Using “git” and github

Wyatt Newman
August, 2016

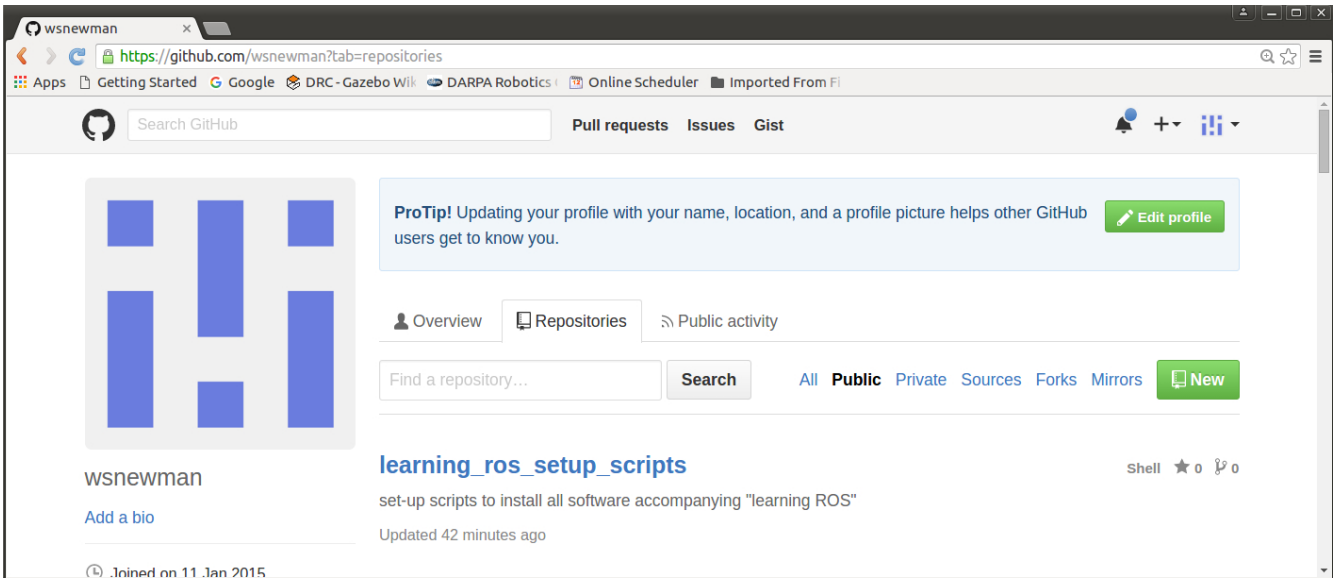
Organizing projects and keeping track of code revisions can be done effectively with the tool “git” (see http://git-scm.com/book/en/Getting-Started) together with a hosting service, such as “github” (https://github.com) or “bitbucket” (https://bitbucket.org). “Git” is a valuable tool when working individually on projects, and such a tool is essential when working collaboratively on software-development projects.

These notes will focus on using github. To use github, you will need an account, which is free. (For the academic community, using one's academic e-mail to register is preferred, as this offers more attractive features than the generic free account).

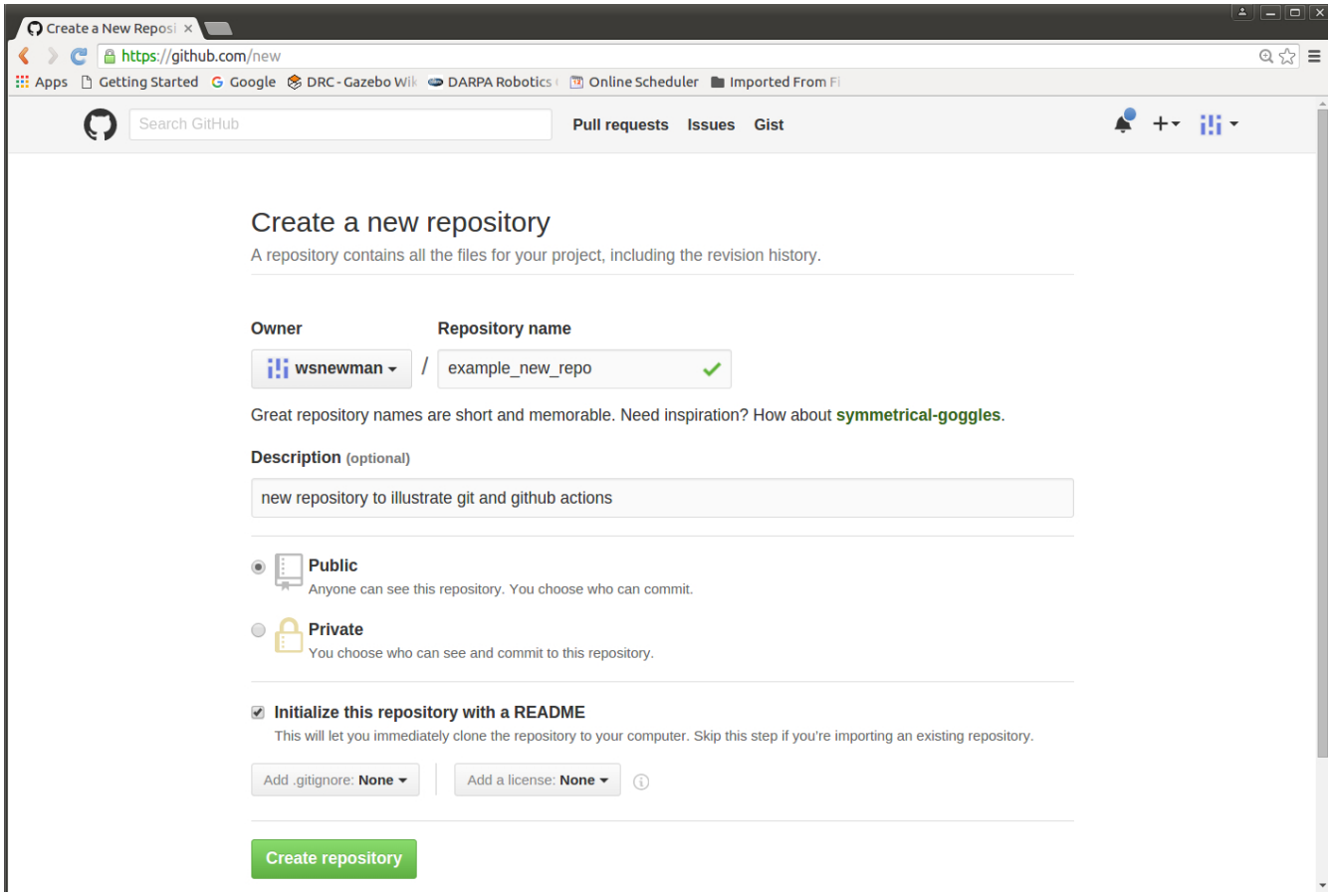
The reader is encouraged to review the on-line github guides at https://guides.github.com/, which are more extensive than the present introduction.

Creating a new repository on Github:
Using a browser, navigate to https://github.com and login. When your account is new, you will have no “repositories” (projects) under your account. If you are starting a new project, you will want to create a new repository. (If you are joining an existing project, you will want to “fork” an existing repository, which is covered later).

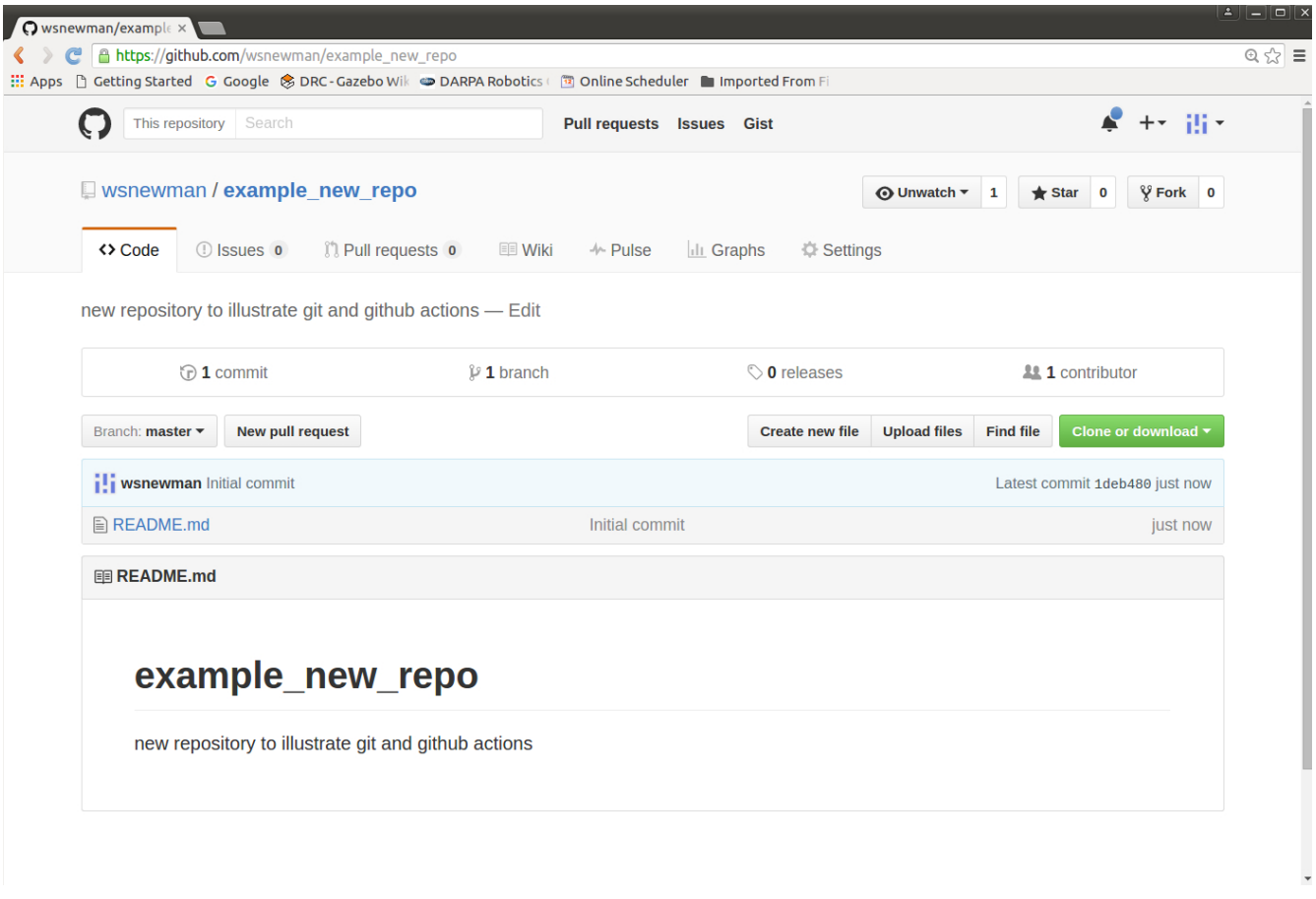
To create a repository, navigate to the “repositories” tab and click the green “New” button. (In the screenshot below, there are existing repositories already, e.g. “learning_ros_scripts”, but creating a new repository is the same process when creating your first repository).



After clicking the “New” button, you will be presented with a form to fill you regarging your new repository. You should fill in the name of your new repository (“example_new_repo”, in this example), optionally add a description, select if the repository is to be public or private, and check “Initialize this repository with a README.” (note: if you are using a free account, you may not have the option of making your repository private). With the form completed, click “Create Repository.”



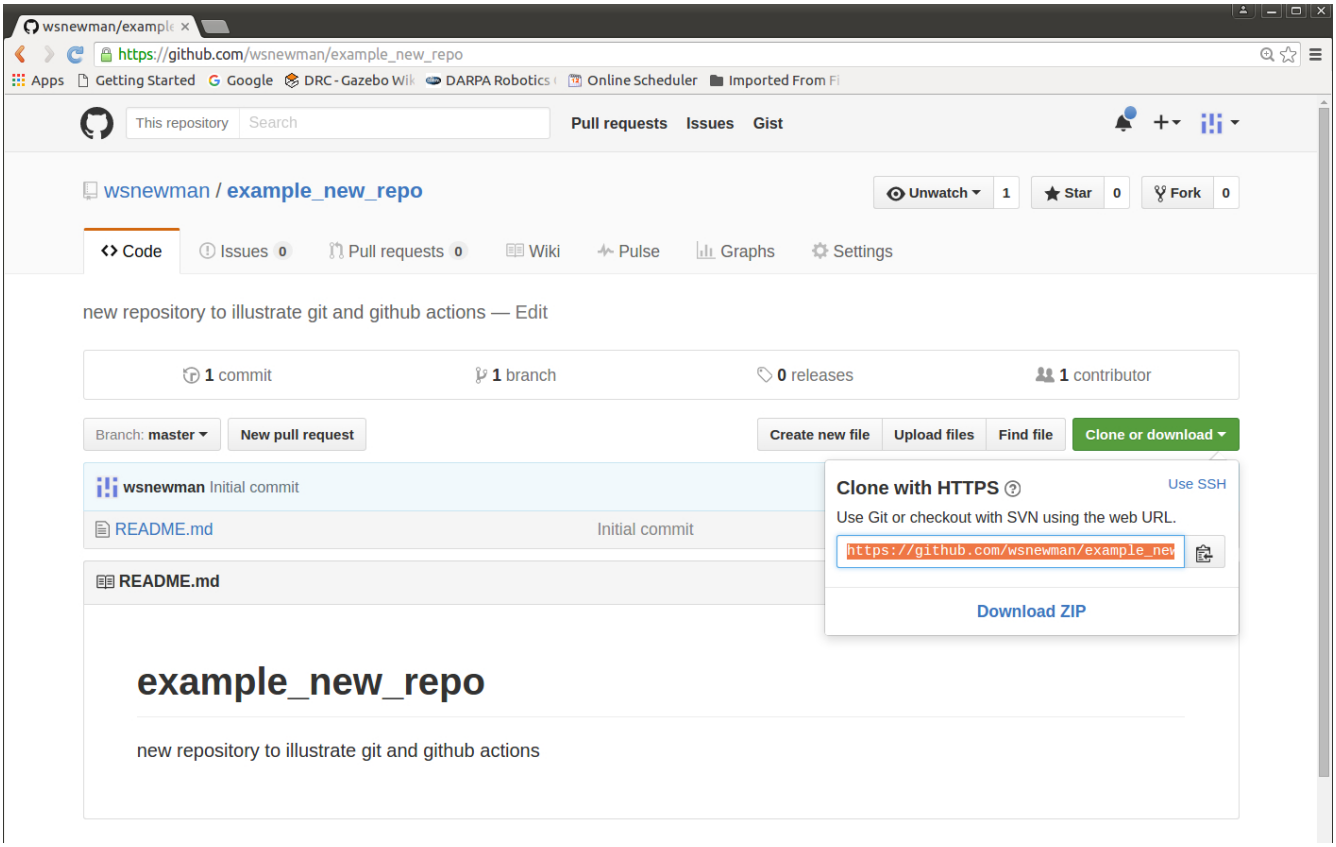
The result of creating this repository will be a screen that appears as below:



This new repository has only a single file, “README.md.” You should edit this file to provide useful explanations for the new repository.

Note that, at this point, the new repository only exists in the “cloud” (hosted by github). Nothing on your computer's local disk has changed. Repositories in the cloud can be copied to local disk by “cloning.”

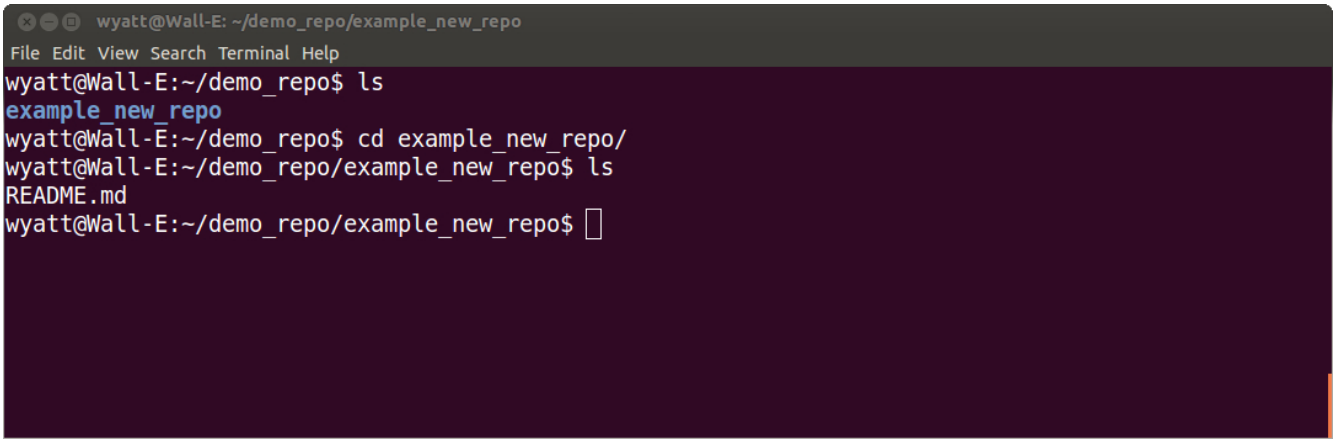
Cloning a Repository from github:
“Cloning” is the operation of copying all contents of a repository stored on-line onto a computer's local disk. To clone a repository from github, use a browser to navigate to the repository of interest. Note the green button “Clone or download.” By clicking this button, a URL will be presented, which for the present example is: “https://github.com/wsnewman/example_new_repo.git.” Copy this URL to your clipboard buffer (either highlight the text and right-click/copy or, or click the “copy” icon).



