Using git in XLiFE++

Nicolas Kielbasiewicz

May 15, 2014

The aim of this document is to explain the main useful features and commands of the scm called git. For each command, only a few options will be explained. To learn more about a specific command, you may use :

```
git help [<command>]
```

If you want to get help for a specific command, you may give it after the help keyword. For example, if you want help for the git init command, you will type : git help init.

Documentation, tutorials, ... are also available on the Git website :

http://git-scm.com/

Contents

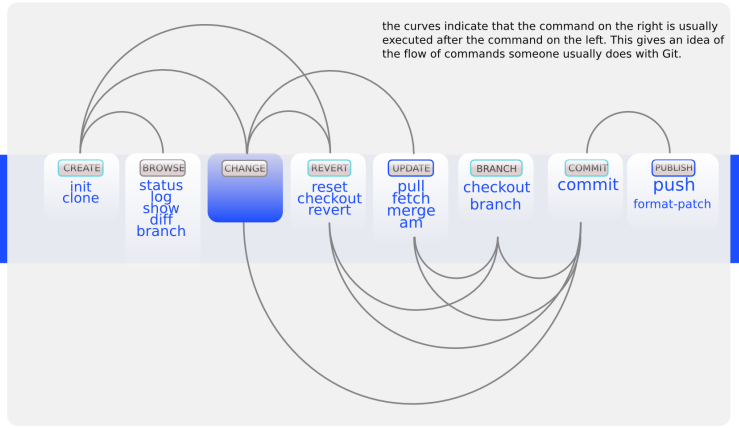
Git Cheat Sheet

http://git.or.cz/

Remember: git command --help

Global Git configuration is stored in $HOME/.gitconfig (git config --help)

Commands Sequence



Create

From existing data
```
cd ~/projects/myproject
git init
git add .
```

From existing repo
```
git clone ~/existing/repo ~/new/repo
git clone git://host.org/project.git
git clone ssh://you@host.org/proj.git
```

Show

Files changed in working directory
```
git status
```

Changes to tracked files
```
git diff
```

What changed between $ID1 and $ID2
```
git diff $id1  $id2
```

History of changes
```
git log
```

History of changes for file with diffs
```
git log -p $file $dir/ec/tory/
```

Who changed what and when in a file
```
git blame $file
```

A commit identified by $ID
```
git show $id
```

A specific file from a specific $ID
```
git show $id:$file
```

All local branches
```
git branch
```
(star '*' marks the current branch)

Cheat Sheet Notation

$id : notation used in this sheet to represent either a commit id, branch or a tag name
$file : arbitrary file name
$branch : arbitrary branch name

Concepts

Git Basics

```
master    : default development branch
origin    : default upstream repository
HEAD      : current branch
HEAD^     : parent of HEAD
HEAD~4    : the great-great grandparent of HEAD
```

Revert

Return to the last committed state
```
git reset --hard
```
⚠ you cannot undo a hard reset

Revert the last commit
```
git revert HEAD
```
Creates a new commit

Revert specific commit
```
git revert $id
```
Creates a new commit

Fix the last commit
```
git commit -a --amend
```
(after editing the broken files)

Checkout the $id version of a file
```
git checkout $id $file
```

Branch

Switch to the $id branch
```
git checkout $id
```

Merge branch1 into branch2
```
git checkout $branch2
git merge branch1
```

Create branch named $branch based on the HEAD
```
git branch $branch
```

Create branch $new_branch based on branch $other and switch to it
```
git checkout -b $new_branch $other
```

Delete branch $branch
```
git branch -d $branch
```

Update

Fetch latest changes from origin
```
git fetch
```
(but this does not merge them).

