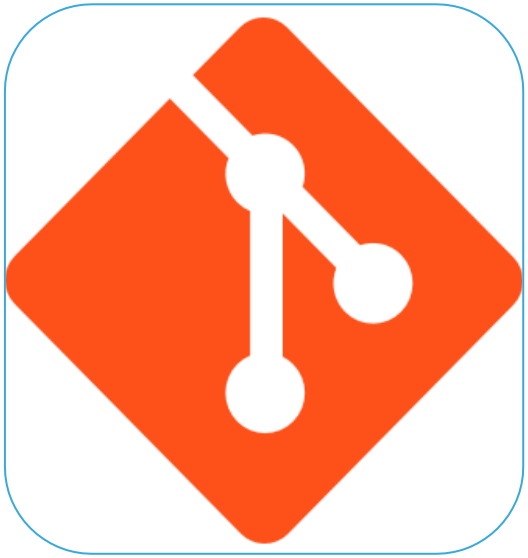
Git training

Skander Hamza
https://www.linkedin.com/in/hamzas/



Agenda

- Day 1
  - Git presentation
    - About version control
    - Why Git
  - Git basics
    - How it works
    - Git basic commands + Lab 1
    - Git branching + Lab 2
    - Merge conflict resolution + Lab 3
    - Rebasing + Lab 4

- Day 2
  - Detached head
  - Git reset & revert
  - Git reflog
  - Git additional + labs
    - Git GUI tools
    - Git patching and tagging
  - Git workflows + Lab 5
  - Git best practices

About version control

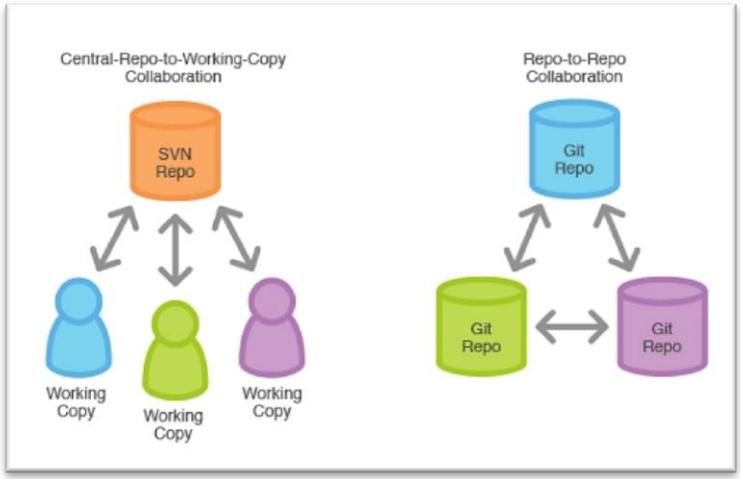
- Source code tracking and backup
  - Version control software records text files changes over time
  - Change history is saved
    - It can recall each specific version
    - It compares changes over time

  ➔ If a mistake is done, the recover is easy

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

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

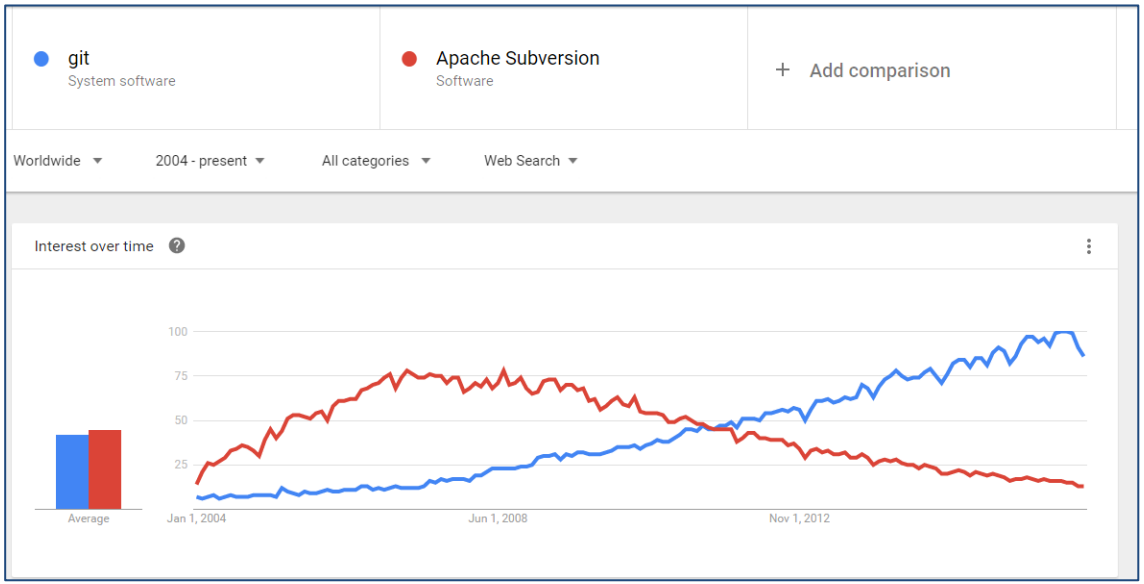
- There are two types of VCS
  - Centralized: CVS, SVN
  - Distributed : Bazaar, Git



Why Git

- 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

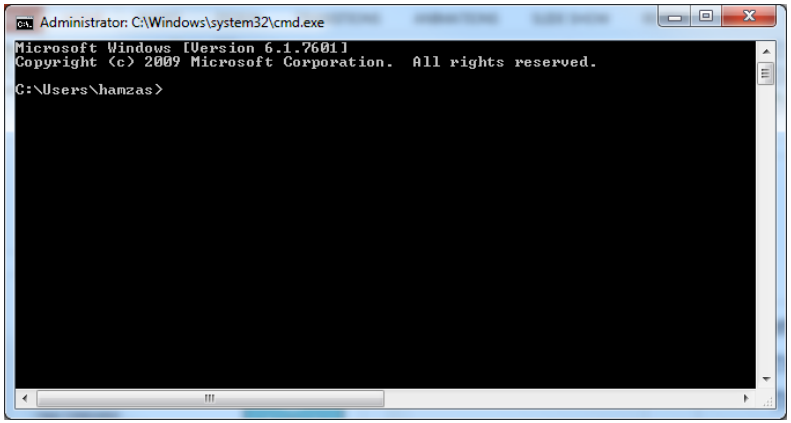
- Git is the new fast-rising star of version control system and many major open source project use Git :
  - Linux Kernel
  - Fedora
  - Android
  - VLC
  - Twitter



The command line

- There are two main ways to use Git
  - The original command line tools
  - There are many graphical user interfaces of varying capabilities
- The command line
  - The only place where you can run **all** Git commands
  - If you know how to run the command line, you can also figure how to run the GUI
    - The opposite is not necessarily true.
- Graphical clients is a matter of personal taste
  - All users will have the command-line tool installed and available

we will train on terminal



How it works

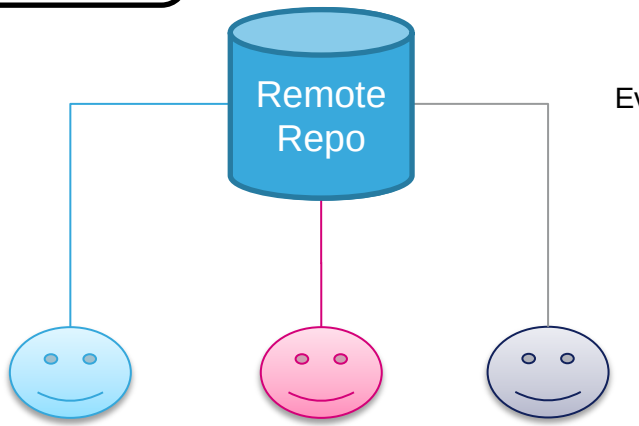
Basics:
- Origin: default remote
- Master : default branch

