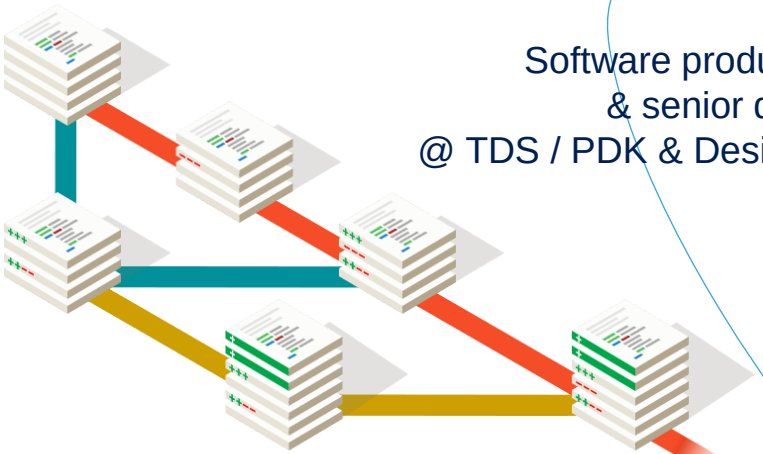
git advanced

Learning-Catalogue code: 070174

Sébastien SNAIDERO

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



life.augmented

Agenda

- Day 1
  - Git basics debrief + lab
  - Reset & revert + lab
  - Detached head & reflog + lab
  - Git submodule + lab
  - Git subtree + lab
  - Repo tool + lab

- Day 2
  - Gerrit + lab
  - Pull Requests + lab
  - Hooks
  - Workflow discussion
  - Best Practices

Basics debrief

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

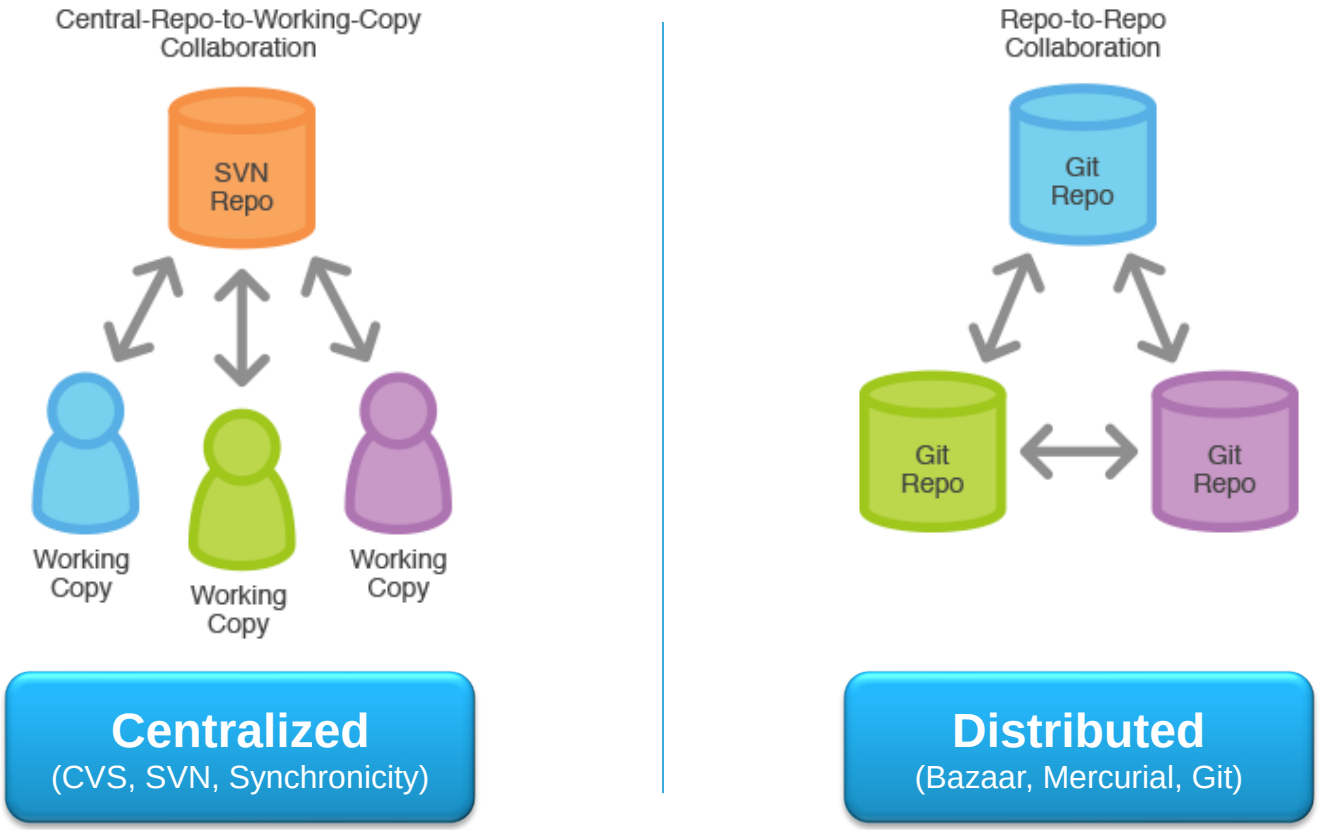
- Helps collaboration
  - Allows the merge of all changes in a common version

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

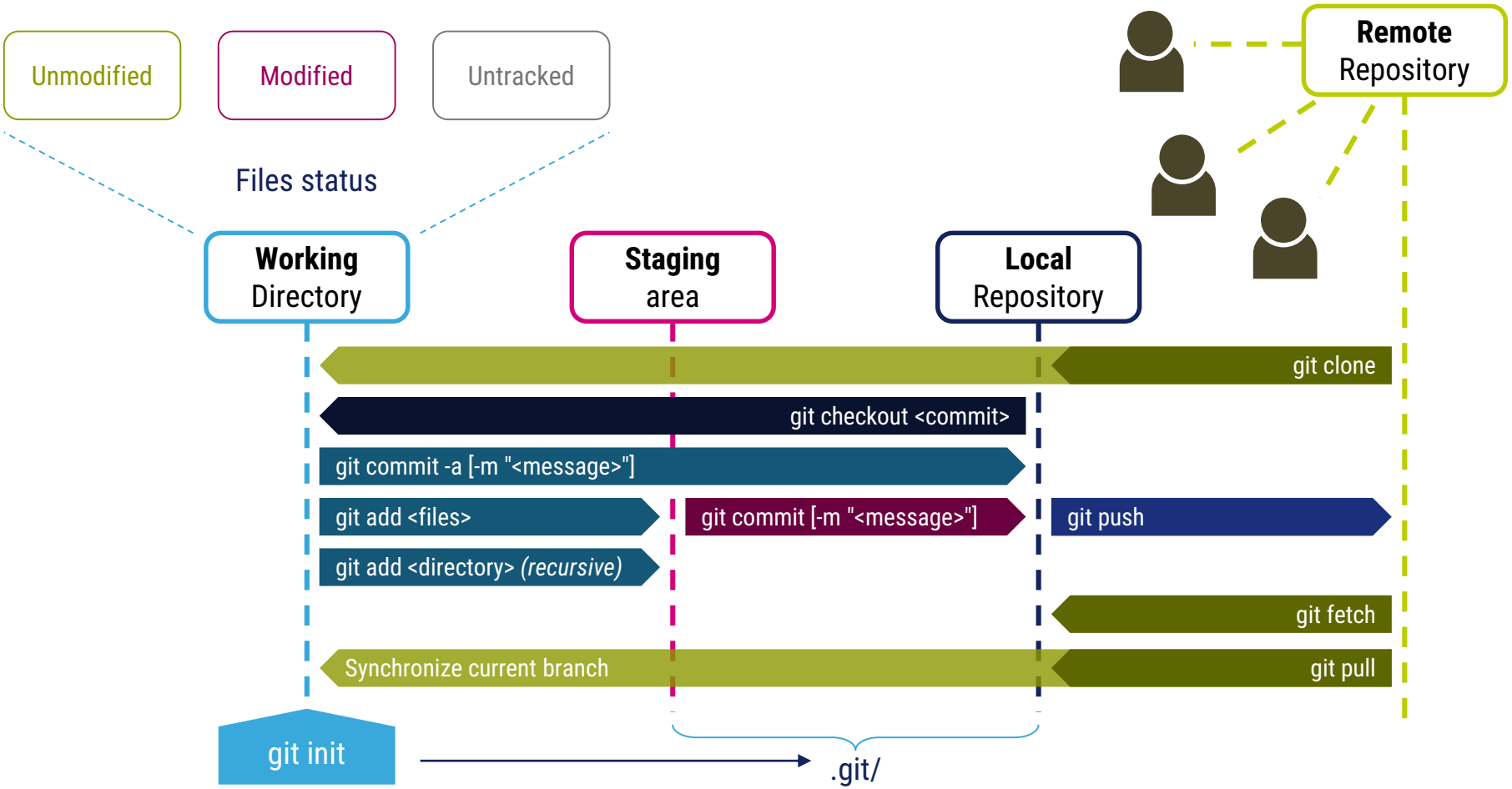
About Version Control

How ?

