This paper was originally presented at the Southwest Fox conference
in Gilbert, Arizona in October, 2015. http://www.swfox.net

Version Control Faceoff - Git vs Mercurial

Rick Borup
Information Technology Associates, LLC
701 Devonshire Drive
Champaign, IL 61820
217.359.0918
rborup@ita-software.com

Git and Mercurial are both popular Distributed Version Control Systems (DVCS), but it seems like Git is getting all the love these days. What are the real differences between the two? Is one better than the other? Which one should you choose? What's all this fuss about GitHub? How can the new FoxBin2Prg project on VFPX help you use a DVCS to full advantage with VFP? Come to this session for practical advice to improve your version control skills as we explore and discuss answers to these and other questions.

Introduction

The purpose of a version control system is to keep track of changes to files over time. Version control systems are most often used to track changes to software source code, but can be used with any type of file. For example, I'm using a version control system (Mercurial) to track the changes to this whitepaper as I write it.

Version control systems are sometimes referred to by other names such as source control or revision control. For our purposes, those terms all mean the same thing.

Git and Mercurial are two of the most popular distributed version control systems in use today. They share many similarities, but are different enough that knowing one does not guarantee you can switch seamlessly to the other. This paper introduces both Git and Mercurial and provides enough information to get you going in either or both of them, while describing their similarities and differences and linking to other resources to help you advance beyond the basics. It also introduces the FoxBin2Prg project on VFPX and helps you learn how to use it with Git and Mercurial.

How distributed version control systems work

The fundamental concept in a version control system is the repository. The repository is the place where the history of source code changes over time is stored. Depending on the version control system being used, the repository might be a database or it might be a collection of flat files in a hierarchy of special-purpose folders and subfolders. Either way, you don't have to understand the inner workings of your version control system – you only need to know how to use it.

In centralized version control systems, developers check out source code files from a central repository and check them back in again when they're done making changes. Developers must be able to get a connection to the remote server in order to check out a file, and a file cannot be checked out again if it's already checked out by someone else. This introduces two dependencies that can hinder productivity: a developer cannot check out a file if the connection to the server is unavailable, and two developers cannot check out the same file at the same time.

In distributed version control systems (DVCS) like Git and Mercurial, each developer has an individual copy of the repository residing on their own local machine. Developers do not need to check out a file from a central repository to work on it, and any number of people can work on their own local copy of the same file at the same time. Each developer commits their changes to their own local repository, independent of any changes another developer might be making to the same code. Changes made by one developer can subsequently be merged with changes made by another developer, either by exchanging commits directly with one another or by using a shared central repository.

Basic concepts and terminology

The syntax and nomenclature used by Mercurial and Git differ in some places, but conceptually the two systems are almost identical. This makes it easy to describe both

systems at the same time. Here are some of the basic concepts and terminology with which you need to be familiar:

The *repository* (or repo, for short) is where the history of source code changes over time is stored. The *local repository* is the one on the developer's own machine. For any given project, any repository other than the developer's local one is considered to be a *remote repository*. If two developers want to share changes directly with one another, developer A's local repo can serve as developer B's remote repo, and vice versa. Some development teams use this approach while others instead choose to use a shared remote repository accessible by everyone on the team.

Version control systems track files, but not all files in a project need to be tracked. Files that have been placed under version control are referred to as *tracked files* or *versioned files.* Other files in the same project that are not placed under version control are called *untracked* or *non-versioned* files.

Mercurial and Git both provide a way to *ignore* certain types of files, such as executables, that you never want to track.

The folder where the developer's local copy of the source code files is stored is called the *working directory*, and the files themselves are collectively referred to as the *working copy*. In Git and Mercurial, the local repository is a subfolder under the root of the working directory. In Git it's the .git folder and in Mercurial it's the .hg folder.

Shared remote repositories do not have a working directory – they contain only the history of changes, not the source code files themselves. For this reason they are called *bare repositories*. Bare repositories are typically located on a file server or other shared resource. They do not need a working copy because developers would not make changes directly to those files anyway.

When you want to check if there are any new or modified files in your working directory, you ask for the *status*.

After you've modified a file or files in the working directory, you *commit* those modifications to the local repository. Commits are accompanied by a *commit message* describing the changes.

When you want to sync your working copy with a particular revision in the repository, you *checkout* that revision in Git or *update* to that revision in Mercurial.

When you are ready to share changes from your local repository, you *push* them to a remote repository.

When you need to get changes made by others, you *pull* them from a remote repository.

If you want to create a complete copy of an existing repository in a new location, you *clone* it.

Branches are divergent lines of development within a repository. There is always one main branch. In Git it's the *master* branch while in Mercurial it's the *default* branch. Developers can create other branches and give them names.

It's often necessary to combine the changes made to a file by one developer with the changes made to the same file by another developer, or by the same developer in a different branch. This is called doing a *merge*.

If a merge is attempted after two different sets of changes have been made to the same lines of code—for example, by two different developers or on two different branches—a *merge conflict* can occur. Merge conflicts must be *resolved* before the merge can be completed.

Comparison of basic commands

Following is a quick comparison of the most frequently used commands in Git and Mercurial. Many of them share the same name in both systems, although sometimes they behave differently. Others have the same behavior but different names.

Function	Git	Mercurial	Comments
Create a repository	init	init	The repository is created in subfolder .git for Git and in .hg for Mercurial.
Start tracking a file	add	add	Git requires *add* after each modification (called *staging*). Mercurial does not.
List the files being tracked	ls-files	manifest	
Commit a set of changes to the local repository	commit	commit	Both require a commit message
View the history of commits to the local repository	log	log	
Push changes to a remote repository	push	push	By default, Mercurial pushes all branches. Git pushes only the current branch.
Pull changes from a remote repository	pull	pull	In Git, use *fetch* to pull without updating the working copy. In Mercurial,

			use the –u option to update the working copy.
Get the status of the working directory	status	status	Lists new and modified files. Git indicates if the files have been staged.
Get a list of existing branches	branch --list	branches	
Change to a specified branch	branch	branch	
Sync working copy with a specific revision or branch	checkout	update	
Resolve a merge conflict	-	resolve	Git puts conflict markers in the file(s), which must be edited to resolve the conflict.
Create a copy of an existing repository in a new folder	clone	clone	
Show the differences between revisions for specified file(s)	diff	diff	

How to install and get started with Git on Windows

Git's home on the Web is https://git-scm.com/. Git's roots are in Linux, so it runs natively on Linux and Apple operating systems. For Windows you need to install a project called *Git for Windows* from the "Downloads for Windows" link on the Git homepage.

Git for Windows enables you to run Git from the command line on a Windows machine. It also comes with a graphical user interface called *Git Gui*. This paper is based on Git for Windows v2.5.0, which was released on Aug. 18, 2015. Git for Windows implements most but not all Git commands. After installing it, check the release notes at file:///C:/Program%20Files/Git/ReleaseNotes.html for details.

Installing Git for Windows

The Git for Windows installer works like any other installer, but be prepared to encounter a few dialogs asking you to make decisions you might not expect. The first asks you to choose if and how to modify your Windows path in order to use Git. If you want to be able to run Git from the Windows Command Prompt, select the second option as shown in **Figure 1**. This tells the installer to add `C:\Program Files\Git\cmd` to your Windows PATH so you can run Git commands from the command prompt in any folder.

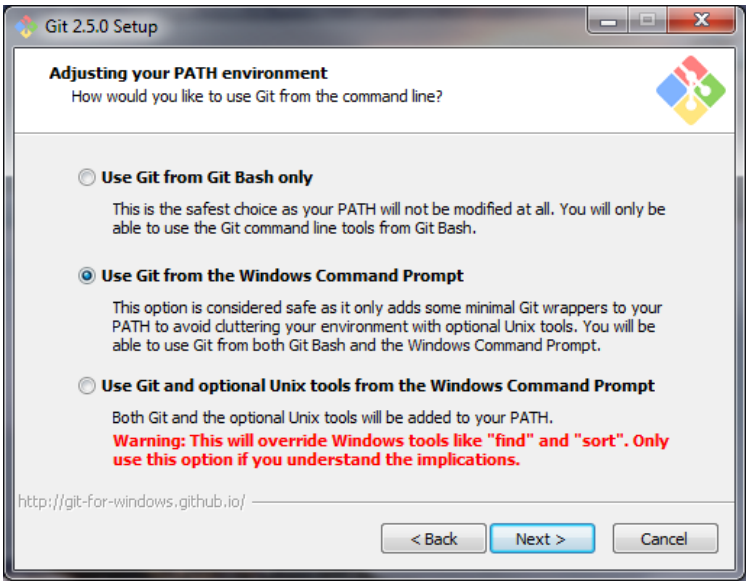

Figure 1. Choose the second option if you want to be able to run Git commands from the command prompt.

The next dialog (**Figure 2**) asks you make a decision about SSH. The first option is the default.

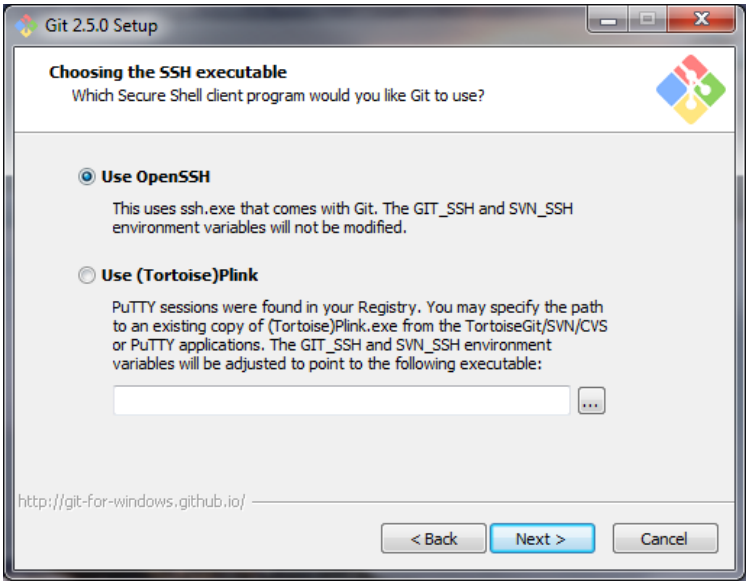

Figure 2. OpenSSH is the default.

The third dialog asks you to choose a line ending option. This can make a difference in a cross-platform development environment because Windows, Linux / Unix, and Apple OS each use different line ending characters. The choices are explained in the dialog shown in **Figure 3**. The first option is the default.

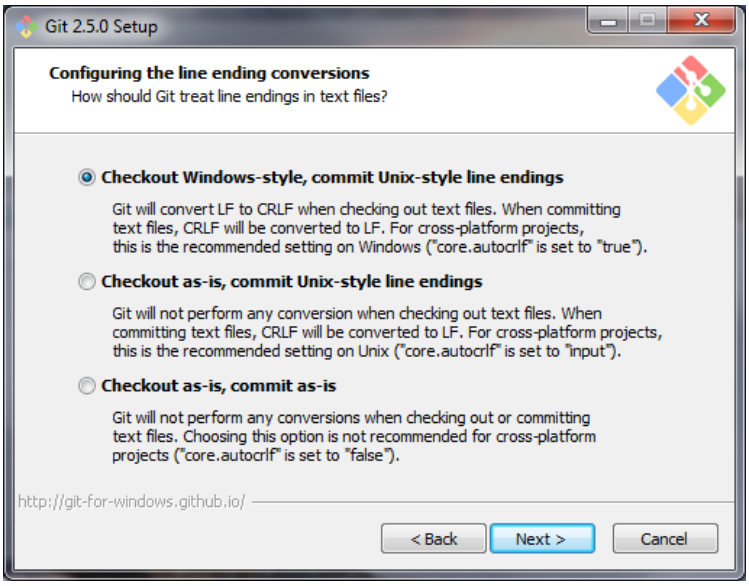

Figure 3. The first choice is suitable for developers working in cross-platform development environments. The third option is OK for single-platform development environments.

The fourth dialog offers a choice between two options for terminal emulation – in other words, how you prefer to use Git from a command prompt. The first option, which is the default, installs an emulator for Git Bash. This offers what the installer describes as a more fully featured experience—better use of colors, more information on the command prompt, resizable window, etc.—but takes some getting used to if you're not familiar with Linux. Choosing the first option does not prevent you from also using Git from the Windows command prompt, so I don't see any downside to it. See **Figure 4** for more information.

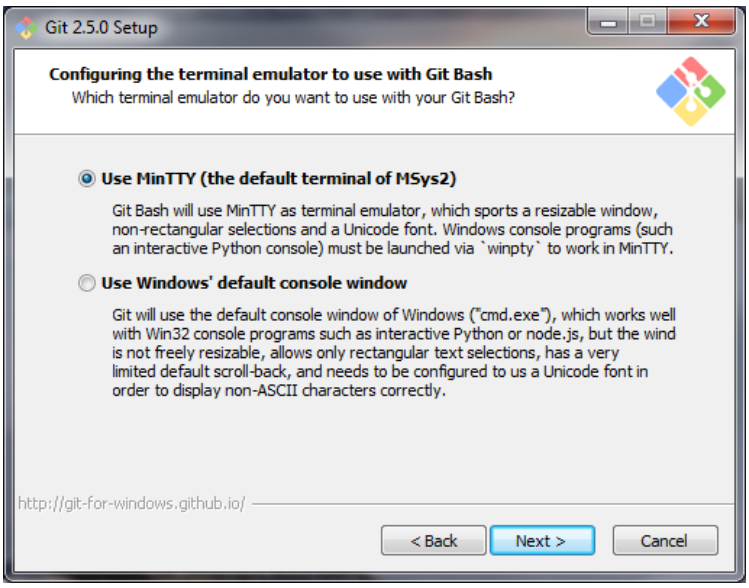

Figure 4. You get a more fully featured shell by selecting MinTTY as the terminal emulator.

The fifth and final dialog enables you to turn on experimental features. In Git 2.5.0 there is only one, the file system caching options. I chose not to turn in on so I can't comment on how it works.

