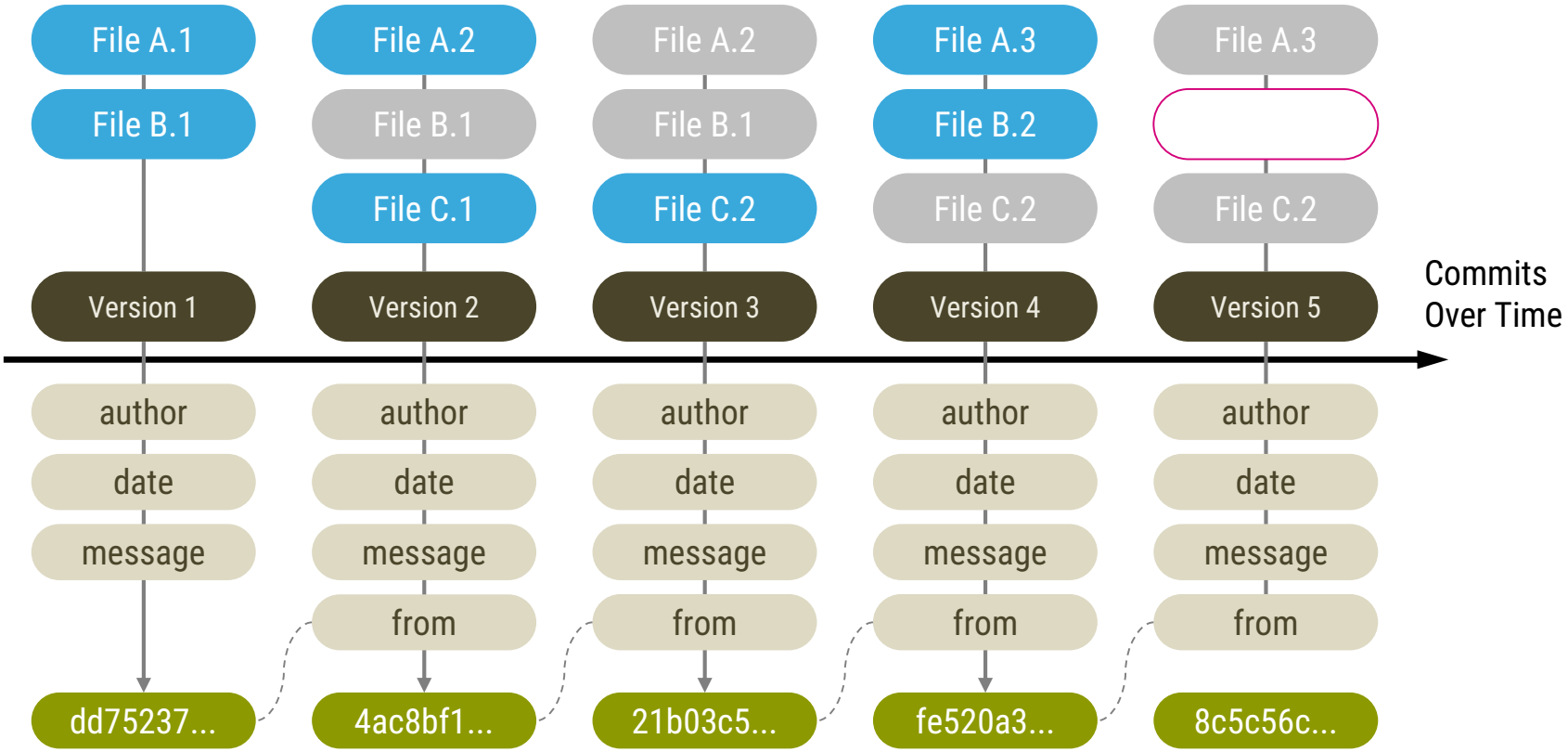
- Initial configuration
  - `git config --global user.name "<First> <Last>"`
  - `git config --global user.email "first.last@st.com"`
  - `git config --global core.editor vi`

Lab 0

Git & Training Setup

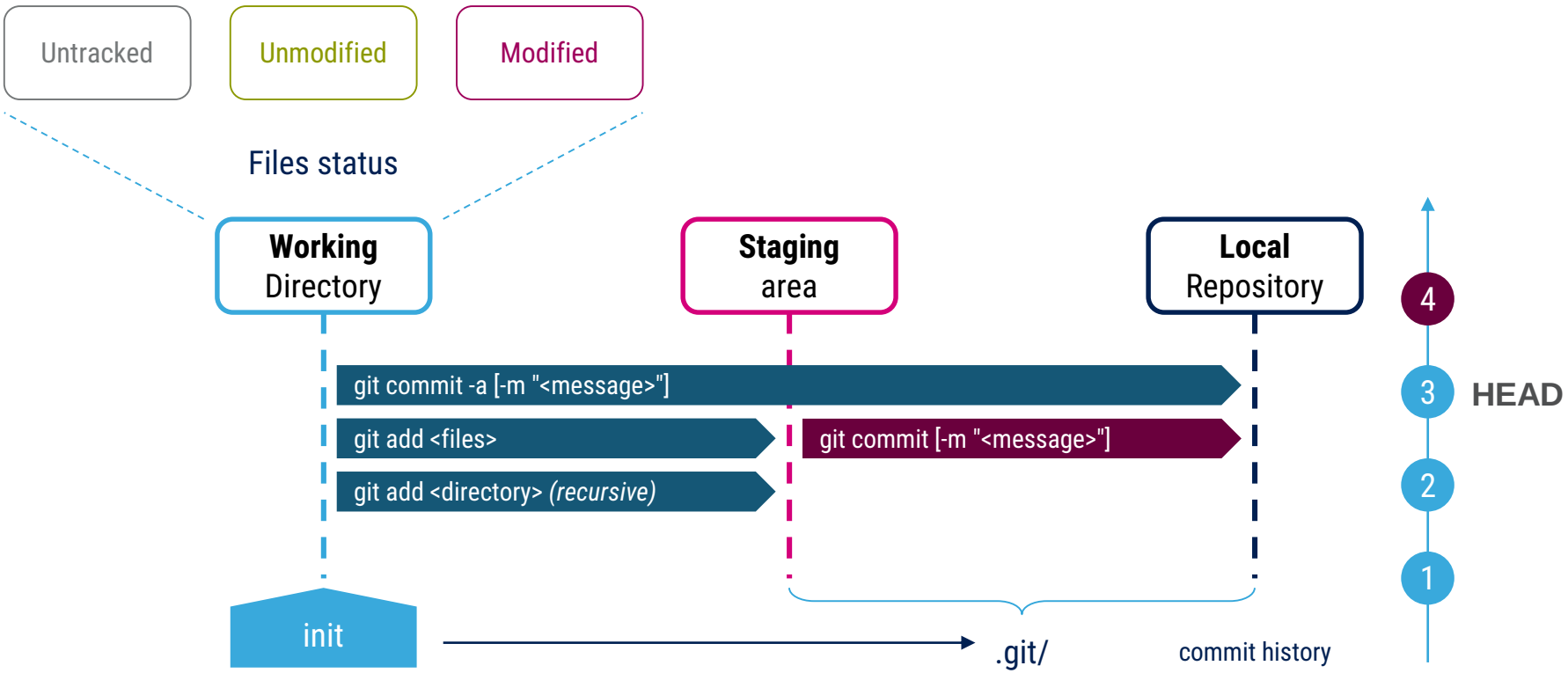
Versioning in Git

- Snapshots, not differences ➔ Speed, branching
- Mostly add data ➔ Things never lost without confirmation



SHA-1 *[ShaOne]* (unique 160 bits / 40 hexadecimal characters checksum for version & integrity)

Working locally



Commit message

- On commit an editor will be open to let you provide a mandatory message about your commit
- Anatomy of the commit message

```
HEADER - Short single line description

More detailed explanatory plain text, if
necessary (optional). This BODY area must be
separated from HEADER with a blank line if
used in the commit message.

Once entered in the BODY area, you can:
- Insert other blank lines
- Structure your text with sections

It can be as long as needed.
```

HEADER (mandatory)