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



How it works ?

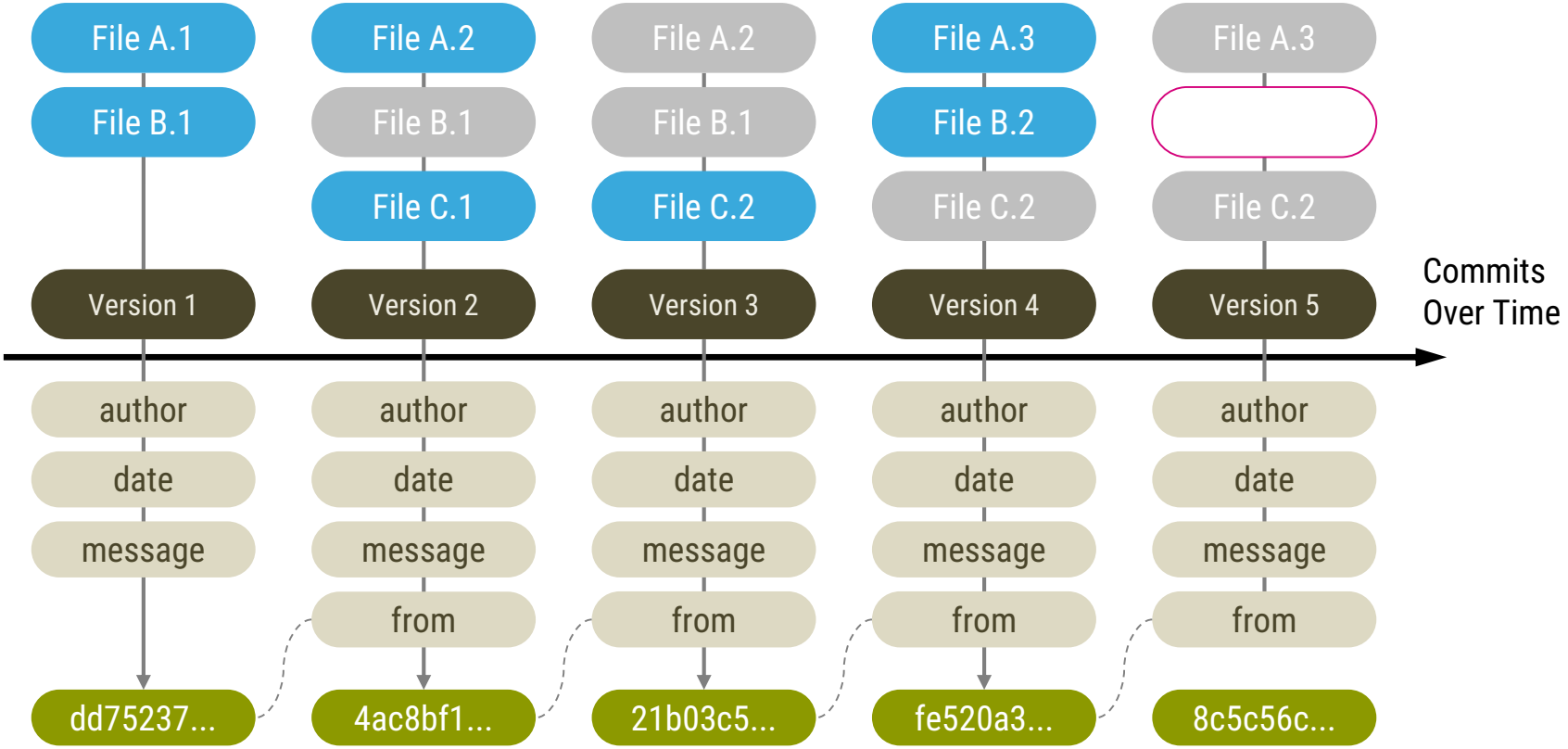


git clone = mkdir + git init + git remote add origin + git fetch + git checkout --track origin/master

Note: Remote repository can be a local repository

Versioning in Git

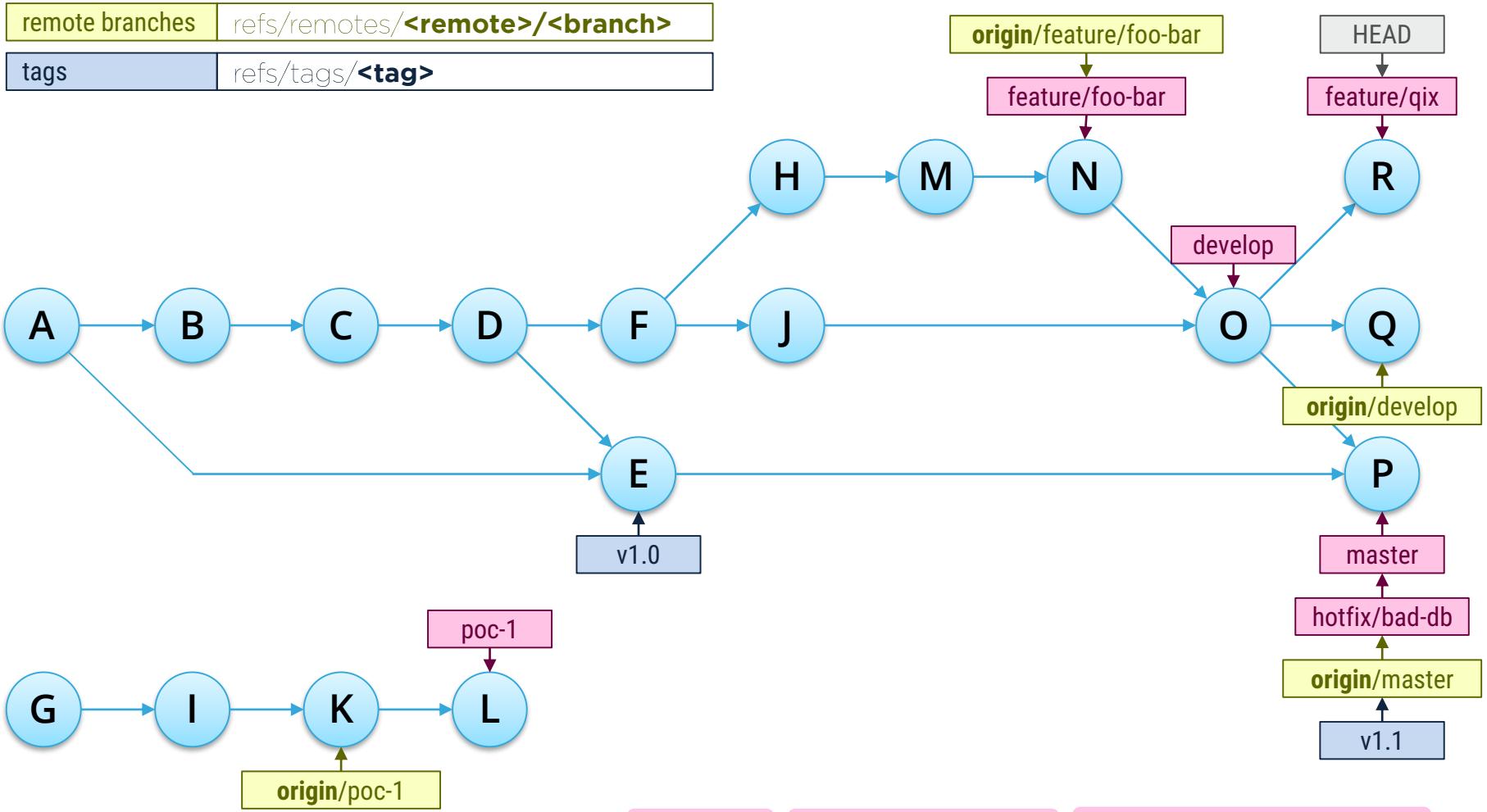
- Snapshots, not differences ➔ Speed, branching
- Mostly add data ➔ Committed things never lost without confirmation



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

References

local branches	refs/heads/**<branch>**
HEAD	**HEAD** (ref to a local branch)
remote branches	refs/remotes/**<remote>/<branch>**
tags	refs/tags/**<tag>**