Pull latest changes from origin
```
git pull
```
(does a fetch followed by a merge)

Apply a patch that some sent you
```
git am -3 patch.mbox
```
(in case of a conflict, resolve and use git am --resolved)

Publish

Commit all your local changes
```
git commit -a
```

Prepare a patch for other developers
```
git format-patch origin
```

Push changes to origin
```
git push
```

Mark a version / milestone
```
git tag v1.0
```

Useful Commands

Finding regressions
```
git bisect start          (to start)
git bisect good $id       ($id is the last working version)
git bisect bad $id        ($id is a broken version)

git bisect bad/good       (to mark it as bad or good)
git bisect visualize      (to launch gitk and mark it)
git bisect reset          (once you're done)
```

Check for errors and cleanup repository
```
git fsck
git gc --prune
```

Search working directory for foo()
```
git grep "foo()"
```

Resolve Merge Conflicts

To view the merge conclicts
```
git diff          (complete conflict diff)
git diff --base  $file  (against base file)
git diff --ours  $file  (against your changes)
git diff --theirs $file  (against other changes)
```

To discard conflicting patch
```
git reset --hard
git rebase --skip
```

After resolving conflicts, merge with
```
git add $conflicting_file  (do for all resolved files)
git rebase --continue
```

Zack Rusin
Based on the work of:
Sébastien Pierre
Xprima Corp.

2 Useful git commands for XLiFE++

2.1 Configuring git

2.1.1 With commands

When you use git for the first time, you have to configure some information about your name, your email

```
git config --global user.name "Your Name"
git config --global user.email "your.email@example.com"
```

If you have to use patches management by email, you also will have to define some information about your email address.

```
git config --global sendemail.smtpserver <smtp.example.com>
git config --global sendemail.smtpserverport 465
git config --global sendemail.smtpencryption tls
git config --global sendemail.smtpuser <your-id>
git config --global sendemail.smtppass <your-passwd>
```

2.1.2 With file edition

There is another way to define those global parameters, namely define directly the ~/.gitconfig file. The previous commands edit this file and the result is :

```
[user]
        name = Your Name
        email = your.email\example.com[sendemail]smtpserver = smtp.example.comsmtpuser =
            <your-id>smtppass = <your-passwd>smtpserverport = 465smtpencryption =
            <your-encryption>
```

2.1.3 Example with Gmail

Here is the ~/.gitconfig for a gmail address :

```
[user]
        name = Your Name
        email = <your-id>\gmail.com[sendemail]smtpserver = smtp.gmail.comsmtpuser =
            <your-id>gmail.com
        smtpserverport = 587
        smtppass = <your-passwd>
        smtpencryption = ssl
```

2.2 Create a repository

2.2.1 Create an empty repository

To create an empty git repository, you may go to the root directory of your project and use the following command :

```
git init [--bare]
```

This create what is called the GIT_DIR, namely a directory named .git in your root directory. The GIT_DIR contains your config for the project and in a zipped way, the list of commits and tracked files. There is a useful option you may use, –bare, in order to create a bare repository, which consist of just the GIT_DIR. This option is very useful when you want to have a main repository to which every project member is synchronized and you want to spare memory.

2.2.2 Clone a distant repository

When you clone a distant repository, you create a repository connected to the distant one, getting all tracked files, the history of commits and everything you need to work in the project. To do so, you will use the following command :

```
git clone [--bare] <repository>
```

Now, the distant repository is your remote called origin, namely the repository with whom you are connected.
For XLiFE++, you can go to the sources tab on the forge website.

2.3 Browse a repository

2.3.1 Files changed in working directory

To know the list of modified files, deleted ones or created ones, you can use the following command :

```
git status
```

For example, if you added two files toto.txt and tata.txt, the git status output will be :

```
# On branch master
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       tata.txt
#       toto.txt
nothing added to commit but untracked files present (use "git add" to track)
```

If you didn't have any modification, the git status output should have been :

```
# On branch master
nothing to commit (working directory clean)
```

2.3.2 Changes to tracked files

To know the list of changes in tracked files, you can use the following command :

```
git diff [<id1> <id2>] [<file>]
```

Without any arguments, the git diff command gives every change in every file since the last commit. If you specify a file, then only changes of this file will be shown. If you specify 2 commits id, then changes between the two commits will be shown.

