This paper was originally presented at the Virtual Fox Fest online conference in October, 2020. https://virtualfoxfest.com/

Getting Comfortable with Git

Rick Borup
Information Technology Associates, LLC
701 Devonshire Dr, Suite 127
Champaign, IL 61820 USA
www.ita-software.com
rborup@ita-software.com

Have you started using Git but don't feel totally comfortable with it? Are you thinking about getting started with Git but are reluctant to jump in because it looks complicated and scary? Or are you an experienced Git user who feels there are still areas of Git you'd like to know more about? If any of these describe you, this session is for you. Together we'll explore the ins and outs of Git along with some tips and tricks so you'll be able to use Git effectively in your own work.

Introduction

So, you want to get comfortable using Git? Two words:

Use it!

Seriously, the best way to get comfortable with Git is to use it as often as you can. Pick a project you're already working on, put it under Git version control, and use Git every time you work on the project. You'll make some mistakes along the way - that's OK. While you're learning, or when you're exploring some new Git feature you haven't tried before, make a backup of your project folder before you begin and you'll always have a safety net in case things go wrong.

This session covers the parts of Git you'll likely use most often. For some this may be an introduction, for others a review. If you're new to Git this should give you enough to get started. If you're an experienced Git user I hope you find a nugget or two of new information. When you want more depth there are plenty of resources for you. Some of the ones I've found useful are listed in the Resources section at the end of this paper.

So jump in, get going, and good luck along the way!

You will learn

- How Git sees the world
- About command line Git and Git GUI's
- About various branching strategies in Git
- How to research and track revision history in Git
- About 'hunky' development
- What's new in the latest releases of Git

Notes on the syntax of sample code

In some of the sample code for Git commands I use the Visual FoxPro "&&" token as a familiar way to demark inline comments. Inline comments are not valid in Git. Unless otherwise noted, anything following && in a line of sample code for Git in this paper should be interpreted as a comment, not as part of the command.

```
C:\>git init    && this is a comment
```

When a folder name is relevant to a code example, the command prompt is written with the appropriate drive/path/folder, as in *C:\myProject>*. If the folder name is not relevant, the command prompt is written merely as *C:\>* or even just the prompt symbol *>*. This is done to keep the sample code lines as short as possible; it does not necessarily imply the command should be run from the root of the C: drive.

```
C:\myProject>git log  && this line of code should be run from C:\myProject
```



```
C:\>git log  && current working tree folder is assumed

>git log && current working tree folder is assumed
```

If a sample line of code is written without any command prompt, the command prompt is implied.

```
git log  && the > command prompt is implied
```

In examples combining command prompt entries and output from those commands, the commands are always be preceded by a command prompt while the output lines are not.

```
C:\myProject>git init
Initialized empty Git repository in C:\myProject  && output from git init
```

In some cases, blank lines in the actual output from Git have been removed to save space.

The tilde (~)indicates a line continuation, meaning the content is too long to show on one line in this paper. The following is a single line in the Git configuration file:

```
mergetool.BeyondCompare4.cmd="C:/Program Files/Beyond Compare 4/bcomp.exe" ~
                             "$LOCAL" "$REMOTE" "$BASE" "$MERGED"
```

Aliases are convenient shortcuts to longer commands. For example, I use *glog* as an alias for *log --graph --oneline* and *stat* as an alias for *status*. You may see these aliases in some of the sample code. See *Aliases* in the section on *Configuring Git* for information on creating aliases.

Parameters passed to a command sometimes need to be surrounded by quote marks. Single quote marks are OK if you're working in PowerShell, but double quote marks are required when working from the Windows command prompt (cmd.exe).

```
PS> git commit -m 'the commit message'  && single quotes are ok in PowerShell

C:\>git commit -m "the commit message"  && double quotes are required
```

How Git sees the world

As developers, our view of the world is our source code files. We usually see and think about them the way they're organized in the project manager, solution explorer, or whatever other interface we're using in our work. In a typical VFP project, for example, we think about our source code files in terms of the tree structure in the Project Manager, perhaps with subfolders for different types of files. In version control system lingo, the collection of physical source code files on our local hard drive is called the *working copy* or the *working tree*.

Git, on the other hand, does not care much about how we see things. To a certain extent it does not even care that the source code files exist outside the repository. Git sees our source code files as a set of objects and pointers describing the current state of each file

and the way each one has evolved over time. From Git's point of view, the working copy is simply a derivative structure providing a way for humans to view and modify files.

You do not need to understand Git's internals to use it, but it's helpful to know at least a little about how Git operates. Internally, Git stores four kinds of objects: commits, trees, tags, and blobs (binary large objects). You can think of commit, tree, and tag objects as pointers. Blob objects are snapshots of the content of source code files, stored in a compressed format. Git creates a unique, 40-character SHA-1 hash, called the hash ID, for every object and uses these IDs to identify and locate the objects in the repository.

When you modify a source code file and commit it to the repository, Git creates a commit object, a tree object, and one or more blob objects. Among other things, the commit object stores information about the identity of the person making the commit, the date and time the commit is made, and the commit message. Git creates a new blob object for each file that is being committed, capturing the current state of the file. The commit object points to its tree object which in turn points to one or more blob objects.

On disk, Git stores objects in a folder/subfolder structure. The first two characters of each object's hash ID are the subfolder name and the remaining 38 characters are the file name. An internal structure called the index links each file's path and file name in the working tree to the hash ID of its blob object in the repository.

When you make a commit, it is recorded on a branch. If you have not created any other branches the commit is recorded on the default branch, which is called the *master* branch. (You can use a different name if you want to – see the *How to Change the Name of the Default Branch* tip in the *Branching and Merging* section.) An internal reference called the HEAD points to the most recent commit. When you commit a set of modifications to the repository, Git modifies the HEAD pointer so it points to the new commit object.

Figure 1; Linear progression of commits. The HEAD pointer points to the most recent commit.

Although Git uses all 40 characters of each object's hash ID internally, it's customary and more convenient for humans to refer only to the first seven characters. This is called the abbreviated ID. Get used to this – whenever you're viewing the log of changes or need to reference a specific commit, you'll typically see and use the first seven characters of the hash ID. Figure 2 is the same as Figure 1 but shows the first seven characters of the hash IDs as you would see them in Git's abbreviated-form log of changes.

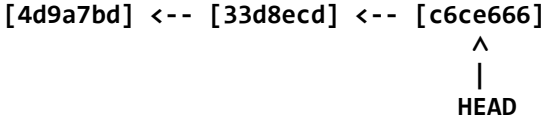

Figure 2; Same series of commit as in Figure 1, identified with their abbreviated hash IDs.

Every commit must have a commit message, which (ideally) describes the modifications made in that commit. Commit messages are created by the developer prior to committing. There are general guidelines for good commit messages – for example, see *How to write a Git commit message* at https://chris.beams.io/posts/git-commit/. You may have your own preference and/or your organization may have its own guidelines. Regardless, writing good commit messages is an essential part of using Git effectively.

Commit messages are shown in the log of changes. Listing 1 is one way to view the log of changes, shown in one-line format with abbreviated commit IDs.

Listing 1: Note the reference to the HEAD pointer and identification of the *master* branch on the top line.

```
>git log --oneline
c6ce666 (HEAD -> master) 3rd commit
33d8ecd 2nd commit
4d9a7bd 1st commit
```

The most recent commit is always on the top line, with older commits following in reverse chronological order. The log continues to grow as long as modifications are being made and committed to the repository.

Git treats the commit message as part of the content of the commit and includes it when creating the commit's hash ID. If you amend a commit, even if only to change its commit message, Git replaces the original commit with a new one using a different hash ID. Never amend a commit once you've pushed it to a remote repository or otherwise shared it with others in any way, because doing so would result in a discontinuity between your version and the other developers' versions of the repository.

Command line Git

You can interact with Git from the command line or by using a Git GUI. There are several Git GUI's to choose from. Even if you end up adopting one for your daily work, it's important to learn and understand command line Git.

On the command line, Git is invoked with the *git* verb followed by a command and most often a series of options and/or parameters. On Windows machines, command line Git can be run from a command prompt or from PowerShell. The new Windows Terminal enables you to use either one from a single interface.