Next, open up a terminal window on your computer, navigate to where you would like the repository to be cloned, and enter the command:
git clone https://github.com/wsnewman/example_new_repo.git
(you can paste the long URL that you copied from github).

An example result of performing this action is shown below:

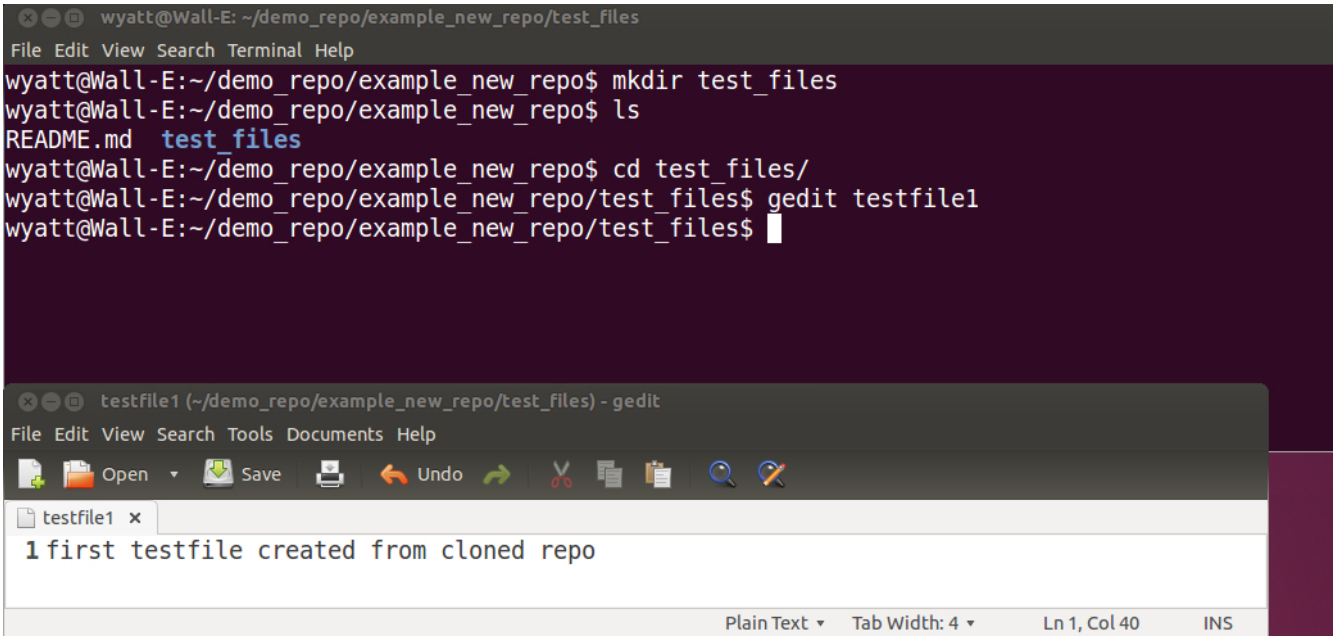
After performing this action, a new directory is present: “example_new_repo”. As shown below, navigating to this directory shows that it contains a file called “README.md” (which has been copied from the repository on github).



The new directory “example_new_repo” is more than a generic directory—it is a repository. It is a repository by virtue of the fact that it contains a hidden directory called “.git”. This hidden directory contains information about the organization of files in this repository. This directory is auto-generated, and it should not be edited by the user.

At this point, there are two identical copies of the repository “example_new_repo”--one in the cloud, and one on local disk. When working with code development, one edits files in the repository on the local disk, then “pushes” the local changes to the github-hosted repository to get these two versions into sync.

As an example, we can create a new directory and a new file in the local repository as follows:



In the above, the directory “test_files” was created, and a new text file “testfile1” was created using “gedit.” The local repository now contains items that are not present in the github-hosted version. To reconcile these, one “commits” changes then “pushes” them back to the on-line source. These operations can be performed with text commands typed to a terminal. Alternatively, a graphical tool “git gui” can be more intuitive and helpful. If it is not already installed, the git gui can be installed with:

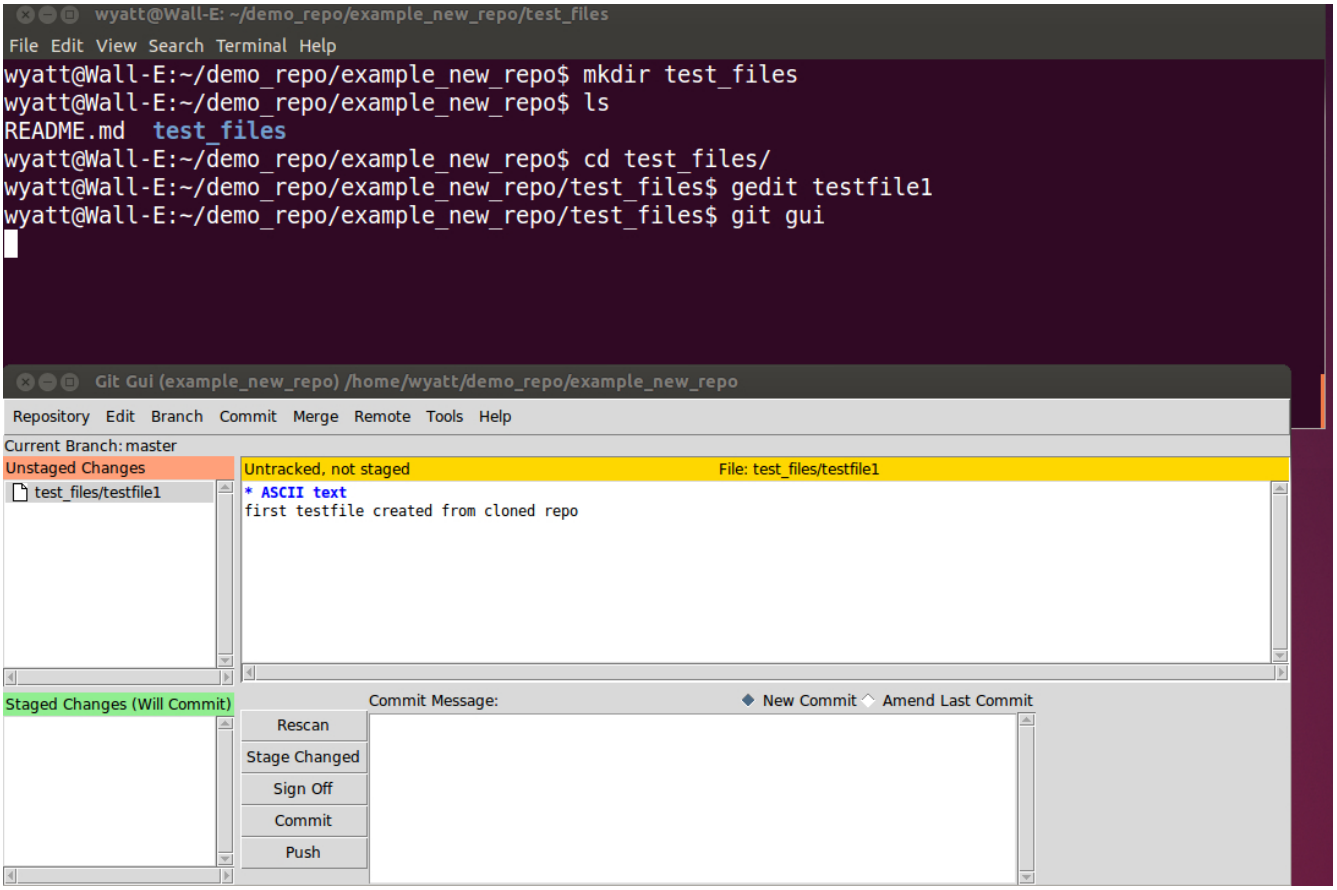
sudo apt-get --yes --force-yes install git-gui

Using git-gui is described in the following. Although it may appear tedious, at first, with some practice it becomes quick and natural and helps the user avoid making mistakes or oversights.

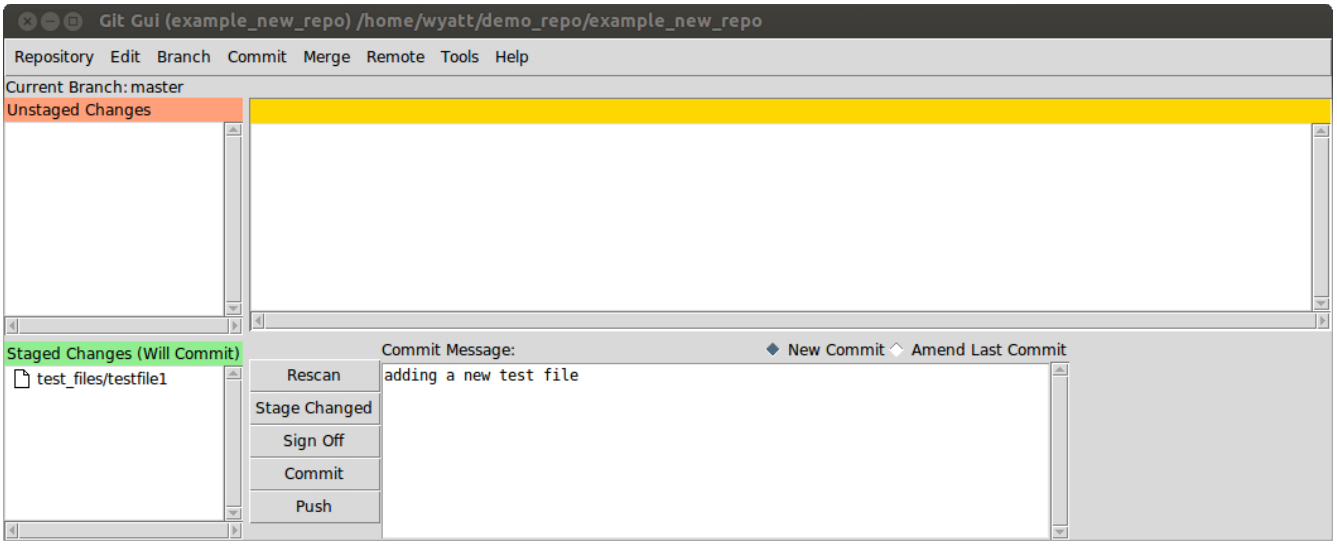
Git-gui is started from a terminal. Within this terminal, first navigate to the repository (cd to the directory that was cloned), then enter:

git gui

A window will appear, as below:

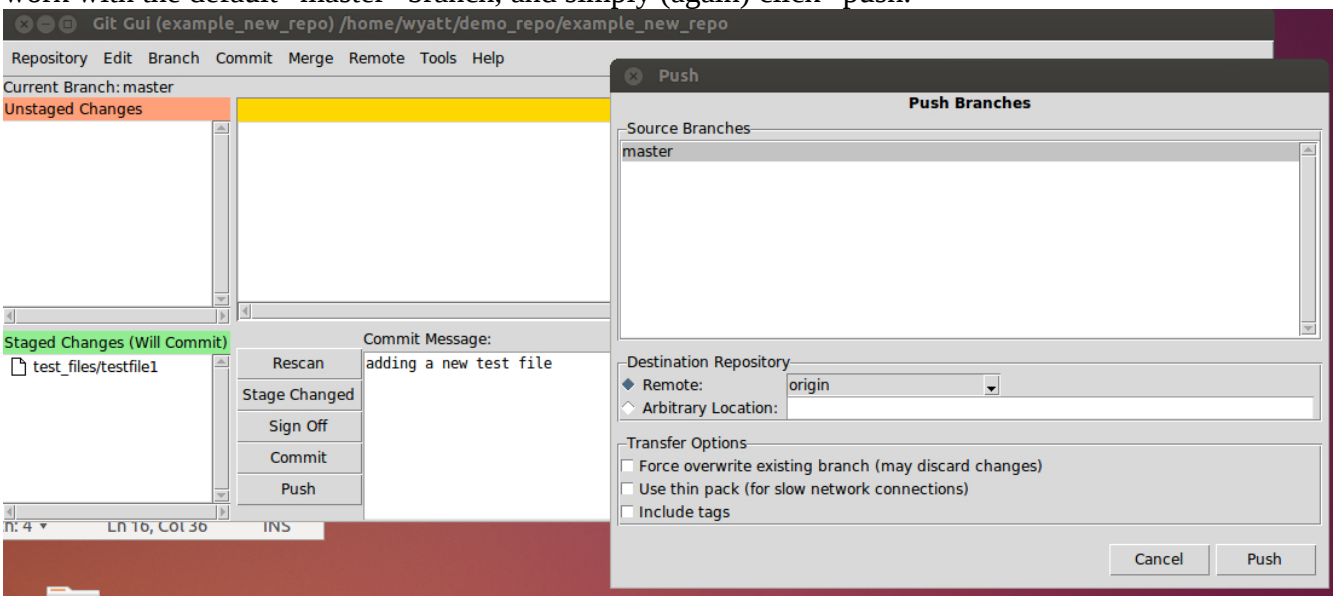


The upper-left panel shows a list of files that have been edited locally but which are not updated with the on-line repository. By clicking on the name of one of these files (there is only one, in the present example), all changes to this file are shown in the upper-right window. One can review these changes and, if warranted, "commit" these changes to be reconciled with the on-line repository.

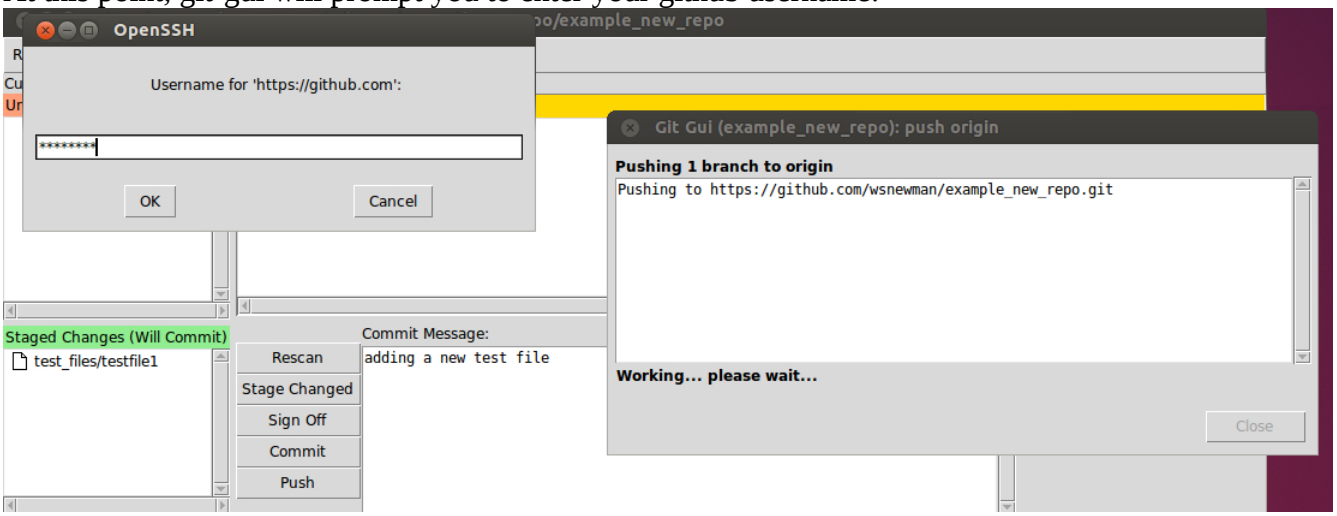


By clicking on the icon next to the (uncommitted) file name, the file disappears from the "unstaged changes" window and reappears in the "stated changes" window. At this point, the user must enter a "commit message"--a relevant comment regarding the changes that have been made. In this instance, the commit comment is "adding a new test file." After entering the message, press the "commit"

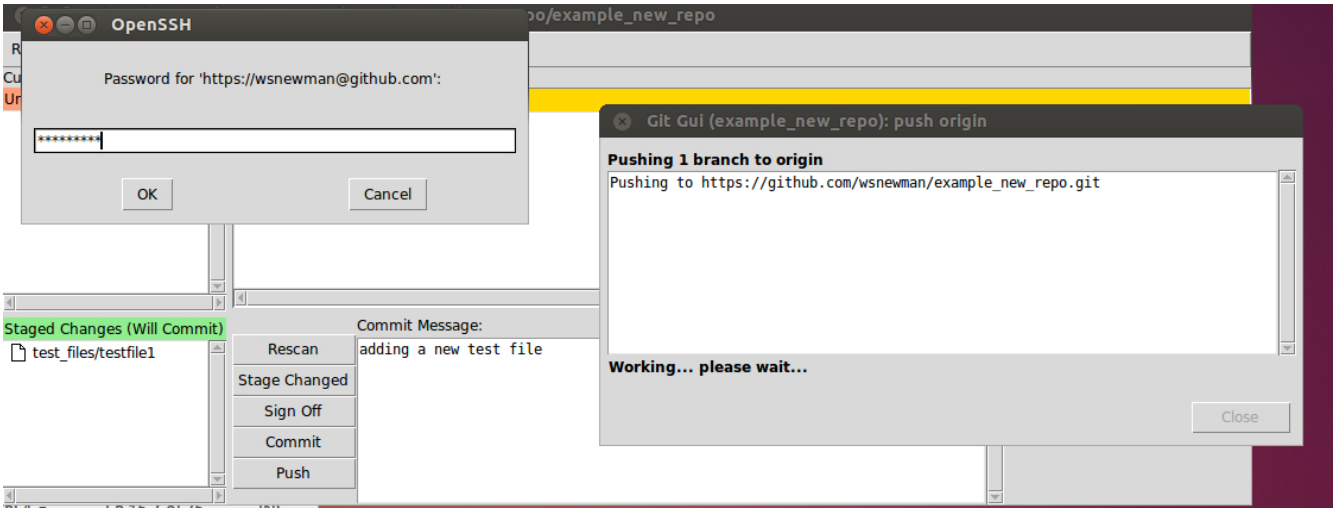
button. At this point, the changes are “staged” to be integrated with the on-line repository. To do so, press the “push” button, and git-gui will respond as below. If you are using “branches” (a convenient option in git), you may choose to which branch you want to push. For now, you can ignore this option, work with the default “master” branch, and simply (again) click “push.”