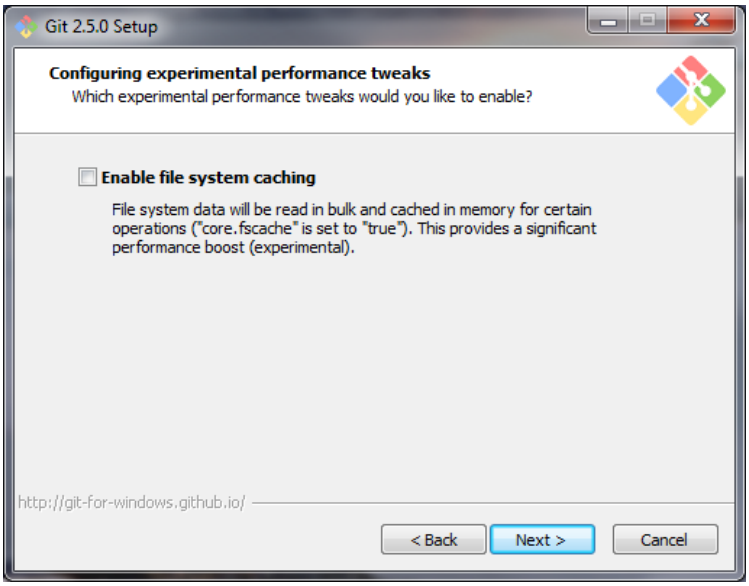

Figure 5. This dialog is for experimental features.

From there on the installation offers no more surprises and proceeds as expected.

You don't need anything other than the command line to use Git, but as a developer accustomed to working with visual development tools like Visual Studio or Visual FoxPro you are most likely going to prefer using a graphical user interface with Git. Git for Windows comes with *Git Gui*, a functional GUI but not necessarily the one you'll like best. There are others you can choose to download and install separately, including:

TortoiseGit – https://tortoisegit.org (replaces the old link on code.google.com)
SourceTree – http://www.sourcetreeapp.com (from Bitbucket)
GitKraken – www.gitkraken.com (beta, from Axosoft)

SourceTree works with both Git and Mercurial, which is a point in its favor if you plan to work with both types of repositories.

Configuring Git

The first thing to do after installing Git on a new machine is to configure it with your name and email address. This is required because Git associates every change made to tracked files with the person who made it. If you set these values in your PC's global (per-user) configuration file, Git uses them for every project you work with on that PC. You can override the global defaults with per-repository settings for specific projects if that becomes necessary.

You can set the global defaults for your name and email address from the command line like this:

```
git config --global user.name "Your Name"
git config --global user.email you@somewhere.com
```

If you're working from a GUI interface, look for the appropriate menu item to edit these and other configuration settings. In Git Gui they're available from the Edit | Options menu.

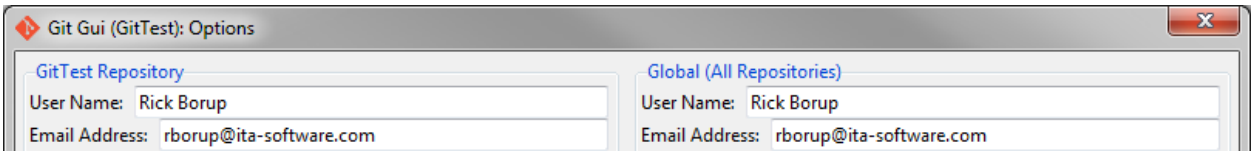

Figure 6. Git Gui provides a form to enter your local and global user name and email address.

There are many other configuration settings you may want or need to change as you use Git, but for starters you only need those two.

Once installed, you can verify that Git is working as expected by opening a Command Prompt and using the `git` command to display Git's version number.

```
C:\>git --version
git version 2.5.0.windows.1
```

If Git responds with the version number, your installation is OK. If Windows responds with the message *'git' is not recognized as an internal or external command, operable program, or batch file*, you probably need to modify your PATH to include `C:\Program Files\Git\cmd`.

GitHub Desktop

An alternative way to install Git on a Windows machine is to download and install *GitHub Desktop* from http://desktop.github.com/. GitHub Desktop (formerly GitHub for Windows) includes a version of Git itself along with a desktop GUI specifically designed for use with GitHub. GitHub Desktop is probably a good choice if you expect to interact extensively with projects on GitHub, but you don't need it if you're not going to use GitHub. See the section on GitHub for more information.

Resources for learning and using Git

There are many resources, both online and in print, to help you get going with Git.

If you want to experiment without actually installing Git on your computer, try the interactive "Try Git" webpage at https://try.github.io/levels/1/challenges/1. This is a scripted series of steps designed to help you learn about and become familiar with Git commands.

After installing Git, a good place to go is the Documentation page at https://git-scm.com/doc, which has links to the online version of the book *Pro Git* [1] as well as to several short training videos. The entire *Pro Git* book is available for free at https://git-scm.com/book/en/v, where you can read it in your browser or download it in PDF and other formats.

The Documentation page also has a link to the official Git reference manual, which is an HTML version of the *man* (manual) pages included with Git itself. The reference manual is the first place to go for details about Git commands.

How to install and get started with Mercurial on Windows

Installing Mercurial on Windows

Like Git, Mercurial is also free. You can download command-line Mercurial by itself from https://mercurial.selenic.com/, or you can download a package that includes both command line Mercurial and the TortoiseHg graphical user interface from http://tortoisehg.bitbucket.org/. Either way, installation is straightforward and offers no surprises.

If you choose the TortoiseHg installer, the TortoiseHg GUI is most likely going to become your go-to interface to Mercurial. If you work with both Git and Mercurial, or if you want an alternative to TortoiseHg, SourceTree is an attractive choice because it works with both Git and Mercurial.

Configuring Mercurial

Like Git, Mercurial requires you to set up your name and email address so it can associate them with the changes you make. If you're working from the command line, open the per-user (global) configuration file in your default text editor with this command:

```
hg config -e
```

In the [ui] section of the configuration file, enter your name and email address on a single line, like this:

```
[ui]
Username = Your Name <you@somewhere.com>
```

If you're working from the TortoiseHg shell you can access this and other settings from the Settings menu. Other GUIs should provide something similar.

As with Git, once Mercurial is installed on your machine you can verify that it is working as expected by opening a Command Prompt and using the **hg** command to display Mercurial's version number.

```
C:\>hg version
Mercurial Distributed SCM (version 3.4.2)
(see http://mercurial.selenic.com for more information)

Copyright (C) 2005-2015 Matt Mackall and others
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

If Mercurial responds with the version number and the rest of the information, your installation is OK. If Windows responds with the message *'hg' is not recognized as an*

internal or external command, operable program, or batch file, you probably need to modify your PATH to include `C:\Program Files\TortoiseHg\` (assuming you installed Mercurial via TortoiseHg).

Resources for learning and using Mercurial

There are also many resources to help you get started with Mercurial. Introductory tutorials can be found at https://mercurial.selenic.com/wiki/BeginnersGuides, including a quick-start guide at https://mercurial.selenic.com/wiki/QuickStart. A more comprehensive guide can be found at https://mercurial.selenic.com/guide. The book *Mercurial: The Definitive Guide* is available online for free at http://hgbook.red-bean.com/.

The official TortoiseHg shell documentation is online at http://tortoisehg.readthedocs.org/en/latest/, including its own quick-start guide at http://tortoisehg.readthedocs.org/en/latest/quick.html.

I've written and published three white papers about DVCS and Mercurial, which are available for download in PDF format from my website. They're geared mostly towards Visual FoxPro developers but should be a useful introduction for anyone.

- *VFP Version Control with Mercurial* – http://bit.ly/IgXzhM
- *Advanced Topics in Mercurial* – http://bit.ly/16j5SIB
- *Multi-track Development Strategies in DVCS* – http://bit.ly/NHgonB

Standard DVCS workflow

Using a DVCS implies learning to incorporate its workflow into your daily routine. The standard workflow is very much the same for both Git and Mercurial.

Initialize a repository

The first step is to create the repository for the project, which is accomplished with the `init` command. This needs to be done only once for each project. You can create a repository for a new project before you start work on it, or you can create a repository for an existing project at any time.

The initialization step creates the set of folders and subfolders that comprise the repository. In Git, these are located in the .git subfolder under the project's root folder. On *nix machines the leading dot indicates a hidden folder. On Windows machines the .git folder is created with the 'hidden' attribute.

In Mercurial, the repository is created as a .hg subfolder under the project's root folder. The Windows 'hidden' attribute is not set on the .hg folder.

Decide which files to ignore

In any project there are usually several types of files you do not want to place under version control. The types of files to ignore typically include executable files, temporary files, certain binary files such as images, and so on.

Rather than having to deal with each of these exceptions one by one, Git and Mercurial enable you to create a file specifying the folders and file types in your project that should be ignored. In Git this file is named .gitignore, while in Mercurial it's called .hgignore. Files and folders whose names match entries in the ignore file are automatically excluded when you run commands that operate on groups of files, such as adding all new files to the repository.

Appendix A is a sample file for Visual FoxPro projects. The syntax is the same in both systems so this file can be used both as .gitignore with Git and as .hgignore with Mercurial. A project's *ignore* file tends to get modified from time to time over the project's lifecycle, so it's a good idea to include it in the repository.

Create and modify source code files

Creating new files and making changes to existing one is of course the nuts and bolts of the development process. As you save your work to the working directory, the DVCS keeps track of which files are new or have been modified.

Add new files to be tracked

Git and Mercurial detect files in the working directory, other than those specified in the *ignore* file, that are not yet tracked. Although the DVCS is aware of these files, they are not automatically added to the repository until you tell it to do so. As you would expect, this is what the `add` command is for.

Add files to the staging area

Git uses the concept of a staging area (sometimes called the index) to hold a reference to files to be included in the next commit. Although it automatically detects files that have been modified since the last commit, Git does not automatically mark them for inclusion in the next commit. Instead, the `add` command must be used before every commit to tell Git which files to commit, even for files that were already added to the repository in earlier commits.

Mercurial does not require files to be staged before committing. Newly created files that are not yet being tracked need to be added, but only once. Thereafter, all tracked files that have been modified are automatically included in the next commit without staging.

Commit changes to the local repository

When changes are committed, the DVCS creates a snapshot of the current state of each new or changed file and stores it in the local repository. Each commit is identified with a unique SHA-1 hash to distinguish it from other commits. When all changes to tracked files have been committed, the working directory is in sync with the local repository and is said to be "clean".

For a solo developer, committing changes to the local repository may be the last step in the workflow. However, there are additional steps if a remote repository is being used.

Push changes to a remote repository

After a developer makes source code modifications and commits them to her local repository, she may want to share them with others developers working on the same project. The `push` command is used to upload new commits from the local repository to a shared remote repository where other developers can access them. A remote repository can also be useful for the solo developer, serving both as an offsite backup and a way to transfer changes between two or more machines.

When collaborating with a team using a shared central repository, it's important to remember that other developers may have pushed commits you don't have yet. If this is true, your push fails because your local repo is not in sync with the shared remote. To avoid this problem, you should always check for newer changes on the remote repo before pushing your own changes. If newer changes are found on the remote, pull them into your local repository, merge them with your changes, commit the merged files to your local repo, and then push your new commits up to the remote repo.

Keep in mind that in Git, the `push` command pushes commits only for the current branch. That means pushing and pulling from the master branch can be done without affecting anything you might be doing in a development branch.

Pushing files to a remote repository in Mercurial is equivalent to doing the same in Git, except that Mercurial pushes changes from all branches by default. Like Git, Mercurial rejects a push if there are newer changes in the remote repository.

Pull changes from a remote repository

The `pull` command checks for newer commits in a remote repository and downloads them to the local repository. From there, those changes must be merged into the working copy. In Git, the `pull` command automatically launches a merge while in Mercurial it does not. Both systems have way to override their default behavior.

Merge others' changes with yours

Merging is the process of integrating changes to a file with a different version of the same file. When there is no doubt what the merged code should be—for example, if a new block of code was added in a place where there was no code before—then merging is a seamless process. However, if there have been two different sets of changes to the same area of code then a merge conflict can result. Merge conflicts are detected by the DVCS but must be resolved by the developer before the merge can be completed. Merge conflicts are resolved either by accepting the changes from developer A, accepting the changes from developer B, or modifying the code to produce a combination of the two.

Git vs Mercurial similarities and differences

Both Git and Mercurial are command-line tools. Their full functionality can be driven from the command line with no need for a GUI, and many developers work with them this way. On the other hand, developers who are accustomed to working in a visual IDE like Visual Studio.NET and Visual FoxPro will probably prefer a graphical user interface over the command line.

Several different GUIs are available for Git and Mercurial. All of them drive the underlying tool with the same commands available from the command line, so in that sense a GUI can't do anything the tool itself can't do.

Staging

Both Git and Mercurial use the *add* command to start tracking changes to a file. In Mercurial you only need to do this once. From then on, whenever you perform a commit, Mercurial's default behavior is to include all the tracked files that have been modified since the previous commit.

Git is different. When you modify a tracked file, Git knows that the file has been changed but does not automatically include it in the next commit. To include a modified file in the next commit, you first have to stage it. This is done using the same `add` command that was initially used to begin tracking the file. If you don't stage a modified file by adding it again, that file is not included in the next commit. For convenience, you can use the "–A" option to stage all new and modified files in a single `add` command rather than having to add each one individually.

Branching

Branching is the process of creating a divergent line of development. Branching enables the developer to experiment or make changes for a new line of development without affecting the state of the files in the release branch or other lines of development.

A common example is when development begins on a new feature. The main branch, which typically represents the current release version of the code, needs to remain intact in case it's needed again before work on the new feature is completed – for example, if it becomes necessary to create a hotfix for the current release. If work on the new feature is occurring on a development branch, you can switch back to the main branch, create the hotfix, build and release from that, then switch back to the development branch and continue work on the new feature. When the new feature is complete, the development branch can be merged back into the main branch and the new release built from there.

Although conceptually similar, branching in Git is different than branching in Mercurial.

Branching in Git

In Git, branches are lightweight pointers to the HEAD (i.e., the most recent) commit in the specified branch. When you create a branch, Git creates a new pointer to the same commit as you're currently working on. To work in the new branch (or in an existing branch) you

first *checkout* the desired branch name. Git moves the HEAD pointer to the branch, and all subsequent commits are then recorded on that branch until you checkout another branch.

When you're done with whatever changes are made in the branch, you can decide whether to keep them (if you want them to become part of the release version) or discard them (if you were simply experimenting with something). If you decide to keep them, you commit them to the repository, checkout the main branch, merge the changes from the development branch back into the files on the main branch, and then commit the merged files.

After you're done with a branch in Git, you have the option to delete it. Deleting a branch removes its pointer and the branch effectively disappears from the commit history.

See https://git-scm.com/book/en/v2/Git-Branching-Branches-in-a-Nutshell for more information on branching in Git.

Branching in Mercurial

When you create a new branch in Mercurial it's not actually created until you do the first commit on it. Once you've created or switched to a branch, all subsequent commits occur on that branch until you switch to a different branch.

The next step is the same as Git. When you're done with whatever changes are made in the branch, you can decide whether to keep them or discard them. If you decide to keep them, you commit them to the repository, update to the main branch, merge the changes from the development branch back into the files on the main branch, and then commit the merged files.

Mercurial keeps an immutable history of all changes made to tracked files. Once a branch is created it becomes a permanent part of the repository, so there is no concept of deleting a branch in Mercurial. If a branch is not going to be used again you can mark it as closed. Closing a branch lets others know the intent is not to use that branch anymore, but doesn't actually prevent it from being used again.

Pushing branches to a remote repository

If you're working with remote repositories and there are multiple branches in your repository, you need to be aware of one major difference between Mercurial and Git. When pushing revisions to a remote repository, Git pushes only the *current* branch, while Mercurial's default behavior is to push *all* branches. Mercurial does provide a way to push only a specified branch, but that's not how it works by default.

Command line examples

The best way to first learn Git or Mercurial is to use the command line. Using the command line requires you to write each command by name, which not only helps you learn the names of the commands and what they do but also teaches you the sequence of steps in the workflow. Although you will almost certainly want to use a graphical user interface in your

daily work, you'll have a much better understanding of what the GUI is doing behind the scenes if you first become familiar with the command line.

The basic workflow

This section uses a simple example to illustrate the basic workflow in Git and Mercurial using the command line. A file named foo.txt is created in Notepad. A repository is initialized and foo.txt is added and committed. A couple of changes are then made to foo.txt, including the creation of a new branch. The new branch is then merged back into the main branch and the workflow is complete.

The example is broken up into small sections for easy comparison, showing first how to do it in Git and then how to do the same thing in Mercurial. The words in square brackets are my comments – they are not part of the actual commands you type in or the responses you receive from Git and Mercurial.

Git commands are invoked with the word 'git' while Mercurial commands are invoked with 'hg' (Hg is the chemical symbol for Mercury). If you want to run these examples on your own machine, create two temporary folders, one for Git and the other for Mercurial, and then reproduce the steps below.

Create foo.txt, initialize a repo, add foo.txt, and commit it

The commands for the first step are identical in both Git and Mercurial.

