Lecture 7: Version Control

Course: Biomedical Data Science

Parisa Rashidi

Fall 2018

Agenda

- Version Control
- Python (continued)
  - More python notebook example
  - Example library: MNE for processing electroencephalography (EEG) and magnetoencephalography (MEG) data

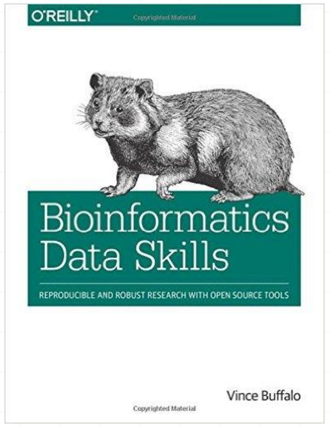
Disclaimer

The following slides are based on:

GIT for Beginners, Anthony Baire

http://people.irisa.fr/Anthony.Baire/git/git-for-beginners-handout.pdf

And Chapter 5


O'REILLY
Bioinformatics Data Skills
REPRODUCIBLE AND ROBUST RESEARCH WITH OPEN SOURCE TOOLS
Vince Buffalo

Installing git

- https://git-scm.com/book/en/v2/Getting-Started-Installing-Git

Git

- You might all have downloaded code from GitHub, how to create your own repository?
- Why to go through this trouble?

Use case 1: keeping an history

The life of your software/article is recorded from the beginning

- at any moment you can revert to a previous revision [1]
- the history is browseable, you can inspect any revision [2]
  - when was it done ?
  - who wrote it ?
  - what was change ?
  - why ?
  - in which context ?
- all the deleted content remains accessible in the history

[1] let's say your not happy with your latest changes
[2] this is useful for understanding and fixing bugs

Use case 2: working with others

VC tools help you to:

- share a collection of files with your team
- merge changes done by other users
- ensure that nothing is accidentally overwritten
- ~~know who you must blame when something is broken~~

Use case 3: branching

You may have multiple variants of the same software, materialised as **branches**, for example:

- a main branch
- a maintainance branch *(to provide bugfixes in older releases)*
- a development branch *(to make disruptive changes)*
- a release branch *(to freeze code before a new release)*

VC tools will help you to:

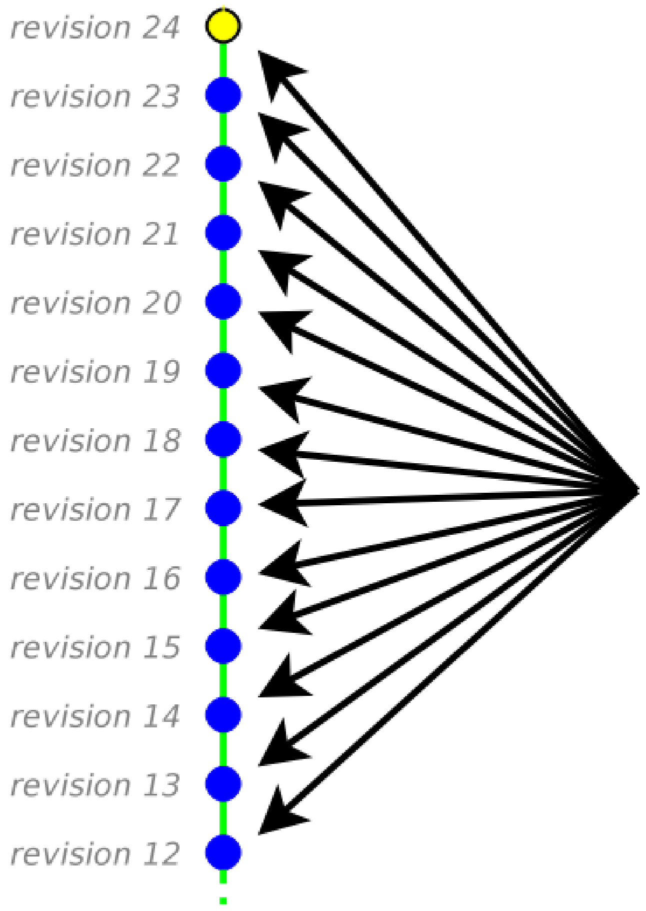
- handle multiple branches concurrently
- merge changes from a branch into another one

Use case 4: working with external contributors

VC tools help working with third-party contributors:

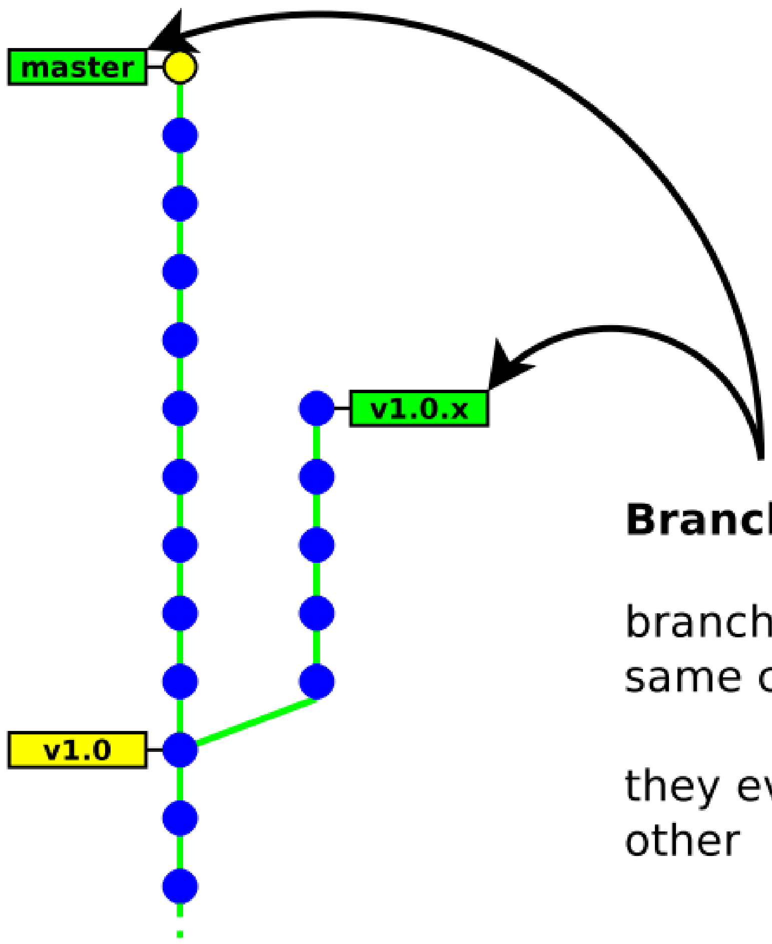
- it gives them visibility of what is happening in the project
- it helps them to submit changes (patches) and
  it helps you to integrate these patches
- forking the development of a software and merging it back into mainline[3]

Some illustrations



Revisions

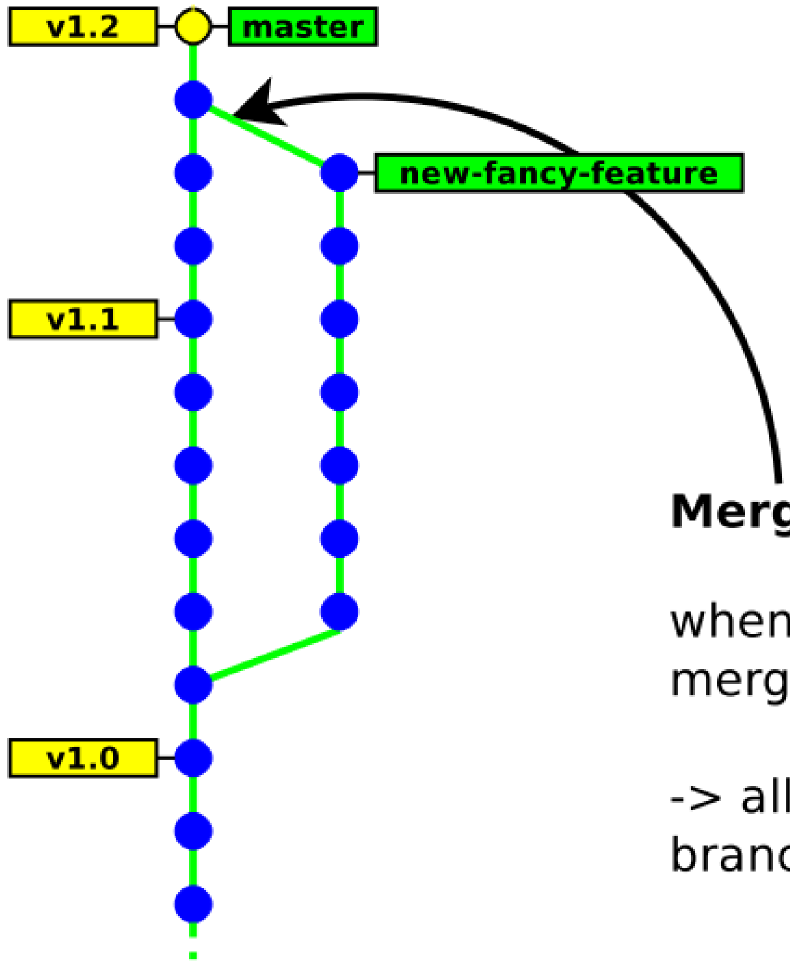
each revision:
- introduces changes from the previous revision
- has an identified author
- contains a textual message describing the changes



