



# Digital Humanities and Digital Knowledge

Introductory Activities 2018-19

Tuesday, October 30<sup>th</sup>

## Tools for Developers

Francesco Poggi

*fpoggi@cs.unibo.it*

Department of Computer Science and Engineering  
University of Bologna - Italy



# Outline

## Version Control System (VCS)

- Git (basic concepts and workflow)
- GitHub
- Handson

## Digital Object Identifiers (DOI)

- Basic concepts
- Crossref
- FigShare



# What is version control?

**Version control** is a system that records changes to a file or set of files over time so that you can recall specific versions later.

A **Version Control System (VCS)** allows you to:

- revert selected files (or an entire project) back to a previous state
- compare changes over time
- see who last modified something that might be causing a problem, who introduced an issue and when
- recover files with very little overhead
- and more...



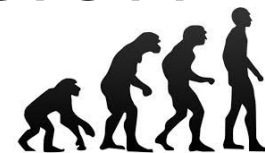
# Problem

What is “version control”, and why should you care? Version control is a **system that records changes** to a file or set of files **over time** so that you can recall specific versions later. Software source code are files that are often version controlled, though in reality you can do this with nearly any type of file on a computer.

If you are a graphic or web designer and want to keep every version of an image or layout (which you would most certainly want to), a Version Control System (VCS) is a very wise thing to use. It allows you to **revert** selected files back to a previous state, revert the entire project back to a previous state, **compare** changes over time, **see who** last **modified** something that might be causing a problem, who **introduced an issue** and when, and more. Using a VCS also generally means that if you screw things up or lose files, you can easily **recover**. In addition, you get all this for very little overhead.



# History pt.1: Local Version Control Systems

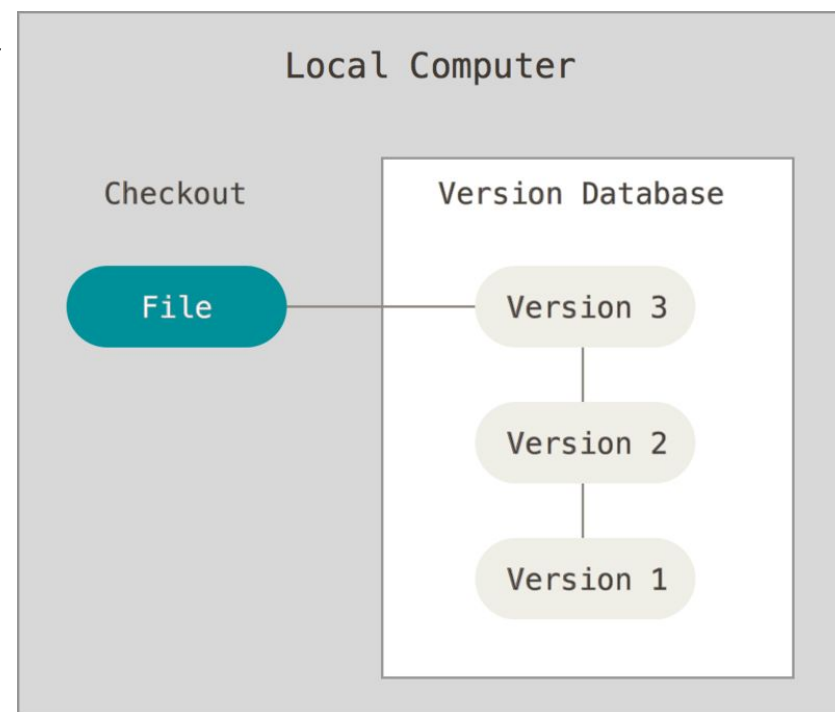


Naïf version control method:  
copy files into another directory

- simple...
- ...but incredibly error prone

⇒ programmers developed  
**Local VCSs:**

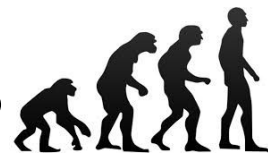
- a simple database that kept all the changes to files under revision control
- re-create what any file looked like at any point in time



E.g.: Revision Control System (RCS).



# History pt.2: Centralized Version Control Systems



The next major issue: how to collaborate with developers on other systems?

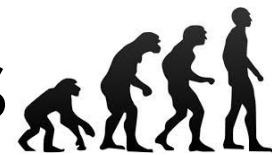
⇒ **centralized Version Control Systems (CVCSs):**

- a single server that contains all the versioned files
- clients that check out files from that central place

E.g.: Concurrent Versions System (CVS), Subversion, and Perforce



# History pt.2: Centralized Version Control Systems



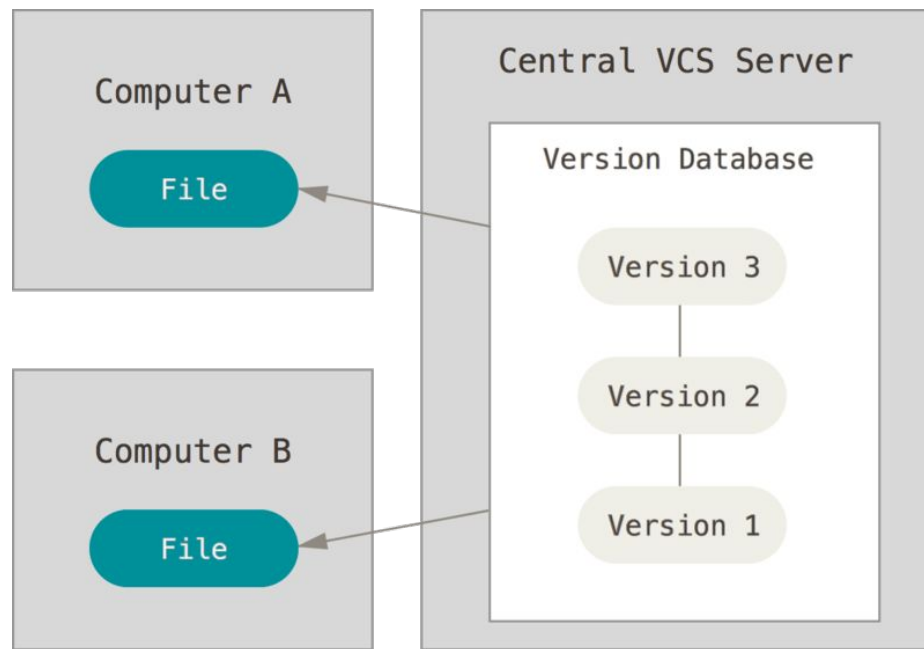
Advantages (compared to LVCSSs):

- collaboration
- users and permissions

Main problem:

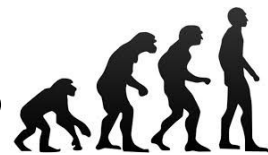
- single point of failure

Since the entire history of the project in a single place, you risk losing everything...





# History pt.3: Distributed Version Control Systems

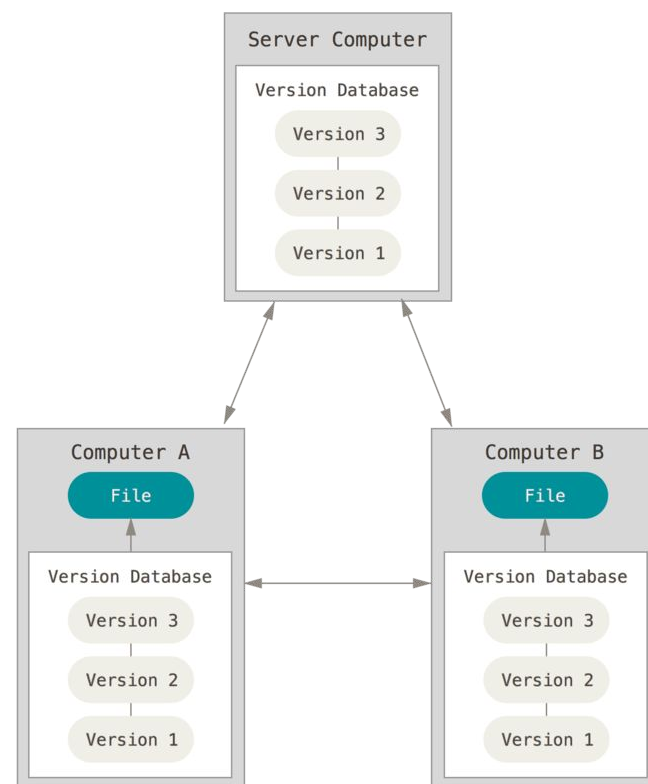


## Distributed Version Control Systems (DVCSs):

- each client fully mirrors the repository, including its full history

If any server dies, any of the client repositories can be copied back up to the server to restore it

E.g.: Git, Mercurial, Bazaar or Darcs







# A Short History of Git

Linux is an open source software project of fairly large scope.