Someone initializes the central repo (git init)

not SVN checkout

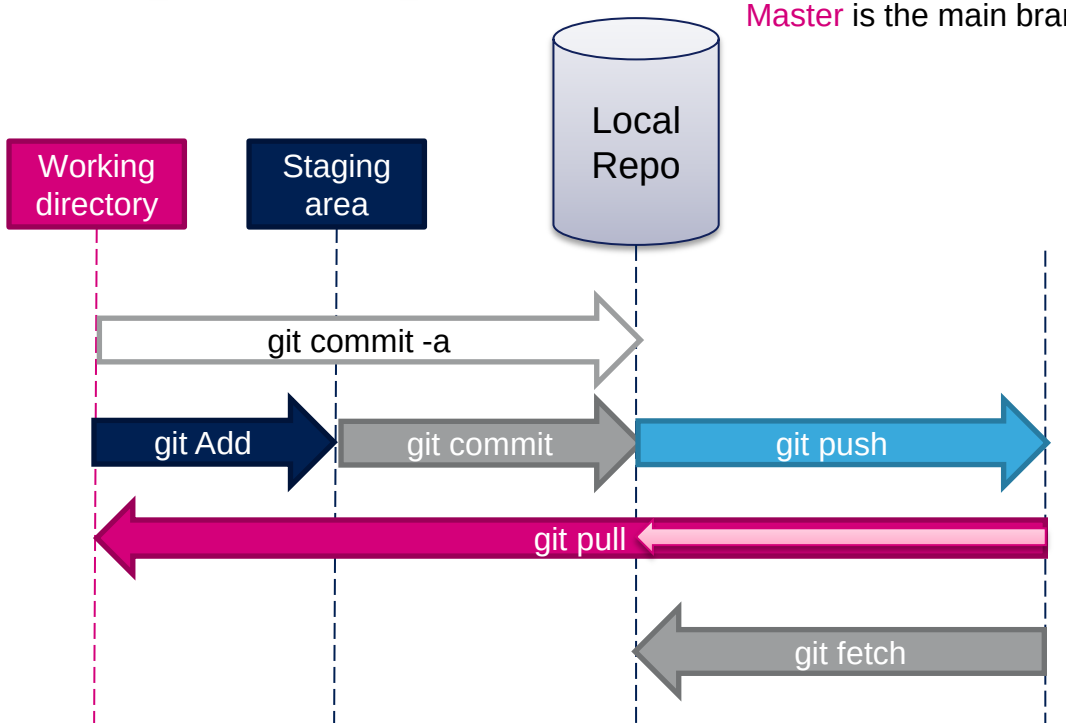
Everybody clones the central repository (git clone)

Git commands
- git init
- git clone
- git add
- git commit
- git push
- git pull
- git fetch



Git automatically adds a shortcut name origin that point back to “parent” repo.

This is the local cloned repository. Master is the main branch



Git essential commands

Basics

Use git help [command] if you're stuck

master : default branch
origin: default remote
HEAD: current point

Create Repository

Create a new local repo

$ git init

Clone existing repository

$ git clone < repo_url >

Local changes

Add files to tracked / staged

$ git add file1 file2
$ git add .

Commit

$ git commit -m "commit msg"
$ git commit -am "commit msg"

List changed / new files on local repo

$ git status

List changes on tracked files

$ git diff

Show entire history

$ git log

Show commit content

$ git show $id

Branches

Merge

Revert

Unmodifying a modified File

$ git checkout -- [file]

Unstage a file

$ git reset [file]

Change last commit

$ git commit --amend

Synchronize

Push local changes to remote

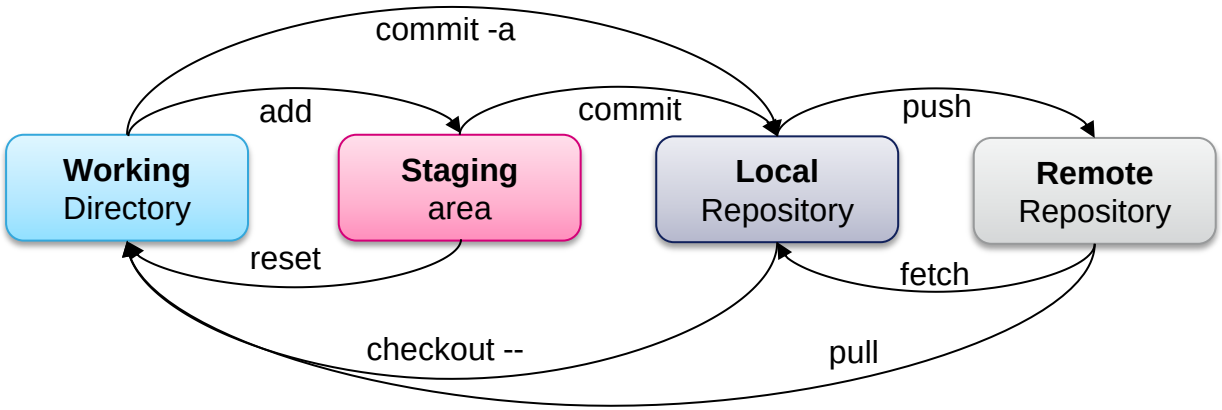
$ git push <remote> <branch>

Get the latest changes (no merge)

$ git fetch <remote>

Fetch and merge last changes

$ git pull <remote> <branch>



Install and configure Git

- Install Git
  - Ubuntu: sudo apt-get install Git
  - Windows: http://git-scm.com/download/win

- First-time Git setup
  - Configuration is done only once, You can change it at any time.
  - git config command allows you to get and set configuration variables
  - Variables are stored in gitconfig file
  - gitconfig file can be stored in three different places
    - Linux:
      - Configuration for every user in system : **/etc/gitconfig**
      - Configuration specific to user : **~/.gitconfig or ~/.config/git/config**
      - Configuration specific to project **:** **.git/config** (in the git directory)
    - Windows:
      - Configuration for every user in system : **C:\Documents and settings\All users\Application Data\git\config**
      - Configuration specific to user : **C:\Users\$USER\.gitconfig**
      - Configuration specific to project **:** **.git/config** (in the git directory)

LAB 1 – Basics

Git essential commands

Basics

Use git help [command] if you're stuck

master : default branch
origin: default remote
HEAD: current point

Create Repository

Create a new local repo

$ git init

Clone existing repository

$ git clone < repo_url >

Local changes

Add files to tracked / staged

$ git add file1 file2
$ git add .

List changed / new files on local repo

$ git status

List changes on tracked files

$ git diff

Commit

$ git commit -m "commit msg"
$ git commit -am "commit msg"

Show commit content

$ git show $id

Show entire history

$ git log

Branches

Merge

Revert

Unmodifying a modified File

$ git checkout -- [file]

Unstage a file

$ git reset [file]

Change last commit

$ git commit --amend

Synchronize

Push local changes to remote

$ git push <remote> <branch>

Get the latest changes (no merge)

$ git fetch <remote>

Fetch and merge last changes

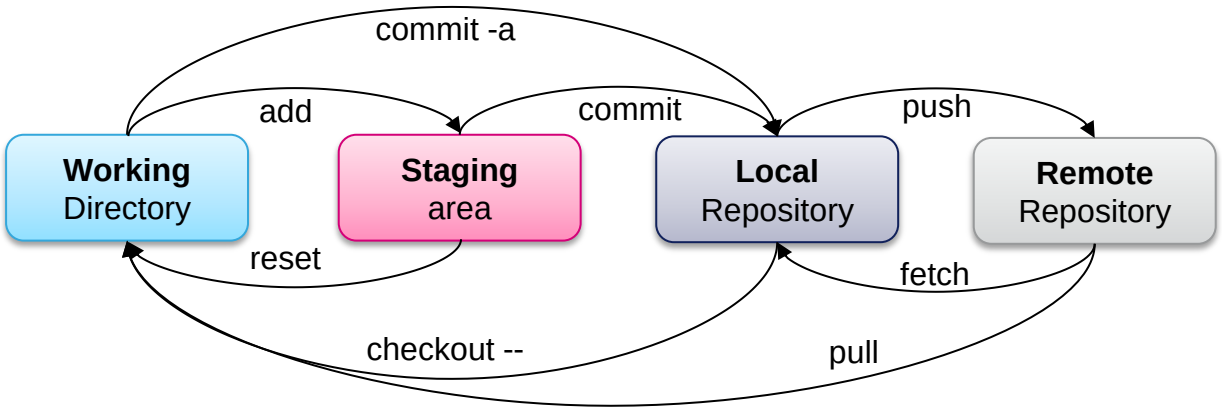
$ git pull <remote> <branch>

Add a new remote

$ git remote add <name> <url>

List remote's name and URL

$ git remote -v



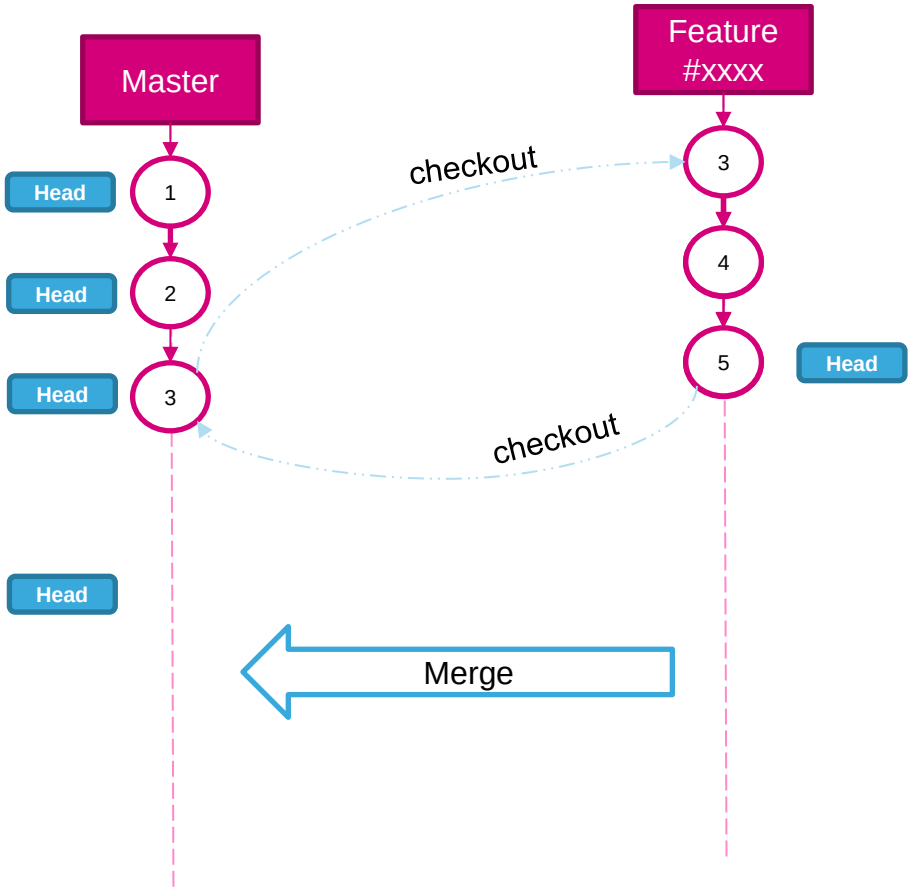
Git commands
- git branch
- git checkout
- git merge

