Software Engineering at MPRI
Advanced tutorial on git, and its extensions

Amélie Ledein
ledein@lsv.fr

December 6, 2020
Remember
Remember

- too easier...
Remember too easier...
... because sometimes we have some difficult problems!
Additional tips

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>`

   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:

Delete files from a repository:

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>`

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:

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.

- To keep the file locally on your computer:
  - `git rm --cached <file(s)>`

- If you don’t want to keep it:
  - `git rm <file(s)>`

  Then you commit and push:
  - `git checkout -b <branch-name>`
  - `git branch <branch-name>` ; `git checkout <branch-name>`

- Switch between your 2 only branches.

- To create a tag:
  - `git tag <version-name> <commit>`

- You only use the terminal, and you 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:
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:

```
```
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)>
     ```
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)>
     ```
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>`

  - Don’t forget the manual:
    ```
    man git ...
    ```

  - Don’t forget the bible:
    ```
    ```
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.
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`
Sommaire

1 Move in the commit tree

2 Undo changes with Git

3 Storage of your files

4 Merging of two branches

5 Bring in changes from a specific commit

6 Find a bad commit in your app
A family story

Remember: A branch is a pointer to a commit!

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.
Remember: A branch is a pointer to a commit!
Remember: A branch is a pointer to a commit!

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

1 Move in the commit tree

2 Undo changes with Git

3 Storage of your files

4 Merging of two branches

5 Bring in changes from a specific commit

6 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 <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.
$ 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.
Sommaire

1. Move in the commit tree
2. Undo changes with Git
3. Storage of your files
4. Merging of two branches
5. Bring in changes from a specific commit
6. Find a bad commit in your app
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.
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

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

- **git stash -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>**
  - Show changes
  - (Ex. `git stash show 0`)

- **git stash show -p <stash-index>**
  - Show changes in full tree-view.

Amélie Ledein
Advanced tutorial
December 6, 2020 11 / 24
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>}).

$ 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

$ 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

- `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.
$ 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.
$ 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 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.
Sommaire

1. Move in the commit tree
2. Undo changes with Git
3. Storage of your files
4. Merging of two branches
5. Bring in changes from a specific commit
6. Find a bad commit in your app
Git merge

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

→ git merge bugFix
Git merge

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

→ git merge bugFix →
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.

→ git rebase master →

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

You can have some conflicts:

```
print("Hello World")
```

```
print("Saluton, Mondo")
```

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>`, `git add <file>` must be done.
Bad situations after merging

- You can have some conflicts:
  
  ```python
  <<<<<<  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:
  
  `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>`, `git add <file>` must be done.
Bad situations after merging

- You can have some conflicts:
  
  ```python
  >>><<<<<<<<<<< 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:

```python
<<<<<<<<< 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:
  ```python
  >>>>>> 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:
  ```python
  >>> 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.
Sommaire

1. Move in the commit tree
2. Undo changes with Git
3. Storage of your files
4. Merging of two branches
5. Bring in changes from a specific commit
6. Find a bad commit in your app
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).

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

Like rebase, this create a new commit.
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. 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
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 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 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 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
Sommaire

1 Move in the commit tree
2 Undo changes with Git
3 Storage of your files
4 Merging of two branches
5 Bring in changes from a specific commit
6 Find a bad commit in your app
Find a bad commit in your app: `git bisect`

```bash
$ 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.

Amélie Ledein
Advanced tutorial
December 6, 2020
23 / 24
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 dichotomonic process.
Exercise

Find and fix the bug that is somewhere here:
https://github.com/amelieled/GL_bisect_GL_MPRI.git