2.3.3 Who changed what in a file ?

To know who changed what in a file, so in order to know who is to blame for the bug you have ;-), you can use the following command :

```
git blame <file>
```

2.4 Manage branches

What makes git very powerful is the management of development branches. A branch is a state of your project in the source control. With git, you can have very easily different states of your project.

2.4.1 Create another branch

To create a branch based on the HEAD, namely the current state of your project in git, you can use the following command :

```
git branch <name>
```

2.4.2 Display all branches names

To know the name of each local branch in your project, you can use the following command :

```
git branch [-r]
```

With the -r option, you will have branches on the remote origin.

2.4.3 Switch to another branch

To switch to another branch, you may use the following command :

```
git checkout <branchname>
```

You may me aware that uncommitted changes do not belong to a branch. So, be aware of what you are doing when you switch to another branch.

2.4.4 Delete a branch

To delete branch, you may use the following command :

```
git branch -d <branchname>
```

All changes made in the branch will be now in the current branch. If you want to delete a branch and all commits in it, you may use :

```
git branch -D <branchname>
```

In both cases, make sure you are in an other branch than the one you want to delete.

2.4.5 Merge a branch in the current one

To merge a branch in the current one, you may use the following command :

```
git merge <branchname>
```

2.5 Update a repository

2.5.1 Pull latest changes from distant repository

To get updates from the remote origin, you can use the following command :

```
git pull
```

2.5.2 Apply a patch that someone sent you

When you receive a patch, by email, you may place it in a particular mailbox – for example patch.mbox – and then use the following command to apply it :

```
git am -3 patch.mbox
```

The -3 option tells to merge each patch. In case of conflicts, you will have to resolve them manually and apply the following command to achieve the patch application :

```
git am --resolved
```

2.6 Publish in a repository

2.6.1 Commit all changes

To commit all changes you made since the last commit (or since the creation of the repository), you can use the following command :

```
git commit -a [-m <msg>]
```

You can specify the commit message with the -m option, followed by the string, between quotes, representing the message.

2.6.2 Push changes to origin

We saw that **git pull** is used to get changes from a distant repository. To set our commits to origin, you can use the following command :

```
git push
```

There is some control mechanism behind the push. Indeed, if there is updates on origin you didn't pull, the push will not work and will warn you to pull before pushing, to prevent time conflicts.
If you generate conflicts, the push will not protect the distant repository insofar as you have no access to it and resolve conflicts. That is why it is often said that pushing commits may be risky.

2.6.3 Manage patches

A safer way to publish changes than pushing is sending a patch. To build a patch from commits, you may use the following command :

```
git format-patch origin [--numbered-files] [-o <dir>]
```

The patch will contain every commit done since the last pull. In case there are several commits to patch, one file per commit will be generated. Their extension will be .patch. The name of the file will be based on a number and the first line of the commit message. If you use the –numbered-files option, the name will only based with a number. By default, generated files are in the current working directory. If you want to have them in a particular directory, you may use the -o option.

Your patch is ready, and now, you want to send it to other developers. For that purpose, you can use the following command :

```
git send-email *.patch
```

It will ask you who to send, who sends the mail, ... and send it.

2.7 Reverting a repository

2.7.1 Fix the last commit

Instead of creating a new commit, you may fix the last one. For this purpose, you will use the –amend option of the command git commit :

```
git commit -a --amend [-m <msg>]
```

2.7.2 Undo changes since the last commit

You are in a state where all the changes done since the last commit must be undone. To do so, you may use the following command :

```
git reset --hard
```

If you omit the –hard option, all commits will be deleted, but files will not be modified.

2.7.3 Undo commits

Undoing a commit generates a new commit which is the inverse of the commit undone. That is why the command dedicated to undo commit is git revert :

```
git revert <commit>
```

For example, if you want to revert the commit HEAD, you create a new commit undoing it.