1991-2002: changes to the software were passed around as archived files.

In 2002, the Linux project began using a proprietary DVCS called BitKeeper.

In 2005, the BitKeeper's free-of-charge status was revoked.

Linus Torvalds (the creator of Linux) started the development of a new VCS: Git.



# Git: Goals

Git's main goals:

- Speed
- Simple design
- Strong support for non-linear development (thousands of parallel branches)
- Fully distributed
- Able to handle large projects like the Linux kernel efficiently (speed and data size)



# Basic Concepts: Snapshots

The major difference between Git and other VCSs is the way Git thinks about its data.

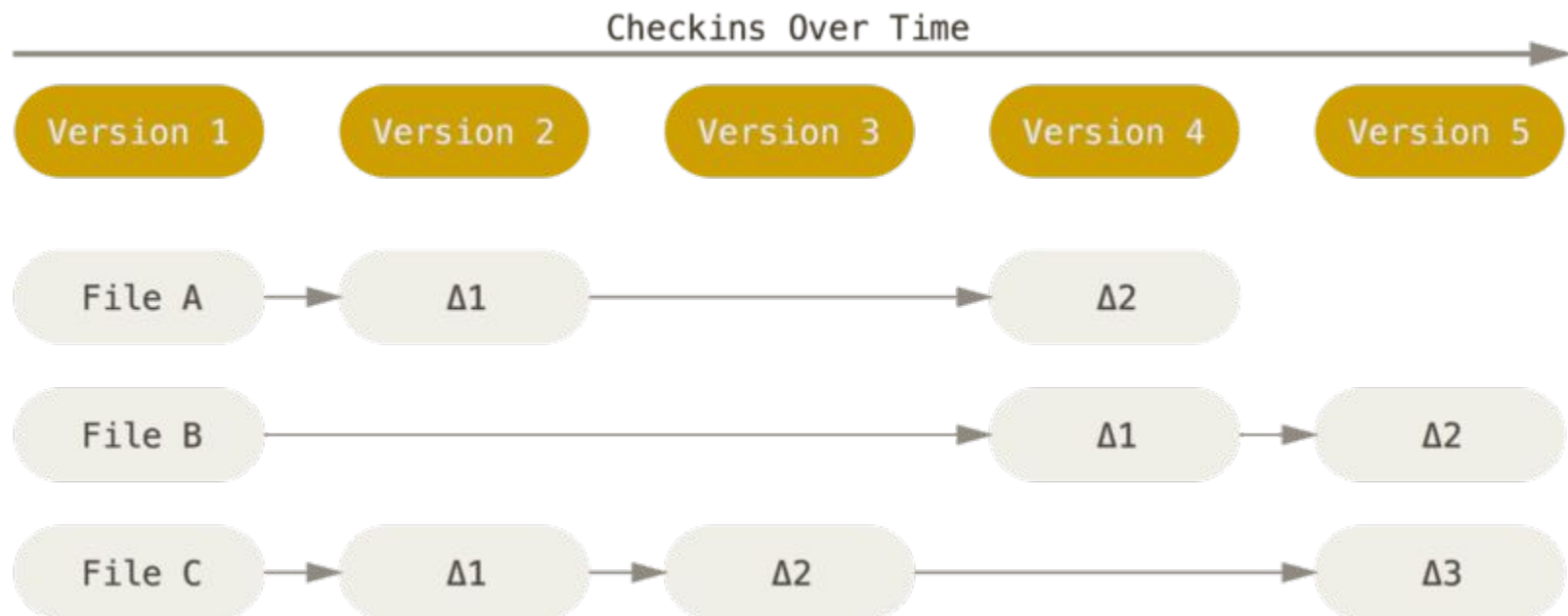
Conceptually, most other systems store information as a **list of file-based changes**:

- a set of files
- the changes made to each file over time (i.e. deltas)



# Basic Concepts: Snapshots

## Delta-based version control system





# Basic Concepts: Snapshots

Instead, Git thinks of its data like a series of **snapshots** of a miniature filesystem (i.e. your files and directories).

Every time you **save the state** of your project, Git **takes a picture** of what all your files look like at that moment and stores a reference to that snapshot.

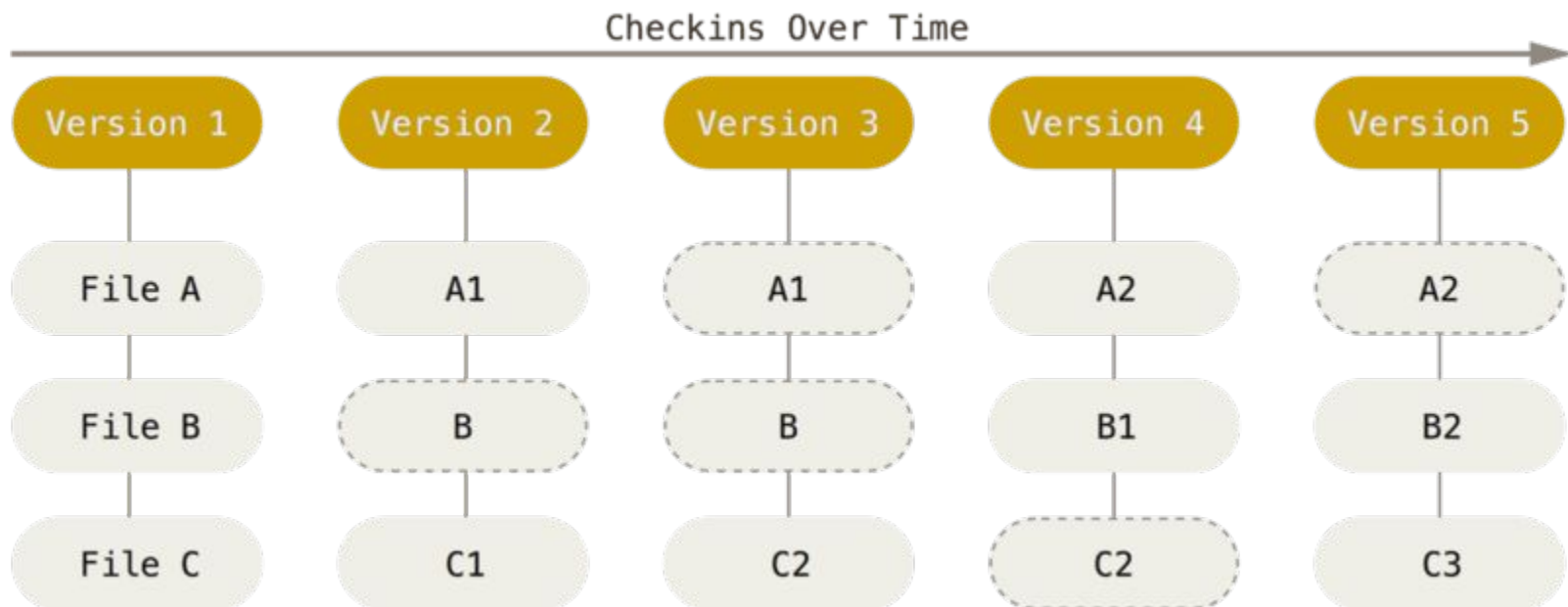
Snapshots are pictures of how your files looks like at a given point in time.

To be efficient, if files have not changed, Git stores a link to the previous identical file it has already stored.



# Basic Concepts: Snapshots

Git thinks about its data more like a stream of snapshots.





# Basic Concepts: Commit

**Commit:** the act of creating a snapshot

Can be a noun or a verb:

- “I committed code”
- “I just made a new commit”

Essentially, a project is made up of a bunch of commits.



# Basic Concepts: Commit

Commits contain three pieces of information:

- information about how the files changed from previously
- a reference to the commit that came before it
  - called the “parent commit”
- a hash code name (see next)





# Basic Concepts: Integrity

A **checksum** is a small-sized datum derived from a block of digital data for the purpose of detecting errors which may have been introduced during its transmission or storage.

Everything in Git is check-summed before it is stored, and is then referred to by that checksum.

The checksum mechanism (i.e. algorithm) used by Git is called a SHA-1 hash.