```
[Git]
notepad foo.txt
  Fox rocks! [add this line and save the file]
git init
git add foo.txt
git status
  new file: foo.txt [this text is rendered in green if your shell supports colors]
git commit -m "initial commit"

[Mercurial]
notepad foo.txt
  Fox rocks! [add this line and save the file]
hg init
hg add foo.txt
hg status
  A foo.txt [status 'A' means the file was newly added]
hg commit -m "initial commit"
```

Modify foo.txt and commit the changes

The steps are also very similar in this step, but note that in Git the modified file needs to be staged before committing. Mercurial does not use staging.

```
[Git]
notepad foo.txt
  Fox rocks!
  Git rocks, too! [add this line and save the file]
```

```
git status
  modified: foo.txt [red indicates the file has been modified but not yet staged]
git add foo.txt [OR git add -A]
git status
  modified: foo.txt [green indiates the file is staged for the next commit]
git commit -m "modified foo.txt"

[Mercurial]
notepad foo.txt
  Fox rocks!
  Hg rocks, too! [add this line and save the file]
hg status
  M foo.txt [status 'M' means the file has been modified]
hg commit -m "modified foo.txt"
```

Create a 'dev' branch and switch to it

```
[Git]
git branch --list [list all branches]
  * master
git branch dev  [creates a new pointer to the same commit you're currently on (HEAD)]
git branch --list
    dev
  * master [the current branch is 'master']
git checkout dev [moves the HEAD pointer to the 'dev' branch]
  switched to branch 'dev'
git branch -- list
  * dev [the current branch is now 'dev']
    master

[Mercurial]
hg branches [list all branches]
  default [the current branch is 'default']
hg branch dev [tell Mercurial to create or switch to branch 'dev' on the next commit]
hg branches
    default ['default' is still the only branch until the next commit]
```

Modify foo.txt and commit the changes to the 'dev' branch

```
[Git]
notepad foo.txt
  Fox rocks!
  Git rocks, too!
  This line was added in branch 'dev'. [add this line and save the file]
git status
  on branch dev
  changes not staged for commit
  modified: foo.txt [modifed but not yet staged]
git add foo.txt
git status
  on branch dev
  changes to be committed:
  modified: foo.txt [modified and staged for the next commit]
git commit -m "change in dev branch"
git branch -- list
```

```
  * dev [the current branch is still ‘dev’]
    master

[Mercurial]
notepad foo.txt
  Fox rocks!
  Hg rocks, too!
  This line was added in branch 'dev'. [add this line and save the file]
hg status
  M foo.txt
hg commit -m "change in dev branch"
hg branches
  dev [the ‘dev’ branch now exists and is the current branch]
  default (inactive)
```

Merge the ‘dev’ branch back into the main branch

```
[Git]
git checkout master [moves the HEAD pointer back to the ‘master’ branch]
  switched to branch 'master' [NOTE – This changes foo.txt in the working directory!]
type foo.txt
  Fox rocks!
  Git rocks, too! [the third line is not there in the ‘master’ branch]
git merge dev [merge the dev branch into the master branch]
type foo.txt
  Fox rocks!
  Git rocks, too!
  This line was added in branch 'dev'. [now the third line is there in ‘master’]
git branch --list
    dev
  * master [the current branch is ‘master’]
git status
  on branch master
  nothing to commit, working directory clean

[Mercurial]
hg update default
  1 files updated, ... [NOTE – This changes foo.txt in the working directory!]
type foo.txt
  Fox rocks!
  Hg rocks, too! [the third line is not there in the ‘default’ branch]
hg merge dev
  1 files updated, ...
  (branch merge, don't forget to commit)
type foo.txt
  Fox rocks!
  hg rocks, too!
  This line was added in branch 'dev'. [now the third line is there in ‘default’]
hg branches
  default
  dev (inactive)
hg status
  M foo.txt [the merged file was not automatically committed to the repo]
hg commit -m "merged branch dev" [commit takes place on the default branch]
hg status
```

```
[nothing]
```

If you're using a GUI you get a graphical representation of the revision history. **Figure 7** is the revision history from the Git example as displayed by Git Gui. **Figure 8** is the Mercurial example as displayed by TortoiseHg.

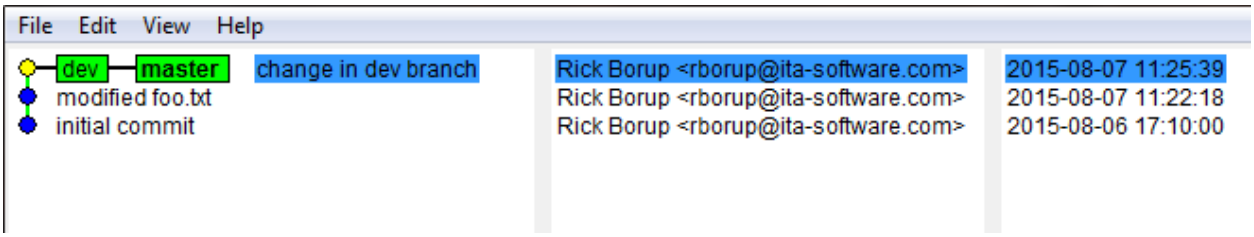

Figure 7. The revision history for the above example in Git looks like this in Git Gui.

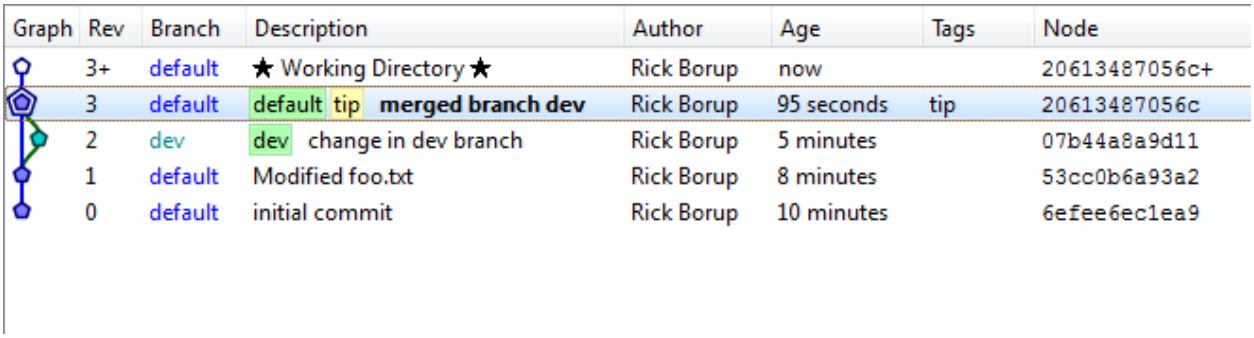

Figure 8. The revision history for the above example in Mercurial looks like this in TortoiseHg.

Extending the workflow to a remote repository

Most development projects under distributed version control involve the use of a remote repository. From the perspective of any individual developer, the remote repository is any repository for the same project other than her local one. While developer A's local repository could serve as a remote repo for developer B, and vice versa, it's common practice to establish a central repository to be shared by all developers on the team. This approach also works for solo developers, where the remote repository can act as a conduit between development machines and also as an off-site backup, depending on where it's physically located.

The two key aspects of a shared remote repository are (a) all members of the team are authorized to pull from and push to it, and (b) it's a bare repository, meaning it does not have a working directory. If it's cloud-based, the central remote repository is probably accessible only via HTTPS or SSH. For smaller teams within the same office, the shared repository might be a folder on a mapped drive and therefore accessible via the file system. Either way, the centralized remote repository needs to be a bare repository because pushing files to a non-bare repository could result in inconsistencies between its working copy and what was pushed.

The following examples show how to create a remote repository and push changes to it in both Git and Mercurial. The scenario is that the developer is working on a project on her local hard drive located in folder C:\Temp\GitTest. In order to share her code with other members of the development team, she creates a remote repository on mapped drive F:, which is also accessible to the other members of the team. The remote repository is created as a subfolder under F:\GitCentral, where *GitCentral* is simply an arbitrary name I like to use for a root-level folder containing the remote repositories for several projects. The name of the remote repository folder for the GitTest project does not have to have the same as the name of the working copy folder on the developer's machine, but using the same name for both makes it easier to keep things straight.

Working with a remote repository in Git

Alice, the lead developer on this project, reaches a point in her work where she's ready to share her code with other members of the team. Working from the command line, she begins by changing to the shared F: drive where she creates the F:\GitCentral\GitTest folder and initializes a bare repository in it. Note the `--bare` option on the `init` command.

```
C:\Temp\GitTest>cd F:
F:\>md GitCentral [GitCentral is an arbitrary name for this folder]
F:\>cd GitCentral
F:\GitCentral>md GitTest
F:\GitCentral>cd GitTest
F:\GitCentral\GitTest>git init --bare [create a bare repository, no working copy]
```

Switching back to the working copy on her C: drive, she tells Git to add a reference to the new remote repository on the F: drive. This creates the link Git uses to push files to the remote repository, and later to fetch files from it. At this point all work is being done on the master branch, which is referenced in the command.

```
F:\GitCentral\>C:
C:\Temp\GitTest>git remote add master F:\GitCentral\gittest
```

Alice wants to preview what will happen when she tells Git to push her work to the new remote repository, so she runs the push command with the –n option. Git responds that a new branch called master will be created on F:\GitCentral\GitTest and shows that her master branch will be pushed to it.

```
C:\Temp\GitTest>git push –n master master [preview what will happen when we push]
   To F:\GitCentral\GitTest
    * [new branch]      master -> master
```

Satisfied that everything is in order, Alice goes ahead and pushes her changes to the remote repository. Git displays its progress and confirms that the push was successful.

```
C:\Temp\GitTest>git push master master
   Counting objects: 12, done.
   Delta compression using up to 4 threads.
   Compressing objects: 100% (6/6), done.
   Writing objects: 100% (12/12), 999 bytes | 0 bytes/s, done.
```

```
   Total 12 (delta 1), reused 0 (delta 0)
   To F:\GitCentral\GitTest
    * [new branch]      master -> master
```

At this point, running the *git log* command from the F:\GitCentral\GitTest folder shows the same revision history as on Alice's local repository. The difference is that none of the actual source code files exist in F:\GitCentral\GitTest – it contains only the repository.

Bob, the junior member of the team, is assigned to work on Alice's project. He starts by making a clone of the project source code from the same remote repository Alice pushed to. He decides to work in his C:\Temp folder so he switches to that folder and runs the Git clone command. In the following code, C: refers to Bob's local hard drive, not Alice's.

```
C:\>cd Temp
C:\Temp>git clone F:\GitCentral\GitTest
   Cloning into 'GitTest'...
   done
C:\Temp>cd GitTest
```

Bob wants to check and see what he got, so he does a `dir` on his GitTest project folder. He sees that he got the source code file foo.txt.

