# Introduction

## Purpose

This paper summarizes operations available from the GitDesktop application and relations with the GitHub website and the git Configuration Management tools. This paper identifies almost all of the icons employed and describes how the various icons, buttons and displays can be used to take advantage of the GitHub services.

## Scope

This paper is intended to bridge a gap in the available Git-related documentation that is relevant to the CSSA CAP project and describe how the tools play together. The primary scope is to:

* Identify key terms relevant to Git, GitDesktop and GitHub
* Identify key icons/buttons/views of the GitDesktop application
* Point to relevant reference materials where much more detail can be found

This paper is definitely NOT intended to be a comprehensive description of Git, GitHub, or the GitDesktop application.

## Definitions

Some key definitions are included below, with only the most salient aspects relevant to this summary paper are identified. Where desired, detailed definitions should be sought from the listed reference materials (e.g., [3] is recommended as a succinct reference for computing professionals).

* Git – the core git Source Code Configuration Management tools
* Git Command Line – the textual command line interface to the git tools
* GitHub – an internet-based git repository service that implements git, plus additional custom tools that integrate with those git tools, providing a convenient graphical interface to Git with
* GitDesktop – a shortend reference to “GitHub Desktop”
* GitHubDesktop –another variant of the official name “GitHub Desktop”
* GitHub Desktop - the official name of the application that provides a graphical interface to git services and GitHub services.
* **Remote** – the common term to refer to the Git repository that is stored within a remotely located data service, e.g., the internet-based GitHub data server. Note: based on git services, a “remote” repo can exist in pretty much any location that is accessible via current generation network protocols (e.g., http, file, ssh). GitHub-based remote repositories will use the http service by default, and now you can forget about that trivia and focus on the fact that the GitHub repository you access is the “remote”. Compare and contrast “local” vs. “remote”. *Note: understanding the distinction between remote and local is critical to git-based CM operations.*
* **Local** – the common term to refer to your own “local copy” of the Git repository within which you perform your daily work activities semi-independent of other developers (i.e., except for the coordination points between your “local” and the “remote”). Compare and contrast “local” vs. “remote”. *Note: understanding the distinction between remote and local is critical to git-based CM operations.*
* Commit - this term is used as both a verb and a now. The “commit action” creates a “commit entity” within the user’s “***local***” repository. The “commit entity” provides a record of all changes within the designated files, whether it is one file or several/many/hundreds/etc files that are modified, providing a discrete bundle of information that encapsulates all of those changes into a single entity.  *The “commit” is a fundamental artifact in all git operations.*
* Push – this term is a basic git technology term refering to the action whereby a “commit” from the user’s local repository is copied/pushed –TO-- the remote repository.
* Pull – this term is a basic git technology term refering to the action whereby one or more “commits” from the remote repository will copied/pulled –FROM-- the remote repository.
* Publish – this is a GitDestop term associated with the initial “Push” to the remote repository. GitDesktop has a Publish button that will be visible when that operation is a viable choice based on the user’s local repository (i.e., if the user’s local repository includes one or more commits that have never been pushed to the remote repository).
* Sync - this is a GitDestop term associated with the combination of Push and Pull operations to and from the remote repository and local repository in order to synchronize the content of remote and local.

## References

* [1] GitDesktop tutorial – some tutorial materials ccessible via the GitHubDesktop application. Go to upper right corner to the “Gear Icon” 🡪 Button.Click(Gear Icon) 🡪 Button.Click(Tutorial Command)
* [2] Introducing GitHub: A Non-Technical Guide, Peter Bell & Brent Beer, O’Reilly
* [3] Git Pocket Guide, Richard E. Silverman, O’Reilly
* [4] Git, Ry’s Git Tutorial, Ryan Hodson, Kindle Books, Amazon.com

# Background

* git is not the first CM tool, and it improves on previous capabilities, integrating some important lessons learned. Legacy tools include: rcs, sccs, cvs, svn, et al.
* git philosophy: git continues the trend toward utilization of distributed CM capabilities with the implementation of separate/distributed, but coordinated, CM repositories. With Git’s distributed sharing paradigm, each user has a complete “local” copy of the “remote” CM repository, can perform work within the local repository for some period of time independent of the changes being made by others in their own repositories or the shared “remote” repository.
* Git keeps everything. All versions of all files that are recorded/committed to the git repository are archived and can be retrieved upon command.
* GitDesktop provides some very convenient graphical representation of branches, commits, and file differences that greatly facilitate git operations using GitHub
* GitDesktop is designed to work with Git locally and with GitHub remotely.
* A user’s git operations and GitDesktop operations can continue with basic operations using the local repository while the GitHub remote is unavailable, leaving GitHub synchronization of branches to be performed whenever GitHub access is restored; the result is you can continue to work productively if your internet provider’s service goes out for a while.

# Example GitDesktop operations

The purpose of this section is to identify the icons and views of the GitHubDesktop application’s user interface and associate the icons/buttons/views with the operations that the “tools” (i.e., emphasis on the plural) will execute when clicked.

* There are multiple tools that execute their services in coordinated fashion to achieve the combined GitHubDesktop/GitHub/Git operations.
* The contents, including icons, buttons, etc are updated dynamically by the GitDesktop app to show only the operations that are relevant based on the current state of the user’s operations and the repository content. This feature can be a bit disconcerting at first glance since the visible items change, but it is very useful in the normal workflow.

## Definition of some GitDesktop window elements

Several of the basic icons/buttons of the GUI are illustrated in the screen capture of Figure 1-GitDesktop Sample View. This is one of many possible view combinations that may be presented by GitDesktop, but it illustrates several key components.

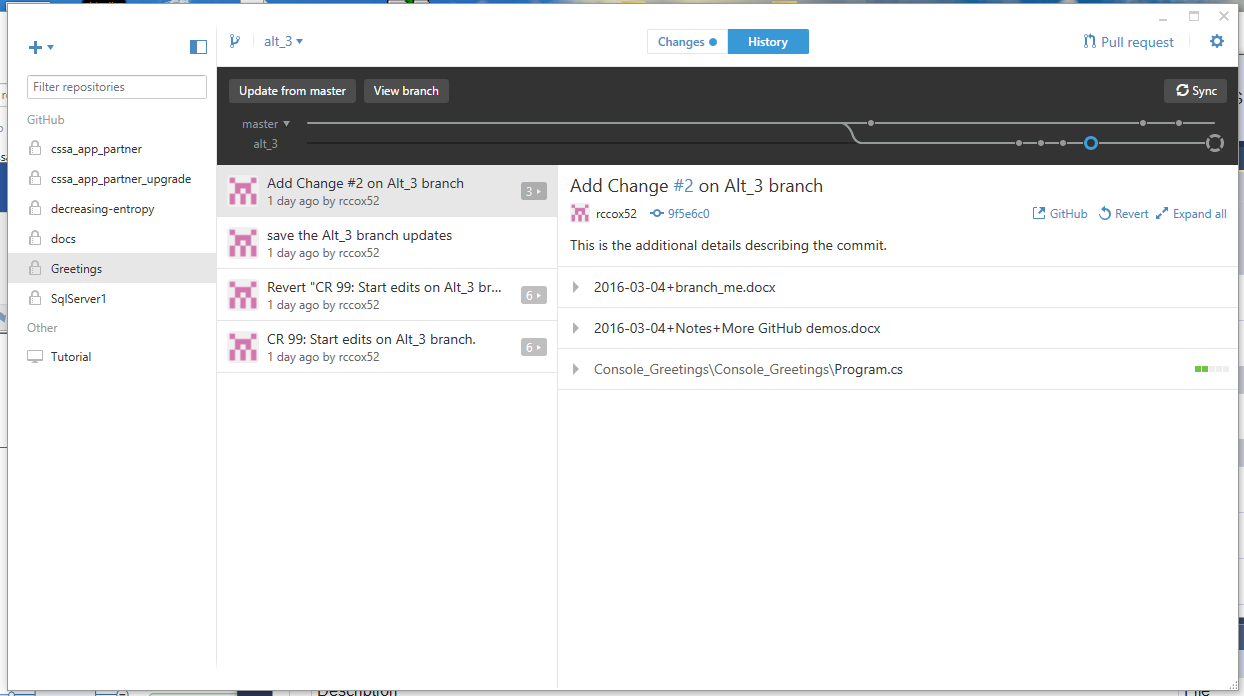
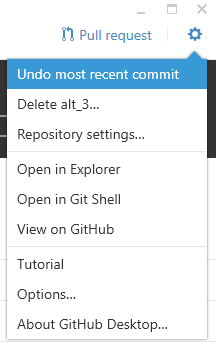


Figure 1-GitDesktop Sample View

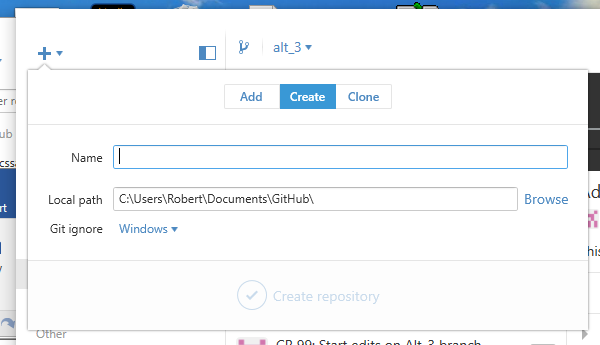
**Icons/Buttons/GUI Elements**

* “+” icon – Where {in the top left corner}; What: {when clicked, it shows a dialog window with options to Add/Create/Clone a repository.)
  + Graphic:
* Collapse/Expand icon – Where {in the top left corner}; What: {when clicked, it changes the display of the repository list, hiding the list when “collapse” is selected and showing the list when “Expand” is selected; helping the user manage the available space on the display).
  + Graphic: 
* Create Branch icon - Where {in the top left side, to the right of the Collapse/Expand icon}; What: {when clicked, it provides the option to create a new branch within the user’s local repository, with a user-specified name, with the branch created relative to another existing user-specified branch}
  + Graphic: 
* Current Branch/Select Branch - Where {to the right of the Create Branch Icon}; What: {it shows the current selected banch, and when clicked, it provides a drop down list from which an existing branch can be selected}. After selection of a branch, the content of the current work directory is updated from the local repository to match the content of the selected branch. As many files as necessary are copied from the local repository to the working directory to make the working directory match the repository. Before extracting any files from the repository, git first determines if the file extractions woud overwrite any existing unsaved changes. The operation is NOT performed if unsaved changes would be lost and a warning is generated to inform the user.
  + Graphic: 
* Changes/History Buttons - the git repository viewing mode is determined by the button setting. The Changes button, when selected, controls the presentation of the local work directory changes relative to the local repository. The History Button, when selected controls the presentation of the sequential list of commits to the repository.
  + Graphic: 
  + 
* Pull Request – TBD – skip this for now.
  + Graphic: 
* Gear Icon – more commands
  + Graphic: 
  + Command Menu:
  + 
  + Open in Explorer – provides a quick and convenient method to open an explorer window in the folder in which the local repository is located.
  + Open in Git Shell – provides a quick and convenient method to open the Gill Shell (command line interpreter) in the folder in which the local repository is located.
  + View on GitHub – provide a quick and convenient method to open the Browser on the URL where the GitHub repository is located.