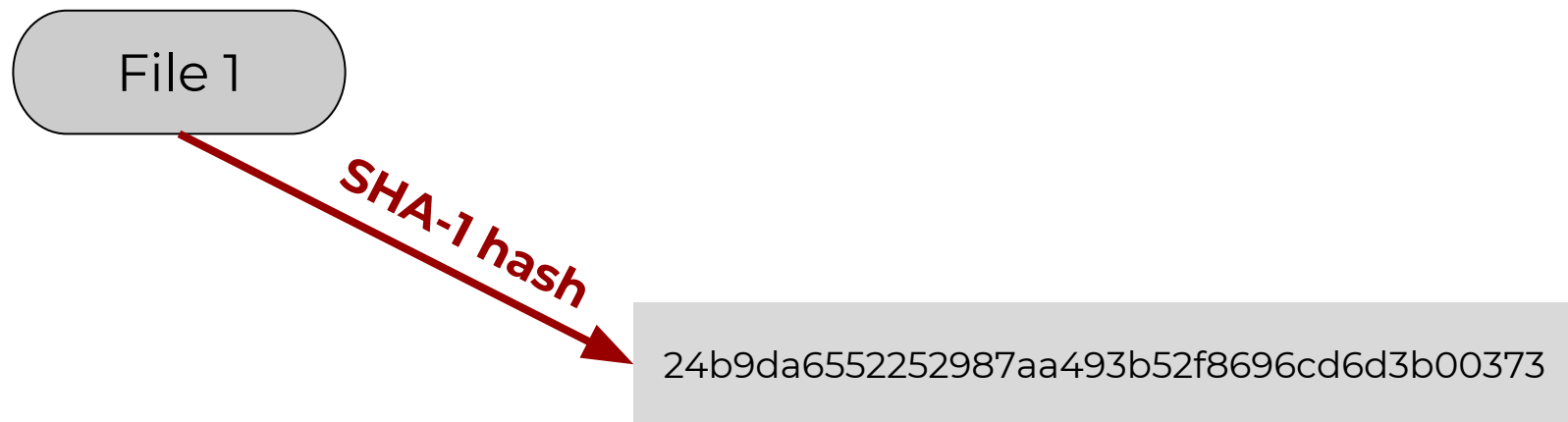
A 40-character string composed of hexadecimal characters (0–9 and a–f) and calculated based on the contents of a file or directory structure in Git.



# Basic Concepts: Integrity

Git stores everything in its database not by file name but by the hash value of its contents

- hash names are used as identifiers





# Basic Concepts: Repositories

**Repository** (often shortened to “**repo**”):

A collection of all the files and the history of those files

- consists of all your commits
- the place where all your hard work is stored



# Basic Concepts: Repositories

## Repositories:

- Can live on a local machine or on a remote server (e.g. GitHub)
- The act of copying a repository from a remote server is called **cloning**
- Cloning from a remote server allows team to work together
- The process of downloading commits that don't exist on your machine from a remote repository is called **pulling** changes
- The process of adding your local changes to the remote repository is called **pushing** changes



# Basic Concepts: Branches

All commits in Git live on some **branch**

But there can be many, many branches

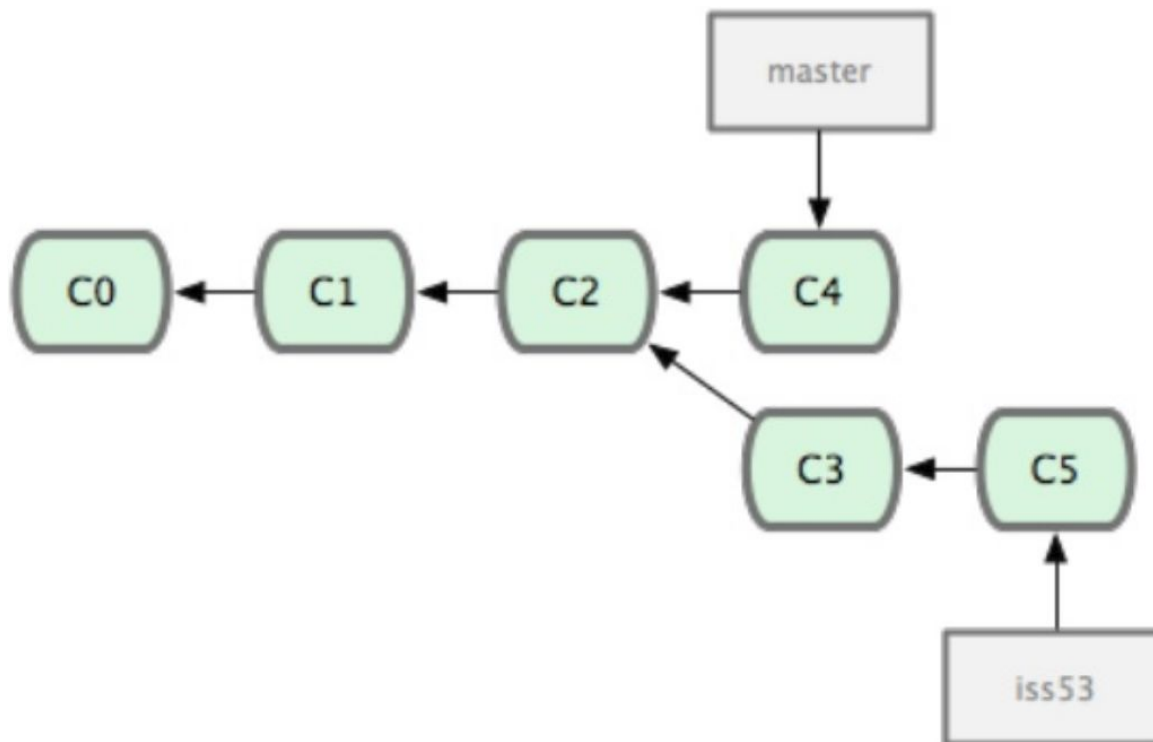
The main branch in a project is called the **master** branch (doesn't have to be called master, but almost always is!)

Branching means you diverge from the main line of development (master) and continue to do work without messing with that main line.



# Basic Concepts: Branches

Branching off of the master branch





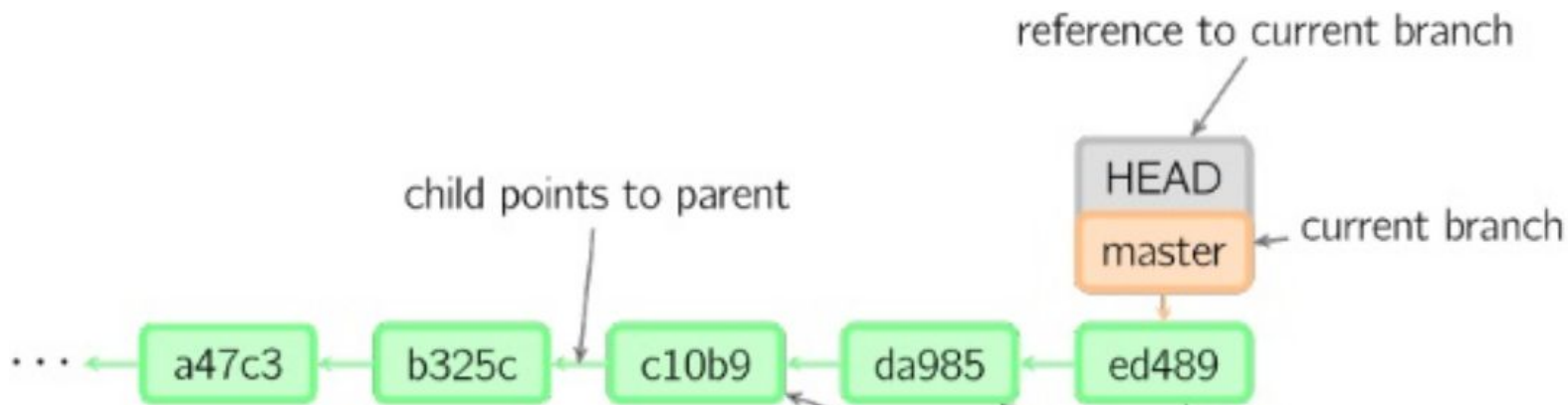
# Basic Concepts: HEAD

So, what does a typical project look like?

- A bunch of commits linked together that live on some branch, contained in a repository

What it is **HEAD**?