At this point, git-gui will prompt you to enter your github username:

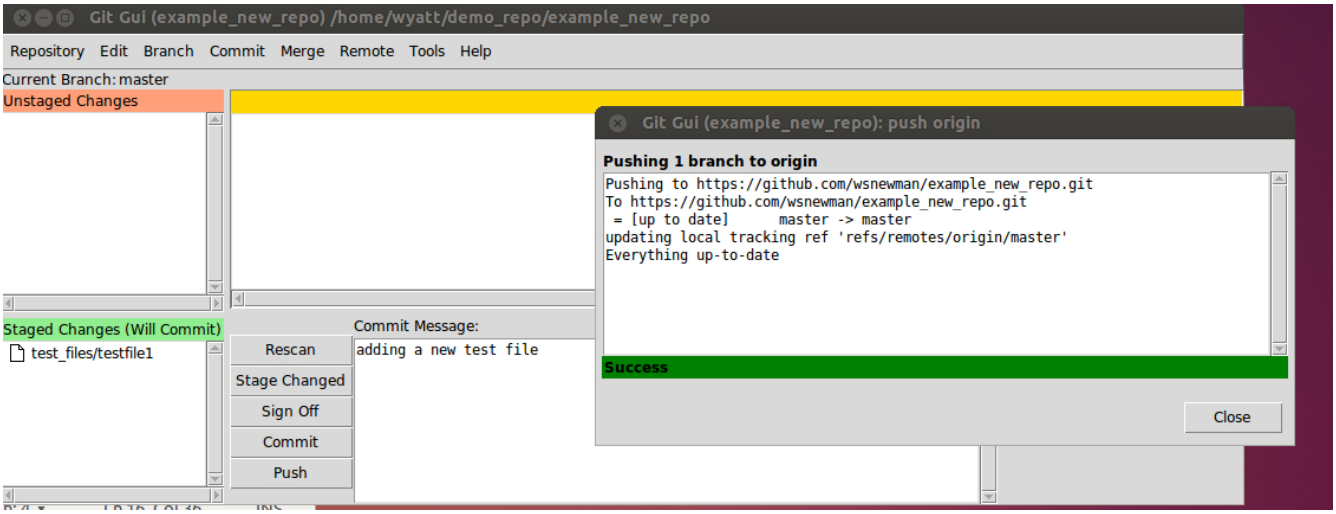


and your github password:

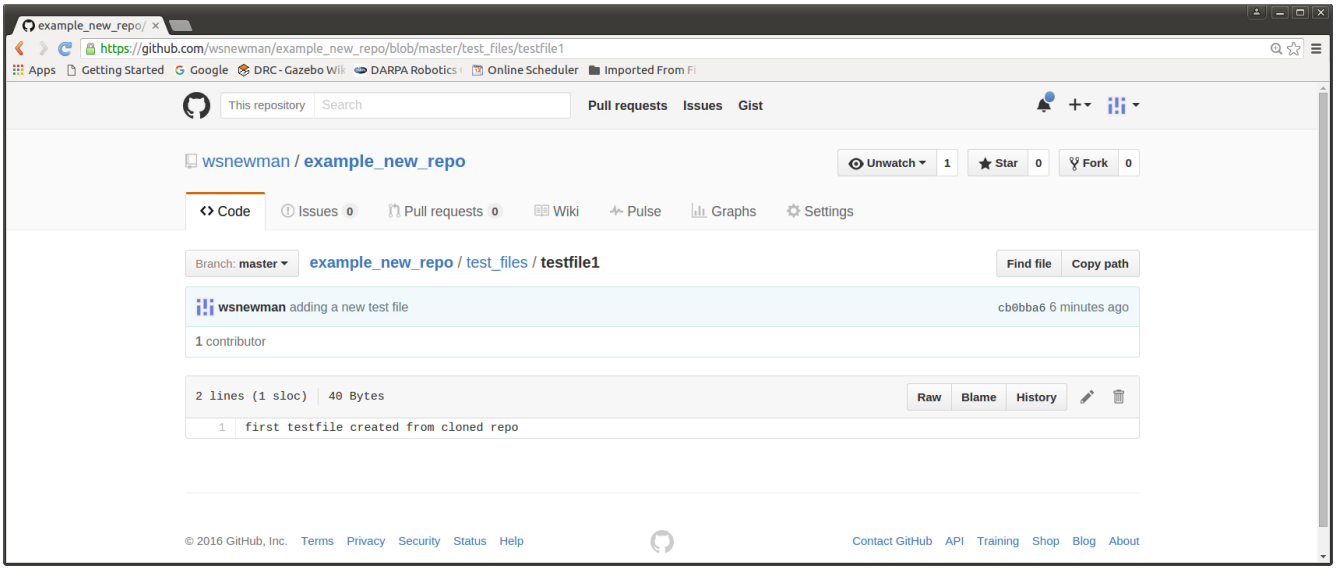


(Note: one can optionally set up keys such that your userid and password are entered automatically, which is handy when performing frequent “push” updates).

The result of this “push” operation should be a “Success” message, as shown below.



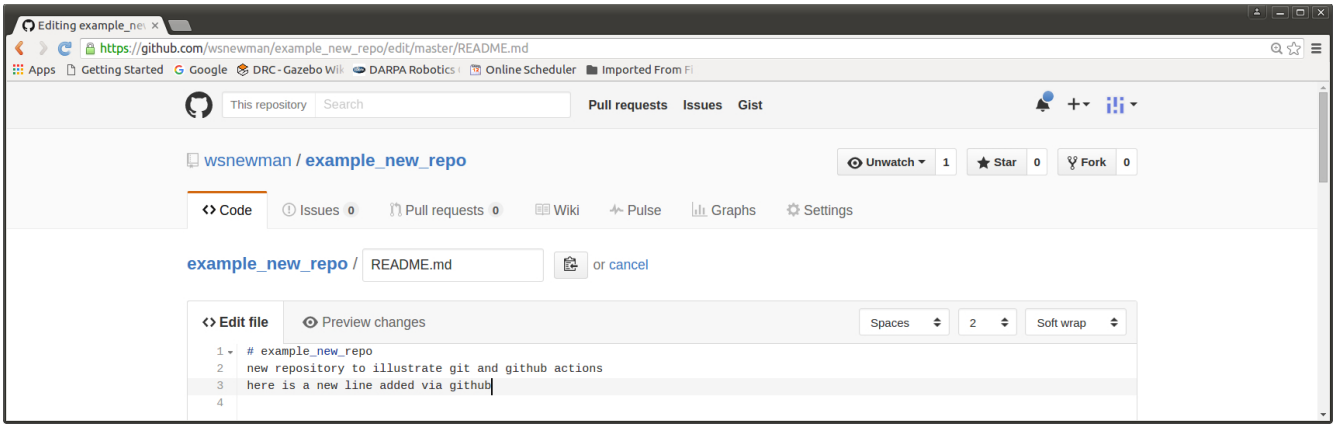
At this point, we can re-visit the github repository via a browser. The on-line repository version will be updated, as shown below. In this view, we can see that the repository now contains an additional directory and a new file, the contents of which are “first testfile created from cloned repo.”



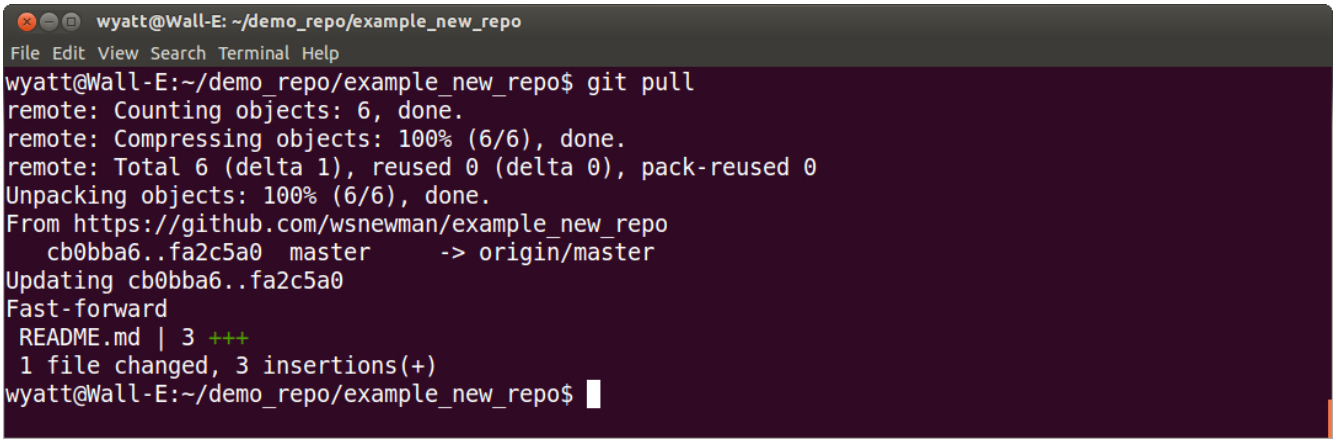
Following the above steps, one can continue to make local modifications and synchronize them with the on-line version. This already have the benefit of providing a back-up of your project.

Another immediate advantage is that one can continue to make changes from different computers and keep all versions in sync. When starting work on a different computer, simply clone the repository, as described above, make any desired changes, then "push" the changes back to the github repository. The on-line version should be considered the "master" version, and changes to the master version should get reflected to local copies before making new changes.

When resuming work within a local repository, one option is to delete the current repository, then re-clone it from the on-line master. However, a faster way to resume work is to "pull" any master-repository changes into the local repository. To illustrate this, the on-line repository was edited via a browser, adding a line to the README file, as follows:



The on-line repository version is now newer than the local version. To get the local version to import the changes existing in the on-line version, enter the command “git pull”, as shown below:



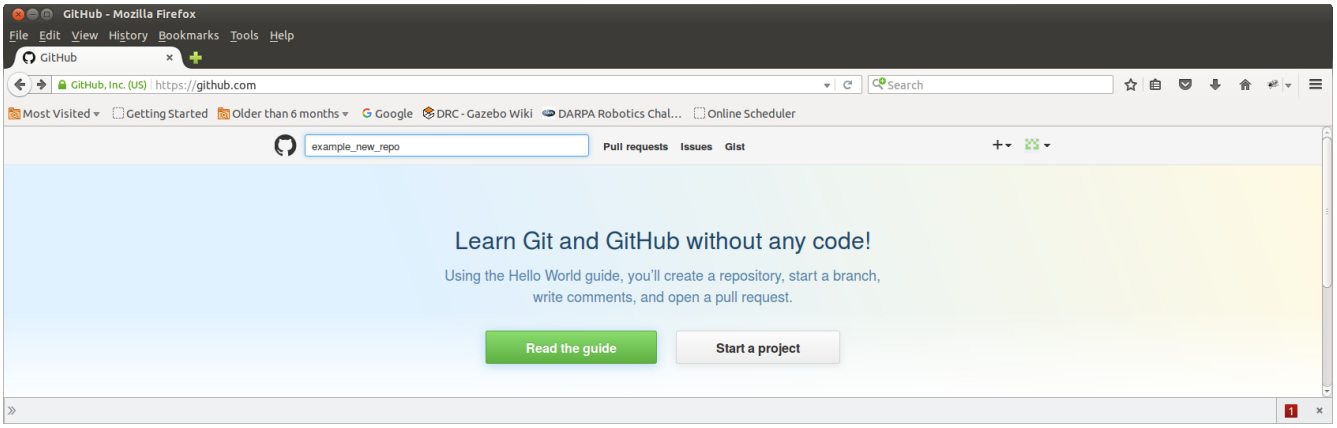
The result shows that the local copy of the file README.md was changed and is now updated to be in sync with the on-line repository.

To summarize so far, having created a project, the normal work-flow would proceed as follows. If you are working from a new machine, “clone” a copy of the repository, perform new work, then “push” the changes (e.g. using git-gui) to github before signing off. It is important that any changes you desire to keep should be pushed back to github, else your work may be lost and/or you will have version conflicts in the future.

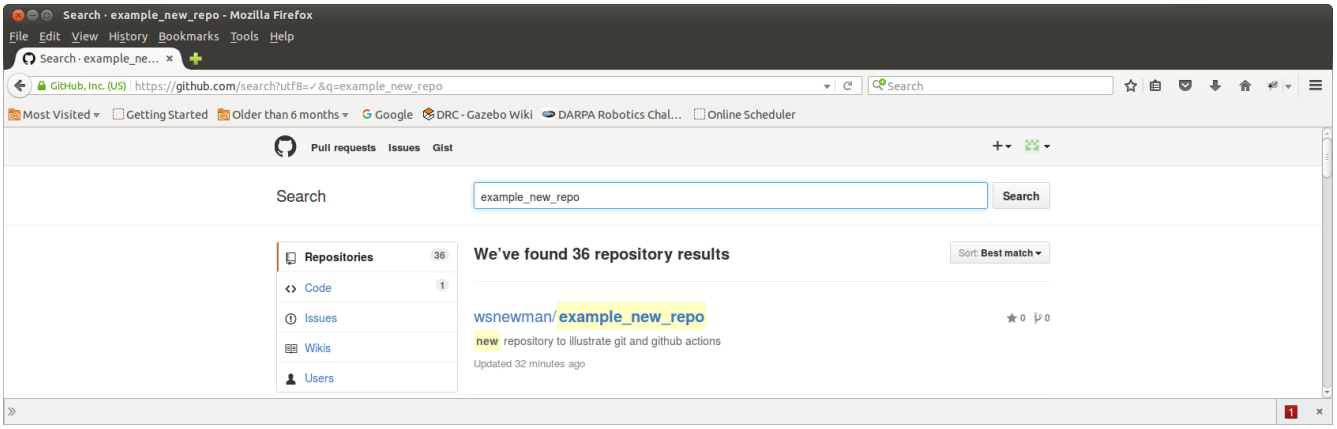
If working from a machine that already has a clone of your repository, start by commanding “git pull” from within this repository. Make your desired changes/additions, then “push” your results back to bitbucket.

Using github as described above has some benefits, such as synchronizing work across multiple machines, maintaining an on-line back-up of your projects, and maintaining a record of changes (with associated comments). This is worthwhile in itself—but the greater value of using “git” is realized when collaborating within a team. Organizing collaboration requires some new commands, including “forking”, “pull requests” and “merging.”