Branches

branches are different variants of the same collection of files

they evolve independently of each other



Merging

when the new feature is ready, it can merged back into the master branch

-> all changes done in the feature branch are imported

Architecture:

- **centralised** $\rightarrow$ everyone works on the same unique repository
- **decentralised** $\rightarrow$ everyone works on his own repository

Concurrency model:

- **lock before edit** (mutual exclusion)
- **merge after edit** (may have conflicts)

History layout:

- **tree** (merges are not recorded)
- **direct acyclic graph**

Atomicity scope: **file** vs **whole tree** **GIT**

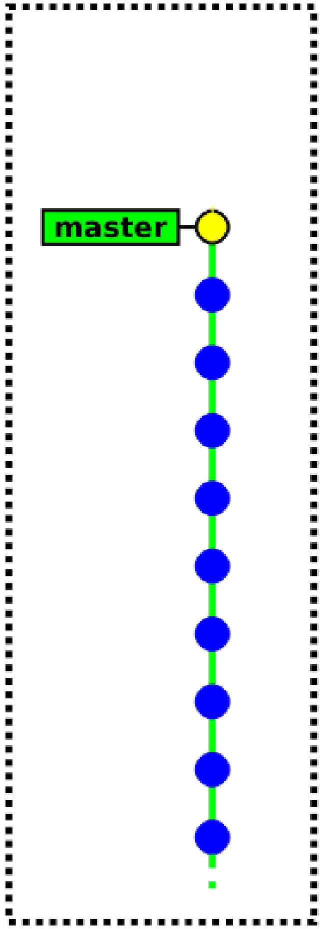
Other technical aspects

Space efficiency: storing the whole history of a project requires storage space *(storing every revision of every file)*
→ most VC tools use delta compression to optimise the space *(except Git which uses object packing instead)*

Access method: A repository is identified with a URL. VC tools offer multiple ways of interacting with remote repositories.

- dedicated protocol (*svn:// git://*)
- direct access to a local repository (*file://path* or just *path*)
- direct access over SSH (*ssh:// git+ssh:// svn+ssh://*)
- over http (*http:// https://*)

