

Software Engineering at MPRI

Tutorial on git, and its extensions

Amélie Ledein
ledein@lsv.fr

September 29, 2022

Introduction

Maybe, you already know...

Maybe, you already know...

- **Rigor**

- Code, methodology, specifications and tests

Introduction

Maybe, you already know...

- **Rigor**

- Code, methodology, specifications and tests

- **Adaptability**

- Components could be re-used in a (slightly different) context.

Maybe, you already know...

- **Rigor**

- Code, methodology, specifications and tests

- **Adaptability**

- Components could be re-used in a (slightly different) context.

- **Modularity**

- Segment project in **modules** with clearly defined **interfaces**.

- Develop and test independently, facilitate re-use and change.

Maybe, you already know...

- **Rigor**

- Code, methodology, specifications and tests

- **Adaptability**

- Components could be re-used in a (slightly different) context.

- **Modularity**

- Segment project in **modules** with clearly defined **interfaces**.

- Develop and test independently, facilitate re-use and change.

- **Abstraction**

- Do not specify implementation **details**, i.e. things that can easily change.

Introduction

Maybe, you already know...

- **Rigor**

- Code, methodology, specifications and tests

- **Adaptability**

- Components could be re-used in a (slightly different) context.

- **Modularity**

- Segment project in **modules** with clearly defined **interfaces**.

- Develop and test independently, facilitate re-use and change.

- **Abstraction**

- Do not specify implementation **details**, i.e. things that can easily change.

OK, but how can you share your code?

Maybe, you already know...

- **Rigor**

- Code, methodology, specifications and tests

- **Adaptability**

- Components could be re-used in a (slightly different) context.

- **Modularity**

- Segment project in **modules** with clearly defined **interfaces**.

- Develop and test independently, facilitate re-use and change.

- **Abstraction**

- Do not specify implementation **details**, i.e. things that can easily change.

OK, but how can you share your code?

- USB-key

Introduction

Maybe, you already know...

- **Rigor**

- Code, methodology, specifications and tests

- **Adaptability**

- Components could be re-used in a (slightly different) context.

- **Modularity**

- Segment project in **modules** with clearly defined **interfaces**.

- Develop and test independently, facilitate re-use and change.

- **Abstraction**

- Do not specify implementation **details**, i.e. things that can easily change.

OK, but how can you share your code?

- USB-key

Introduction

Maybe, you already know...

- **Rigor**

- Code, methodology, specifications and tests

- **Adaptability**

- Components could be re-used in a (slightly different) context.

- **Modularity**

- Segment project in **modules** with clearly defined **interfaces**.

- Develop and test independently, facilitate re-use and change.

- **Abstraction**

- Do not specify implementation **details**, i.e. things that can easily change.

OK, but how can you share your code?

- USB-key

- Email

Introduction

Maybe, you already know...

- **Rigor**

- Code, methodology, specifications and tests

- **Adaptability**

- Components could be re-used in a (slightly different) context.

- **Modularity**

- Segment project in **modules** with clearly defined **interfaces**.

- Develop and test independently, facilitate re-use and change.

- **Abstraction**

- Do not specify implementation **details**, i.e. things that can easily change.

OK, but how can you share your code?

- USB-key

- Email

Maybe, you already know...

- **Rigor**

- Code, methodology, specifications and tests

- **Adaptability**

- Components could be re-used in a (slightly different) context.

- **Modularity**

- Segment project in **modules** with clearly defined **interfaces**.

- Develop and test independently, facilitate re-use and change.

- **Abstraction**

- Do not specify implementation **details**, i.e. things that can easily change.

OK, but how can you share your code?

- ~~USB-key~~

- ~~Email~~

- Dropbox, Google Drive, etc.

Introduction

Maybe, you already know...

- **Rigor**

- Code, methodology, specifications and tests

- **Adaptability**

- Components could be re-used in a (slightly different) context.

- **Modularity**

- Segment project in **modules** with clearly defined **interfaces**.

- Develop and test independently, facilitate re-use and change.

- **Abstraction**

- Do not specify implementation **details**, i.e. things that can easily change.

OK, but how can you share your code?

- ~~USB-key~~

- ~~Email~~

- ~~Dropbox, Google Drive, etc.~~

The answer is...

The answer is...



git

*or any version control systems (VCS).
(systèmes de gestion de version, in French)*

- 1 The basics of git
 - Git in a nutshell
 - Basic commands
 - GitHub/GitLab
- 2 Branching, merging and problems
 - Data structures
 - Branching
 - Move in the commit tree
 - Undo changes with Git
 - Merging of two branches
- 3 Additional commands
 - Additional tips
 - Storage of your files
 - Bring in changes from a specific commit
 - Find a bad commit in your app

- 1 The basics of git
 - Git in a nutshell
 - Basic commands
 - GitHub/GitLab
- 2 Branching, merging and problems
 - Data structures
 - Branching
 - Move in the commit tree
 - Undo changes with Git
 - Merging of two branches
- 3 Additional commands
 - Additional tips
 - Storage of your files
 - Bring in changes from a specific commit
 - Find a bad commit in your app

What is a version control system?

What is a version control system?

- Software system that allows you to maintain and manage all versions of a set of files.

What is a version control system?

- Software system that allows you to maintain and manage all versions of a set of files.
- Why a version management system?
 - Revolve easily to a previous version.
 - Follow the evolution of the project over time.
 - Allow parallel work on disjointed parts of the project and manage the competing modifications.
 - Facilitate the detection and correction of errors.
 - etc.

What is a version control system?

- Software system that allows you to maintain and manage all versions of a set of files.
- Why a version management system?
 - Revolve easily to a previous version.
 - Follow the evolution of the project over time.
 - Allow parallel work on disjointed parts of the project and manage the competing modifications.
 - Facilitate the detection and correction of errors.
 - etc.
- An example:



- A free and open source distributed version control system (DVCS)
- Designed to handle everything from small to very large projects with speed and efficiency

Git in detail

On your computer

Stash
(Remise)

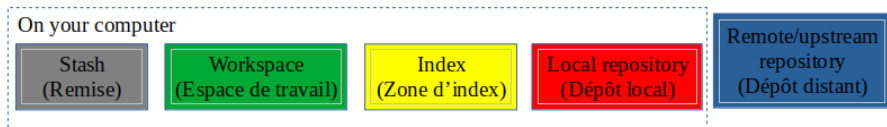
Workspace
(Espace de travail)

Index
(Zone d'index)

Local repository
(Dépôt local)

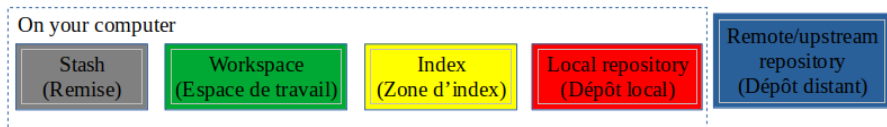
Remote/upstream
repository
(Dépôt distant)

Git in detail



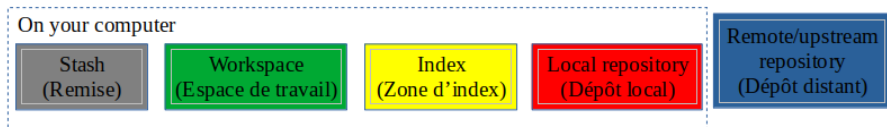
- **Stash (remise):** A place to hide modifications while you work on something else.