```
C:\Temp\GitTest>dir
   08/21/2015  04:09 PM     <DIR>        .
   08/21/2015  04:09 PM     <DIR>        ..
   08/21/2015  04:09 PM              100 foo.txt
                  1 File(s)           100 bytes
                  2 Dir(s)  8,504,459,264 bytes free
```

But wait! Bob knows that the remote repository is a bare repository and therefore does not have a working copy, so the file foo.txt does not exist on the remote. How, then, did it end up in the working directory on his local hard drive?

The answer is that, as part of the clone command, Git performed an automatic checkout to the head of the master branch. This created foo.txt in Bob's working directory, ready for him to begin adding his own changes. At this point, Bob and Alice both have the same version of foo.txt.

The clone operation also created a reference in Bob's local repository linking to the origin of the repository he just cloned. Bob checks this by running the Git remote command with the –v (or --verbose) option.

```
C:\Temp\GitTest>remote –v
   origin F:\GitCentral\GitTest (fetch)
   origin F:\GitCentral\GitTest (push)
```

When Bob is ready to push changes back to the remote repository to share them with Alice, Git uses this information to know where to push them. Two-way collaboration is now possible: Alice can pull Bob's changes from the remote into her local repo and merge them

with her own work, and Bob can pull Alice's changes from the remote into his local repo and merge them with his work.

Working with a remote repository in Mercurial

The process for creating a remote repository and pushing changes to it is very similar in Mercurial. Alice first initializes a new Mercurial (Hg) repository in F:\HgCentral\HgTest.

```
C:\Temp\hgtest>cd F:
F:\>md HgCentral
F:\>cd HgCentral
F:\HgCentral>md HgTest
F:\HgCentral>cd HgTest
F:\HgCentral\HgTest>hg init [Mercurial does not require a "bare" option]
```

She then pushes the source code from her local repository to the newly created remote repository on the F: drive.

```
F:\HgCentral\HgTest>C:
C:\Temp\HgTest>hg push F:\HgCentral\HgTest
   pushing to F:\HgCentral\HgTest
   searching for changes
   adding changesets
   adding manifests
   adding file changes
   adding 3 changesets with 3 changes to 1 files
```

The repository on F: is now up to date with Alice's latest changes, and Bob can clone it the same way he did in Git. Working on his own local machine, Bob does the following:

```
C:\>cd Temp
C:\Temp>hg clone F:\GitCentral\GitTest
   destination directory: HgTest
   updating to branch default
   1 files updated, 0 files merged, 0 files removed, 0 files unresolved
C:\Temp>cd HgTest
C:\Temp\HgTest>dir
   08/22/2015  04:40 PM     <DIR>      .
   08/22/2015  04:40 PM     <DIR>      ..
   08/22/2015  04:40 PM     <DIR>      .hg
   08/22/2015  04:40 PM            100 foo.txt
                  1 File(s)          100 bytes
                  3 Dir(s)  8,503,918,592 bytes free
```

You may have noticed that the .hg folder shows up when you run a dir command on a folder containing a Mercurial repository, but the .git folder does not show up when you run a dir command on a folder containing a Git repository. The difference is that the .git folder gets created with the 'hidden' attribute. It's there, but you can only see it with `dir /ah`.

How to set up and use repositories on GitHub and Bitbucket

GitHub and Bitbucket are both cloud-based commercial services for hosting source code repositories. There are two main reasons you might want to use them:

1. you want to share code publicly with the world, or
2. you want to share code privately with a team but don't want to run your own server.

In addition to their ability to serve as remote repositories to support local development, GitHub and Bitbucket provide many features designed to make it easier for several developers to collaborate on the same project. These features are based on the workflow concepts of *cloning*, *forking*, and creating *pull requests*.

Cloning

Cloning is the process of duplicating a repository into a new directory. If you find an interesting project on GitHub or Bitbucket that you want to contribute to, or if you are assigned to work on a project hosted on one of those services, you can clone it into a folder on your own machine where you can begin to explore and make changes.

Cloning is part of core Git and Mercurial, not something unique to GitHub or Bitbucket. You can use the `clone` command to create a duplicate of any repository to which you have access, including your own. To clone repositories to which you have file system access, for example on a local hard drive or file server, you simply pass the path to the source and destination. For example, to clone the repository in C:\temp\gittest into a new folder on your desktop you would use the comand

```
C:\>git clone c:\temp\gittest c:\users\<username>\desktop\gittest
```

Developers do not have file system access to repositories on GitHub and Bitbucket, so these services provide a URL that can be used to clone a hosted repository over secure HTTP. For example, the URL to clone Rick Strahl's wwDotnetBridge project from GitHub is

https://github.com/RickStrahl/wwDotnetBridge

and the URL for Matt Slay's GoFish4 project from its on Bitbucket is

https://bitbucket.org/mattslay/gofish4.

The browser interface to GitHub and Bitbucket simplifies things by providing a clickable link or button on each project's home page that you can use to clone the project without using the command line.

Forking

Forking is the process of duplicating a GitHub or Bitbucket repository into your own account on the same service. Forked projects show up in your account with the original project name preceded your account name.

You fork a project when you want to use someone else's work as the starting point for your own. Forks are visible to the project owner (and to everyone else, if it's a public project). By creating a fork, you are signaling your intention to diverge from the main line of development, typically to add a new feature and/or to make changes to the existing code.

Thereafter, the fork may continue to stand on its own as an alternative to the original, or the developer might request the owners of the project to pull her changes into the original repository. The latter is done by creating a pull request.

Pull requests

A pull request is a notice to the owners of a hosted project asking them to consider incorporating someone else's changes. Without a pull request, the developer who made the changes would have to find some other way to contact the owners of the project, such as by email, which could be difficult or even impossible. GitHub and Bitbucket facilitate this communication by automatically notifying the owners when a pull request is created on one of their projects.

Projects to which anyone can contribute operate under a *fork and pull* model. Anyone can fork the project, make changes or enhancements, and then create a pull request. In this model, pull requests aren't merely a private communication between the developer and the project's owners. Because everyone can see them, the pull requests can also serve as the trigger for a back-and-forth conversation about the merits of the proposed changes, potentially involving other developers as well. Under the fork and pull model, the final decision about whether to accept a pull request rests with the project's owners.

Projects to which only members of a development team can contribute tend to operate under a *share repository* model. In this model, each developer may have authorization to push changes to the shared repository. Pull requests are not required, but can still be useful as a way to notify other team members and generate discussions about proposed changes.

Note the pull requests are a feature of services like GitHub and Bitbucket. They are not part of core Git or Mercurial.

Working with GitHub

You do not need to download anything special to use GitHub – all you need is a Web browser, a GitHub account, and Git itself (command line Git). If you are using a GUI interface like TortoiseGit or SourceTree you can use it with GitHub the same way you would work with any other remote repository, once your credentials to access the GitHub project have been configured. There is a growing number of extensions and special purpose interfaces specifically designed for GitHub. One of these is GitHub Desktop – more on that later.

Setting up a GitHub account

To set up an account on GitHub go to https://github.com, pick a username and password, supply an email address, and you're ready to go.

GitHub offers both free and paid accounts. A free account entitles you to create public repositories that can be accessed by anyone, so they're suitable for code you want to share with the world. You'll want a private repository for proprietary code, and for that you'll need a paid account. GitHub paid plans start at $7 per month.

Using GitHub from the command line

Every project on GitHub has a unique URL based on the GitHub account name and the name of the project. In this example, work begins by cloning an existing project from GitHub onto the developer's local machine. To clone a project from GitHub onto your local machine, all you need is the URL and permission to access the project. If the project is public, anybody who knows or can discover the URL can clone it. GitHub has a search feature to help you find interesting public projects.

This example starts with a project named GitTest hosted on the SWFoxDemo account on GitHub. The URL for that project is https://github.com/SWFoxDemo/GitTest.git. Use the following two commands to create a clone of this project in your desktop folder.

```
C:\>cd \users\<username>\desktop
C:\users\<usernname>\desktop>git clone https:\\github.com\SWFoxDemo\GitTest
```

This creates a new folder named GitTest on your desktop. To verify this, change to that folder and check its contents.

```
C:\users\username\desktop>cd gittest
C:\users\username\desktop\gittest>dir
C:\users\username\desktop\gittest>
   08/25/2015  11:06 AM    <DIR>          .
   08/25/2015  11:06 AM    <DIR>          ..
   08/25/2015  11:06 AM               281 foo.txt
                  1 File(s)            281 bytes
                  2 Dir(s)  828,372,832,256 bytes free
```

If you are the GitHub account owner or an authorized user, you can push your changes back up to the GitHub repository. The local reference to the origin of the clone was automatically created by Git when you cloned the project. You can use Git's `remote` command to verify the destination before pushing.

```
C:\users\username\desktop\gittest>git remote –v
   Origin  https://github.com/SWFoxDemo/GitTest.git (fetch)
   Origin  https://github.com/SWFoxDemo/GitTest.git (push)
```

Use the following command to push your changes back to the GitHub repo. "Origin" is the default nickname for the location on GitHub from which the project was cloned, so this command tells Git to push the changes from your master branch back to their origin. If not already signed in, Git prompts for the GitHub account username and password.

```
C:\users\username\desktop\gittest>git push origin master
   Username for 'https://github.com': SWFoxDemo
   Password for 'https://github.com': xxxx
```