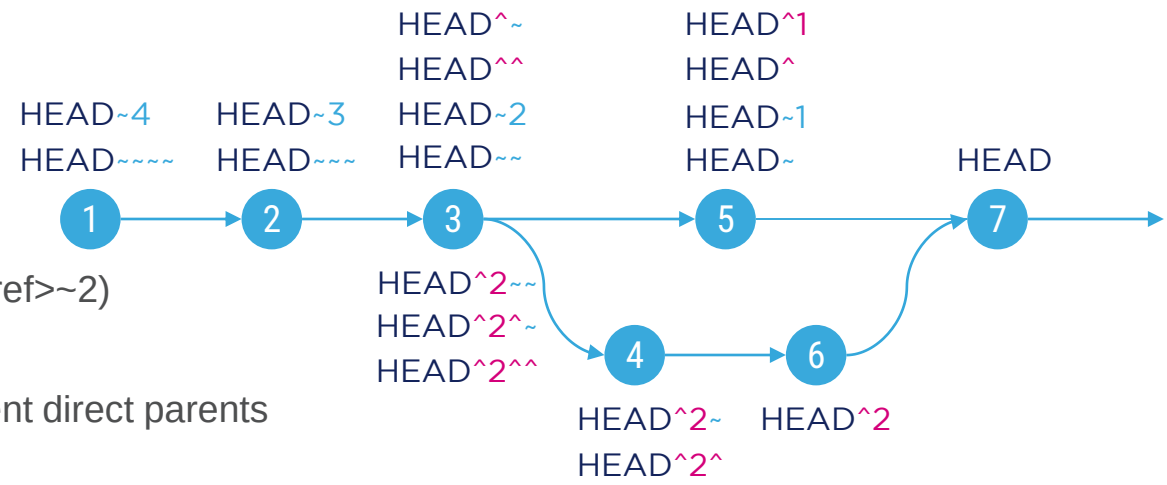
git branch | git branch <branch> | git branch (-d | -D) <branch>

git checkout <branch> | git checkout -b <branch>

git tag [-a] <tag> | git push <tag> | git push --tags

References (cont.)

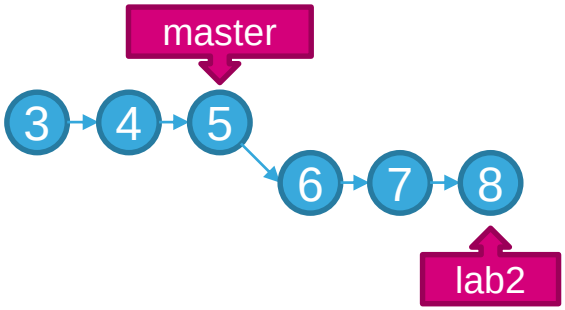
- Short SHA1
  - 6-8 first characters are usually enough
    - 2^2N objects in project without collision (N length of short SHA1)
    - Linux Kernel project uses only 12 characters (875 000 commits, 7 millions objects)
- Relative
  - <ref>~
    - Parent of <ref>
  - <ref>~#
    - Ancestor of <ref>
      (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

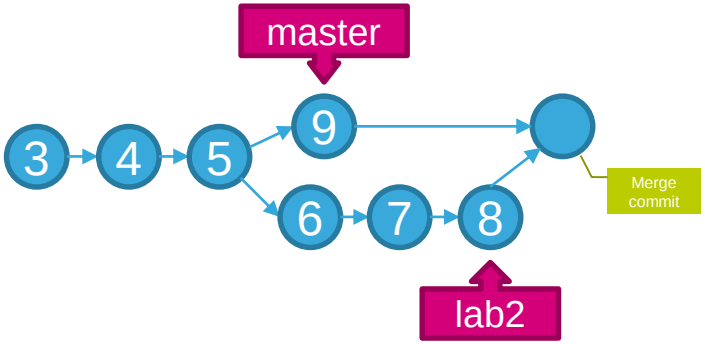
Merging branches strategies

- Fast forward



  - Moves branch pointer

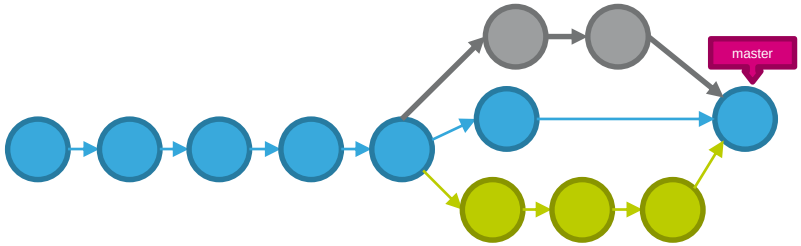
- Recursive



  - Creates a merge commit

Some behaviors

- git merge [--ff] <branch> (default)
  - ff if possible otherwise recursive
- git merge --no-ff <branch>
  - Force merge commit (i.e. recursive)
- git merge --ff-only <branch>
  - Fail if not possible to fast forward
- git merge <branch1> <branch2>
  - Octopus merge



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
  - git checkout --ours <file>
  - git checkout --theirs <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)

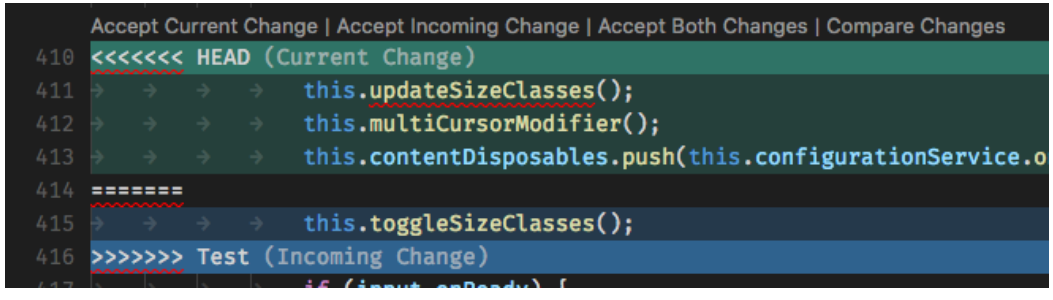
- Merge process can be canceled using `git merge --abort`
- Merge conflict can show common ancestor code
  - Set configuration option `merge.conflictStyle` to `diff3`
  - This will present conflicts in the form

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

- Merge process can be assisted by GUI tool (Meld, Kdiff3) ...

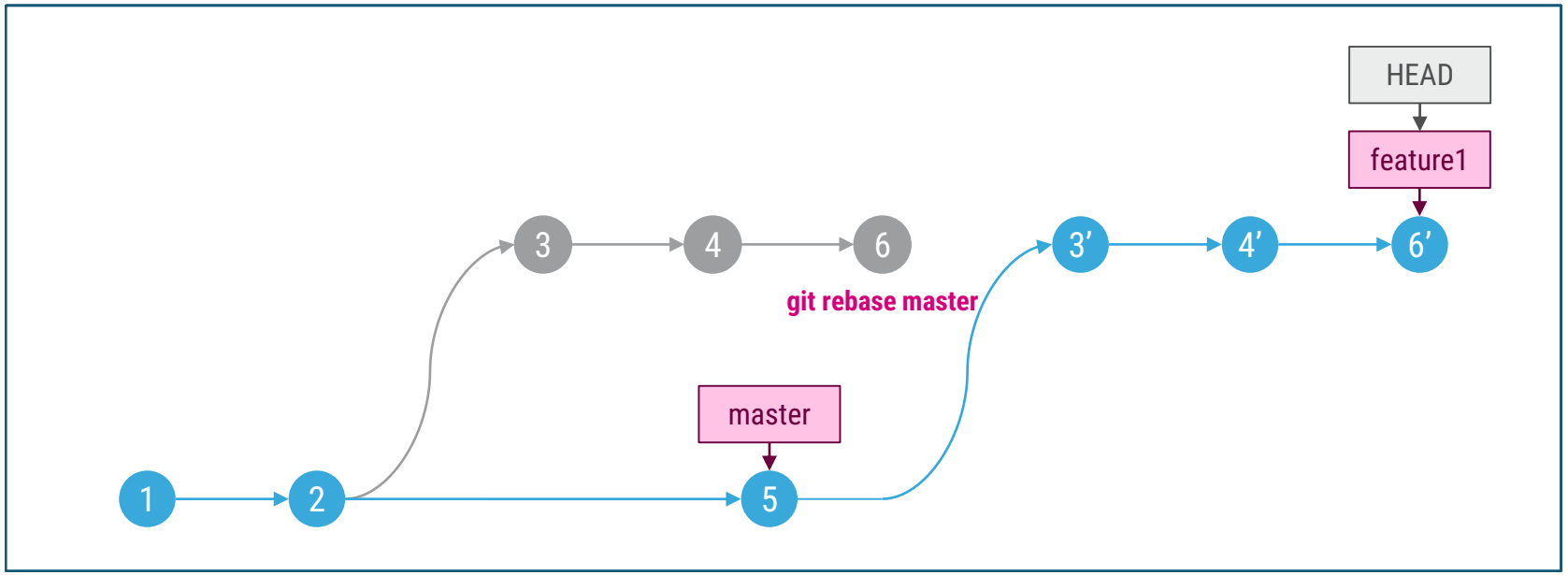
```
[merge]
   tool = meld
[mergetool "meld"]
   cmd = meld --auto-merge \"$LOCAL\" \"$BASE\" \"$REMOTE\" --output \"$MERGED\" --label \"MERGE (REMOTE BASE MY)\"
   trustExitCode = false
[mergetool]
   prompt = false
   keepBackup = false
```

- ... and even proposed in your IDE (VSCode, ...)