3 Git GUIs

3.1 The Git GUI on Unix systems

Git provides the command `git gui` to visualize changes to index, like added files, deleted files or modified files since last commit, and other features such as history of commits. git also provides gitk, a Tcl/Tk gui to show directly history of commits. In fact, when you want the history of commits in git gui, it launches gitk.

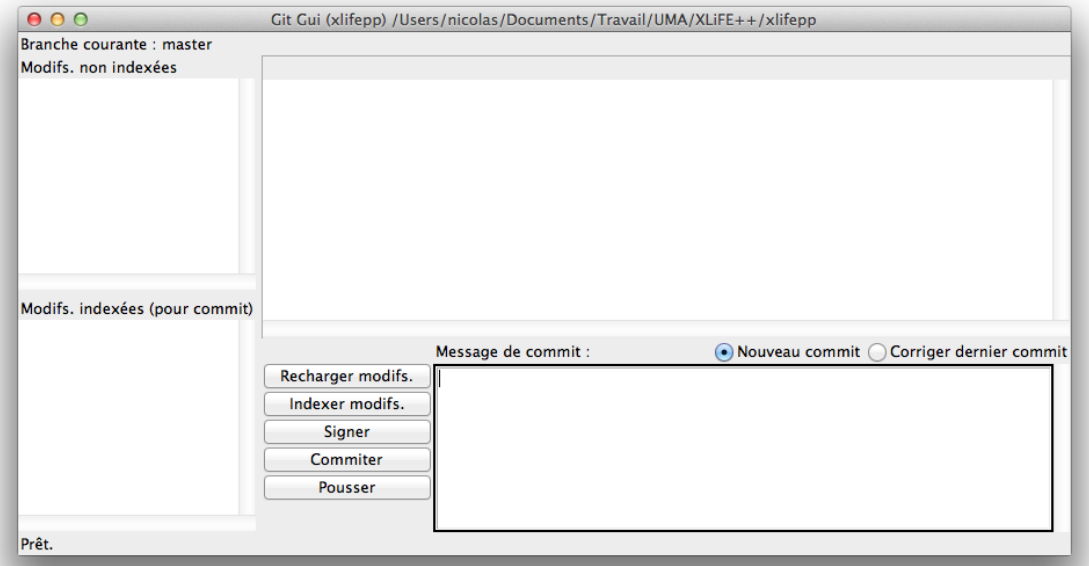


Figure 1: The git gui user interface

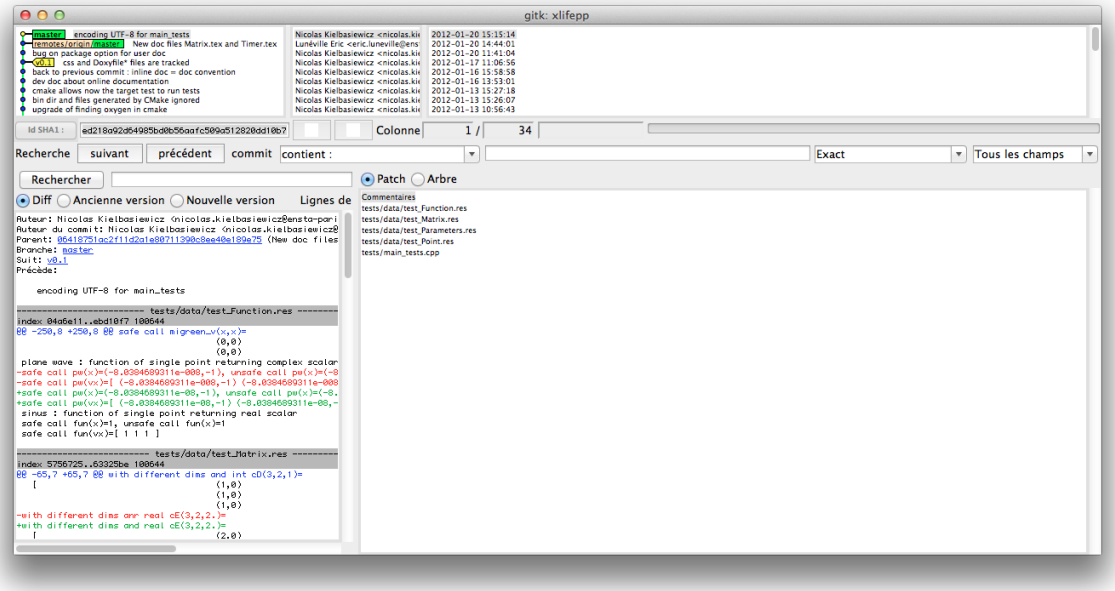


Figure 2: The gitk user interface

3.2 TortoiseGit on Windows

TortoiseGit is a contextual Windows application for Git. When you are in a directory under the control of git, you have access to contextual menus with a right click on your mouse. In the TortoiseGit submenu, you can configure for example your public key to pull/push with the XLiFE++ repository on the forge.