Options

Command line options often have a long form and a short form. The long form is preceded by a double dash while the short form is preceded by a single dash. There is never a space between the dash(es) and the option.

```
git commit -a
git commit --all
```

A capital letter usually indicates forcing a command.

```
git branch -d newfeature   && delete the newfeature branch
git branch -D newfeature   && force deletion of the newfeature branch
```

Parameters

Command line parameters are preceded by a double dash followed by a space followed by the parameter. This should not be confused with the double-dash version of options.

```
git restore -- foo.txt   && restore foo.txt to its unmodified state
```

Sometimes you'll want to use both an option and a parameter.

```
git restore --staged -- foo.txt   && unstage foo.txt
```

Path and file names

Windows uses forward slashes to pass options to commands that take options, such as the */on* option with the *dir* command to list the results in alphabetical order by name.

```
dir *. /on
```

It may not be widely known that Windows supports both forward slashes and backslashes when entering path and file names in the command window. For example, the following command is valid even though the Windows file system uses backslashes.

```
C:\>cd /swfox2020/ComfyWithGit/Code  && command using forward slashes
C:\SWFox2020\ComfyWithGit\Code>      && response displays backslashes
```

Because of its location on the keyboard, I find the forward slash much easier to type than the backslash. You'll see forward slashes used to specify path and file names in commands throughout this paper, except when a backslash is required or appears in the response from the command.

In Git literature it's common to see */path/to/file* used as a proxy for the actual full path and file name, as in

```
git add -- path/to/file
```

where *path/to/file* identifies a specific file such as *Prgs/main.prg*.

Similarly, *pathspec* is used to indicate a file or set of files, as in

```
git add -- <pathspec>
```

where *pathspec* could be something like *Prgs/*.prg*.

Configuring Git

Git uses three configuration files.

- *system* (universal) level – Program Files/Git/etc/gitconfig
- *global* (user) level – users/<username>/.gitconfig
- *local* (repository) level – path/to/repository/.git/config

Note the dot (period) character preceding the name of the global configuration file. In *nix systems, a leading dot means a hidden file. Windows does not hide dot-files.

Git reads and applies settings from the bottom up – local first, then global, then system. Local (repository-level) settings override global (user-level) settings, which in turn override system settings.

Git's configuration files are INI-format files.

```
[section]
property=value
; this is a comment
```

Configurations can be set from the command line or by editing the configuration file directly in a text editor. The following command sets the global username:

```
git config --global user.name 'Rick Borup'
```

The same thing could be accomplished by editing the global configuration file directly.

```
git config --edit --global
```

The section containing the user's name and email address looks like this:

```
[user]
   name = Rick Borup
   email = rborup@ita-software.com
```

You can open any of the configuration files in the default text editor by specifying which one you want to edit.

```
git config --edit --local
git config --edit --global
git config --edit --system
```

The basic settings

The first thing to do after installing Git is to set your username and email address. This lets other developers know who made each commit and how to get in touch with you.

```
git config --global user.name 'Rick Borup'
git config --global user.email 'rborup@ita-software.com'
```

Tools

You may also want to configure Git to use your favorite file editor, diff tool, and merge tool.

File editor

The default file editor on Windows in Notepad. If you prefer a different editor, search the Web for the options you may need to supply for Git to use it. The following shows the setting for Visual Studio Code, with the setting for Notepad commented out.

```
[core]
   ;editor=notepad.exe
   editor = 'C:/Users/rick/AppData/Local/Programs/Microsoft VS Code/Code.exe' --wait
```

Diff tool

The diff tool is used for visual display of diffs between two versions of a file. This does not affect how diffs are displayed in the command line interface in response to the *diff* command, but rather what happens if you run the *difftool* command. The settings for Beyond Compare 4 look like this:

```
[diff]
   guitool = beyondcompare4
[difftool "beyondcompare4"]
   path = C:/Program Files/Beyond Compare 4/bcomp.exe
   cmd = \"C:/Program Files/Beyond Compare 4/bcomp.exe\" \"$LOCAL\" \"$REMOTE\"
```

Merge tool

The merge tool is the visual tool used to resolve merge conflicts. The settings for Beyond Compare 4 look like this:

```
[merge]
   tool = BeyondCompare4
[mergetool "BeyondCompare4"]
   path = C:/Program Files/Beyond Compare 4/bcomp.exe
   cmd = \"C:/Program Files/Beyond Compare 4/bcomp.exe\" ~
         \"$LOCAL\" \"$REMOTE\" \"$BASE\" \"$MERGED\"
```

Other settings

Set *autocrlf* true if your file system uses CRLF line endings (like Windows) but the repository uses LF. This is useful for sharing code among developers who may be working on diverse file systems.

```
git config --global core.autocrlf true
```

Git is case sensitive by nature. The NTFS file system on Windows is not. Setting *ignorecase* true tells Git to treat *somefile.txt* the same as *SomeFile.TXT* or any other variation of the

same name with different casing. This is particularly useful for VFP developers, as VFP is notorious for changing the case of file names and/or extensions during development.

```
git config --global core.ignorecase true
```

I believe *Autocrlf* and *ignorecase* are both set true by default for Git installations on Windows.

Aliases

Aliases are convenient shortcuts for frequently used commands. Aliases are created with the *config* command and are stored in Git's configuration files. For examples, developers like me who were accustomed to working in Mercurial are used to typing *stat* to see the status of a repository. Git requires the full word *status* and does not recognize *stat*, but it's easy to create an alias so it does.

```
git config --global alias.stat status
```

Similarly, you might want to create a shortcut for a frequently used version of the *log* command. My favorite is *log --graph --oneline* but that's a lot to type every time, so creating an alias saves me a lot of keystrokes. I use *glog* (for graphical log) as the alias for that command.

```
git config --global alias.glog 'log --graph --oneline'
```

Aliases can be removed by deleting them from the configuration file or using the config command with the *--unset* option.

```
git config --local alias.foo log      && foo now runs the log command
git config --local --unset alias.foo  && removes foo as an alias
```

Listing configuration settings

The config command's *--list* option provides an easy way to view your configuration settings without having to open and inspect the configuration files.

```
git config --list --global
```

This provides a nicely formatted list of the options in the specified configuration file. Listing 2 is the partial output of the settings on my machine.

Listing 2: Partial output from the *git config --list* command.

```
>git config --list
user.name=Rick Borup
user.email=rborup@ita-software.com
core.autocrlf=true
core.editor='C:/Users/rick/AppData/Local/Programs/Microsoft VS Code/Code.exe' --wait
core.ignorecase=true
merge.tool=BeyondCompare4
diff.guitool=beyondcompare4
```

```
difftool.beyondcompare4.path=C:/Program Files/Beyond Compare 4/bcomp.exe
difftool.beyondcompare4.cmd="C:/Program Files/Beyond Compare 4/bcomp.exe" ~
                            "$LOCAL" "$REMOTE"
mergetool.BeyondCompare4.path=C:/Program Files/Beyond Compare 4/bcomp.exe
mergetool.BeyondCompare4.cmd="C:/Program Files/Beyond Compare 4/bcomp.exe" ~
                             "$LOCAL" "$REMOTE" "$BASE" "$MERGED"
```

Use the *--show-origin* option in conjunction with *--list* to see where each setting is configured. Listing 3 is the partial output from this command.

Listing 3: Partial output from the *git config --list --show-origin* command.

```
>git config --list --show-origin
file:C:/Program Files/Git/etc/gitconfig core.autocrlf=true
file:C:/Program Files/Git/etc/gitconfig core.fscache=true
file:C:/Program Files/Git/etc/gitconfig core.symlinks=true
file:C:/Program Files/Git/etc/gitconfig core.ignorecase=true
file:C:/Program Files/Git/etc/gitconfig pull.rebase=false
file:C:/Users/rick/.gitconfig  user.name=Rick Borup
file:C:/Users/rick/.gitconfig  user.email=rborup@ita-software.com
file:C:/Users/rick/.gitconfig  core.autocrlf=true
```

Getting help

Git is a large and complex piece of software. Fortunately, it comes with an extensive HTML help file. This file is installed on your local machine so it's always available even when you're offline.

To get help for any Git command, simply type *git help* followed by the name of the command to open the HTML page for the requested command in your default browser.

```
git help merge  && open the help page for the merge command
```

Most Git command have a long list of formats and options. You won't need to know them all, but it's nice to be able to call up the help file from the command line whenever you need a refresher or want to explore something new.

Local workflow

Special considerations for VFP

When you're working on a project in the VFP IDE, you're modifying VFP's binary files — the class libraries, forms, reports, labels, menus, and the project manager file itself. If there are different changes to the same file in two different branches of a repository, version control systems attempt to merge them and enable developers to intervene to resolve any conflicts.