- A reference to the most recent commit in the current branch (you can think of the HEAD as the "current branch")





# Basic Concepts: the Three States

Git has three main states that your files can reside in:

- **Committed** means that the data is safely stored in your local database.
- **Modified** means that you have changed the file but have not committed it to your database yet.
- **Staged** means that you have marked a modified file in its current version to go into your next commit snapshot.

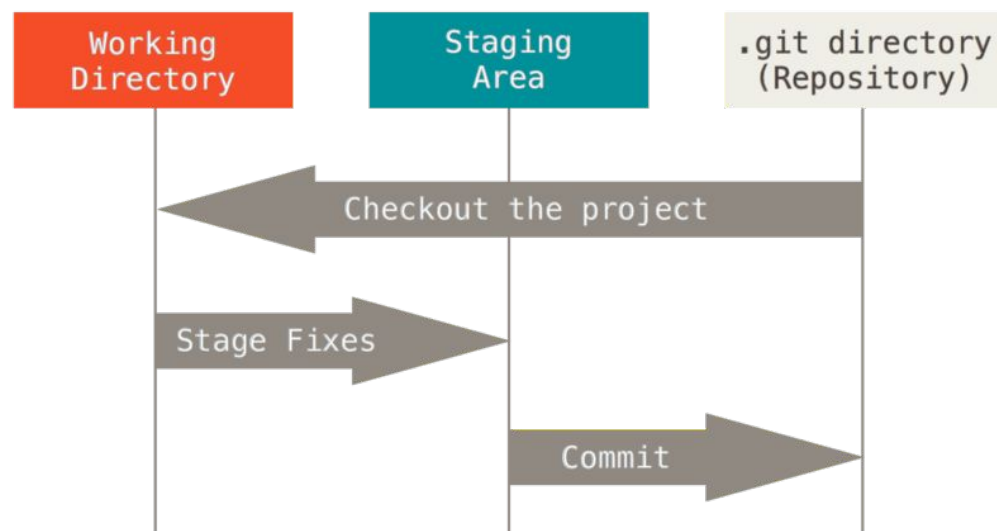




# Basic Concepts: Main Project Sections

The three main sections of a Git project are:

- **.git directory:** where Git stores the project metadata and object database.
- **Working directory:** contains files and directory ready to be used or modified.
- **Staging area:** a file storing information about what will go into your next commit. Its technical name in Git parlance is the “index”.

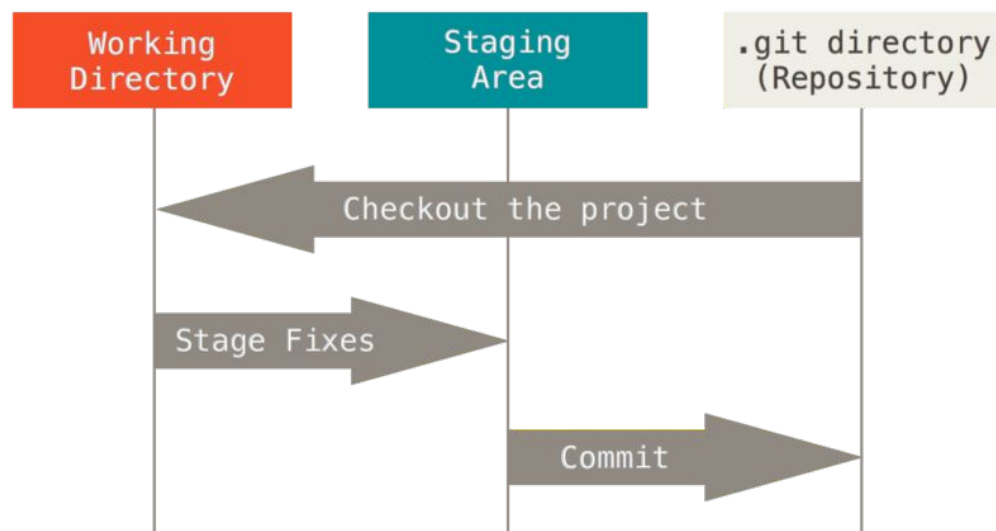




# Basic Concepts: Basic Git Workflow

The basic Git workflow goes something like this:

- you **modify** files in your *working tree*.
- you selectively **stage** just those changes you want to be part of your next commit, which adds only those changes to the *staging area*.
- you do a **commit**, which takes the files as they are in the staging area and stores that snapshot permanently to your *Git directory*.





# Git: the Command Line

There are two ways to use Git:

- the original command-line tools
- many graphical user interfaces exist, with varying capabilities

We'll use the command line because:

- it implements all Git commands (while most GUIs implement a partial subset of Git functionality)
- if you know the command-line version, you can also figure out how to run the GUI version (while the opposite is not true)
- all Git users will have the command-line tools installed and available (while graphical clients are optional)



# Git: Syntax and Use

`git [options] <command> [<args>]`

Example:

```
$ git --help
```

```
usage: git [--version] [--exec-path[=<path>]] [--html-path] [--man-path]
[--info-path] [-p|--paginate|--no-pager] [--no-replace-objects] [--bare]
[--git-dir=<path>] [--work-tree=<path>] [--namespace=<name>] [-c name=value]
[--help] <command> [<args>]
```

The most commonly used git commands are:

<code>add</code>	Add file contents to the index
<code>branch</code>	List, create, or delete branches
<code>checkout</code>	Checkout a branch or paths to the working tree
<code>clone</code>	Clone a repository into a new directory
<code>commit</code>	Record changes to the repository
<code>diff</code>	Show changes between commits, commit and working tree, etc
<code>fetch</code>	Download objects and refs from another repository
<code>...</code>	<code>...</code>

See 'git help <command>' for more information on a specific command.



# Getting a Git Repository

You typically obtain a Git repository in one of two ways:

1. you can take a local directory that is currently not under version control, and turn it into a Git repository, or
2. you can clone an existing Git repository from elsewhere.

In either case, you end up with a Git repository on your local machine, ready for work.



# 1. Initializing a Repository in an Existing Directory

Go to the project directory that you want to start controlling it with Git:

```
$ cd /home/user/my_project
```

and initialize a new local repository (i.e. set up all the tools Git needs to begin tracking changes made to the project):

```
$ git init
```

This command creates a new subdirectory named `.git` (i.e. the `.git` directory).

At this point, **nothing** in your project is **tracked yet**





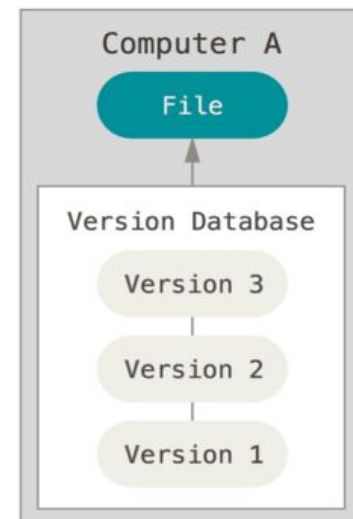
# 1. Initializing a Repository in an Existing Directory

**To start version-controlling** files, you should **begin tracking** those files and do an **initial commit**.

You can accomplish that with a few git add commands that specify the files you want to track, followed by a git commit:

```
$ git add file1 file2 file3  
$ git add directory1/*  
$ git commit -m 'initial project version'
```

Now you have a local project under version control.





# 1. Adding Remote Repositories

**To start collaborating**, you have to:

1. create a new **remote repository** (empty)







# 1. Adding Remote Repositories

**To start collaborating**, you have to:

1. add a new **remote repository** (empty)
2. “link” the local repository to the remote repository.

Remote repositories are versions of your project that are hosted on the Internet or network somewhere.

To add a new remote Git repository you must specify a shortname and a url:

```
$ git remote -u add <repo-shortname> <repo-url>
```

origin: is the default name Git gives to the server you cloned from.