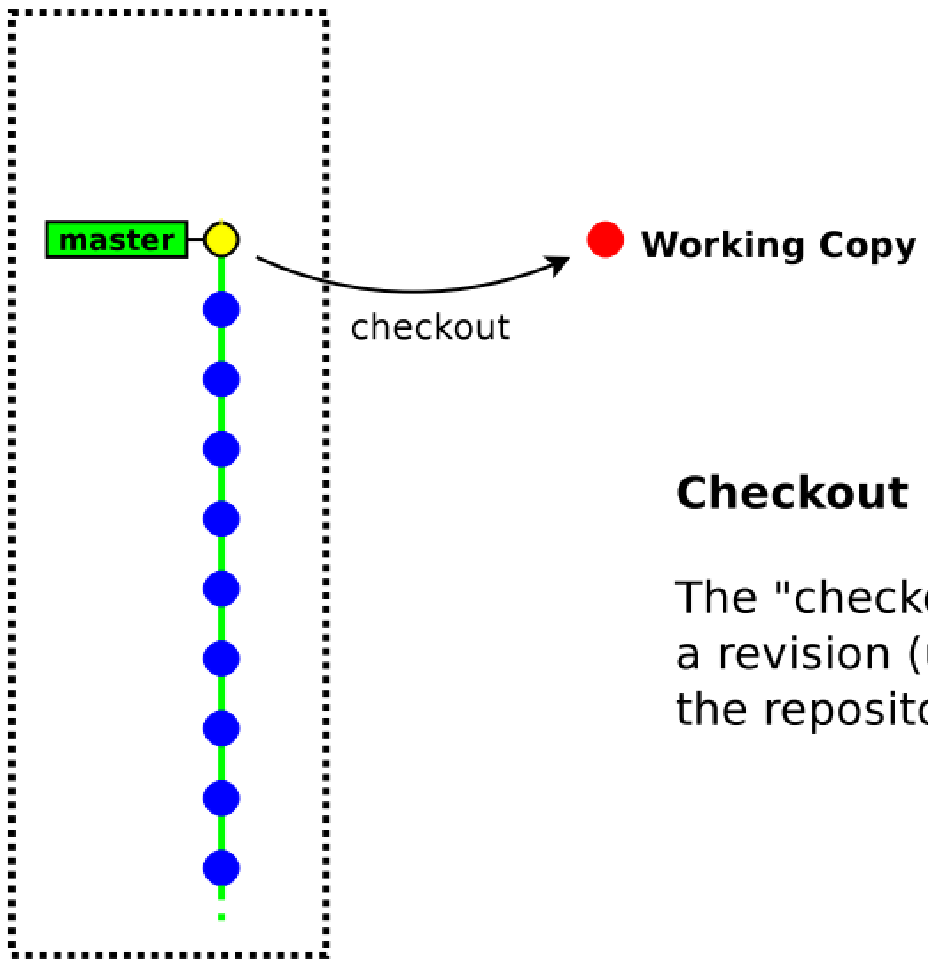
Creating new revisions



A repository is an opaque entity,
it cannot be edited directly

We will first need to extract
a local copy of the files

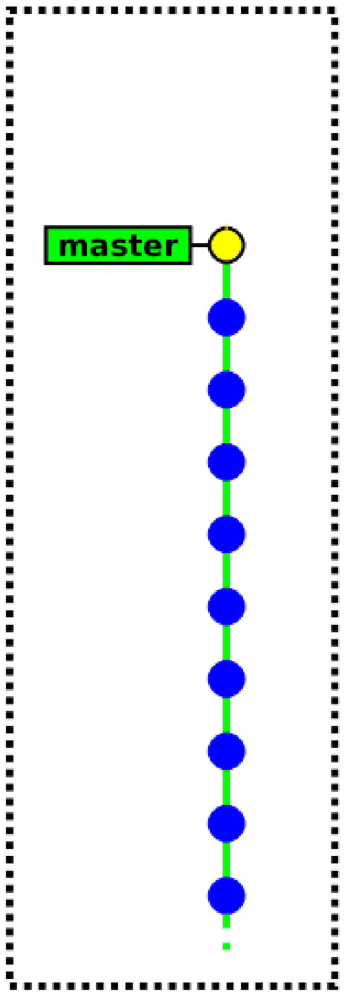
Creating new revisions

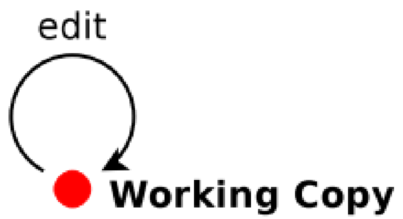


Checkout

The "checkout" command extracts a revision (usually the latest) from the repository.

