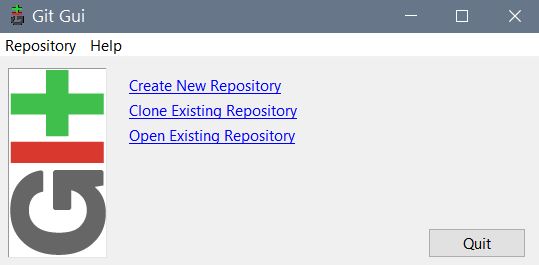
Introduction to version control using Git

Miroslav Kučera

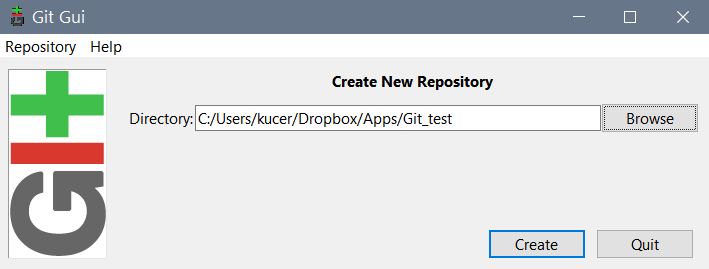
February 6, 2017

1. Git basics
   1. Creating a Git repository

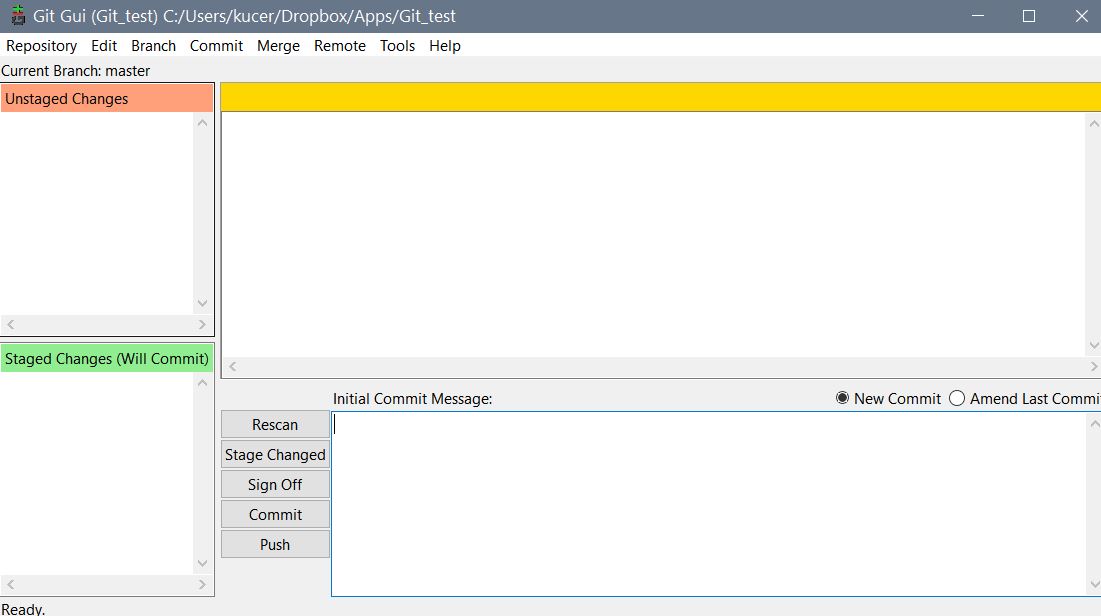
Create a folder Git\_test in a location of your choosing. Then initialize a new Git repository in Git\_test. First, open Git GUI app:



Navigate to Git\_test and select Create

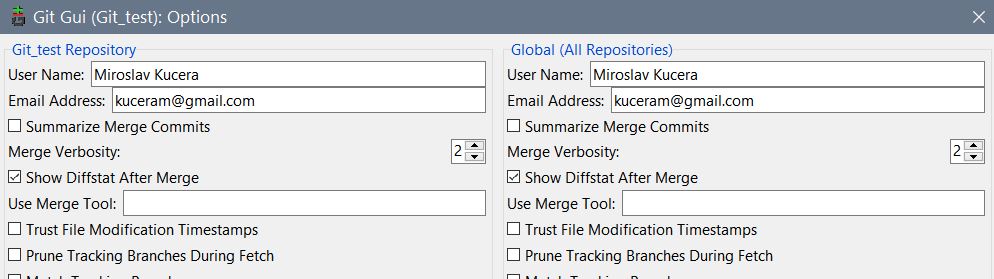


Once the new repository is initialized, the Git GUI appears. As there is no content to be tracked yet, all fields will be empty.