The problem with binary files is they can't be merged.

To address this problem, VFP community members have developed and released tools to create text-equivalent versions of a project's binary files. These text-equivalent files can be

merged like any other text-based file, and the binary files can then be re-generated from the merged text file.

The big question for VFP developers is what to include the repository:

- include only the binary files, or
- include only the text-equivalent files, or
- include both the binary files and the text-equivalent files.

Include only the binary files

If only the binary files are included, it is not possible to merge two different sets of changes to the same file. You're forced to choose one or the other and then rely on some form of external communication among the developers to know what changes to incorporate into the result. This is a serious impediment to collaborative development and is therefore the least favorable choice, in my opinion.

Include only the text-equivalent files

If only the text-equivalent files are included, the merge conflict resolution mechanism works but a clone of the repository has none of the binary files needed to work on the project in the VFP IDE. All the binary files would need to be re-generated from their text-equivalents before the project could be modified or the application built for the first time. Going forward, sharing and merging changes among developers via the text-equivalent files works great if each developer remembers to regenerate the binaries for all modified files before working with them in the VFP IDE. There is also a small risk that re-generating the binaries for a file could fail. I've seen that happen, and when it does the text-equivalent file needs to be corrected manually, if that's possible, or the binaries must be recovered from some other source.

Include both the binary files and the text-equivalent files

Including both the binary files and their text equivalents is the safest and most robust choice. The main disadvantage is that the repository grows larger more quickly because updated snapshots of all three files—e.g., myForm.scx, myForm.sct, and the text-equivalent text file for that pair—are stored in the repository every time the file is modified and committed. Including both types of files in the repository solves the problem of a potential mismatch between the binaries and their text-equivalents when updates are shared among developers, but the binaries still require special handling if a conflict occurs during a merge operation. The section on merge conflicts has example of this and what to do about it.

Inspecting differences

After making changes to a source code file, you may want to review how it differs from the original. Git's *diff* command shows what's been changed in a file. If a diff tool such as BeyondCompare or KDiff3 has been configured, it can be invoked from the command line with the *difftool* command. Both commands accept a parameter to specify the file whose changes you want to see.

```
git diff -- <path/to/file>
git difftool -- <path/to/file>
```

Without any other options, Git compares the state of the file in your working tree (the modified file) to the state of the same file in the index as it was most recently committed on the current branch. The differences are shown in *unified diff format*.

The example below illustrates how *git diff* displays changes to a file named *readme.txt*. The old content of the line that was changed is shown in red preceded by a minus sign, while the new content of that same line is shown in green preceded by a plus sign. In this example, only the release date was changed.

```
> git diff -- readme.txt
diff --git a/readme.txt b/readme.txt
index 22f5985..424b9c3 100644
--- a/readme.txt
+++ b/readme.txt
@@ -1,7 +1,7 @@
 What's New in myApp
 -----------------------------------
-Version 9.2.101 - June 19, 2019
+Version 9.2.101 - July 23, 2020
 -----------------------------------
 1. Initial release.
```

If a file has already been staged but not yet committed, you can still view the changes by adding the *--staged* option to the *diff* command.

```
> git diff --staged -- readme.txt
```

You can also use git diff to compare two versions of a file by specifying the commit for each one. For example, if the *develop* branch has a newer version of readme.txt than the *master* branch, you can inspect the differences by comparing *master:readme.txt* with *develop:readme.txt*, like this:

```
git diff master:readme.txt..develop:readme.txt
```

If the versions you want to compare are not on the tip of their respective branches, you can use the commit IDs in place of the branch names, like this:

```
git diff 1b792cc:readme.txt..fc6d326:readme.txt
```

What was changed in a commit?

The *git show* command displays information about an object in the repository. If the object is a commit, *git show* displays what's been changed in the commit. The following example tells Git to show what was changed in commit ID c9d5191. The output begins with log

information about the commit, including the author, date, and commit message, followed by the diff of the modified file(s).

```
>git show c9d5a91
commit c9d5a91c08d56c837849f51d753b47caf712eead (HEAD -> master)
Author: Rick Borup <rborup@ita-software.com>
Date:   Thu Jul 23 17:30:50 2020 -0500
    Update for Southwest Fox 2020
diff --git a/forms/frmmyform.SCT b/forms/frmmyform.SCT
index f074b56..e396461 100644
Binary files a/forms/frmmyform.SCT and b/forms/frmmyform.SCT differ
diff --git a/forms/frmmyform.sc2 b/forms/frmmyform.sc2
index a8678be..871143b 100644
--- a/forms/frmmyform.sc2
+++ b/forms/frmmyform.sc2
@@ -30,7 +30,7 @@ DEFINE CLASS frmmyform AS form
        AutoCenter = .T.
        BorderStyle = 1
-       Caption = "FoxCon 2015 Demo App"
+       Caption = "Southwest Fox 2020 Demo App"
```

To view the changes to a specific file, include the file name as a parameter.

```
>git show c9d5a91 -- readme.txt
commit c9d5a91c08d56c837849f51d753b47caf712eead (HEAD -> master)
Author: Rick Borup <rborup@ita-software.com>
Date:   Thu Jul 23 17:30:50 2020 -0500
    Update for Southwest Fox 2020
diff --git a/readme.txt b/readme.txt
index 22f5985..424b9c3 100644
--- a/readme.txt
+++ b/readme.txt
@@ -1,7 +1,7 @@
 What's New in myApp
 ----------------------------------
-Version 9.2.101 - June 19, 2019
+Version 9.2.101 - July 23, 2020
 ----------------------------------
 1. Initial release.
```

You can also use the alternate syntax of the branch name or commit ID followed by a colon followed by the file name.

```
>git show c9d5a91:readme.txt
```

Inspecting the history of a file

The *git log* command shows the history of commits. Sometimes, rather than seeing the entire log or some portion of it in chronological order, you'd instead like to know which commits modified a specific file. To do this, pass the name of the file as a parameter to the *log* command. The following example shows that *readme.txt* was modified in two commits, c9d5a91 and bd5c751.

```
>git glog -- readme.txt
* c9d5a91 (HEAD -> master) Update for Southwest Fox 2020
* bd5c751 Initial commit
```

To see what changed between those two commits, include their commit IDs as parameters to the *git diff* command.

```
>git diff c9d5a91 bd5c751 -- readme.txt
diff --git a/readme.txt b/readme.txt
index 424b9c3..22f5985 100644
--- a/readme.txt
+++ b/readme.txt
@@ -1,7 +1,7 @@
 What's New in myApp
 ----------------------------------
-Version 9.2.101 - July 23, 2020
+Version 9.2.101 - June 19, 2019
 ----------------------------------
 1. Initial release.
```

Staging

Staging is the intermediate step between making changes to files in your working tree and committing those changes to the repository. You may have made changes to several files and are ready to commit some but not all of them, or you may be ready to commit all of them but want to separate them into two or more commits. Staging enable you to tell Git which file(s) you want to include in the next commit.

Files are staged with the *git add* command. You can add each file individually by name or you can use the *--all* option to stage all new, modified, and deleted files in one step. In Git 2.x the following three commands are equivalent:

```
git add --all
git add -A
git add .
```

The third one, *git add* followed by a space and a period, is the easiest to type and is probably the one preferred by most developers.

Your working tree probably contains certain types of files you don't want to add to the repository – for example, EXEs, DLLs, and others you don't modify directly and whose version history you don't care about tracking. Git uses an exclusion file named *.gitignore* as the place to list the files it should ignore (the leading dot means it's a hidden file in *nix systems). The entries in this file can be individual file names or *pathspecs* like *.exe, *.dll, etc. Folder names should be terminated with a slash character.

The entries are case sensitive, so on a Windows system it's advisable to list both the uppercase and lowercase format for each type of file; for example, *.exe and *.EXE. This is especially true for VFP file name extensions because, with apologies to Forrest Gump, VFP is like a box of chocolates: you never know what you're going to get.

Each working tree can have its own *.gitignore* file, but if there are certain types of files you know you'll never want Git to include in any repository you can create a global .gitignore file that applies to all Git repositories. ZIP files and EXE files might be good examples of this. See Appendix A for how to set up a global exclusion file.