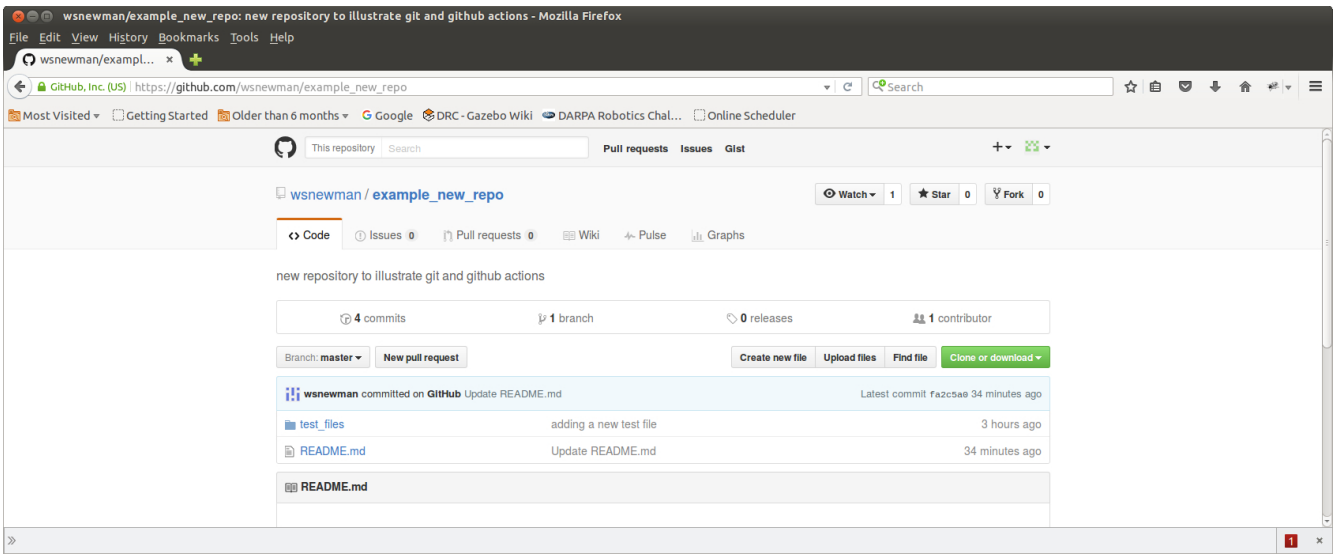
Forking a repository:
To become a collaborator on an existing project, or to create your own project based on an existing project, one can “fork” a repository. To illustrate this, the figure below shows a view of a new github account under a separate login ID. There are no repositories present yet in this new account. Within the window presented, one can search for existing repositories via the “search github” window.



Entering “example_new_repo” in the search window shows multiple results, the first of which is our desired example.

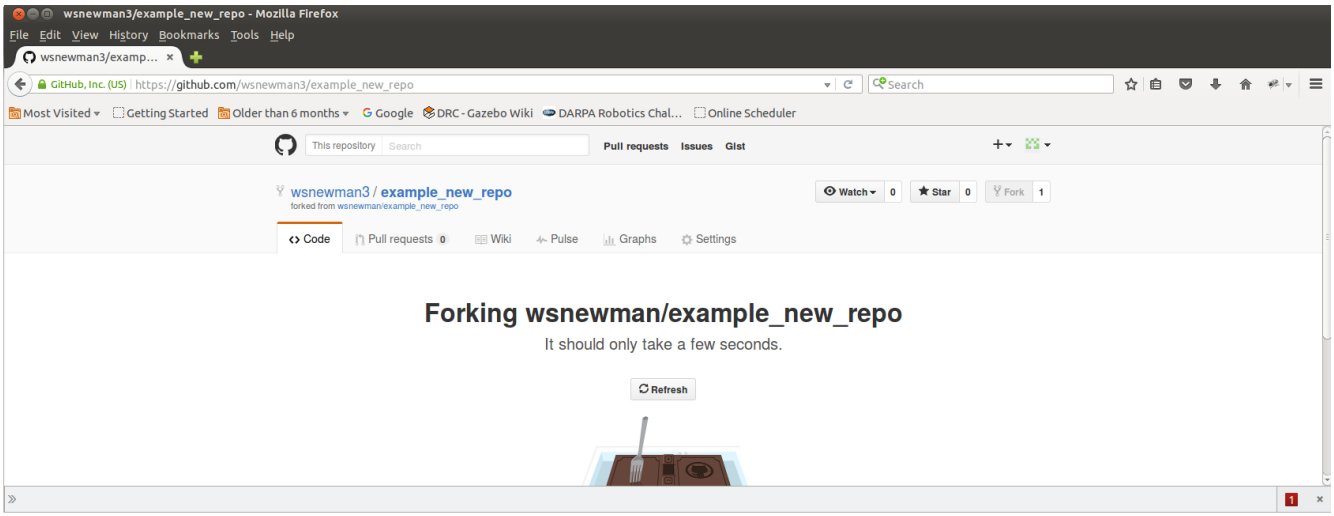


From this view, click on the desired repository to bring up the following view:

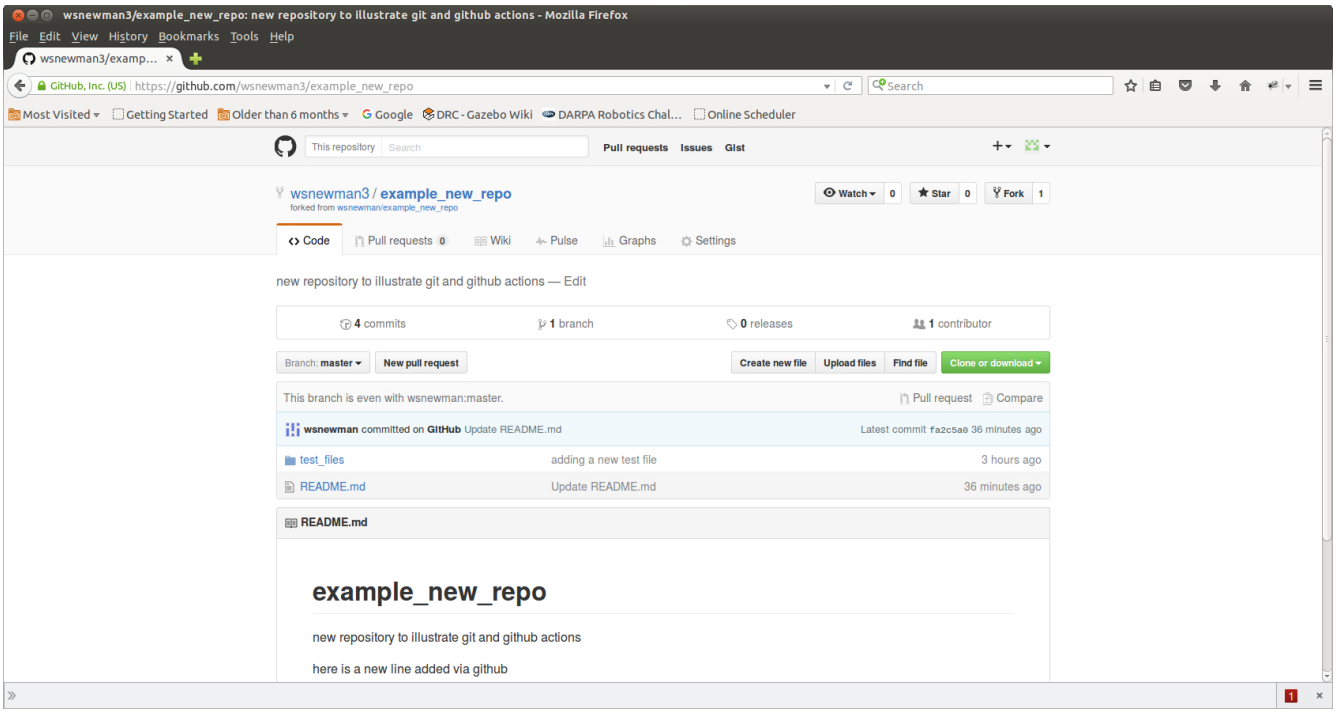


This is the repository we desire, but it is not owned by the current user. As the current user, we can make a “fork” of this repository by clicking the button labelled “fork.”

Github will respond with a display showing that the fork operation is in progress:



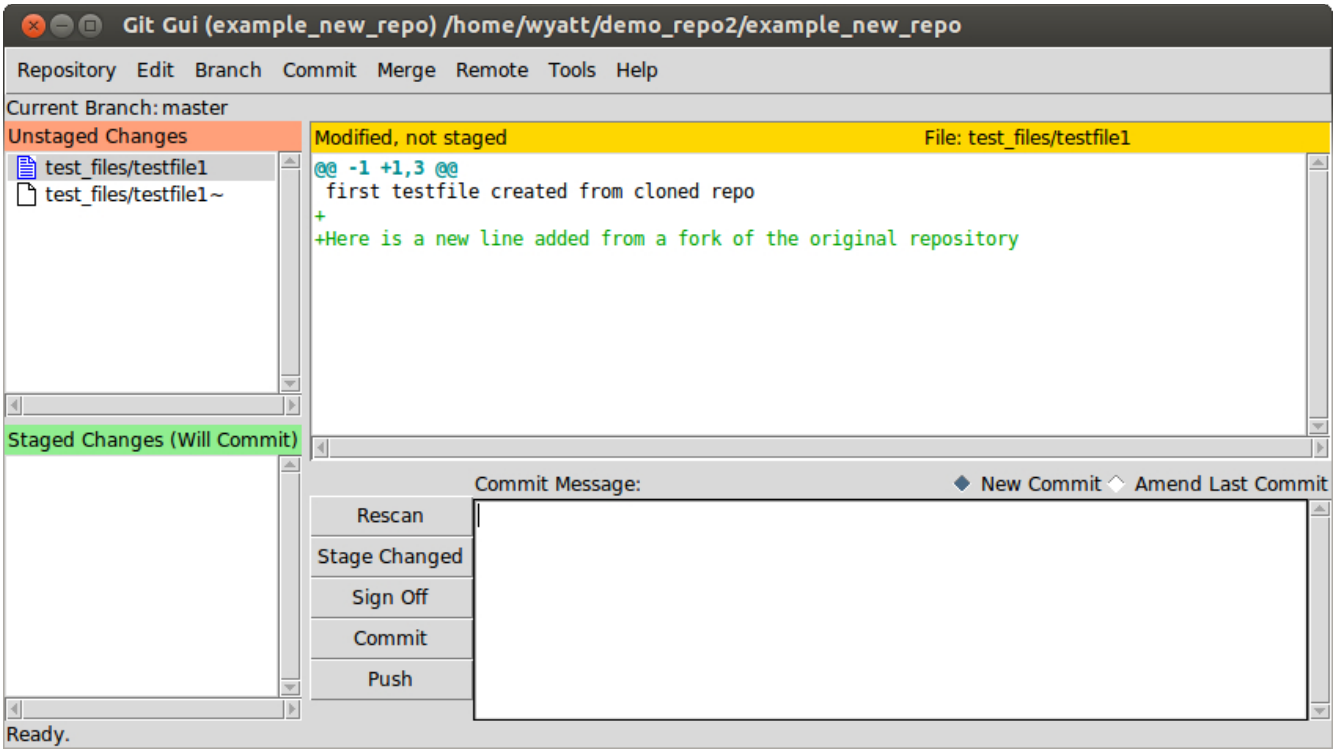
When this is completed, the view changes to the following:



This is a verbatim copy of the example_new_repo respository, except that it is now owned by the current github user.

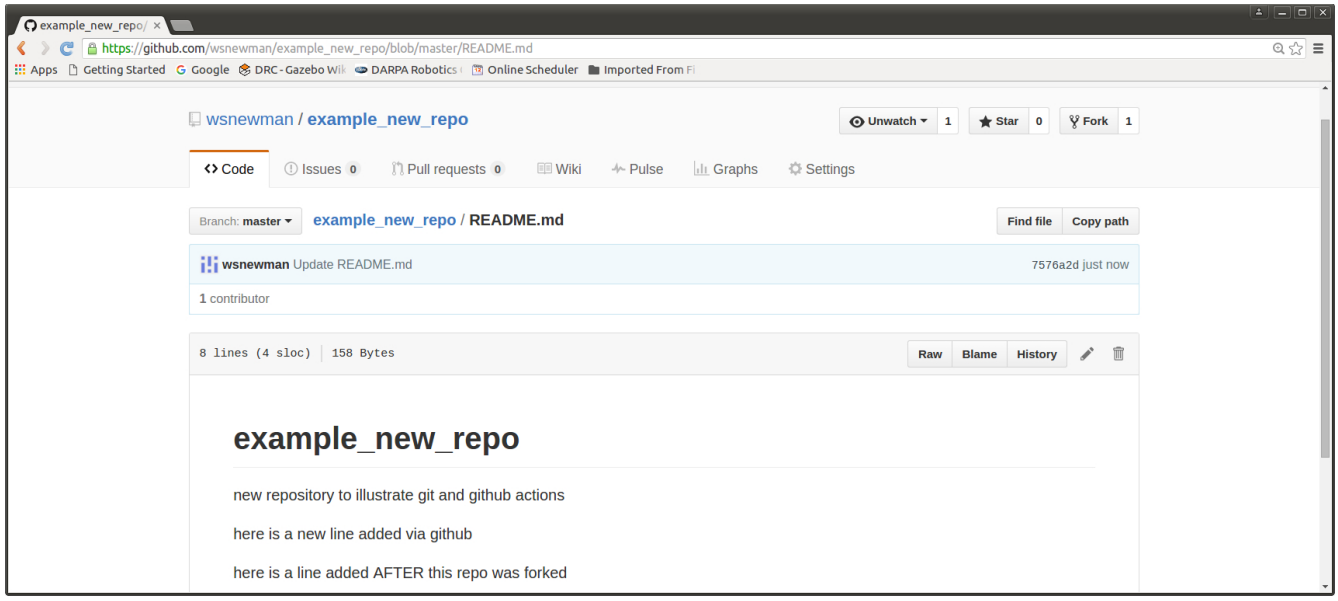
It is now possible to make a local “clone” of this forked repository, exactly as shown earlier, and changes can be “pushed” back to this repository. Forking has thus provided a starting point for new work based on an existing project. Greater value is realized when one contributes back to the original project, using a “pull request.”

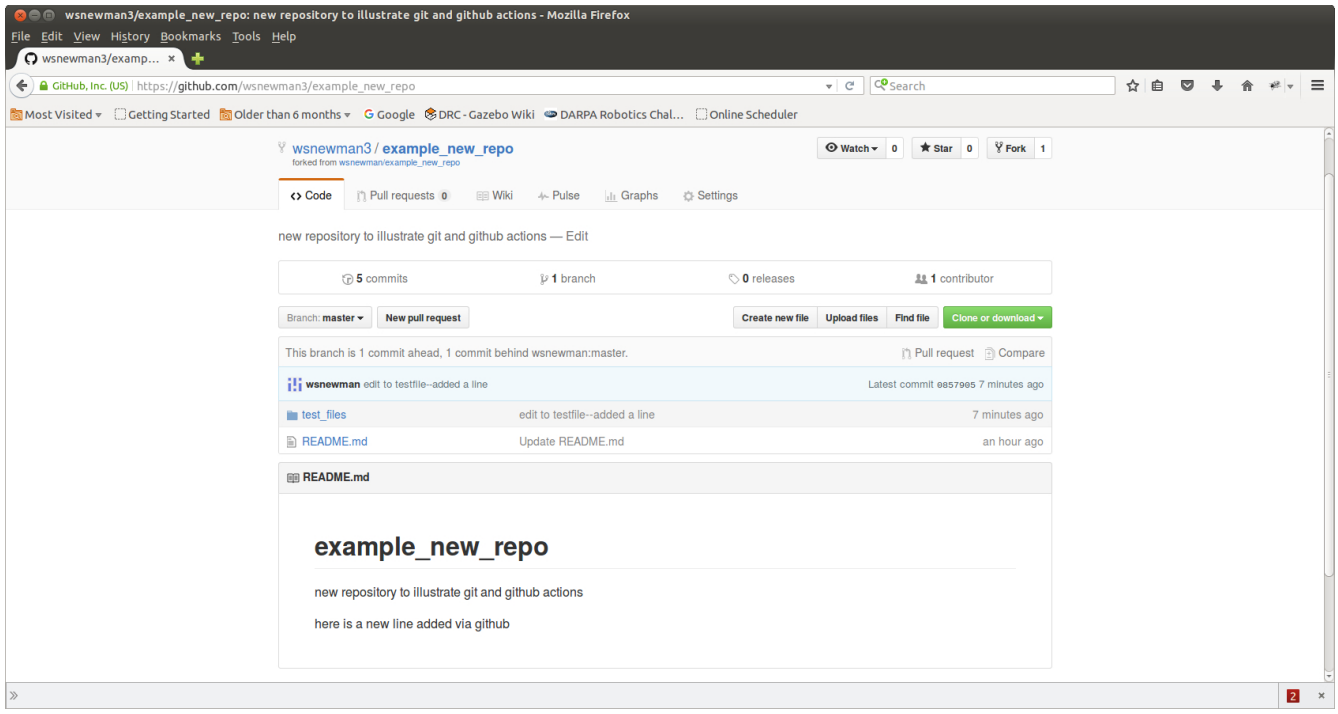
Pull requests:
Our “fork” is a new repository that resides on bitbucket. We can “clone” this repository to any computer locally and start making changes. For the present example, “testfile1” has been edited to add the line “Here is a new line addeed from a fork of the original repository.” Git-gui is then used to view this change:



which is then committed and pushed back to the forked repository. Our local changes are now synchronized with and preserved in the on-line version. However, the new fork is now out of sync with the original repository. This is not an issue if the new changes are never intended to get merged into the original repository. However, there are good reasons to keep these repositories in sync. First, the original repository presumably will get changes that should get incorporated into the forked repository. Further, if our new work is to contribute to the project, we need to get our changes merged into the original.