Creating new revisions



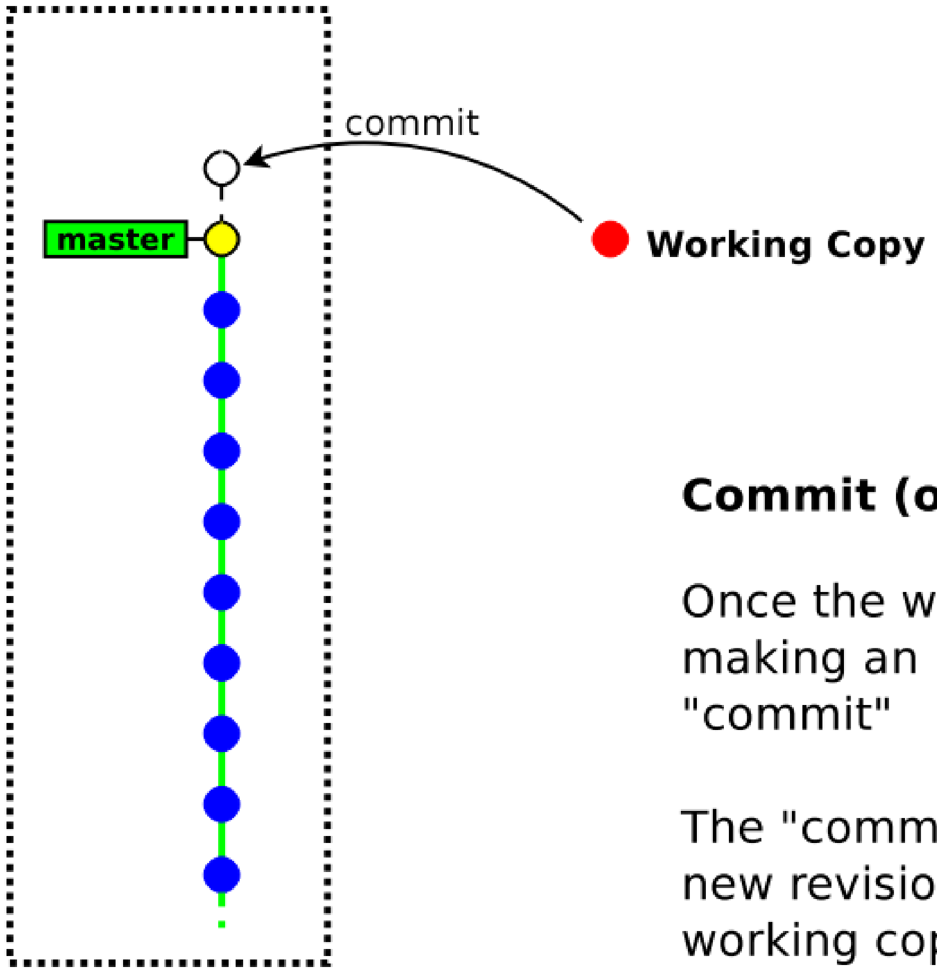


Edition

The working copy is hosted in the local filesystem

It can be edited with any editor, it can be compiled, ...

Creating new revisions



Commit (or Checkin)

Once the working copy is ready for making an new revision, we do a "commit"

The "commit" command creates a new revision from the current working copy

What shall be stored into the repository ?

You should store all files that are not generated by a tool:

- source files (`.c .cpp .java .y .l .tex` . . .)
- build scripts / project files (`Makefile configure.in Makefile.am CMakefile.txt wscript .sln`)
- documentation files (`.txt README` . . .)
- resource files (images, audio, . . .)

You should not store generated files
(*or you will experience many unneccessary conflicts*)

- `.o .a .so .dll .class .jar .exe .dvi .ps .pdf`
- source files / build scripts when generated by a tool (like autoconf, cmake, lex, yacc)