Rebase branche

- Synchronize the branch with last updates



- It eliminates the need for huge (time consuming) merge commits in favor of frequent smaller & easier conflicts solving
- Can be used to rebase <branch> over origin/<branch> (pull strategy)
- Rebasing allows fast forward merge
  - Results to a linear history with default merge strategy
- Only rebase local branches, **never rewrite public history**

Rewrite **local** history

- 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, remove, merge bug & fix, group style updates, ...
  - `git rebase -i <from-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
```

Lines executed
from top to bottom

  - Actions
    - **remove line** ignore the commit
    - **p, pick** replay the commit
    - **s, squash** merge this commit with the previous one (edit commit message)
    - **f, fixup** merge this commit with the previous one (discard commit message)
    - **e, edit** use commit, but stop for amending
      - `modify` ➔ `git add` ➔ `git commit --amend` ➔ `git rebase --continue`
  - lines can be re-ordered (i.e. to group actions)

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 : workflow

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)

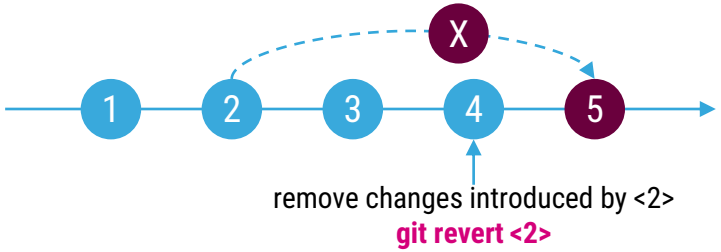
BLANK LINE (mandatory in the BODY)

BODY (technically optional)

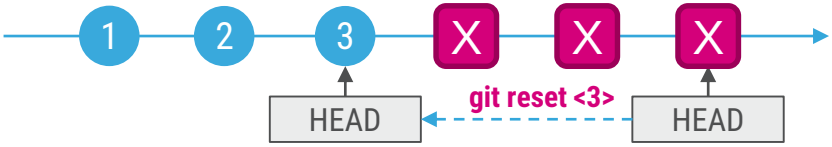
- `git commit -m "<message>"` will create a commit message with a single Header line containing <message> without opening an editor

Reset & Revert

- Revert
  - Creates a new commit, **nothing is removed from history**
  - `git revert [-n] [<commit>]`
  - If no commit specified defaults to HEAD, reverting last commit
  - `-n` will perform dry-run to let you know what would be done without executing



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



- Options, mitigate effects
  - `--soft`

    Commit is ***removed***, but all the ***changes*** are ***kept*** in ***staging*** area
  - `[--mixed]` (this is the default)

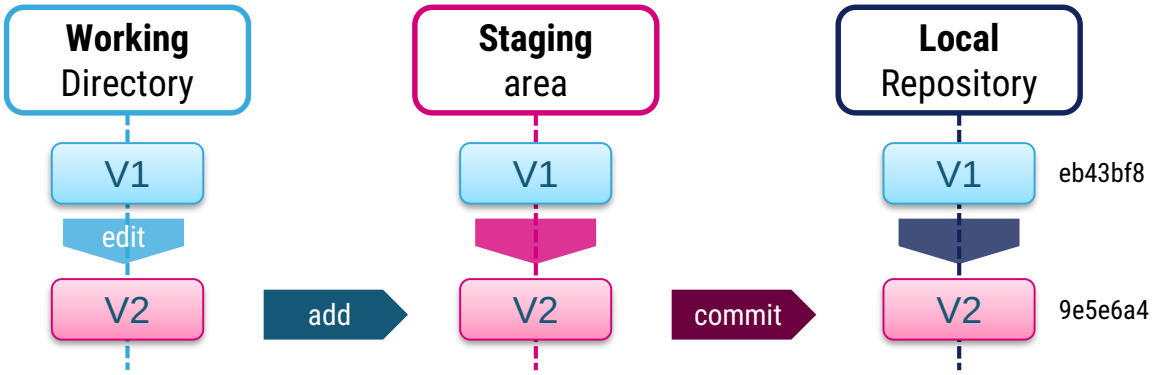
    Staging area is ***cleared***, but all the ***changes*** are ***kept*** in ***working*** directory
  - `--hard`

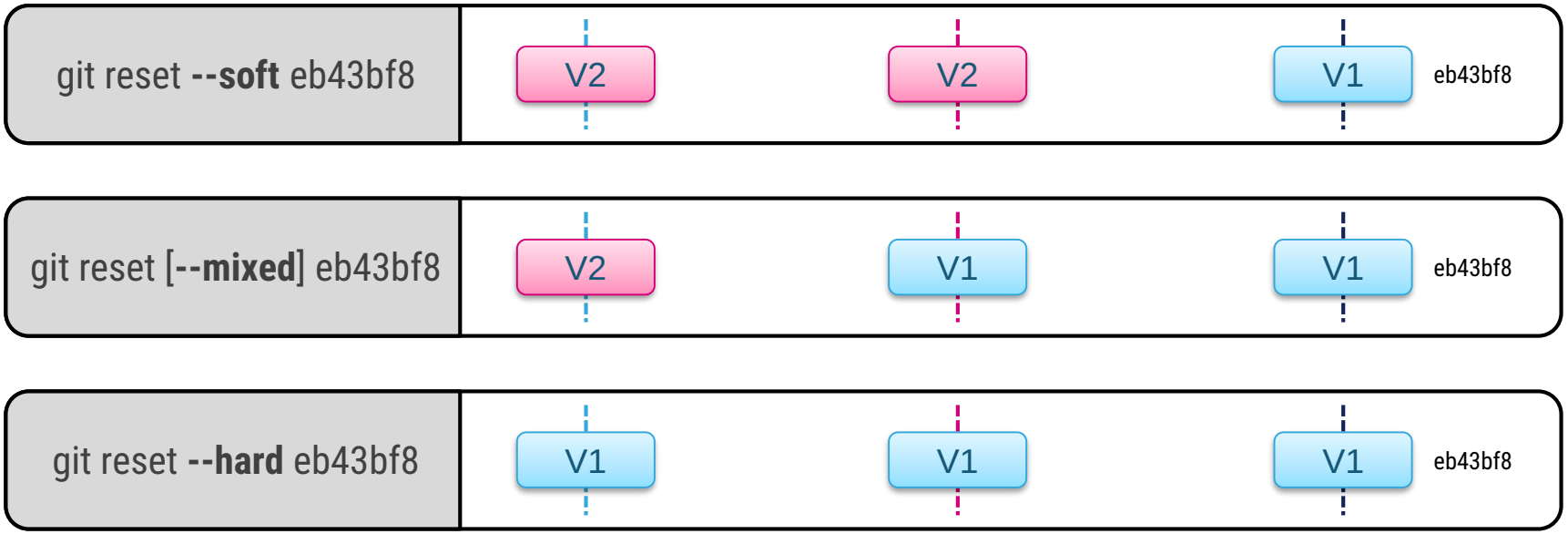
    Everything is lost on local branches ⚠

Public branches

Private branches

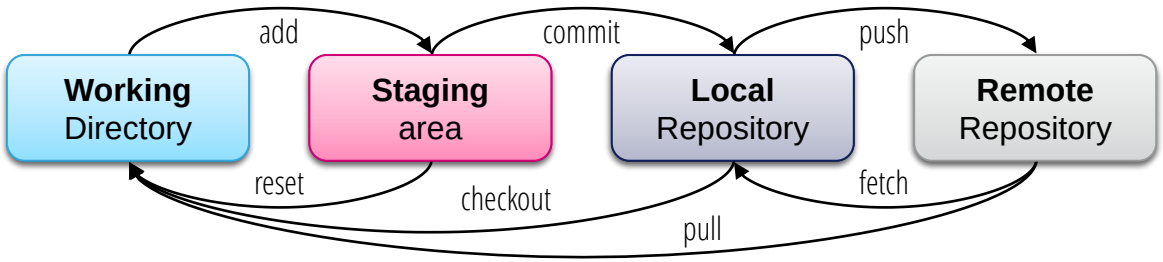
Reset demystified





Lab : revert & reset

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> ...]

Undo to <commit>, keeping history
$ git revert <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>

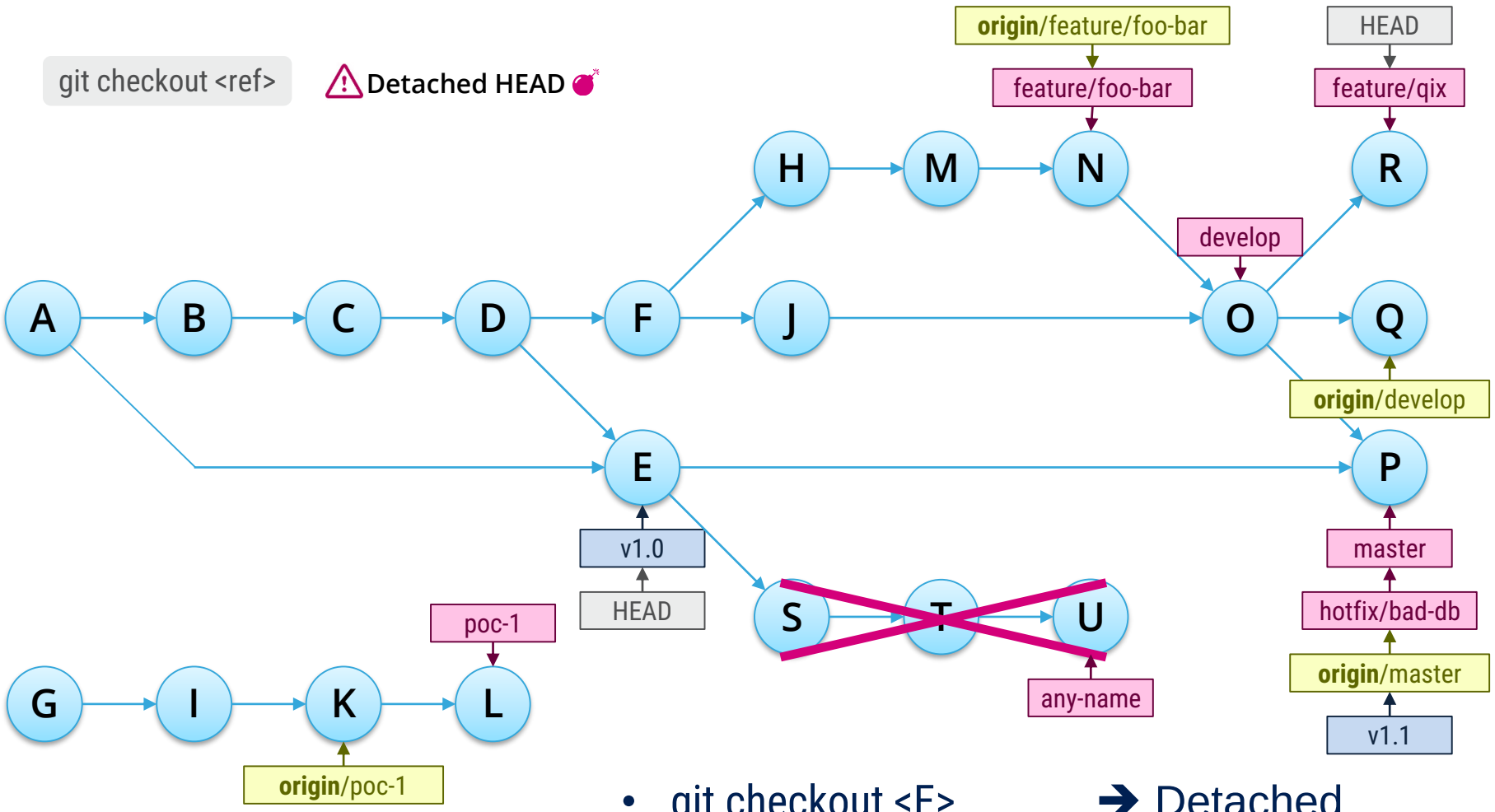
Delete a remote branch
$ git push origin --delete <branch>

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

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

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

Detached HEAD



- git checkout <E> ➔ Detached
- git commit ➔ S ➔ T ➔ U
- git checkout feature/qix ➔ Lost
- Fix loss by creating branch before leaving

life.augmented

git reflog

- Git never loses anything
  - Even if you rebase, amends commits, rewrite history, leave unreachable commits…
  - All contents are kept stored in the local repository for some times

- Reflog shows an ordered list of the commits that HEAD has pointed to
  - The output is as following ➔