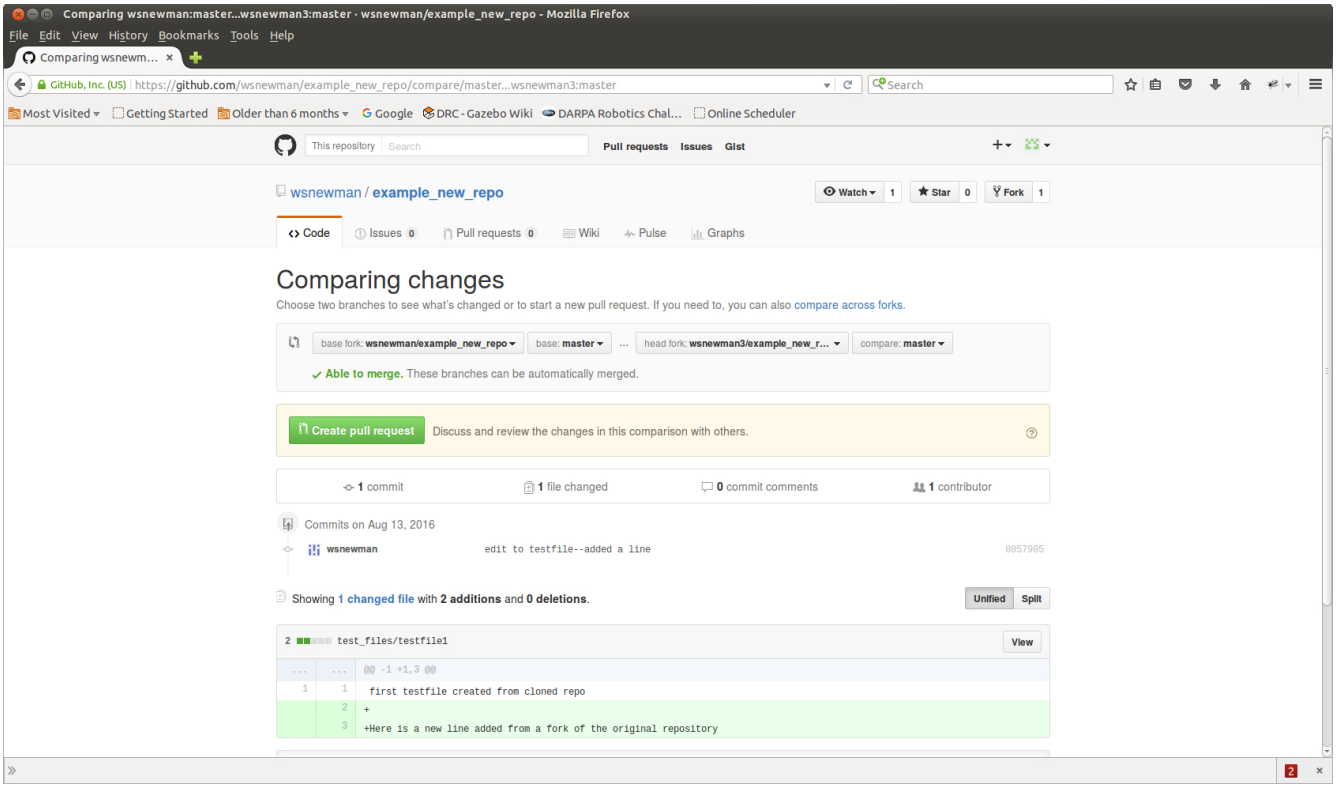
First, consider merging changes from the original repository into our fork. The original repository is again edited,



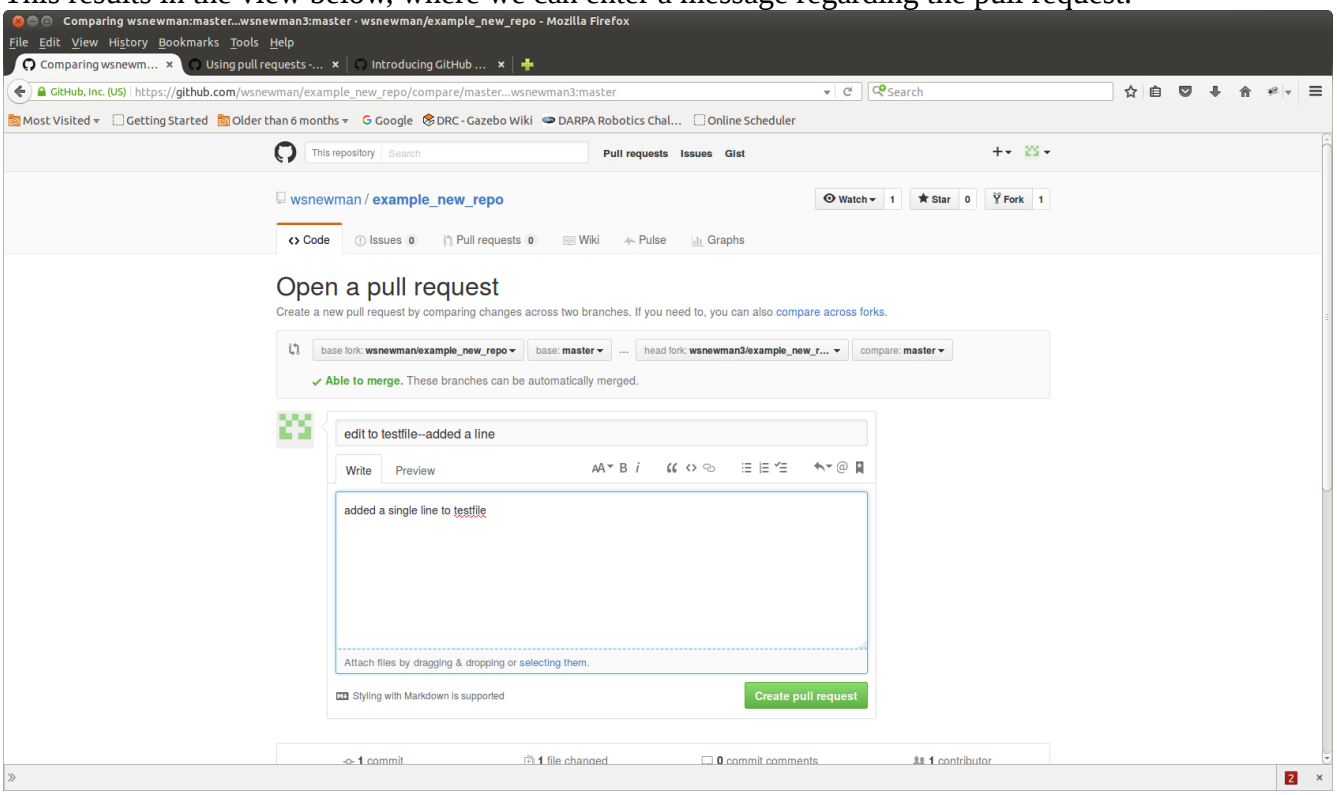


In our view of the forked repository, note the message “This branch is 1 commit ahead, 1 commit behind wsnewman:master.” This is telling us that both the original repository and the forked repository have new changes. We will want to import the changes from the original, and we will want to contribute our fork's changes back to the original.

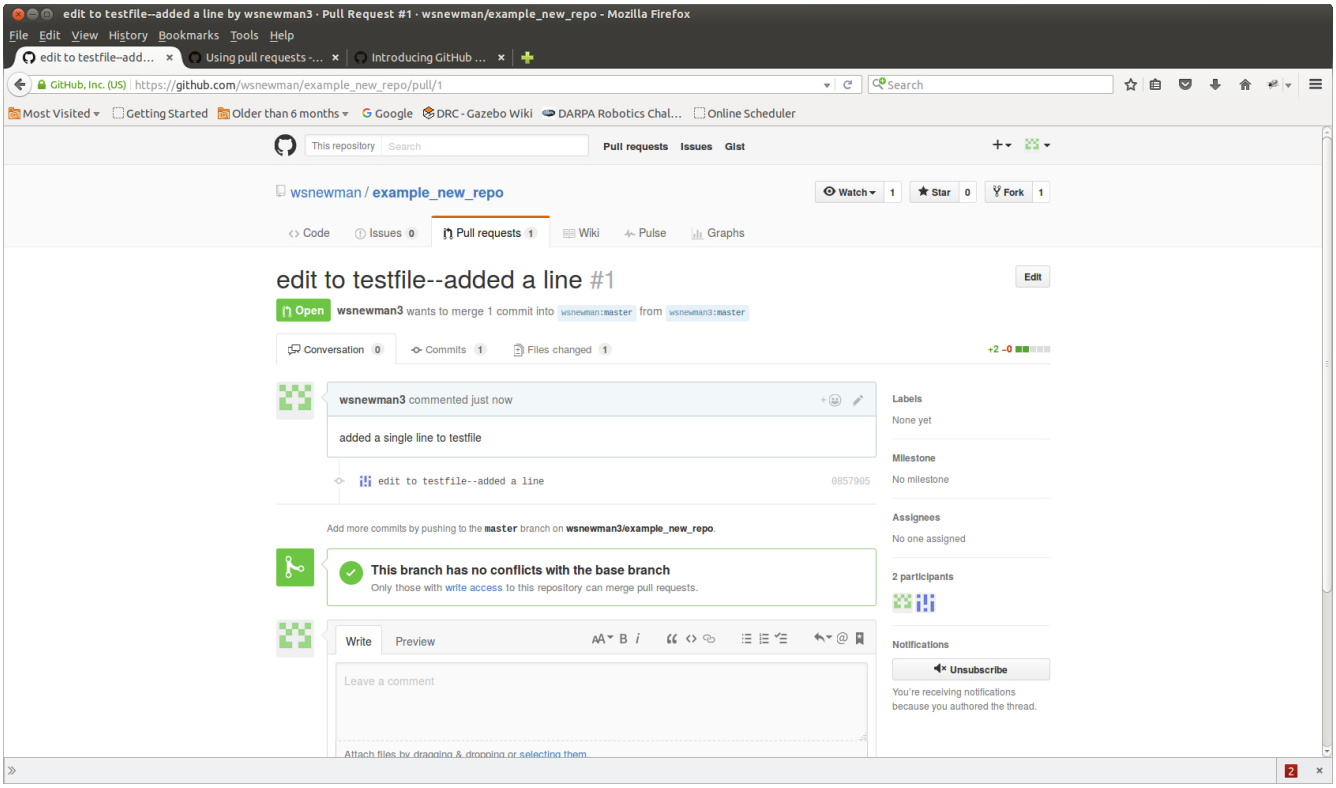
To update our fork with the lastest changes from the original, click the button “compare.”



This display shows us that the fork has a change that does not exist in the original. To request merging this change into the original (not owned by the current user), click the button “create pull request.” This results in the view below, where we can enter a message regarding the pull request.

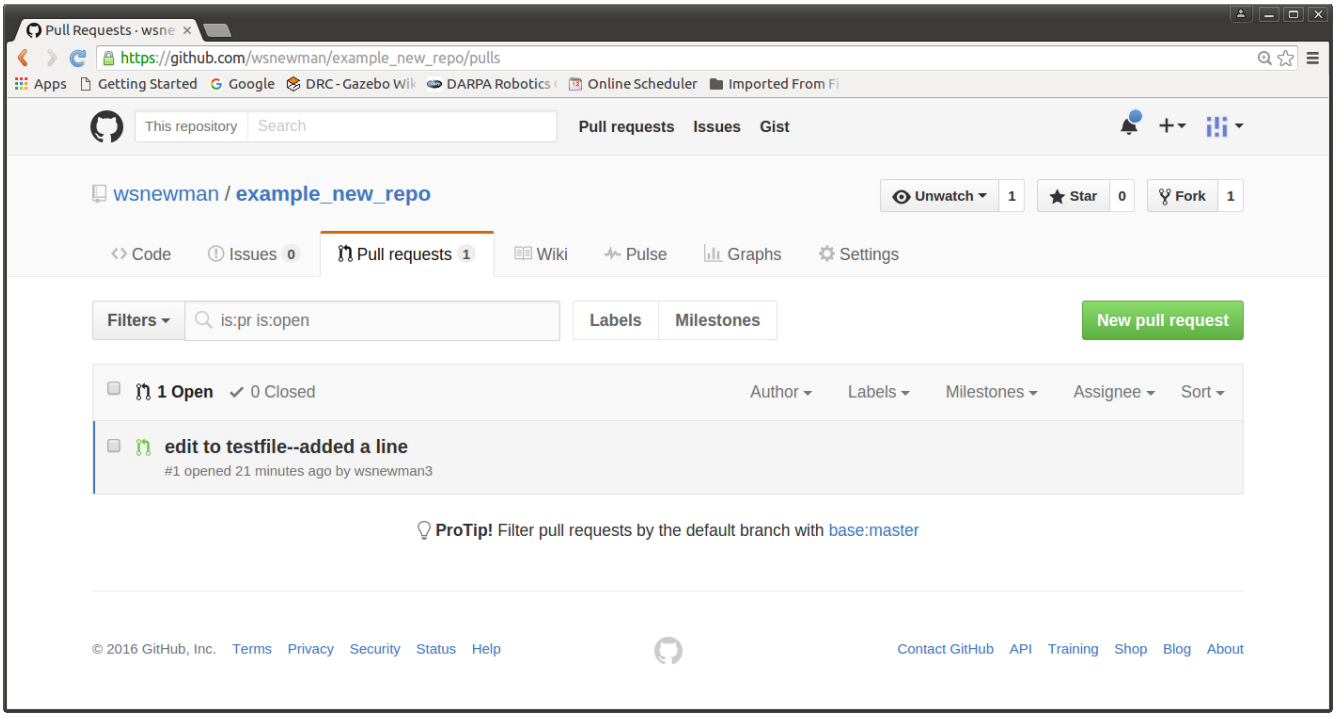


We again click “create pull request,” which produces the following output:

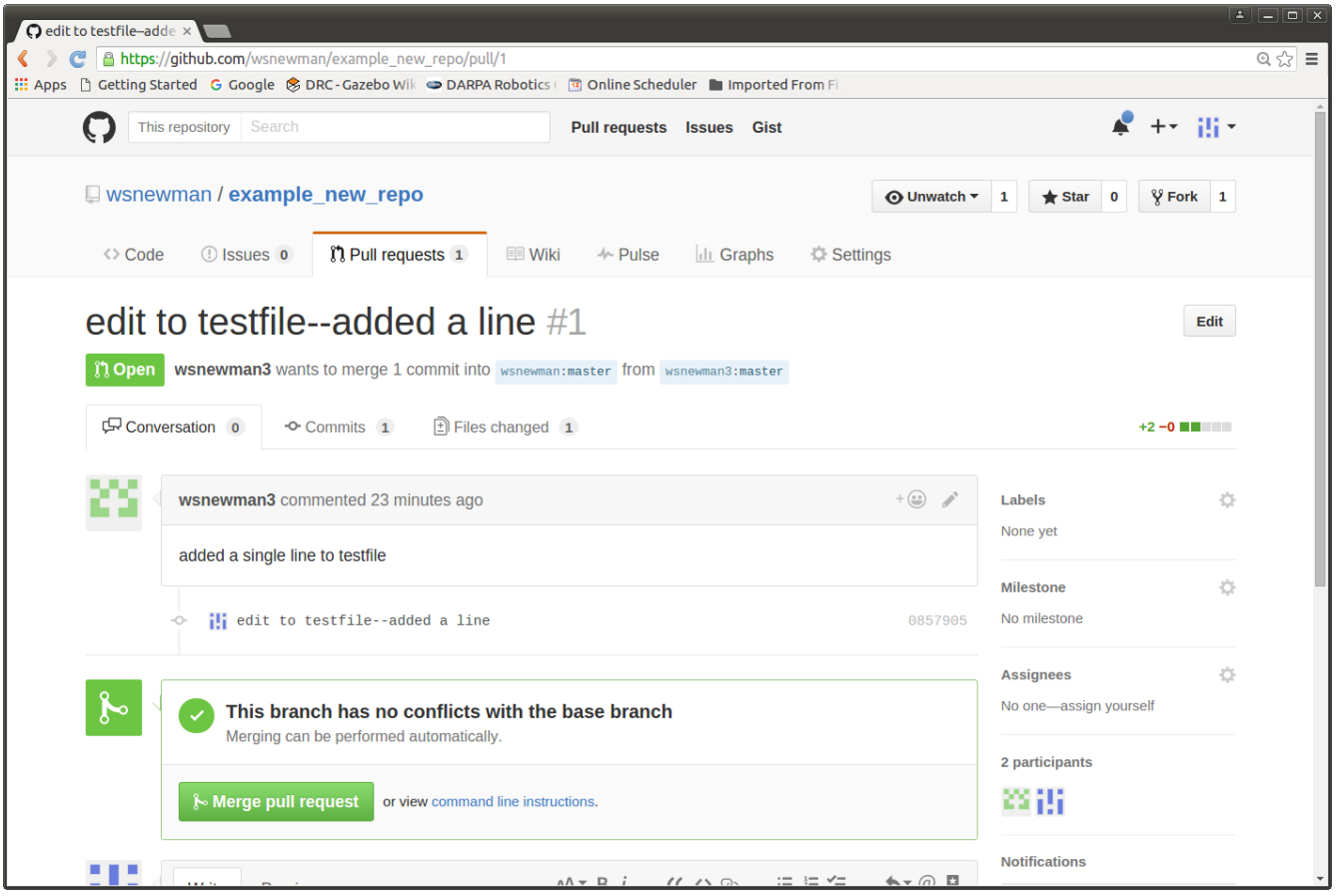


We see that our request has been entered. However, our proposed changes to the original may or may not be accepted by the owner.