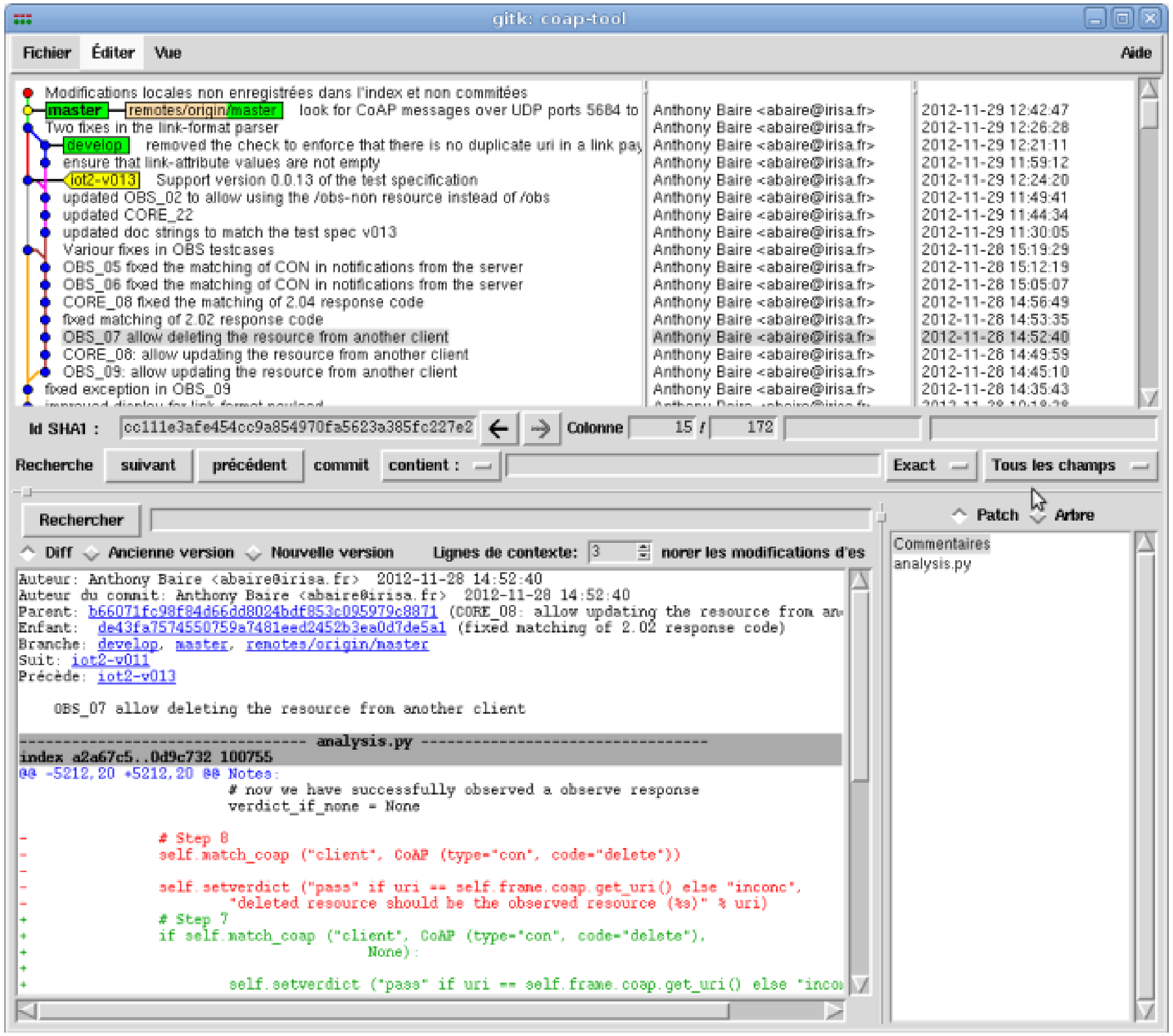
Guidelines for committing

- commit often
- commit independent changes in separate revisions
- in commit messages, describe the rationale behind of your changes (*it is often more important than the change itself*)

Git Commands

Version Control Layer	Local commands	**add** annotate apply archive bisect blame **branch** check-attr **checkout** cherry-pick **clean commit diff** filter-branch grep **help init log merge mv** notes rebase rerere **reset** revert **rm** shortlog show-branch **stash status** submodule **tag** whatchanged
	Sync with other repositories	**am** bundle **clone** daemon fast-export fast-import **fetch format-patch** http-backend http-fetch http-push imap-send mailsplit **pull push** quiltimport **remote** request-pull send-email shell update-server-info
	Sync with other VCS	archimport cvsexportcommit cvsimport cvsserver **svn**
	GUI	citool **difftool gitk gui** instaweb **mergetool**
VC Low-Level Layer		checkout-index check-ref-format cherry commit-tree **describe** diff-files diff-index diff-tree fetch-pack fmt-merge-msg for-each-ref fsck **gc** get-tar-commit-id ls-files **ls-remote** ls-tree mailinfo merge-base merge-file merge-index merge-one-file mergetool--lib merge-tree mktag mktree **name-rev** pack-refs parse-remotes patch-id prune read-tree receive-pack reflog replace rev-list rev-parse send-pack **show show-ref** sh-setup stripspace symbolic-ref update-index update-ref upload-archive **verify-tag** write-tree
Utilities		**config** var web--browse
Database Layer		cat-file count-objects hash-object index-pack pack-objects pack-redundant prune-packed relink repack show-index unpack-file unpack-objectsupload-pack verify-pack
Database (blobs, trees, commits, tags)		

Git GUIs: `gitk` → browsing the history



```
Modifications locales non enregistrées dans l'index et non commitées
master  remotes/origin/master  look for CoAP messages over UDP ports 5684 to   Anthony Baire <abaire@irisa.fr>   2012-11-29 12:42:47
Two fixes in the link-format parser                                             Anthony Baire <abaire@irisa.fr>   2012-11-29 12:26:28
  develop  removed the check to enforce that there is no duplicate uri in a link pay  Anthony Baire <abaire@irisa.fr>   2012-11-29 12:21:11
  ensure that link-attribute values are not empty                               Anthony Baire <abaire@irisa.fr>   2012-11-29 11:59:12
iot2-v013  Support version 0.0.13 of the test specification                     Anthony Baire <abaire@irisa.fr>   2012-11-29 12:24:20
  updated OBS_02 to allow using the /obs-non resource instead of /obs           Anthony Baire <abaire@irisa.fr>   2012-11-29 11:49:41
  updated CORE_22                                                               Anthony Baire <abaire@irisa.fr>   2012-11-29 11:44:34
  updated doc strings to match the test spec v013                               Anthony Baire <abaire@irisa.fr>   2012-11-29 11:30:05
  Variour fixes in OBS testcases                                                Anthony Baire <abaire@irisa.fr>   2012-11-28 15:19:29
  OBS_05 fixed the matching of CON in notifications from the server             Anthony Baire <abaire@irisa.fr>   2012-11-28 15:12:19
  OBS_06 fixed the matching of CON in notifications from the server             Anthony Baire <abaire@irisa.fr>   2012-11-28 15:05:07
  CORE_08 fixed the matching of 2.04 response code                              Anthony Baire <abaire@irisa.fr>   2012-11-28 14:56:49
  fixed matching of 2.02 response code                                          Anthony Baire <abaire@irisa.fr>   2012-11-28 14:53:35
  OBS_07 allow deleting the resource from another client                        Anthony Baire <abaire@irisa.fr>   2012-11-28 14:52:40
  CORE_08: allow updating the resource from another client                      Anthony Baire <abaire@irisa.fr>   2012-11-28 14:49:59
  OBS_09: allow updating the resource from another client                       Anthony Baire <abaire@irisa.fr>   2012-11-28 14:45:10
fixed exception in OBS_09                                                       Anthony Baire <abaire@irisa.fr>   2012-11-28 14:35:43
```