```
Counting objects: 3, done.
Delta compression using up to 4 threads.
Compressing ojbects: 100% (2/2), done.
Writing objects: 100% (3/3), 339 bytes | 0 bytes/s, done.
Total 3 (delta 1), reused 0 (delta 0)
To https://github.com/SWFoxDemo/GitTest.git
   4793f9c..4a7055d  master -> master
```

If you are not the GitHub account owner or other authorized user, you cannot push changes back to the GitHub repository. Instead, you need to notify the account owners that you have some changes you'd like them to consider by creating a pull request.

Using GitHub from a browser

While you can use GitHub from the command line, you'll get a much more fully featured experience when using it from a browser. Some features, such as pull requests, are only available when you're logged in to your account using a browser.

Let's say I want to fork the GitTest repository from the SWFoxDemo account into my own personal GitHub account, make a change, and create a pull request for the owners of SWFoxDemo to consider.

First, I log in to GitHub using my own account. Once logged in, I perform a search for the desired project. As you might expect, there are many projects named "GitTest" on GitHub, so I use the advanced search feature to find the one owned by SWFoxDemo account.

Figure 9. GitHub's advance search enables you to find a project by its name and owner.

The search yields only one result. Clicking on its link brings up the GitHub homepage for that project.

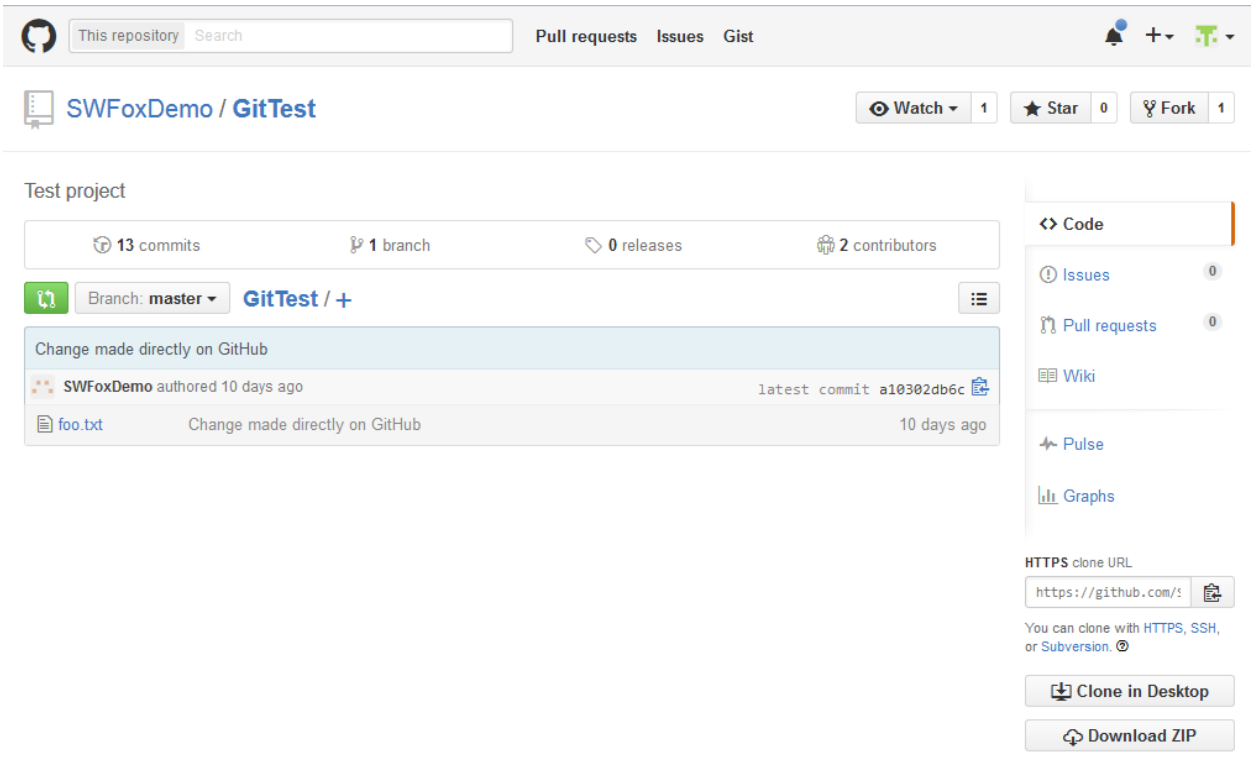

Figure 10. The homepage for the GitTest project published by the SWFoxDemo user on GitHub.

I want to fork this project into my own account, so I click on the Fork button. This creates a fork of the GitTest project under my own GitHub account. I make a change by adding a wild idea to foo.txt.

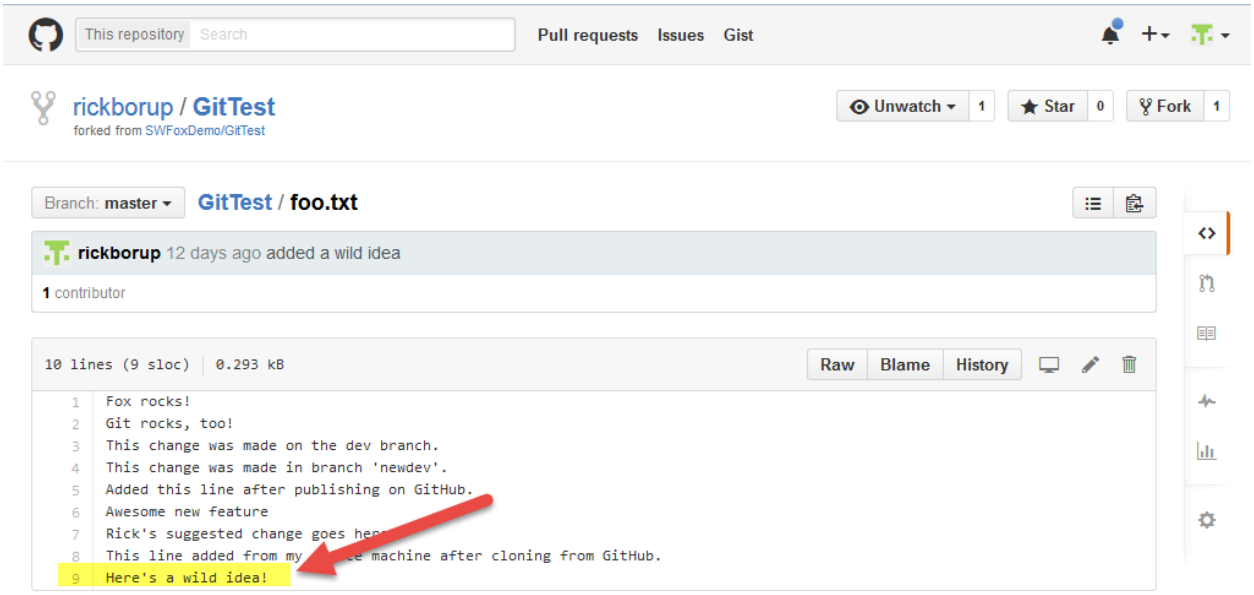

Figure 11. I added a wild idea to the project in the fork under my own GitHub account.

Clicking the Pull Request link on my fork's GitHub homepage takes me to the Pull Request page. From there I can create a new pull request to notify the owners of this GitTest project that I want them to consider pulling my change.

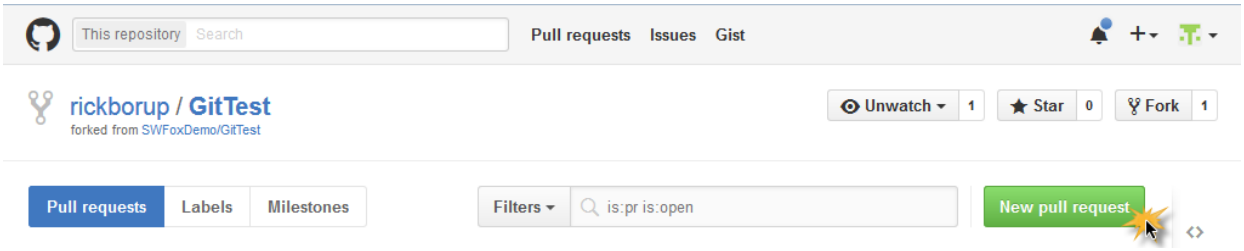

Figure 12. GitHub provides a button to create a new pull request and notify the owners that someone wants to contribute to their project.

The owners of the project receive an email notifying them that a pull request has been created. **Figure 13** is the email generated by GitHub for the pull request in this example.

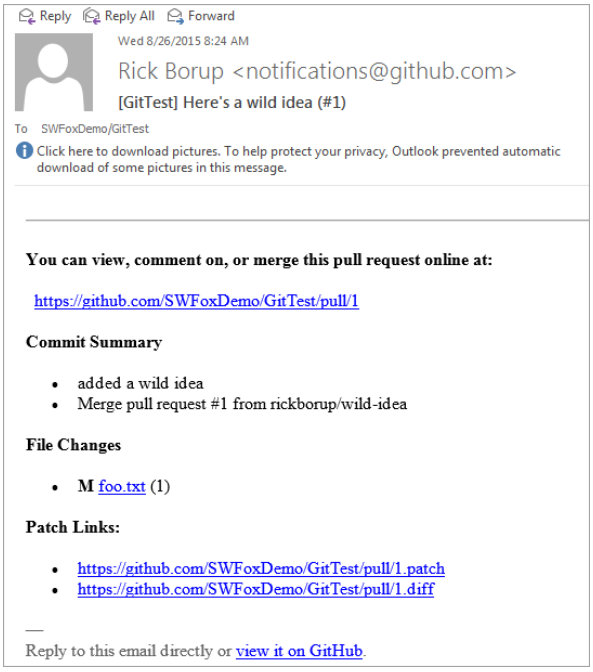

Figure 13. The owners of the project receive an email notifying them that a pull request has been created.

The owners of the SWFoxDemo account review the pull request on their GitHub account. They decide the suggested change is an awesome idea and accept it. The change now becomes part of the original GitTest repository and can be seen in its commit history.

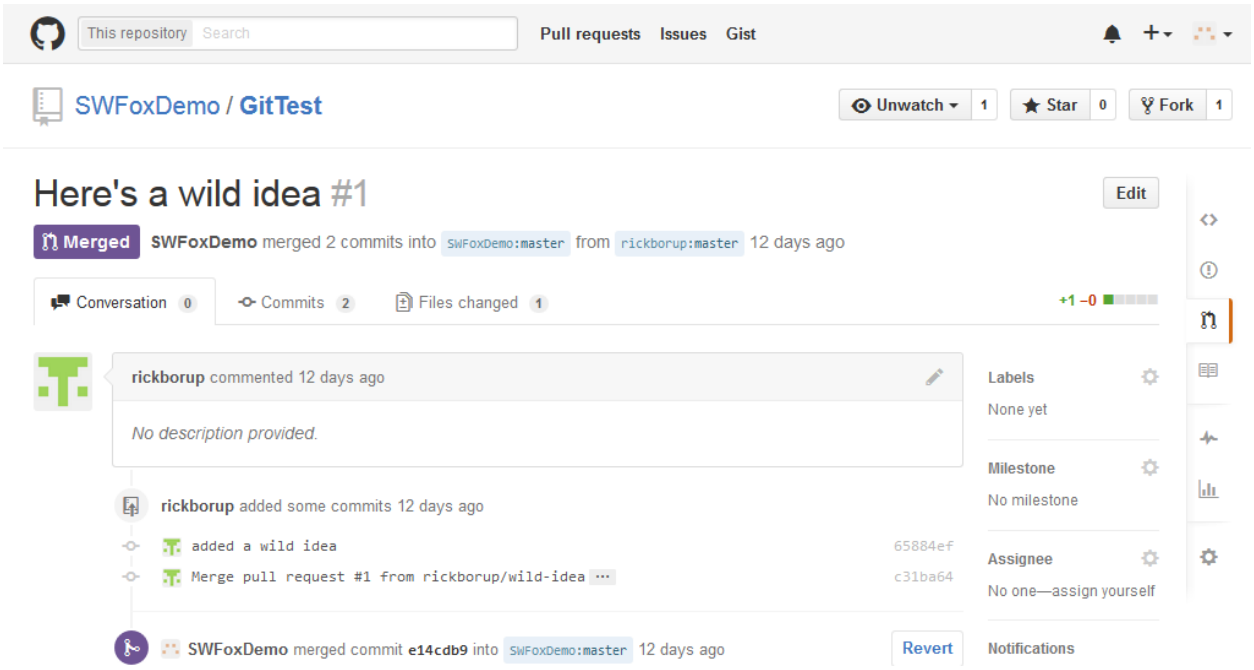

Figure 14. After accepting the pull request, the new feature is incorporated into the revision history of the original project.

GitHub Desktop

GitHub released *GitHub Desktop for Mac and Windows* (or *GitHub Desktop* for short) in August, 2015. This app "is designed to simplify essential steps in your GitHub workflow and replace GitHub for Mac and Windows with a unified experience across both platforms". [1] Like its predecessors, GitHub Desktop provides seamless interaction with GitHub from your desktop without having to touch the command line.

You do not need GitHub Desktop to use GitHub from either a Mac or a Windows machine—all you need is Git, a GitHub account, and a browser—but the app is intended to make it easier. GitHub Desktop is available as a free download from https://desktop.github.com. The Getting Started with GitHub Desktop tutorial at https://help.github.com/desktop/guides/getting-started/ is a good place to start learning how to use it.

Here are a couple of screenshots to give you an idea what the GitHub Desktop user interface looks like. **Figure 15** shows that foo.txt has been modified but not yet committed, as reflected in the status indicator in the upper right. ① Clicking on the "Commit to master" link ② commits the changes to the master branch in the local repository.

[1] https://github.com/blog/2046-github-desktop-is-now-available

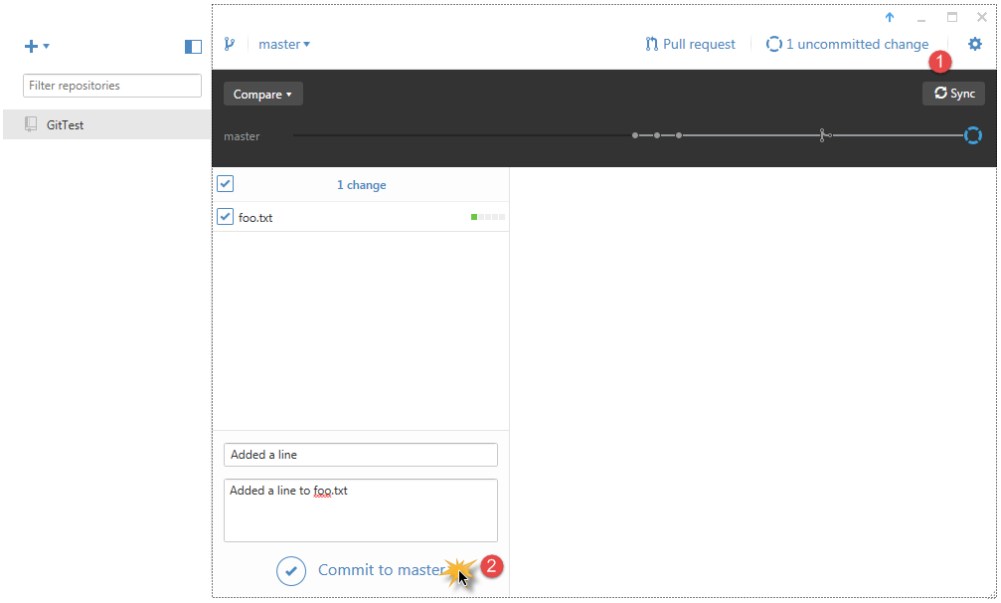

Figure 15. The GitHub Desktop interface shows 1 uncommitted change and provides a clickable button to commit it.

After committing the changes to the local repository, GitHub Desktop shows that there are no local changes, meaning the local repo is clean.

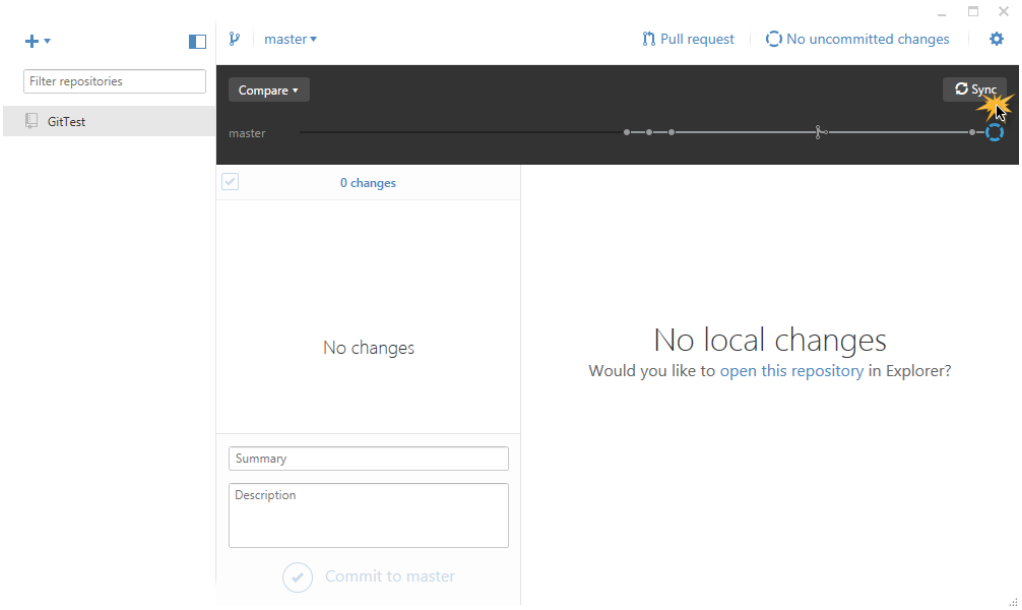

Figure 16. After committing, the local repository is clean and can by synced with GitHub by clicking the Sync button.

Because this particular local GitTest repository was originally cloned from the SWFoxDemo project on GitHub, it's already configured to push to that remote. Syncing the local repo with GitHub is as simple as clicking the Sync button (upper right in **Figure 16**).

Working with Bitbucket

Bitbucket is a cloud-based commercial service for hosting both Git and Mercurial repositories. You do not need to download anything special to use Bitbucket – all you need is a Bitbucket account, a Web browser, and Mercurial and/or Git installed on your machine.

Setting up a Bitbucket account

To set up an account, go to https://bitbucket.org and sign up. Bitbucket is free for small teams (up to five users) and offers plans starting at $10 per month for larger teams. Unlike Git, Bitbucket offers private repositories even with its free account, making it an ideal solution for individual developers and small teams who work with proprietary code.

Using Bitbucket from the command line

This example starts with a project name HgTest that already exists on the developer's local machine. The first step is to create a new repository on Bitbucket if one does not already exist.

Figure 17 illustrates how to create a repository on Bitbucket. This one is named HgTest, which is the same name as the existing local project. It's not necessary to use the same name, but it helps keep things straight.

The local HgTest project is already under Mercurial version control on the local machine, so Mercurial is selected for the type of repository on Bitbucket. If the "private repository" check box is left blank, the repository is public and anyone can find it. Note that Bitbucket includes FoxPro as one of the choices for the source code language. Marking a VFP project as FoxPro helps others who may be searching for FoxPro projects on Bitbucket.

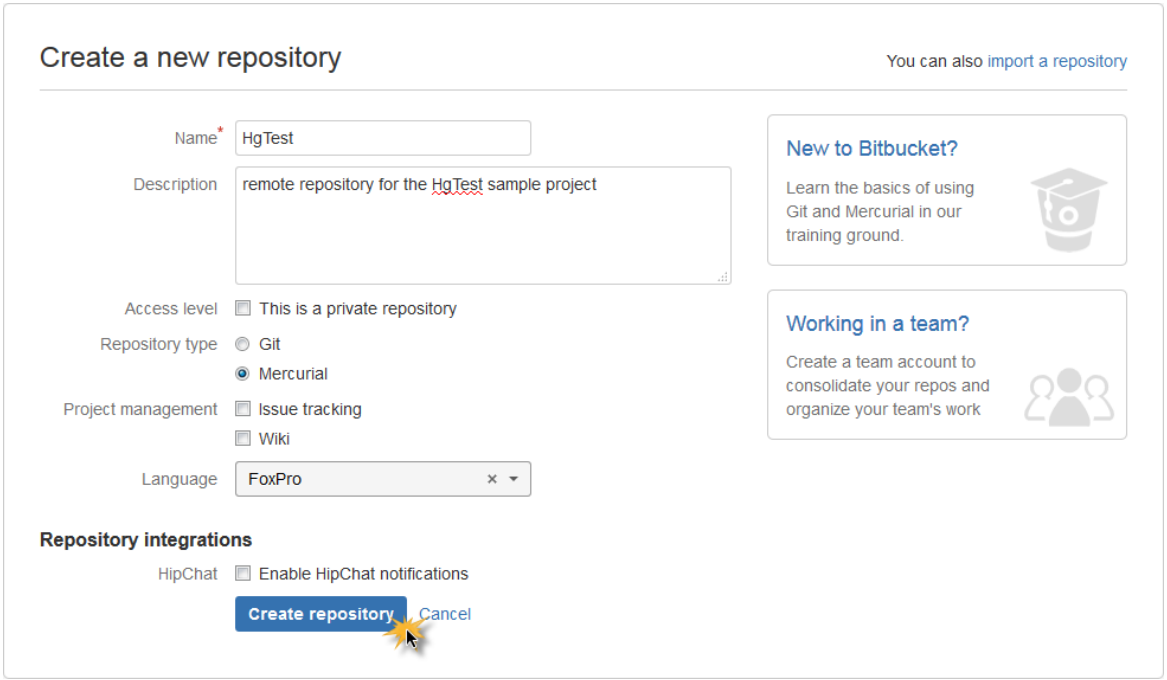

Figure 17. You can select either Git or Mercurial when creating a new repository on Bitbucket.

Once the remote repository has been created, you can use Mercurial's `outgoing` command to check if there are any commits in the local repo that are not yet on Bitbucket.