```
hamzas@TUNCWL0102 MINGW64 ~/Desktop/gitTraining/git/myProject (master)
$ git reflog
1c0fc90 HEAD@{0}: merge lab2: Fast-forward
b63e25f HEAD@{1}: checkout: moving from lab2 to master
1c0fc90 HEAD@{2}: rebase -i (finish): returning to refs/heads/lab2
1c0fc90 HEAD@{3}: rebase -i (squash): my final commit (combination of 3 commits)
9d048e0 HEAD@{4}: rebase -i (squash): # This is a combination of 2 commits.
ce9a9ba HEAD@{5}: rebase -i (start): checkout master
3fb5771 HEAD@{6}: commit: my commit number 3
eec06b7 HEAD@{7}: commit: my commit number 2
ce9a9ba HEAD@{8}: commit: my commit number 1
b63e25f HEAD@{9}: checkout: moving from master to lab2
b63e25f HEAD@{10}: commit (initial): initial commit
```

    - The most recent HEAD is first
    - First column is the commit ID at the point the change was made
    - The representation name@qualifier is reflog reference.
      - Reflog displays transactions for only one ref at a time, the default ref is HEAD
      - We can display changes on any branch

```
hamzas@TUNCWL0102 MINGW64 ~/Desktop/gitTraining/git/myProject (master)
$ git reflog show master
1c0fc90 master@{0}: merge lab2: Fast-forward
b63e25f master@{1}: commit (initial): initial commit
```

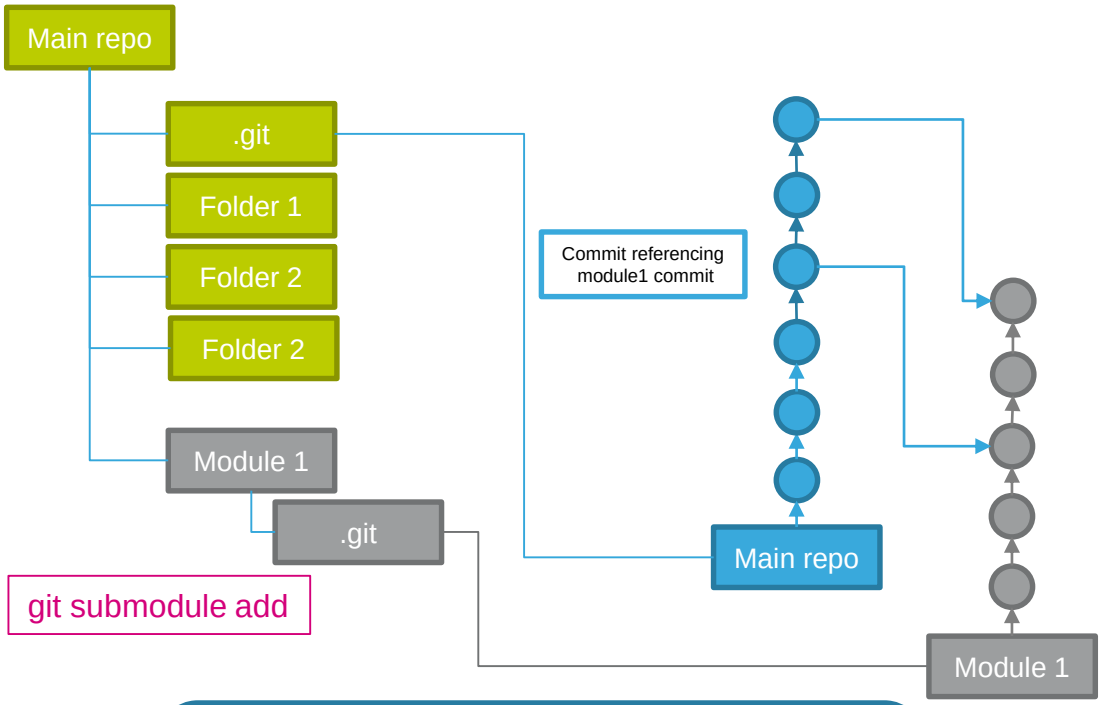
    - The final part is the type of the operation
      - And the commit message if the operation performs a commit

- Reflog expires
  - Default = 90 days

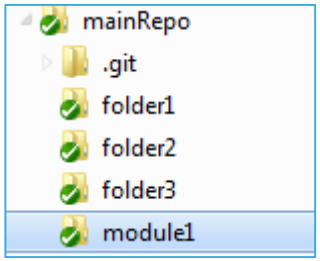
Submodule

Git uses commands
- git submodule add
- git clone --recurse-submodules
- git submodule update --init

submodule



- Adds external repositories in a subdirectory on your repo
  - Point these repositories in a commit
- Submodules act like traditional git repositories



- When cloning mainRepo
  - Folders are present
  - Submodules content is not downloaded
  - Solution :
    - git clone --recurse-submodules
    - git clone && git submodule update --init

life.augmented

Lab: submodule

Git submodule debrief

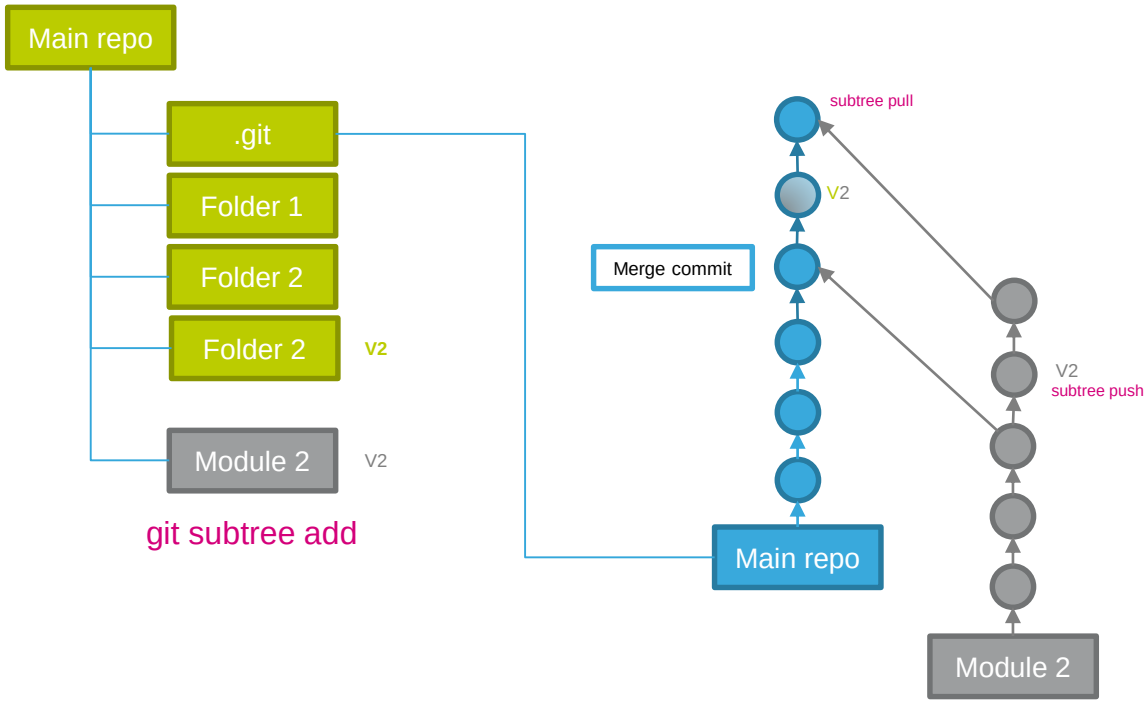
- Commands
  - Git submodule add <name> <url>
    - Git creates a .gitmodules file
      - Unique file for all submodules
      - Containes paths and url (only)
  - Git clone <url> && git submodule init && git submodule update
    - Shortcut #1 : git submodule update --init
    - Shortcut #2 : git clone --recurse-submodules <url>

- Submodules are not attached to any branch → « detached HEAD »
  - Submodules are referenced by their commit ID only

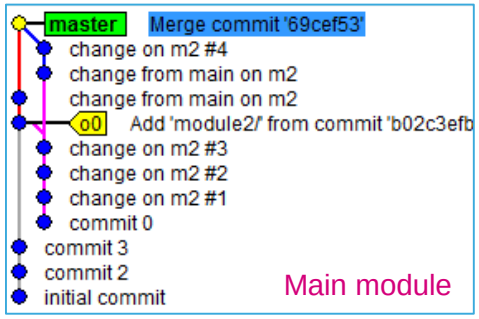
- Stay vigilant with your management of submodule
  - git push on main repo will not push the submodules' code
    - Do not push with references to commits that do not exist on the sub modules remotes
  - Solution → **Always push submodule before the parent project.**

Git uses commands
- git subtree add
- git subtree push
- git subtree pull

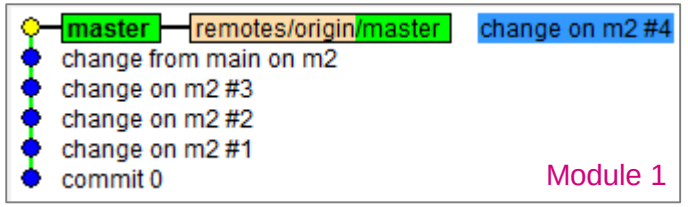
subtree



- Adds external repositories in a subdirectory on your repo
  - Repositories are merged
- They are part of main project