Git uses colors to convey information when you're working from a command line. The *status* command uses red to indicate files that have been modified but not yet staged.

```
> git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   myapp.PJT
        modified:   myapp.PJX
```

Files that are staged and ready to be committed are shown in green.

```
> git add .
> git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   myapp.PJT
        modified:   myapp.PJX
```

Git won't let you stage a file if another process is holding a lock on that file. I most often run into this when I forget to close the project manager in VFP before trying to stage changes that include the project manager files. In the example below, *git status* shows the project files *myApp.PJT and myApp.PJX* are modified and ready to be staged.

```
>git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   myapp.PJT
        modified:   myapp.PJX
```

If you try to stage those files while the project manager is still open in the VFP IDE, Git responds as follows:

```
>git add .
```

```
error: open("myapp.PJT"): Permission denied
error: unable to index file 'myapp.PJT'
fatal: updating files failed
```

The solution is to close the VFP project manager and run *git add* again.

Committing

Every commit requires a commit message. Git accepts almost anything as a commit message, but a good commit message describes the changes in a way that makes it easy for anyone reading the history to understand. There is a good reference on how to write a commit message at https://chris.beams.io/posts/git-commit/,

Commit messages can be a single line or several lines. Single line commit messages are quick and easy, but multi-line commit messages are preferred for all but the simplest commits.

Use the *-m* option to include your commit message on the command line.

```
git commit -m 'fixed issue #365'
```

If you're committing from the command line and leave off the *-m* option, Git opens your default text editor and waits for you to enter the commit message. This is the best way to go if you want to write a multi-line commit message from the command line. Git finishes the commit after you save the file and exit the text editor. Closing the text editor without saving the file aborts the commit.

Git GUIs such as Sourcetree typically provide a space for you to write the commit message directly in their interface, so you do not have to use an external text editor.

Partial commits

Staging selected files

One of the cool things about Git is that you don't have to stage and commit all modified files at once. You can make your commits as granular as you want. For example, if you've modified a form and the project files got updated when you built the EXE, *git status* would show this:

```
> git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   forms/frmmyform.SCT
        modified:   forms/frmmyform.sc2
        modified:   forms/frmmyform.scx
        modified:   myapp.PJT
        modified:   myapp.PJX
```

```
        modified:   myapp.pj2
```

If you decide you want to commit the modified form files but hold off on the project files until later, you can stage only the form files.

```
>git add -- forms/frmmyform.*  && assuming only those three files match that pattern
```

The status command then shows those three files ready to commit while the project files remain modified but not staged.

```
> git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   myapp.PJT
        modified:   myapp.PJX
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   forms/frmmyform.SCT
        modified:   forms/frmmyform.sc2
        modified:   forms/frmmyform.scx
```

Doing a commit at this point commits only the three form files while leaving the project files in their modified state in your working tree.

Stashing changes

Git's stash is exactly what it sounds like – a place where you can stash work in progress without having to commit it to a branch. Files in the stash can be retrieved by popping them off the stash back onto the working tree. If the changes are not needed the stash can simply be cleared without popping it.

Entries in the stash have names. This enables the stash to contain more than one entry at a time. Unless you specify a different name, Git uses the default name *WIP on <branch name>...* for each stash entry it creates, where *WIP* stands for work in progress.

With no options, the *stash* command copies all work in progress—i.e., all modified tracked files in the working tree—to the stash and resets the working tree to a clean state. As an example, say you make some changes to myProgram.prg but do not commit them. The status command shows the modified file.

```
> git status
On branch develop
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   myProgram.prg
no changes added to commit (use "git add" and/or "git commit -a")
```

If you want to clear your working tree but preserve your changes without committing them, you can stash your work in progress, go on to work on something else, and retrieve your earlier work from the stash later. Git generates a default name for each stash entry reflecting the branch you are working on along with the ID and commit message of the HEAD of the branch at the time the stash is created.

```
> git stash
Saved working directory and index state WIP on develop: 7a57f2f Remove unnecessary ~
                                        local memvars
```

After stashing your work in progress, the *status* command confirms your working tree is clean.

```
> git status
On branch develop
nothing to commit, working tree clean
```

The *stash* command's *list* option lists the contents of the stash, while the *show* option shows details.

```
> git stash list
stash@{0}: WIP on develop: 7a57f2f Remove unnecessary local memvars

> git stash show
 myProgram.prg | 279 ++++++++++++++++++++++++++++++-------------------------------
 1 file changed, 173 insertions(+), 106 deletions(-)
```

The *pop* option retrieves work from the stash, reapplies it the working tree, and clears the entry from the stash.

```
> git stash pop
On branch develop
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   myProgram.prg
no changes added to commit (use "git add" and/or "git commit -a")
Dropped refs/stash@{0} (c7cc2e393b9260aa570485d727216ef1463a9143)
```

The stash is useful in many situations, including when you need to pull changes from a remote repository without affecting your work in progress.

Working with hunks

Previous examples showed how staging enables you to commit only selected files from a working tree that also contains other modified files. But what if you've made two unrelated sets of changes to the same file and want to handle them as two separate commits. Can you do that?

Yes – enter the concept of hunks. A hunk is a set of modified lines in a file. Git uses a heuristic to determine what qualifies as a hunk and what separates one hunk from another. You see hunks when you view the output from a *diff* command.

If a file contains more than one hunk of modified lines, you can choose which one(s) to stage using the *-p* (short for *--patch*) option of the *add* command.

```
git add -p
```

Quoting from the Git help page for the add command, the *-p* option tells Git to "Interactively choose hunks of patch between the index and the work tree and add them to the index. This gives the user a chance to review the difference before adding modified contents to the index."

Here's the scenario: you make two sets of unrelated changes to a file without committing. You then decide you'd like to commit those changes separately instead of as one commit. One reason for this would be to achieve a more granular history in the repository.

The assumption is that Git identifies the two sets of changes as different hunks. As far as I know, you do not have any control over this; Git's algorithm decides what constitutes a hunk and where the boundaries are between hunks. Working with a simple text file I found that modifications separated by several intervening lines tend to be treated as separate hunks while modifications separated by only a few lines are treated as a single hunk, but again, Git's heuristics make that determination.

Here an example using a file named *animals.prg* with class definitions for family pets: a cat, a dog, and a fish. Listing 4 is the initial state of file as committed to the repository.

Listing 4: The initial contents of animals.prg.

```
DEFINE CLASS Cat AS Custom
PROCEDURE Speak()
WAIT WINDOW NOWAIT 'meow'
ENDPROC && Speak
ENDDEFINE && Cat

DEFINE CLASS Fish as Custom
PROCEDURE Speak()
WAIT WINDOW NOWAIT 'fish cannot speak'
ENDPROC && Speak
ENDDEFINE && Fish

DEFINE CLASS Dog AS Custom
PROCEDURE Speak()
WAIT WINDOW NOWAIT 'woof'
ENDPROC && Speak
ENDDEFINE && Dog
```

The Dog and Cat classes are then modified so the animal makes a different sound when the animal is happy than when it's not happy. Fish don't speak so no change is made to the Fish

class. In this example the Fish class is arbitrarily placed between the Dog and the Cat classes so Git will treat the changes as two different hunks. Listing 5 is the modified file.

Listing 5: The modified animals.prg. The modified lines are shown in italics.

```
DEFINE CLASS Cat AS Custom
disposition = 'happy'
PROCEDURE Speak()
WAIT WINDOW NOWAIT IIF( this.disposition = 'happy', 'purr', 'hiss')
ENDPROC && Speak
ENDDEFINE && Cat

DEFINE CLASS Fish as Custom
PROCEDURE Speak()
WAIT WINDOW NOWAIT 'Fish cannot speak'
ENDPROC && Speak
ENDDEFINE && Fish

DEFINE CLASS Dog AS Custom
disposition = 'happy'
PROCEDURE Speak()
WAIT WINDOW NOWAIT IIF( this.disposition = 'happy', 'woof', 'grrr')
ENDPROC && Speak
ENDDEFINE && Dog
```

Running *git add* with the *-p* option detects the changes to the Cat class as one hunk and the changes to the Dog class as another hunk, prompting for what to do with each hunk individually. There are several possible responses to the prompt but the only ones to care about for this example are 'y' for yes and 'n' for no.