```
C:\Temp\hgtest>hg outgoing https://bitbucket.org/SWFoxDemo/hgtest
comparing with https://bitbucket.org/SWFoxDemo/hgtest
http authorization required for https://bitbucket.org/SWFoxDemo/hgtest
realm: Bitbucket.org HTTP
user: SWFoxDemo
password: xxxx
searching for changes
...
changeset:   3:42caf8f4b5a9
tag:         tip
parent:      1:c4e615247419
parent:      2:c1b10dffee97
user:        Rick Borup <rborup@ita-software.com>
date:        Sat Aug 01 23:10:20 2015 -0500
summary:     Merge with dev
```

Mercurial detects changes that are not yet on Bitbucket. Use the `push` command to push the changes to Bitbucket.

```
C:\Temp\hgtest>hg push https://bitbucket.org/SWFoxDemo/hgtest
pushing to https://bitbucket.org/SWFoxDemo/hgtest
http authorization required for https://bitbucket.org/SWFoxDemo/hgtest
realm: Bitbucket.org HTTP
user: SWFoxDemo
password: xxxx
searching for changes
remote: adding changesets
remote: adding manifests
remote: adding file changes
remote: added 4 changesets with 3 changes to 1 files
```

The repository on Bitbucket is now in sync with the local repository for this project.

Using Bitbucket from a browser

To give you a feel for it, **Figure 18** shows the Bitbucket interface when viewing the homepage of a project named HgTest while logged in as user SWFoxDemo, the owner of the project.

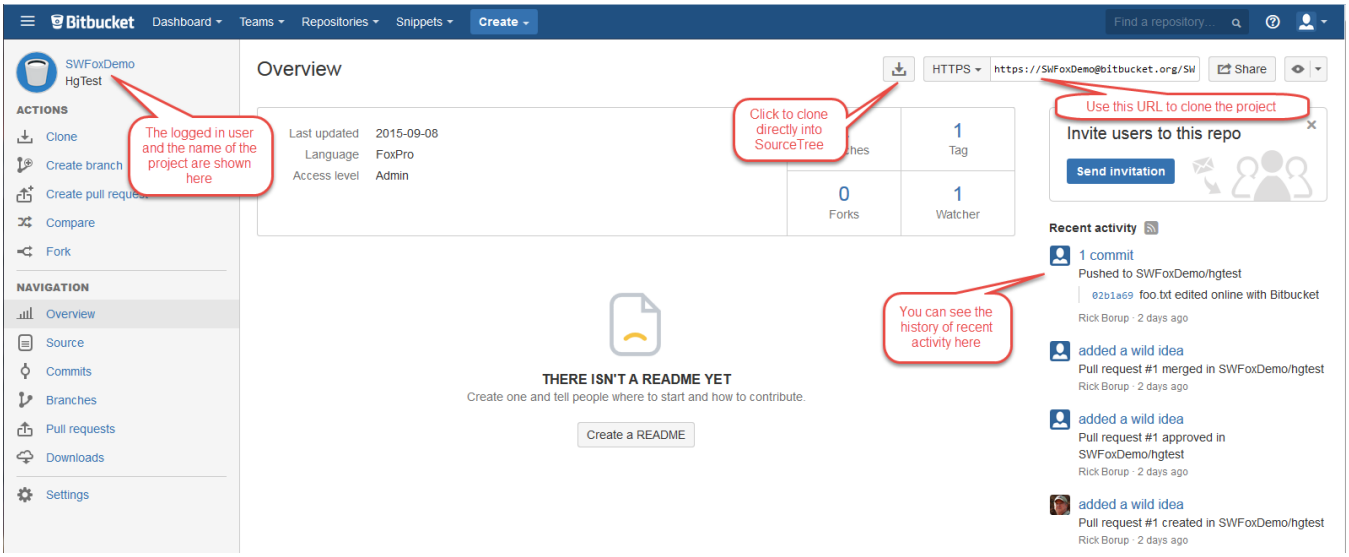

Figure 18. The homepage of the HgTest project on Bitbucket when logged in as its owner, SWFoxDemo.

The upper left corner displays the logged in username and the name of the project. At the upper right is the URL to clone this project. To its left is a button to clone the project directly into SourceTree. SourceTree is a stand-alone graphical user interface for Git and Mercurial, which, like Bitbucket itself, is also a product from Atlassian.

The right margin lists recent activity on the project, with the latest on top. In Figure 18 you can see that among other activity, a pull request was recently received, approved, and merged in this project.

If a Bitbucket project is public, it is discoverable by searching. Once an interesting project is found, you can fork it, make your own changes, and create a pull request if you want the owners of the project to consider merging your contribution. Although the Bitbucket interface looks different than GitHub, the process of creating clones, forks, and pull requests is conceptually the same as it is for GitHub.

FoxBin2Prg

FoxBin2Prg is a VFPX project created by Fernando D. Bozzo to help VFP developers manage their binary source code files on version control systems.

The primary advantage of FoxBin2Prg over tools like SCCTEXT is that FoxBin2Prg enables two-way conversions. Not only can it generate text-equivalent file from VFP binaries (SCX, VCX, etc.), but more importantly it can also regenerate the binaries from their text equivalents. An additional benefit is that FoxBin2Prg generates PRG-style text files. This makes it easy for developers to see and understand the differences between two revisions of the same file because they're working with a familiar file structure.

FoxBin2Prg is not tied to any particular DVCS, so it can be used with equal effectiveness whether you're using Git, Mercurial, or some other version control system.

Installing FoxBin2Prg

If you are using Thor, the best way to install FoxBin2Prg is from Thor's Check for Updates menu. If you are not using Thor, download FoxBin2Prg.zip from VFPX and extract it into the folder of your choice. FoxBin2Prg is released as an EXE and also comes with full source code.

The FoxBin2Prg homepage at http://vfpx.codeplex.com/wikipage?title=FoxBin2Prg on VFPX includes basic information for getting started. At a minimum, read this one before installing and using FoxBin2Prg.

The *FoxBin2Prg Internals and Configuration* page has more extensive documentation at http://vfpx.codeplex.com/wikipage?title=FoxBin2Prg%20Internals%20and%20Configuration&referringTitle=FoxBin2prg. Read that one for a more thorough explanation of how to use and control FoxBin2Prg.

Integrating FoxBin2Prg into your workflow

When working with a DVCS it's often necessary to merge two or more sets of changes to the same source code file. Binary source code files like PJX, SCX, VCX, FRX, and MNX files cannot be diff'd or merged, so text-equivalent files are needed to support those functions. After merging, the VFP binary files must be regenerated so you can work with them in the VFP IDE. These two steps need to be integrated into your daily development workflow.

Convert VFP binaries to text files

Before committing your changes to the local repository, use FoxBin2Prg to generate text equivalent files for all new or modified VFP binary files. In FoxBin2Prg, the text equivalent files have the same name as the binary but with a .??2 file name extension. For example, the text equivalent file for a VFP form named MyForm.scx is MyForm.sc2. The FoxBin2Prg documentation refers to these collectively as the "tx2" files, so I'll do the same here. The file name extensions are configurable in the FoxBin2Prg.cfg file in case you need to use something different.

The tx2 files contain all the information from all the files comprising a VFP binary. For example, MyForm.sc2 contains all the information from MyForm.scx and MyForm.sct.

Be sure to add the tx2 text files to your repository and include them in your commits. You'll need them when you want to merge changes into your own working copy, and other developers will need them when they want to pull your changes to merge with their own.

Convert text files back to VFP binaries

The tx2 text files support the ability to perform diff and merge operations on two revisions of the same file. For example, you can pull someone else's changes to MyForm.sc2 from another repository and merge them with your local copy of MyForm.sc2. If there are no merge conflicts, you end up with a clean text file. If a merge conflict is detected, you can resolve it by making changes to the merged text file. Either way, the VFP binaries can then

be regenerated from the merged text file so you can go on to work with MyForm.scx in the VFP IDE as usual.

There is more than one way to use FoxBin2Prg. How you integrate it into your workflow will depend on which way you choose.

Using FoxBin2Prg with "Send To" shortcuts

One way to use FoxBin2Prg is to create shortcuts in your Windows "Send To" folder. This enables you to pass a file name to FoxBin2Prg as a parameter via the Windows Explorer context menu. Three types of shortcuts are possible: binary to text, text to binary, and the "interactive" mode. In interactive mode, FoxBin2Prg determines what action to take based on the type of file you send to it, so you don't really need the other two.

You can configure your shortcuts to use foxbin2prg.exe, which has no dependencies other than filename_caps.exe and filename_caps.cfg (which are included with FoxBin2Prg), or you can configure them to use foxbin2prg.prg, in which case all of its supporting files must also be present in the same path.

The FoxBin2Prg documentation provides the details on how to create these shortcuts. **Figure 19** shows three shortcuts in place in the "Send To" in my Windows user profile folder.

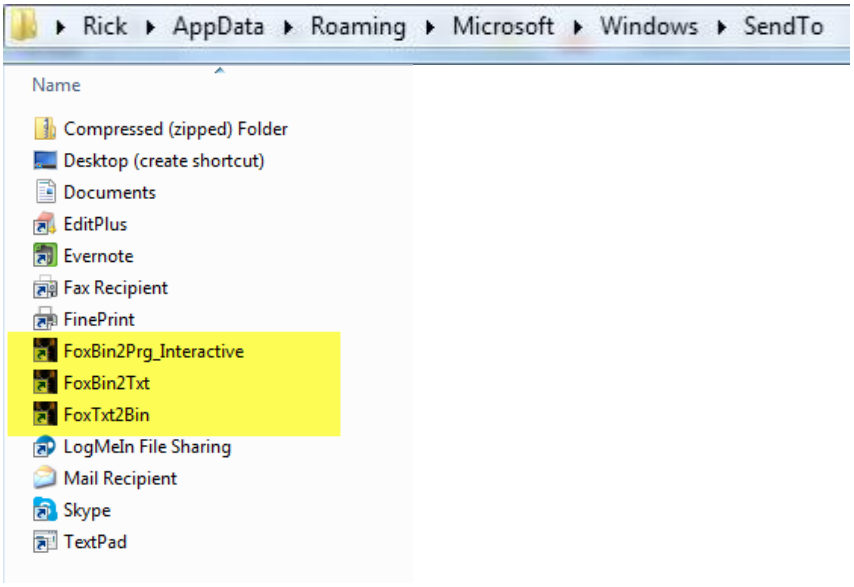

Figure 19. Adding shortcuts in your Windows "Send To" folder enables you to invoke FoxBin2Prg functions on folders and files from the Windows shell.

You can name the shortcuts whatever you want – what's important is their properties. **Figure 20** shows the property sheet for the *FoxBin2Prg_Interactive* shortcut I created to run FoxBin2Prg in interactive mode. When you right-click on a file name in Windows Explorer and choose Send To | FoxBin2Prg_Interactive, Windows launches FoxBin2Prg and passes the selected file name to it. FoxBin2Prg then performs the appropriate conversion depending on the type of file.

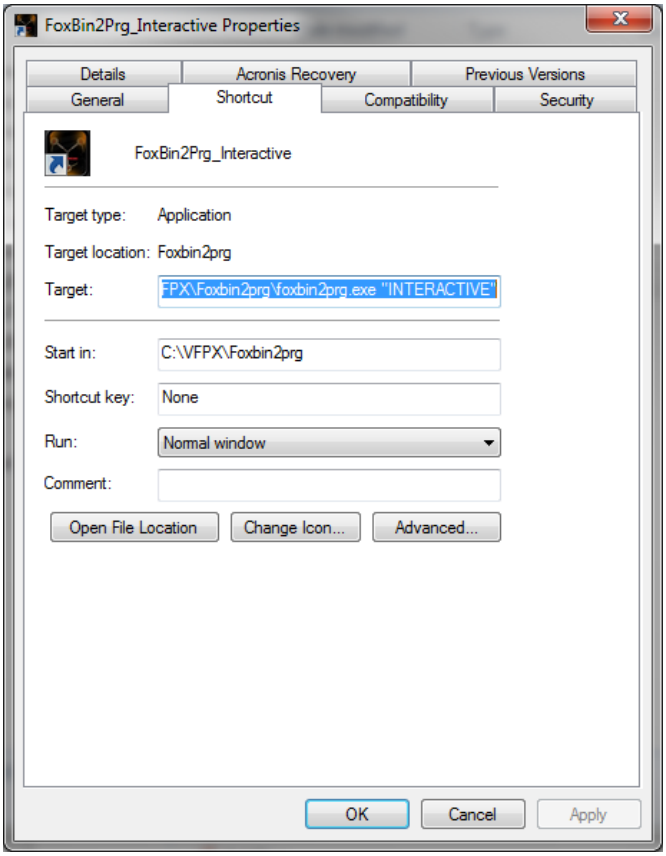

Figure 20. This shortcut in the Windows user profile "Send To" folder launches FoxBin2Prg in interactive mode.

One caveat with this approach is that the Windows "Send To" menu is available only by right-clicking on a file name in the Windows shell – i.e., in the Windows file explorer or a substitute like PowerDesk or Explorer[2]. Unfortunately, neither the Visual FoxPro IDE nor version control GUIs like TortoiseHg or SourceTree are integrated with the Windows shell. This means that in order to use these shortcuts you need to have Windows Explorer or some other shell running alongside your development environment and switch to it in order to invoke FoxBin2Prg from the "Send To" menu.

SourceTree, the DVCS GUI for Git and Bitbucket, allows you to create custom actions. This opens up an intriguing possibility: can you create a SourceTree custom action and tie it to FoxBin2Prg's interactive function? The answer is yes, as shown in **Figure 21**.

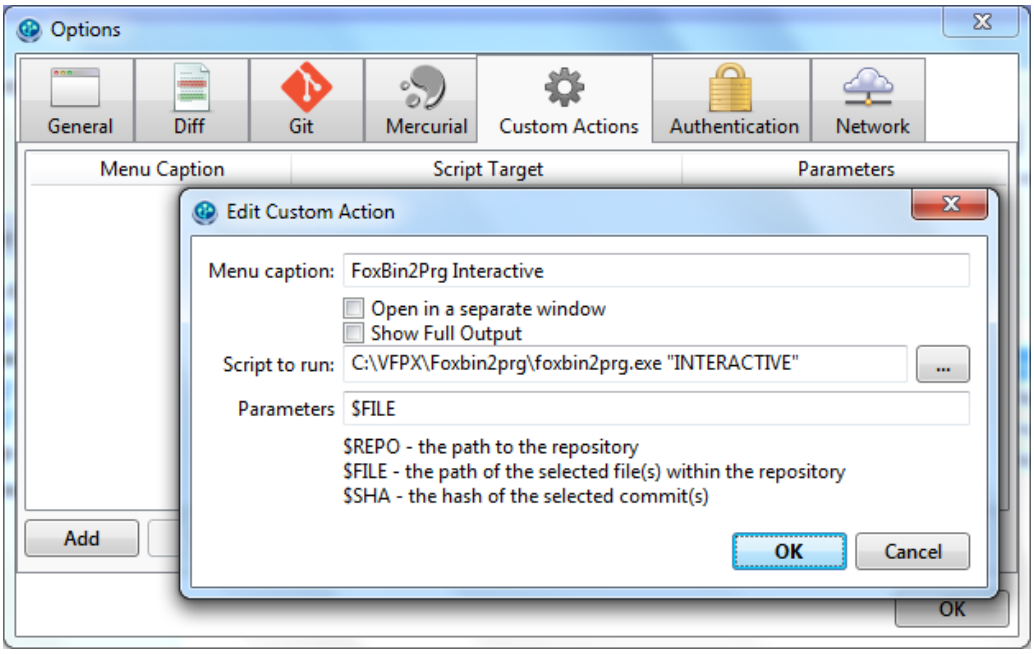

Figure 21. You can create a custom action in SourceTree. This one invokes FoxBin2Prg in interactive mode.

With the custom action in place, you can run FoxBin2Prg on any file in the project by right-clicking on the file name within the SourceTree window as shown in **Figure 22**. If you select a binary file, FoxBin2Prg creates the text equivalent. Select a text equivalent file and FoxBin2Prg creates the corresponding binaries. If you structure your workflow to perform the appropriate conversions just before committing and just after pulling, you can work entirely within the SourceTree window.

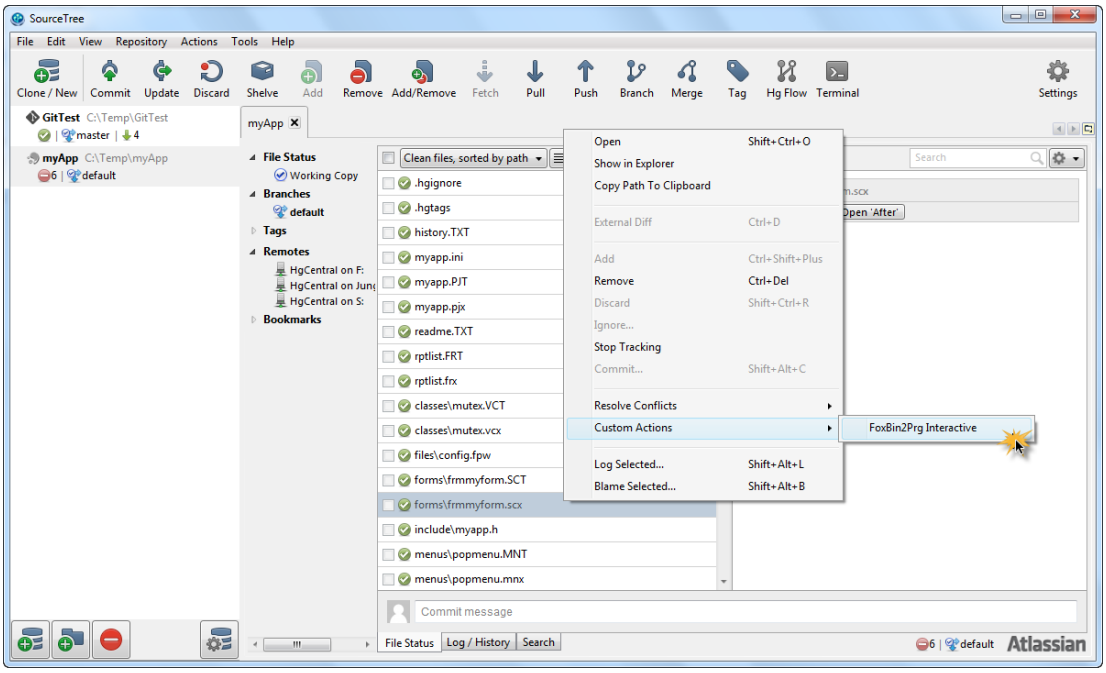

Figure 22. In this example, a custom action is used to invoke FoxBin2Prg in interactive mode on the file named frmMyForm.scx. Because it is a VFP binary file, FoxBin2Prg generates the text equivalent file for it.

Using FoxBin2Prg from Thor

If you installed FoxBin2Prg from Thor, you can invoke its functions at the project level instead of working with individual files. The options are to convert all binary files in a project to text files, convert files with changed time stamps, convert recently changes files, or generate binaries from all text files. These are available from the Thor Tools | Applications | FoxBin2Prg | Projects menu, as shown in **Figure 23**, or from the Thor Tool Launcher if you prefer to work that way.

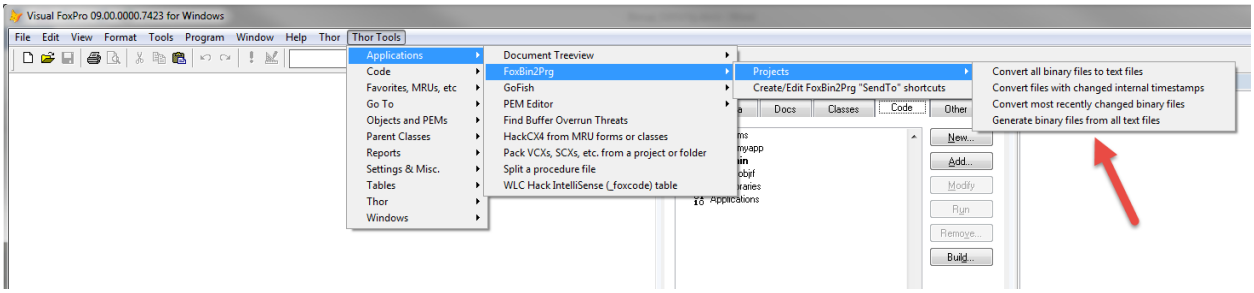

Figure 23. If you installed FoxBin2Prg from Thor, its functions are available from the Thor Tools | Applications menu.

Convert all binary files to text files

This function generates the tx2 text files for all binary files in a project. FoxBin2Prg uses the open project if there is one, otherwise it prompts you to select a project.

Run this function when you first begin using FoxBin2Prg on a project. Be sure the tx2 files are marked to be included in the repository and then do a commit. This defines your starting point for future diffs and merges.

Run this function again after adding or modifying source code files but before committing to the repository. Based on its name, I expected this function to create new tx2 files for all the binaries in the project. What I found, however, is that it generates new text files only for the binaries that are out of sync with their existing tx2 file. For example, if a change is made to only one report in a project that contains several forms and reports, the Convert All Binary Files to Text Files function generates a new tx2 file only for that report and not for other unchanged binaries.