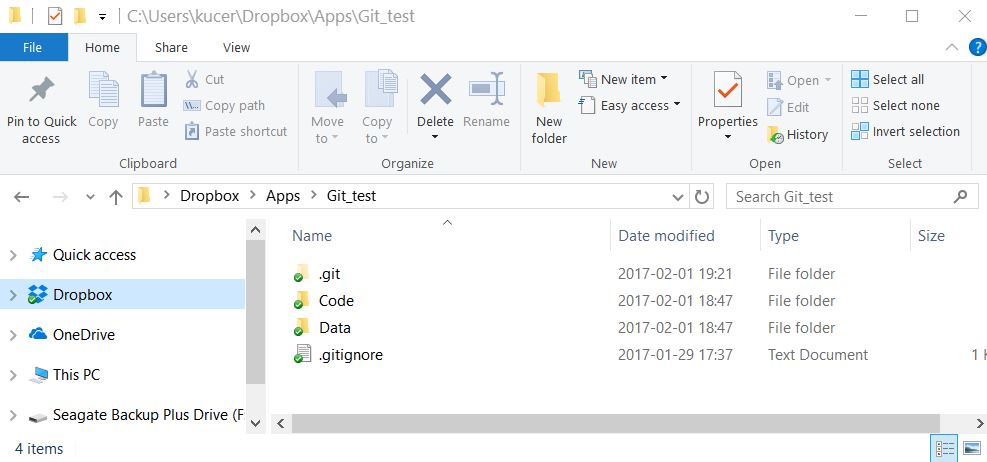
* 1. Configuring Git

If this is your first time using Git, take the opportunity to configure (some parameters of) Git. From the menu, choose Edit -> Options and fill in your user name and email both globally and locally. These will be associated with the commits you'll be making.



* 1. Basic operations: stage and commit

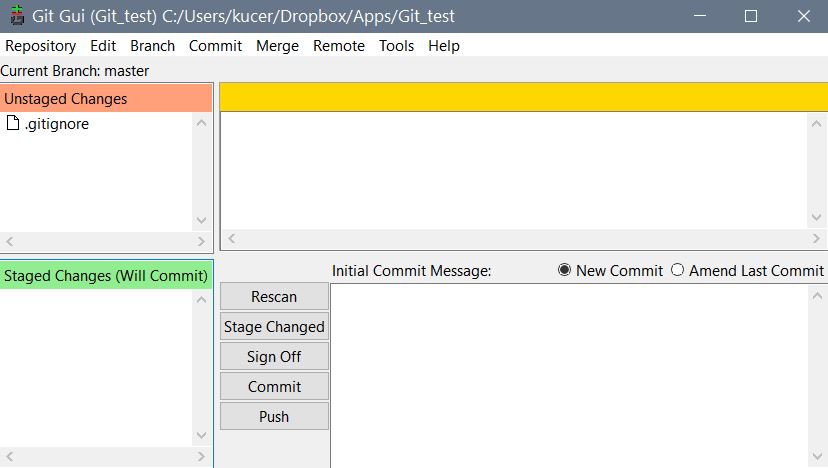
Create two (empty for now) folders inside Git\_test called Code and Data, and also create a file called .gitignore (check this footnote[[1]](#footnote-1) for instructions how to do it in Windows).

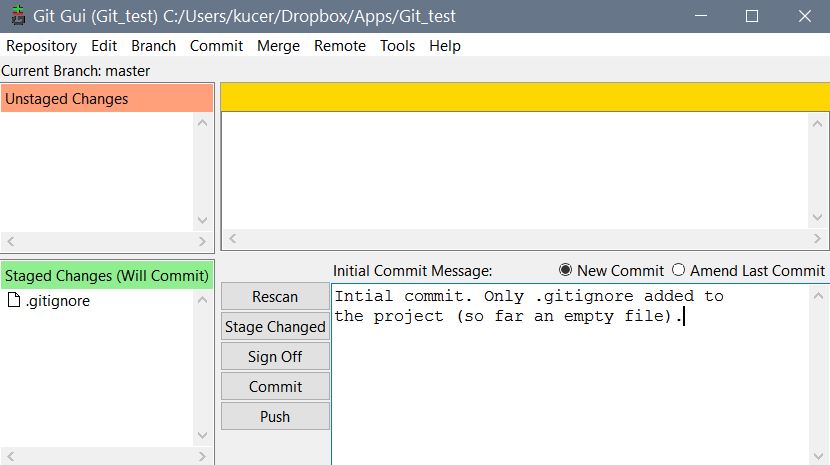


Note the presence of a hidden folder .git. This is your Git repository which stores all necessary information about the project.

Do the initial commit: only .gitignore will be ready for staging and committing because it's the only new/changed file in the repository.[[2]](#footnote-2) Code and Data are empty folders and, as such, are ignored.

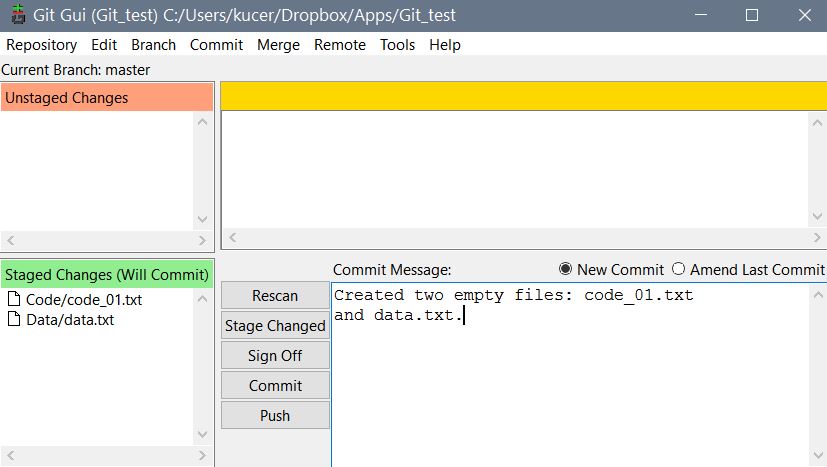
Stage Changed -> Yes (to Stage Untracked) -> Type in a comment describing succinctly the additions, changes, etc. that are to be committed into Git -> Commit:





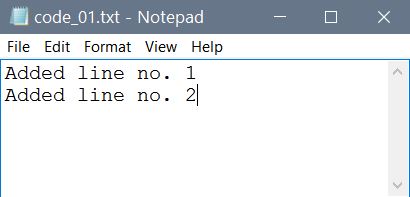
So far, there isn't much content in our Git repository that we could track. We will add some and also create some of the project's history through the following steps:

* Add an empty file called data.txt to Data folder, and an empty file called code\_01.txt to Code folder. Stage and commit these additions to Git.[[3]](#footnote-3)



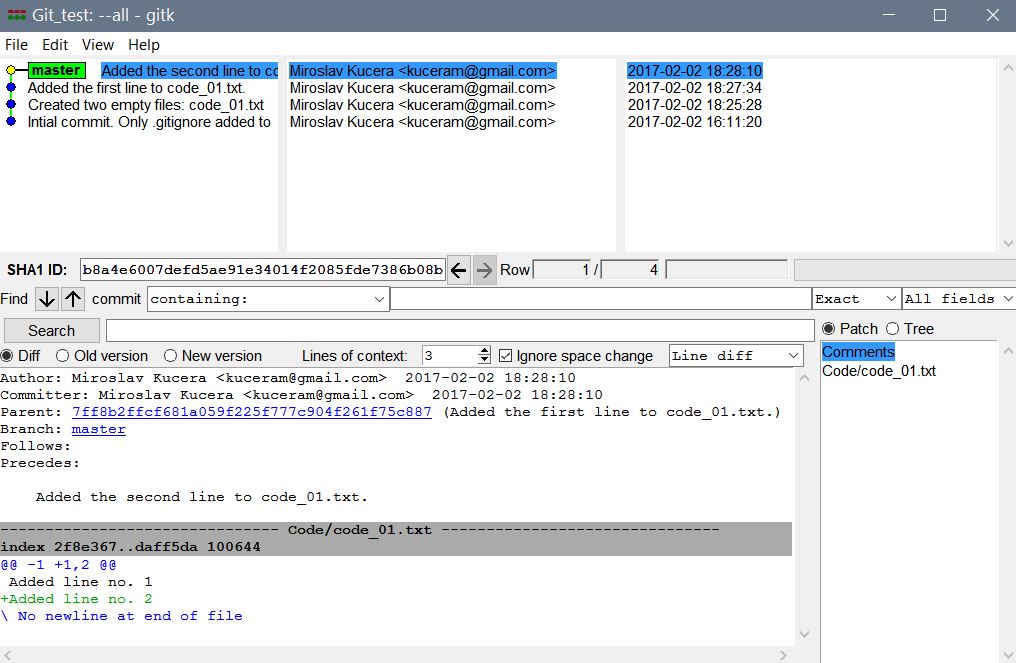
* Add the first line of code (it doesn't have to make sense) to code\_01.txt, stage and commit.
* Add the second line of code to code\_01.txt, stage and commit.

After all these changes, your data.txt will remain empty and your code\_01.txt may look like this:

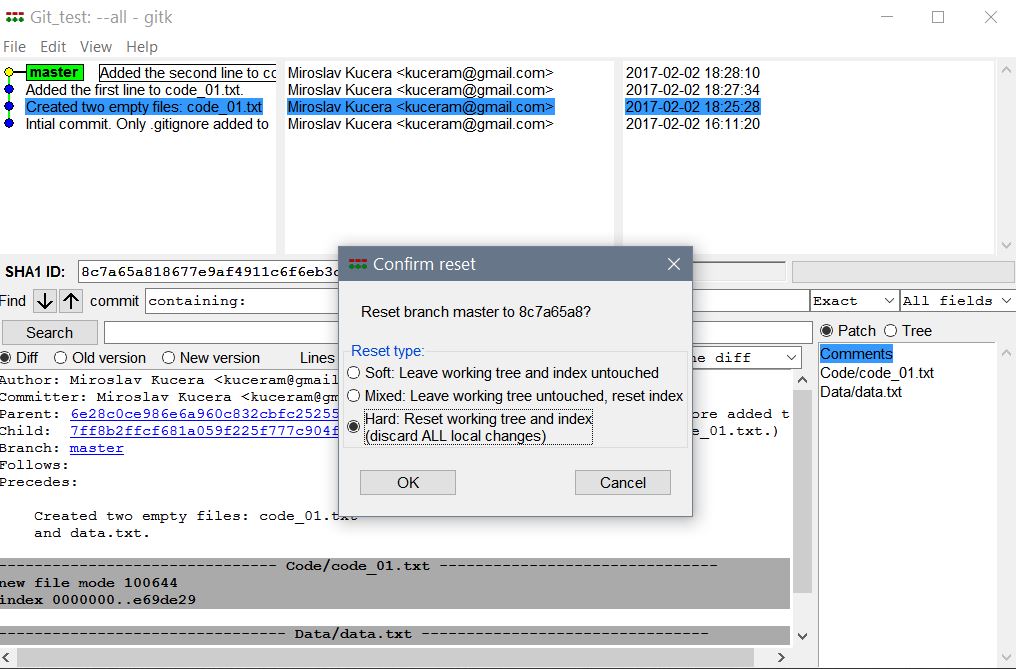


* 1. Navigating and utilizing Git's history

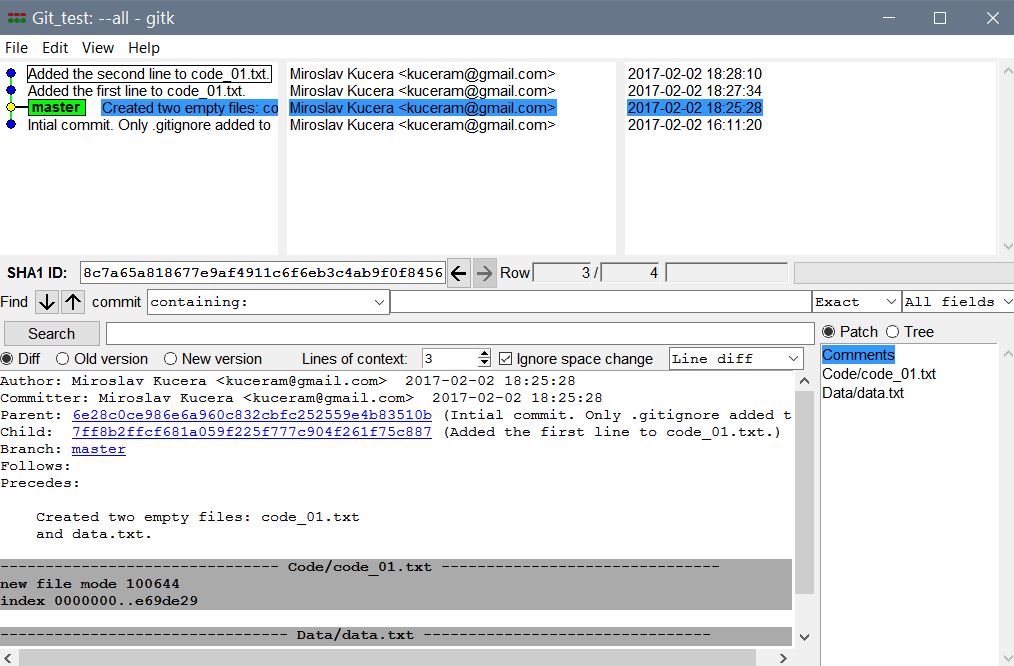
Start Git GUI, open your desired repository. From the GUI's menu choose Repository -> Visualize All Branch History. A new window appears:[[4]](#footnote-4)



The history window not only provides information but also allows a number of useful operations. One of them is the "infinite undo/redo". In other words, it is possible to revert your project to any past point (commit) that is recorded in the history simply by: Right click on the commit in the master branch that you want to go back to -> Left click to invoke a menu from which choose Reset master branch to here -> From a dialog window that appears, choose Hard reset.