≠ SVN checkout

Git branching

- Branching : you diverge from the main line of development and continue to work without impacting that main line.
  - Some people refer to Git's branching model as its "killer feature": lightweight, fast, easy

You can create a new branch for the feature to implement (git branch)



Then switch to that branch (checkout)

Do work (commits) in your feature branch

Now, to time to merge the work in master branch:
- Go back to master branch (checkout)
- Merge changes into master branch (merge)

Git essential commands

Basics

Use git help [command] if you're stuck

master : default branch
origin: default remote
HEAD: current point

Create Repository

Create a new local repo

$ git init

Clone existing repository

$ git clone < repo_url >

Local changes

Add files to tracked / staged

$ git add file1 file2
$ git add .

List changed / new files on local repo

$ git status

List changes on tracked files

$ git diff

Commit

$ git commit -m "commit msg"
$ git commit -am "commit msg"

Show commit content

$ git show $id

Show entire history

$ git log

Branches

Create branch named <branch>

$ git branch <branch>

Switch to a <branch>

$ git checkout <branch>

Create and checkout a new branch

$ git checkout -b <branchName>

List all branches

$ git branch -a

Delete a branch

$ git branch -D <branchTodelete>

Delete a remote branch

$ git push origin --delete <branch>

Merge

Merge <branch> into current branch

$ git merge <branch>

Revert

Unmodifying a modified File

$ git checkout -- [file]

Unstage a file

$ git reset [file]

Change last commit messages

$ git commit --amend

Synchronize

Push local changes to remote

$ git push <remote> <branch>

Get the latest changes (no merge)

$ git fetch <remote>

Fetch and merge last changes

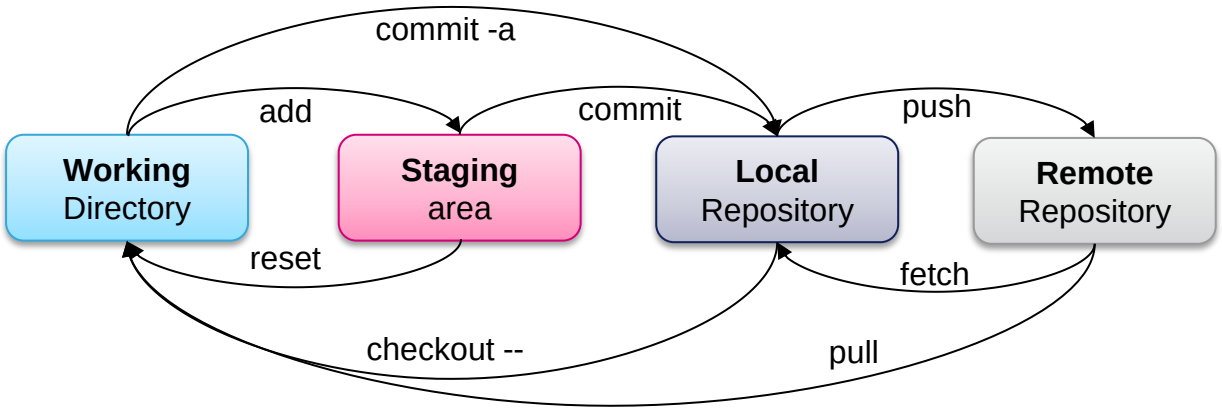
$ git pull <remote>

Add a new remote

$ git remote add <name> <url>

List remote's name and URL

$ git remote -v



LAB 2 – Branching

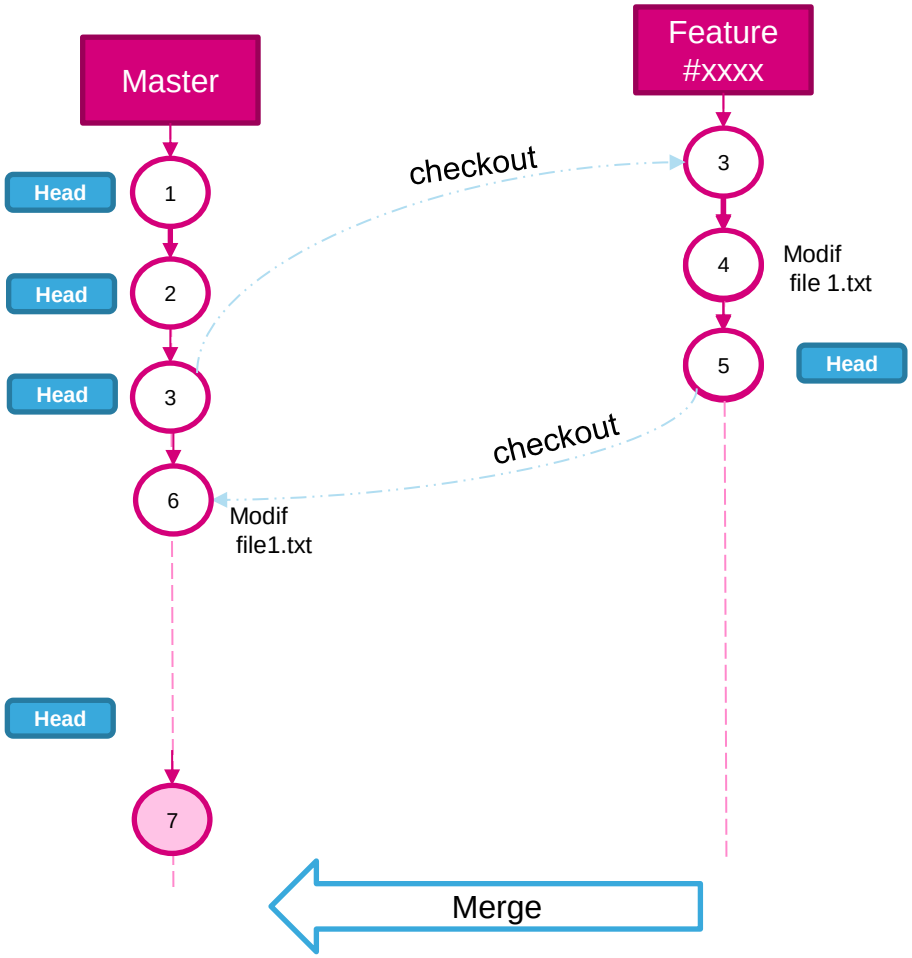
Merge conflict resolution

- Merge conflict: two people changed the same file
  - If one person decided to delete a line while the other person decided to modify it, Git simply cannot know what is correct

- Use git diff to view merge conflicts
- Use git status to list conflicting files
  - Conflicting files are listed as unmerged
- Manually fix merge conflicts into each file
  - Local changes between **<<<<< HEAD** and **=======**
  - Remote changes **=======** and **>>>>> branchName**
- Mark each file resolution using git add <file>
- To cancel the merging operation
  - Gig merge --abort

Git can also work with a large number of GUI tools

- git config --global merge.tool kdiff3
- git mergetool



Git additional commands

Tagging

Patching

Debugging

Cherry pick

Rebase

Resolve merge conflicts

To view merge conflicts

$ git diff

To discard conflicting patch

$ git reset --hard

$ git rebase --skip

After resolving conflig, merge with

$ git add [conflict_file]