When you rebuild the EXE from a project with the “Recompile all files” option turned on, the external (file system) “Modified” date and time stamp for all the binary files is updated even though no substantive changes were made to the source code. The DVCS therefore detects these as newly modified files, but Convert All Binary Files to Text Files recognizes that no real changes were made.

Convert files with changed internal timestamps

The table structure for Visual FoxPro binary source code files includes columns for a UniqueID and a Timestamp. VFP generates the values for these fields and updates the appropriate rows when the source code is modified in the Designer.

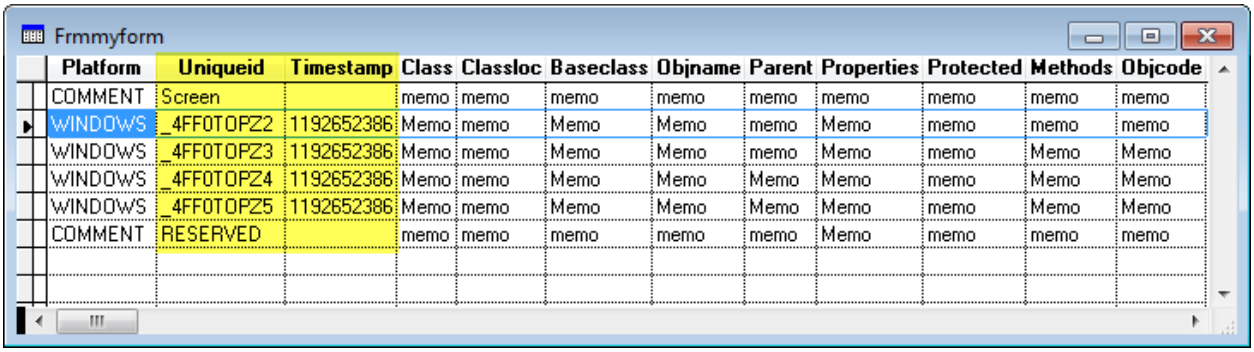

Figure 24. VFP binary source code files, like this one for a form named frmMyFrom, have columns for a UniqueID and a Timestamp.

Figure 24 shows the initial value of these columns in the frmMyForm.scx file. After changing the position of a control in the form, VFP has updated two of the Timestamp values as shown in **Figure 25**.

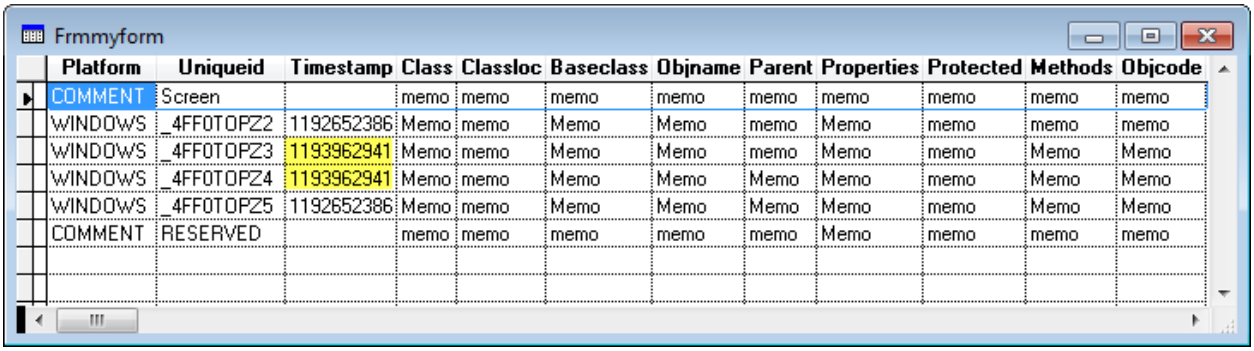

Figure 25. The Timestamp values in two rows of the SCX file have been updated after changing the position of a control on this form.

FoxBin2Prg's "Convert files with changed internal timestamps" function looks at the timestamps in each row to determine if the file has been changed. If so, it generates a new tx2 file, otherwise not.

As pointed out in the description of this function in Thor's Tool Launcher, a modified source code file could be overlooked by this FoxBin2Prg function if you modified the SCX with a tool like HackCX without updating the Timestamp column. In this case it's recommended to use the "Convert most recently changed files" function.

Convert most recently changed files

This function generates the tx2 files for binary files that have recently changed. You get to decide what recent means, either in terms of a number of files or a number of days (see **Figure 26**). This function converts files that have been modified even if the value in the Timestamp column has not changed.

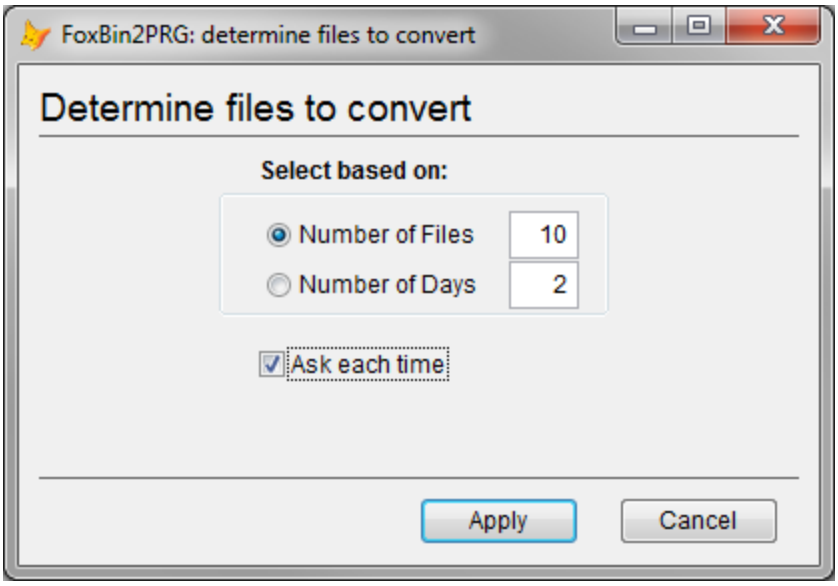

Figure 26. You get to decide what recent means, either in terms of the number of files or the number of days.

For example, if HackCX4 is used to modify the value of the Top property for a label in a form, the values in the Timestamp column of the affected rows of the SCX file do not change. Therefore the process of generating new tx2 files based on changes to Timestamp values would bypass that form.

To be on the safe side in situations when there is any doubt about which files will get converted, I suggest using the "Send To" shortcut method to explicitly generate new tx2 files for any source code file you've modified outside of its Designer.

Generate binary files from all text files

This function does exactly what it says – it generates all the binary files from their corresponding tx2 file. The original binary files are preserved by renaming them with a .bak extension, e.g., MyForm.sct.bak and MyForm.scx.bak.

Run this function after pulling and merging changes into the tx2 files in order to ensure that all binaries are back in sync with the merged text files. If you do not want to generate new binaries for all files, use the "Send To" shortcut method to convert only the one(s) you know have been changed.

Using FoxBin2Prg programmatically

Another way to use FoxBin2Prg is use it programmatically as an object. FoxBin2Prg provides an API for this purpose, which is explained in its documentation.

One developer who has already done this is Mike Potjer, and fortunately for us Mike has shared his work with the VFP community on Bitbucket and Thor.

Git Utilities for VFP

Mike Potjer has written a cool tool for using Git with VFP. Git Utilities integrates with FoxBin2Prg to provide a menu-drive workflow for pre-commit and post-commit actions. Git Utilities is available as a Thor update and can also be found on Bitbucket at https://bitbucket.org/mikepotjer/vfp-git-utils. Read the documentation (which also includes a good introduction to setting up Git) before installing and using Git Utilities.

Git Utilities installs as an item on the Thor Tools | Applications menu. The available functions are Post-checkout file synchronization, Prepare for Git commit, Restore file timestamps, Save file timestamps, and Show Git repos in the project.

Inspecting the Git Utilities source code is a good way to see how the FoxBin2Prg API can be used. I have only begun to work with Git Utilities, so I can't comment on it other than to say it looks very interesting and I intend to pursue it further.

Summary

Git and Mercurial are both distributed version control systems (DVCS). They share many similarities but also have some significant differences. Now that you've learned something about both, what do you think? Is one better than the other? If you're trying to decide between the two, which one should you chose? Can they both be installed on the same machine without interfering with one another?

The answer to the last question is "Yes". The answer to the other questions, of course, is that it's up to you.

If you're already using one of them, I see no reason to switch to the other unless you're working on a project that requires it. To me as a Windows developer, Mercurial has a more natural feel, but that's partly because I have no experience with Linux so some things in Git seem foreign. If installation size is a consideration, Mercurial with TortoiseHg (61.7MB for 356 files in 72 folders) is certainly a smaller installation than Git for Windows (404MB for 5,167 files in 577 folders) even without a 3rd party GUI, but with today's large hard drives that's probably not a concern.

Whether you choose Mercurial or Git, you can work from the command line or by using one of the available graphical user interfaces, or both. Using the command line is the best way to learn, but as a VFP or Visual Studio developer you will most likely feel more comfortable using a GUI once you're comfortable with the basic workflow.

GitHub and Bitbucket are cloud-based commercial hosting services designed to make it easy to share code with others and to work collaboratively as part of a geographically dispersed team. Projects hosted on these services can be either public or private.

The FoxBin2Prg project on VFPX generates text equivalent files from VFP source code binaries, enabling developers to perform diff and merge operations that cannot be done on the binary files. The true power of FoxBin2Prg lies in its ability to re-generate the binaries from the text file. There are several ways to use FoxBin2Prg, including from Thor.

Git's rapidly rising popularity is due in part to it being adopted by the Visual Studio community, and Microsoft itself has released integrated tools in support of this. I know many VFP developers also work with C# and Web technologies in Visual Studio, so the references section of this paper includes several links to resources for using Git with Visual Studio.

Although it may currently lag behind Git in popularity, Mercurial is still widely used and is being actively maintained and enhanced by its developers. Both are fine choices for developers who want to adopt a DVCS into their development process.

Biography

Rick Borup is owner and president of Information Technology Associates, LLC, a professional software development, computer services, and information systems consulting firm he founded in 1993. Rick earned BS and MBA degrees from the University of Illinois and spent several years developing software applications for mainframe computers before turning to PC database development tools in the late 1980s. He began working with FoxPro in 1991, and has worked full time in FoxPro and Visual FoxPro since 1993. He is a co-author of the books Deploying Visual FoxPro Solutions and Visual FoxPro Best Practices For The Next Ten Years. He has written articles for FoxTalk and FoxPro Advisor, and is a frequent speaker at Visual FoxPro conferences and user groups. Rick is a Microsoft Certified Solution Developer (MCSD) and a Microsoft Certified Professional (MCP) in Visual FoxPro.

Bibliography

[1] Scott Chacon & Ben Straub, *Pro Git (Second Edition)*, Apress
https://progit2.s3.amazonaws.com/en/2015-08-05-a4623/progit-en.651.pdf

Other References

Getting started

Getting Started with Git and Mercurial
https://www.atlassian.com/dvcs/overview/dvcs-options-git-or-mercurial

How to Use GitHub for Beginners
https://www.youtube.com/watch?v=E8TXME3bzNs

GitHub Guides
https://guides.github.com/

Mercurial for Git Users
https://mercurial.selenic.com/wiki/GitConcepts

The Hg-Git Plugin for Mercurial
http://hg-git.github.io/

Pro Git book online
https://git-scm.com/book/en/v2/

Git Command Reference online
https://git-scm.com/docs

Installing GitHub Desktop
https://help.github.com/desktop/guides/getting-started/installing-github-desktop/

Using Git with Visual Studio

GitHub Extensions for Visual Studio (VS2015, Phil Haack)
http://www.youtube.com/watch?v=a_kh1JJCNMQ

GitHub Inside Your Visual Studio (VS2015, Phil Haack)
http://haacked.com/archive/2015/04/30/github-in-your-visual-studio/

Download and install the GitHub for VS2015 extension from
https://aka.ms/ghfvs
https://visualstudiogallery.msdn.microsoft.com/75be44fb-0794-4391-8865-c3279527e97d
https://visualstudio.github.com/

The Open Sourcing of the GitHub Extension for Visual Studio (Phil Haack)
http://haacked.com/archive/2015/07/20/ghfvs-oss/

Setting up GitHub to work with Visual Studio 2013 Step-by-Step
http://michaelcrump.net/setting-up-github-to-work-with-visual-studio-2013-step-by-step/

Use Visual Studio and Team Foundation Server with Git
https://msdn.microsoft.com/Library/vs/alm/Code/git/overview

Microsoft Application Lifecycle Management: Create, Connect, and Publish using Visual Studio with Git
http://blogs.msdn.com/b/visualstudioalm/archive/2013/02/06/set-up-connect-and-publish-using-visual-studio-with-git.aspx

Visual Studio 2013 Integration
https://github.com/techtalk/SpecFlow/wiki/Visual-Studio-2013-Integration

Tutorials: Getting Started on Windows with Visual Studio
https://github.com/mono/MonoGame/wiki/Tutorials:-Getting-Started-on-Windows-with-Visual-Studio

Using Git with Visual Studio 2013 Jump Start (video tutorial)
https://www.microsoftvirtualacademy.com/en-US/training-courses/using-git-with-visual-studio-2013-jump-start-8306

Using GitHub with Visual Studio 2010/12/13 (video)
https://www.youtube.com/watch?v=Ijfypw7qJgg

How do I add an existing Solution to GitHub from Visual Studio 2013
http://stackoverflow.com/questions/19982053/how-do-i-add-an-existing-solution-to-github-from-visual-studio-2013

Appendix A

This sample file represents the folders and file types you may typically want version control systems to ignore for Visual FoxPro projects. It can be used either as .gitignore for projects under Git version control or as .hgignore for projects under Mercurial.

Lines beginning with a hash symbol are comments. For example, the entries in the VFP Project Files section are included for reference but are commented out, meaning those file types will not be ignored and therefore will be included in the repository.

The entries are case sensitive, which is why most appear in both lower case and upper case.