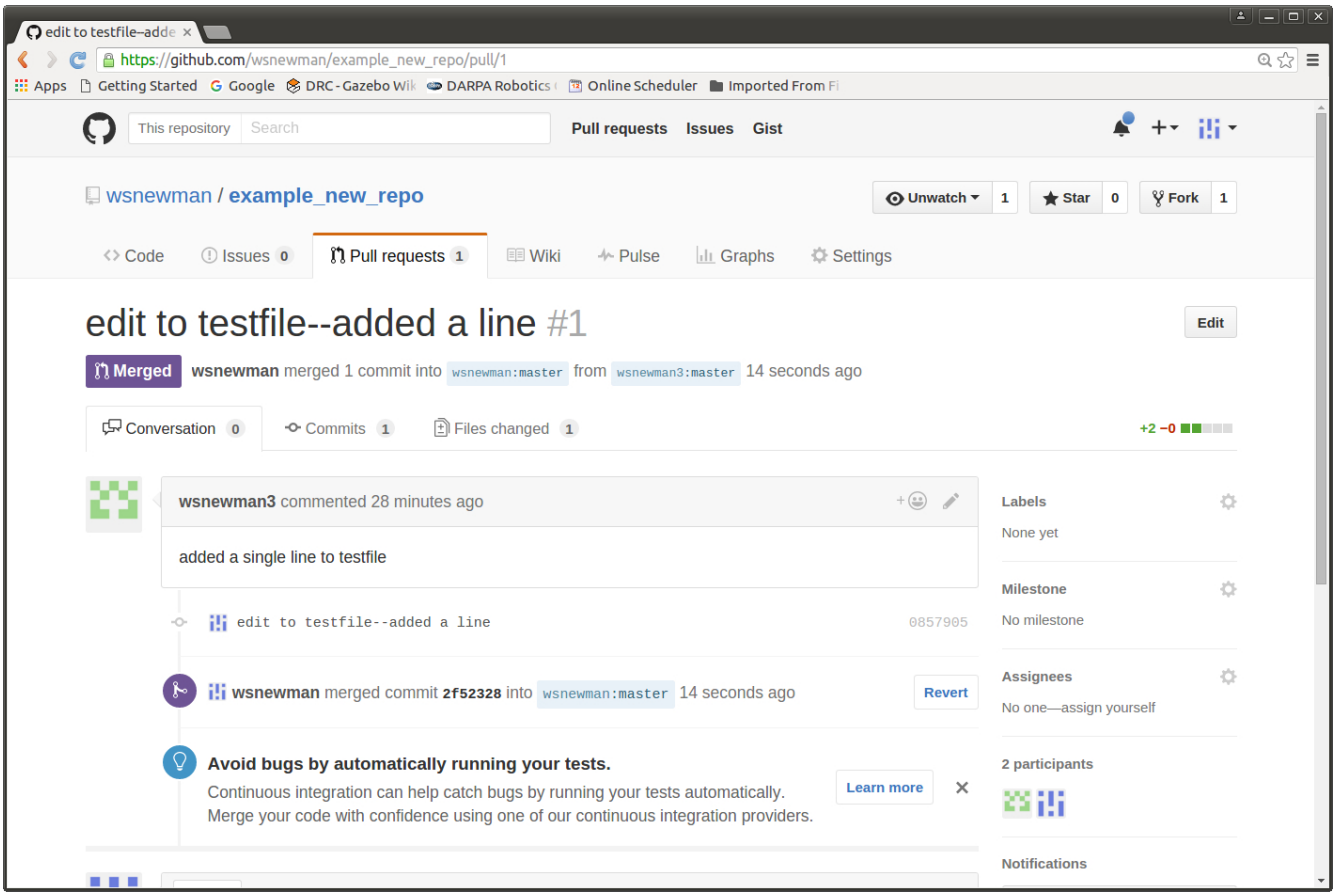
Switching to the owner's view on-line via github, we see that the original repository lists a pull requests. Clicking on the pull-request tab displays current pull requests.



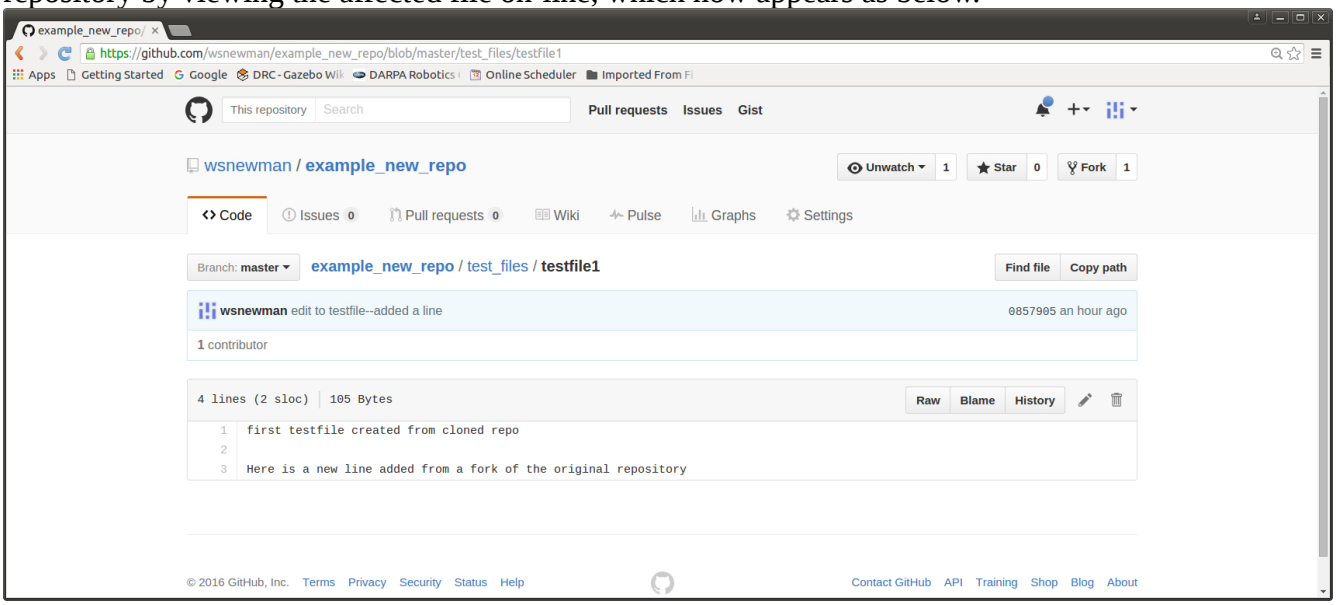
Selecting “edit to testfile—added a line”, and then clicking on the corresponding link presents the following display:



(There are options cut off below this screenshot, e.g. to reject the pull request). To accept the pull request, click on “merge pull request.” Again, only the owner of this repository can approve this merge. The user will be re-prompted to confirm the merge. The github display will then appear as follows:



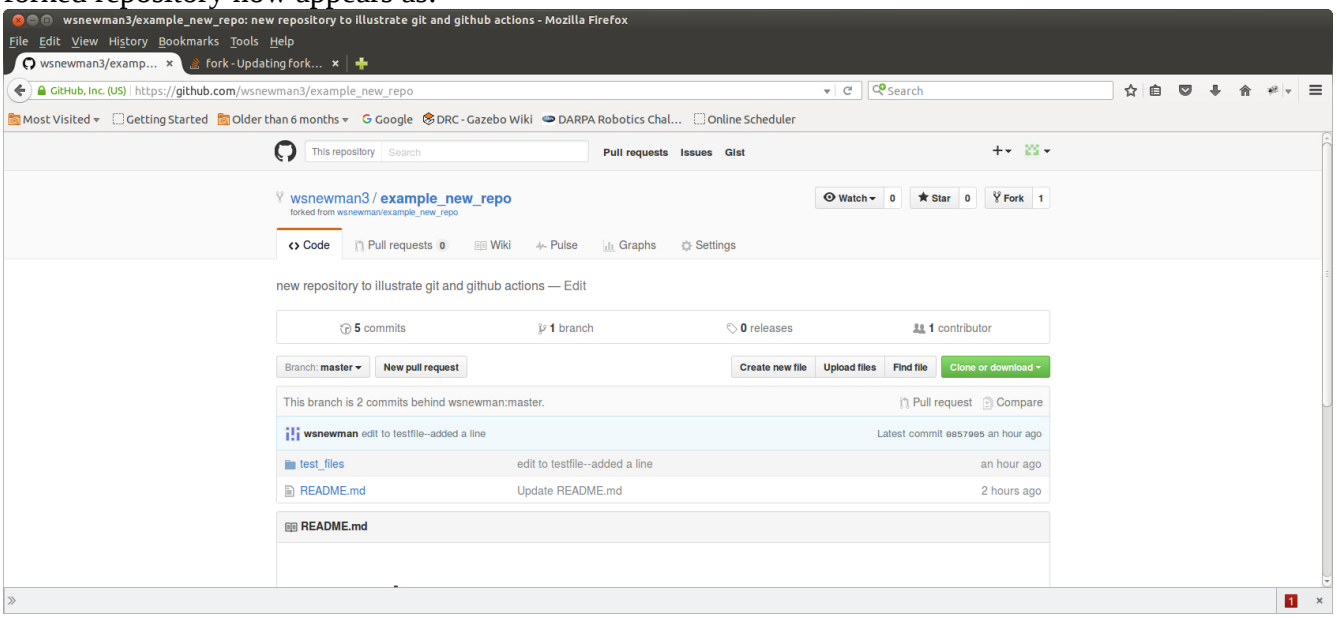
We can confirm that the change from the fork (the pull request) has been merged into the original repository by viewing the affected file on-line, which now appears as below.



We see that the new line added to the testfile appears in our original repository. The contributor who

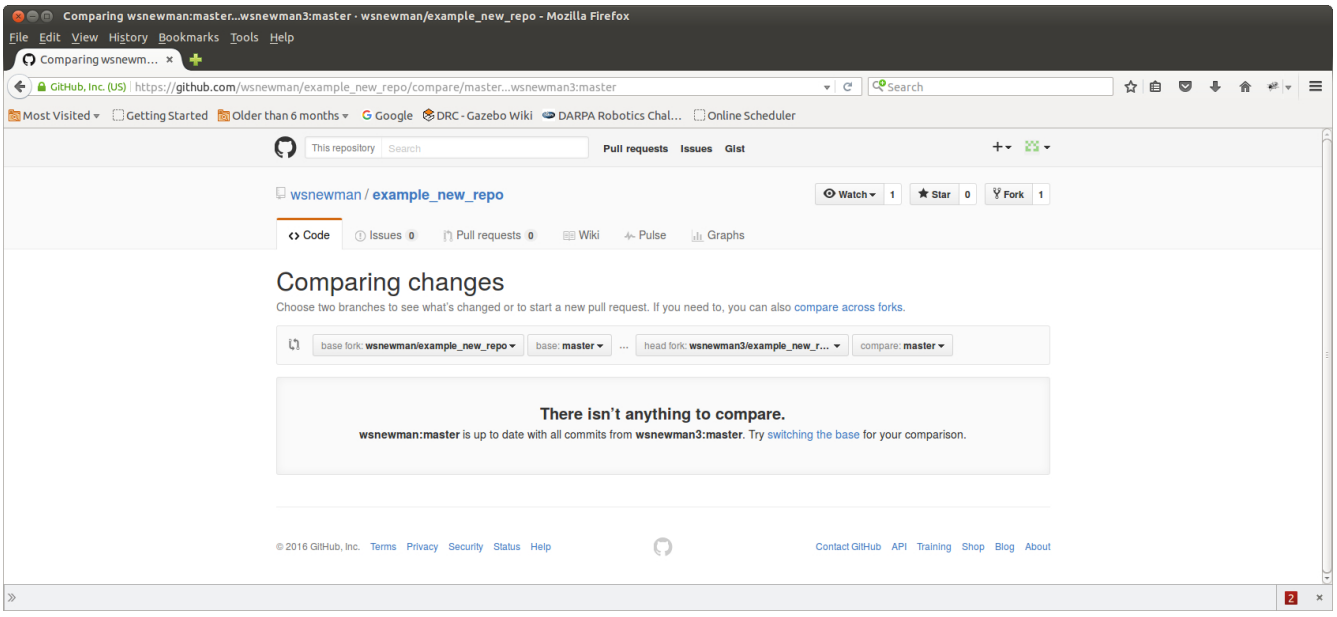
forked this repository has now successfully contributed back to the original project.

Returning to the collaborator's account (from which we forked the original repository), the view of the forked repository now appears as:

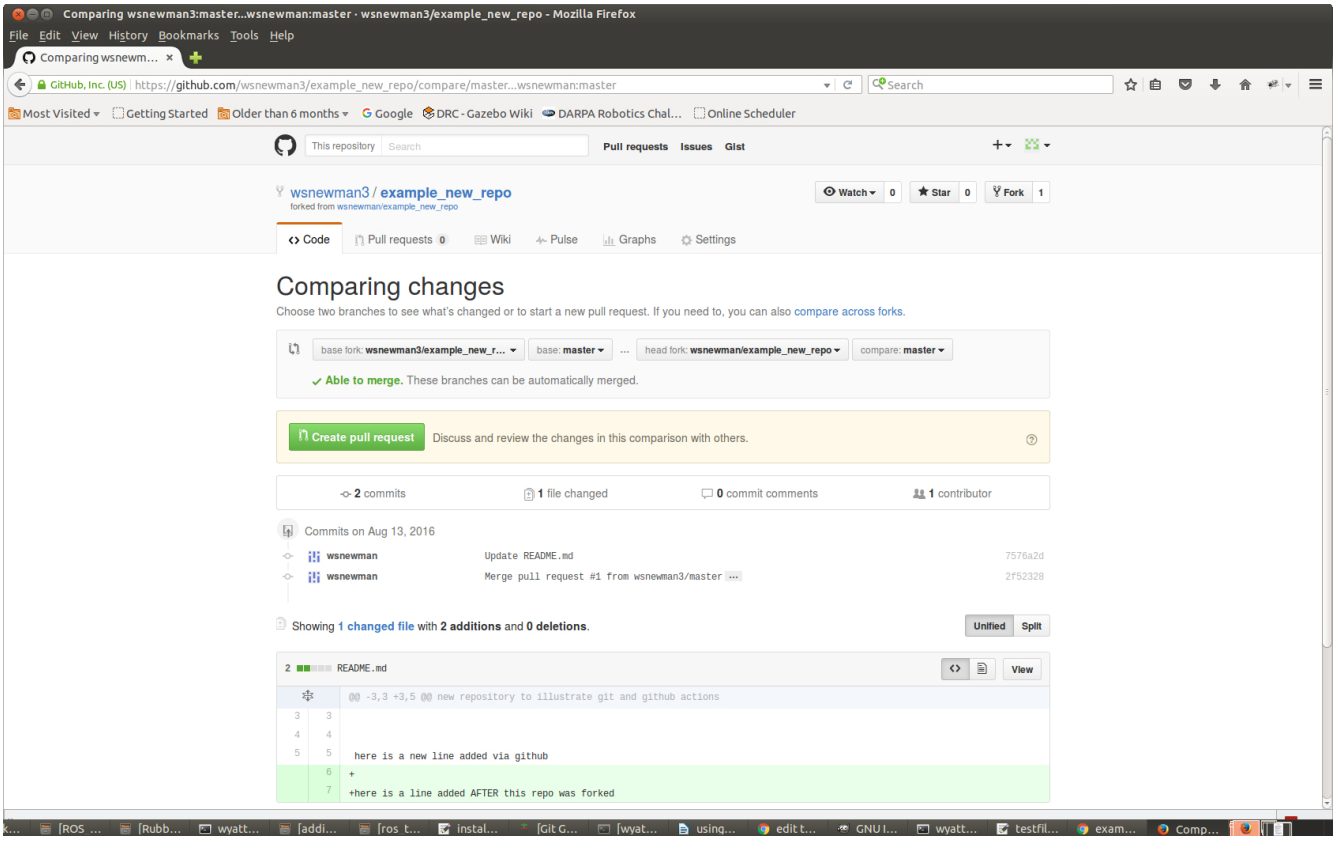


This view notes that “This branch is 2 commits behind wsnewman:master.” However, this branch is no longer “ahead” of the wsnewman:master, since our changes in the fork are now also part of the original repository.