$ git rebase --continue

Others

Stash

LAB 3
Merge conflict resolution

Git commands
- git rebase
- git rebase -i

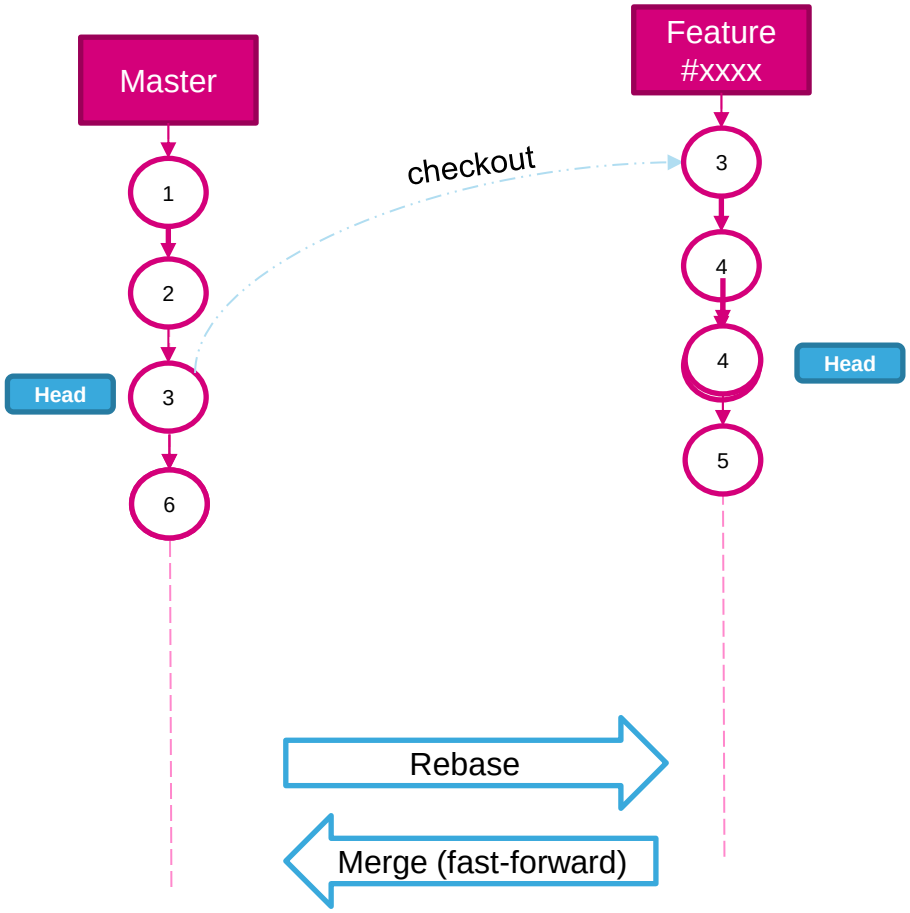
Rebasing

- There are two main ways to integrate changes from one branch into another: the merge and the **rebase**.

- Rebasing allows fast forward merge
  - It eliminates the need for merge commits
  - Results to a linear history

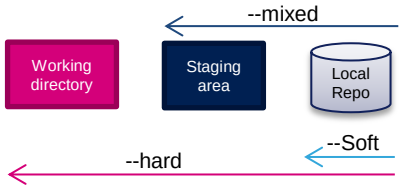
- Merge or Rebase ?
  - In general the way to get the best of both worlds is
    - Rebase local changes made and haven't shared yet
    - Merge to master
    - Push to remote

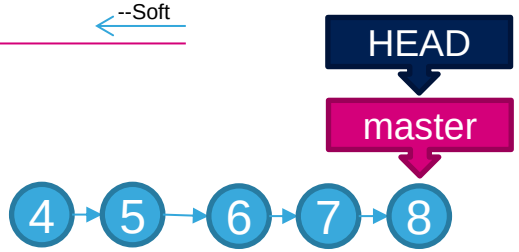
- The interactive rebase (git rebase -i)
  - Allows to clean up the commit history
    - To have a clean, meaningful history
    - Before merging into the master branch
  - Opens an editor
    - Replace the pick command before each line by
      - squash (to combine commits)
      - edit (to edit commits)



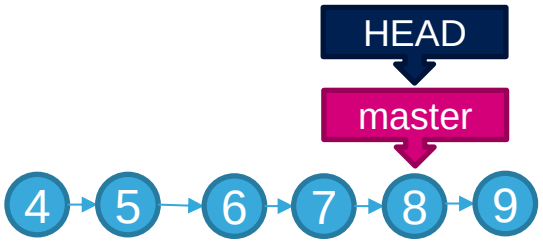
Reset & Revert

- Reset
  - Moves branch pointer to a specific commit
    - git reset HEAD~1
  - What Modifications became
    - git reset --soft : modification are kept on index
    - git reset -- mixed: the default option
    - git reset --hard: modification are LOST ⚠





- Revert
  - Undoes a commit by creating a new commit containing it's opposite.
  - This is a safe way to undo changes, as it do not re-write the commit history



reset vs revert

- git reset should be used for undoing changes on a private branch
- git revert should be used for undoing changes on a public branch

Git uses commands
- git reflog

Git reflog

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

- 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

Git additional commands

Tagging

Patching

Debugging

Cherry pick

Rebase

Rebase master content into current br

$ git rebase master

Interactively rebase current branch

$ git rebase –i <branch>

Resolve merge conflicts

To view merge conflicts

$ git diff

To discard conflicting patch

$ git reset --hard

$ git rebase --skip

After resolving conflig, merge with

$ git add [conflict_file]

$ git rebase --continue

Others

Stash

LAB 4
Rebasing

Git training – Day 2

Skander Hamza

Agenda

- Day 1
  - Git presentation
    - About version control
    - Why Git
  - Git basics
    - How it works
    - Git basic commands + Lab 1
    - Git branching + Lab 2
    - Merge conflict resolution + Lab 3
    - Rebasing + Lab 4

- Day 2
  - Detached head
  - Git reset & revert
  - Git reflog
  - Git additional + labs
    - Git GUI tools
    - Git patching and tagging
  - Git workflows + Lab 5
  - Git best practices

Git essential commands

Basics

Use git help [command] if you're stuck

master : default branch
origin: default remote
HEAD: current point

Create Repository

Create a new local repo

$ git init

Clone existing repository

$ git clone < repo_url >

Local changes

Add files to tracked / staged

$ git add file1 file2
$ git add .

List changed / new files on local repo

$ git status

List changes on tracked files

$ git diff

Commit

$ git commit -m "commit msg"
$ git commit -am "commit msg"

Show commit content

$ git show $id

Show entire history

$ git log

Branches

Create branch named <branch>

$ git branch <branch>

Switch to a <branch>

$ git checkout <branch>

Create and checkout a new branch

$ git checkout -b <branch>

List all branches

$ git branch -a

Delete a branch

$ git branch -D <branchTodelete>

Delete a remote branch

$ git push origin --delete <branch>

Merge

Merge <branch> into current branch

$ git merge <branch>

Revert

Unmodifying a modified File

$ git checkout -- [file]

Unstage a file

$ git reset [file]

Change last commit messages

$ git commit --amend

Synchronize

Push local changes to remote

$ git push <remote> <branch>

Get the latest changes (no merge)