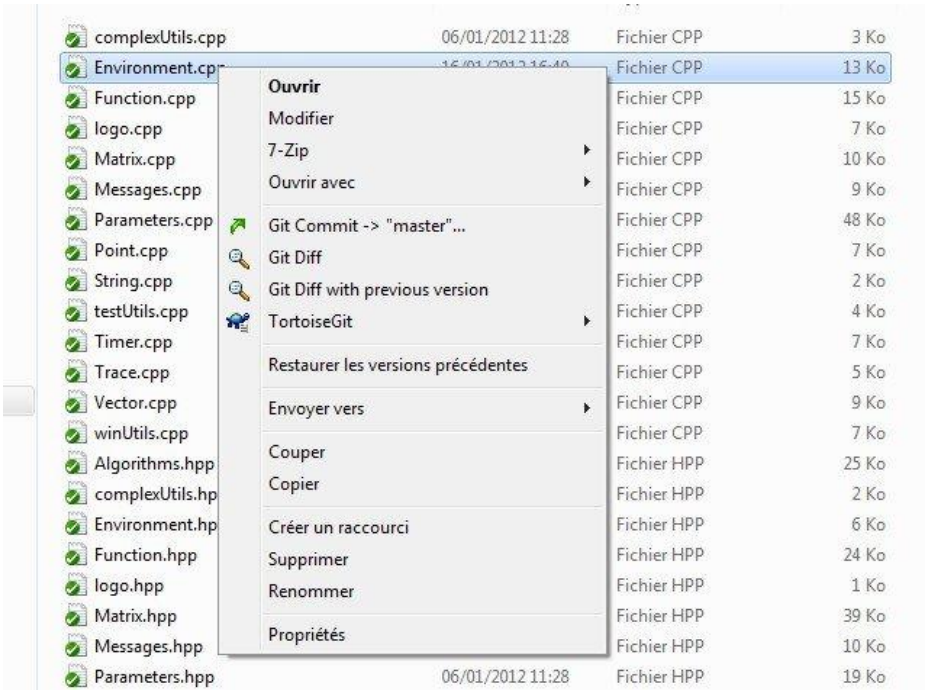


Figure 3: The git gui user interface

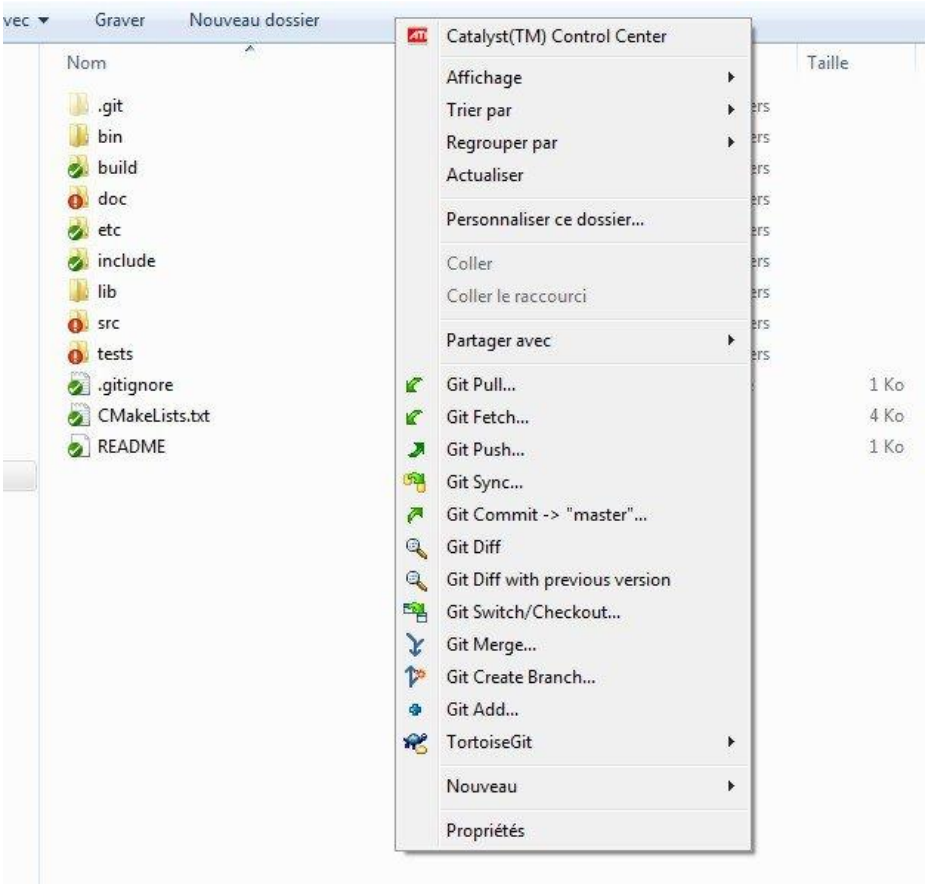


Figure 4: The gitk user interface