Git in detail



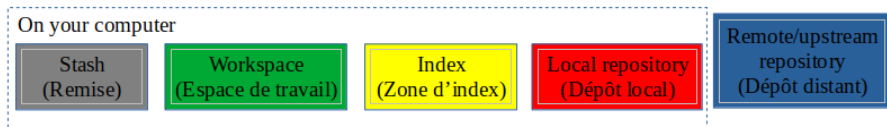
- **Stash (remise)**: A place to hide modifications while you work on something else.
- **Workspace (espace de travail)**: Local checkout.

Git in detail



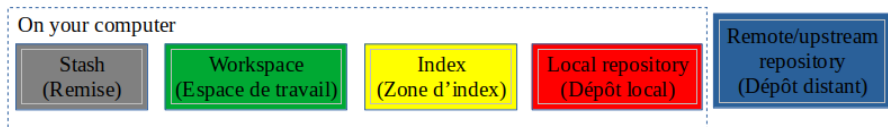
- **Stash (remise):** A place to hide modifications while you work on something else.
- **Workspace (espace de travail):** Local checkout.
- **Index (zone d'index), staging area, staged files** or (current directory) cache: Files you want to commit. Before you "commit" (checkin) files, you need to first add them to the index.

Git in detail



- **Stash (remise):** A place to hide modifications while you work on something else.
- **Workspace (espace de travail):** Local checkout.
- **Index (zone d'index), staging area, staged files** or (current directory) cache: Files you want to commit. Before you "commit" (checkin) files, you need to first add them to the index.
- **Local repository (dépôt local):** A subdirectory named `.git` that contains all of your necessary repository files - a Git repository skeleton.

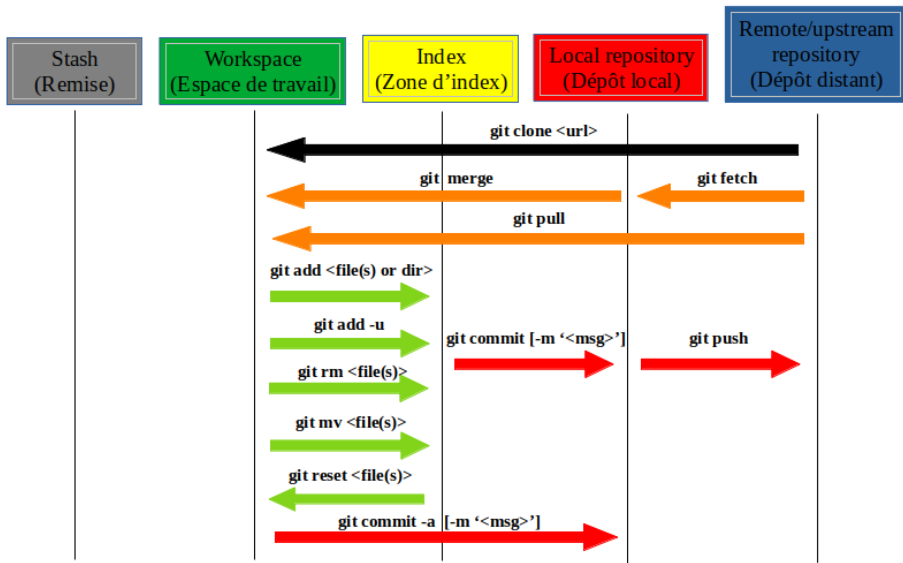
Git in detail



- **Stash (remise):** A place to hide modifications while you work on something else.
- **Workspace (espace de travail):** Local checkout.
- **Index (zone d'index), staging area, staged files** or (current directory) cache: Files you want to commit. Before you "commit" (checkin) files, you need to first add them to the index.
- **Local repository (dépôt local):** A subdirectory named `.git` that contains all of your necessary repository files - a Git repository skeleton.
- **Remote/upstream repository (dépôt distant):** Versions of your project that are hosted on the Internet or network, ensuring all your changes are available for other developers.

The default name is `origin`.

- 1 The basics of git
 - Git in a nutshell
 - **Basic commands**
 - GitHub/GitLab
- 2 Branching, merging and problems
 - Data structures
 - Branching
 - Move in the commit tree
 - Undo changes with Git
 - Merging of two branches
- 3 Additional commands
 - Additional tips
 - Storage of your files
 - Bring in changes from a specific commit
 - Find a bad commit in your app



Synchronization with the remote repository

\$ `git clone <url>`

Retrieve an entire repository from hosted location via URL.

\$ `git fetch <alias>`

Fetch down all the branches from that Git remote.

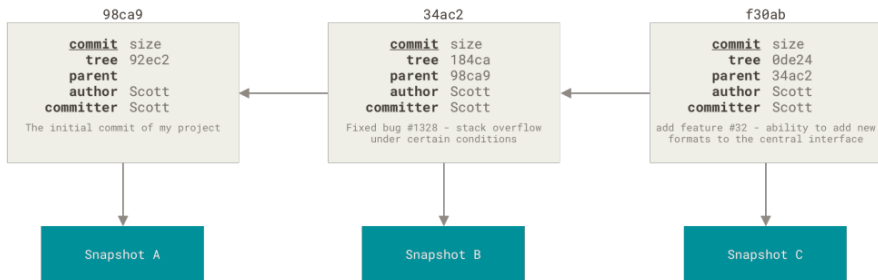
\$ `git merge <alias/<branch>`

Merge a remote branch into your current branch to bring it up to date.

\$ `git pull`

Fetch and merge any commits from the tracking remote branch.

Commit



Make modifications

```
$ git add <file(s)>
```

Add a file (or several) as it looks now to your next commit (stage).

```
$ git rm <file(s)>
```

Delete the file (or several) from the project and stage the removal for commit.

```
$ git mv <old-name-file> <new-name-file>
```

Rename the file and stage the renaming.

```
$ git mv <existing-path> <new-path>
```

Change an existing file path and stage the move.

```
$ git reset <file(s)>
```

Unstage a file (or several) while retaining the changes in working directory.

```
$ git commit [-m "<descriptive message>"]
```

Commit your staged content as a new commit snapshot.

```
$ git push <alias> <branch>
```

Transmit local branch commits to the remote repository branch.