$ git fetch <remote>

Fetch and merge last changes

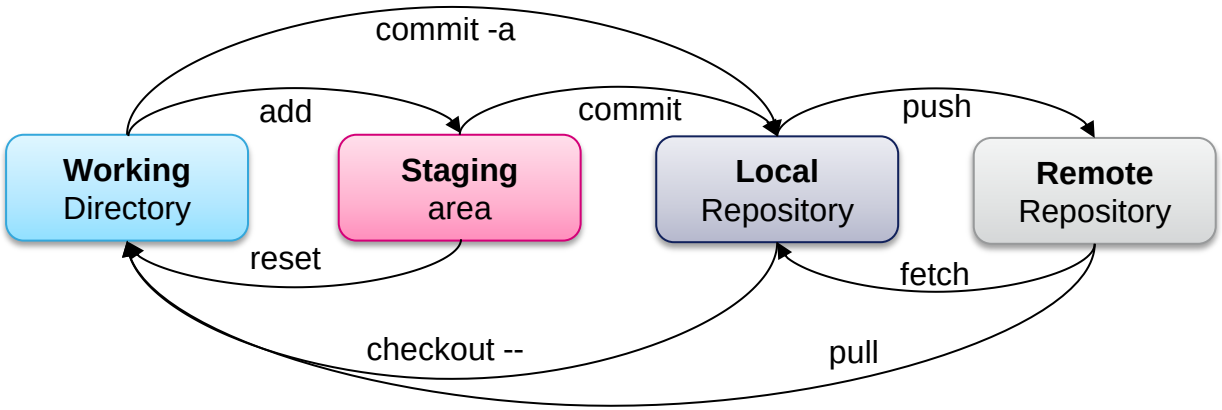
$ git pull <remote> <branch>

Add a new remote

$ git remote add <name> <url>

List remote's name and URL

$ git remote -v



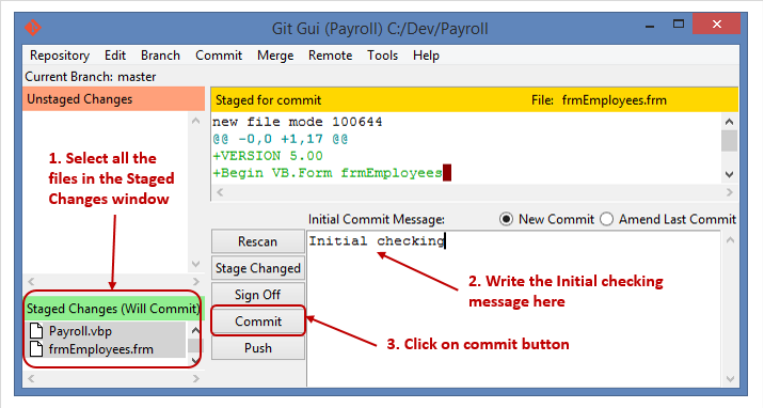
Git commands
- git gui
- gitk

GUI tools

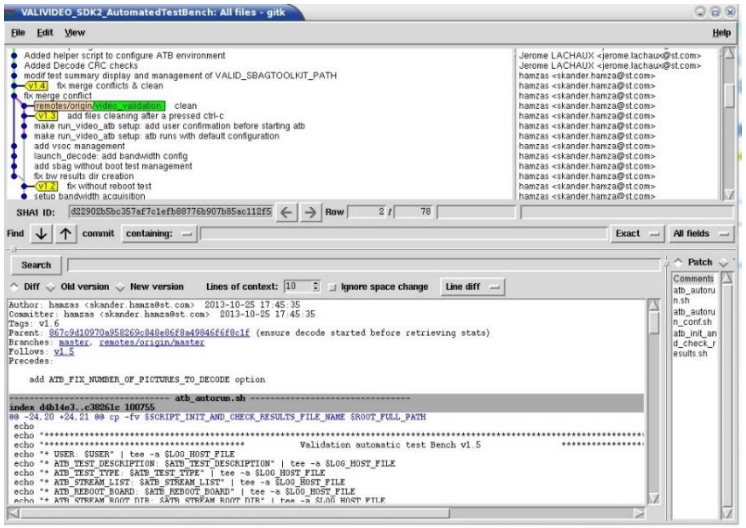
git gui & gitk

- Git comes with built-in GUI tools for committing and browsing
  - There are several third-party tools for users looking for platform-specific experience.

- 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



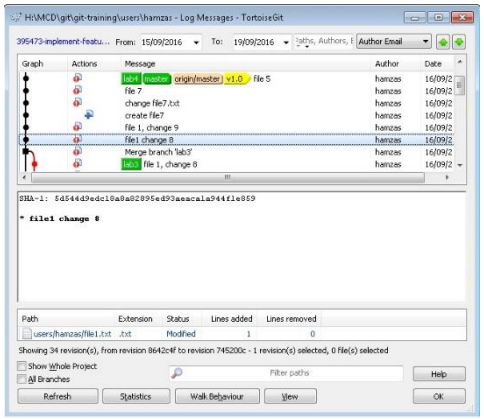
- gitk (for browsing)
  - Display changes
    - visualizing the commit graph,
    - showing commit's information
    - Showing the file in the trees of each version



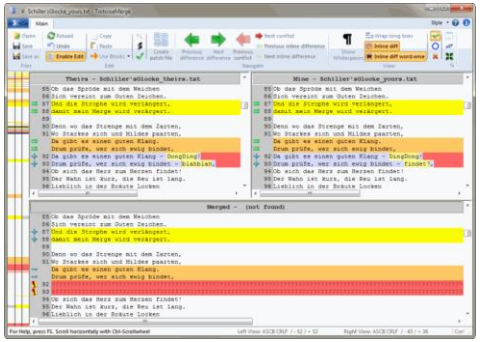
GUI tools

Tortoise

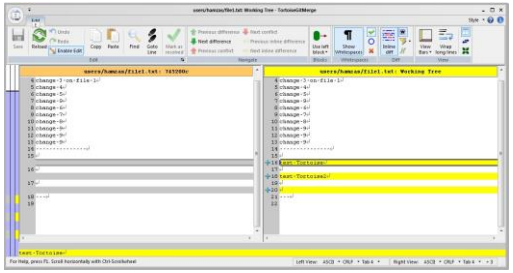
- Tortoise: the power of Git in a Windows Shell Interface
  - It's open source and can fully be build with freely available software.
- Features of Tortoisegit
  - Provides overlay icons showing the file status
  - Power context menu with all commands



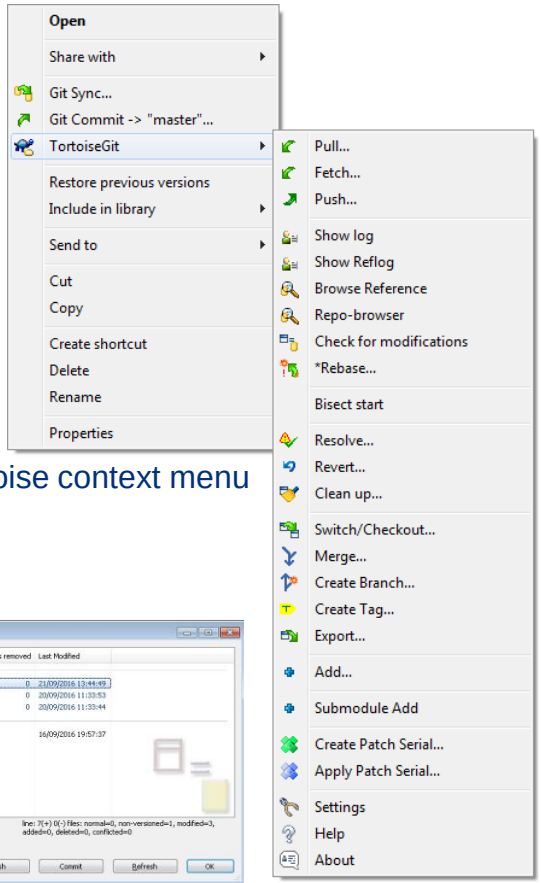
Tortoise gitLog



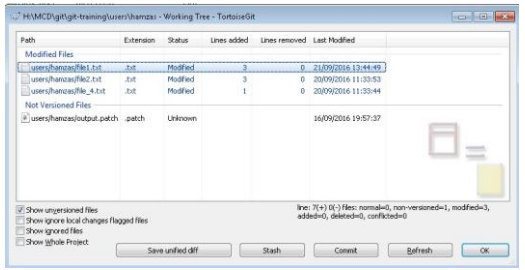
Tortoise mergetool



Tortoise gitDiff



Tortoise context menu

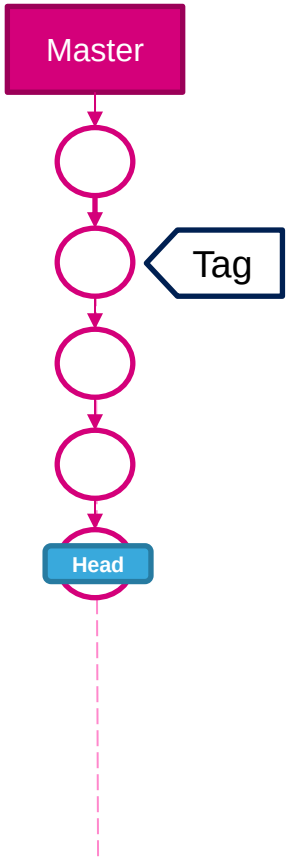


Tortoise commit dialog

Git commands
- git tag -a v1.0 – m “msg”
- git format patch
- git apply
- git am

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 v1,0 –m "version 1,0 stable”
    - Push a tag: git push <remote> tagName
    - Push all tags: git push <remote> --tags
    - Checkout tag: git checkout tagName

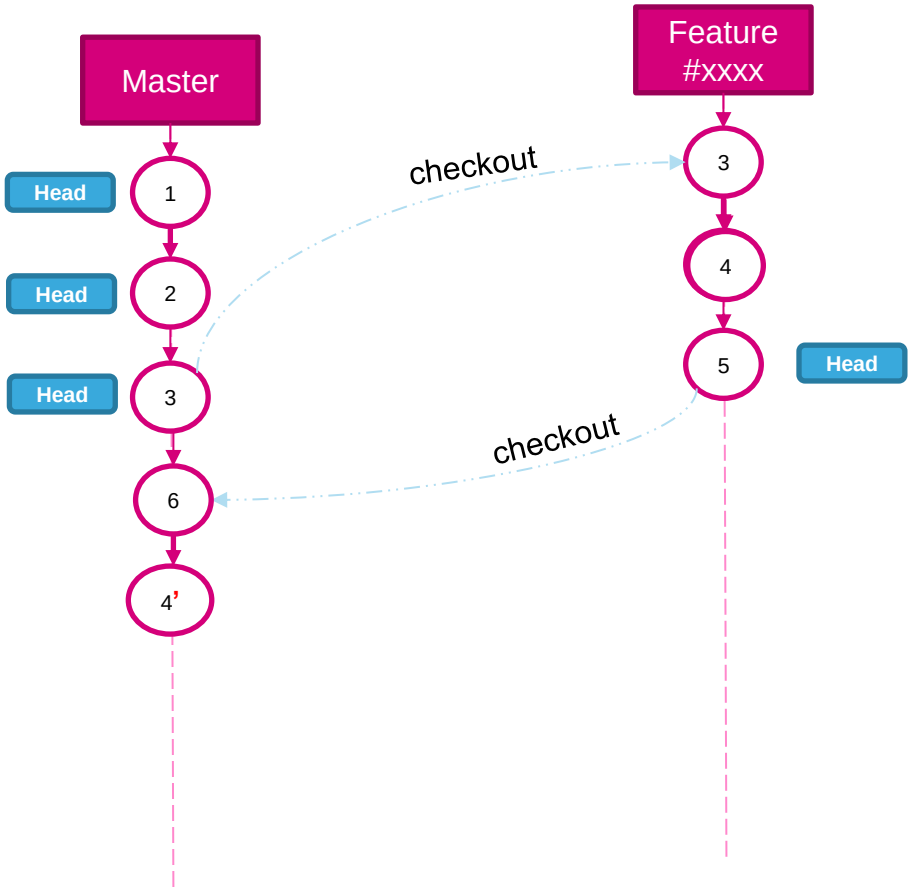


- Patching
  - Creating a patch is a good way to share changes that your are not ready to push
  - Patch is simply a concatenation of the diffs for each of the commits
    - Method 1 (no commit)
      - Create patch : git diff from-commit to-commit > output-patch-file
      - Apply patch : git apply output-patch-file (no commit)
    - Method 2 (with commit, more formal and keeps authors name)
      - Create for last 2 commits: git format-patch -2
      - git am name_of_patch_file