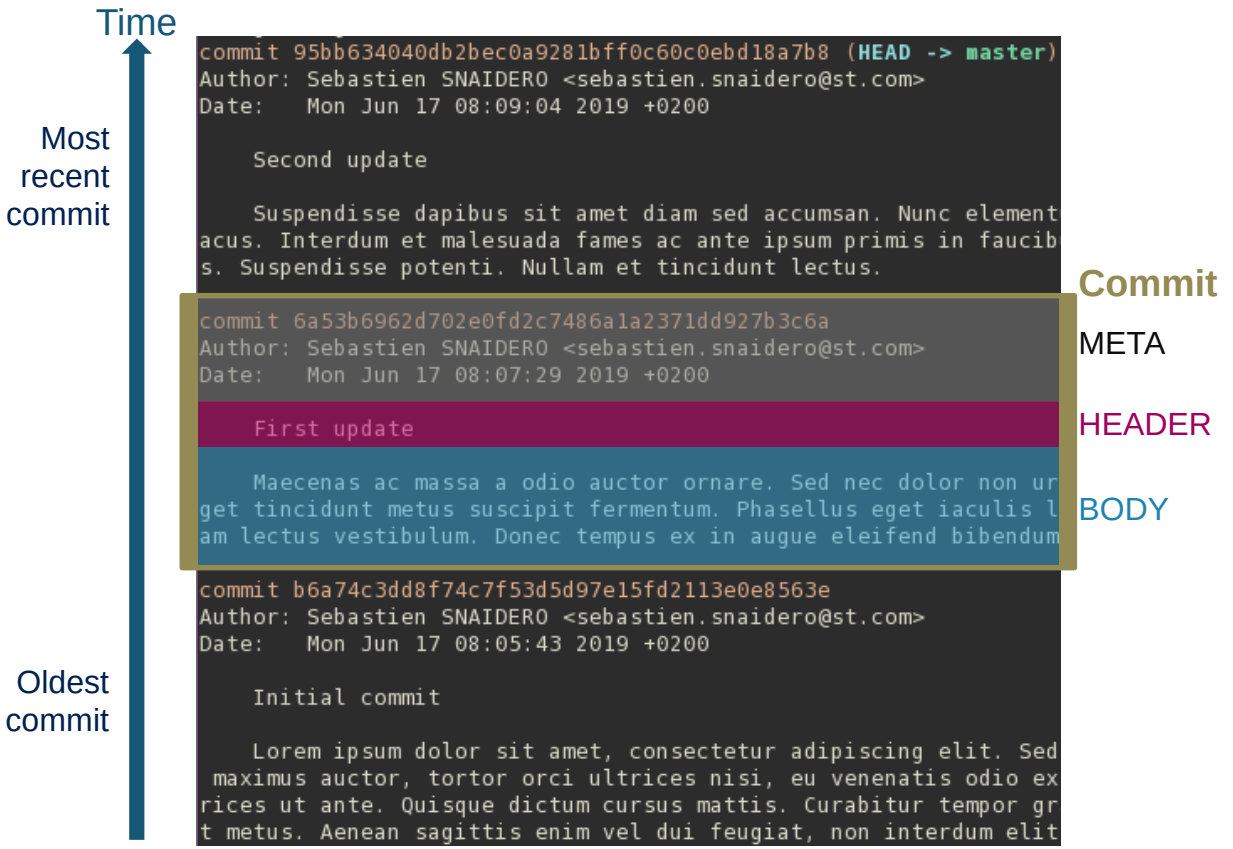
BLANK LINE (mandatory in the BODY)

BODY (technically optional)

- `git commit -m "<message>"` will create a commit message with a single Header line containing <message> without opening an editor

Explore commits' history

- Get the full history
  - with `git log`

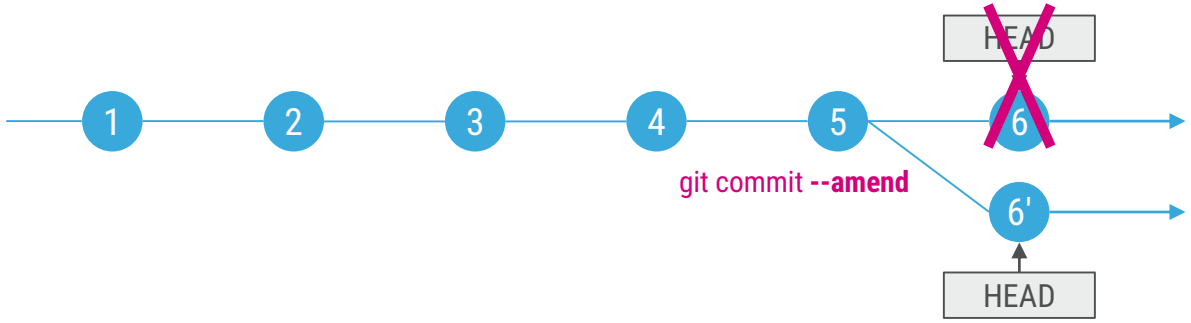


- Get the summary history
  - with `git log --oneline`

```
95bb634 (HEAD -> master) Second update
6a53b69 First update
b6a74c3 Initial commit
```

Update your last commit
git commit --amend

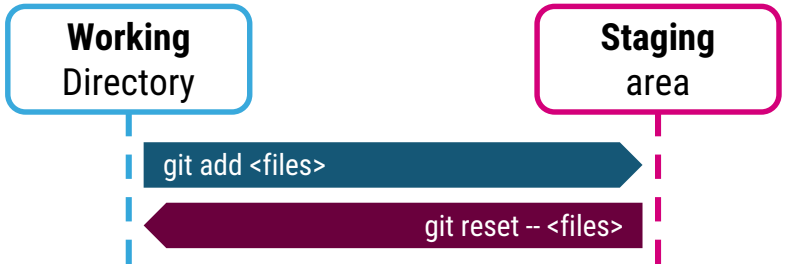
- Re-create the last commit
  - with the content that may have been added to the staging area
  - with edition of the commit message



- Even with no change at all, this will create a new commit (with a new SHA1) as at least commit date will change

Undo & remove (1/2)

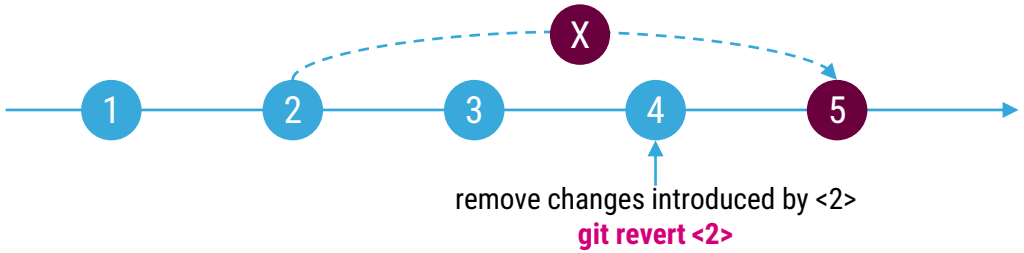
- Unstage changes with `git reset -- <file>`
  - All changes send to staging area are sent back to working directory
  - The exact opposite of `git add`



- Discard local changes with `git checkout -- <file>`
  - All changes in Working directory are ***lost***
  - You can interactively select what to discard using `-p` option

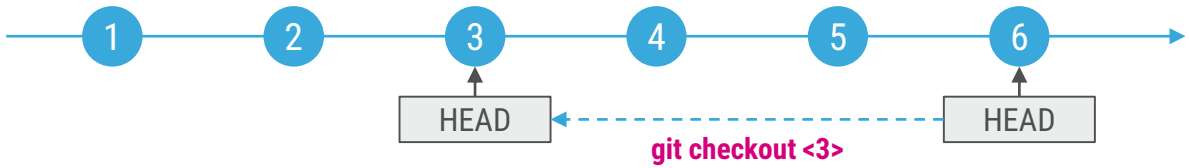
Undo & remove (2/2)

- Remove untracked files with `git clean`
  - As destructive command, it needs to be forced with `-f` to perform
  - Can perform a dry-run to check what would be done using `-n`
  - By default removes only files, to remove directories also use `-d`
- Revert a commit with `git revert [-n] [<commit>]`
  - Creates a new commit, nothing is removed from history
    - Keep history safe (mandatory if shared)
  - If <commit> not specified defaults to HEAD, reverting last commit
  - `-n` will perform dry-run to let you know what would be done without executing



Move across history

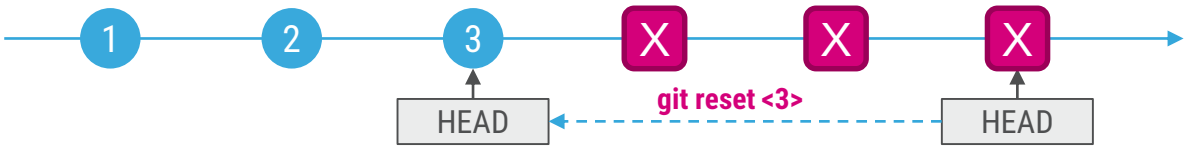
`git checkout <commit>`



- Align the content of your Working directory with a given commit
  - move HEAD in the hierarchy of commits
  - view all the files as they were at this point
- Requires your working directory to be clean
  - no modified or deleted files in working directory or staging area
  - Using `--force|-f` will remove any of the local changes (working & staging)

Roll back history
`git reset <commit>`

- Reset moves ***HEAD*** & ***branch*** in the hierarchy of commits, destroying the history



- Depending on the options, will have different effects
  - `--hard`
    - ***Everything is lost***, your ***working*** & ***staging*** areas are fully synchronized with <commit>
  - `[--mixed]` (this is the default)
    - The ***staging*** area is ***cleared***, but all the ***changes*** introduced along the reset history are ***kept in*** your ***working*** directory, so you can edit & create new commit from it
  - `--soft`
    - The ***working*** area is ***not touched*** & all the ***changes*** introduced along the reset history are ***stored in*** your ***staging*** area, so you can still update & commit from it

What did I do ? What can I do ?
`git status`, `git diff`

- View the textual diffs you are introducing
  - `git diff` shows differences between ***Working*** & ***Staging*** areas
  - `git diff --staged` show differences between ***Staging*** area & ***Local*** repository
  - Options
    - `--word-diff` switch to inline diff with `[-...-]` & `{+...+}`
    - `--color-words` inline mode without delimiter `removed` & `new`
      - May look more friendly to read, but limitation to see space changes
    - `-b` ignores white space amount between words & trailing spaces
- Get the status of ***Working*** & ***Staging*** areas & help on capabilities
  - `git status` is an ***all-in-one*** command you must rely on to know what you can do
  - It show ***untracked***, ***modified***, ***deleted***, ***staged*** files
  - It tells you on which branch you are *(master)*
  - It provides you contextual help that guides you through the different actions you can perform from the state of your workspace

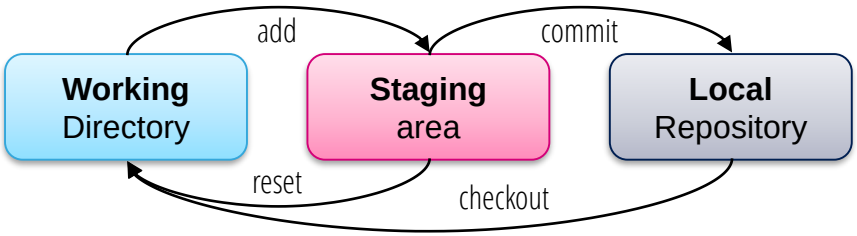
Lab 1

Using Git with Local Repo

git essential commands

Create repository

Init or create <dir> as Git repo (defaults to '.')
$ git init [<dir>]



Make changes

List actions, changed & new files in local repo
$ git status

Show diffs on tracked files in working dir
$ git diff [<file>]

Show staged diffs that will be commited
$ git diff --staged [<file>]

Stage given file(s)
$ git add <file> [<file> ...]