The Cat hunk comes up first. Reply 'n' to not stash it, i.e., to keep those changes in place for the upcoming commit (see line 14). The Dog hunk comes up next. Reply 'y' to stash it and preserve those changes for the second commit (see line 24).

```
1. > git add -p
2. diff --git a/animals.prg b/animals.prg
3. index a7419be..15dff26 100644
4. --- a/animals.prg
5. +++ b/animals.prg
6. @@ -1,6 +1,7 @@
7. DEFINE CLASS Cat AS Custom
8. +disposition = 'happy'
9. PROCEDURE Speak()
10.-WAIT WINDOW NOWAIT 'meow'
11.+WAIT WINDOW NOWAIT IIF( this.disposition = 'happy', 'purr', 'hiss')
12.ENDPROC && Speak
13.ENDDEFINE && Cat
14.(1/2) Stage this hunk [y,n,q,a,d,j,J,g,/,s,e,?]? y  && yes, stage this hunk
15.@@ -11,7 +12,8 @@ ENDPROC && Speak
16.ENDDEFINE && Fish
17.DEFINE CLASS Dog AS Custom
18.+disposition = 'happy'
19.PROCEDURE Speak()
```

```
20. -WAIT WINDOW NOWAIT 'woof'
21. +WAIT WINDOW NOWAIT IIF( this.disposition = 'happy', 'woof', 'grrr')
22. ENDPROC && Speak
23. ENDDEFINE && Dog
24. (2/2) Stage this hunk [y,n,q,a,d,K,g,/,s,e,?]? n  && no, do not stage this hunk
```

Running the status command at this point shows animals.prg both as a staged file (the version with the changes to the Cat class) and as a modified, unstaged file (the version with the changes to the Dog class).

```
> git stat
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   animals.prg
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   animals.prg
```

Commit the changes to the Cat class.

```
> git commit -m 'Happy cat'
[master a128427] Happy cat
 1 file changed, 2 insertions(+), 1 deletion(-)
```

The remaining modified animals.prg contains the changes to the Dog class. To finish the process, stage and commit it in the usual manner.

```
>git add .
> git commit -m 'Happy dog'
[master 36db477] Happy dog
 1 file changed, 2 insertions(+), 1 deletion(-)
```

The log now shows two separate commits even though you made both changes to animals.prg at the same time.

```
>git glog
* 36db477 (HEAD -> master) Happy dog
* a128427 Happy cat
* 7895482 Initial commit
```

The *stash* command also accepts a *-p* option. As an alternative to the solution above, you could use the *-p* option to stash one hunk, commit the other hunk, pop the stash, and then commit again.

Congratulations, you've master 'hunky' development! See the Resources section of this paper for links to more information about hunks.

Viewing history

The *git log* command

The *git log* command displays the history of commits. There are many variations and options for this command.

```
git log                          && the default format
git log --abbrev-commit          && show abbreviated commit IDs
git log --oneline                && show only one line for each commit

git log --graph                        && show history in graphical format
git log --graph --abbrev-commit        && graphical, abbreviated commit IDs
git log --graph --oneline              && graphical, one line per commit
```

With no options, *git log* displays the full log information.

```
>git log
commit 321dcd1bb3617f5c73d4da3624bbc8ae66da576e (HEAD -> develop)
Merge: 7d9363b 990d100
Author: Rick Borup <rborup@ita-software.com>
Date:   Tue Aug 25 15:39:30 2020 -0500
    Merge branch 'myNewFeature' into develop
```

The *--abbrev-commit* option shows the short SHA-1 IDs instead of the long version.

```
>git log --abbrev-commit
commit 321dcd1 (HEAD -> develop)
Merge: 7d9363b 990d100
Author: Rick Borup <rborup@ita-software.com>
Date:   Tue Aug 25 15:39:30 2020 -0500
    Merge branch 'myNewFeature' into develop
```

The *--oneline* option displays a condensed version of the log with abbreviated commit IDs and only one line per commit.

```
>git log --oneline
321dcd1 (HEAD -> develop) Merge branch 'myNewFeature' into develop
990d100 finished work on my new feature
7cf40a6 still working on my new feature
f0389ab working on my new feature
7d9363b add line 3
3d71ff0 (master) add line 2
993bef2 Initial commit
```

Without the *--graph* option, the log is displayed in linear format, as in the example above. The *--graph* option add branching and merging visualization to the log.

```
>git log --graph --oneline
*   321dcd1 (HEAD -> develop) Merge branch 'myNewFeature' into develop
|\
| * 990d100 finished work on my new feature
| * 7cf40a6 still working on my new feature
```

```
| * f0389ab working on my new feature
|/
* 7d9363b add line 3
* 3d71ff0 (master) add line 2
* 993bef2 Initial commit
```

To see the revision history of a single file, pass the path and filename as a parameter to the *log* command.

```
>git log --oneline -- readme.txt
c9d5a91 (HEAD -> master) Update for Southwest Fox 2020
bd5c751 Initial commit
```

Gitk

Git comes with a visual repository browser called *gitk*. It's a handy way to quickly explore a repository, especially if you're not using a third-party Git GUI. Just run it from the command line in any folder with a Git repository.

```
>gitk
```

Figure 3 is the output for the repository from the hunks example. Clicking on a commit in the upper left panel displays information about the that commit in the lower panels.

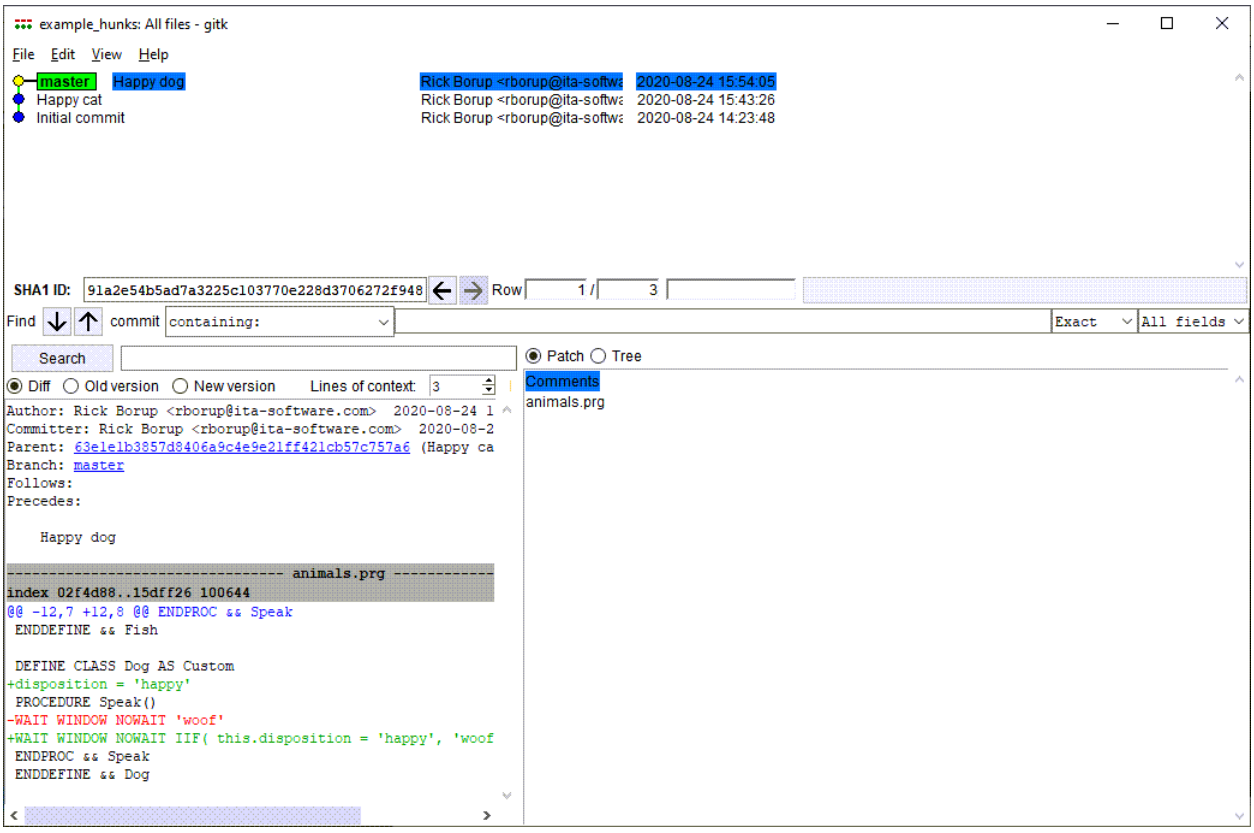

Figure 3: Gitk is a visual repository browser that comes with Git.

Bonus for VFP developers: gitk can also be launched from the *Repository browser* item on the right-click menu in the treeview of Doug Hennig's VFPX Project Explorer. Documentation for gitk can be found at https://git-scm.com/docs/gitk.