Git commands
- git cherry-pick “commit ID”

Cherry pick

- Takes the patch that was introduced in a commit and tries to reapply it on the branch you’re currently on.
  - git cherry-pick “commit ID”
  - Ex: git cherry-pick af24a94 (commit 4)



Git commands
- git blame
- git grep
- git bisect

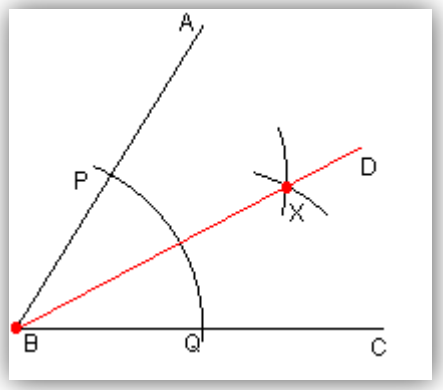
Debugging

- Git proposes a couple of tools to help issues debug

- git blame : 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 : 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
  - Basic commands: start, bad, good
    - git bisect start
    - git bisect bad HEAD
    - git bisect good $id
  - git bisect –help



Git uses commands
- git stash
- git stash pop

Git stash

Stash definition: store (something) safely in a specified 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 and you want to 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
    - Call git stash to save your work (working directory is now clean)
  - Switch branch; work on something else
  - Go back to your initial branch
    - Call git stash pop to get your modification back

Ignoring files

- 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
- Pattern are like simplified regular expressions
  - Exemple
    - *.o : ignore any files ending in .o
    - *~ : ignore any files whose names end with a ~
- Add don’t forget to add and commit the .gitignore file !

Git additional commands

Tagging

Create Tag
$ git tag –a <tagName> –m "msg"
List Tags
$ git tag
Delete Tag
$ git tag -d <tagName>
Push Tag
$ git push <remote> --tags
$ git push <remote> <tagName>

Rebase

Rebase master content into current br
$ git rebase master
Interactively rebase current branch
$ git rebase –i <branch>

Stash

Stash modification
$ git stash
Apply the modification back
$ git stash pop

Patching

Method 1 (no commit)

Share change from $id1 to $id2
$ git diff $id1 $id2 > output.patch
To apply
$ git apply output.patch

Method 2 (with commit)

Generate patch for last 2 commits
$ git format-patch -2
Apply patch
$ git am file.patch

Resolve merge conflicts

To view merge conflicts
$ git diff
To discard conflicting patch
$ git reset --hard
$ git rebase --skip
After resolving conflicts, merge with
$ git add [conflict file]
$ git rebase --continue

Debugging

Who did what
$ git blame
Find in source code
$ git grep
Finding regressions
$ git bisect

Cherry pick

Apply a change introduced by a commit
$ git cherry-pick $id

Others

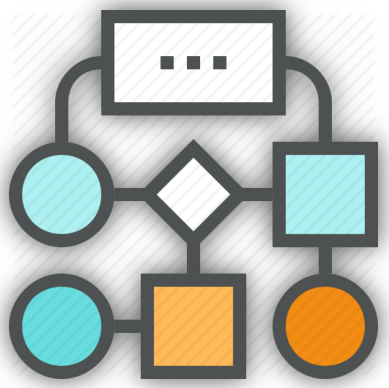
Global Ignore files
$ edit .gitignore
Configure aliases
$ git config --global alias.c "commit"
Use git help
$ git command --help

LAB 5
Workflow

Git workflow

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

- Workflow based on “squashing” commits:
  - Create a new branch for the feature to implement
  - Do work in your feature branch, committing early and often
  - Interactive rebase (squash) your commits
    - Final Commit name should be : “issue #xxx : story description”
  - Merge changes to master
  - Update master branch with upstream
  - Push to upstream



Git workflow commands

- git clone <repo>
- git checkout -b issue-01-storyDescription
- Do your work : git commit -am « commit message » @ each step
- git rebase -i master
  - Git will display an editor with the list of commits to be modified
  - Tell Git what we to do, change these lines to:
    - pick 3dcd585 Adding Comment model, migrations, spec
    - squash 9f5c362 Adding Comment controller, helper, spec
    - squash dcd4813 Adding Comment relationship with Post
    - squash 977a754 Comment belongs to a User
  - Save and close the file. We got a new editor where we can give the new commit message
    - Use the story ID and title for the subject and list the original commit in the body
    - [isssue #01] User can add a comment to a post
- git checkout master
- git merge issue-01-storyDescription
- git push origin master
  - If update rejected then you need to integrate the remote changes
    - Fetch the remote
    - git rebase origin/master

Git best practices

Commit Often, Perfect Later, Publish Once

- Review code using git diff before committing

- Commits
  - Do commit early and often
  - Do make useful commit messages
    - The commit log of a well-managed repository tells a story.
    - Do small commits (50 chars or less) than implements a single change to the code
    - Write message in the imperative present tense “fix bug” and not “fixed bug” or “fixes bug”
      - Use this kind of template : art #xxxx : short description
  - Test before commit, don’t commit half-done work

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

In closing

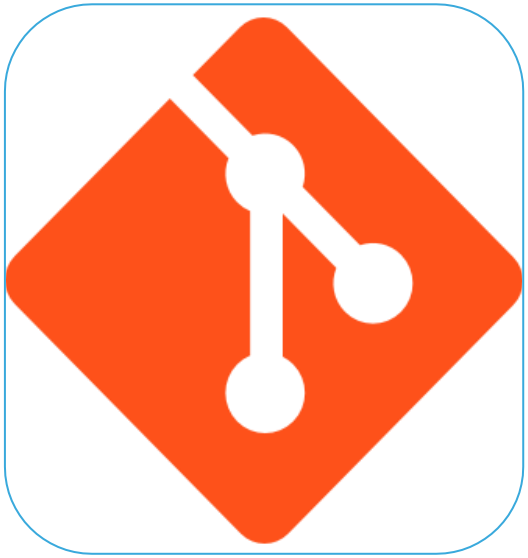
- This is just the tip of the iceberg of what is Git.

- Some interesting references:
  - What is a version control
  - Why Git
  - A Git workflow for Agile team
  - Git best practices
  - Pro Git (free ebook)
  - Git bisect

- Finally !
  - git command --help ☺

Labs

Git training – Lab 1
Basics



Lab 1,1 – basics

1. *Open Git Bash*
2. *Create “Hello” directory and change into it*
3. *Use init command to create a git repository in that directory :*
   - *Observe that a .git directory is created*
4. *Git configuration*
   - Your identity
     - git config --global user.name “Hamza Skander”
     - git config --global user.email skander.hamza@sofrecom.com
   - *List all configuration*
     - *git config --list*

Lab 1,1 – first commit

1. *Create file1.txt*
   - *Observe the output of git status. file1.txt is on the untracked area*
   - *Observe also the help proposed by git status*
2. *Use add command to add the file to the staged area*
   - *Use status command to confirm the staging success*

Lab 1,1 – basics - first commit

3. Use commit command to commit the content of the staged area
   - Observe the commit creation message:

```
[master d9151ed] add file1.txt
 1 file changed, 1 insertion(+)
 create mode 100644 users/hamzas/file1.txt
```

   - Which branch you committed to
   - What SHA-1 checksum the commit has (d9151ed)
   - How many files were changed
   - Statistics about lines added and remove

Lab 1,2 – basics

1. Make a change to file1.txt
2. Use diff command to view changes details
3. Use status command to see the working repository situation
   - file1.txt is modified and not staged
4. add the file to the staged area, confirm using the status command
5. Commit the modifications
6. Use commit --amend to modify last commit message
7. Modify again file1.txt and add it to the staged are
8. Use commit --amend to append those modification to the last commit

Lab 1,2 – basics

1. Use the log command to see all the commits you made
2. Use the show command to see a commit detail
3. Modify file1.txt
   - Check status ;
     - Observe the help proposed by git
   - use the command checkout -- to undo the modification
   - Check status
4. Modify again file1.txt
   - Check status ; add it to staged area ; check status
     - Observe that in git status command , Git tells you how to unstage a file
   - Use the command reset to unstage the file
   - Check status ; use the checkout-- command to undo the modification