# 1. Adding Remote Repositories

Example:

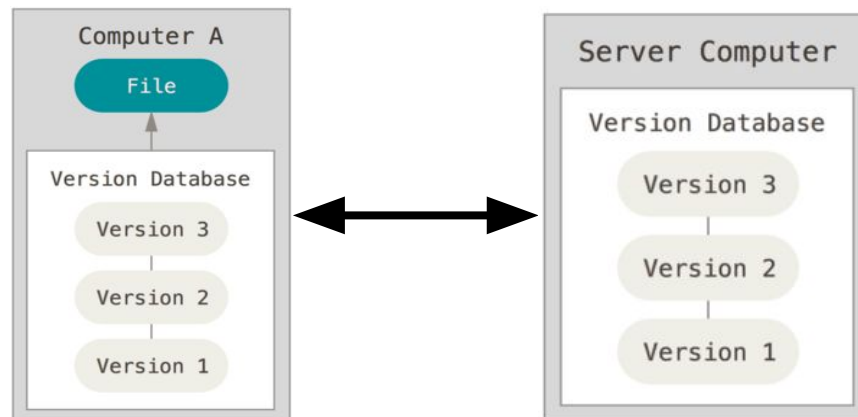
```
$ git push -u origin master
Username for 'https://github.com': fpoggi
Password for 'https://fpoggi@github.com':
remote:
remote: Create a pull request for 'master' on GitHub by visiting:
remote:      https://github.com/fpoggi/dhdk-empty/pull/new/master
remote:
To https://github.com/fpoggi/dhdk-empty.git
 * [new branch]      master -> master
Branch master set up to track remote branch master from origin.
```

**Note:** *origin* is the default name Git gives to the server you cloned from.



# 1. Adding Remote Repositories

Now we have the local repository and the remote repository initialized and “linked”



At any time, it is possible to check remotes URLs and shortnames:

```
$ git remote -v
```

```
origin https://github.com/fpoggi/dhdk-empty.git (fetch)
```

```
origin https://github.com/fpoggi/dhdk-empty.git (push)
```



## 2. Cloning an Existing Repository

**git clone** is the command to get a copy of an existing Git repository (e.g. a project you'd like to contribute to):

- a working copy is received (full copy of nearly all data that the server has).
- every version of every file for the project history is pulled down by default

E.g.: cloning the official Mozilla Firefox code base:

```
$ git clone https://github.com/mozilla/gecko-dev
```

**Note:** Git has a number of different transfer protocols you can use. The previous example uses the *https* protocol. Alternatives are *git* and *ssh* (e.g. `user@server:path/to/repo.git`)



# Handson

Exercise 1:

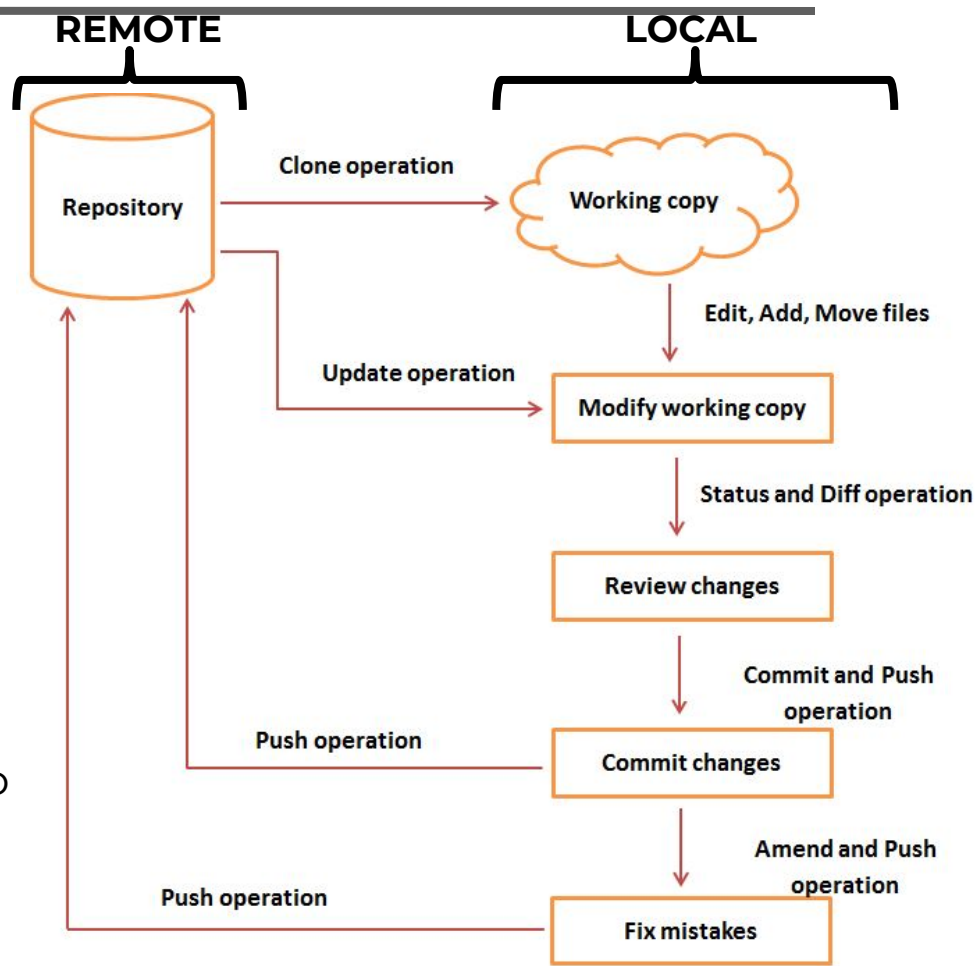
[https://docs.google.com/document/d/1Yu7vA33v2RHLodTd\\_gRYKMTja9WF2IDc8XfR8pqxS0KU/edit?usp=sharing](https://docs.google.com/document/d/1Yu7vA33v2RHLodTd_gRYKMTja9WF2IDc8XfR8pqxS0KU/edit?usp=sharing)



# Basic Concepts: General Git Workflow

General **workflow** is as follows:

1. **clone** the Git repository as a working copy
2. **modify** the working copy by adding/editing files
3. **update** the working copy by taking other developer's changes
4. **review** the changes before commit
5. **commit** changes. If everything is fine, then you push the changes to the repository
6. After committing, if you realize something is wrong, correct (**fix**) the last commit and push the changes to the repository





# 1. The add Command

Use the add command to:

1. begin tracking a new file (e.g. README.txt)
2. modify an existing file - already under revision control (e.g. the red line in file1.txt)

Before proceed, you can check the project status

```
$ git status
```

```
On branch master
```

```
Untracked files:
```

```
(use "git add <file>..." to include in what will be committed)
```

```
README.txt
```

```
Changes not staged for commit:
```

```
(use "git add <file>..." to update what will be committed)
```

```
(use "git checkout -- <file>..." to discard changes in working directory)
```

```
modified:   file1.txt
```

```
no changes added to commit (use "git add" and/or "git commit -a")
```

README.txt

```
Hello world!  
Have a nice day!
```

file1.txt

```
First line.  
This is a new line
```



# 1. The add Command

Now you can add both files to the staging area:

```
$ git add README.txt file1.txt
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

    README

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working
directory)

    modified:   file1.txt
no changes added to commit (use "git add" and/or "git commit -a")
```

and then check the project status:

```
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   file2.txt
    modified:   hello.txt
```

README.txt

```
Hello world!
Have a nice day!
```

file1.txt

```
First line.
This is a new line
```





# The `commit` Command

Now that your staging area is set up the way you want it, you can **commit** your changes (i.e. put them into the Git database):

```
$ git commit -m "Phase two - dhdK"  
[master 463dc4f] Phase two - dhdK  
2 files changed, 2 insertions(+)  
create mode 100644 README.txt
```

Git also provides a simple **shortcut** to skip the staging area. Adding the `-a` option to the `commit` command makes Git automatically stage every file *that is already tracked* before doing the commit, letting you skip the git add part:

```
$ git commit -a -m 'added new benchmarks'  
[master 83e38c7] added new benchmarks  
1 file changed, 5 insertions(+), 0 deletions(-)
```



# The `mv` Command

If you rename a file in Git, no metadata is stored in Git that tells it you renamed the file.

To **rename** (or **move**) a file in Git, you have to explicitly inform Git:

```
$ git mv README.txt README
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    renamed:    README.txt -> README
```

However, this is equivalent to running something like this:

```
$ mv README.txt README
$ git rm README.txt
$ git add README
```

Remember that also in this case changes have to be committed.



# The `rm` Command

To **remove** a file from Git, you have to remove it from your tracked files (i.e. from your staging area) and then commit.

The `rm` command does that, and also removes the file from your working directory.

```
$ git rm README.txt
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

       deleted:    file1.txt
```

Remember that also in this case changes have to be committed.



# The `pull` Command

The `pull` command is used to perform two operations in one single step:

- get data from your remote projects (fetch)
- then automatically merge the remote branch into your current branch.

```
$ git pull
```

which is equivalent to the following two commands

```
$ git fetch origin HEAD  
$ git merge HEAD
```



# The push Command

The `push <remote> <branch>` to push your master branch to your server:

```
$ git push origin master
```

This command works only if you cloned from a server to which you have write access and if **nobody has pushed in the meantime**, otherwise the push will be rejected.