  - You can modify both main & subproject ; commit ; push
    - Ordinary Push: push commit to main
    - Subtree push: push only subtree modifications
  - You can update subtree version → subtree pull
    - New merge is done

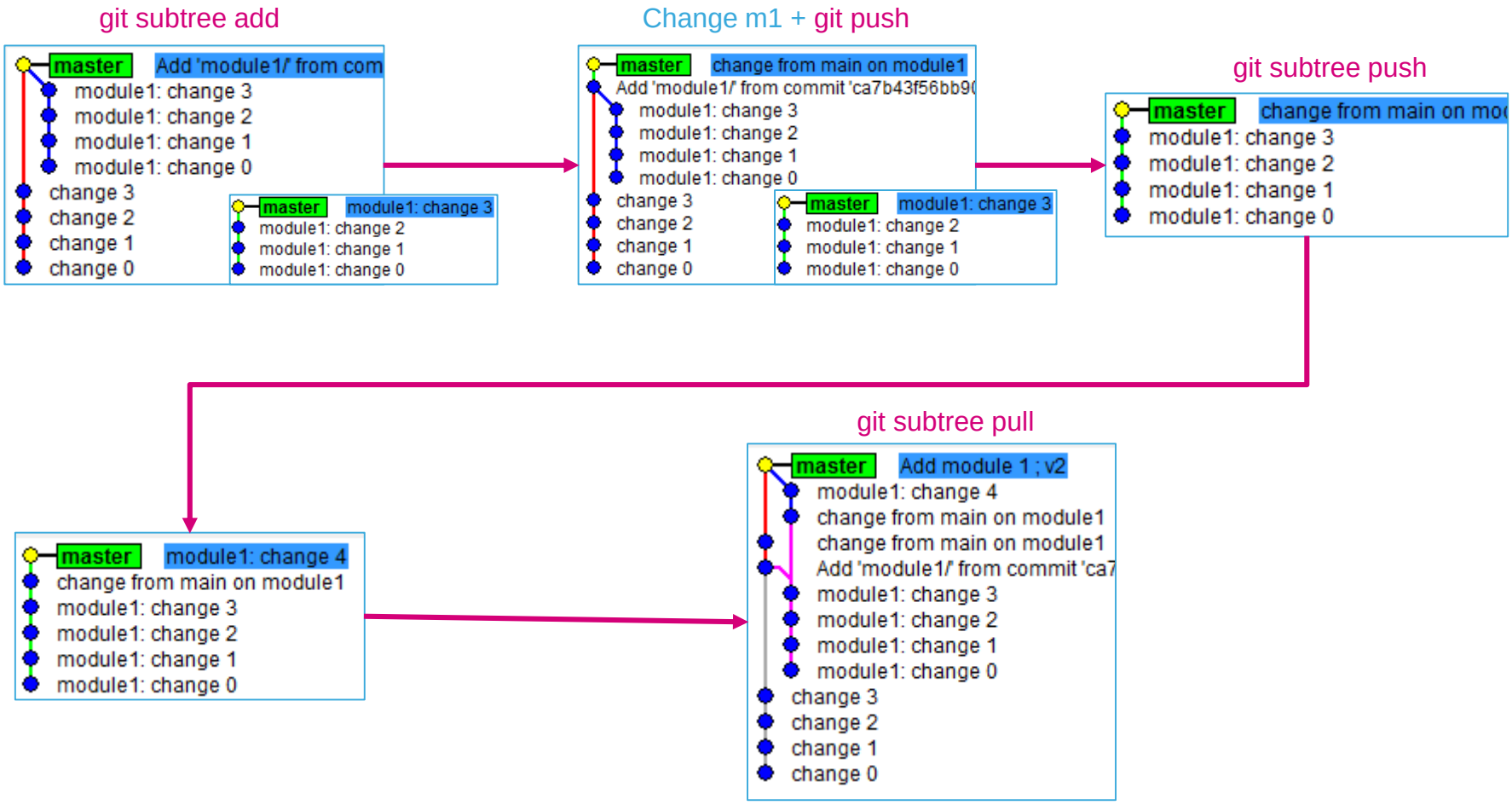


git subtree push



life.augmented

subtree


git subtree add
master
Add 'module1/' from com
module1: change 3
module1: change 2
module1: change 1
module1: change 0
change 3
change 2
change 1
change 0
master
module1: change 3
module1: change 2
module1: change 1
module1: change 0
Change m1 + git push
master
change from main on module1
Add 'module1/' from commit 'ca7b43f56bb9
module1: change 3
module1: change 2
module1: change 1
module1: change 0
change 3
change 2
change 1
change 0
master
module1: change 3
module1: change 2
module1: change 1
module1: change 0
git subtree push
master
change from main on mo
module1: change 3
module1: change 2
module1: change 1
module1: change 0
master
module1: change 4
change from main on module1
module1: change 3
module1: change 2
module1: change 1
module1: change 0
git subtree pull
master
Add module 1 ; v2
module1: change 4
change from main on module1
change from main on module1
Add 'module1/' from commit 'ca7
module1: change 3
module1: change 2
module1: change 1
module1: change 0
change 3
change 2
change 1
change 0

Lab 3: subtree

Subtree Vs Submodules

- The laziest way is to have a monorepo
  - It's sometimes better to keep things separated
    - To make things more organized and prevent your repo to became too big,

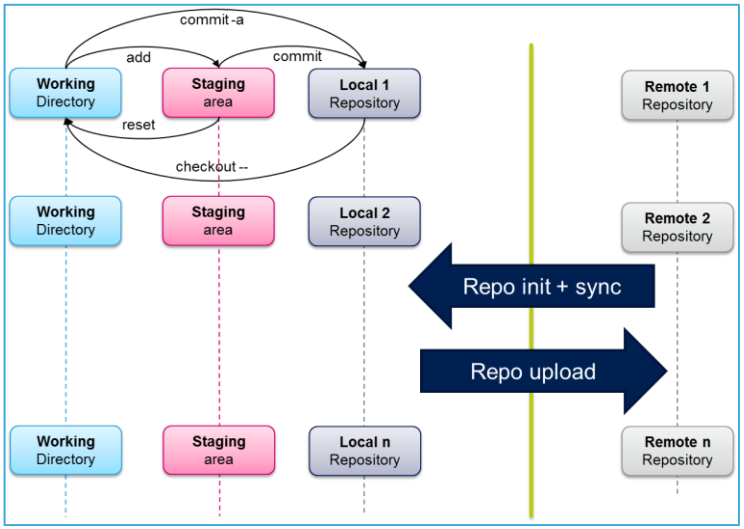
Submodules	Subtree
Harder	Easier
Link to a commit ref in another repo	Code is merged and is part of main repo
Requires additional steps	Just clone; pull; push
Smaller repo size	Bigger repo size
Easier to push ; harder to pull	Easier to pull ; harder to push

Repo

Repo tool

- Tool external to Git
  - Built by google for android developpement

- Clones multiple repositories into one local working directory
  - Single command allows ; pull ; push ; status, on Multiple repositories,
    - Repo command is an executable python script
  - Uses a manifest file. It's an XML file containing
    - repositories URL
    - folders to clone to
    - revision to checkout



Manifest

Remote description

Default values

Projects description

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <manifest>
3
4    <remote name="origin"
5        fetch  = "ssh://gitolite@codex.cro.st.com/stm32supportandthirdparties/Git-Training/"
6        review = "ssh://hamzas@gerrit.st.com:29418/stm32supportandthirdparties/Git-Training/"/>
7
8    <default remote ="origin" revision="master" sync-j="8"/>
9
10   <project path="modules/m1"          name="repo/module1"          remote="origin" />
11   <project path="modules/m2"          name="repo/module2"          remote="origin" />
12   <project path="component/m3"        name="repo/module3"          remote="origin" />
13   <project path="component/m4"        name="repo/module3"          remote="origin" />
14
15
16 </manifest>
17
```