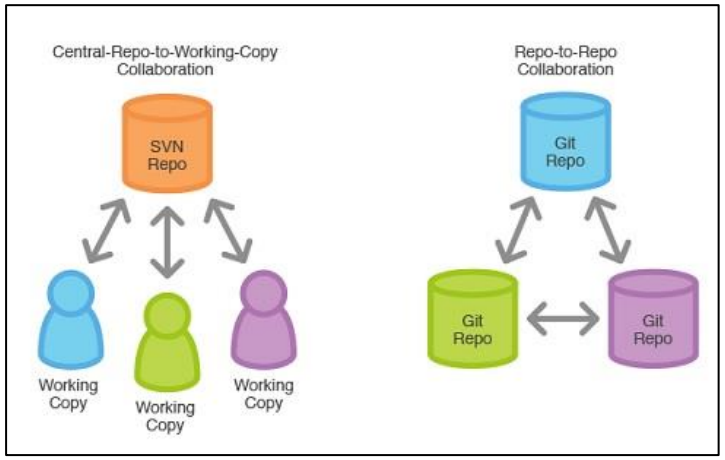
Lab 1,3 – basics – remote

1. Exit the current git directory
2. Use Clone command to clone the remote provided by the trainer
3. Move into the new clone project
4. Create a new commit including following activity
   - Into Users folder, create a folder with your name and create two new files into it
     - users/yourName/
   - Create two files (file_1.txt & file_2.txt) into that folder
   - Use add command to add your folder (use the folder name)
     - Use status command to notice that all the folder content is staged
   - Commit with following message “yourName: add file 1 & 2”

Lab 1,3 – basics – remote

1. Use Fetch remote command to get the repo modification
   - Use log command with following options to observe the local repo updates
     - git log --graph --oneline --all --decorate
   - Add the previous command as an alias and test it
     - git config --global alias.lg "log --graph --oneline --all --decorate“
     - git lg
2. Use Pull remote to update your local repo
   - Use log command to observe the local repo updates
     - use git lg alias
3. Use Push command to push your commit to remote
   - Confirm using gitk --all command

Lab 1,4 – basics – add remote



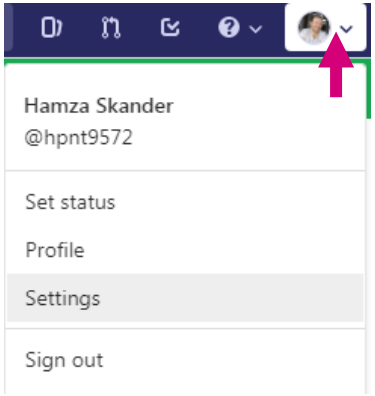
Synchronize

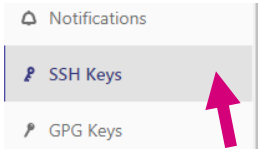
Push local changes to remote
$ git push <remote> <branch>
Get the latest changes (no merge)
$ git fetch <remote>
Fetch and merge last changes
$ git pull <remote> <branch>
List remote's name and URL
$ git remote -v
Add a new remote
$ git remote add <name> <url>

1. Use remote -v command to check your remote name and address
   - Origin is the default remote name
2. Use remote add command to add new remote named backup
   - Remote address is provided by the trainer
   - Use remote -v to confirm that the remote is added
3. Pull the new remote
4. Push to the new remote

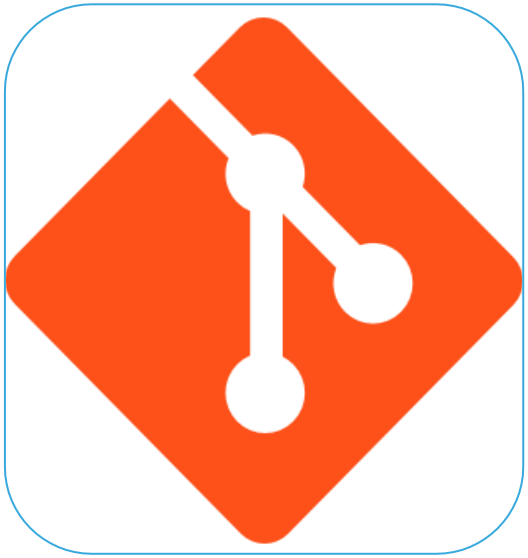
Install ssh keys on windows

- To access your Git repositories you will need to create and install SSH keys
- To do this you need to run git Bash. Open it from your start menu
  - Configure proxy
    - $ git config --global http.proxy http://10.115.64.109:8080
    - $ git config --global http.sslverify false
  - Generate ssh key through the command :
    - ssh-keygen.exe -t rsa -C “firstname.name@sofrecom.com”
    - Press enter at each command dialog (3 times)
  - Go to folder: C:\Users\username\.ssh
    - Open id_rsa.pub
    - Copy all content
- Go to https://gitlab.forge.orange-labs.fr
  - Go to your account setting
  - Click on the Add keys button
    - Paste id_rsa.pub content there





Git training – Lab 2
Branching

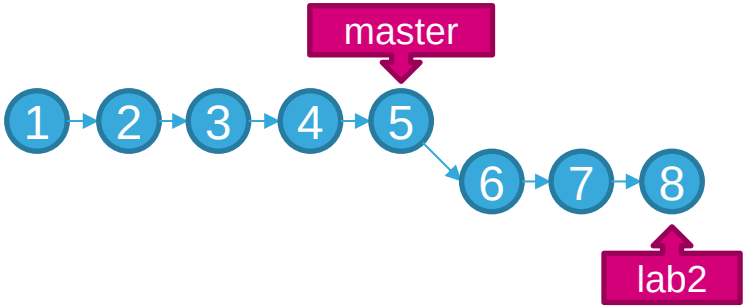


Lab 2,1 – Branching (1)

1. Use branch command to create a new branch named lab2
   - Use branch -a command to list your branches
     - remotes/origin/master : is the branch master on the remote
2. Use checkout command to switch to the new branch
   - Use status command to confirm the switch
3. Create two new commit including the following activity
   - Into users/youName
     - First commit : Modify file_2.txt ; commit
     - Second commit Create file_3.txt ; commit
4. Use checkout command to switch back to master branch

Lab 2,1 – merging ; fast forward (2)

1. Use the merge command to merge the lab2 branch work
   - Observe the **Fast-forward** merge message
   - Confirm the merge success using the log command
   - Observe the merge success using gitk command

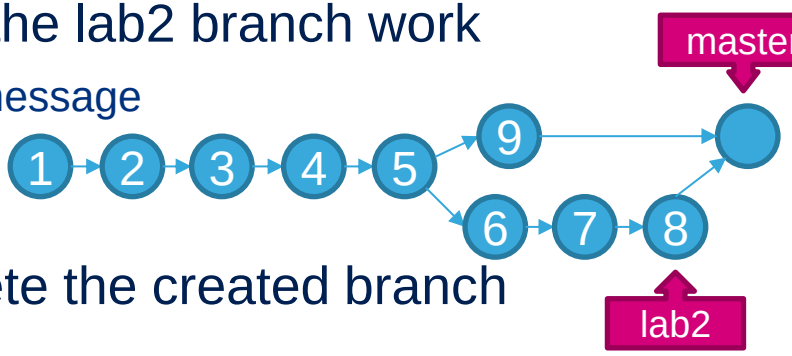


2. Use status command to ensure that your working directory is clean
   - Observe the message “your branch is ahead of 'origin/master' by xx commits”

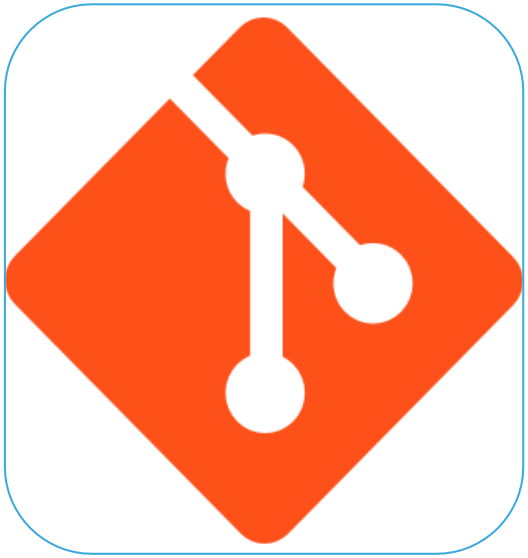
```
Updating f2c1668..462fd8c
Fast-forward
 users/hamzas/file_1.txt | 2 ++
 1 file changed, 2 insertions(+)
```

Lab 2,2 – merging ; merge commit

1. Use checkout command to switch to lab2 branch
   - Use status command to confirm the switch
2. Create one new commit including the following activity
   - Into users/yourName: : Modify file_2.txt ; commit
3. Use checkout command to switch back to master branch
4. On branch master one commit including the following activity
   - Into users/youName: First commit : modify file1.txt
5. Use the merge command to merge the lab2 branch work
   - The editor is opened: enter merge commit message
   - Observe the merge commit creation
6. Use the branch -D command to delete the created branch
   - Confirm with the command branch –a