```
## Ignore selected files in a Visual FoxPro project.

#-----------------------------------------------------------
# VFP Project Files
#-----------------------------------------------------------
#
# Comment out or remove this section if you want
# these files to be included in the repository.
#
#*.pjt
#*.PJT
#*.pjx
#*.PJX

#-----------------------------------------------------------
# VFP labels, menus, reports, screens, and class libraries
#-----------------------------------------------------------
#
# Comment out or remove this section if you want these
# files to be included in the repository.
#
#*.lbt
#*.LBT
#*.lbx
#*.LBX
#*.frt
#*.FRT
#*.frx
#*.FRX
#*.mnt
#*.MNT
#*.mnx
#*.MNX
#*.mpr
#*.MPR
#*.sct
#*.SCT
#*.scx
#*.SCX
#*.spr
#*.SPR
```

```
#*.vct
#*.VCT
#*.vcx
#*.VCX

#-----------------------------------------------------------
# Common VFP project subfolders
#-----------------------------------------------------------
[Bb]mps/
[Dd]ata/
[Ii]cons/
[Ii]mages/
[Tt]emp/
[Tt]est/
[Ww]izards/

#-----------------------------------------------------------
# VFP database files
#-----------------------------------------------------------
*.dbc
*.DBC
*.dct
*.DCT
*.dcx
*.DCX
*.dbf
*.DBF
*.cdx
*.CDX
*.fpt
*.FPT
*.idx
*.IDX

#-----------------------------------------------------------
# VFP compiled files and other binary code files
#-----------------------------------------------------------
*.app
*.APP
*.dll
*.DLL
*.exe
*.EXE
*.fll
*.FLL
*.fxp
*.FXP
*.mpx
*.MPX
*.ocx
*.OCX

#-----------------------------------------------------------
# VFP backup and other misc files
#-----------------------------------------------------------
```

```
*.err
*.ERR
*.log
*.LOG
*.mem
*.MEM
*.tbk
*.TBK
*.vue
*.VUE

#------------------------------------------------------------
# WLC Project Builder backup files
#------------------------------------------------------------
*.ftk
*.FTK
*.fxk
*.FXK
*.ltk
*.LTK
*.lxk
*.LXK
*.mtk
*.MTK
*.mxk
*.MXK
*.stk
*.STK
*.sxk
*.SXK
*.vtk
*.VTK
*.vxk
*.VXK

#------------------------------------------------------------
# Common graphics files
#------------------------------------------------------------
*.bmp
*.BMP
*.cur
*.CUR
*.gif
*.GIF
*.ico
*.ICO
*.msk
*.MSK
*.png
*.PNG
*.jpg
*.JPG
*.tif
*.TIF
```

```
#-----------------------------------------------------------
# Common documentation and Microsoft Office files
#-----------------------------------------------------------
*.chm
*.CHM
*.doc
*.DOC
*.dot
*.DOT
*.hlp
*.HLP
*.rtf
*.RTF
*.pdf
*.PDF
*.xls
*.XLS
*.xlt
*.XLT

#-----------------------------------------------------------
# Other misc files
#-----------------------------------------------------------
*.bak
*.BAK
*.bat
*.BAT
*.dat
*.DAT
*.old
*.OLD
*.orig
*.ORIG
*.sav*
*.SAV*
*.tmp
*.TMP
*.wav
*.WAV
*.zip
*.ZIP
```