Stage all updates in directories
$ git add <dir> [<dir> ...]
$ git add .

Stage all updates in working directory
$ git add -A

Commit staged changes to local repo
$ git commit [-m "<commit_message>"]

Update current commit
$ git commit --amend

Revert changes

Remove any local change in <file>
$ git checkout -- <file> [<file> ...]

Unstage <file>, keeping changes
$ git reset <file> [<file> ...]

Revert to <commit>, keeping changes
$ git reset <commit>

Revert to <commit>, loosing changes
$ git reset --hard <commit>

Explore history

Show all changes applied in <commit>
$ git show <commit>

Show current branch (entire with --all) history
$ git log [--all] [--graph] [--oneline]

Branching

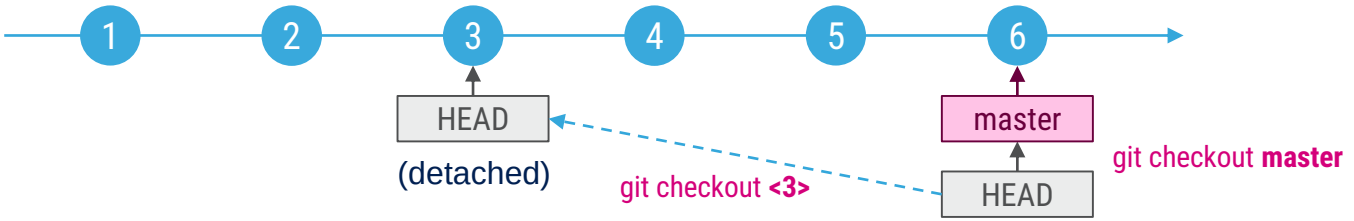
Objective

At the end of this module, you will be able to

- Create a branch
- Switch between branches
- Merge your branch
- Rebase your branch
- Resolve conflicts

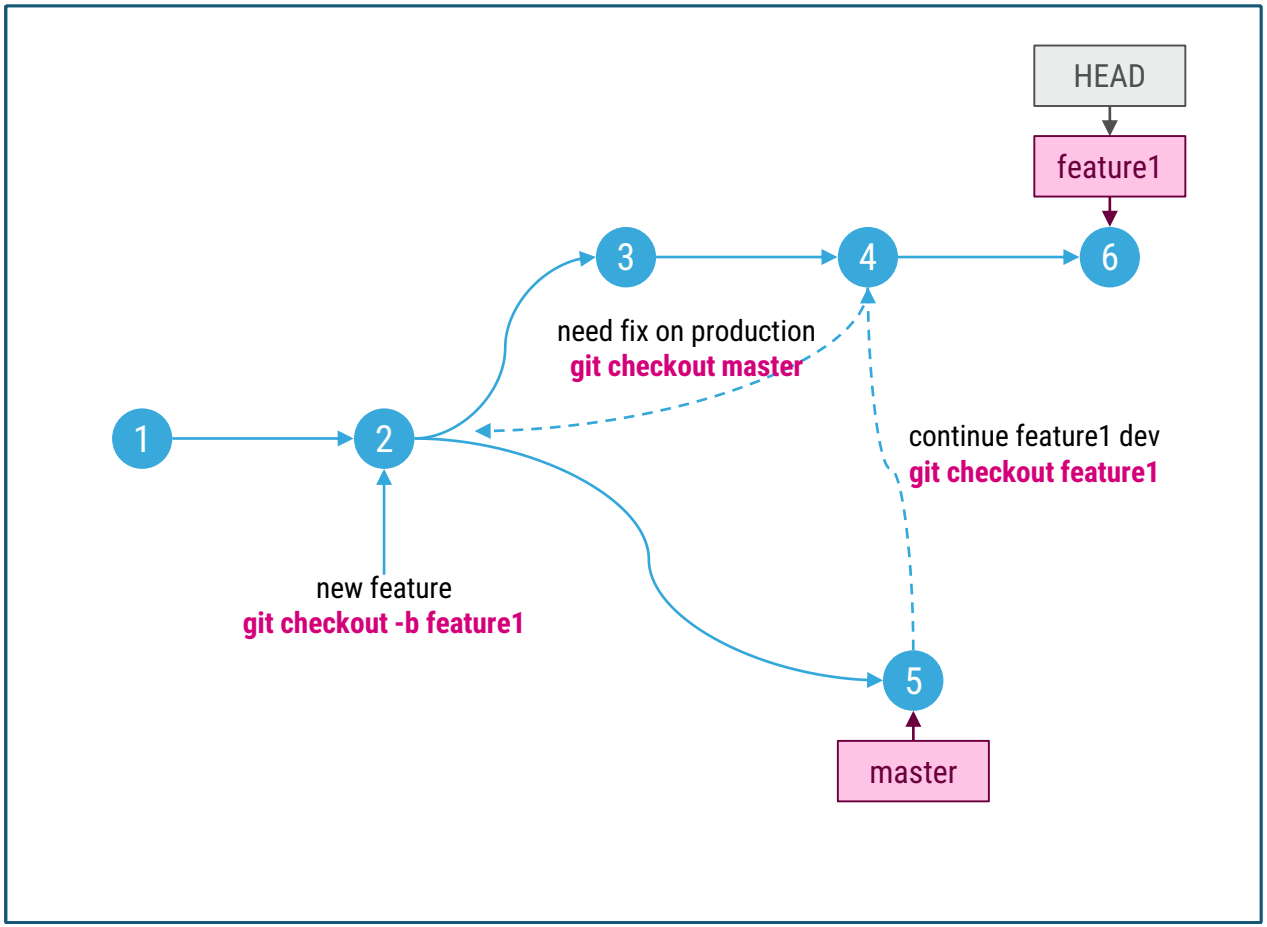
What is a branch ?

- A branch is a ***reference*** pointing to a ***commit***
  - Default branch is **master**
  - **HEAD** is the current commit in the workspace
    - Usually attached to a branch
    - Can be DETACHED



Create & switch branches

- Collaborate & synchronize on team projects
- Switch between features development, fixes & support



Explore branches in commits' history

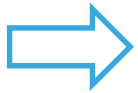
- Get the visual path of branches & merges in parent history
  - with `git log --oneline --graph`

```
c265e4e (HEAD -> develop) last update
a21fb45 Merge branch 'develop' into tmp
806377e Third update
6a53b69 First update
b6a74c3 Initial commit
```

```
* c265e4e (HEAD -> develop) last update
*   a21fb45 Merge branch 'develop' into tmp
|\
| * 806377e Third update
|/
* 6a53b69 First update
* b6a74c3 Initial commit
```

- Get the visual path of branches & merges in all branches
  - with `git log --oneline --graph --all`

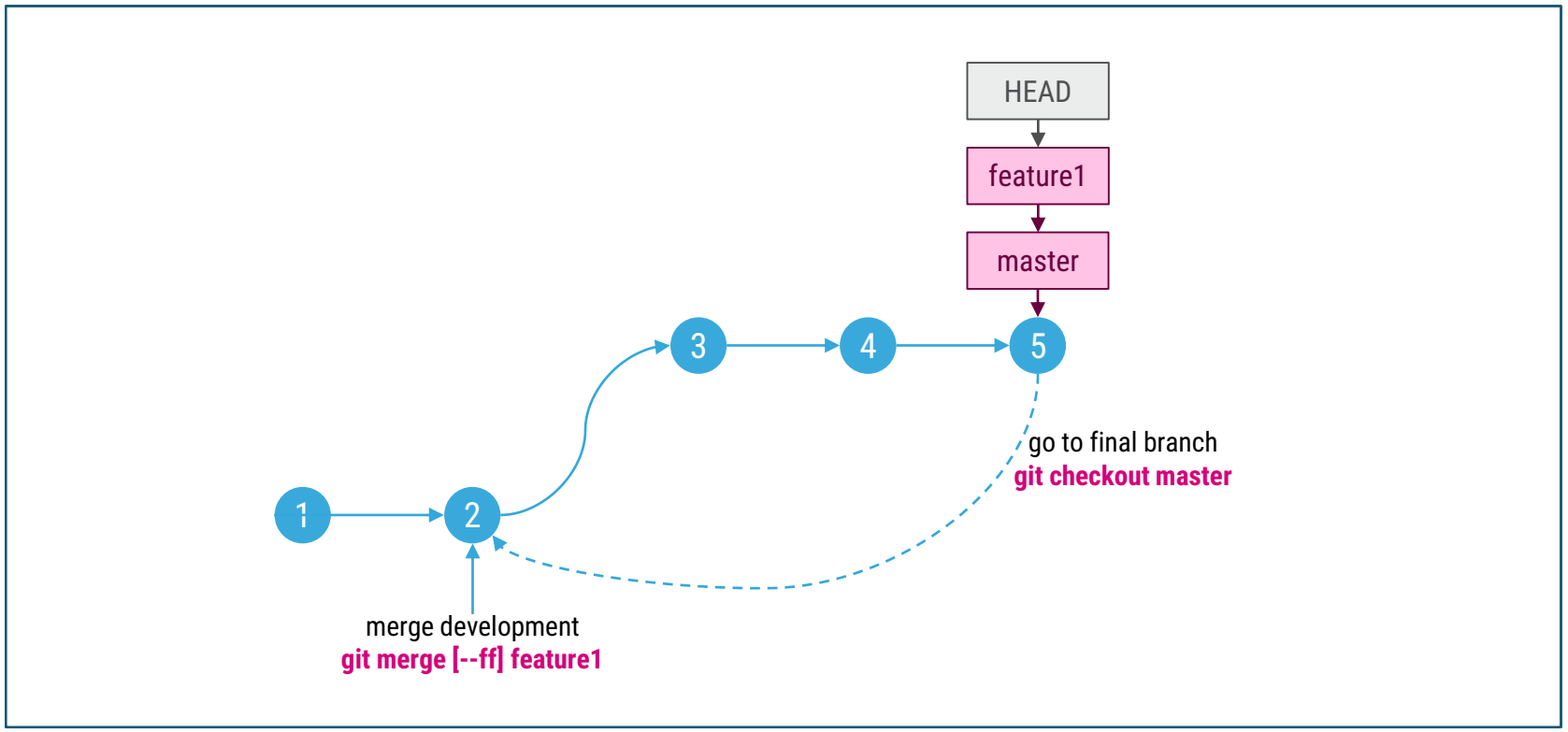
```
* c265e4e (HEAD -> develop) last update
*   a21fb45 Merge branch 'develop' into tmp
|\
| * 806377e Third update
|/
* 6a53b69 First update
* b6a74c3 Initial commit
```



```
* c265e4e (HEAD -> develop) last update
| * 7f83792 (master) Second update
|/
*   a21fb45 Merge branch 'develop' into tmp
|\
| * 806377e Third update
|/
* 6a53b69 First update
* b6a74c3 Initial commit
```