## “+” icon operations

It provides a function that is similar to a drop-down menu, allowing the selection of one of three operations:

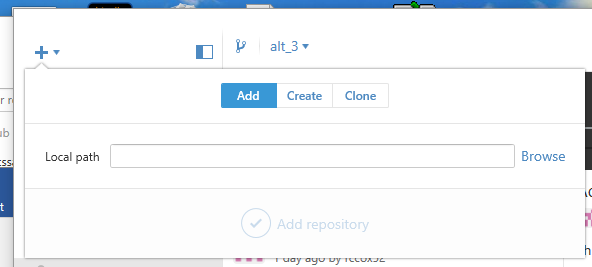
* Add – from the user’s local file system, add an existing “local” git repository to the list of repositories known/tracked by the GitDesktop application.
* Create – within the user’s local file system, create a new “local” git repository
* Clone - within the user’s local file system, create a cloned copy of a specified remote repository, saving the copy to the user-specified local folder



## Repository Add Operation

The Add operation is based on the user’s existing local repository from a local folder.

* Add button - The “Add” button is selected, indicate the ADD operation will be performed.
* Local path – the user must specify the path specification to the local git repo of interest
* Browse button – the user can go to the File selection dialog to navigate to the folder of interest instead of manually typing the path specification.
* Add repository button – the button is enabled after the user has specified some path text (presumably a valid path to an existing git repository)



## Repository Create Operation

The Create operation is based on the user’s existing local file system within which a local git repository will be created for the first time:

* Create button - The “Create” button is selected, indicating the CREATE operation will be performed.
* Name – the user must enter a user-defined, user-meaningful name for the new git repo
* Local Path – the user must specify the fully qualified path to the file system location where the new git repo will be created
* Brows Button – instead of typing the path manually, the Browse button accesses the file selection dialog from which the user can navigate to and select a specific location for the new repo
* Create repository – after entering the required parameters, the user clicks this button to execute the “git init” command that creates the new repository within the local file system. NOTE: this create operation has no immediate impact on any remote repository. e.g., the GitDesktop will show the “View on GitHub command is disabled immediately after the Create command since the new repository does not yet exist on the remote (i.e., the GitHub server)

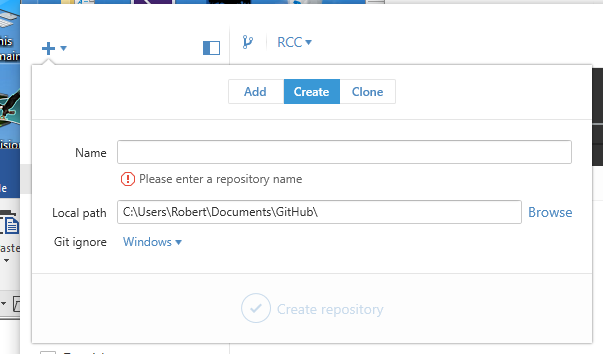


Figure 2-Create Repository Dialog

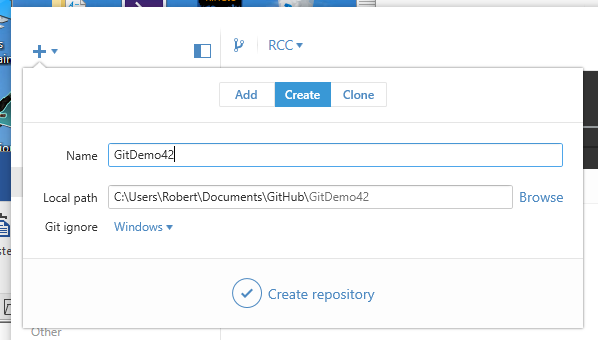


Figure 3-Create Dialog Repository, Ready to GO!

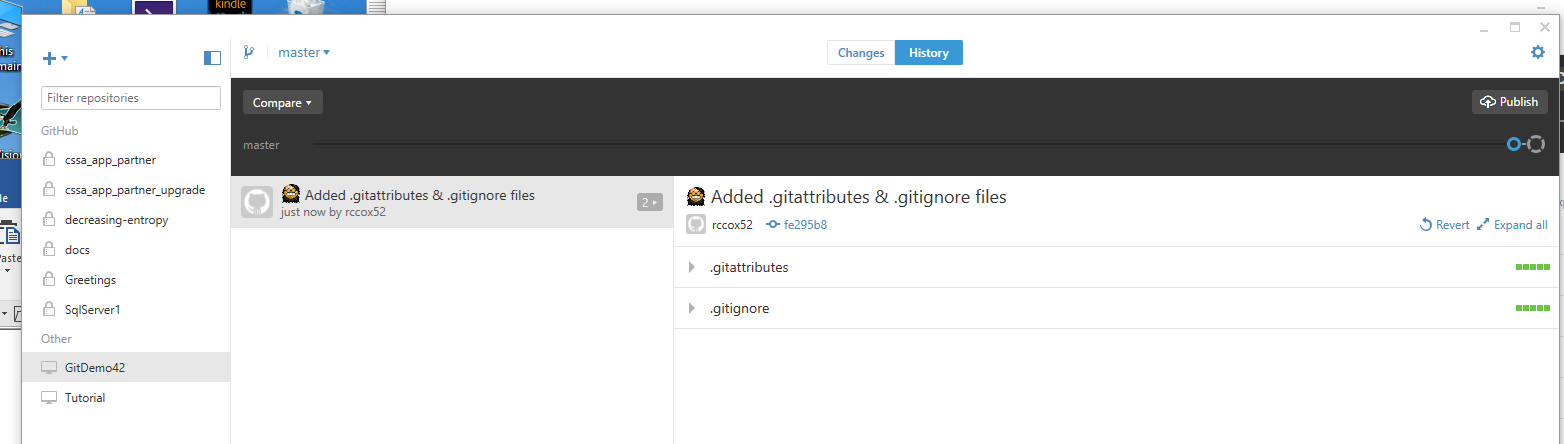


Figure 4-New GitDemo42 Repository Created

Cursor.Pointing(GitDemo42) 🡪 Button.Click(RightMouseButton) 🡪 Command.Click(Open in Explorer)

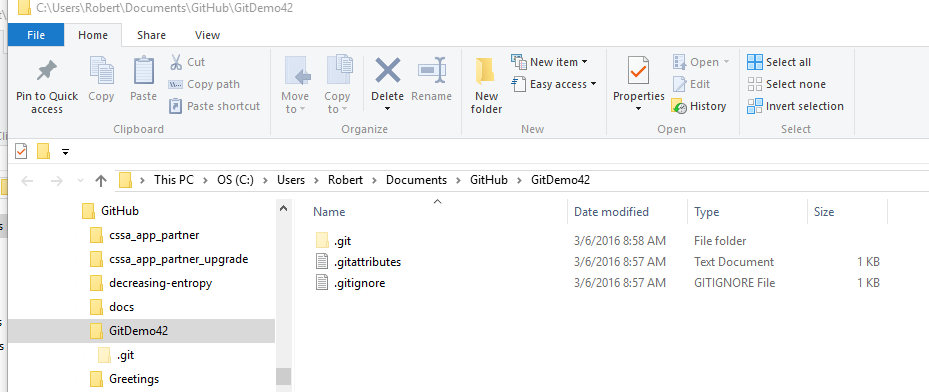


Figure 5-Quick transition to local git repository with Windows Explorer

A new git repository (with the default git infrastructure) has been created at the user-specified location. The windows folder is now ready to use as a git repository, and the GitDestop app tracks its location and status. GitDesktop shows the new repo in the “other” section of the repo list since it does not exist on the GitHub remote site. Also, the “master” branch shows:

* Commits that have been created -- just one so far and the Blue Open-Center circle icon indicates that the commit has not yet been pushed/published to the remote on GitHub; plus the presence of the Publish button in the upper right indicates that the local repository commit/commits have never been pushed to the remote. The changes committed to the local repository only exist within this local repository.
* Status of the local work directory – the larger four(2) segmented circle icon represents the content of the local work directory. When this icon is clicked, the GitDesktop state will be updated to display status info for the local work directory relative to the local repository.

## Repository Clone Operation

The clone command is operates relative to a remote repository from GitHub and the user’s local work environment. A local repository is created, and the content of the remote repository is copied to to the local repository, resulting in an exact/cloned copy of the remote repository.

The user’s interaction with the command includes:

* Clone Button – selection of the clone command highlights/enables the Clone button
* Remote Repository Panel – the left panel provides a list of the accessible repositories on GitHub. The example capture shows the CSSA repository and the separate rccox52 repository. Additional repositories could be included in the list depending on the user’s access to various repositories on GitHub.
* Clone Repository – after a GitHub repository has been selected, the button is enabled and can be used to execute the git clone command, creating a local copy of the specified remote repository within the local file system.

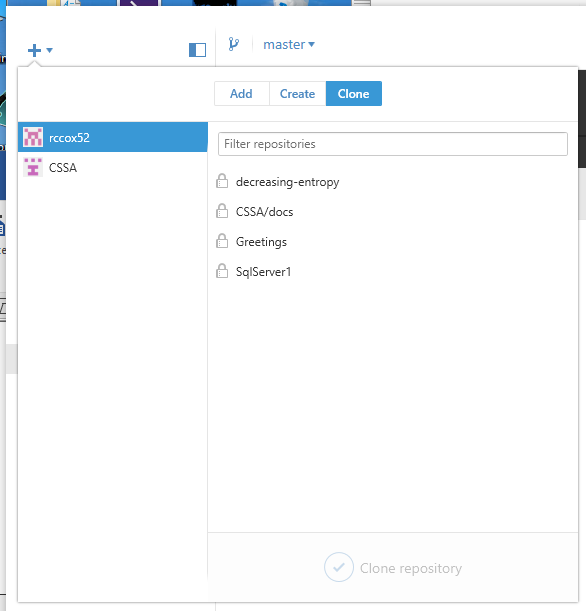


Figure 6-Repository Clone Command

## Create Branch Operation

After clicking on the Add Branch icon, the Create New branch dialog, as shown in Figure 2-Create New Branch Dialog is presented in which the user can specify the parameters of the new branch, then click “Create new branch” to execute the operation.

* “Name” – enter a user-defined, user meaningful name for the new branch
* “From branch” – from the combo box, the user must select one of the branches that have already been created.
* After selection of parameters, the “Create new branch” button is enabled.
* When the “Create new branch” is clicked, a new git branch is created within the local repository with no immediate impact to the remote repository. (i.e., the branch is created locally, but it has not been pushed/published to the remote)

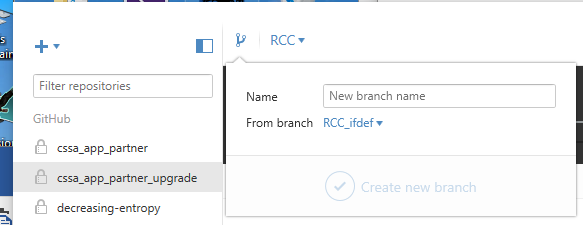


Figure 7-Create New Branch Dialog

## Branch Selection Operation

The branch selection and identification icon is shown within the row of icons, to the immediate right of the Create Branch icon. The icon clearly identifies the currently selected branch by name (i.e., in this instance it is “master”) as illustrated in Figure 2-Branch Selection & Identification Icon. The example in the screen capture shows the “master” branch since it is currently selected. Point to the icon and Button.Click(LeftMouseButton) to see the drop-down list showing the existing branches from which the user can choose.