Tagging history

A tag is a pointer to a commit. When you create a tag, you give it a name. From then on, unless the tag is deleted, you can always refer to that specific commit by its tag name.

Tags can be used for any purpose. Common uses include marking milestones during development, identifying release versions, etc. Git's log output shows the tag name next to the commit with which it's associated. This makes it easy to identify important steps in the evolution of a project's changes when viewing the log.

Git provides for two types of tags, lightweight and annotated. Lightweight tags have only a name while annotated tags also have meta data such a descriptive message. Suppose you're working on a new project and want to mark the point at which you showed your work in progress to the client. This is a good use for a tag because it marks the point in the project's history where the client last saw the work and you know they haven't seen anything past that point (at least until you do the next demo).

Assuming HEAD points to the commit you want to tag, you can create a lightweight tag named Demo1 like this:

```
git tag Demo1
```

or you can create an annotated tag like this:

```
git tag -a Demo1 -m 'As shown to client on 7/15/2020'
```

If development has proceeded past the commit you wish to tag—in other words, if HEAD no longer points to the desired commit—you can tag an earlier commit by including its ID in the command. The following applies the annotated tag named *Demo1* to the commit whose ID is e26eaeb.

```
git tag -a Demo1 e26eaeb -m 'As shown to client on 7/15/2020'
```

Running the *git tag* command with no arguments displays the list of tags.

```
> git tag
Demo1
```

As noted earlier, Git's *show* command displays information about a Git object. For tag objects it displays full information about the tag along with information about the commit, including the commit message. In the following example the tag message and the commit message are annotated for clarity.

```
> git show Demo1
tag Demo1
Tagger: Rick Borup <rborup@ita-software.com>
Date:   Fri Jul 17 15:36:57 2020 -0500
As shown to client on 7/15/2020  <-- tag message
commit e26eaeb20e171d64a4002c67465ab387b9b03de3 (HEAD -> master, tag: Demo1)
Author: Rick Borup <rborup@ita-software.com>
Date:   Wed Jul 15 11:31:26 2020 -0500
    Last-minute cleanup for demo to client <-- commit message
```

This is followed by the *diff* information for the file(s) modified in that commit.

Tags are not automatically pushed to remote repositories, but you can push them by adding the *--tags* option to the push command.

```
git push --tags origin
```

Branching and merging

A branch is a divergent line of development. Git makes branching easy by using the concept of 'lightweight' branches, meaning branches can come into and go out of existence as needed. Unlike some other version control systems, Git branches can be deleted from the repository when they're no longer needed.

Git users make a distinction between long-lived branches and temporary or short-lived branches. The latter are sometimes called feature branches because they typically exist only for the duration of the development of a particular feature. You may also see them referred to as topic branches.

The *master* branch is an example of a long-lived branch. Every Git repository has a master branch, although in some repositories it may be called *main* or some other name. This branch is automatically created when a repository is initialized. Unless you create other branches, all commits are made on *master*.

How to change the name of the default branch

The default branch in Git has historically been called the *master* branch. Although it has long been possible to change this to another name, most existing literature refers to it as the *master* branch and my guess is most developers still use it that way, too.

These days, however, many developers prefer to use a more neutral name like *main* or *default*. Changing the name of the master branch in an existing repository can be easily done using the *branch* command's *-m* option.

```
git branch -m master main
```

If you're working with a remote repository, make the corresponding change on the remote by pushing the new *main* branch with the *-u* option to update the remote tracking branch.

```
git push -u origin main
```

Scott Hanselman's blog post has additional information including what to do if someone has a local clone of the repository. See https://www.hanselman.com/blog/EasilyRenameYourGitDefaultBranchFromMasterToMain.aspx

That takes care of existing repositories, but what about new repositories? Do you always have to start with *master* as the default and then change it? As of Git 2.28.0, the answer is no.

What's new in Git 2.28.0

Git 2.28.0 introduced a new configuration option to specify a different name for the default branch when creating a new repository. The following example shows how to configure Git to use *main* as the default branch for new repositories going forward.

```
git config --global init.defaultBranch main
```

See the *Highlights from Git 2.28* blog post at https://github.blog/2020-07-27-highlights-from-git-2-28/ for more information about this and other new features.

Creating and switching branches

When you create a branch, you give it a name. You can use any name that conforms to the Git reference format requirements.[1]

Branches are created using the *branch* command. It's important to remember that creating a branch simply creates a new pointer in the Git repository, so you need to check out the new branch before committing to it.

```
git branch develop    && create a new branch named develop
git checkout develop  && switch to the develop branch
```

The *checkout* command is also used to switch back and forth between existing branches. Checking out a branch does two things: it switches to the other branch and it modifies the working tree by restoring files to their state in that branch. Git 2.23.0 introduced two new commands to separate these operations from one another.

What's new in Git 2.23.0

Git 2.23.0 introduced two new commands, *git switch* and *git restore*. These two commands separate functions that are handled as one in the *git checkout* command.

Switch is for branches. It tells Git to change to a different existing branch or, with the *-c* option, to create a new branch and switch to it one step. Like *checkout*, the *switch* command modifies files in the working tree if their content in the branch is different.

```
git switch <branch name>     && switch to existing <branch name>
```

[1] See file:///C:/Program%20Files/Git/mingw64/share/doc/git-doc/git-check-ref-format.html

```
git switch -c <new branch>   && create and switch to <new branch>
```

Restore is for files. It restores a file or files in the working tree to a previous state. The *restore* command takes a *pathspec* as a parameter. Without any other options, the *restore* command restores the specified file(s) to their state in the HEAD of the current branch. This is useful when you want to discard uncommitted changes. The *restore* command modifies the working tree.

```
git restore -- <pathspec>  && restore specified file(s) to their unmodified state
```

If the unwanted changes have already been committed, use *restore* with the *--staged* option to unstage them. This option modifies both the working tree and the index.

```
git restore --staged -- <pathspec>   && unstages the specified file(s)
```

The Git help file has more information about these two commands, and the release notes for v2.23.0 describe the intent behind adding them – see https://raw.githubusercontent.com/git/git/master/Documentation/RelNotes/2.23.0.txt.

When there is more than one branch in a repository, you can switch back and forth among them as needed. For example, after committing changes to the *develop* branch you may want to switch back to the *master* branch and merge *develop* into it.

Git does not allow you to switch branches if the working tree contains modified files that would be overwritten. In this situation you must discard, stash, or commit your modified files before switching to the other branch.

Branching strategies

Development branch

It's common practice, although by no means required, to create a branch named *develop* and to use it for all development work. This separates development work from work that has been released (or is at least ready for release). The *develop* branch is a long-lived branch because it exists for the entire duration of the project.

The idea is to treat the *master* branch as the release branch. All commits are made on the *develop* branch as work progresses, leaving the *master* branch unchanged. When ready to release an update, the *develop* branch is merged back into the *master* branch and the release is built from there. The commit from which the release is built can be tagged with a version number to identify the release in the version history.

Feature branches

Beyond that it's up to each team how to create and employ whatever branching strategy works best for them. Short-lived branches can be created as needed for feature development, hot fixes, experimentation, or any other purpose.

The advantage of a using feature branches is that the *develop* branch remains unchanged until the feature under development is either merged back in or abandoned. Git's lightweight branching model makes it easy to delete a feature branch once work on that feature is complete.

One common practice is to preface the name of a feature branch with the word *feature*, as in *feature-MyAwesomeIdea* or *feature/MyAwesomeIdea*, but this is not required. Using a slash creates a hierarchical representation of the feature branch structure when viewed in some Git GUIs.

Issue branches

For projects where issue tracking is being used, separate branches might be created for the work being done to address each issue. Issue branches are like feature branches, although typically narrower in scope. Issue branches can be created off the development branch or off a feature branch, or wherever else is appropriate. They might have a name like *Issue-23* for work being done to address issue #23. Like feature branches, issue branches are short-lived and can be deleted when work is complete.

Experimental branches

Developers often want to explore an idea to see if it pans out. Doing so may require making changes to existing code or adding new code that won't be kept unless the experiment works out. Git's lightweight branching model enables developers to create branches at any time for any reason. If the work committed on an experimental branch is going to be kept, the branch can be merged back in to mainstream development. If, on the other hand, the work committed on the experimental branch is to be discarded the branch can simply be deleted.