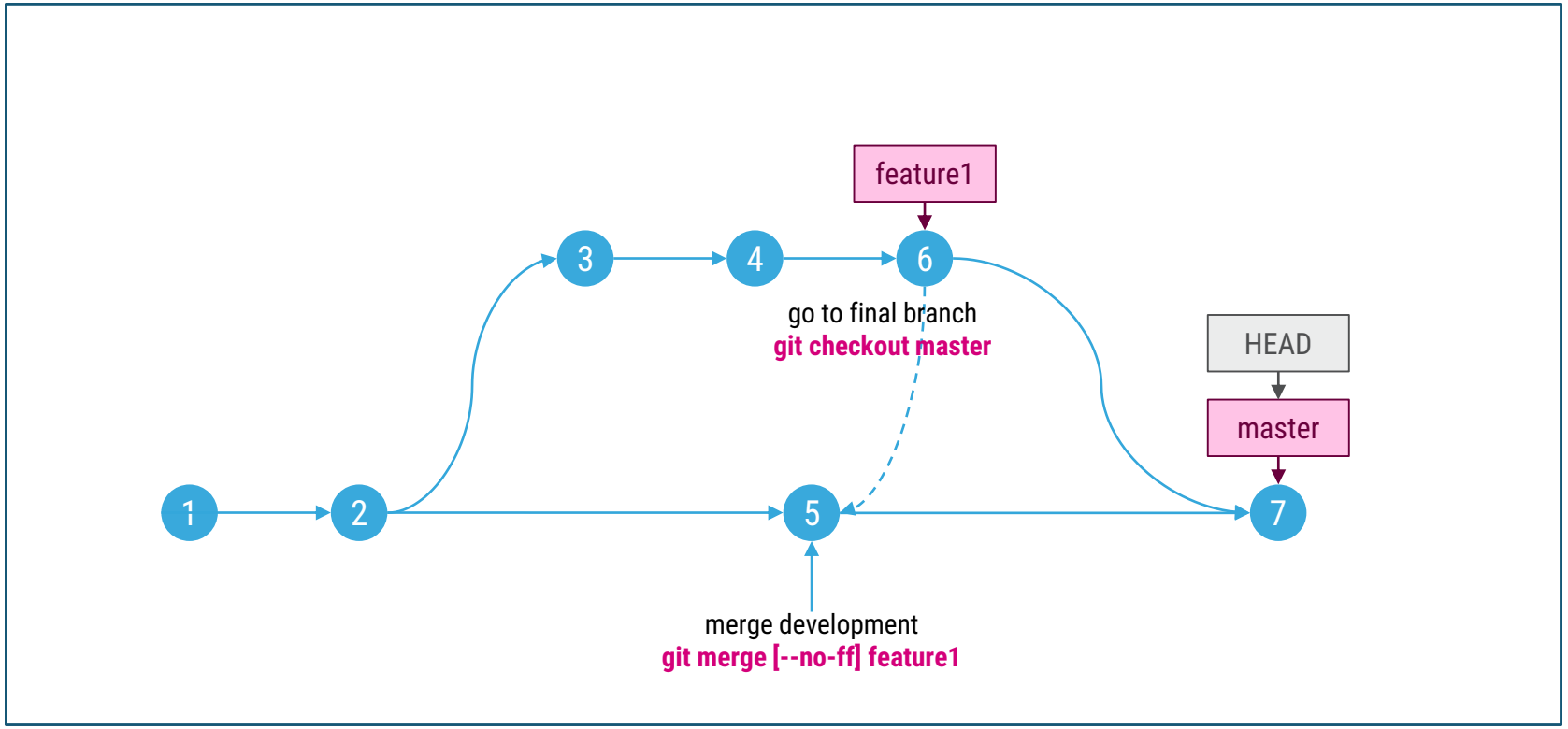
Merge branches
fast forward

- Default when possible



Merge branches
with new commit (no fast forward)

- Automatically used when not possible to fast forward



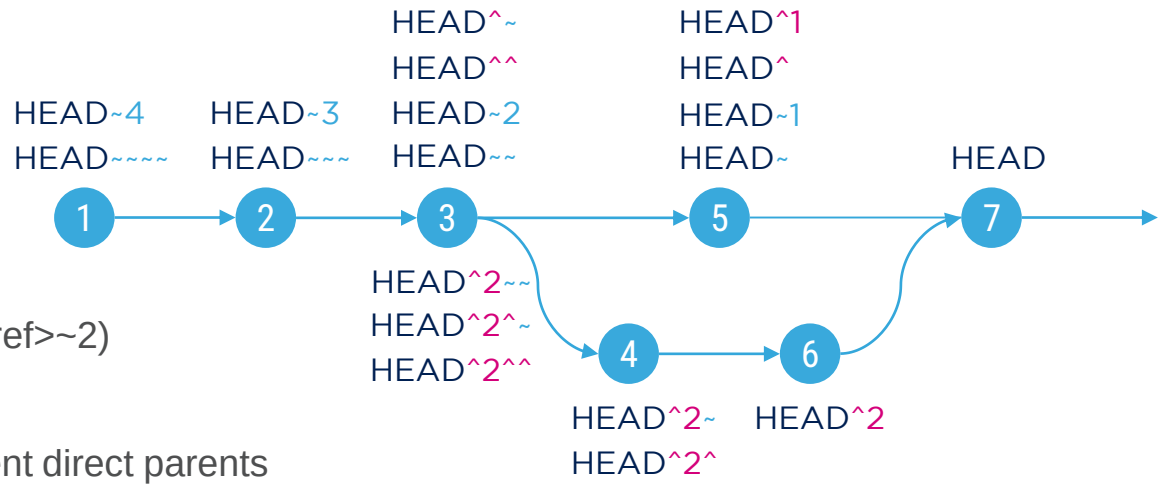
- You can force this behavior when fast forward would be applied
  - Use `--no-ff` option in command line or set `merge.ff` to `false` in configuration

Lab 2

Branching & Merging

Referencing commits

- Short SHA1
  - 6-8 first characters are usually enough
    - 2^2N objects in project without collision (N length of short SHA1)
    - Linux Kernel project uses only 12 characters (875 000 commits, 7 millions objects)
- Named reference (branch, HEAD, tag, ...)
- Relative
  - <ref>~
    - Parent of <ref>
  - <ref>~#
    - Ancestor of <ref>
      (i.e. grandparent for <ref>~2)
  - <ref>^ / <ref>^#
    - Select amongst different direct parents
      (i.e. in case of merge)



<ref>~ = <ref>~1 = <ref>^ = <ref>^1

Merge branches
Solve conflicts (1/2)

- Two people changed the same piece of file
  - e.g. line deletion vs line edition

➔Git can't figure which update to select

```
$ git merge branch_to_merge
Auto-merging file.txt
CONFLICT (content): Merge conflict in file.txt
Automatic merge failed; fix conflicts and then commit the result.
```

- Use `git status` to list conflicts
- Fix merge conflict marked in each file

```
<<<<<<< HEAD
... HEAD branch code ...
=======
... Merged branch code ...
>>>>>>> merged-branch
```

- Mark resolved files as solved with `git add <file>`
- `git commit` to finalize the merge process

Merge branches
Solve conflicts (2/2)

- Merge process can be canceled using `git merge --abort`
- Merge conflict can show common ancestor code
  - Set configuration option `merge.conflictStyle` to `diff3`
  - This will present conflicts in the form

```
<<<<<<< HEAD
... HEAD branch code ...
||||||| merged common ancestor
... ancestor code ...
=======
... Merged branch code ...
>>>>>>> merged-branch
```

- Merge process can be assisted by GUI tool
  - git config --global merge.tool kdiff3
  - git mergetool

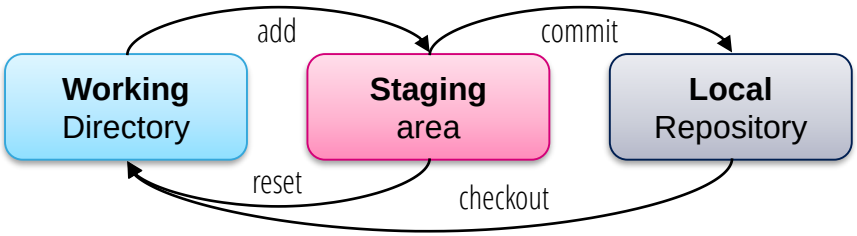
Lab 3

Solving Merge Conflicts

git essential commands

Create repository

Init or create <dir> as Git repo (defaults to '.')
$ git init [<dir>]



Make changes

List actions, changed & new files in local repo
$ git status

Show diffs on tracked files in working dir
$ git diff [<file>]

Show staged diffs that will be commited
$ git diff --staged [<file>]

Stage given file(s)
$ git add <file> [<file> ...]