In this case, you'll have to fetch their work first and incorporate (merge) it into yours before you'll be allowed to push.



# The diff Command

The `diff` command is usually used to answer two different questions:

- What have you changed but not yet staged?

```
$ git diff
diff --git a/myfile.csv b/myfile.csv
index 7898192..be5afde 100644
--- a/myfile.csv
+++ b/myfile.csv
@@ -1,3 @@
 a
+Added the second line
+And also the thid line
```

- What have you staged that you are about to commit?

```
$ git diff --staged
diff --git a/myfile.csv b/myfile.csv
new file mode 100644
index 0000000..7898192
--- /dev/null
+++ b/myfile.csv
@@ -0,0 +1 @@
+This is the first line
```



# The merge Command

Merging is Git's way of putting a forked history back together again.

The `git merge` command lets you take the independent lines of development created by `git branch` and integrate them into a single branch.

Two main cases:

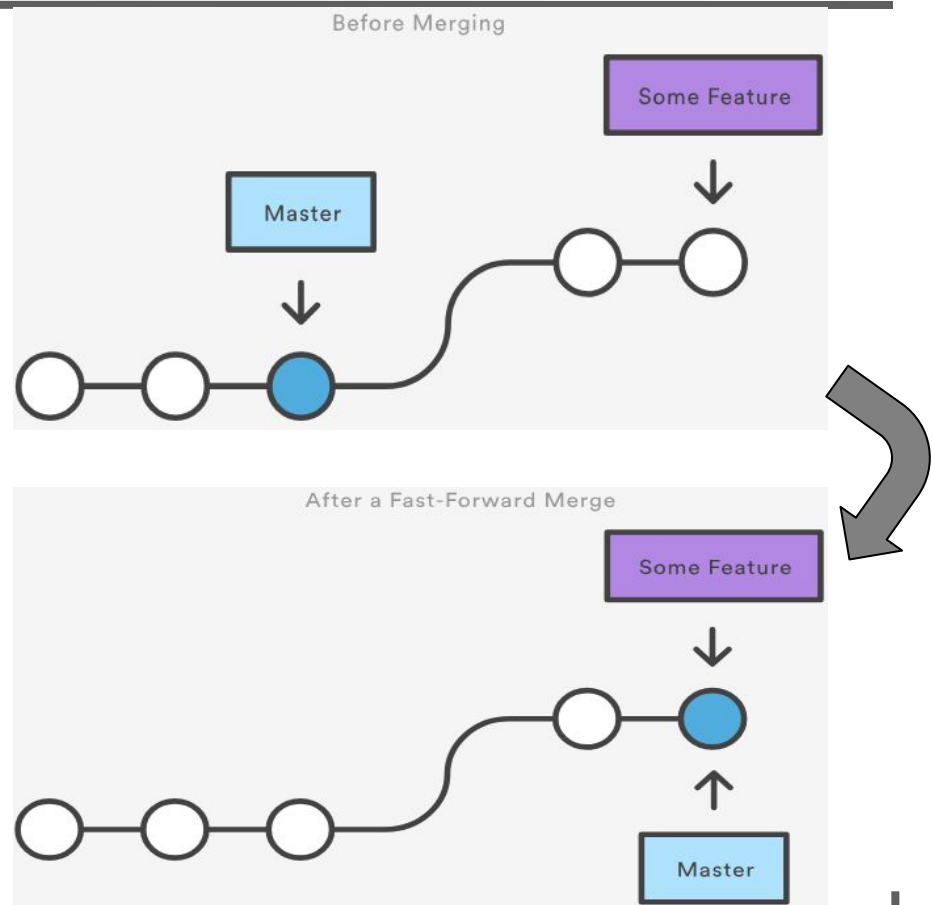
- Fast Forward Merge
- 3-way merge



# The merge Command

A **fast-forward merge** can occur when there is a linear path from the current branch tip to the target branch.

Solution: move (i.e., “fast forward”) the current branch tip up to the target branch tip.





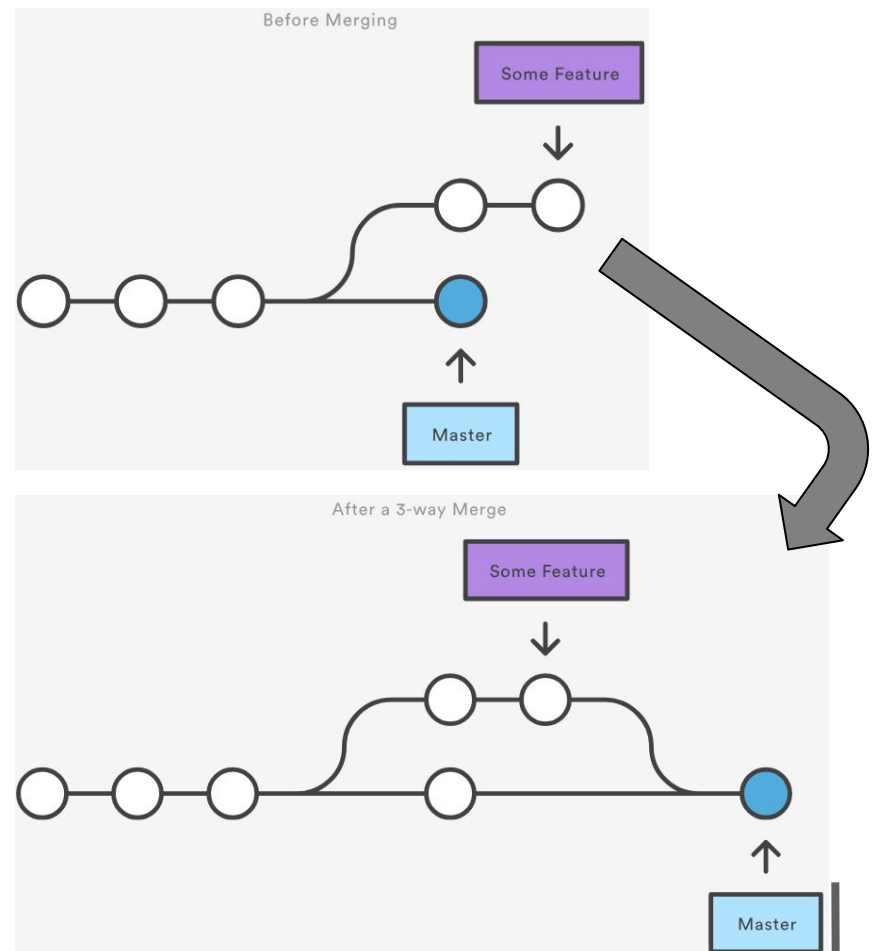


# The merge Command

A fast-forward merge is not possible if the branches *have diverged*.

When there is not a linear path to the target branch, Git has no choice but to combine them via a **3-way merge**.

3-way merges use a dedicated commit to tie together the two histories.





# Handson

## Exercise 2:

[https://docs.google.com/document/d/1Yu7vA33v2RHLodTd\\_gRYKMTja9WF2lDc8XfR8pqxS0KU/edit?usp=sharing](https://docs.google.com/document/d/1Yu7vA33v2RHLodTd_gRYKMTja9WF2lDc8XfR8pqxS0KU/edit?usp=sharing)



# GitHub

## What is GitHub?

- <https://github.com/>
- founded in 2008
- one of the largest web-based git repository hosting services (i.e. remote repositories)
- allows for code collaboration with anyone online
- add functionality on top of git:
  - UI, documentation, bug tracking, feature requests, pull requests, etc.
- also has an Enterprise Edition for business



# Handson

Exercise 3 and 4:

[https://docs.google.com/document/d/1Yu7vA33v2RHLodTd\\_gRYKMTja9WF2IDc8XfR8pqxS0KU/edit?usp=sharing](https://docs.google.com/document/d/1Yu7vA33v2RHLodTd_gRYKMTja9WF2IDc8XfR8pqxS0KU/edit?usp=sharing)



# Resources

---

<https://git-scm.com/book/en/v2/>

<https://classroom.udacity.com/courses/ud775/>

<https://www.codecademy.com/learn/learn-git>

<https://guides.github.com/>



# The Digital Object Identifier (DOI)

The Digital Object Identifier (DOI) system provides an infrastructure for:

- persistent unique identification of objects of any type
- interoperable exchange of managed information

on digital networks.