Selection of a different branch from the list will cause the user’s local repository to switch to the specified branch and the GitDesktop display will be updated to show the new branch selection. Note: in some cases, the contents of the user’s local work directory may contain modified files whose changes would be lost if the branch change is executed immediately. Don’t worry. Unintentional overwriting/discarding unsaved local changes is prevented. Notification of the condition is shown to the user, after which the user must decide how to proceed before switching branches (i.e., save or discard local changes).

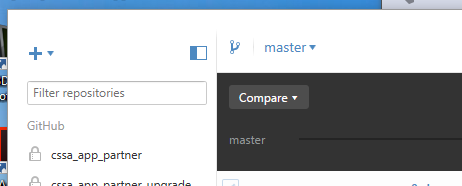


Figure 8-Branch Selection & Identification Icon

## Changes/History View Selection Operation

At the middle of the GitDesktop window, along the top edge, GitDesktop presents two buttons that show the current viewing selection in addition to providing the command option.

Changes Button – click this button to view the local git repository status relative to the local repository and the remote repository

History Button – click this button to view the historical record of commits recorded within the local repository and the status relative to the remote repository.



Figure 9-Branch History Graphical Presentation

The branch history is presented graphically within the horizontal stripe with the branch label on the left side. The large segmented circle icon on the far right represents the current work directory while the blue circle icons (only one in this example) identify the commits that have been recorded to the repository for the named branch (i.e., only one branch is shown “master”). The blue circle icons that have the open center indicate a commit that does not exist on the GitHub remote. A Publish or Sync operation is required if the user wants to copy the commit to the remote.

A more interesting example is shown for the Greetings repository, with a sequence of commits identified in the left sub-panel and summary of the changes listed in the right sub-panel for the currently selected commit (i.e., these are the panels shown just below the black strip, the “Branch Graph”, that illustrates the branches and commits with their sequencing relations) . The simple example shows four (4) commits listed within the left sub-panel and graphically rendered within the “Branch Graph”, three of which have been pushed to the GitHub remote one after the other (shown with closed circles), but the last commit (the large blue icon with the hollow center) has not yet been pushed to the remote, thus the changes only exist within the local repository. The right panel indicates that three files have been modified and committed to the local repository as a part the currently selected commit.

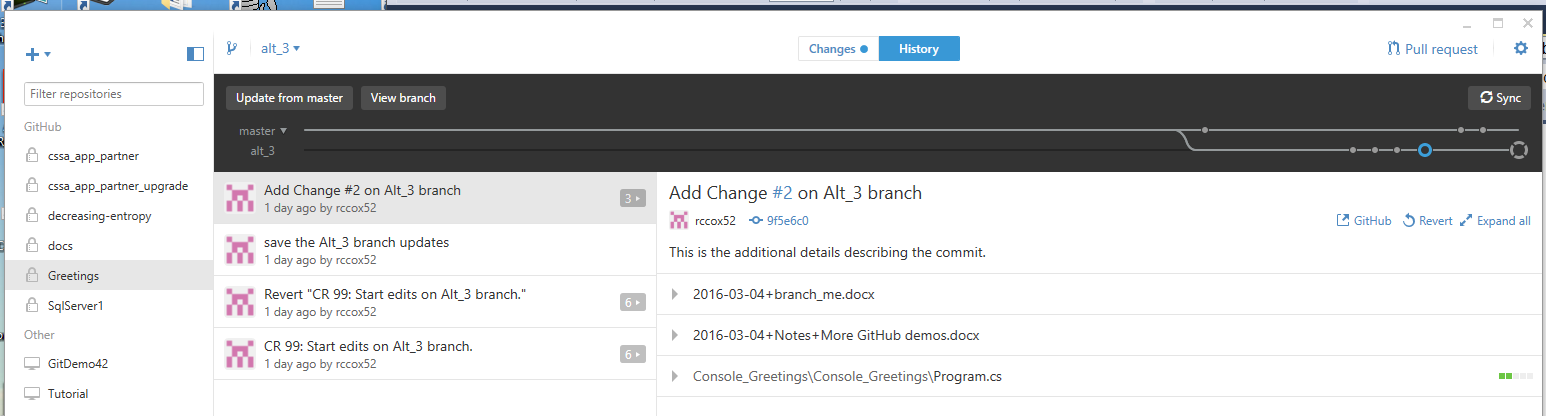


Figure 10-Presentation of Commit Information

The numbered icon next to the commit item within the left sub-panel indicates how many files were modified as a part of that specific commit. Clicking on the numbered icon expands the view to show the full list of modified files, after which selection of a specific file will update the right panel to show the changes within that particular file. The presentation of changes/differences is very easy to read, boiling down to a combination of simple deletions and additions. A change within a single line will be reported as the deletion of the “changed” line, followed by insertion of the new line. In the exampe, it clearly shows that two lines of new text were inserted at line 32. The example text was entered while operating on the alt\_3 branch, with manually entered text indicating that I was entering my second code change, change #2, on my alt\_3 branch. After making the commit to Git, I can see that the tool is recording exactly what I intended, the commit resides on the alt\_3 branch and it records the two lines of new code inserted into my simple console application. With a series of experiments with the simple code and the repository, I can convince myself that the tool behaves as I would expect, recording my changes on the alt\_3 branch as long as I continue to work on that branch.

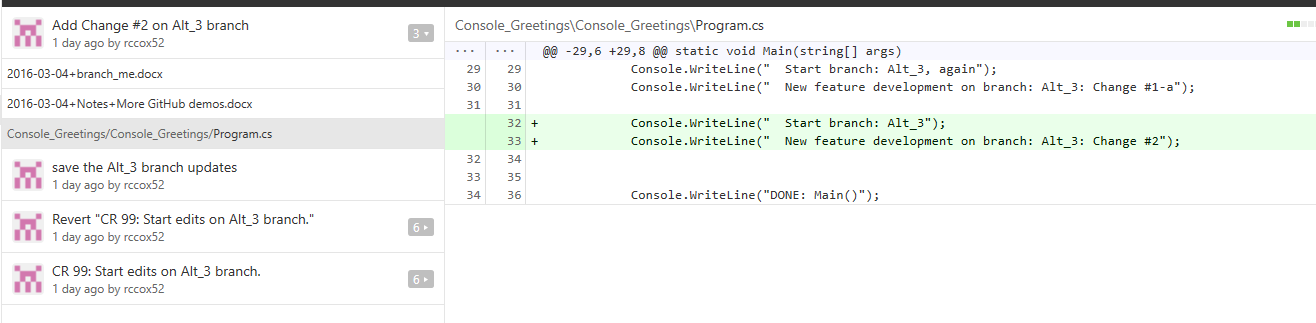


Figure 11-Identification of changes within a commit

## Changes View Operation

GitDesktop also makes it easy to identify changes to files within my work directory. Instead of viewing the commit history, I can switch viewing modes in one of two ways:

* Changes Button – click on this button to change to the “Changes View Mode”
* Work Directory Icon – click on the last icon on the right (i.e., the large four-segment open circle) of the “Branch graph” to change to the “Changes View Mode”

In the “Changes View Mode” illustrated in Figure 7-Changes View Mode, GitDesktop will highlight the files that contain changes that have not bee committed to the local repository.

For the selected file from the left sub-panel, the changes that are contained within the selected file are shown in the right sub-panel.

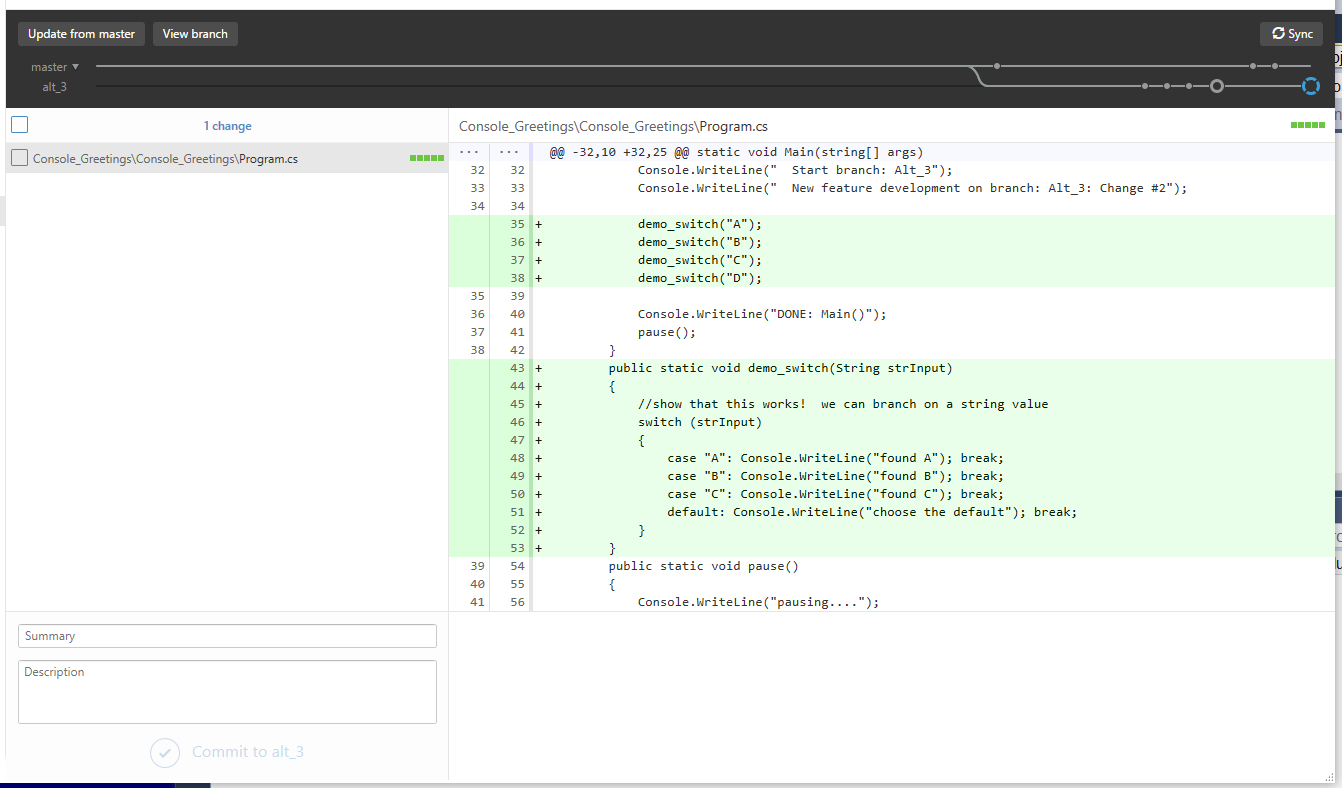


Figure 12-Changes View Mode

## Commit Changes Operation

In order to commit changes to the local repository, the user must:

* CheckBox – enter a check in the “changed files” that are to be copied to the repository as a part of the commit operation. BEWARE: One should examine the list of checked files carefully and ensure that ONLY the ones desired to be a part of the commit operation have checks. It is easy to have ALL files checked by default   
  Summary – enter a summary text description for the commit
* Description – enter a more detailed description for the commit
* Commit to “BranchName” – after entering information into the Summary/Description text boxes, the “Commit” button is enabled and the changes can be recorded to the local repository.

NOTE: This operation has no immediate impact on the remote repository. The local repository can be updated any number of times without impacting the remote repository in any way. Coordination with the remote comes into play when changes are Published/Synced with the remote.