Stage all updates in directories
$ git add <dir> [<dir> ...]
$ git add .

Stage all updates in working directory
$ git add -A

Commit staged changes to local repo
$ git commit [-m "<commit_message>"]

Update current commit
$ git commit --amend

Revert changes

Remove any local change in <file>
$ git checkout -- <file> [<file> ...]

Unstage <file>, keeping changes
$ git reset <file> [<file> ...]

Revert to <commit>, keeping changes
$ git reset <commit>

Revert to <commit>, loosing changes
$ git reset --hard <commit>

Branches

master default branch name
HEAD current point

Create <branch> at HEAD
$ git branch <branch>

Switch to <branch>
$ git checkout <branch>

Create and switch to <branch> at HEAD
$ git checkout -b <branch>

List all branches
$ git branch -a

Delete a local branch
$ git branch -D <branch>

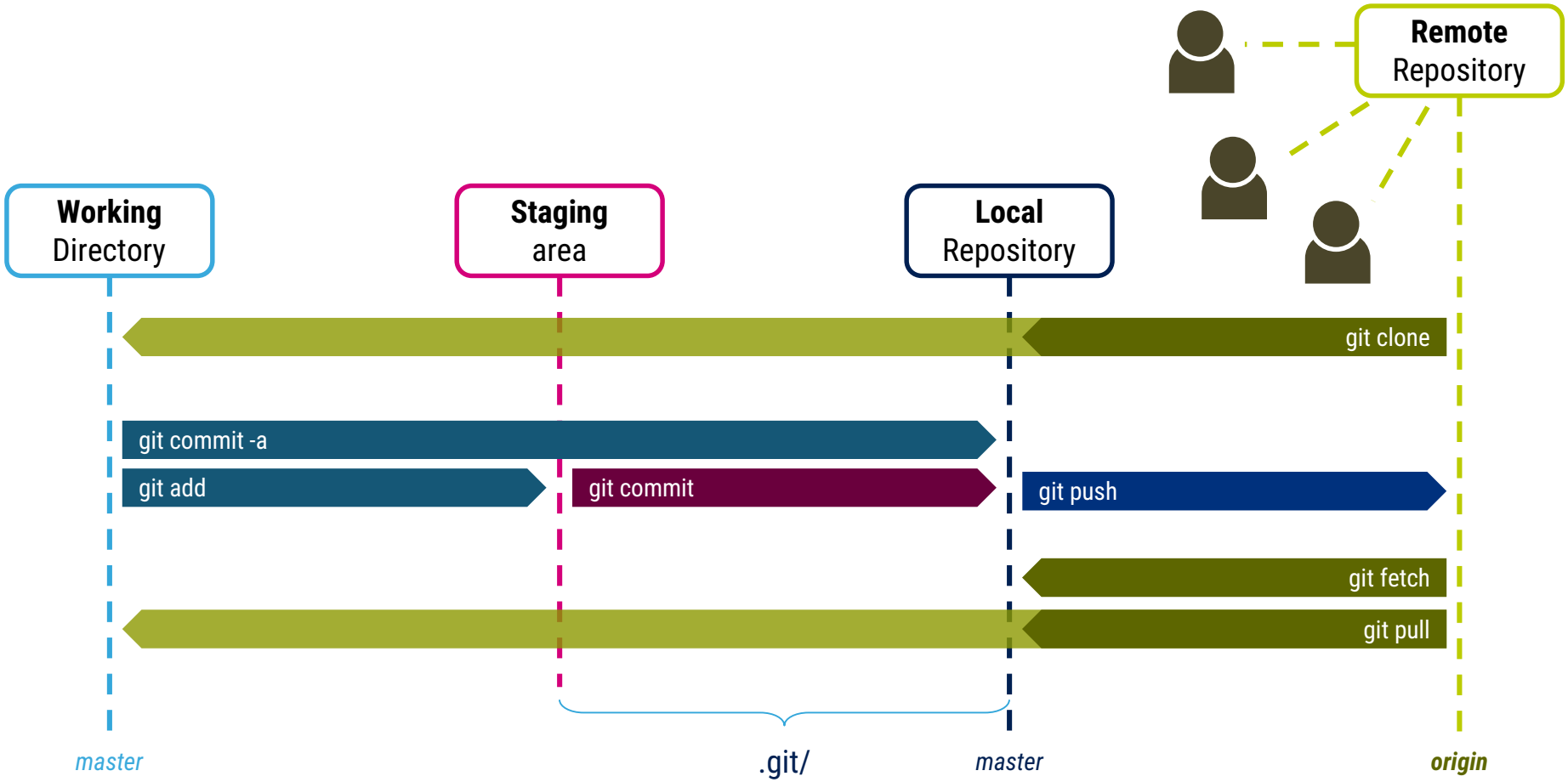
Merge <branch> into current branch
$ git merge <branch>

Explore history

Show all changes applied in <commit>
$ git show <commit>

Show current branch (entire with --all) history
$ git log [--all] [--graph] [--oneline]

Working with remote repository



Pull strategies

- Merge (default)
  - use `git pull` before `git push`

```
*   b97f4c7 (HEAD -> master, origin/master, origin/HEAD) Merge Pierre & Jimmy
|\
| *   d5fb0c1 Merge Pierre
| |\
| | * 7c0aafa Pierre
| * | 984cf99 Jimmy
| |/
* | 30098f9 Roberto
|/
* dac8414 snaidero: add files 1 & 2
```

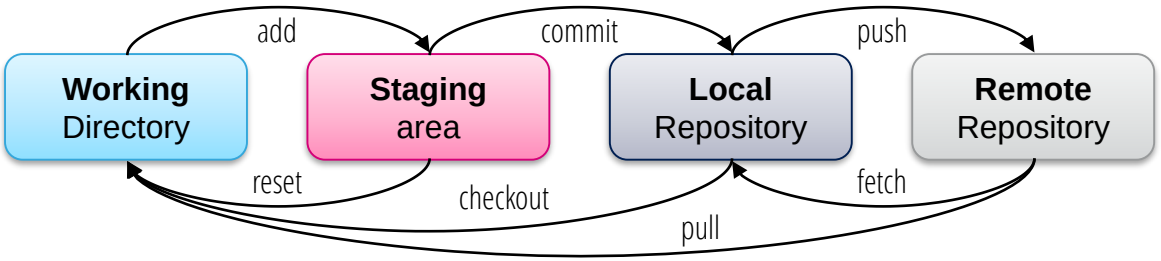
- Rebase
  - use `git pull --rebase` before `git push`
  - Set `git config --global pull.rebase preserve` for permanent behavior

```
* f1dddeb (HEAD -> master, origin/master, origin/HEAD) Roberto
* 87684b6 Jimmy
* 322f50e Pierre
* dac8414 snaidero: add files 1 & 2
```

Lab 4

Using Git with Distant Repo

git essential commands



Create repository

Init or create <dir> as Git repo (defaults to '.')
$ git init [<dir>]

Clone an existing repo
$ git clone <repo_url> [<local_repo_name>]

Make changes

List actions, changed & new files in local repo
$ git status

Show diffs on tracked files in working dir
$ git diff [<file>]

Show staged diffs that will be commited
$ git diff --staged [<file>]

Stage given file(s)
$ git add <file> [<file> ...]

Stage all updates in directories
$ git add <dir> [<dir> ...]
$ git add .

Stage all updates in working directory
$ git add -A

Commit staged changes to local repo
$ git commit [-m "<commit_message>"]

Update current commit
$ git commit --amend

Explore history

Show all changes applied in <commit>
$ git show <commit>

Show current branch (entire with --all) history
$ git log [--all] [--graph] [--oneline]

Revert changes

Remove any local change in <file>
$ git checkout -- <file> [<file> ...]

Unstage <file>, keeping changes
$ git reset <file> [<file> ...]