Hotfix branches

It's no surprise that bugs sometimes make it through to release versions of software. By the time a bug is discovered and reported, development may have continued beyond the release version. In this situation, one solution is to switch to the release branch, create a hotfix branch to fix the bug, merge the hotfix branch back into the release branch, and release an update. The hotfix branch can then also be merged into the current development branch to ensure that the bug does not resurface with the next release.

Merging

At some point, work done on a branch typically needs to be brought back into its parent branch. That's what merges are for.

The workflow goes like this:

- Create a new branch for your work and switch to it
- Do some work, making commits to your working branch along the way
- If the work is simply an experiment and you decide not to keep it, switch to the parent branch and delete the working branch.
- If the work is ready to be integrated into mainline development, switch to the parent branch and merge the working branch. The working branch can then be deleted or kept, whichever you prefer.

Example of merging

Before beginning development on a new feature, it's a good idea to check the status of the repository to confirm you're on the desired branch and that there are no uncommitted modified files in your working tree.

```
>git status
On branch develop                       && you're on the develop branch
nothing to commit, working tree clean   && there are no modified files
```

Create and switch to a new branch to work on a new feature.

```
>git switch -c myNewFeature             && create and switch to myNewFeature branch
Switched to a new branch 'myNewFeature'
```

Cycle through the modify-and-commit cycle as many times as necessary.

```
(do some work)
>git add .
>git commit -m 'working on my new feature'

(do some more work)
>git add .
>git commit -m 'still working on my new feature'

(finish the work)
>git add .
>git commit -m 'finished work on my new feature'
```

At some point, the work is done and ready to be merged back into mainline development. Best practice is to view the log to confirm what's been committed and what will be merged. In Listing 6, note that HEAD points to the *myNewFeature* branch (top line) and is three commits ahead of the *develop* branch (bottom line).

Listing 6: The starting point for merging the *myNewFeature* branch into the *develop* branch.

```
>git glog                               && alias for log --graph --oneline
* 4dd37b2 (HEAD -> myNewFeature) finished work on my new feature
* 3771bdc still working on my new feature
* 6d5b398 working on my new feature
* 0d4c6e9 (develop) add line 3
```

Switch to the *develop* branch.

```
>git switch develop
Switched to branch 'develop'
```

Merge the *myNewFeature* branch into *develop*. Note the reference to *Fast-forward* on the third line. We'll come back to that in a minute.

Listing 7: The output from the *merge* command.

```
>git merge myNewFeature
Updating 0d4c6e9..4dd37b2
Fast-forward
 foo.txt | 3 +++
 1 file changed, 3 insertions(+)
```

View the log again to confirm the results of the merge are what you expected. Note that the HEAD reference now points to the same commit for both the *develop* branch and the *myNewFeature* branch.

Listing 8: The after merging the *myNewFeature* branch into the *develop* branch.

```
>git glog
* 4dd37b2 (HEAD -> develop, myNewFeature) finished work on my new feature
* 3771bdc still working on my new feature
* 6d5b398 working on my new feature
* 0d4c6e9 add line 3
```

You can stop here and everything will be fine going forward. The reference to the *myNewFeature* branch will, however, remain in the repository and will continue to show up in the log. If desired, you can get rid of the branch reference by deleting the branch. The *branch* command has a *-d* option for this.

```
>git branch -d myNewFeature
Deleted branch myNewFeature (was 4dd37b2).
```

After deleting the *myNewFeature* branch, history is the same as before only without the reference to the deleted branch.

Listing 9: The log after deleting the *myNewFeature* branch.

```
>git glog
* 4dd37b2 (HEAD -> develop) finished work on my new feature
* 3771bdc still working on my new feature
* 6d5b398 working on my new feature
* 0d4c6e9 add line 3
```

Some developers like to delete references to obsolete branches because it cleans up history. Others feel deleting branches removes a potentially important piece of the history of development. Git doesn't really care, other than to the extent those branch objects continue to exist in the repository, so take your pick.

Types of merges

Git has three types of merges: *fast-forward*, *commit* merge, and *squash* merge. A fourth, the *rebase* merge, is different than the other three and is not discussed here.

Git defaults to a fast-forward merge whenever it can. If there have been no changes on the parent branch to the files being merged, then all Git needs to do is to modify the HEAD reference of the target branch to point to the same commit as the HEAD of the branch being merged. A fast forward merge does not generate a new commit, and you won't see a branching structure reflected in the graphical log because the history of development is linear.

If you want the history to reflect the branching structure, use the *--no-ff* option with the *commit* command. This tells Git not to do a fast forward but to use a commit merge instead. The process begins with the state of the repository in Listing 6 but adds the --no-ff option to the commit command.

Listing 10: Performing a no-fast-forward merge.

```
>git switch develop
>git merge --no-ff myNewFeature
Merge made by the 'recursive' strategy.
 foo.txt | 3 +++
 1 file changed, 3 insertions(+)
```

The log now reflects a branching structure, as show in Listing 11.

Listing 11: The log after the no-fast-forward merge of the *myNewFeature* branch into the *develop* branch.

```
>git glog
*   321dcd1 (HEAD -> develop) Merge branch 'myNewFeature' into develop
|\
| * 990d100 (myNewFeature) finished work on my new feature
| * 7cf40a6 still working on my new feature
| * f0389ab working on my new feature
|/
* 7d9363b add line 3
```

Can the *myNewFeature* branch be safely deleted after a no-fast-forward merge? The answer is yes, although, as before, doing so removes the reference to the branch from history and in the case makes it difficult for anyone reading the history later to know what was done to create the branching structure in the first place.

Listing 12: The log after deleting the *myNewFeature* branch following a merge commit.

```
>git glog
*   321dcd1 (HEAD -> develop) Merge branch 'myNewFeature' into develop
|\
| * 990d100 finished work on my new feature
| * 7cf40a6 still working on my new feature
| * f0389ab working on my new feature
|/
```



```
* 7d9363b add line 3
```

The third type of merge is a squash merge. There may be times when you want to merge a feature branch back into the mainline development work, but you don't want the history to reflect all the steps along the way. As its name implies, a squash merge squashes all the commits in the branch being merged into a single commit on the target branch.

Starting from same point in Listing 6 as the previous two examples, a squash merge is performed as follows.

```
>git switch develop
>git merge --squash myNewFeature
Updating 7d9363b..990d100
Fast-forward
Squash commit -- not updating HEAD
 foo.txt | 3 +++
 1 file changed, 3 insertions(+)
```

The output shows a fast-forward merge was used, but the difference here is that foo.txt, in its final state on the *myNewFeature* branch, has been added to the index as a staged file and still needs to be committed. This can be seen by running the *status* command.

```
>git status
On branch develop
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   foo.txt
```

To complete the squash merge, do a commit in the normal way and provide a commit message.

```
>git commit -m 'add my new feature'
[develop 6727ec6] add my new feature
 1 file changed, 3 insertions(+)
```

The log now shows only the single "squashed" commit instead of the three that were actually used in the development of the new feature. Compare Listing 13 to Listing 12 (the commit merge) and to Listing 9 (the fast-forward merge) to see how the three different types of commits result in different histories for the same set of changes.

Listing 13: The log after doing a squash commit.

```
>git glog
* 6727ec6 (HEAD -> develop) add my new feature
* 7d9363b add line 3
```

Merge conflicts

A merge conflict occurs when there are different changes to the same section of the same file in the two sources being merged. Git can sometimes resolve merge conflicts by itself, but when it cannot then developer intervention is required.

What a merge conflict occurs, it's helpful to think about your version of the code as "ours" and the incoming version of the code as "theirs". There are three ways to resolve a merge conflict:

- Keep your version of the code ("ours")
- Accept the incoming version of the code ("theirs")
- Edit the file and resolve the conflict manually, usually resulting in a combination of "ours" and "theirs".

If you decide to take the first or second option, Git lets you pass one or the other of those two terms as options to the checkout command.

```
git checkout --ours    && keep our version of the file and discard theirs
git checkout --theirs  && keep their version of the file and discard ours
```

If you decide on the third option you can open the file in any appropriate editor to view the conflicting areas and modify them as appropriate. If you like to use a merge tool you can invoke it from the command line. If a merge tool has been configured, Git opens it. Otherwise you can specify the desired merge tool on the command line.

```
git mergetool -- path/to/file
```

Once the conflict is resolved, tell Git to continue the merge.

```
git merge --continue
```

If a merge detects conflicts in more than file, handle each file individually.