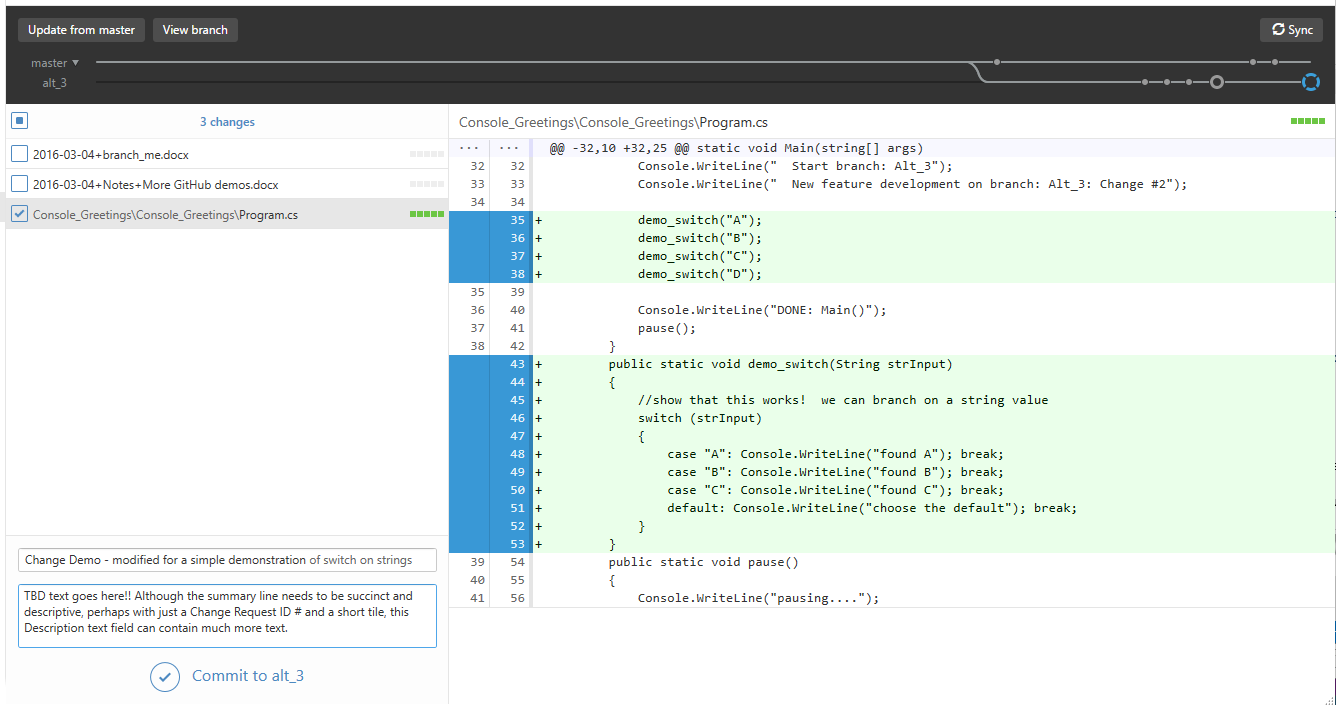
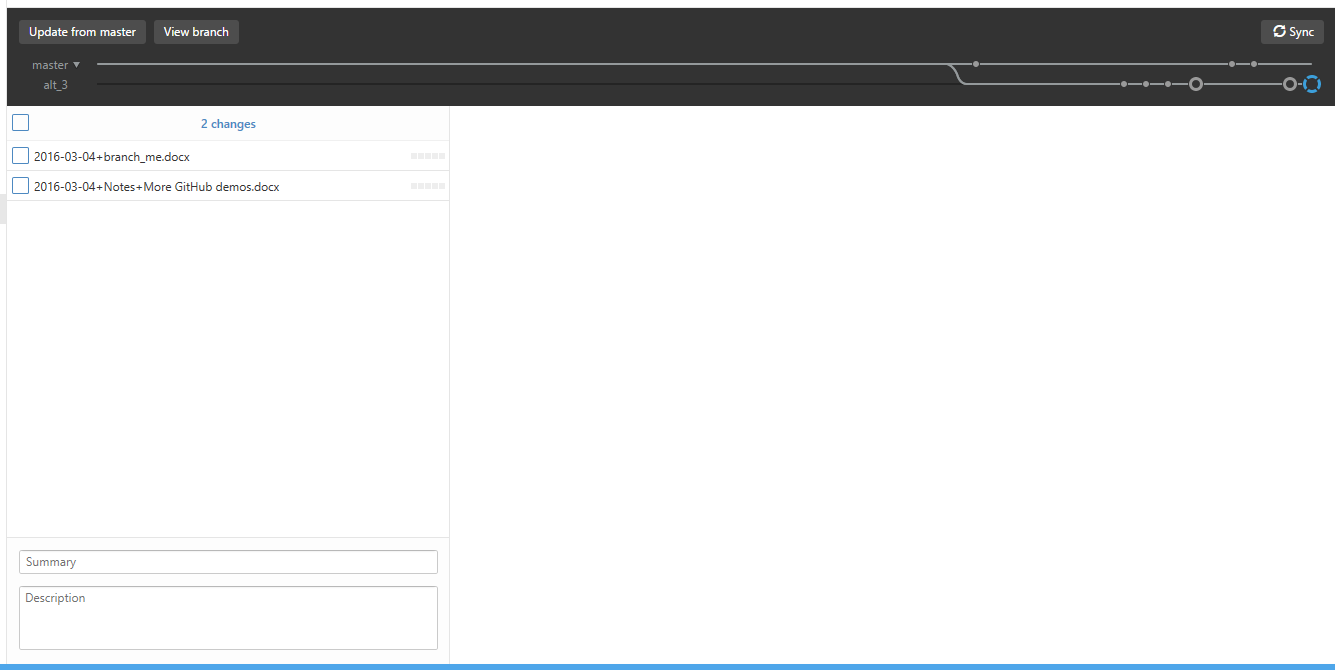


Figure 13-Change Demo

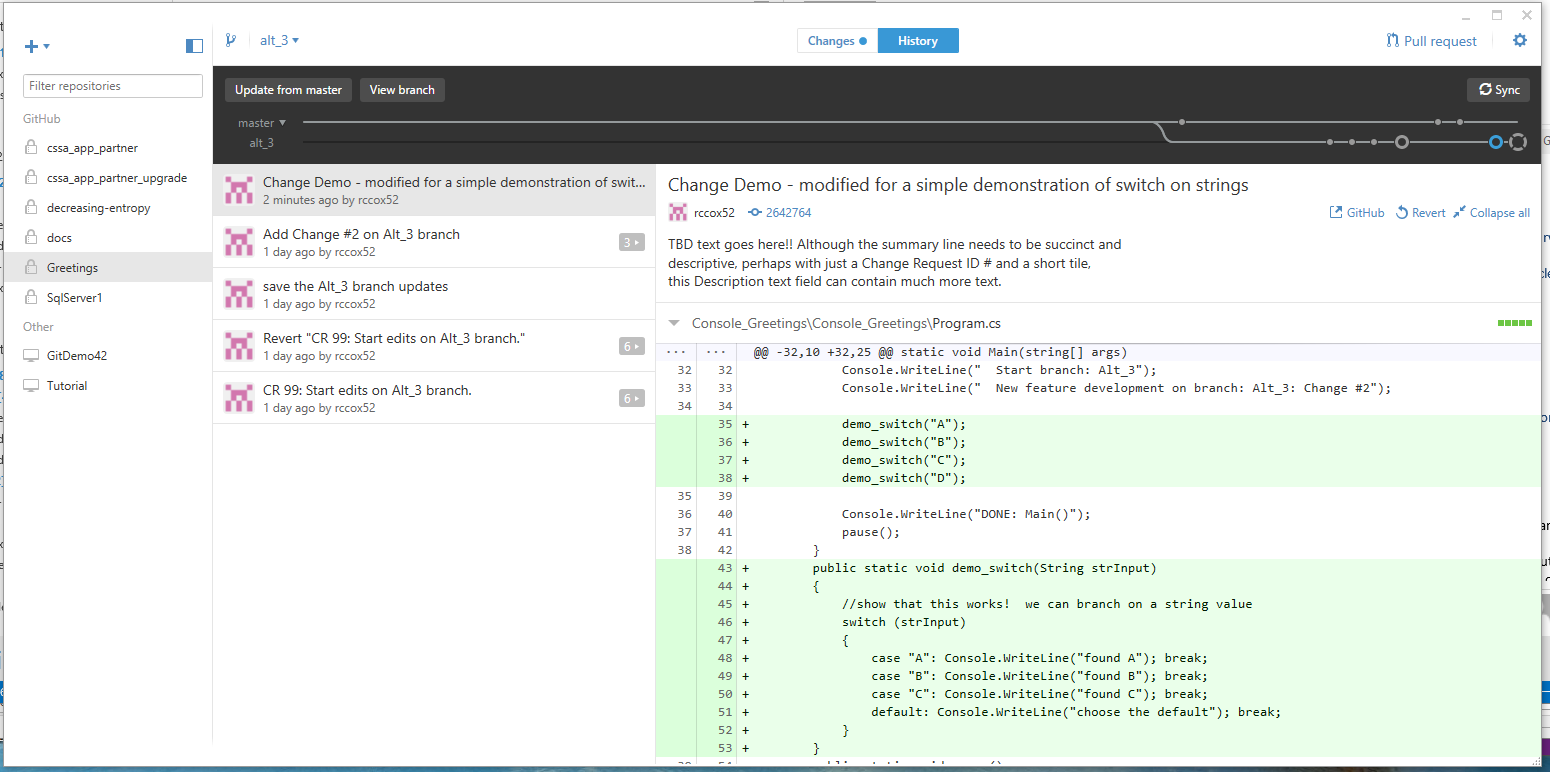
The Commit button is now enabled for the change on branch alt\_3.

Button.Click(“Commit to alt\_3”)



After the commit, the remaining two changed files are still in the list of changed files since they were not included in the commit operation. The branch graph has been updated to show a second commit that has not been pushed to the remote repository.

Button.Click(“History”)



Click on the last commit (open Blue Circle) on the branch graph.

The display now shows the result of having created the commit in the local repository. The latest commit matches the “summary” text that was provided to the commit command. The listing of changes in the right sub-panel match the changes that were shown before the commit. The “Branch Graph” now shows the additional commit“ on the alt\_3 branch. There are now two local commits that have not been pushed to the remote.

## Merge Commits Operation

At some point, it is necessary to consider merging the changes that have been implemented on the separate master and alt\_3 branches. The branch path illustrates that there have been three separate commits on the master path since the alt\_3 branch was forked form the master. The alt\_3 branch includes five (5) separate commits, two of which have not been commited to the remote repository (i.e., the changes represented by those two commits exist ONLY within the local repository.