Revert to <commit>, keeping changes
$ git reset <commit>

Revert to <commit>, loosing changes
$ git reset --hard <commit>

Synchronize

Push local changes to <remote>
$ git push <remote> <branch>

Get changes in <remote> (no merge)
$ git fetch <remote>

Get changes in <remote> & merge
$ git pull <remote> <branch>

Get all available remotes with URLs
$ git remote -v

Add a new remote repo
$ git remote add <name> <url>

Branches

master	default branch name
origin	default remote name
HEAD	current point

Create <branch> at HEAD
$ git branch <branch>

Switch to <branch>
$ git checkout <branch>

Create and switch to <branch> at HEAD
$ git checkout -b <branch>

List all branches
$ git branch -a

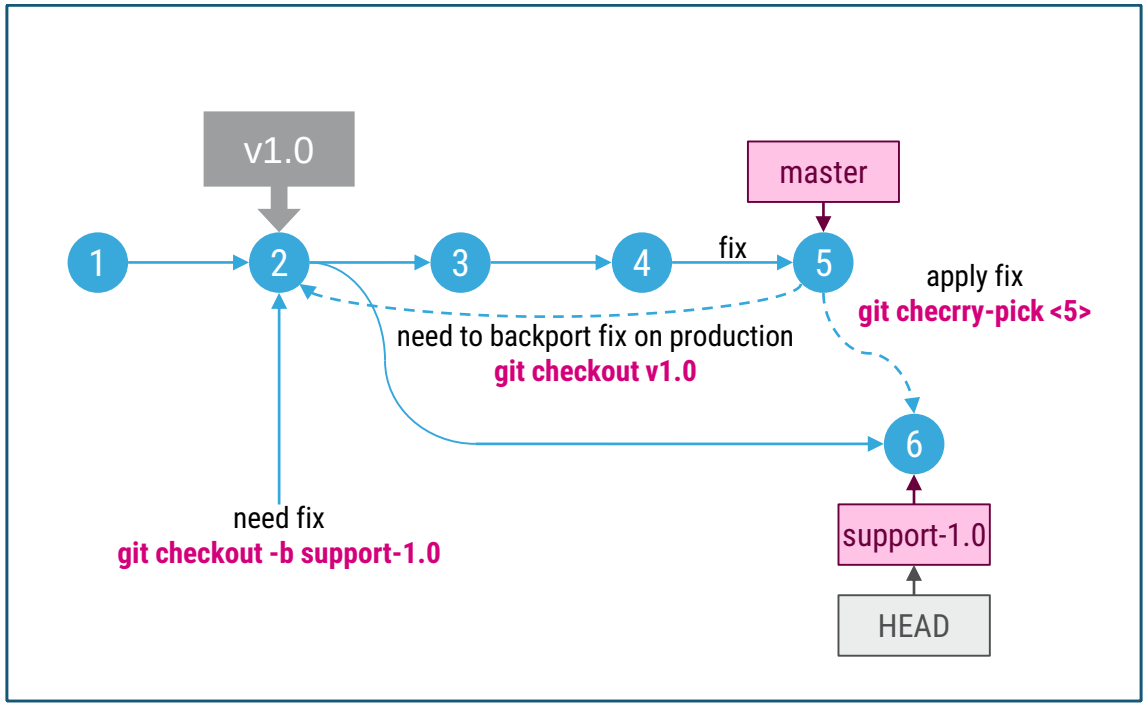
Delete a local branch
$ git branch -D <branch>

Delete a remote branch
$ git push origin --delete <branch>

Merge <branch> into current branch
$ git merge <branch>

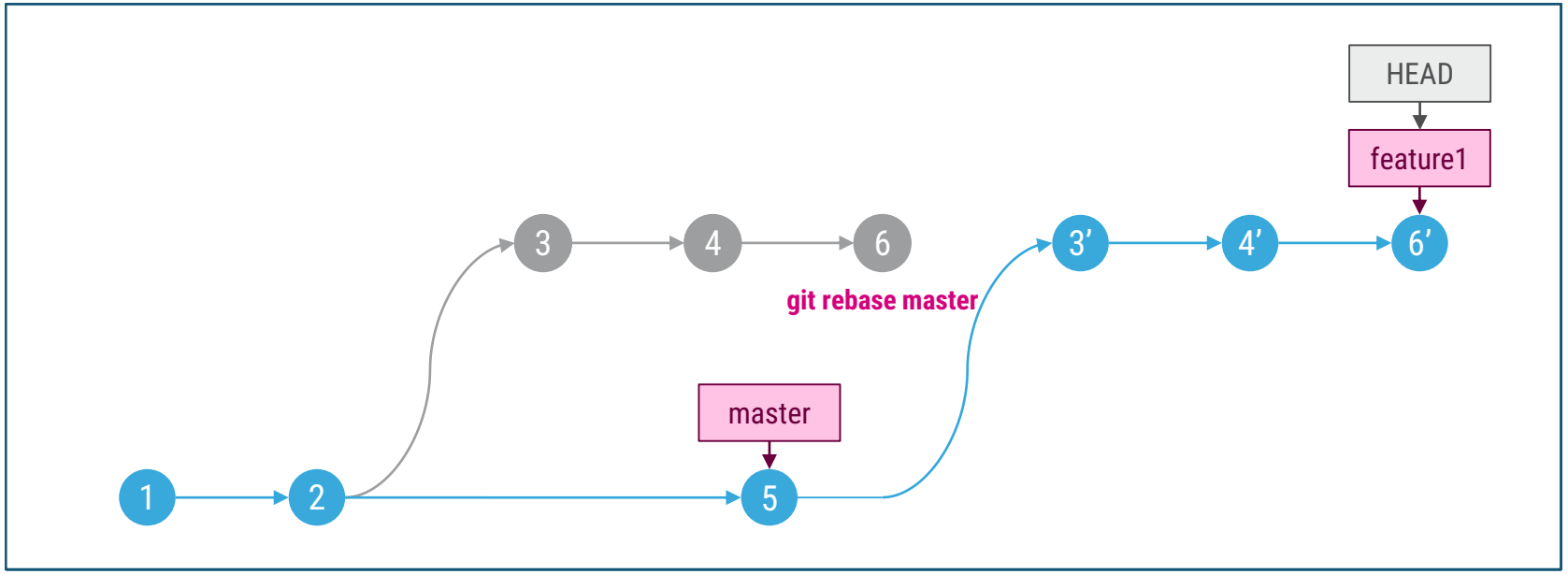
Replay commits
Cherry picking

- Applies the content of a <commit> to the HEAD
- Useful to apply fix into maintenance branch
- `git cherry-pick <commit>`



Rebase branche

- Synchronize the branch with last updates



- Can be used to rebase <branch> over origin/<branch>
- Only rebase local branches, never rewrite public history

Lab 5-6

Rebasing

Rewrite local history
Squashing

- Last commit: forgot to add a file / need to change message
  - `git commit --amend`
    ➔ Update the last commit instead of creating a new one
- A series of commits (i.e. branch)
  - Re-order, merge bug & fix, group style updates, ...
  - `git rebase -i <commit>`
    ➔ Opens an editor with the list of the commits & actions to be done (defaults ***pick***)

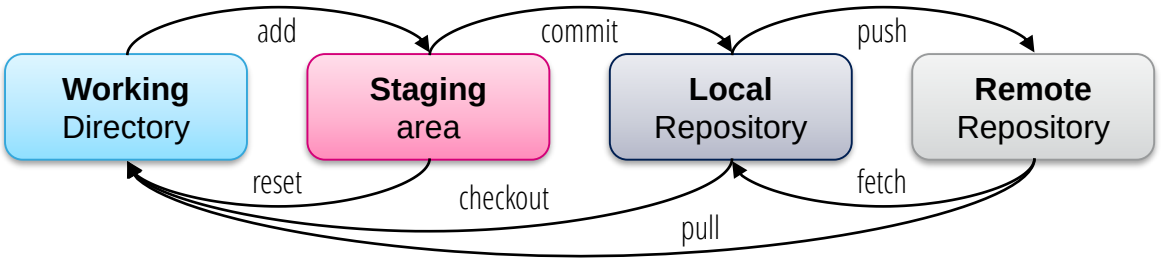
```
pick  fbde9fd   Add Readme.md
pick  70aaed5   Update Readme
pick  ccf2779   Update Readme again
pick  8c706b5   Update Readme again and again
```

  - Actions
    - **p, pick** replay the commit
    - **s, squash** merge this commit with the previous one
    - **e, edit** use commit, but stop for amending
      - `modify` ➔ `git add` ➔ `git commit --amend` ➔ `git rebase --continue`
  - line order can be changed to group actions

Lab 7

Rewrite local history

git essential commands



Create repository

Init or create <dir> as Git repo (defaults to '.')
$ git init [<dir>]

Clone an existing repo
$ git clone <repo_url> [<local_repo_name>]

Make changes

List actions, changed & new files in local repo
$ git status

Show diffs on tracked files in working dir
$ git diff [<file>]

Show staged diffs that will be commited
$ git diff --staged [<file>]

Stage given file(s)
$ git add <file> [<file> ...]

Stage all updates in directories
$ git add <dir> [<dir> ...]
$ git add .

Stage all updates in working directory
$ git add -A

Commit staged changes to local repo
$ git commit [-m "<commit_message>"]

Update current commit
$ git commit --amend

Explore history

Show all changes applied in <commit>
$ git show <commit>

Show current branch (entire with --all) history
$ git log [--all] [--graph] [--oneline]

Revert changes

Remove any local change in <file>
$ git checkout -- <file> [<file> ...]

Unstage <file>, keeping changes
$ git reset <file> [<file> ...]

Revert to <commit>, keeping changes
$ git reset <commit>

Revert to <commit>, loosing changes
$ git reset --hard <commit>

Synchronize

Push local changes to <remote>
$ git push <remote> <branch>

Get changes in <remote> (no merge)
$ git fetch <remote>

Get changes in <remote> & merge
$ git pull <remote> <branch>

Get all available remotes with URLs
$ git remote -v

Add a new remote repo
$ git remote add <name> <url>

Branches

master default branch name
origin default remote name
HEAD current point

Create <branch> at HEAD
$ git branch <branch>

Switch to <branch>
$ git checkout <branch>

Create and switch to <branch> at HEAD
$ git checkout -b <branch>

List all branches
$ git branch -a

Delete a local branch
$ git branch -D <branch>

Delete a remote branch
$ git push origin --delete <branch>

Merge <branch> into current branch
$ git merge <branch>

Rebase current branch over <branch>
$ git rebase <branch>

Rebase current branch interactively
$ git rebase -i <branch>

git training - Day 2

life.augmented

Agenda

DAY 1

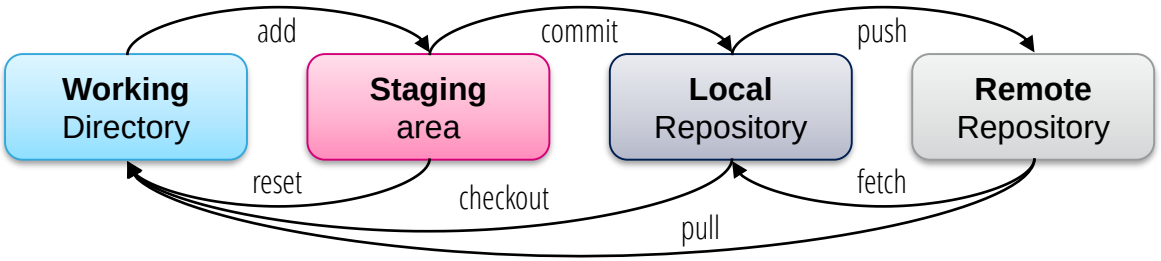
- Introduction
  - About version control
  - Why Git ?
- Install & configure Git
- Work with local repo
- Branching
  - Branches
  - Merge
- Work with distant repo
- Rebasing