Informative commands

- Setup:

Configuring user information used across all local repositories.

```
$ git config --global user.name "[firstname lastname]"
```

Set a name that is identifiable for credit when review version history.

```
$ git config --global user.email "[valid-email]"
```

Set a email address that will be associated with each history marker.

Note : Add `export EDITOR=emacs` (or `vim`, etc.) to the `.bashrc` file.

To configure correctly your editor with Git.

- To collect information:

```
$ git status
```

Show modified files in working directory, staged for your next commit.

```
$ git diff
```

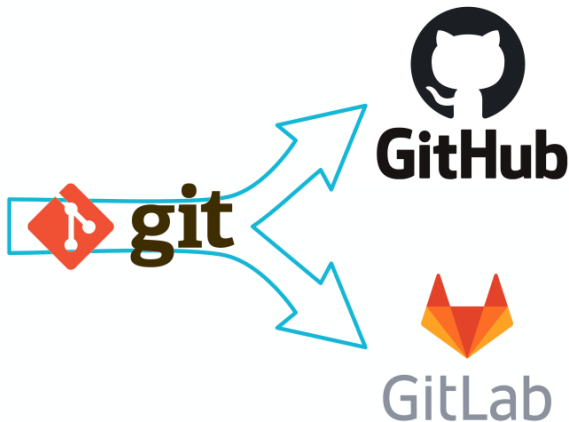
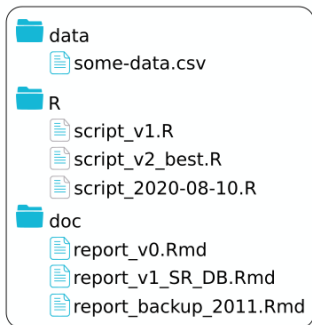
Diff of what is changed but not staged.

```
$ git diff --staged
```

Diff of what is staged but not yet committed.

- 1 The basics of git
 - Git in a nutshell
 - Basic commands
 - **GitHub/GitLab**
- 2 Branching, merging and problems
 - Data structures
 - Branching
 - Move in the commit tree
 - Undo changes with Git
 - Merging of two branches
- 3 Additional commands
 - Additional tips
 - Storage of your files
 - Bring in changes from a specific commit
 - Find a bad commit in your app

Git and GitHub/GitLab



Graphical interface

The screenshot shows the GitHub repository page for Deducteam/lambdapi. The repository is in the 'Code' tab, showing the 'master' branch with 3 branches and 2 tags. The repository has 14 pull requests, 54 issues, and 3,246 commits. The repository description is 'Proof assistant based on the $\lambda\Pi$ -calculus modulo rewriting'. The repository contains a file tree with folders 'docs', 'editors', 'libraries', and 'src'. The 'libraries' folder has a description 'File management and module mapping Improvement (#289)'. The repository also has a README and a license.

Search or jump to... Pull requests Issues Marketplace Explore

Deducteam / lambdapi Unwatch 14 Star 113 Fork 24

<> Code Issues 54 Pull requests 14 Actions Projects Wiki Security Insights

master 3 branches 2 tags Go to file Add file Code

authors [WIP] Induction (#407) 53cc077 9 days ago 3,246 commits

docs	[WIP] Induction (#407)	9 days ago
editors	[WIP] Induction (#407)	9 days ago
libraries	File management and module mapping Improvement (#289)	7 months ago
src	[WIP] Induction (#407)	9 days ago

About

Proof assistant based on the $\lambda\Pi$ -calculus modulo rewriting

proof-checker proof-assistant rewriting dependent-types logical-framework

Readme View license

- See current code
- See each commit
- See each issue
- Do integration continuous
- etc.

The screenshot shows the Gitter chat interface for the repository `Deducteam/lambdapi`. The interface is divided into three main sections: a left sidebar for navigation, a central chat area, and a right sidebar for user avatars.

Left Sidebar: Contains navigation icons for home, search, and user management. Below these is a list of conversations under the heading "ALL CONVERSATIONS". Two conversations are listed: `lambdapi` (selected) and `lambdapi-emacs-mode`.

Central Chat Area: Displays the chat history for the selected conversation. The header shows the repository name `Deducteam/lambdapi`. The chat content includes:

- A message: "and it can get complicated to debug (Why3 users had issues with this when testing my server)" followed by a quote: "Does it really? It seemed to me that some language server can be used with any lsp client (I'm not clear on the architecture of LSP, so I might confuse things up)".
- A response: "If we develop extensions, yep :)"
- A message from `Gabriel Hondet @gabrielhdt` (mailed 19 14:23): "Ah right right"
- A message from `Emilio Jesús Gallego Arias @ejgallego` (mailed 19 14:23): "I think that for now the only extension is this extra goals request so it is not like super-complex"
- A message from `DIVERIO Diego @vycastor` (mailed 19 14:25): "When developing an emacs mode, you also have to provide some configuration to the LSP client you're using underneath, but it's pretty basic"
- A message from `Emilio Jesús Gallego Arias @ejgallego` (mailed 19 14:25): "yup, that was one of the advantages of eglot, it kinda worked 'config-free'"
- A message from `DIVERIO Diego @vycastor` (mailed 19 14:25): "So to support two different clients, we'll have to provide two emacs configuration, even if short"

At the bottom of the chat area is a text input field with the placeholder "Click here to type a chat message. Supports GitHub flavoured markdown." and a "GO TO BOTTOM" button.

Right Sidebar: Displays a vertical list of user avatars participating in the chat.

Zulip - Finite_Set_Coq

File Edit View History Window Tools Help

ZULIP

Gap intervals list | 2 | (no description)

Gap intervals list Notifications gitlab

Aug 30 16:30

GIL_bot

[Gap_intervals_list] Amélie pushed 3 commits to branch master. Commits by Amélie\ Ledein (2) and camel (1).

- Update? (4880beb)
- Test de preuve equiv : echec (2fb59ed)
- Ancien fichier avec une pseudo-proof (is_empty_spec) faite par quivalence sur les arbres (38b4cc4)

[Gap_intervals_list] Amélie pushed 1 commit to branch master. Commits by camel (1).

- Add fonctor interface (0f9ad47)

SEP 19

GIL_bot

[Gap_intervals_list] Amélie pushed 1 commit to branch master. Commits by camel (1).

- The term j has type U.Z_finite_subset while it is expected to have type Z. (4e16116)

SEP 19

GIL_bot

[Gap_intervals_list] Amélie pushed 1 commit to branch master. Commits by camel (1).

- Aide Chantal (-501bda)

[Gap_intervals_list] Amélie pushed 1 commit to branch master. Commits by camel (1).

- Problem elt_list and structure.elt_list (4cd3785)

[Gap_intervals_list] Amélie pushed 1 commit to branch master. Commits by camel (1).