Now, viewing our forked repository and clicking on the “compare” button yields:

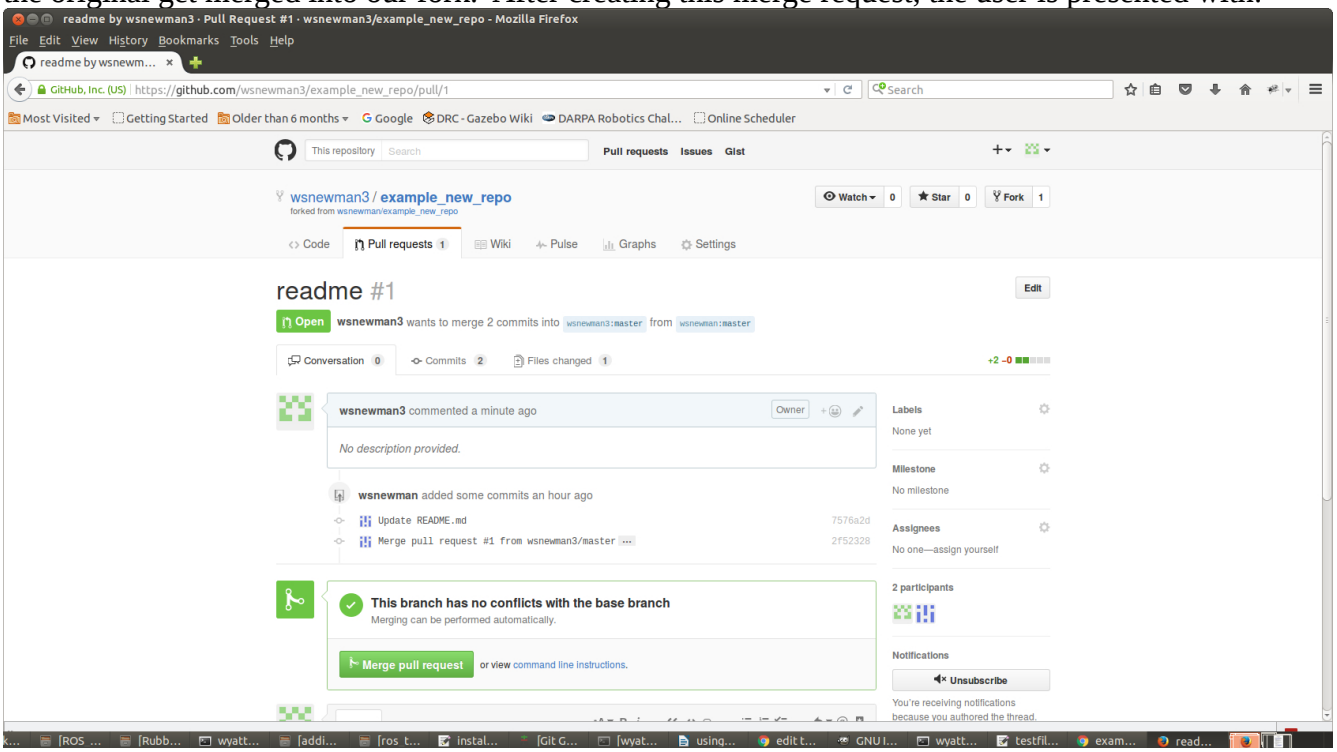


Clicking on the link “switching the base” will reverse the from/to of the comparison, showing:

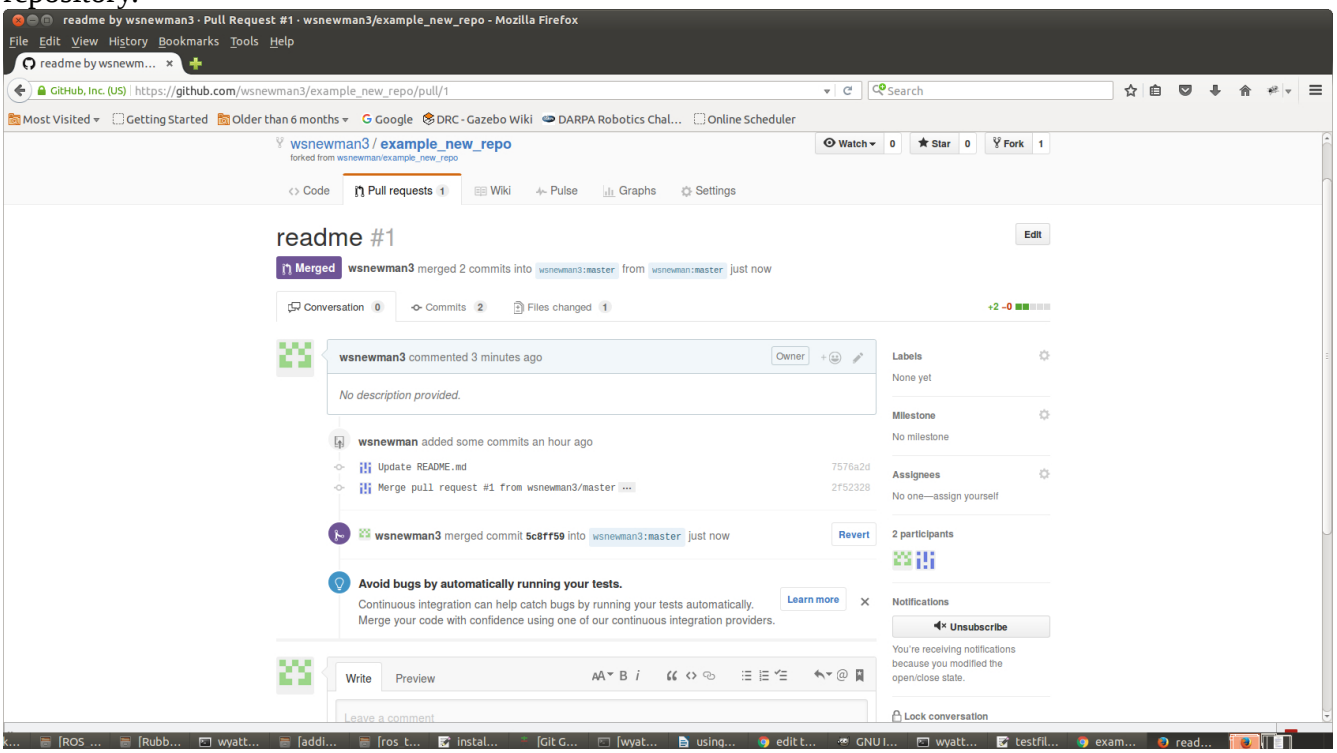


This shows that the change made to the README file in the original repository is not yet reflected in the forked repository.

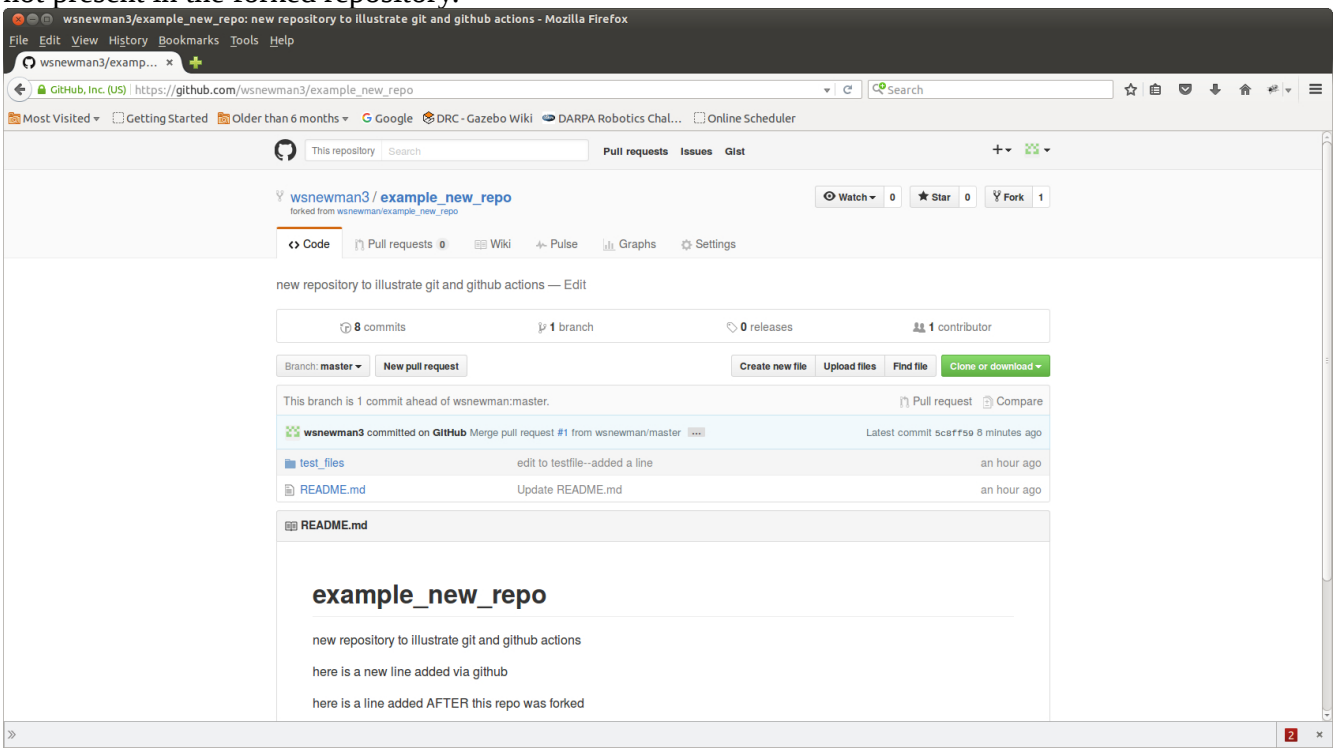
From this view, we can click “create pull request.” In this case, we are requesting that changes from the original get merged into our fork. After creating this merge request, the user is presented with:



In this case, the owner of the *fork* can self-approve their own pull request by clicking “merge pull request.” This results in the following display, confirming acceptance of the changes from the original repository.



After accepting this change into the fork, viewing the forked repo on github shows that the change is not present in the forked repository.



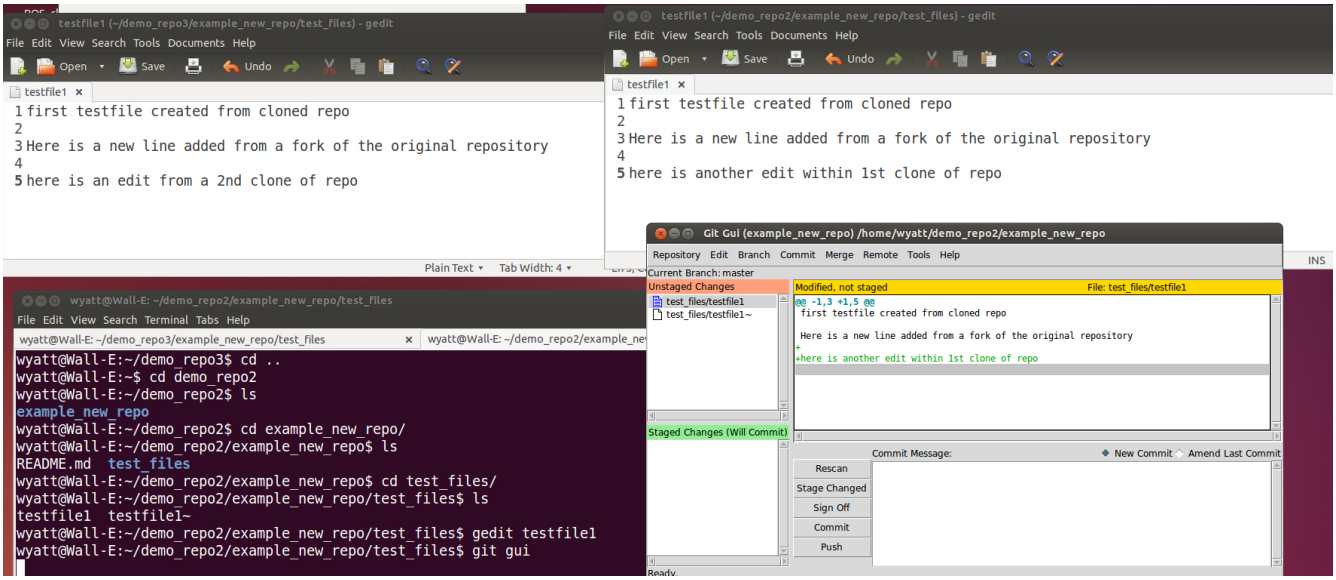
Since the fork has been updated, any clones of that fork on local machines should also get updated with a “git pull” command.

This shows the process for bi-directional updates—merging changes from an original repository into a forked repository, and contributing changes back from a fork into an original.

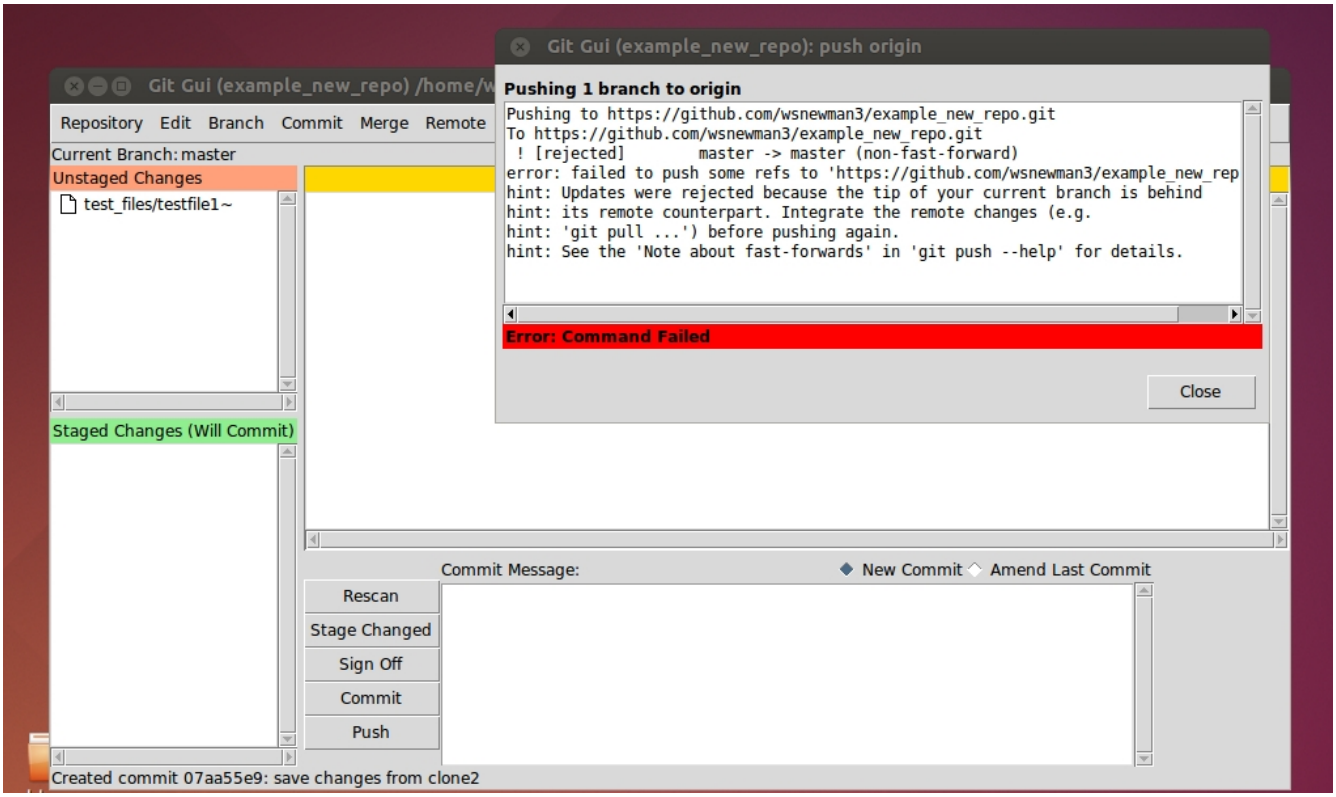
When working collaboratively, a concern is that multiple contributors will make changes that conflict with each other. Fortunately, tools exist to resolve such conflicts.