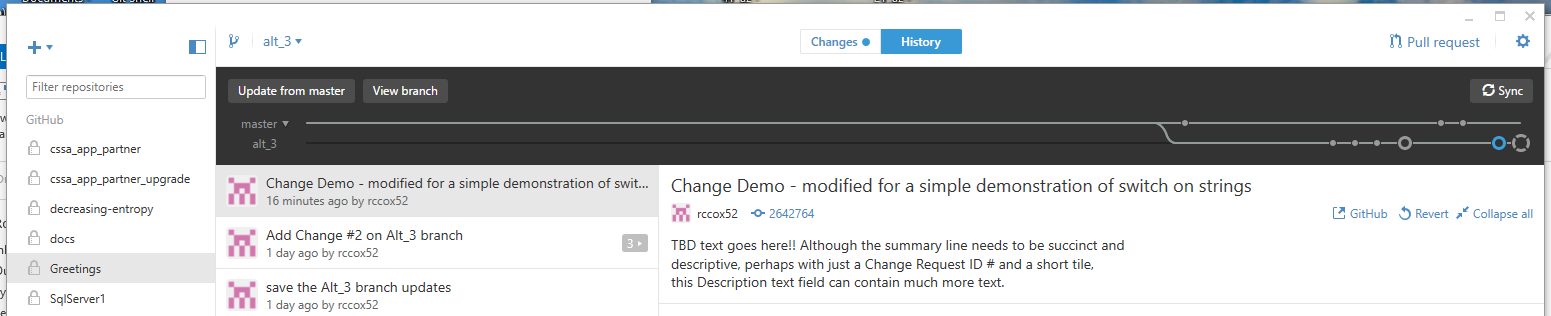
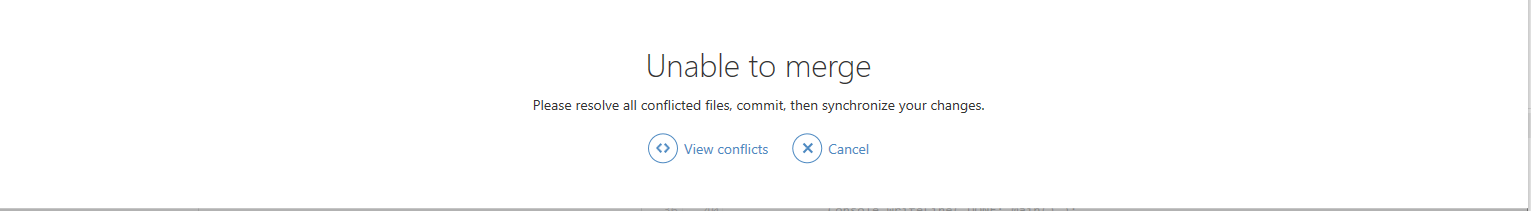


Figure 14-Repository Branch Merging Scenario 1

A preferred approach to executing the merge of changes is to perform the merge on a branch that is separate from the master branch in order to maximize the time that the master branch is in a consistent state. To that end, this example will use the “update from master” command to pull the updates from the two visible commits into the alt\_3 branch.

Button.Click(“update from master”)



The popup indicates some conflicts in the files that have been updated. There were multiple updates on the master branch while work was being performed on the alt\_3 branch.

Button.Click(“View conflicts”)

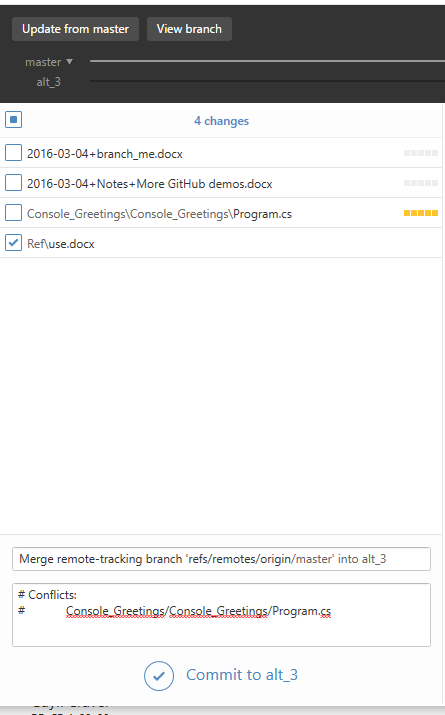
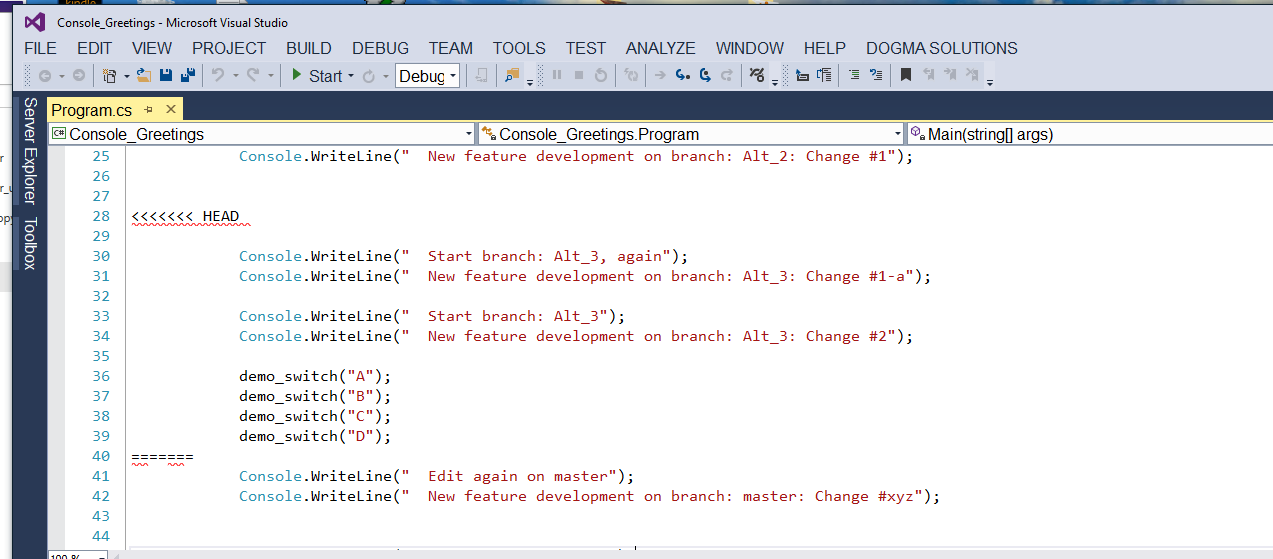


Figure 15-Merge results, including conflicts reported by git

Git marks the **conflicting code** within the listed files. In this example, there is one file, program.cs, that contains conflicts. A manual edit cycle is required to resolve such conflicts. Look for the conflict markers inserted by git.

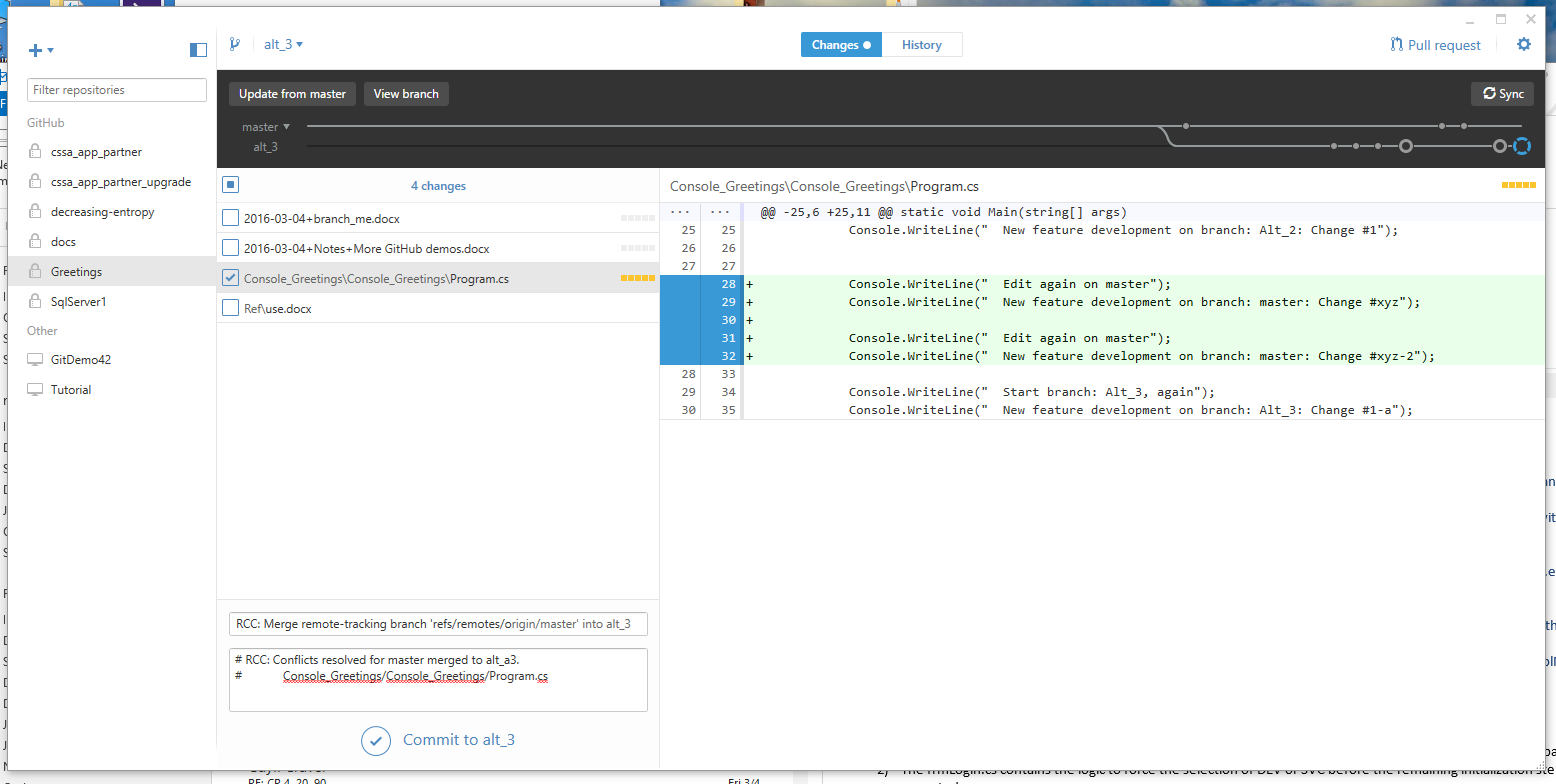
Start Visual Studio to examine the conflicts.