- New interface foncteur (ca20236)

16:31

21:48

10:20

11:32

12:16

USERS

- Amélie (you)
- Catherine

Invite more users

Drafts (0)

New topic New private message Reply



Travis CI

Let's go!

1. Install git
2. Create an account on GitLab or GitHub
3. Create a repository for your group
4. Give me the link

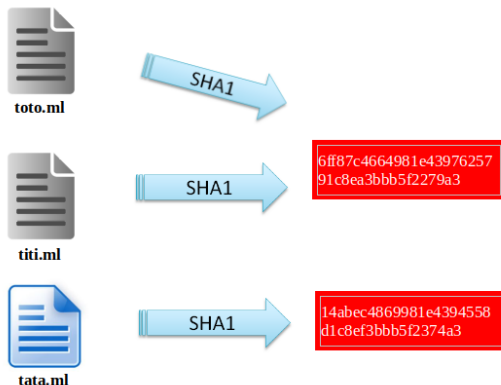
Do the exercise 1 in parallel:

1. Clone the repository from
`https://github.com/amelieled/SE\_GIT\_MPRI\_2022.git`
2. Add at least 5 new items in the grocery list.
3. Fix the 5 errors.
4. Add a new section.

- 1 The basics of git
 - Git in a nutshell
 - Basic commands
 - GitHub/GitLab
- 2 Branching, merging and problems
 - Data structures
 - Branching
 - Move in the commit tree
 - Undo changes with Git
 - Merging of two branches
- 3 Additional commands
 - Additional tips
 - Storage of your files
 - Bring in changes from a specific commit
 - Find a bad commit in your app

- 1 The basics of git
 - Git in a nutshell
 - Basic commands
 - GitHub/GitLab
- 2 Branching, merging and problems
 - **Data structures**
 - Branching
 - Move in the commit tree
 - Undo changes with Git
 - Merging of two branches
- 3 Additional commands
 - Additional tips
 - Storage of your files
 - Bring in changes from a specific commit
 - Find a bad commit in your app

SHA1 is a hashing algorithm taking an input up to 2^{64} bits, and returns a unique sequence of 40 hexadecimal characters.



By hashing the contents of a file, Git obtains a series of unique digits symbolizing the file. Then, Git backs up only the files which are different hash (Git does not care about the names of the files, it only considers the content.).

There are four Git objects:

- The **Blob** (*Binary Large Object*): It more commonly represents a **file**.
- The **Tree**: It more commonly represents a **directory** or **folder** of your application.

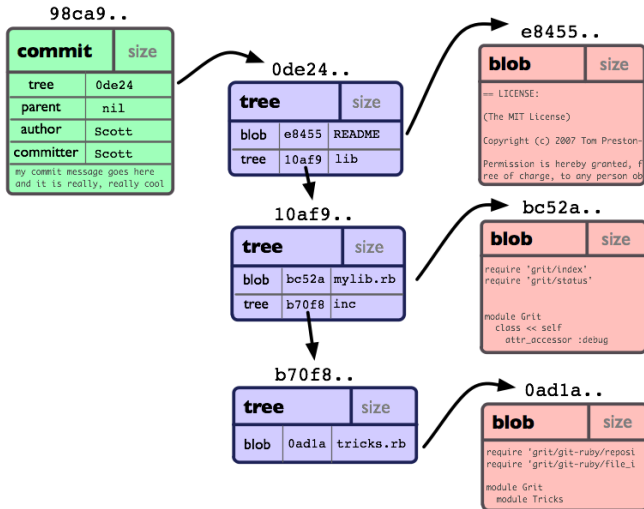
Its content is the list of SHA1s of Blobs or other Trees that it can contain. What gives a tree structure of files.

- The **Commit**: This is the complete state of your project at a given moment, i.e. a **snapshot**.

Its content is the SHA1 of the source Tree, and various information such as the name of the commit, the name of the author, the date, etc.

- The **Tag**: This is an object used to qualify a **particular commit** by giving it a comment.

An example



Each element has a unique SHA1.

How Git stores its information?

Only thanks to the directory `.git` at the root of your project.

- **config**: file relating to the configuration of the Git environment, such as information about the developer (name, email, etc.);
- **description**: contains information about your project;
- **objects/**: it is in this directory that all Git objects are stored (commits, tags, trees, blobs);
- **refs/***: contains information on local branches of the repository;
- **logs/***: contains log messages;
- **index**: file containing information about the status of the next commit.
- **HEAD**: pointer to current branch;
- **hooks/**: folder containing "hooks" or "triggers", i.e. actions/scripts that can be executed in pre or post condition.

Some questions

Some questions

- **How find a particular object?**

Some questions

- **How find a particular object?**

Thanks to SHA1, in particular: `.git/objects/`

Some questions

- **How find a particular object?**

Thanks to SHA1, in particular: `.git/objects/`

Example: **How find the blob "4A558..."?**

Some questions

- **How find a particular object?**

Thanks to SHA1, in particular: `.git/objects/`

Example: **How find the blob "4A558..."?**

See on `/.git/objects/4A/558...`

Some questions

- **How find a particular object?**

Thanks to SHA1, in particular: `.git/objects/`

Example: **How find the blob "4A558..."?**

See on `/.git/objects/4A/558...`

- **This structure is too heavy?**

Some questions

- **How find a particular object?**

Thanks to SHA1, in particular: `.git/objects/`

Example: **How find the blob "4A558..."?**

See on `/.git/objects/4A/558...`

- **This structure is too heavy?**

The answer is no thanks to the "Zlib" compression system that Git uses to compress the data and maintain the correct weight.

Some questions

- **How find a particular object?**

Thanks to SHA1, in particular: `.git/objects/`

Example: **How find the blob "4A558..."?**

See on `/.git/objects/4A/558...`

- **This structure is too heavy?**

The answer is no thanks to the "Zlib" compression system that Git uses to compress the data and maintain the correct weight.

Demonstration:

```
$ sudo apt install qpdf
```

```
$ zlib-flate -uncompress < FILE
```

- 1 The basics of git
 - Git in a nutshell
 - Basic commands
 - GitHub/GitLab
- 2 Branching, merging and problems
 - Data structures
 - **Branching**
 - Move in the commit tree
 - Undo changes with Git
 - Merging of two branches
- 3 Additional commands
 - Additional tips
 - Storage of your files
 - Bring in changes from a specific commit
 - Find a bad commit in your app

Branch

\$ `git branch`

List your branches.

A star (*) will appear next to the currently active branch.

\$ `git branch <branch-name>`

Create a new branch at the current commit.

\$ `git branch -d <branch-name>`

Delete the specified branch.

\$ `git checkout <branch-name>`

Switch to another branch and check it out into your working directory.