Git training – Lab 3
Merge conflicts



Lab 3 – Merge Conflicts

1. Use checkout -b command to create a new branch named lab3
   - Use status command to confirm branch creation and switch
2. Create two new commit including the following activity
   - Into users/yourName
     - First commit : modify file1.txt ; commit
     - Second commit modify again file1.txt ; commit
3. Use checkout command to switch back to master branch
4. Create one new commit including the following activity
   - Into users/yourName
     - First commit : modify file1.txt ; commit

Lab 3 – Merge Conflicts

5. Use the merge command to merge the work done in lab3 branch
   - Notice the git merge conflict message
   - Use git status command to see the files concerned by the merge conflict

6. Let's resolve the conflict
   - Open file1.txt
     - Local changes between **<<<<< HEAD** and **=======**
     - Remote changes **=======** and **>>>>> branchName**
   - Edit the file and fix the resolution then save

7. Use add command to confirm the merge conflict resolution

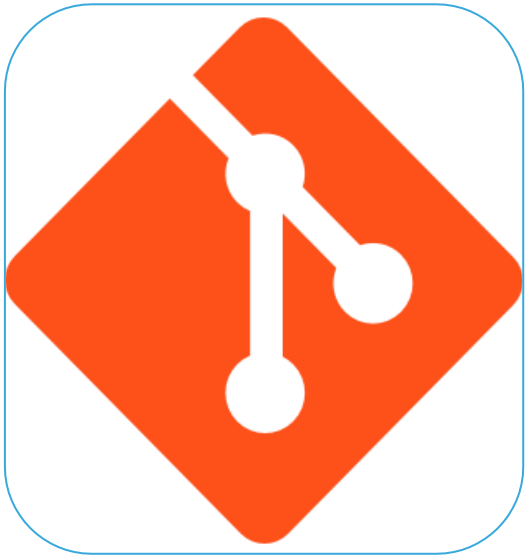
8. Use status command to view the merge status
   - Notice the message: all conflict fixed but you are still merging

Lab 3 – Merge Conflicts

9. Use commit to conclude merge
   - Notice that conflict resolution is done through a new commit : the merge commit

10. Use log commands to confirm the merge

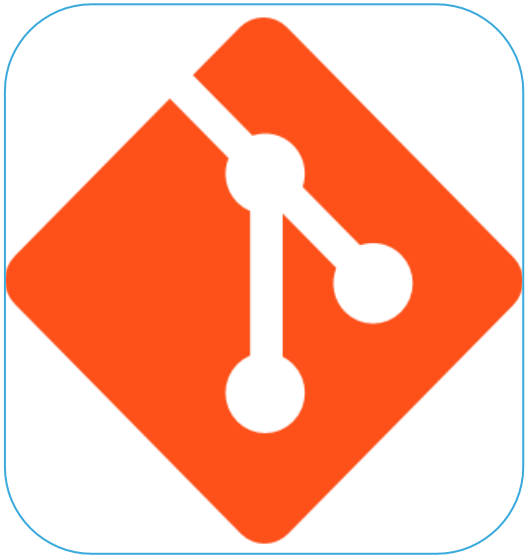
Git training – Lab 4
Rebasing



Lab 4 – Rebase

1. Use checkout -b command to create a new branch named lab4
   - Use status command to confirm branch creation and the switch to
2. Create two new commits including the following activity
   - Into users/yourName
     - First commit : modify file1.txt ; commit
     - Second commit modify again file1.txt ; commit
3. Checkout to master branch and create one commit:
   - Into users/yourName
     - First commit : Create a new file
4. Checkout lab4 branch
5. Use rebase command to rebase lab4 branch content with master
   - Use log command to view rebase operation effect

Git training – Lab 5
Workflow



Lab 5 – workflow (1)

- Clone the workflow repository
  - git clone https://gitlab.forge.orange-labs.fr/hpnt9572/workflow.git
- Let's say your task name is "*issue #1 : implement feature 1*"
  1. Checkout a new task branch name with the task id and a short descriptive title
     - git checkout -b issue1-implement-feature
       - The ID to easily associate the track with its tracker
       - The description if for a human little hint on what's in it
  2. Do you work on this branch
     - Into users/yourName : Perform **four** Commits of your choose (change 1, change 2..)
  3. use interactive rebase to squash all commits together
     - git rebase -i master

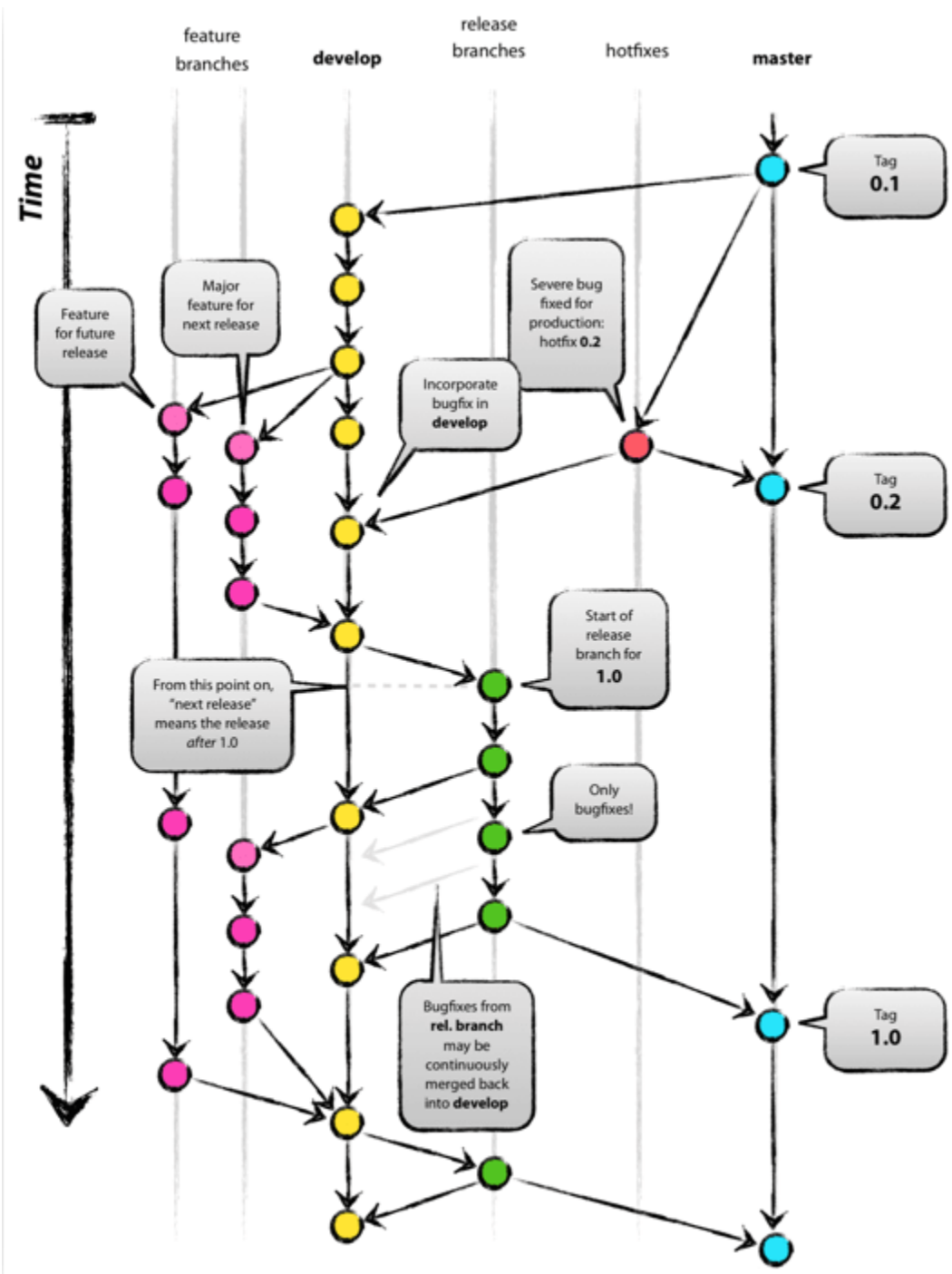
Lab 5 – workflow (2)

6. git will display an editor window with lists of your commits
    - pick 3dcd585 Adding Comment model, migrations, spec
    - pick 9f5c362 Adding Comment controller, helper, spec
    - pick 977a754 Comment belongs to a User
    - pick 9ea48e3 Comment form on Post show page
  - Now we tell git what we want to do (squash)
    - pick 3dcd585 Adding Comment model, migrations, spec
    - squash 9f5c362 Adding Comment controller, helper, spec
    - squash 977a754 Comment belongs to a User
    - squash 9ea48e3 Comment form on Post show page
  - Save and close the file
    - This will squash all commit together into one commit
- Git displays a new editor where we can **give the new commit a clear message**
  - Message must be written on the first line (lines after are commit message details)
  - We will use the task ID and tile : **issue #01 : implement feature 1**
  - Save and close the editor

Lab 5 – workflow (3)

7. Merge your changes back into master
   - git checkout master
   - git merge issue1-implement-feature
     - It must be a fast-forward merge
8. Finally push your change to upstream
   - If, meanwhile origin is updated do:
     - git fetch origin
     - git rebase origin/master
9. Use gitk --all to observe the result

Git workflow


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