The marker shows changes between conflict markers:

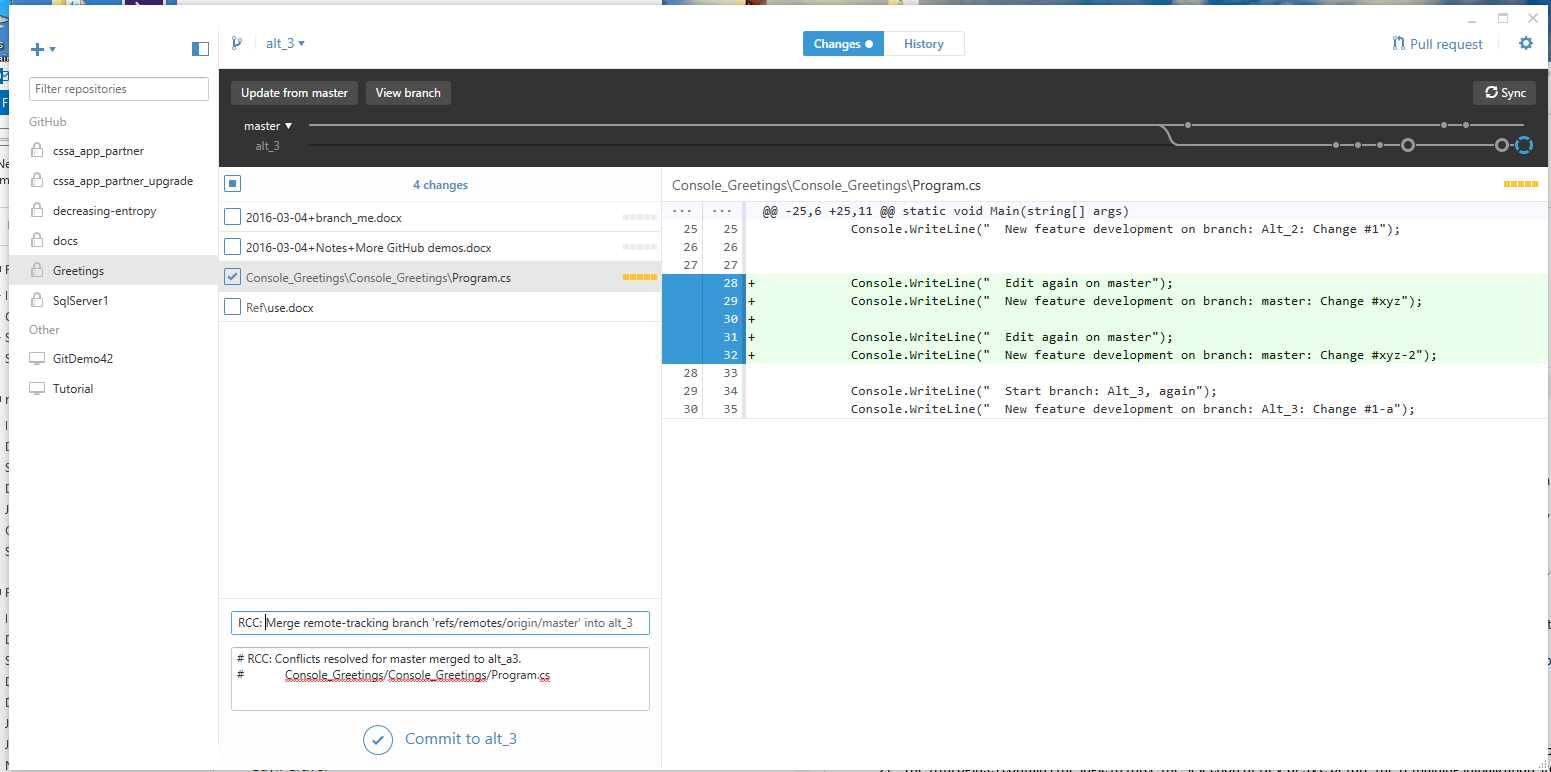
1. “<<<<<< Head”,
2. “=====”
3. “>>>>>>> refs/remotes/origin/master” represent conflicting changes. The user must manually determine which changes should be retained within the merged code since the automated merge process cannot determine which code should be used.

* Perform the manual edits on the highlighted code.
* Rebuild, rerun unit tests & integration tests to determine the level of confidence in the updates.
* Run…. It looks good. The changes all look fine.
* Examine the updates within GitDesktop. All is good.
* Edit the summary comment, as desired.
* Edit the description, as desired.
* Ensure that the checkboxes within the left sub-panel only include the files that we want to commit to the repo.
* Double check the status.
* Try to build, run, test the new code configuration.
* If it looks good, proceed to the next step.



There are still no permanent changes within the local repo yet. Nothing has been committed to the local repo. Both master and local alt\_3 branches are unmodified.

Button.Click(“Commit to alt\_3”) 🡪 save changes into the local repository.



During the commit operation, GitDesktop provides some animation showing the updates to the “Branch Graph” in progress. The result of the merge is illustrated inFigure 16-Merge results, master branch into alt\_3 branch, with manual conflict resolution .

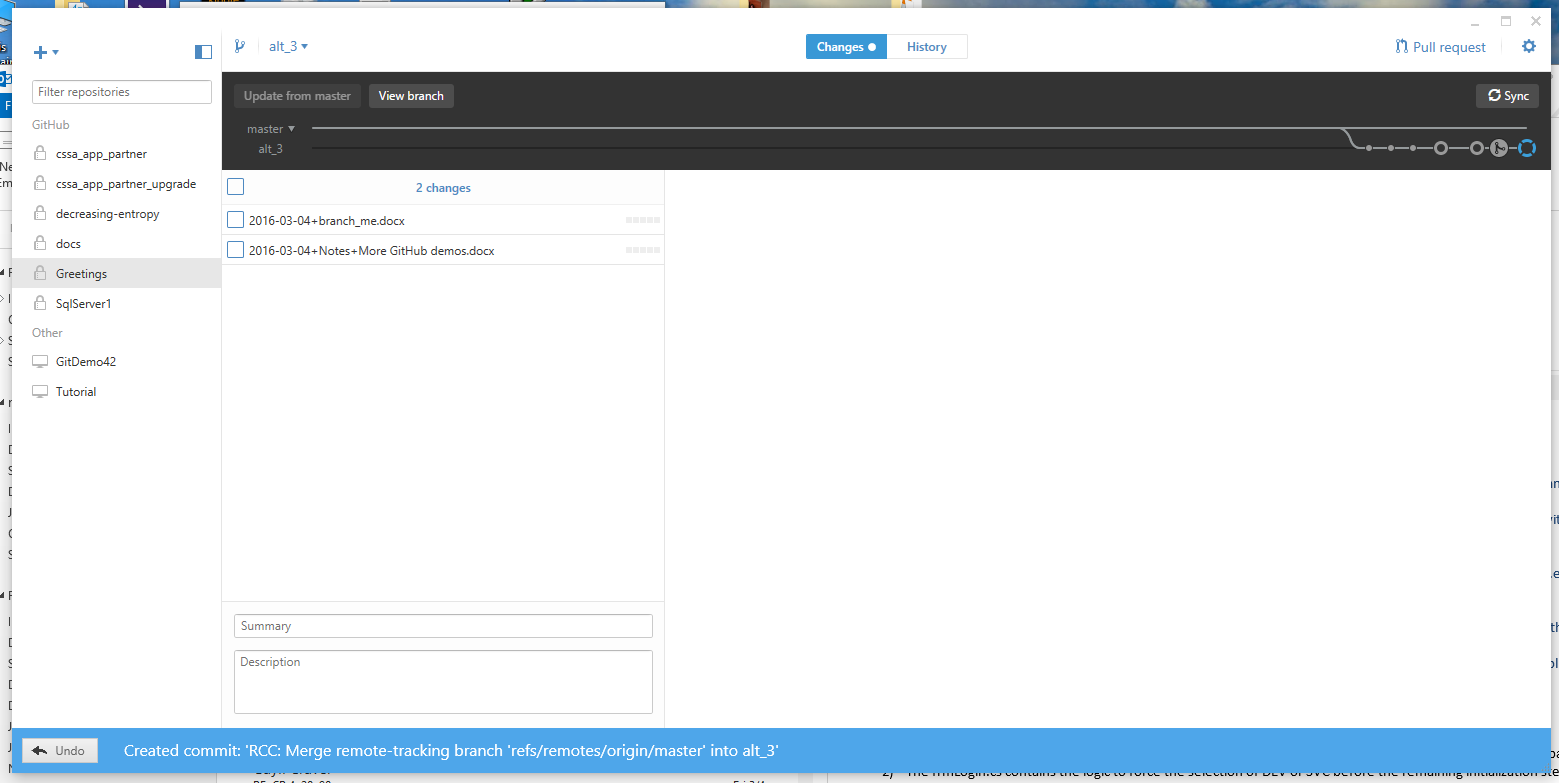


Figure 16-Merge results, master branch into alt\_3 branch, with manual conflict resolution

One should notice that the “Branch Path” clearly shows that the alt\_3 branch commits still have not been integrated into the master branch.

Point to the new “merge icon”, , on the alt\_3 branch and click to select. The display panels now show the commit history, including the merge and the content of the merge, as shown in Figure 12-Merge results from master to alt\_3.

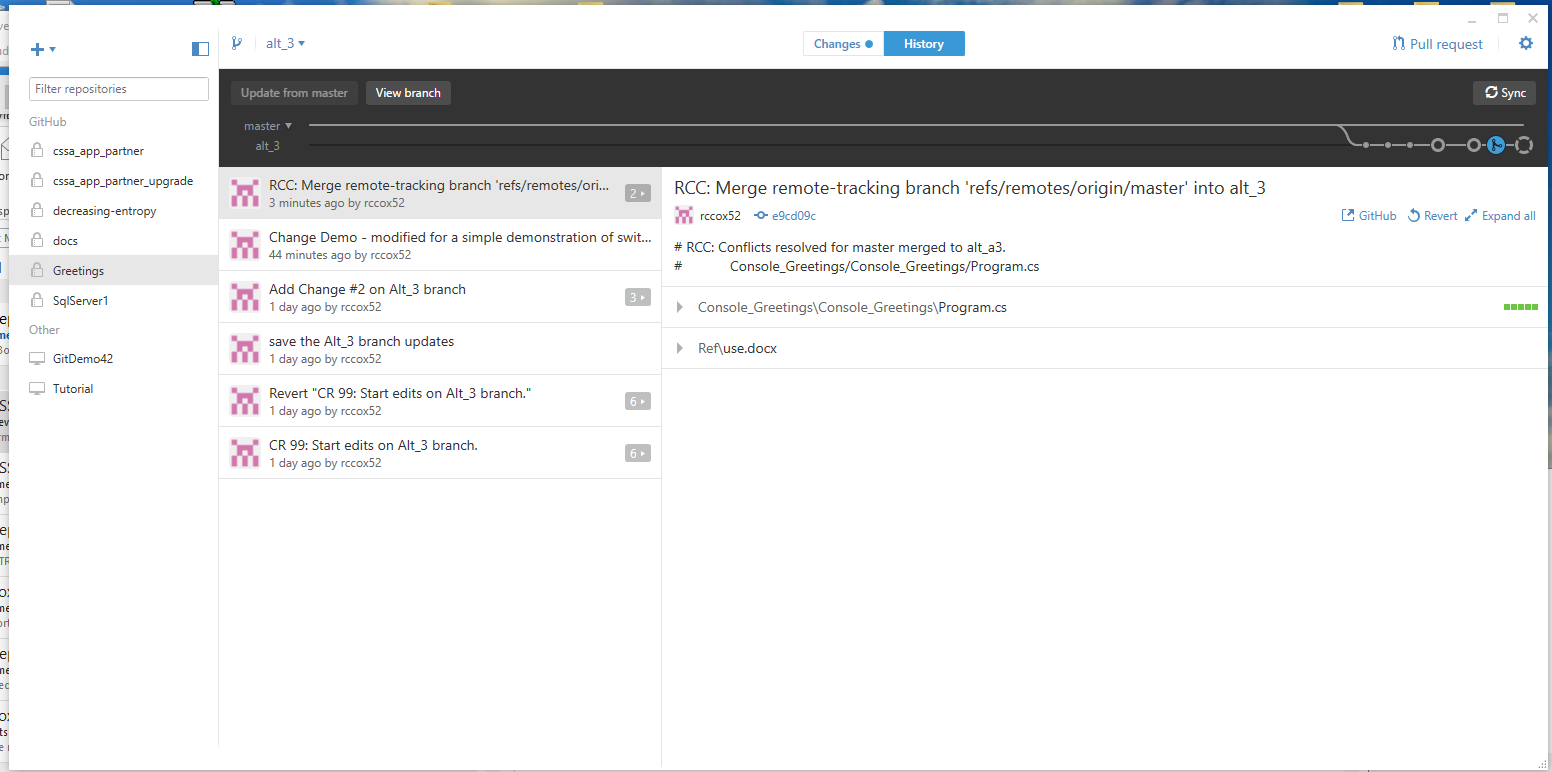


Figure 17-Merge results from master to alt\_3

Everything looks OK.