# The Digital Object Identifier (DOI)

The DOI system provides a ready-to-use packaged system of several components:

1. a specified standard numbering syntax (DOI names)
2. a resolution service
3. a data model incorporating a data dictionary
4. an implementation mechanism through a social infrastructure of organisations, policies and procedures for the governance and registration of DOI names



# 1. DOI Names

A DOI name is an identifier (not a location) of an entity on digital networks.

A DOI name is permanently assigned to an object to provide a resolvable persistent network link to current information about that object, including where the object, or information about it, can be found on the Internet.

While information about an object can change over time, its DOI name will not change.

A DOI name can be resolved within the DOI system to values of one or more types of data relating to the object identified by that DOI name, such as a URL, an e-mail address, other identifiers and descriptive metadata.





# 1. DOI Names

A DOI name is the string that specifies a unique object (the referent) within the DOI System.

The DOI name syntax (standardised as ANSI/NISO Z39.84-2005) is made up of a prefix element and a suffix element separated by a forward slash. For instance:

**Registrant code:** a unique alphanumeric string assigned to an organization that wishes to register DOI names. May be further divided into sub-elements by full stops.

**10.1000/123456**

**Directory indicator:** is always "10", identifies DOI names within the wider resolution system.

**Unique suffix:** may be a sequential number, or it may incorporate an identifier generated from or based on another system used by the registrant (e.g. ISBN, ISSN, ISTC).



# 1. DOI Names

The DOI name is *case-insensitive* and may incorporate any printable characters from the Unicode Standard.

The DOI name is an **opaque string** for the purposes of the DOI System: no definitive information should be inferred from the DOI name.

Note that the Registrant code (of specific organization) in a DOI name does not provide evidence of the ownership of rights or responsibility of any intellectual property.

Such information can be asserted in the associated DOI name metadata.



# 1. DOI Names

When displayed on screen or in print, a DOI name is normally preceded by a lowercase "doi:" unless the context clearly indicates that a DOI name is implied.

- the DOI name 10.1006/jmbi.1998.2354 is displayed as doi:10.1006/jmbi.1998.2354.

**To enable resolution** of the DOI name via a standard web hyperlink (e.g. in web browsers), DOI names may be attached to the address for an appropriate proxy server<sup>1</sup>.

- the DOI name 10.1006/jmbi.1998.2354 would be made an active link as <http://dx.doi.org/10.1006/jmbi.1998.2354>.

1. The International DOI Foundation (IDF) maintains a list of approved proxy servers (e.g. <http://dx.doi.org/> resolves DOI names in the context of web browsers using the Handle System resolution technology).



## 2. DOI Name Resolution

Resolution is the process of submitting a specific DOI name (input) to the DOI System and receiving in return (output) the associated values held in the DOI name resolution record for one or more types of data relating to the object identified by that DOI name.

This may include, but is not restricted to, types of data such as a location (URL), an e-mail address, another DOI name, descriptive metadata, etc.

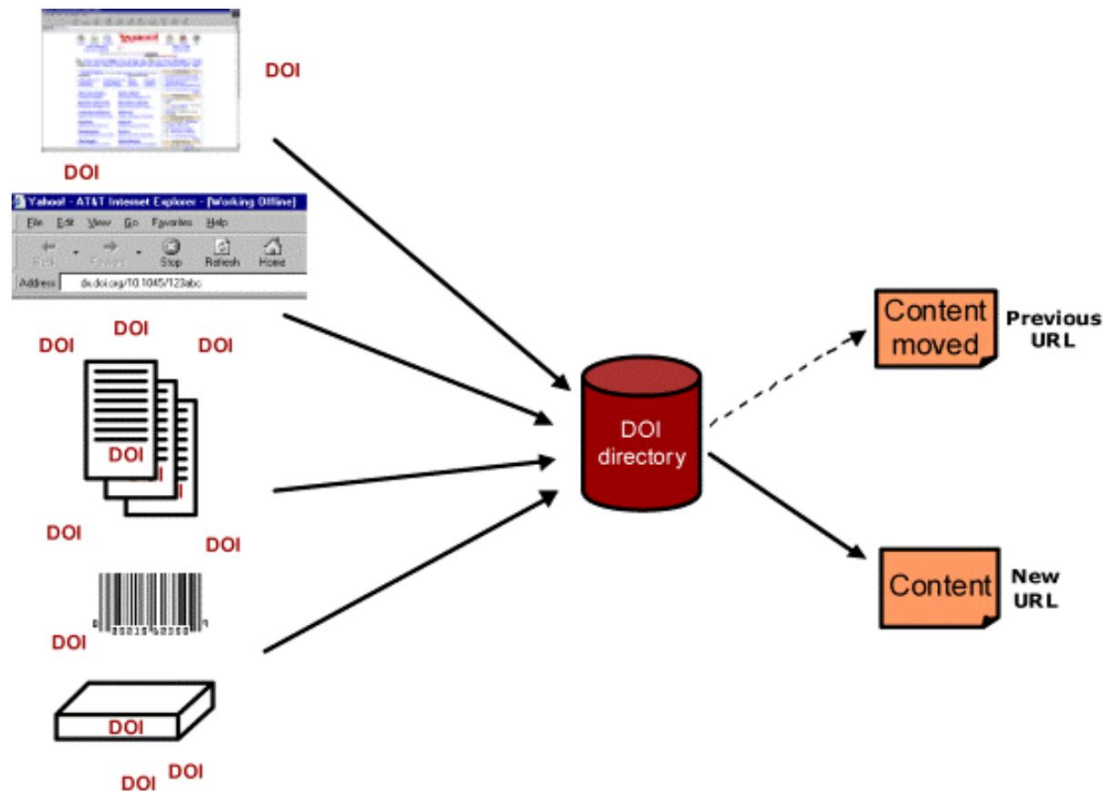
DOI name resolution records may include one or more URLs, where the object may be located, and other information provided about the entity to which a DOI name has been assigned. E.g. names, identifiers, descriptions, types, classifications, locations, times, measurements, relationships to other entities etc.

## 2. DOI Name Resolution

The initial implementation of the DOI System uses a single redirection from a DOI name to a digital location of the entity (or information about it).

### **Persistent** identifiers:

- e.g. content originally at one location and moved to a new one
- through a single change in the DOI System directory, all instances of the DOI name identifying that content will automatically resolve to the new location, without the user having to take any action.

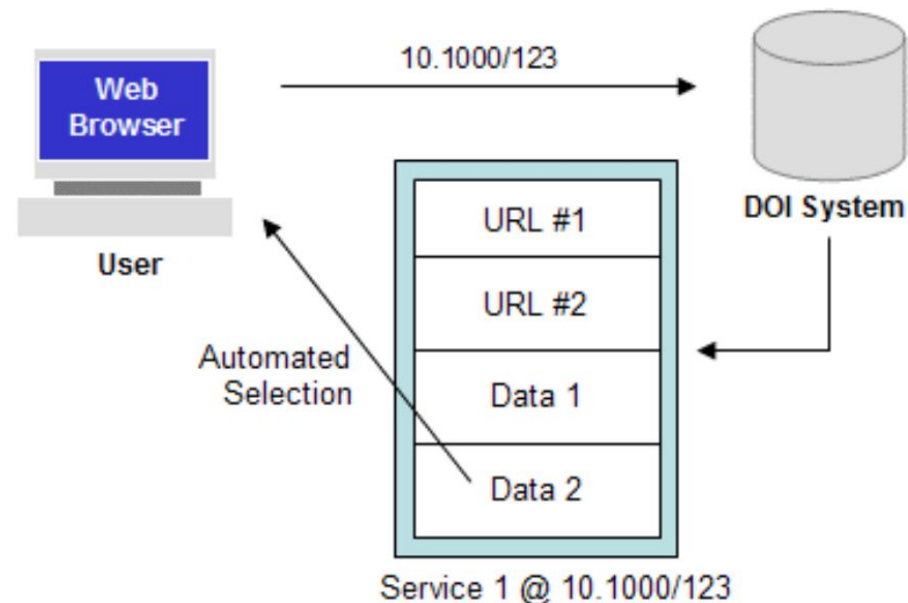




## 2. DOI Name Resolution

**Multiple resolution:** a significant functionality of the DOI System is the capability of delivering more than one typed “value” back from a resolution request.

E.g. the DOI name 10.1000/123 resolves to four values within the DOI System. An application is shown here which has the capability of selecting one of these results on the basis of some information provided in the resolution result and the local application.





## 3. Data Model

Having an identifier alone doesn't help – we want to know “what is this thing that's identified?”