DAY 2

- Day 1 debrief
- GUI tools
- Additional commands
  - Tagging
  - Patching
  - Debugging
  - Stashing
  - Ignoring files
  - Delivering
- Workflows
- Best practices

Day 1 debrief

git essential commands



Create repository

Init or create <dir> as Git repo (defaults to '.')
$ git init [<dir>]

Clone an existing repo
$ git clone <repo_url> [<local_repo_name>]

Make changes

List actions, changed & new files in local repo
$ git status

Show diffs on tracked files in working dir
$ git diff [<file>]

Show staged diffs that will be commited
$ git diff --staged [<file>]

Stage given file(s)
$ git add <file> [<file> ...]

Stage all updates in directories
$ git add <dir> [<dir> ...]
$ git add .

Stage all updates in working directory
$ git add -A

Commit staged changes to local repo
$ git commit [-m "<commit_message>"]

Update current commit
$ git commit --amend

Explore history

Show all changes applied in <commit>
$ git show <commit>

Show current branch (entire with --all) history
$ git log [--all] [--graph] [--oneline]

Revert changes

Remove any local change in <file>
$ git checkout -- <file> [<file> ...]

Unstage <file>, keeping changes
$ git reset <file> [<file> ...]

Revert to <commit>, keeping changes
$ git reset <commit>

Revert to <commit>, loosing changes
$ git reset --hard <commit>

Synchronize

Push local changes to <remote>
$ git push <remote> <branch>

Get changes in <remote> (no merge)
$ git fetch <remote>

Get changes in <remote> & merge
$ git pull <remote> <branch>

Get all available remotes with URLs
$ git remote -v

Add a new remote repo
$ git remote add <name> <url>

Branches

master default branch name
origin default remote name
HEAD current point

Create <branch> at HEAD
$ git branch <branch>

Switch to <branch>
$ git checkout <branch>

Create and switch to <branch> at HEAD
$ git checkout -b <branch>

List all branches
$ git branch -a

Delete a local branch
$ git branch -D <branch>

Delete a remote branch
$ git push origin --delete <branch>

Merge <branch> into current branch
$ git merge <branch>

Rebase current branch over <branch>
$ git rebase <branch>

Rebase current branch interactively
$ git rebase -i <branch>

GUI Clients

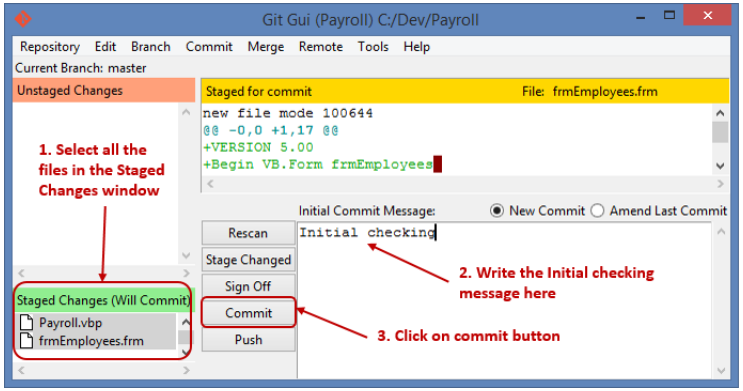
GUI Clients

- Many different GUIs with different capabilities
  - May be platform dependent
  - GitKraken, SmartGit, Git-cola, SourceTree, Git GUI, Tortoise, ...
- Git client can be embedded into IDE / Text Editor
  - VSCode, Atom, Eclipse, ...
- Focus on ***built-in client*** + ***Tortoise*** (8787)

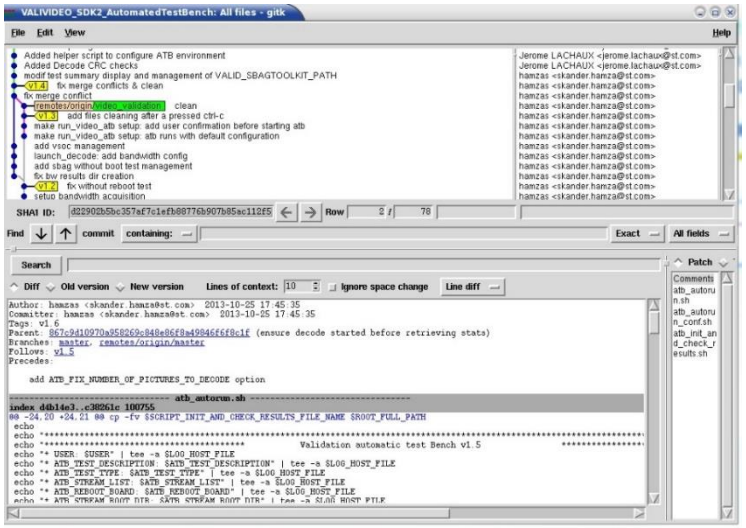
GUI tools
`git gui` & `gitk`

- Git comes with built-in GUI tools for committing and browsing

- `git gui` (for committing)
  - make changes to their repository
    - by making new commits,
    - amending existing ones,
    - creating branches,
    - performing local merges
    - and fetching/pushing to remote repositories

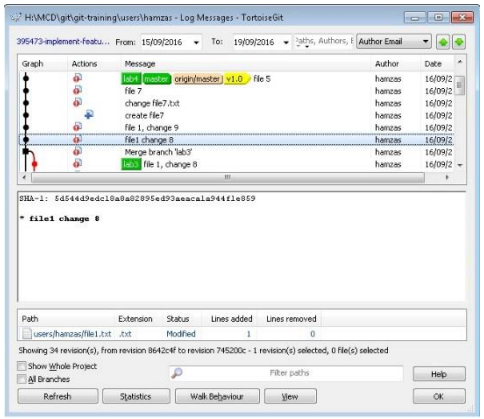


- `gitk` (for browsing)
  - Display changes
    - visualizing the commit graph,
    - showing commit's information
    - Showing the file in the trees of each version

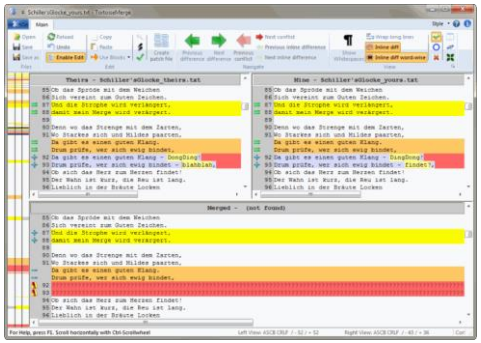


GUI tools
Tortoise

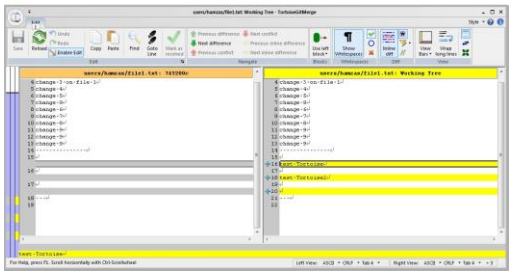
- Tortoise: the power of Git in a ***Windows*** Shell Interface
  - It's open source and can fully be build with freely available software.
- Features of Tortoisegit
  - Provides overlay icons showing the file status
  - Power context menu with all commands



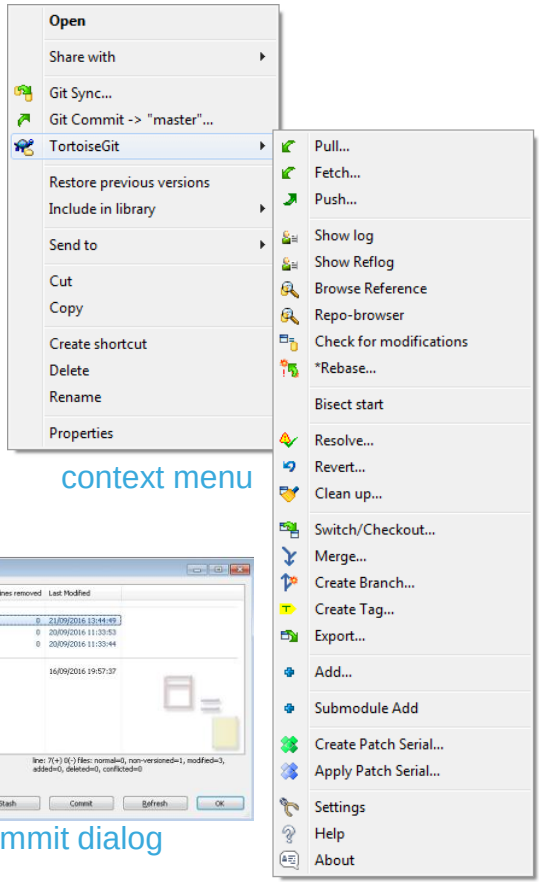
gitLog



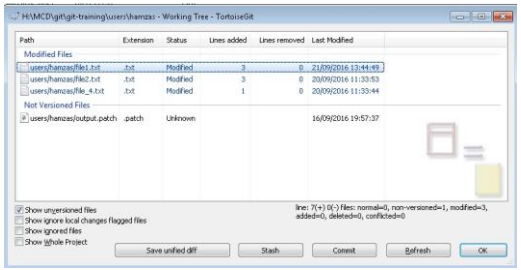
mergetool



gitDiff



context menu



Tortoise commit dialog

Overlay icons

life.augmented

Additional Commands

Work with partial history
Shallow

- Long projects have huge history
  - Takes time to clone the full history
  - Requires useless disk space
- You can clone only the commits on the ***specified branch***

```
git clone -b <branch> --single-branch <remote>
```