All results of this effort remain local, merged into the local repository as a part of the Alt\_3 branch.

Now that this merge is complete, double check the status.

* Try to build, run, test the new code configuration.
* If it looks good, proceed to the next step.

## Publish/Sync with GitHub

Now we have a complete and consistent set of updates on the integrator’s local “merge” branch.

* In visual studio, clean the solution, rebuild everything, run some sanity checks (*its not that I am paranoid, its just because everyone is out to get me*?)
  + MenuBar 🡪 Project 🡪 Clean Solution
  + MenuBar 🡪Project🡪Rebuild Solution
  + Run some sanity checks.
* If all looks good, we can push the updates to the remote on the same named “merge” branch (this still keeps it separate from the master branch until the decision is made to merge with the master)

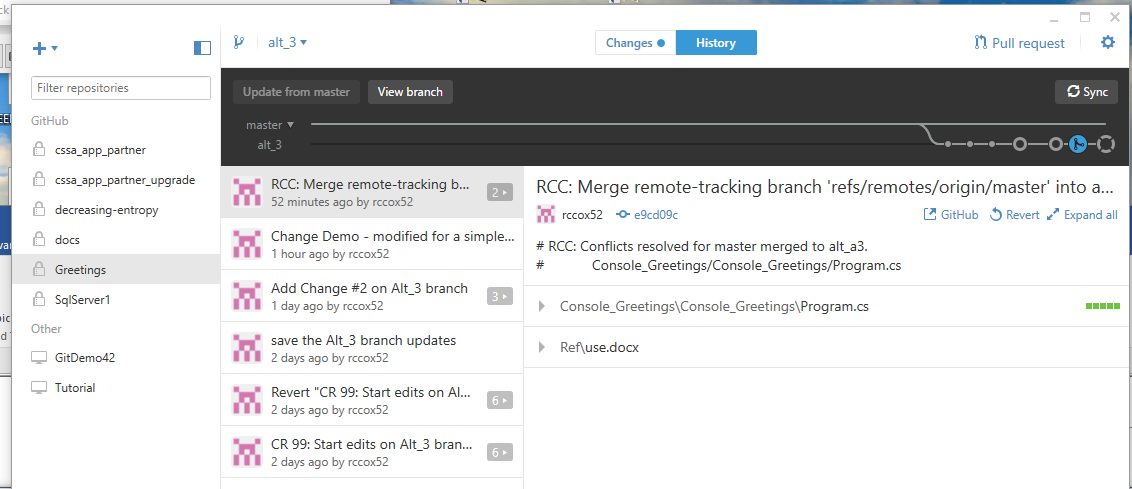
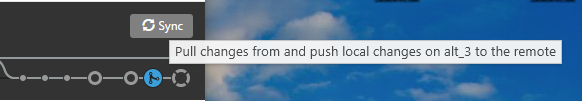


Figure 18-Ready for Synchronization of the local repo with the remote repo on GitHub

* Identify the “Sync” button at the right each of the Branches Graph. Checkout the hint text when pointing to the button. 
* Click.Button(Sync Button) 🡪 Synchronize the commits between the remote and the local
  + Examine the changes in the status, as displayed within the Branch Graph.
  + 
  + All of the commit icons, except for the last one on the right, are now closed circles, indicating the commits have been pushed to the remote. The last commit on the right is now drawn using the unique “Merge” icon: . We can now see that commit contents within the alt\_3 branch of the local repo are now synchronized with the remote repo.
  + Other authorized users of the CSSA GitHub site should now be able to access the content of the merge branch (i.e., alt\_3 in this demo). The reviewers and testers can pull a copy of the branch contents to their local repos simply by switching to that branch. Code review and test can proceed from that point, with the amount and duration of code review/test dependent on timely feedback from management and peers.
* Notify potential peer reviewers and management that the revisions are available on the “merge” branch available from GitHub
  + Provide the name of the merge branch
  + Coordinate with management and peer reviewers to determine how much time should be allocated for the code review for this specific merge and test cycle before merging the results to the master branch OR merging with a “Release candidate” branch.
  + Obtain management decision regarding merge to master vs. merge to a release candidate branch.
* Handoff to integration test: NOTE: In order for the revised code to be readily available to third parties for code review/inspection/test, it must be published/sync’ed to a branch on GitHub. Look carefully at the icons used within the “Branch Graph” to determine if the local repo updates have been pushed to the remote on GitHub.
* Integration test should run their independent suite of the new code configuration, to the extent directed by management for the merge. Note: there may be variations in the amount of regression and duration that seems appropriate for a given merge cycle, so coordination with peers and management regarding suitable duration and effort is recommended.
* If it looks good, proceed to the merge to master operation.

## Merge to master Operation

The actual merge to master operation is not very complicated:

Switch to the master branch.

Point to the “Switch to a different Branch” icon (e.g., we are currently on alt\_3, so we need to switch to the master branch).

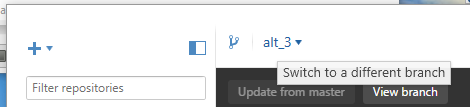
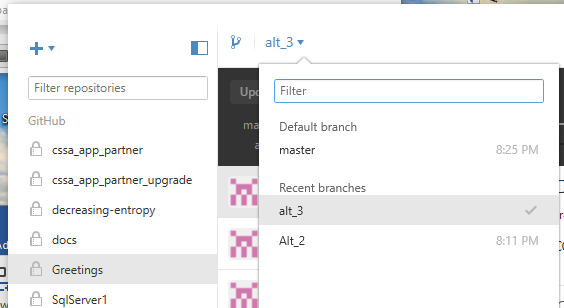
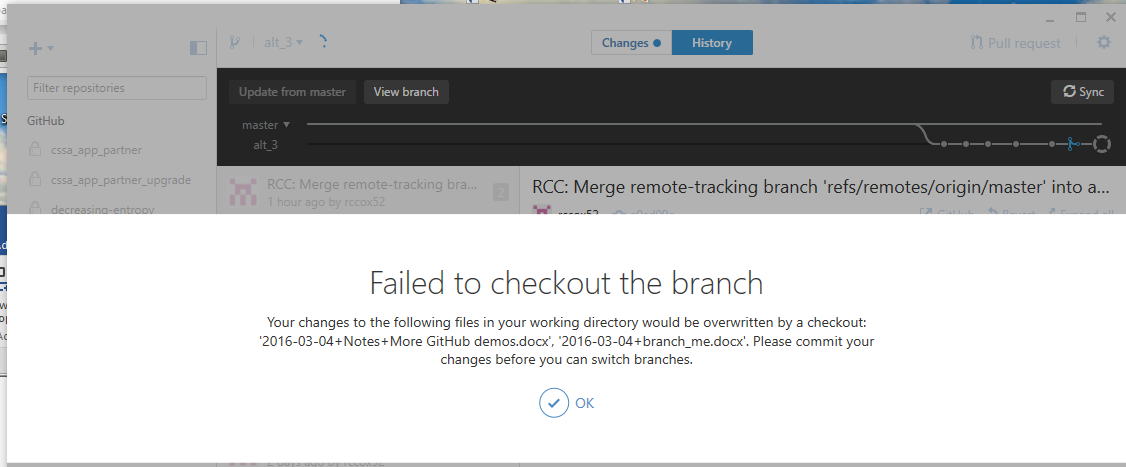


Figure 19-Switch to a different branch option

Button.Click(alt\_3 in this case) 🡪 examine the drop down menu, then choose the master branch.



The following is an example of a failed transition to the different branch. I have some “notes” documentation that will be overwritten by the branch transition, so the tool warns of the impending data loss and prevents the action from completing. Manual intervention is required to decide what to do with the conflicting files: 1) delete them; 2) copy them to some other location to be retained; etc. In this instance, just move them out of the git repository folder.



Done. The offending files have been moved, so retry the switch to master. The switch to master now completes as expected.

Switching back to the master branch shows a number of previous demonstration operations performed with this simple console application.