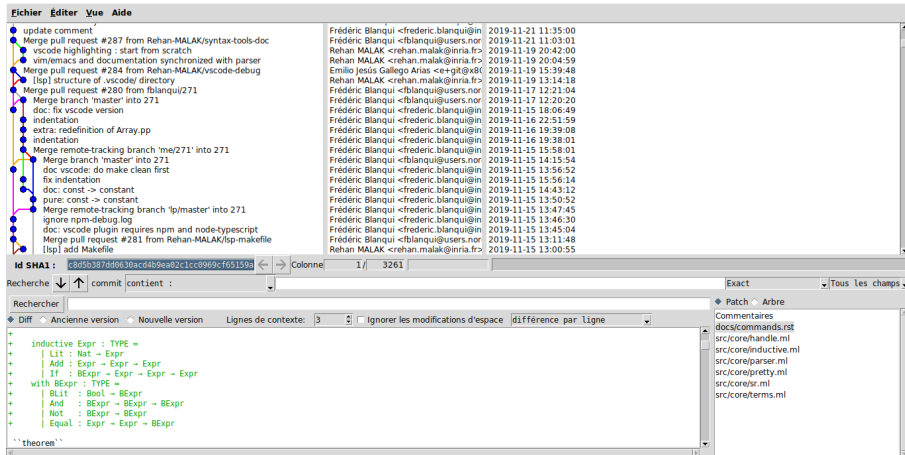
\$ `git merge <branch-name>`

Merge the specified branch's history into the current one.

\$ `git log`

Show all commits in the current branch's history.

Gitk - Graphical interface



Or if you prefer: `git log --graph`.

Examining logs, diffs and object information

```
$ git log branchB..branchA
```

Show the commits on branchA that are not on branchB.

```
$ git log --follow <file>
```

Show the commits that changed file, even across renames.

```
$ git diff branchB...branchA
```

Show the diff of what is in branchA that is not in branchB.

```
$ git log --stat -M
```

Show all commit logs with indication of any paths that moved.

```
$ git show <SHA>
```

Show any object in Git in human-readable format.

→ Easier on GitHub / GitLab

- 1 The basics of git
 - Git in a nutshell
 - Basic commands
 - GitHub/GitLab
- 2 Branching, merging and problems
 - Data structures
 - Branching
 - **Move in the commit tree**
 - Undo changes with Git
 - Merging of two branches
- 3 Additional commands
 - Additional tips
 - Storage of your files
 - Bring in changes from a specific commit
 - Find a bad commit in your app

A family story

A family story

Remember: A branch is a pointer to a commit!

A family story

Remember: A branch is a pointer to a commit!

HEAD is a special pointer to the branch we are currently working on.

A family story

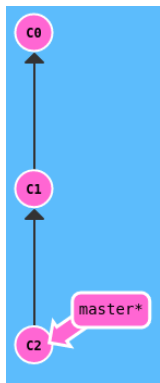
Remember: A branch is a pointer to a commit!

HEAD is a special pointer to the branch we are currently working on.

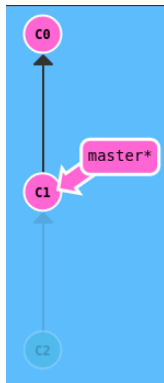
- HEAD: the current branch
- HEAD^: the parent of HEAD
- HEAD~4 : the great-great grandparent of HEAD
- `git branch -f master HEAD~3`: move (forced) the master three-parent branch behind HEAD.

- 1 The basics of git
 - Git in a nutshell
 - Basic commands
 - GitHub/GitLab
- 2 Branching, merging and problems
 - Data structures
 - Branching
 - Move in the commit tree
 - **Undo changes with Git**
 - Merging of two branches
- 3 Additional commands
 - Additional tips
 - Storage of your files
 - Bring in changes from a specific commit
 - Find a bad commit in your app

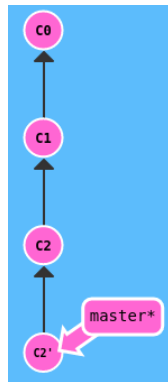
git reset VS git revert



Initial situation



git reset HEAD~1
(rewriting history)



git revert HEAD

git reset VS git revert

git reset VS git revert

```
$ git reset <commit>
```

Revert changes by moving a branch reference backwards in time to an older commit.

In this sense you can think of it as "rewriting history;" git reset will move a branch backwards as if the commit had never been made in the first place.

git reset VS git revert

```
$ git reset <commit>
```

Revert changes by moving a branch reference backwards in time to an older commit.

In this sense you can think of it as "rewriting history;" git reset will move a branch backwards as if the commit had never been made in the first place.

```
$ git reset --hard <commit>
```

Clear staging area, rewrite working tree from specified commit.

WARNING: You must be aware that everything you have coded since the last commit or the last pull will be lost!

git reset VS git revert

```
$ git reset <commit>
```

Revert changes by moving a branch reference backwards in time to an older commit.

In this sense you can think of it as "rewriting history;" git reset will move a branch backwards as if the commit had never been made in the first place.

```
$ git reset --hard <commit>
```

Clear staging area, rewrite working tree from specified commit.

WARNING: You must be aware that everything you have coded since the last commit or the last pull will be lost!

```
$ git revert <commit>
```

While resetting works great for local branches on your own machine, its method of "rewriting history" doesn't work for remote branches that others are using.

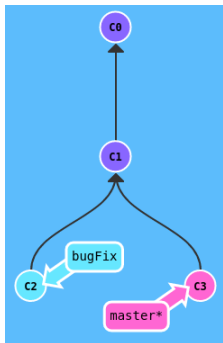
In order to reverse changes and share those reversed changes with others, we need to use git revert.

- 1 The basics of git
 - Git in a nutshell
 - Basic commands
 - GitHub/GitLab
- 2 Branching, merging and problems
 - Data structures
 - Branching
 - Move in the commit tree
 - Undo changes with Git
 - Merging of two branches
- 3 Additional commands
 - Additional tips
 - Storage of your files
 - Bring in changes from a specific commit
 - Find a bad commit in your app

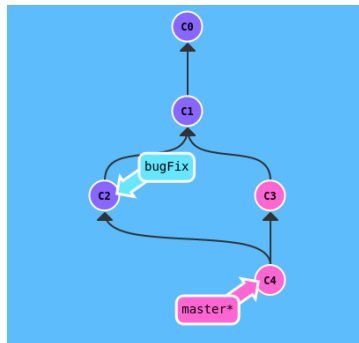
Git merge

Git merge

Merge the modifications of a given branch into the current branch (HEAD).