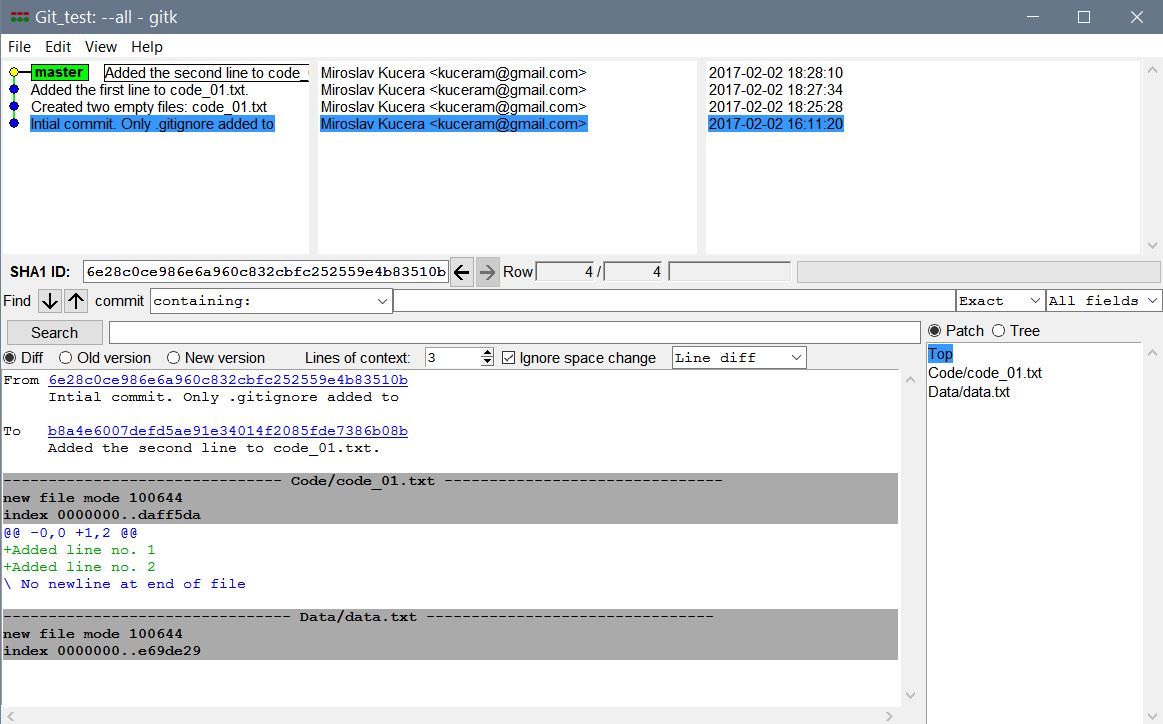


The reset will be reflected in the history "tree". Also when you now open code\_01.txt, you will see that it reverted to the content that it had at the chosen past commit (in this case, it was empty).



Indeed, you can go forward in time and reset the branch to any newer commit in exactly the same ways as you were going back in time.

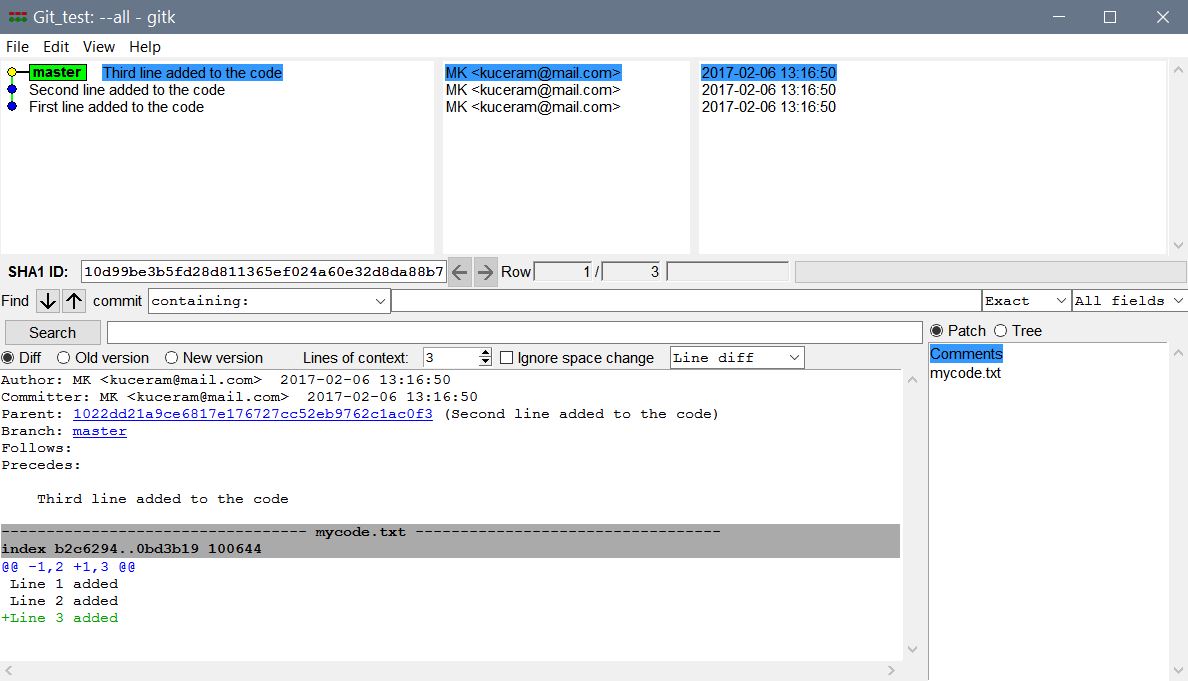
Another useful function that accessible from the history window is the comparison and/or differencing of two commits: (1) Navigate to the most recent commit, left click and select Mark this commit then (2) navigate to the very first initial commit, left click and select Diff this. The window will now show the differences between the two commits.



Unfortunately, some of these very useful functions are not very well implemented in our default Git GUI. There are third-party apps that do a better job and showing the differences in a better way.

* 1. Branching in Git

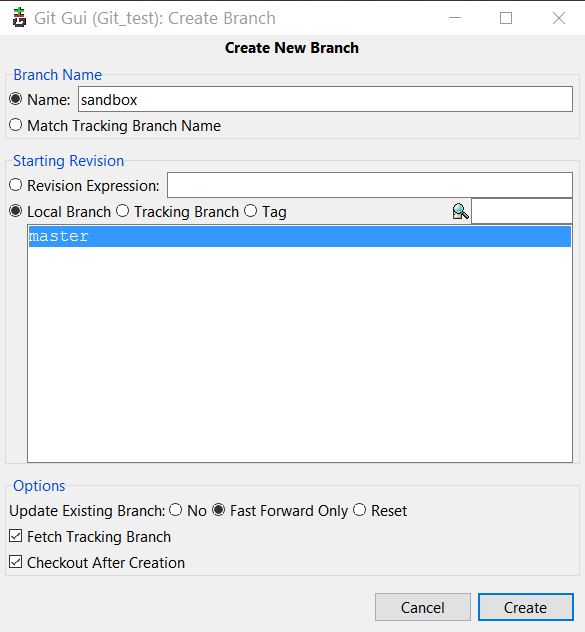
Follow the instructions in the sections above and create your own Git repo. Create a simple text file in it called, e.g., mycode.txt. Lastly, generate some Git history by making some changes to the file and committing them. The result could look like this:



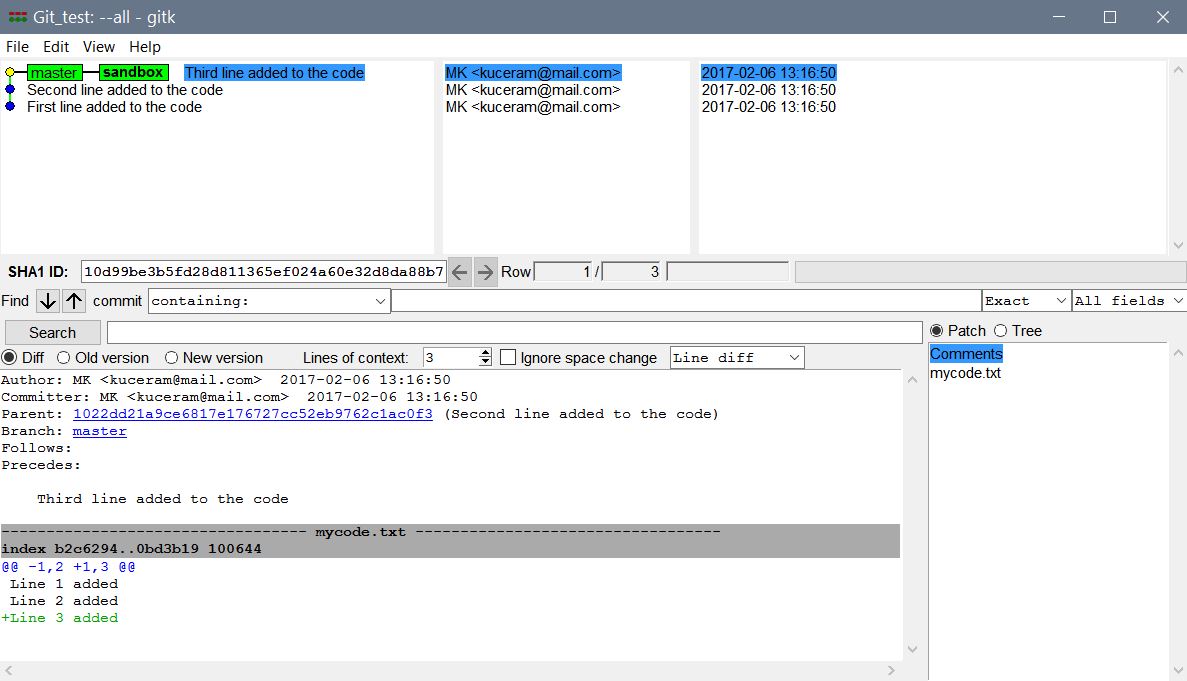
When developing a code, you can, of course, continue to work from the master branch. However, it is sometimes useful to have the ability to "branch out" from the master. There are various reasons why one would want to do that. Some teams, for example, use a branch structure as a "collaboration mechanism".

I find branching useful when you want to test some idea but don't necessarily want it immediately included in the main code. For such situations, I create a "sandbox", a special branch where I can test some half-baked idea, even "shelve" it for a while and later decide whether I want to merge it into the main code or not.

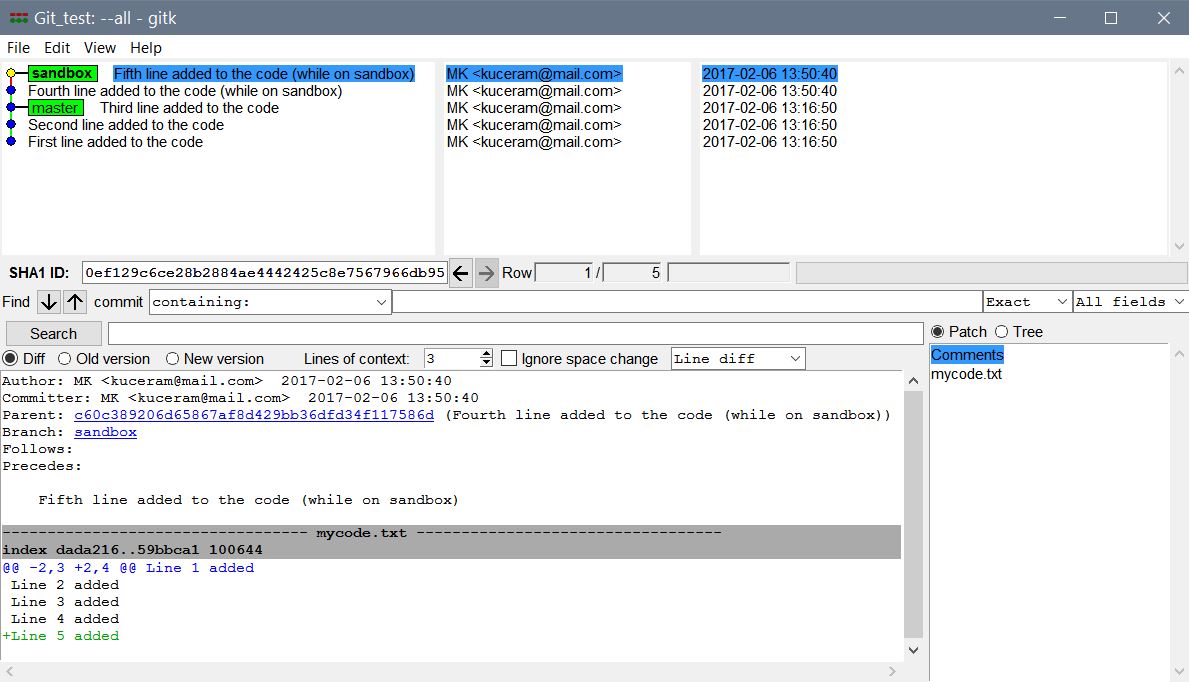
Let us now create a new branch. From Git GUI menu, choose Branch -> Create. In the dialog window type in the name of the new branch and click Create.