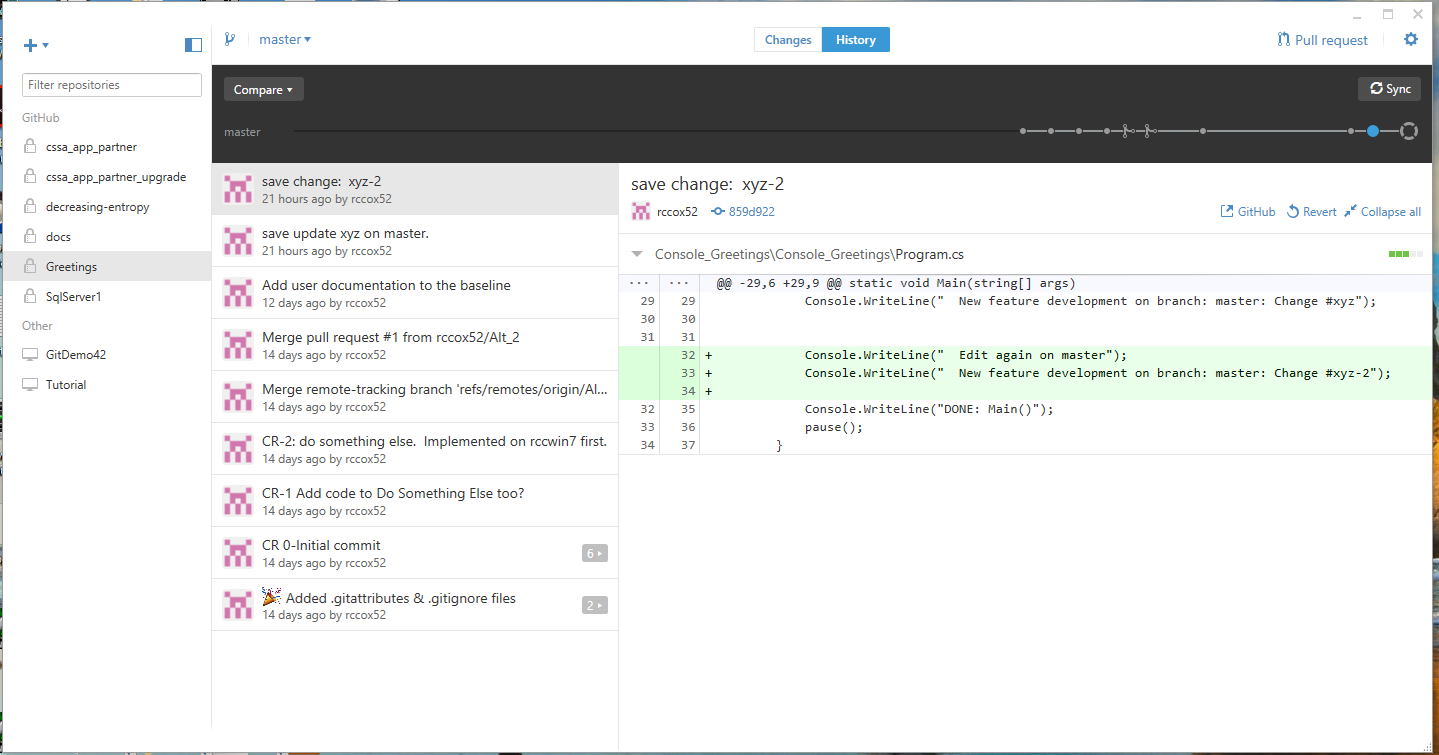


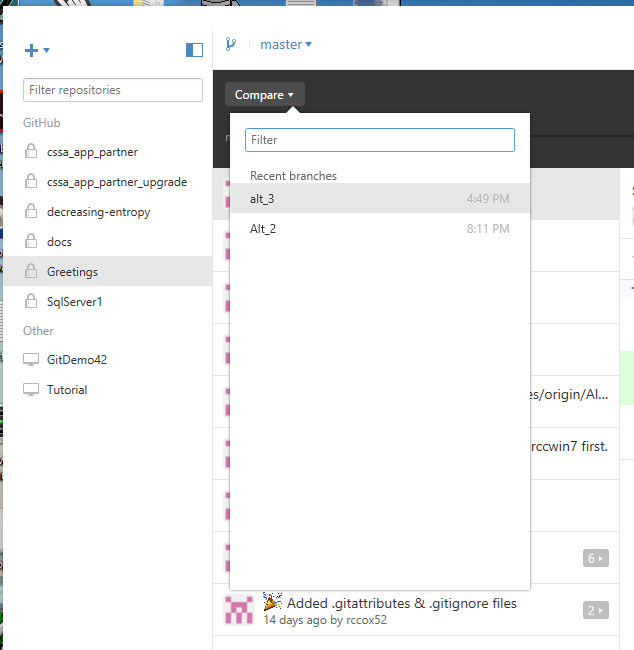
Figure 20-After switching back to master branch

It may seem a bit confusing since the revised display shows no evidence of the continued existing of the alt\_3 merge branch directly within the current display, but alt\_3 still exists and we can access its contents.

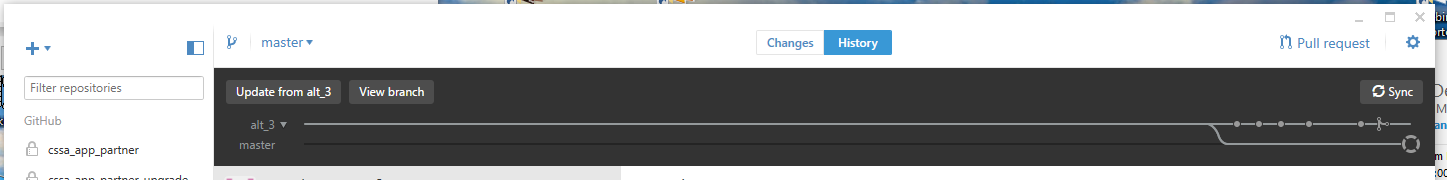
Observe the presence of a new button in this display, at the left side of the Branch Graph.

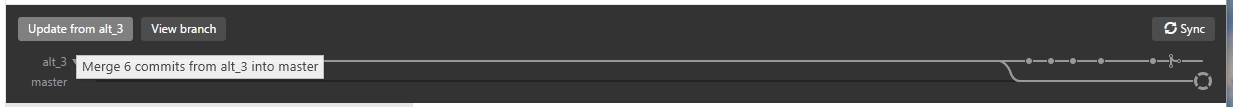
The Compare button can be used to compare the current branch (i.e., master) vs. other branches (e.g., alt\_3).

Button.Click(Compare button) 🡪 observe the popup menu with the option to choose from other existing branches. Pick the alt\_3 branch that we were working on, where the all of the commits have been merged and conflicts have been resolved.



The updated display shows new options, including a graphical representation of both master and alt\_3 within the branch graph, side-by-side. Everything looks like we left it.

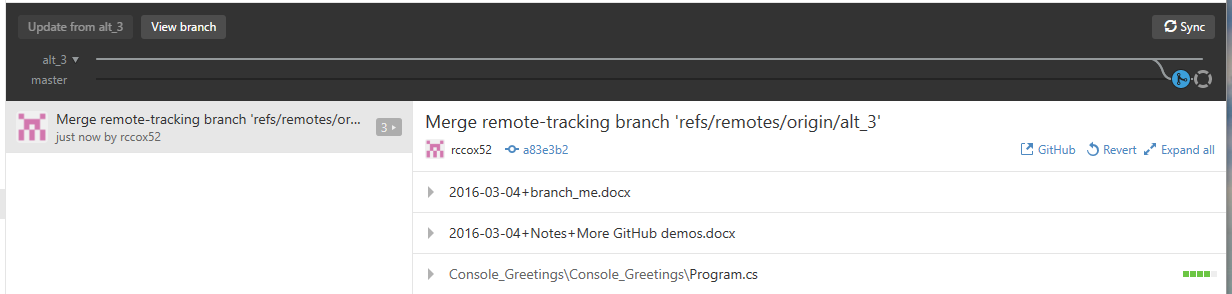




Point.To(Button update from alt\_3) 🡪 observe the tool-tip-text shows we can execut a command to merge six commits from alt\_3 into master. Inspection of the example Branch Graph shows that we have precisely six commits on alt\_3 that should be merged into master. All is good.

Button.Click(Update from alt\_3)

Once again, the merge process is animated, with the result being as shown.

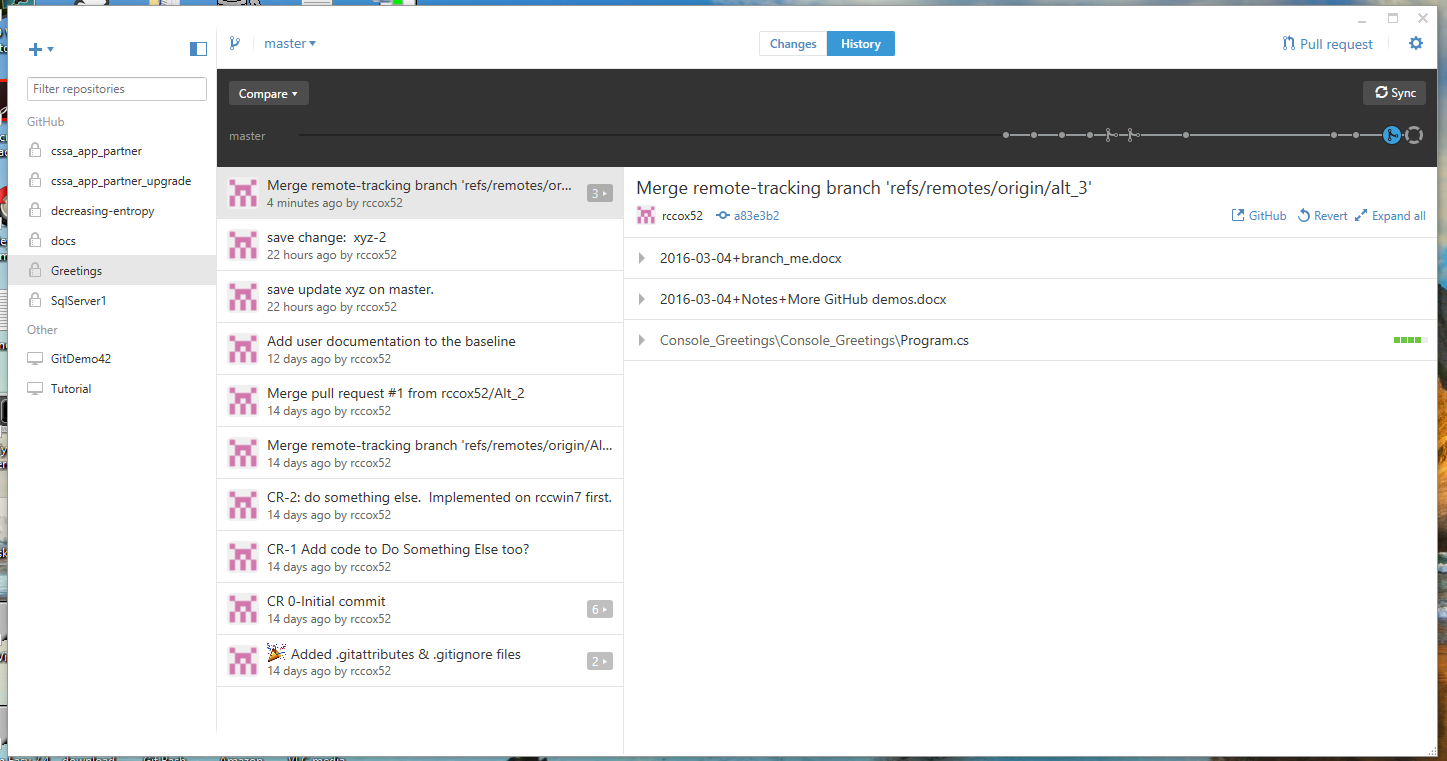


The individual six commits were combined into one “merge commit” on the master branch.

Once again, the merge commit uses the special merge icon: , as now shown on the master branch.

Using this basic paradigm, the content of any branch can be merged to any other branch.

We can now go back to the master branch view and ignore the alt\_3 branch.

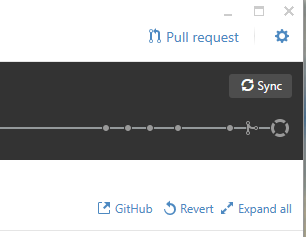


After having merged alt\_3 into the master, at some point in the future we may even decide to delete the alt\_3 branch since master now includes everything after the merge.

## Revert Operation

There may be occasions in which one determines that a change that has already been implemented and committed to the repository has some undesirable properties that should be eliminated. Since Git keeps everything indefinitely, one does not delete commits. However, there is a Revert command available that will generate a new commit that reverses the effect of a specified commit to “revert” the repository to a previous condition.

The Revert command is accessible from the icon on the right hand side of the right sub-panel.



# Example GitHub operations

TBD: Provide a limited description of GitHub features and operations since this could be/has been the topic of entire books (e.g., Introducing GitHub).

# Example git command line operations

TBD: Provide a limited description of basic git command line features and operations since this could be/has been the topic of entire books (e.g., [3] Git Pocket Guide).

Some basic commands of interest:

* git init - create and initialize a new “local” git repository
* git status – check the status of your local work directory, local repository, and
* git add – add one or more files to the list of files that are being tracked by git for a potential future commit to the local repo (Note: there is **no impact** to the local repository at this point)
* git commit – copy a collection of changes into the “local” repository (note: there is **no impact** to the remote repository at this point. Everything could be discarded/deleted and nobody will ever know that some potential changes were being considered)
* Comparison note: the GitDesktop, with its graphical interface, effectively combines the “git add” and “git commit” operations into a single atomic “GitDesktop commit” command for greater ease of use.
* git log – obtain a listing of commits entered into the repository
* git diff – identify the file differences between commits from the repository (Note: The GitDesktop services provide the same operations, but with a much more convenient graphical interface)

# Conclusions

The example usages listed in this paper illustrates almost all of the GitDesktop operations, including illustration of the icons and buttons as they are presented in the GitDesktop interface and the commands that are performed by the tool when the items are clicked.