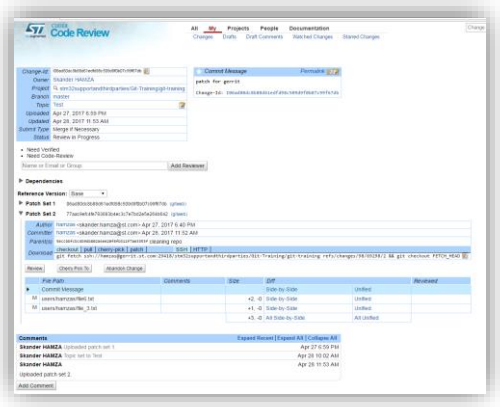
- <remote> : the remote server we are pulling from
- <project> : defines a single repo. Main attributes are
  - Path : where the repository will be cloned into
  - Name : Repository name in git server
  - Remote : the name for remote server as defined
  - Revision : the tag/branch we want to checkout
- <default> : default values for attributes we are using
  - sync-j : number of threads to use when syncing the system

Lab : Repo

Gerrit

Gerrit

- Free web-based collaborative code review tool that integrates with Git
  - Commits are uploaded to gerrit for review
    - There are called change-sets



Review categories: -2 ; -1 ; 0 ; 1 ; 2

- Verified (+1)
- Code review (+2)

Review rules

- Highest vote: enables submitting
  - +2 in code review
  - +1 verified
- Lowest vote (-2): veto for blocking

Gerrit – technical steps

- Clone the central repository
  - git clone <gerrit_url>
- use git to commit local changes then push changes for review
  - git push origin HEAD:refs/for/master
- The patch set is available on gerrit for review
  - Reviewers do the work through the web interface
- Owner can update the patch
  - Use commit --amend
  - Including the *Change-id* into the commit msg (last paragraph)
    - It's what uniquely defines the change in gerrit.
    - ie: Change-Id: I77aac9efc4fe783683b4ec3c7e7bd2e5e264b642

Git push option

<refspec>
What destination ref to update with what source

The format is **<src>:<dst>**
<src> the name of the ref you want to push
<dst> which ref is updated the push

git push origin <develop>:<master>
git push origin :<develop> (deletes develop)

```
commit 77aac9efc4fe783683b4ec3c7e7bd2e5e264b642
Author: hamzas <skander.hamza@st.com>
Date:   Thu Apr 27 18:40:01 2017 +0100

    patch for gerrit

    Change-Id: I06ad80dc8b88d61edfd98c509d9f0b07c99f67db
```

Pull request

Lab : Gerrit

Gerrit Vs pull request

- Gerrit
  - Review every commit
  - Review is done through an additional interface
  - Requires a minimum of investment
    - migration to gerrit + understanding the tool (voting, review, ..) + mastering Git
  - Defines
    - User roles: developer,maintener, integrator
    - Access right per branch
- Pull request
  - Review on branch (diff with parent)
  - Integrated on codex
  - ~zero Investment
  - Features
    - Create requests accross branches
    - Comment on files reviewed
    - Can integrates with Jenkins