Resolving merge conflicts:
To illustrate how merge conflicts can occur, and how to resolve them, a conflict was deliberately created. Two clones of the (forked) example repository were created, emulating two collaborators working on the same project (and using the same repository, not separate forks). In this instance, the two collaborators were both editing the same file, “testfile1.” As a policy, your team should not have multiple developers editing the same file, but this sometimes occurs. In the example, one collaborator inserts the new line “here is another edit within 1st clone of repo,”, and the other collaborator inserts the line “here is an edit from a 2nd clone of repo.”

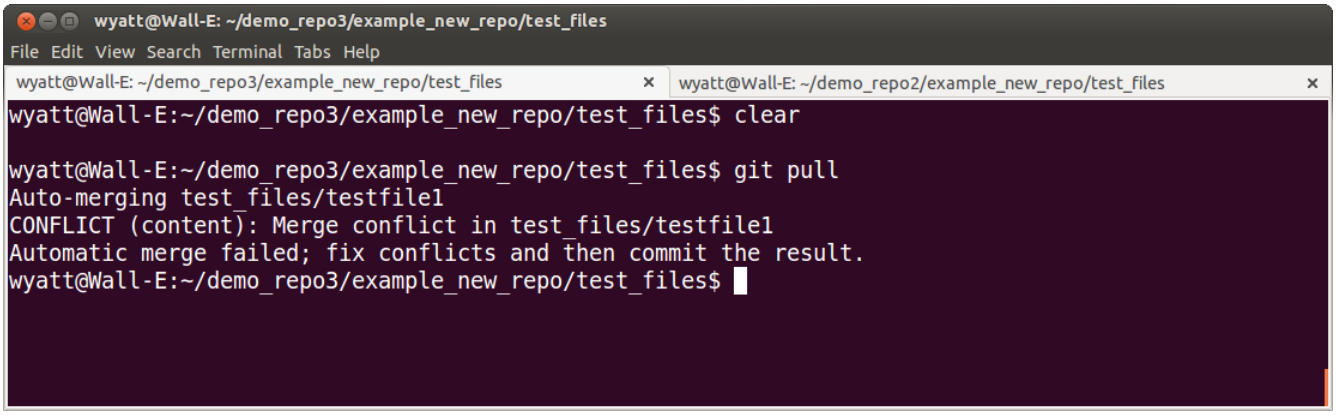
Collaborator 1 is the first to “push” their change to the repository, which is successful.



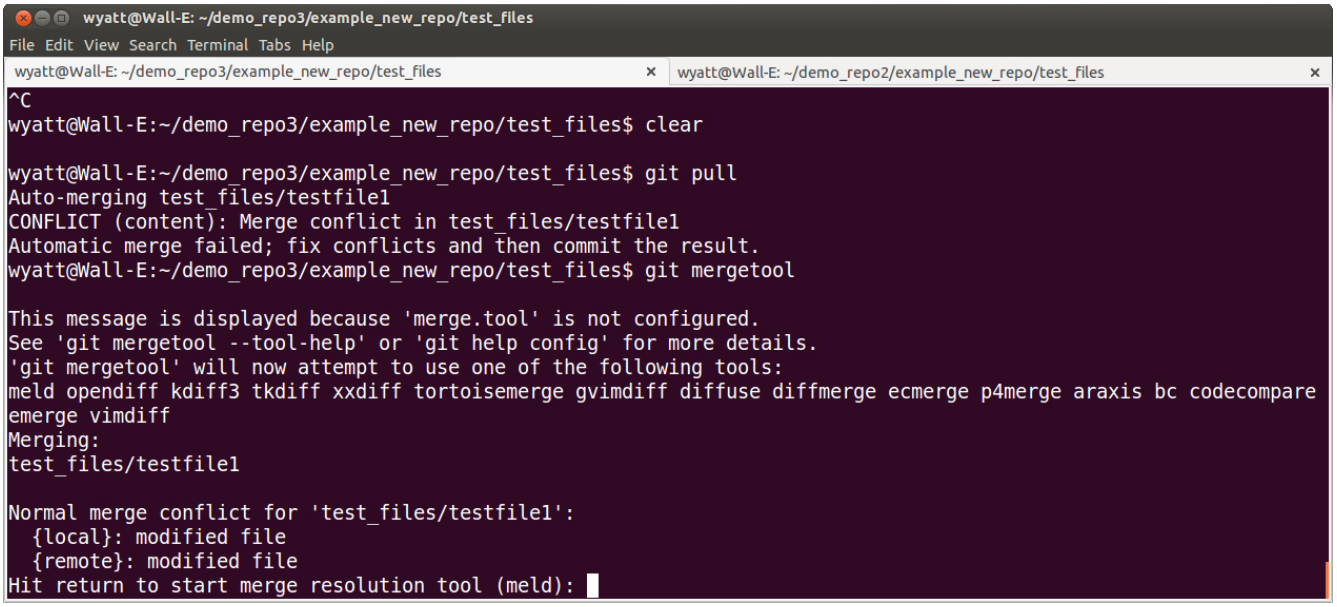
The repository has thus changed while collaborator 2 is still working, and this the 2nd clone is out of sync. When collaborator 2 tries to push her changes, an error is displayed, noting that collaborator 2 is working on an out-of-date clone. Git advises doing a “pull” to get collaborator 2's clone up to date.



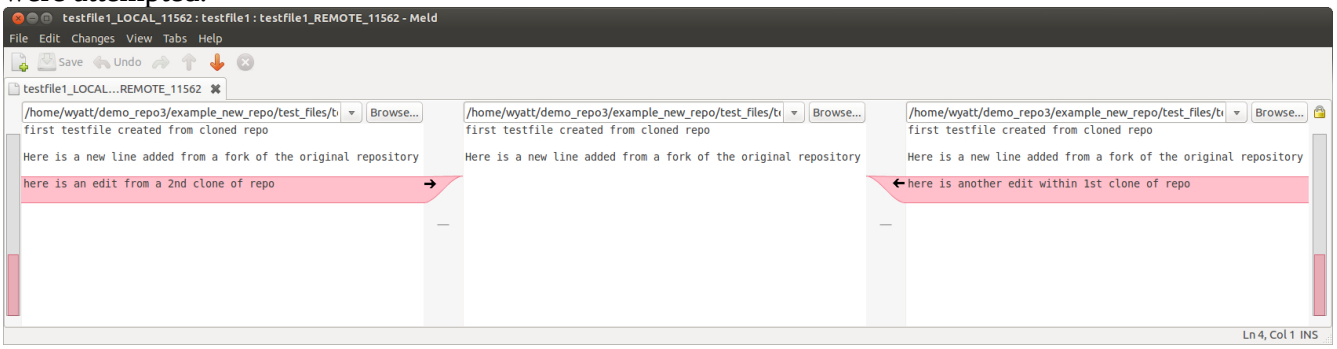
Typically, doing a “git pull” will bring a clone up to date without problems. However, in this case there is a conflict, since there are changes to the same file in the updated repository vs the local clone. Collaborator 2 sees the following error message:



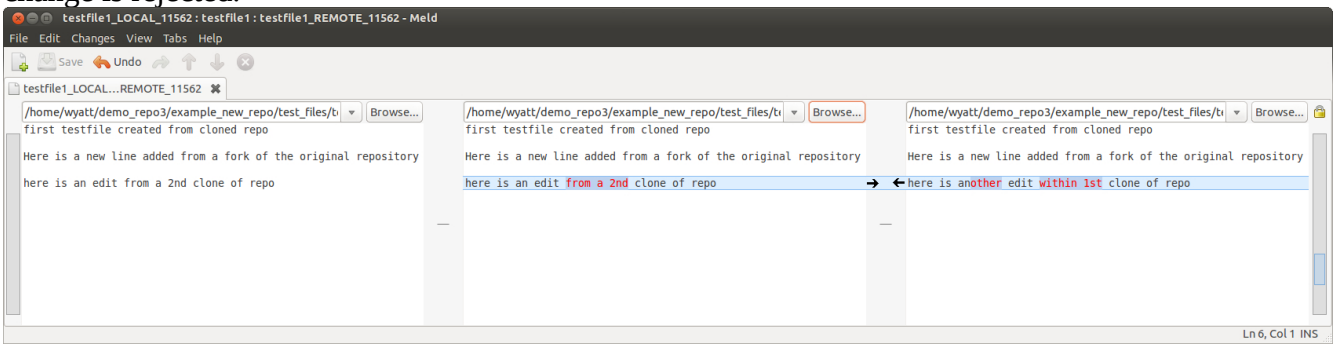
To fix conflicts, collaborator 2 can run: “git mergetool”. There are multiple tools available for resolving merge conflicts, and mergetool can be configured to use your favorite. In the present instance, a mergetool was not specified, and “git mergetool” chose to use the option “meld” by default.



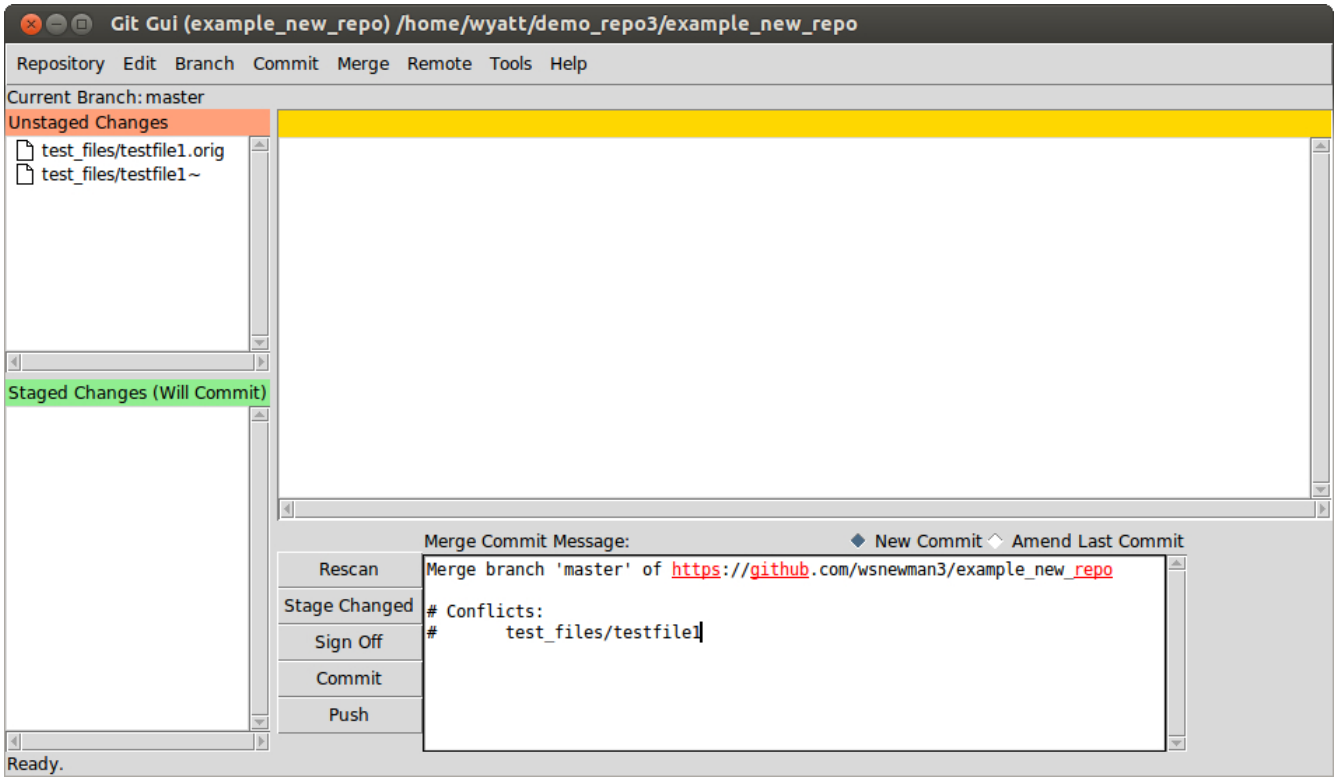
Hitting “return” presents a “meld” window, as below. This window shows changes from collaborator 2 (left pane), from collaborator 1 (right pane) and the original file before either of these two changes were attempted.



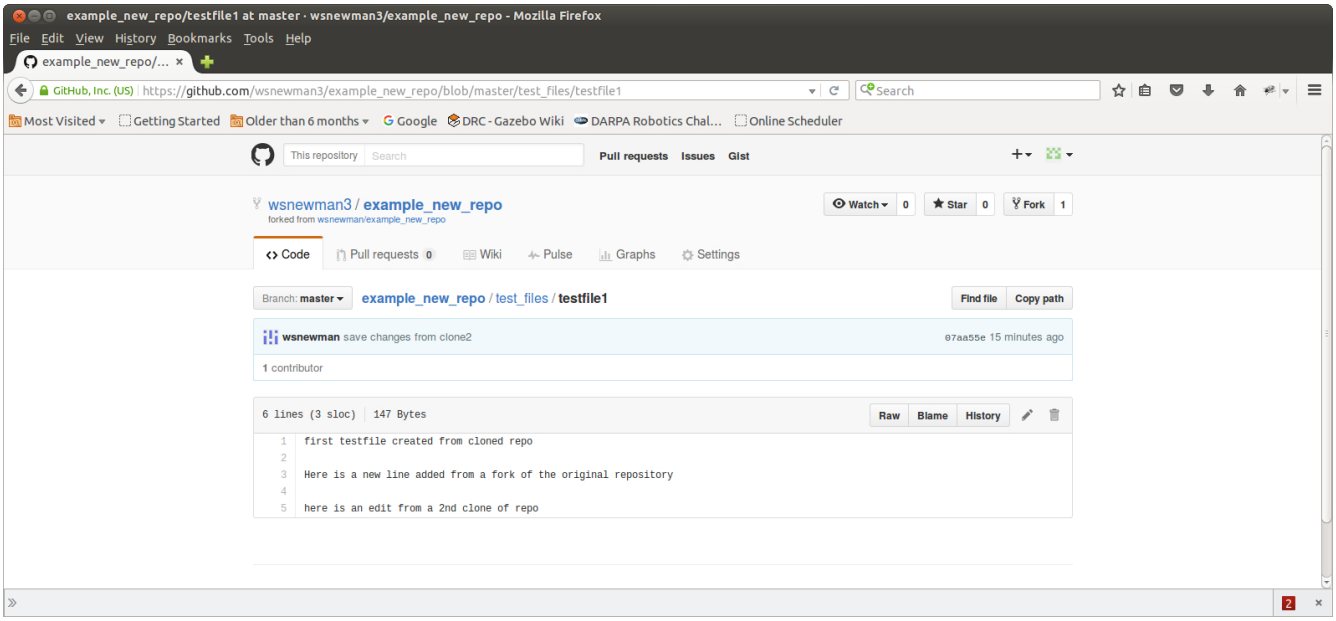
In this case, collaborator 2 chose to accept their own changes by clicking on the red-shaded line in the left pane. This resulted in collaborator 2's change getting inserted in “testfile1” while collaborator 1's change is rejected.



This change is then saved (from meld's “file” menu), and meld is closed. Collaborator 2 then re-attempts to push her changes, using git-gui:



The commit message was auto-generated. Pressing “push” results in a successful update to github. After this operation, viewing the repository on-line shows the following:



We can see that “testfile1” now contains the edit from collaborator 2, but not collaborator 1. The next time collaborator 1 starts working on this project, he should “git pull” from his cloned repository to get this latest update. (Running “git pull” at the start of a session should become habit). The three copies (github master, clone of collaborator 1 and clone of collaborator 2) are now all identical.

It may seem odd that collaborator 2 had the authority to reject collaborator 1's changes. Having collaborators work from the same repository, both with ownership privileges, requires trust in each

other. More generally, distributed collaborators would work with their own forks of a project's repository, and their changes would get contributed to the master via pull requests. Only the owner(s) of the master repository would have the authority to merge pull requests. Merge conflicts can still occur, but they would have to be resolved by one of the owners of the master repository. This may require discussion with the contributors. With this policy, the master repository can always be kept functional, and contributions from a large pool of collaborators can be managed.