Refresh Git history and get



Because, by default, Git creates a new branch and immediately checks it out (i.e., moves into it), your "sandbox" is highlighted and activated. This means that any changes/additions to the code will now be recorded in the "sandbox" history and in the "master" one. Try it out yourself; make some changes to the text file and commit them; then refresh the history:

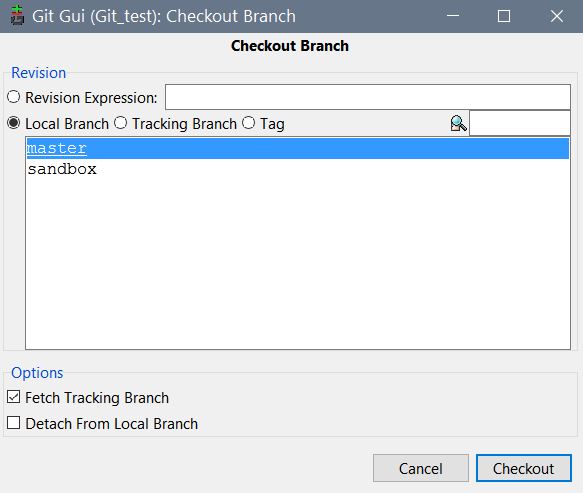


Now suppose you are done playing in the sandbox. You have three options:

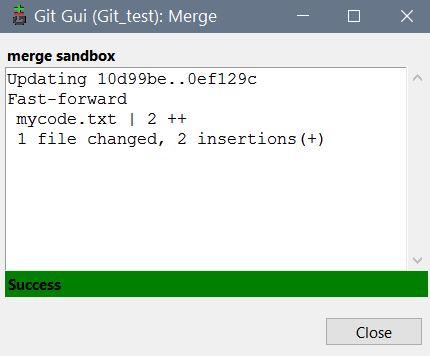
* Merge what you've done in the sandbox into the master (often can call the "trunk" to contrast it with "branch").
* Leave the sandbox be as it is, go back to (checkout) the master and continue working there without incorporating any work you have done in the sandbox. The sandbox stays in the history and you can always go back to it. The downside is that if you want to create another sandbox branch down the road, you'll have to choose a different name for it.
* Delete the sandbox without merging anything from it into the master. You loose the content of the sandbox forever.

Let us try the first option.

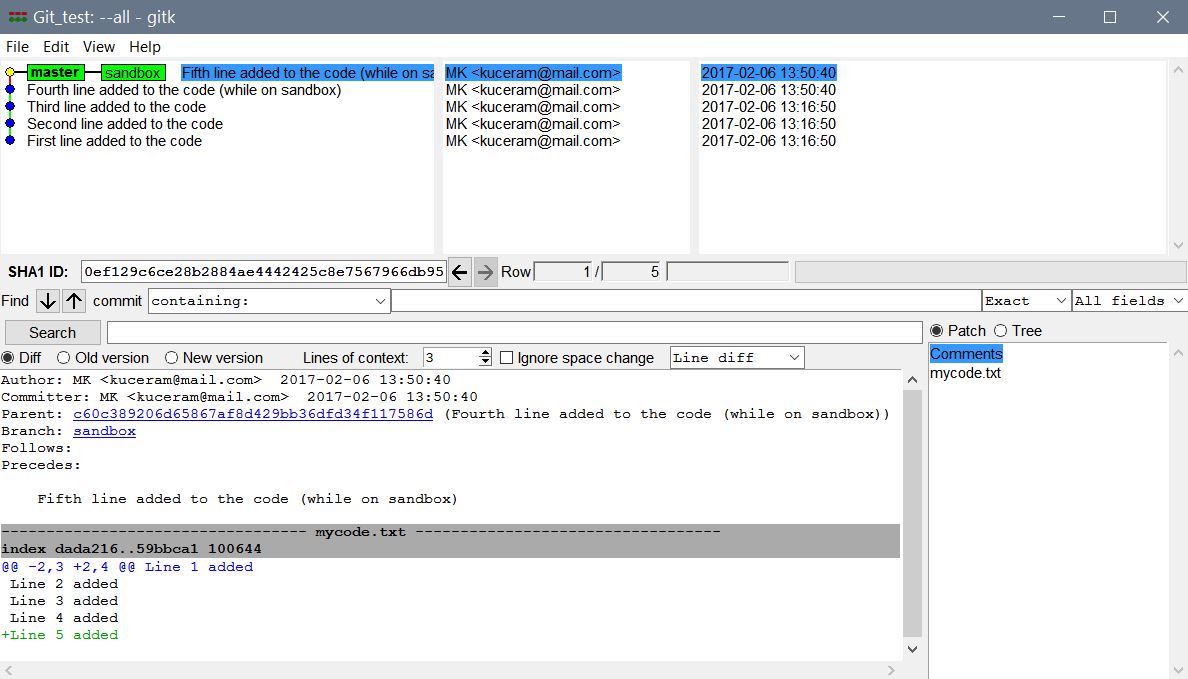
* In order to merge, checkout the master first. From Git GUI menu choose Branch -> Checkout, in the dialog select master and click Checkout.