3.3 GitX on Mac

GitX is a Mac app doing the same work than git gui and gitk but more easy-to-use

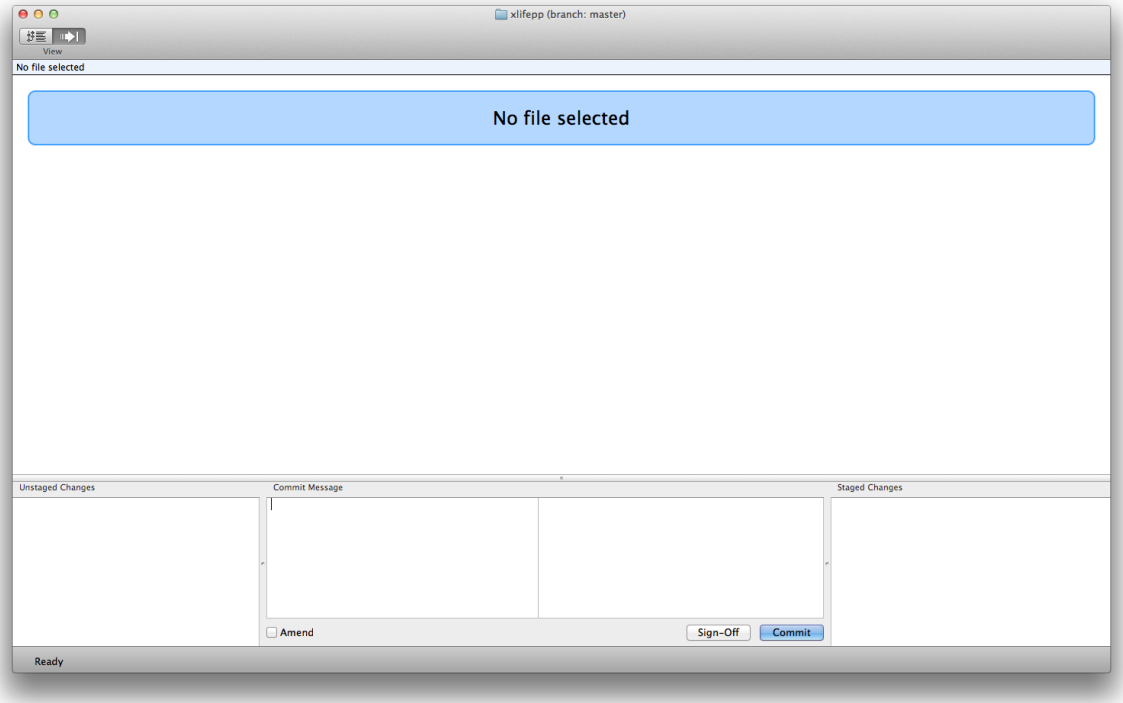