Id SHA1 : cc111e3afe454cc9a854970fa5623a385fc227e2 Colonne 15 / 172

Lignes de contexte: 3

```
Auteur: Anthony Baire <abaire@irisa.fr>  2012-11-28 14:52:40
Auteur du commit: Anthony Baire <abaire@irisa.fr>  2012-11-28 14:52:40
Parent: b66071fc98f84d66dd8024bdf853c095979c8871 (CORE_08: allow updating the resource from an
Enfant:  de43fa7574550759a7481eed2452b3ea0d7de5a1 (fixed matching of 2.02 response code)
Branche: develop, master, remotes/origin/master
Suit: iot2-v011
Précède: iot2-v013

    OBS_07 allow deleting the resource from another client

--------------------------------- analysis.py ---------------------------------
index a2a67c5..0d9c732 100755
@@ -5212,20 +5212,20 @@ Notes:
                        # now we have successfully observed a observe response
                        verdict_if_none = None

-               # Step 8
-               self.match_coap ("client", CoAP (type="con", code="delete"))
-
-               self.setverdict ("pass" if uri == self.frame.coap.get_uri() else "inconc",
-                       "deleted resource should be the observed resource (%s)" % uri)
+               # Step 7
+               if self.match_coap ("client", CoAP (type="con", code="delete"),
+                                       None):
+
+                       self.setverdict ("pass" if uri == self.frame.coap.get_uri() else "incon
```

GUI Tools

- GitHub Mac (OS X). Is Github's client made to work well with repositories on Github. There's also a Windows version.
- SourceTree (Windows, OS X). Very nice interface.
- Git-cola (Windows, OS X, Linux)

Create a new repository

`git init` *myrepository*

This command creates the directory *myrepository*.

- the repository is located in *myrepository/*`.git`
- the (initially empty) working copy is located in *myrepository/*

```
$ pwd
/tmp
$ git init helloworld
Initialized empty Git repository in /tmp/helloworld/.git/
$ ls -a helloworld/
.  ..  .git
$ ls helloworld/.git/
branches  config  description  HEAD  hooks  info  objects  refs
```

Note: The `/.git/` directory contains your whole history,
⚠ **do not delete it**[5]

[5] unless your history is merged into another repository

Commit your first files

```
git add file

git commit [ -m message ]
```

```
$ cd helloworld
$ echo 'Hello World!' > hello
$ git add hello
$ git commit -m "added file 'hello'"
[master (root-commit) e75df61] added file 'hello'
 1 files changed, 1 insertions(+), 0 deletions(-)
 create mode 100644 hello
```

Note: “master” is the name of the default branch created by `git init`

The staging area (aka the "index")

Usual version control systems provide two spaces:

- the **repository**
  (the whole history of your project)