→ `git merge bugFix` →

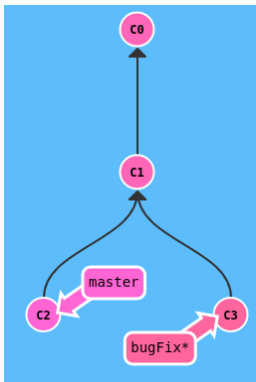


Git rebase

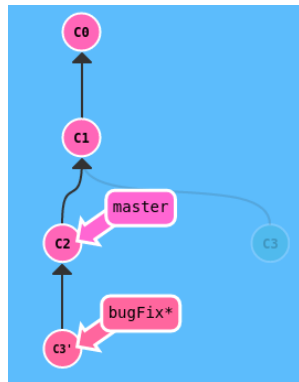
Rebasing essentially takes a set of commits, "copies" them, and plops them down somewhere else.

Git rebase

Rebasing essentially takes a set of commits, "copies" them, and plops them down somewhere else.



→ git rebase master →



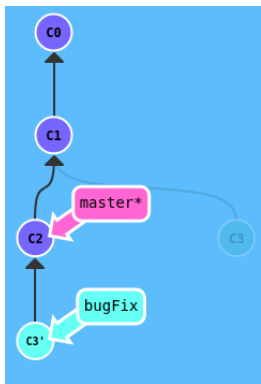
Git rebase

Rebasing essentially takes a set of commits, "copies" them, and plops them down somewhere else.

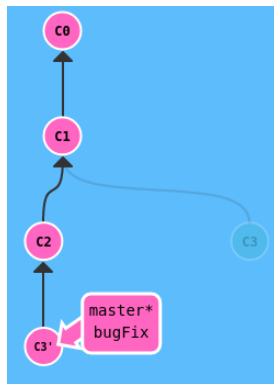


While this sounds confusing, the advantage of rebasing is that it can be used to make a nice linear sequence of commits.

Git rebase



→ `git rebase bugFix` →



Like `git merge`, no new commit because the branch `master` is an ancestor of the branch `bugFix`.

Note: The commit `C3` already exists.

See the difference

- `git pull`
= `git fetch; git merge origin/my-branch`
- Rebase from a branch: `git rebase <branch-name>`
- In case of conflicts, do after each conflict: `git rebase --continue`
- `git pull --rebase`
= `git fetch; git rebase origin/my-branch`

Bad situations after merging

Bad situations after merging

- You can have some conflicts:

```
<<<<<<<< HEAD
```

```
    print("Hello_World")
```

```
=====
```

```
    print("Saluton,_Mondo")
```

```
>>>>>>> Esperanto
```

Bad situations after merging

- You can have some conflicts:

```
<<<<<<<< HEAD
```

```
    print("Hello_World")
```

```
=====
```

```
    print("Saluton,_Mondo")
```

```
>>>>>>> Esperanto
```

- If the remote repository was updated during our changes.
Two possibilities:

Bad situations after merging

- You can have some conflicts:

<<<<<<<< **HEAD**

```
print("Hello_World")
```

=====

```
print("Saluton,_Mondo")
```

>>>>>>> **Esperanto**

- If the remote repository was updated during our changes.
Two possibilities:

- `git pull` is the solution.

Bad situations after merging

- You can have some conflicts:

```
<<<<<<<< HEAD
    print("Hello_World")
=====
    print("Saluton,_Mondo")
>>>>>>> Esperanto
```

- If the remote repository was updated during our changes.
Two possibilities:
 - `git pull` is the solution.
 - You need to avoid the last commit thanks to `git reset HEAD~`.
Then, do `git pull` and create your commit again.

Bad situations after merging

- You can have some conflicts:

```
<<<<<<<< HEAD
    print("Hello_World")
=====
    print("Saluton,_Mondo")
>>>>>>> Esperanto
```

- If the remote repository was updated during our changes.
Two possibilities:
 - `git pull` is the solution.
 - You need to avoid the last commit thanks to `git reset HEAD~`.
Then, do `git pull` and create your commit again.
- `git status` returns files that could not be merged (listed as "unmerged")

Bad situations after merging

- You can have some conflicts:

```
<<<<<<<< HEAD
    print("Hello_World")
=====
    print("Saluton,_Mondo")
>>>>>>> Esperanto
```

- If the remote repository was updated during our changes.
Two possibilities:
 - `git pull` is the solution.
 - You need to avoid the last commit thanks to `git reset HEAD~`.
Then, do `git pull` and create your commit again.
- `git status` returns files that could not be merged (listed as "unmerged")
- To mark conflicts in a resolved <file> file, `git add <file>` must be done.

Let's go again!

Exercise: Fix the conflict!

Play with: `learngitbranching.js.org`

- 1 The basics of git
 - Git in a nutshell
 - Basic commands
 - GitHub/GitLab
- 2 Branching, merging and problems
 - Data structures
 - Branching
 - Move in the commit tree
 - Undo changes with Git
 - Merging of two branches
- 3 Additional commands
 - Additional tips
 - Storage of your files
 - Bring in changes from a specific commit
 - Find a bad commit in your app

- 1 The basics of git
 - Git in a nutshell
 - Basic commands
 - GitHub/GitLab
- 2 Branching, merging and problems
 - Data structures
 - Branching
 - Move in the commit tree
 - Undo changes with Git
 - Merging of two branches
- 3 Additional commands
 - Additional tips
 - Storage of your files
 - Bring in changes from a specific commit
 - Find a bad commit in your app

Additional tips

Additional tips

- Delete files from a repository:

Additional tips

- Delete files from a repository:
 - I pushed a file to the remote repository that shouldn't have gone there.
I want to remove it from the repository.

Additional tips

- Delete files from a repository:
 - I pushed a file to the remote repository that shouldn't have gone there.
I want to remove it from the repository.
 - Two cases:

Additional tips

- Delete files from a repository:
 - I pushed a file to the remote repository that shouldn't have gone there. I want to remove it from the repository.
 - Two cases:
 1. I want to keep the file locally on my computer:
`git rm --cached <file(s)>`

Additional tips

- Delete files from a repository:
 - I pushed a file to the remote repository that shouldn't have gone there. I want to remove it from the repository.
 - Two cases:
 1. I want to keep the file locally on my computer:
`git rm --cached <file(s)>`
 2. I don't want to keep it: `git rm <file(s)>`

Additional tips

- Delete files from a repository:
 - I pushed a file to the remote repository that shouldn't have gone there. I want to remove it from the repository.
 - Two cases:
 1. I want to keep the file locally on my computer:
`git rm --cached <file(s)>`
 2. I don't want to keep it: `git rm <file(s)>`
 - Then I commit, I push.

Additional tips

- Delete files from a repository:
 - I pushed a file to the remote repository that shouldn't have gone there. I want to remove it from the repository.
 - Two cases:
 1. I want to keep the file locally on my computer:
`git rm --cached <file(s)>`
 2. I don't want to keep it: `git rm <file(s)>`
 - Then I commit, I push.
- `git checkout -b <branch-name>`
= `git branch <branch-name> ; git checkout <branch-name>`

Additional tips

- Delete files from a repository:
 - I pushed a file to the remote repository that shouldn't have gone there. I want to remove it from the repository.
 - Two cases:
 1. I want to keep the file locally on my computer:
`git rm --cached <file(s)>`
 2. I don't want to keep it: `git rm <file(s)>`
 - Then I commit, I push.
- `git checkout -b <branch-name>`
= `git branch <branch-name>` ; `git checkout <branch-name>`
- `git checkout -`
Switch between your 2 only branches.