Figure 5: The gitx commit user interface

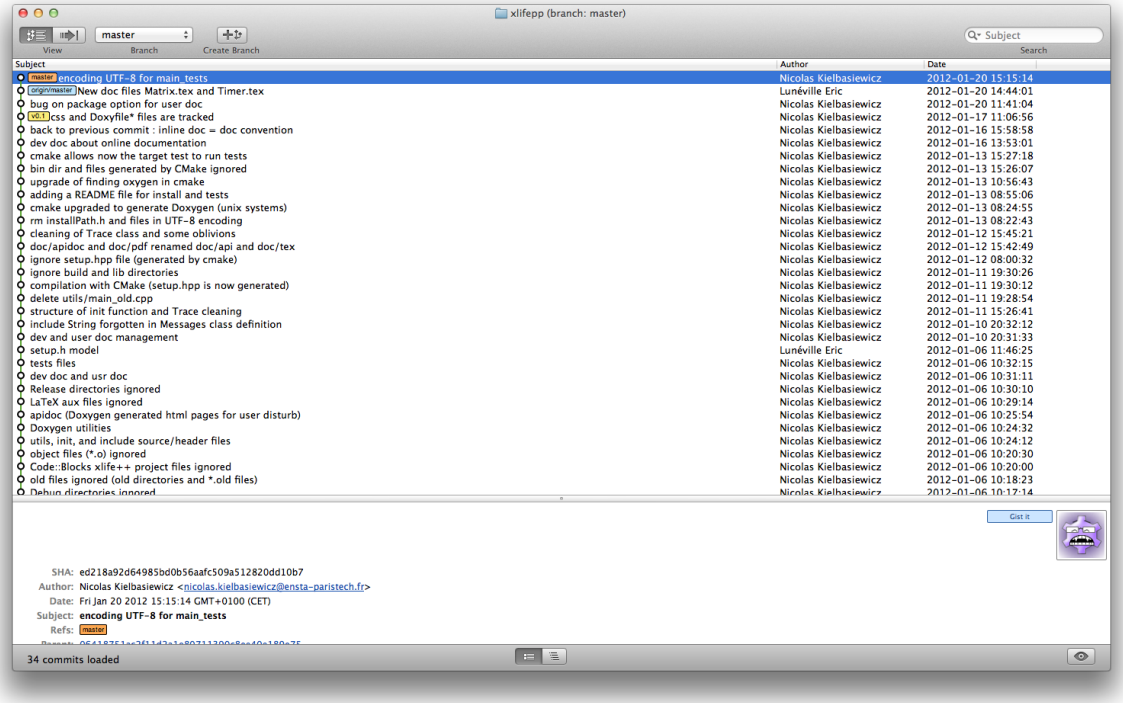


Figure 6: The gitx history user interface