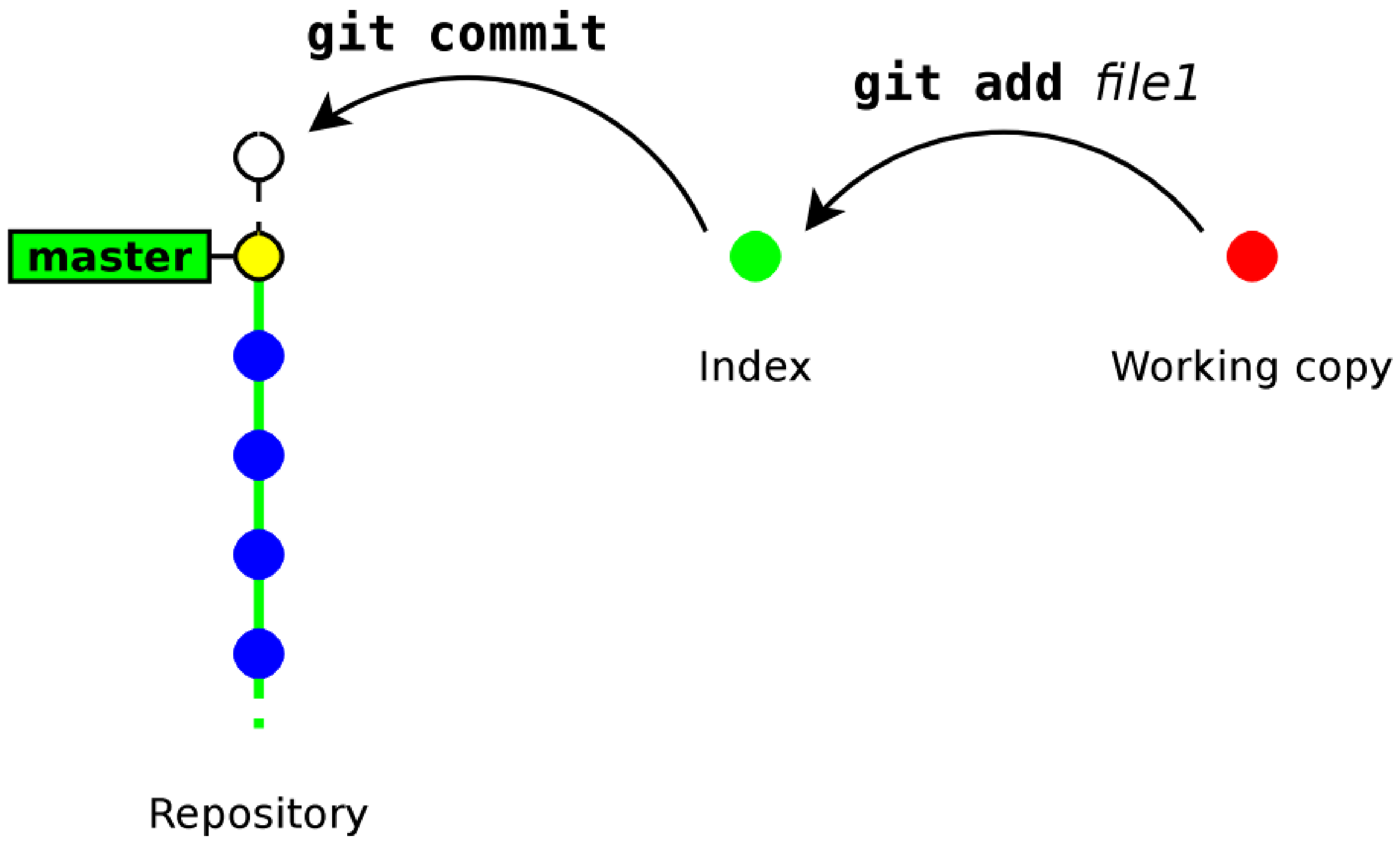
- the **working tree** (or **local copy**)
  (the files you are editing and that will be in the next commit)

Git introduces an intermediate space : the **staging area** (also called **index**)

The index stores the files scheduled for the next commit:

- `git add` *files* $\rightarrow$ copy files into the index
- `git commit` $\rightarrow$ commits the content of the index

The staging area (aka the “index”)



Update a file

```
$ echo 'blah blah blah' >> hello
$ git commit
# On branch master
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
# modified:   hello
#
no changes added to commit (use "git add" and/or "git commit -a")
```

Git complains because the index is unchanged (nothing to commit)

→ We need to run `git add` to copy the file into the index

```
$ git add hello
$ git commit -m "some changes"
[master f37f2cf] some changes
 1 files changed, 1 insertions(+), 0 deletions(-)
```

Bypassing the index[6]

Running `git add` & `git commit` for every iteration is tedious.

GIT provides a way to bypass the index.

`git commit` *file1* [*file2* . . .]

This command commits files (or dirs) directly from the working tree

Note: when bypassing the index, GIT ignores new files:

- "`git commit .`" commits only files that were present in the last commit (updated files)
- "`git add .  && git commit`" commits everything in the working tree (including new files)

Deleting files

`git rm` *file*

→ remove the file from the index and from the working copy

`git commit`

→ commit the index

```
$ git rm hello
rm 'hello'
$ git commit -m "removed hello"
[master 848d8be] removed hello
 1 files changed, 0 insertions(+), 3 deletions(-)
 delete mode 100644 hello
```