Additional tips

- Delete files from a repository:
 - I pushed a file to the remote repository that shouldn't have gone there. I want to remove it from the repository.
 - Two cases:
 1. I want to keep the file locally on my computer:
`git rm --cached <file(s)>`
 2. I don't want to keep it: `git rm <file(s)>`
 - Then I commit, I push.
- `git checkout -b <branch-name>`
= `git branch <branch-name>` ; `git checkout <branch-name>`
- `git checkout -`
Switch between your 2 only branches.
- Create a tag: `git tag <version-name> <commit>`

Additional tips

- Delete files from a repository:
 - I pushed a file to the remote repository that shouldn't have gone there. I want to remove it from the repository.
 - Two cases:
 1. I want to keep the file locally on my computer:
`git rm --cached <file(s)>`
 2. I don't want to keep it: `git rm <file(s)>`
 - Then I commit, I push.
- `git checkout -b <branch-name>`
= `git branch <branch-name>` ; `git checkout <branch-name>`
- `git checkout -`
Switch between your 2 only branches.
- Create a tag: `git tag <version-name> <commit>`
- *I only use the terminal, and I would like to see the graph of branches on it:* `git log --oneline --graph --decorate --graph`

Additional tips

- Delete files from a repository:
 - I pushed a file to the remote repository that shouldn't have gone there. I want to remove it from the repository.
 - Two cases:
 1. I want to keep the file locally on my computer:
`git rm --cached <file(s)>`
 2. I don't want to keep it: `git rm <file(s)>`
 - Then I commit, I push.
- `git checkout -b <branch-name>`
= `git branch <branch-name>` ; `git checkout <branch-name>`
- `git checkout -`
Switch between your 2 only branches.
- Create a tag: `git tag <version-name> <commit>`
- *I only use the terminal, and I would like to see the graph of branches on it:* `git log --oneline --graph --decorate --graph`
- Don't forget the manual: `man git <...>`

Additional tips

- Delete files from a repository:
 - I pushed a file to the remote repository that shouldn't have gone there. I want to remove it from the repository.
 - Two cases:
 1. I want to keep the file locally on my computer:
`git rm --cached <file(s)>`
 2. I don't want to keep it: `git rm <file(s)>`
 - Then I commit, I push.
- `git checkout -b <branch-name>`
= `git branch <branch-name>` ; `git checkout <branch-name>`
- `git checkout -`
Switch between your 2 only branches.
- Create a tag: `git tag <version-name> <commit>`
- *I only use the terminal, and I would like to see the graph of branches on it:* `git log --oneline --graph --decorate --graph`
- Don't forget the manual: `man git <...>`
- Don't forget the bible: <https://git-scm.com/book/en/v2>

- 1 The basics of git
 - Git in a nutshell
 - Basic commands
 - GitHub/GitLab
- 2 Branching, merging and problems
 - Data structures
 - Branching
 - Move in the commit tree
 - Undo changes with Git
 - Merging of two branches
- 3 **Additional commands**
 - Additional tips
 - **Storage of your files**
 - Bring in changes from a specific commit
 - Find a bad commit in your app

Imagine a world where you can...

Imagine a world where you can...

1. Put your changes aside.

Imagine a world where you can...

1. Put your changes aside.
2. Checkout another branch.

Imagine a world where you can...

1. Put your changes aside.
2. Checkout another branch.
3. Apply your changes later.

Imagine a world where you can...

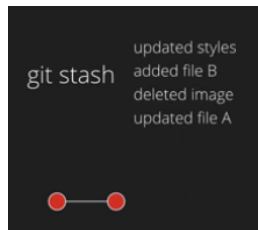
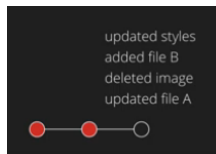
1. Put your changes aside.
2. Checkout another branch.
3. Apply your changes later.

→ **It's real, thanks to** `git stash`!

Imagine a world where you can...

1. Put your changes aside.
2. Checkout another branch.
3. Apply your changes later.

→ **It's real, thanks to git stash!**



The wonderful world of `git stash`

The wonderful world of git stash

- \$ `git stash [push]` or
`git stash push [-m "<descriptive message>"]`
Save modified and staged changes.
- \$ `git stash [push] -u`
Save modified, staged and **untrack** changes.

The wonderful world of git stash

- \$ `git stash [push]` or
`git stash push [-m "<descriptive message>"]`
Save modified and staged changes.
- \$ `git stash [push] -u`
Save modified, staged and **untrack** changes.
- \$ `git stash list`
List stack-order of stashed file changes
(See `stash@{<stash-index>}`).

The wonderful world of git stash

- \$ `git stash [push]` or
`git stash push [-m "<descriptive message>"]`
Save modified and staged changes.
- \$ `git stash [push] -u`
Save modified, staged and **untrack** changes.
- \$ `git stash list`
List stack-order of stashed file changes
(See `stash@{<stash-index>}`).
- \$ `git stash show <stash-index>` (Ex. `git stash show 0`)
Show changes.
- \$ `git stash show -p <stash-index>`
Show changes in full tree-view.

The wonderful world of `git stash`

The wonderful world of git stash

```
$ git stash branch <branch-name> <stash-index>
```

Create a branch from stash.

The wonderful world of git stash

```
$ git stash branch <branch-name> <stash-index>
```

Create a branch from stash.

```
$ git stash pop [<stash-index>]
```

Remove a single stashed state from the stash list and apply it on top of the current working tree state, i.e., do the inverse operation of git stash push.

The wonderful world of git stash

```
$ git stash branch <branch-name> <stash-index>
```

Create a branch from stash.

```
$ git stash pop [<stash-index>]
```

Remove a single stashed state from the stash list and apply it on top of the current working tree state, i.e., do the inverse operation of git stash push.

```
$ git stash apply [<stash-index>]
```

Like pop, but do not remove the state from the stash list.

The wonderful world of git stash

```
$ git stash branch <branch-name> <stash-index>
```

Create a branch from stash.

```
$ git stash pop [<stash-index>]
```

Remove a single stashed state from the stash list and apply it on top of the current working tree state, i.e., do the inverse operation of git stash push.

```
$ git stash apply [<stash-index>]
```

Like pop, but do not remove the state from the stash list.

```
$ git stash drop [<stash-index>]
```

Remove a single stash entry from the list of stash entries. When no <stash-index> is given, it removes the latest one. i.e. stash@{0}.