- we want to know precisely
- precisely enough for automation

There's lots of metadata already: which should be (re-) used

People use different schemes: need to map from one scheme to another (e.g. does “owner” in scheme A mean “owner” in scheme B?)





## 3. Data Model

The underlying model of how data within the DOI System relates to other data.

Two components:

- Data Dictionary + DOI Application Profile Framework

### **Data Dictionary**

- Provides tool for precise description of entity through metadata (and mapping to other schemes)

### **DOI Application Profile Framework.**

- Provides means of relating entities: grouping entities and expressing relationships
- A mechanism for grouping DOI names with similar properties





## 4. DOI System Implementation

The **International DOI Foundation (IDF)** is the governance body of the DOI system, which safeguards (owns or licences on behalf of registrants) all intellectual property rights relating to the DOI system.

It works with Registration Agencies (RAs - see next slide) and with the underlying technical standards of the DOI system components to ensure that:

- *any improvements made to the system (including creation, maintenance, registration, resolution and policymaking of DOI names) are available to any DOI name registrant*
- *no third party licenses might reasonably be required to practice the DOI name standard*

DOI name resolution is freely available to any user encountering a DOI name.



## 4. DOI System Implementation

The DOI system is implemented through a federation of **Registration Agencies (RAs)** which use policies and tools developed through a parent body, the International DOI Foundation (IDF).

The primary role of Registration Agencies (RAs) is to **provide services to Registrants**:

- allocating DOI name prefixes, registering DOI names and providing the necessary infrastructure to allow Registrants to declare and maintain metadata and state data.

This service is expected to encompass **quality assurance** measures, so that the integrity of the DOI system as a whole is maintained at the highest possible level, delivering to users:

- reliable, accurate and up-to-date results
- metadata consistent and in compliance with both DOI system Kernel (the Data Model) and appropriate Application Profile standards.



# Crossref

Crossref (formerly CrossRef) is a non-profit official DOI Registration Agency of the International DOI Foundation.

It was launched in early 2000 as a cooperative effort among publishers to enable persistent cross-publisher citation linking in online academic journals.

Services:

- metadata search: <https://search.crossref.org/>
- <https://apps.crossref.org/SimpleTextQuery>
- REST-API: <https://github.com/CrossRef/rest-api-doc>
  - <https://api.crossref.org/works/10.1037/0003-066X.59.1.29>
  - <https://api.crossref.org/works?query=poggi+francesco+patterns>



# FigShare

Figshare is an online digital repository where researchers can preserve and share their research outputs, thus making them publicly available.

It is free to upload content and free to access, in adherence to the principle of open data.

Users can upload any file (e.g. figures, datasets, images, and videos) in any format, and items are attributed a DOI.

The current 'types' that can be chosen are figures, datasets, media (including video), papers (including pre-prints), posters, code, and filesets (groups of files).

All files are released under a Creative Commons license, CC-BY for most files and CC0 (public domain) for datasets.


Figshare allows researchers to publish negative data.

By encouraging publishing of figures, charts, and data, rather than being limited to the traditional entire 'paper', knowledge can be shared more quickly and effectively.

Figshare also tracks the download statistics for hosted materials, acting in turn as a source for altmetrics.



# FigShare

 **figshare** [My data](#)  [Browse](#) [Upload](#) Francesco Poggi ▾

My data

Projects

Collections

Activity

+ Create a new item

221.64 kB 20 GB

search my data...

	Actions	STATUS	TYPE	CREATED	SIZE
<input type="checkbox"/> Predicting the Results of Evaluation Procedures of Academics: Additional Materials			FILESET	13.7.2018 09:48	4.06 MB
<input type="checkbox"/> Predicting the Results of Evaluation Procedures of Academics: Appendices			FILESET	13.7.2018 09:05	1.3 MB
<input type="checkbox"/> Appendix C: Predictors Codebook (ASN 2012)			PAPER	2.7.2018 15:07	574.49 kB
<input type="checkbox"/> Appendix B: Basic Statics			PAPER	2.7.2018 13:53	702.21 kB
<input type="checkbox"/> Appendix A - List of the recruitment fields (ASN 2012)			PAPER	2.7.2018 12:44	52.23 kB
<input type="checkbox"/> Prediction of the Results of the ASN 2012 using the 15 top predictors - Performance of the SVM algorithm for academic level II (Associate Professor)			FIGURE	2.7.2018 11:39	998.72 kB
<input type="checkbox"/> Prediction of the Results of the ASN 2012 using the 15 top predictors - Performance of the SVM algorithm for academic level I (Full Professor)			FIGURE	2.7.2018 11:38	998.89 kB
<input type="checkbox"/> Prediction of the Results of the ASN 2012 - Performance of the SVM algorithm for academic level I (Full Professor) and II (Associate Professor) - Scientific Areas			FIGURE	2.7.2018 10:49	221.64 kB
<input type="checkbox"/> Prediction of the Results of the ASN 2012 - Performance of the SVM algorithm for academic level II (Associate Professor)			FIGURE	2.7.2018 10:04	982.57 kB



# FigShare

table-5.png 998.72 kB	table-4.png 998.89 kB
table-3.png 221.64 kB	table-2.png 982.57 kB
table-1.png 956.04 kB	

[Manage](#)

needed to publish & get DOI

**Title**

Predicting the Results of Evaluation Procedures of Academics: Additional Materials

**Authors**

Francesco Poggi

Search co-authors by name, full email or ORCID. Hit enter after each.

**Categories**

Informetrics, Applied Computer Science and 1 more

**Item type** (what's this?)

Fileset

**Description**

Additional materials containing the results of the analyses described in the paper entitled "Predicting the Results of Evaluation Procedures of Academics". In the tables, Precision (P), Recall (R) and F-Measure (FM) values are reported for each Recruitment Field (RF) and Area. The results are ordered in descending order with respect to the F-measure values. Non-bibliometric disciplines have a gray background. The data are organized as follows:  
-Table 1: contains the performance of the SVM algorithm for

**Keyword(s)**

ASN 2012 Italian National Scientific Habilitation 2012

Add keywords for easy discovery. Hit enter after each

**References**

Link to references or related content

**Funding**

Add grant number or funding authority

**Licence** (what's this?)

CC BY 4.0

**This item is public** (changes were made but not published)

[Generate private link](#)

<https://figshare.com/s/c9e4f81f82e061d464a2>

This link can be used by non figshare users also.

**DOI** Digital Object Identifier

10.6084/m9.figshare.6814550

[Cancel](#) [DOI](#) [Publish changes](#) [Save changes](#)





# FigShare

My data  Browse Upload Francesco Poggi

table-5.png (998.72 kB)

table-4.png (998.89 kB)

table-3.png (221.64 kB)

table-2.png (982.57 kB)

table-1.png (956.04 kB)

Cite Download all (4.06 MB) Share Embed + Collect ... 5 files

### Predicting the Results of Evaluation Procedures of Academics: Additional Materials

Fileset posted on 13.07.2018, 10:54 by Francesco Poggi

Additional materials containing the results of the analyses described in the paper entitled "Predicting the Results of Evaluation Procedures of Academics".

In the tables, Precision (P), Recall (R) and F-Measure (FM) values are reported for each Recruitment Field (RF) and Area. The results are ordered in descending order with respect to the F-measure values. Non bibliometric disciplines have a gray background.

The data are organized as follows:

- Table 1:** contains the performance of the SVM algorithm for academic level I (Full Professor) using 291 predictors. Analysis of the 284 recruitment fields;
- Table 2:** contains the performance of the SVM algorithm for academic level II (Associate Professor) using 291 predictors. Analysis of the 284 recruitment fields;
- Table 3:** contains the performance of the SVM algorithm for academic level I (Full Professor) and II (Associate Professor). Analysis of the scientific areas;
- Table 4:** contains the performance of the SVM algorithm for academic level I (Full Professor) using the top 15 predictors. Analysis of the 284 recruitment fields;
- Table 5:** contains the performance of the SVM algorithm for academic level II (Associate Professor) using the top 15 predictors. Analysis of the 284 recruitment fields.

18 Views 0 downloads 0 citations

CATEGORIES

- Informetrics
- Applied Computer Science
- Knowledge Representation and Machine Learning

KEYWORD(S)

ASN 2012

Italian National Scientific Habilitation 2012

LICENCE

CC BY 4.0

EXPORT

RefWorks

BibTeX

Ref. manager

Endnote

DataCite

NLM

DC



# Thanks for your attention!

## Questions?