- You can clone the ***branch*** & ***depth*** of history
  - implies --single-branch

```
git clone -b <branch> --depth=<number> <remote>
```

- You can clone only the commits specific to a branch

  (Excluding the parent branch)

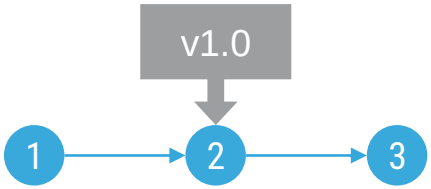
```
git clone -b <branch> --shallow-exclude=<parent-branch> <remote>
```

- You can change the number of fetched commits in a shallow clone

```
git fetch --depth=<number>
```

Tagging & Patching

- Tag
  - Git offers the ability to identify a specific commit with a descriptive label
  - This functionality is typically used to mark release points
    - Create a tag : `git tag –a <tagName> –m <message>`
    - Push a tag: `git push <remote> <tagName>`
    - Push all tags: `git push <remote> --tags`
    - Checkout tag: `git checkout <tagName>`



- Patch
  - Creating a patch is a good way to share changes that your are not ready to push
  - Patch is simply a concatenation of the diffs for each of the commits
    - Method 1 (apply changes without commit)
      - Create patch: `git diff from_commit to_commit > output_patch_file`
      - Apply patch: `git apply output_patch_file`
    - Method 2 (with commit, more formal and keeps authors name)
      - Create for last 2 commits: `git format-patch --stdout -2 > output_patch_file`
      - Apply patch: `git am name_of_patch_file`

Debugging

- Git provides tools to help issues debug
- `git blame <file>` : annotates each line of any file with
  - When was this line modified the last time
  - Which person is the last one that modified that line

- `git grep <pattern>` : find string or regular expression in any of the file in your source code

- `git bisect`: helps to find which specific commit was the first to introduce the bug using binary search
  - Init commands:
    - `git bisect start` Enter bisect mode
    - `git bisect bad <commit1>` Inform Git that <commit1> contains the bug
    - `git bisect good <commit2>` Inform Git that <commit2> does ***not*** contain the bug
  - Git will incrementally checkout versions expecting `git bisect bad|good` to refine research
  - End with `git bisect reset`
  - `git bisect –help` for more information

Git Stash

Stash: store (something) safely in a hidden place

- When use `git stash`
  - You want to switch branches for a bit to work on something else
  - You don’t want to commit a half done work but keep it to get back later
    - ➔ Stash your work (save it) then switch branches
- How to stash a work
  - On current branch, files are modified
    - Use `git stash` to save all the changes you did
    - Use `git stash -u` to save also the untracked files you have in the workspace
  - Switch branch; work on something else
  - Go back to your initial branch
    - Use `git stash pop` to get your modification back and clean the stash
  - To check the content of your stash
    - Use `git stash show`
  - To remove anything currently stashed (no recovery)
    - Use `git stash clear`

Ignoring files & directories

- For files that you don’t want Git to show as being untracked
  - Like automatically generated files (log files, files produced by build system)
- In such cases, you can put those files patterns into `.gitignore` file
- Syntax
  - Blank lines & trailing spaces are ignored
  - Lines starting with `#` are comments
  - Shell glob `*` matches any character except `/`
  - Shell glob `?` matches any one character except `/`
  - Shell glob `[ ]` matches one character in range
  - `**` matches any path
  - `/` forces git to consider filtering on directories only
  - `!` starting line negates the pattern

```
# this is a comment

*.o
!foo.o
*~
**/foo
/*.foo
node_modules/
/bar/**/foo
```

- Don’t forget to add and commit the `.gitignore` file !
- `git clean -x` will remove ignored files in addition to untracked files

Delivering

- Provide a given version of the project
  - without any history
  - without specific data
- git archive -o <archive>.tgz *[--prefix=<prefix>/] [--remote=<remote>]* <ref>
  - archive format inferred from file name (zip, tar, tar.gz, tgz) or `--format=<format>`
  - `--prefix` allows to prepend all files with a path in the archive (i.e. product name)
  - `--remote` allows to fetch data from a remote repository (otherwise exported from a local repository)
  - `<ref>` is the version to be extracted (SHA1, tag, branch, ...)
- Exclude some files from delivery (e.g. .gitignore)
  - Add a `.gitattributes` file to your repo with export-ignore directives like this

```
.gitignore export-ignore
.gitattributes export-ignore
```

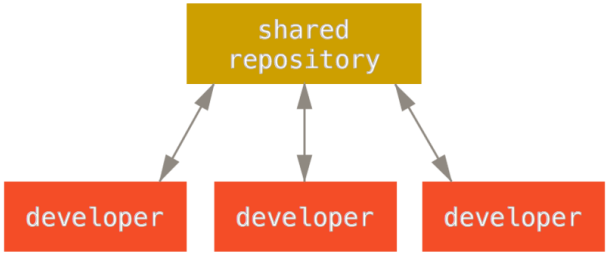
  - Don't forget to add & commit `.gitattribute` file !

Workflows

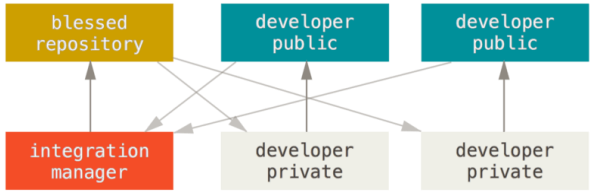
Git workflows

- Git provides the flexibility to design a version control workflow that meets each team needs, there are many usable Git workflows.

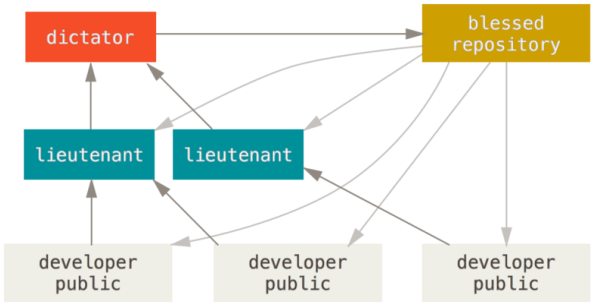
Centralized
(single branch, squashing, small teams)

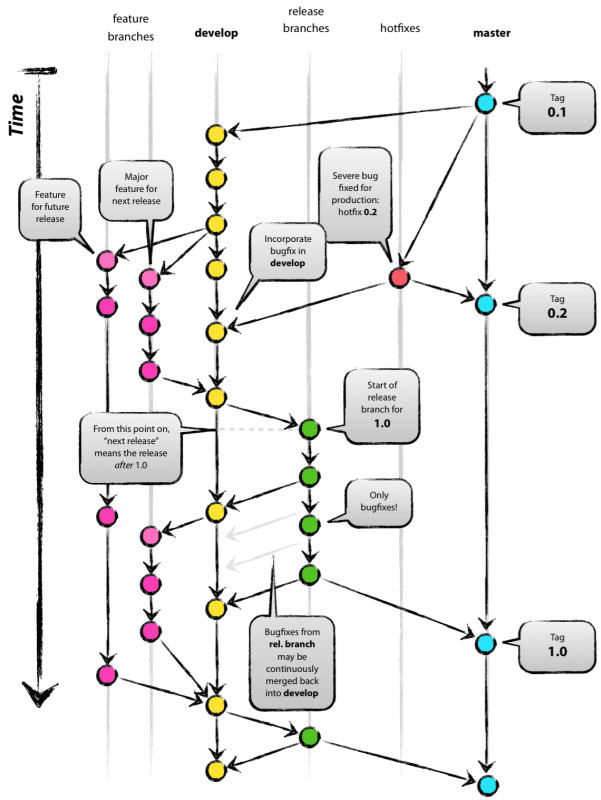


Integration Manager
(pull requests like)



Dictator & Lieutenants
(huge projects)





Gitflow
(strict branching model)

life.augmented

Best Practices

Best practices
1/2

- The commit log of a well-managed repository tells a story.
- Make commit unitary
  - Commit often on small single tasks
  - Easier review, research, revert or cherry picking
- Do make useful commit messages
  - Keep commit ***header*** message short & meaningful
  - Be concise (50 chars for message, < 80 for full header)
  - Write header message in the imperative tense, by competing the sentence
    - If applied, this commit will <header>
  - Force to stay compliant to the fixed ruleshowever, rules may evolve)
- Examples of templates
  - `art #xxxxxx : <short description>`
  - `<type>(<scope>): <short description> (art #xxxxxx)`

Best practices
2/2

- Review code using git diff before committing

- Practice good branching hygiene
  - Do not work directly on master.
  - Create a branch for each development
    (new feature, bug fixes, experiments, ideas)
  - Delete branches as they're merged

- Do commit early and often
  - Implements a single change to the code at a time
  - Don't mix several functional updates
  - Don't mix functional & style updates
  - Test before commit, don't commit half-done work

References

References

- Some interesting references:
  - Why Git
  - A Git workflow for Agile team
  - Git best practices
  - Pro Git (free ebook)
  - Git reference
  - Atlassian tutorial
  - Gitflow workflow

- Most important
  - `git --help`
  - `git <command> --help`
  - `git status`