The wonderful world of git stash

```
$ git stash branch <branch-name> <stash-index>
```

Create a branch from stash.

```
$ git stash pop [<stash-index>]
```

Remove a single stashed state from the stash list and apply it on top of the current working tree state, i.e., do the inverse operation of git stash push.

```
$ git stash apply [<stash-index>]
```

Like pop, but do not remove the state from the stash list.

```
$ git stash drop [<stash-index>]
```

Remove a single stash entry from the list of stash entries. When no <stash-index> is given, it removes the latest one. i.e. stash@{0}.

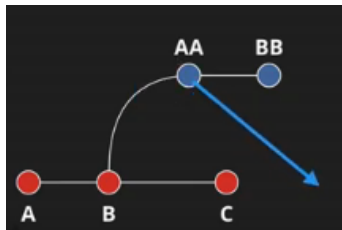
```
$ git stash clear
```

Remove all the stash entries.

- 1 The basics of git
 - Git in a nutshell
 - Basic commands
 - GitHub/GitLab
- 2 Branching, merging and problems
 - Data structures
 - Branching
 - Move in the commit tree
 - Undo changes with Git
 - Merging of two branches
- 3 Additional commands
 - Additional tips
 - Storage of your files
 - Bring in changes from a specific commit
 - Find a bad commit in your app

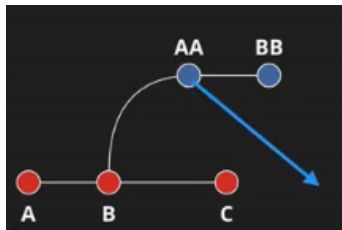
git cherry-pick

It's a very straightforward way of saying that you would like to copy a series of commits below your current location (HEAD).



git cherry-pick

It's a very straightforward way of saying that you would like to copy a series of commits below your current location (HEAD).

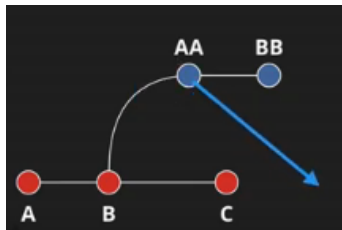


```
$ git cherry-pick <SHA> or <commit-ID> or  
git cherry-pick <commit_1> <commit_2> <...>
```

Like rebase, this create a new commit.

git cherry-pick

It's a very straightforward way of saying that you would like to copy a series of commits below your current location (HEAD).



```
$ git cherry-pick <SHA> or <commit-ID> or  
git cherry-pick <commit_1> <commit_2> <...>
```

Like rebase, this create a new commit.

```
$ git cherry-pick <SHA> or <commit-ID> -n
```

Don't create a new commit. So you need to do `git commit` after.

Git Interactive Rebase

- `git cherry-pick` is great when you know which commits you want and you know their corresponding hashes.

- `git cherry-pick` is great when you know which commits you want and you know their corresponding hashes.
- But what about the situation where you don't know what commits you want?

- `git cherry-pick` is great when you know which commits you want and you know their corresponding hashes.
- But what about the situation where you don't know what commits you want?
- We can use interactive rebasing for this – it's the best way to review a series of commits you're about to rebase.

Git Interactive Rebase

- `git cherry-pick` is great when you know which commits you want and you know their corresponding hashes.
- But what about the situation where you don't know what commits you want?
- We can use interactive rebasing for this – it's the best way to review a series of commits you're about to rebase.
 - `git rebase -i <commit-destination>`

Git Interactive Rebase

- `git cherry-pick` is great when you know which commits you want and you know their corresponding hashes.
- But what about the situation where you don't know what commits you want?
- We can use interactive rebasing for this – it's the best way to review a series of commits you're about to rebase.
 - `git rebase -i <commit-destination>`
 - Boost yourself on <https://learngitbranching.js.org>

- 1 The basics of git
 - Git in a nutshell
 - Basic commands
 - GitHub/GitLab
- 2 Branching, merging and problems
 - Data structures
 - Branching
 - Move in the commit tree
 - Undo changes with Git
 - Merging of two branches
- 3 Additional commands
 - Additional tips
 - Storage of your files
 - Bring in changes from a specific commit
 - Find a bad commit in your app

Find a bad commit in your app : `git bisect`



Find a bad commit in your app : git bisect



```
$ git bisect start
```

Start the "bisect" process.

Find a bad commit in your app : git bisect



```
$ git bisect start
```

Start the "bisect" process.

```
$ git bisect good [<SHA> or <commit-ID>]
```

Say that a commit is good, i.e. without a bug.

When no option is given, the current commit is considered.

Find a bad commit in your app : git bisect



```
$ git bisect start
```

Start the "bisect" process.

```
$ git bisect good [<SHA> or <commit-ID>]
```

Say that a commit is good, i.e. without a bug.

When no option is given, the current commit is considered.

```
$ git bisect bad [<SHA> or <commit-ID>]
```

Say that a commit is bad, i.e. with a bug.

When no option is given, the current commit is considered.

Find a bad commit in your app : git bisect



```
$ git bisect start
```

Start the "bisect" process.

```
$ git bisect good [<SHA> or <commit-ID>]
```

Say that a commit is good, i.e. without a bug.

When no option is given, the current commit is considered.

```
$ git bisect bad [<SHA> or <commit-ID>]
```

Say that a commit is bad, i.e. with a bug.

When no option is given, the current commit is considered.

```
$ git bisect reset
```

Stop the "bisect" process.

Find a bad commit in your app : git bisect



```
$ git bisect start
```

Start the "bisect" process.

```
$ git bisect good [<SHA> or <commit-ID>]
```

Say that a commit is good, i.e. without a bug.

When no option is given, the current commit is considered.

```
$ git bisect bad [<SHA> or <commit-ID>]
```

Say that a commit is bad, i.e. with a bug.

When no option is given, the current commit is considered.

```
$ git bisect reset
```

Stop the "bisect" process.

Note : Each time you used `git bisect good/bad`, the current commit is changed, thanks to a dichotomic process.

Let's go again, again!

Find and fix the bug that is somewhere here:

`https://github.com/amelieled/GL\_bisect\_GL\_MPRI.git`

Or, here, if you prefer Python:

`https://github.com/charlie-j/MPRI-SE-Click-Run`

- Interactive tutorial:
 - learngitbranching.js.org
- Cheat sheet:
 - <https://education.github.com/git-cheat-sheet-education.pdf>
(English version)
 - <https://training.github.com/downloads/fr/github-git-cheat-sheet.pdf> (French version)
 - <https://ndpsoftware.com/git-cheatsheet.html>
(Interactive one - English, French, Chinese, Spanish, German, Korean)
- Reference book : <http://git-scm.com/book>
(<https://git-scm.com/book/fr/v2/> in French)