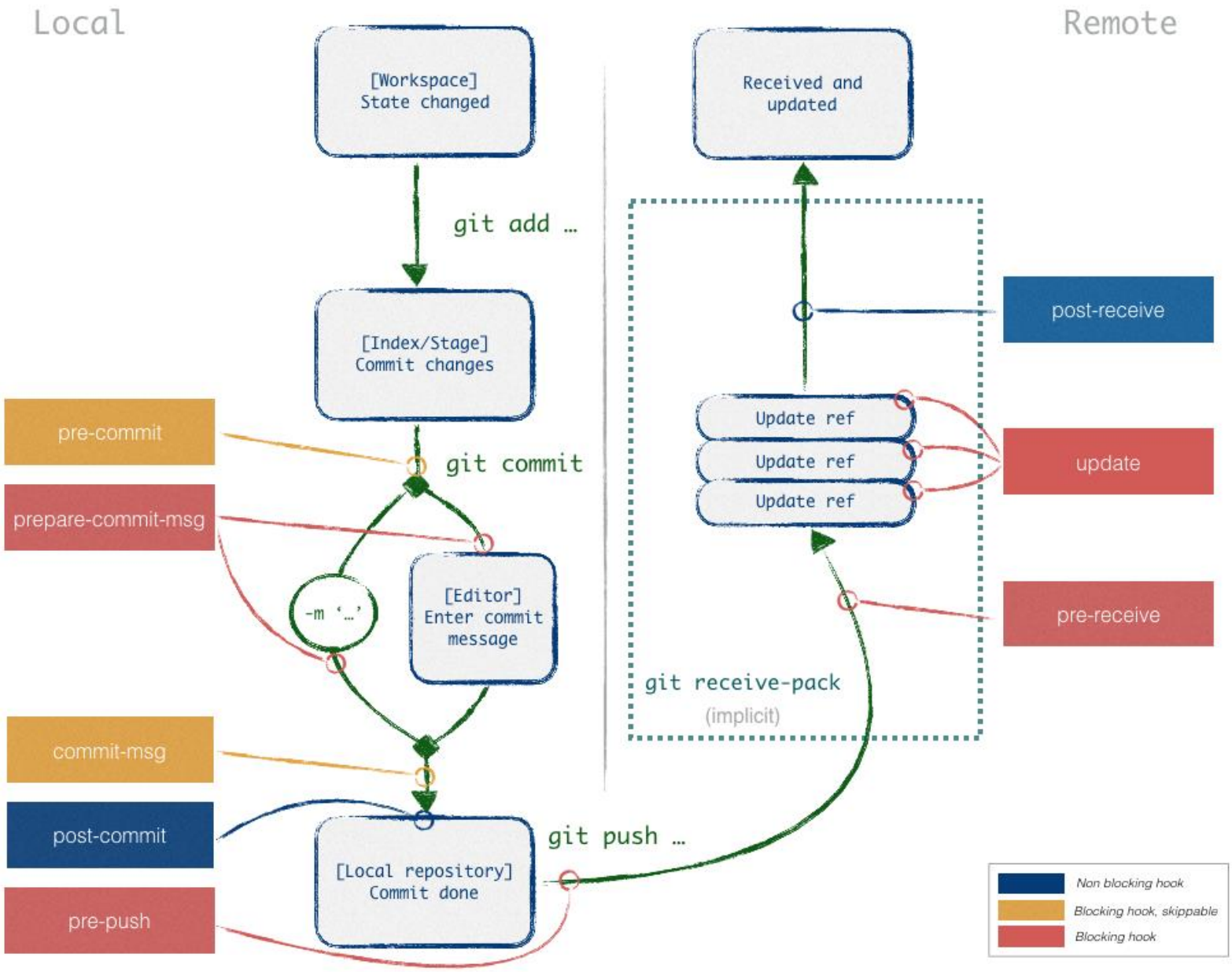
Lab : Pull Requests

Hooks

Hooks concept

- Scripts to be executed on some specific events
  - Can use any available language (perl, python, node, PHP, ruby, ...)
- Not under version control
  - By default: .git/hooks
    - Inactive samples provided (remove .sample extension to activate)
  - Can be at specific external location
    - core.hooksPath option
  - Part of a template (in a hook sub-directory)
    - --template (git init | git clone)
    - $GIT_TEMPLATE_DIR
    - init.templateDir option
- Client side & Server side hooks
  - Actions specific to client (commit, merge, ...)
  - Actions specific to server (push, pull, ...)
    - Can't manage server hooks if no access to the server (i.e. Codex)

Hooks triggers (most popular)



Hooks example

- Check that Header part of commit message if 10+ characters
- .git/hooks/commit-message
- Blocking hook
  - Return code >0 stop the process
- Code sample

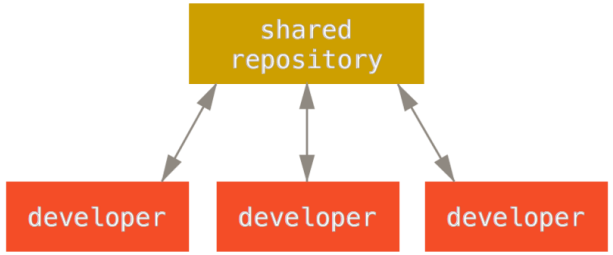
```
#!/bin/sh
length=`head -n 1 $1 | wc -l`
if [ $length -lt 10 ]; then
        echo >&2 Commit header message must be 10+ characters.
        exit 1
fi
```

Workflows

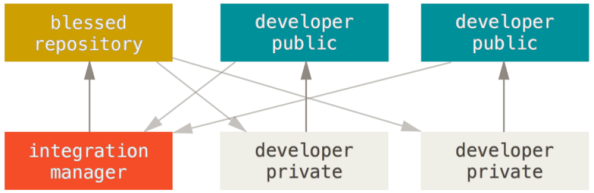
Git workflows

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

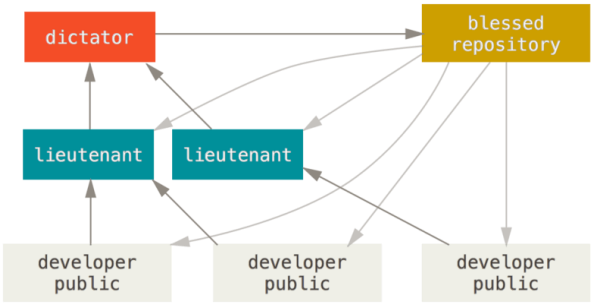
Centralized
(single branch,
squashing,
small teams)



Integration Manager
(pull requests like)



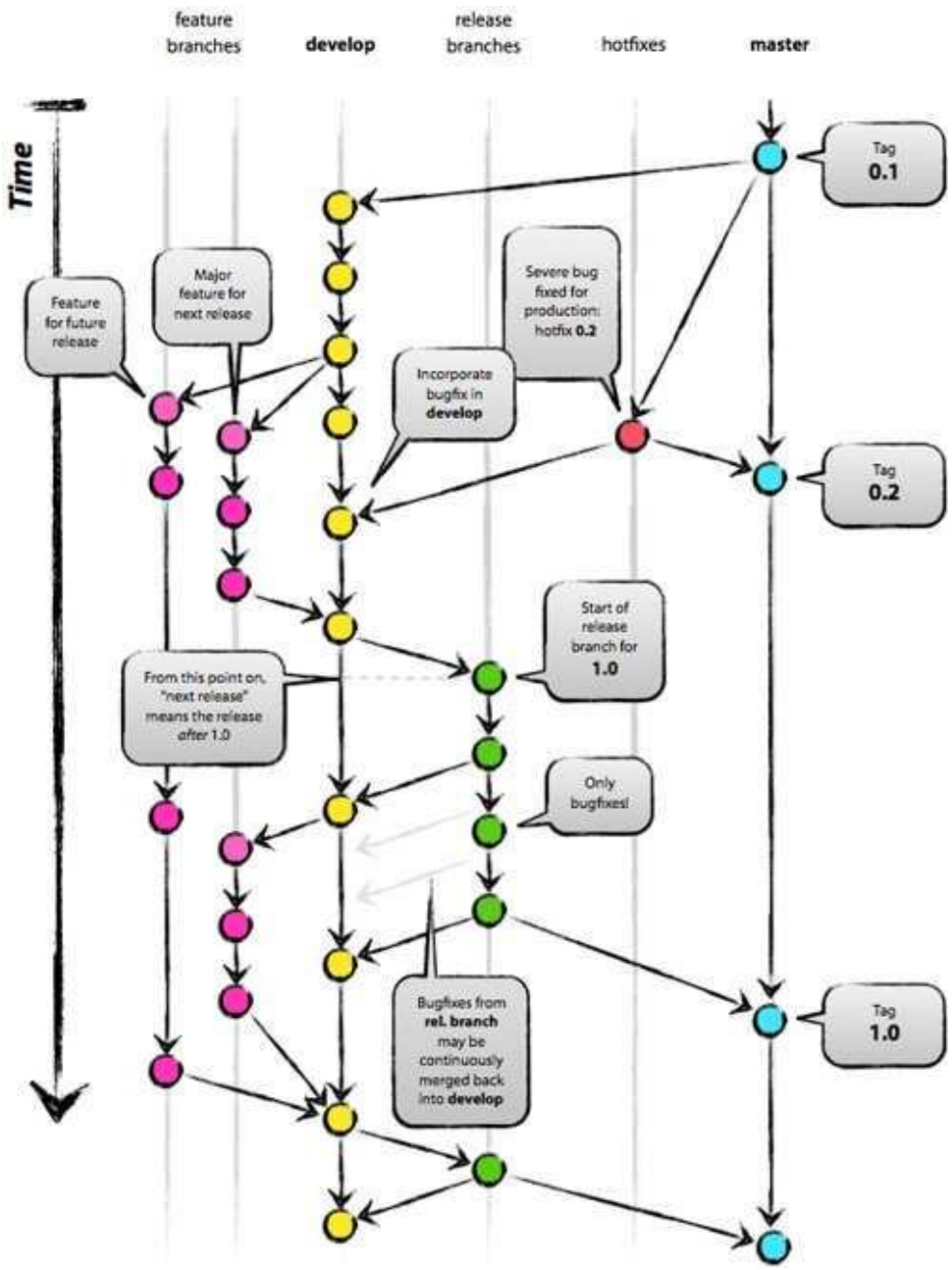
Dictator & Lieutenants
(huge projects)



life.augmented

Gitflow

Vincent Driessen - 2010



- Infinite lifetime branches
  - develop (development)
  - master (production releases)
- Pick your complexity
- short live branches
  - features/*
  - release/*
  - hotfix/*
- --no-ff merges
- Fine grain rights management allowed

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 rules (however, rules may evolve)
- Examples of templates
  - `art #xxxxxx : <short description>`
  - `<type>(<scope>): <short description> (art #xxxxxx)`
  - in footer: `Close: #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

In closing

- This is just more than the tip of the iceberg of what is Git.

- Some interesting references:
  - Git concepts simplified
  - Mastering git submodules
  - Repo
    - Using repo: https://source.android.com/source/using-repo
    - Repo – outil pour gérér des repositories git sans s'arracher les cheveux
    - Get the code using repo
  - Pro Git (free ebook)

- Finally !
  - git status
  - git command --help ☺