* Once on the master, select from the menu Merge -> Local Merge. From the dialog "Merge into master" choose sandbox and click on Merge. You should get the following on the screen:



and, once you refresh the history, you should see something like this



You are now back on master (sandbox has been lumped into it) and you can verify that the content of the text file you were playing with has been updated to reflect what you did in the sandbox.

The following batch file reproduces the example from above. Ignore for now, I will explain in the class.

:: =============================================================================

:: BRANCHING IN GIT

cd "C:\Users\kucer\Dropbox\Apps\Git\Git\_test"

git init

:: Create some content in mycode.txt and commit the changes to get some history

echo Line 1 added > mycode.txt

git add .

git commit -m "First line added to the code"

echo Line 2 added >> mycode.txt

git add .

git commit -m "Second line added to the code"

echo Line 3 added >> mycode.txt

git add .

git commit -m "Third line added to the code"

:: Create a "sandbox" branch for the sake of testing some code that

:: may or may not be merged to the main code later

git branch sandbox

git checkout sandbox

echo Line 4 added >> mycode.txt

git add .

git commit -m "Fourth line added to the code (while on sandbox)"

echo Line 5 added >> mycode.txt

git add .

git commit -m "Fifth line added to the code (while on sandbox)"

:: I like the code I've been trying in the sandbox, so let's merge

:: it into the master branch

git checkout master

git merge --no-ff sandbox -m "Merging sandbox into master"

1. Using Git in a Team Setting

The following batch file generates an example to illustrate collaboration using Git. Ignore for now, I will explain in the class.

:: =============================================================================

:: COLLABORATION USING GIT

::

:: This is a Microsoft batch file containing a sequence of commands that

:: simulate a collaborative workflow when developing a computer code and

:: tracking the development using a Git command-line interface (e.g., Git Bash).

::

:: This batch file is also executable in its entirety:

::

:: (a) In Windows Explorer, either double-click on the file,

:: or right-click and choose "Run".

::

:: (b) In Command Prompt, navigate to the directory with this file (use "cd")

:: and then execute: git\_cmd.bat

::

:: (c) In Git Bash, navigate to the directory with this file (use "cd")

:: and then execute: ./git\_cmd.bat

::

:: Note: The script appears to be case-insensitive

::

:: Microsoft OS commands (good old MS-DOS :)

:: mkdir = make a new directory (i.e., create a new directory)

:: cd = change directory (move from the current directory to another one)

::

:: Git commands

:: (1)

cd "C:\Users\kucer\Dropbox\Apps\Git\Git\_test"

git init --bare Testproject.git

:: (2)

mkdir "C:\Users\kucer\Dropbox\Apps\Git\Git\_test\Developer\_A"

cd "C:\Users\kucer\Dropbox\Apps\Git\Git\_test\Developer\_A"

git clone "C:\Users\kucer\Dropbox\Apps\Git\Git\_test\Testproject.git"

mkdir "C:\Users\kucer\Dropbox\Apps\Git\Git\_test\Developer\_B"

cd "C:\Users\kucer\Dropbox\Apps\Git\Git\_test\Developer\_B"

git clone "C:\Users\kucer\Dropbox\Apps\Git\Git\_test\Testproject.git"

:: (3)

cd ..\Developer\_A\Testproject

echo 'Nothing yet' > readme.txt

:: From now on continue from Git GUI

1. Create an empty text file, e.g. using Notepad, and save it as "gitignore.txt". Rename the file to ".gitignore." (note the initial and closing periods). Once you click OK, the file will become ".gitignore" and you will still be able to edit as any other simple text file. There is also an alternative way that uses Windows command line interface if you want to dig deeper. [↑](#footnote-ref-1)
2. See <add a reference here> for details on the content of .gitignore. [↑](#footnote-ref-2)
3. Note that to have any additions/changes to appear in Git GUI, you need to hit Rescan first. This will identify all files that have changed since the last commit in the Unstaged Changes window. [↑](#footnote-ref-3)
4. The sections and pieces of information in the Git history window will be explained in the class. [↑](#footnote-ref-4)