If you decide you cannot resolve the conflict or want to defer the merge to a later time, perhaps after you've had more time to think about how to resolve it and/or to consult with other team members, you can abort the merge.

```
git merge --abort
```

Merge conflicts can arise when popping a stash entry, too. In this context, "ours" refers to the version in the working tree while "theirs" refers to the version in the stash. The conflict can be resolved in the same way as when merging branches.

Remote repositories

A remote repository is any repository for a given project other than your local repository. Remote repositories can be located anywhere but are most often hosted on some shared resource—typically an in-house server, a cloud server, or a hosting service such as GitHub or Bitbucket—that is accessible by all the developers working on a project.

Remote repositories are usually bare repositories. A bare repository is a repository with no working tree. It does not need one because people do not work on it directly. Instead, developers commit their work to their local repositories and push it to the remote, from which other developers can then pull.

Remote tracking branches

Git uses *remote tracking branches* as the link between a branch in a local repository and the corresponding branch in a remote repository. It's worth noting that a local repository can contain local branches that do not exist on the remote, and a remote repository can contain branches the local is not tracking – i.e., for which there are no remote tracking branches.

When run with no options, Git's *branch* command lists only local branches, such as *master* and *develop*.

```
>git branch
* develop
  master
  test
```

Include the *-a* (short for *--all*) option to include remote tracking branches in the list. In this example the local repository has remote tracking branches for all three local branches, linked to a remote named *origin*.

```
>git branch -a
* develop
  master
  test
  remotes/origin/develop
  remotes/origin/master
  remotes/origin/test
```

Clones and the origin

The *git clone* command creates a copy of a repository. Anyone with access and the appropriate credentials can clone a repository.

```
git clone <source> [<target>]
```

Git creates remote tracking branches in the local clone linking it back to its source. Git thinks of the source of the clone as the *origin* of the clone and uses that term in the name it assigns to the remote tracking branches in the local repository.

The git remote command lists the remotes to which a local repository is linked. Without any options it simply displays the name of the remote(s).

```
>git remote
origin
```

Add the *-v* option (short for *--verbose*) to see the complete reference to the remote. In this example the remote is named *myRepo* located in a folder named *GitCentral* on the same hard drive as the local repo.

```
>git remote -v
origin  C:/GitCentral/myrepo (fetch)
origin  C:/GitCentral/myrepo (push)
```

Adding and removing remotes

The syntax of the command for adding a link to a remote repository is

```
>git remote add <name> <url>
```

The URL is often an online link to a repository stored on GitHub or another cloud-based service but can also be a simple *drive/path/folder*.

```
>git remote add myRemote path/to/myRemote
>git remote -v
myRemote path/to/myRemote (fetch)
myRemote path/to/myRemote (push)
```

Remotes that are no longer needed can be removed from the local.

```
>git remote remove myRemote
```

Adding and removing remote tracking branches

If necessary, you can create remote tracking branches manually with the *git remote* command.

You might choose to delete a local branch when you're done working with it. If you don't need the test branch in your local repository anymore, you can delete it with

```
>git branch -d test
```

Deleting a local branch does not delete the corresponding remote tracking branch if one exists. In this example the local branch named *test* was deleted but the remote tracking branch remains.

```
>git branch -a
* develop
  master
  remotes/origin/develop
  remotes/origin/master
  remotes/origin/test
```

Add the *-r* (short for *--remotes*) option to the branch delete command to delete a remote tracking branch.

```
>git branch -d -r origin/test
Deleted remote-tracking branch origin/test (was 855931c).
```

```
> git branch -a
* develop
  master
  remotes/origin/develop
  remotes/origin/master
```

Working with remote repositories

The three basic commands for working with a remote repository are *push*, *fetch*, and *pull*.

The *push* command uploads objects from a branch in the local repository to the corresponding branch on a remote repository. The *fetch* command works in the opposite direction, detecting and downloading objects not yet in the local repository from a branch on a remote repository. The objects fetched from the remote are stored in the local repository but are not automatically merged; they must be explicitly merged into the local branch. The *pull* command combines both *fetch* and *merge* into one command.

Push

After work has been committed to a branch on the local repository, the developer can push it to a remote that is linked via a remote tracking branch.

```
>git push origin develop  && push local changes on the develop branch to the remote
```

If the local repository from which the changes are being pushed was cloned from the remote, the link between the two and the remote tracking branch were automatically created by the cloning operation. If not, the link and the remote tracking branch can be created manually, first by adding a remote named origin and then pushing with the *-u* (short for *--set-upstream*) option.

```
>git remote add origin <URL to remote>  && URL can be HTTP or local drive/path/folder
>git push -u origin develop  && push local changes on the develop branch to the
                                develop branch on the origin remote
```

Fetch

Fetch is used to retrieve new objects from a remote repository with merging them into the local. If there are newer objects on the remote, Git downloads them into the remote tracking branch in the local repository and displays information about what was fetched.

In the following example, the last line shows that commits between 0d4c6e9 and 4dd37b2 were fetched from the *develop* branch on the remote and stored in the remote tracking branch named *origin/develop* in the local repo.

```
>git fetch origin develop  && fetch new objects on the develop branch from the remote
remote: Enumerating objects: 11, done.
remote: Counting objects: 100% (11/11), done.
remote: Compressing objects: 100% (5/5), done.
remote: Total 9 (delta 1), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (9/9), 759 bytes | 2.00 KiB/s, done.
From C:/GitCentral/myrepo
   0d4c6e9..4dd37b2  develop    -> origin/develop
```


At this point you may want to examine what was fetched before deciding whether to merge it. A Git GUI may provide a visual way to look at the incoming changes. From the command line, the *log* command can be used to display them. In this example, the *develop* branch is compared to the remote tracking *origin/develop* branch showing four commits were fetched.

```
> git log develop..origin/develop
commit 4dd37b27bb35a488aaf07e8c4b339116b540d5e5 (origin/develop)
Author: Rick Borup <rborup@ita-software.com>
Date:   Tue Aug 25 14:51:16 2020 -0500
    finished work on my new feature
commit 3771bdcaa89c4ee701927d88e5586def42c33f94
Author: Rick Borup <rborup@ita-software.com>
Date:   Tue Aug 25 14:50:40 2020 -0500
    still working on my new feature
commit 6d5b39886daaee80fe79a331ae3c39f21def5451
Author: Rick Borup <rborup@ita-software.com>
Date:   Tue Aug 25 14:49:44 2020 -0500
    working on my new feature
commit 0d4c6e99a97ecc253988827facebcd1222706df4
Author: Rick Borup <rborup@ita-software.com>
Date:   Sun Jul 19 16:50:40 2020 -0500
    add line 3
```

or in short log form

```
> git glog develop..origin/develop
* 4dd37b2 (origin/develop) finished work on my new feature
* 3771bdc still working on my new feature
* 6d5b398 working on my new feature
* 0d4c6e9 add line 3
```

Use the *git merge* command to merge the changes on the *origin/develop* remote tracking branch with the local *develop* branch.

```
> git merge origin/develop
Updating c8c2bd1..4dd37b2
Fast-forward
 foo.txt | 4 ++++
 1 file changed, 4 insertions(+)
```

Pull

The git pull command performs the fetch and merge in a single step.

```
>git pull origin develop
```

The advantage is simplicity. The disadvantage is not knowing what will be merged before it happens.

As with any merge, conflicts may arise when merging changes from a remote.

Git GUIs

There are several GUI tools for working with Git. Git itself comes with one called, not surprisingly, *Git GUI*. You'll find a list of other GUIs for Windows on the Git website at https://git-scm.com/download/gui/win. Standard features include a graphical view of history and a mechanism to stage and commit changes, view differences, create and manage branches, and push and pull from remotes.

You can use a Git GUI by itself or in combination with command line GUI. My personal preference is to use both. Sometimes—in fact, most of the time—I find it easier and more efficient to run Git from the command line. Keep a terminal session open while you work and command line Git is always available. Other times the GUI can be helpful, particularly when dealing with commits involving several files, because you can click on file names to stage, discard, or un-stage them instead of needing to type their individual file names or pathspecs into the command line.

Sourcetree

Sourcetree is a free Git GUI from Altassian. Figure 4 shows how the file history for the repository from the earlier 'commit merge' example looks in Sourcetree. Note the graphical representation of the branch and merge portion of the history.

Sourcetree has both a light and a dark mode. The dark mode is easier on the eyes but doesn't reproduce as well for publication, so these screenshots are in light mode.

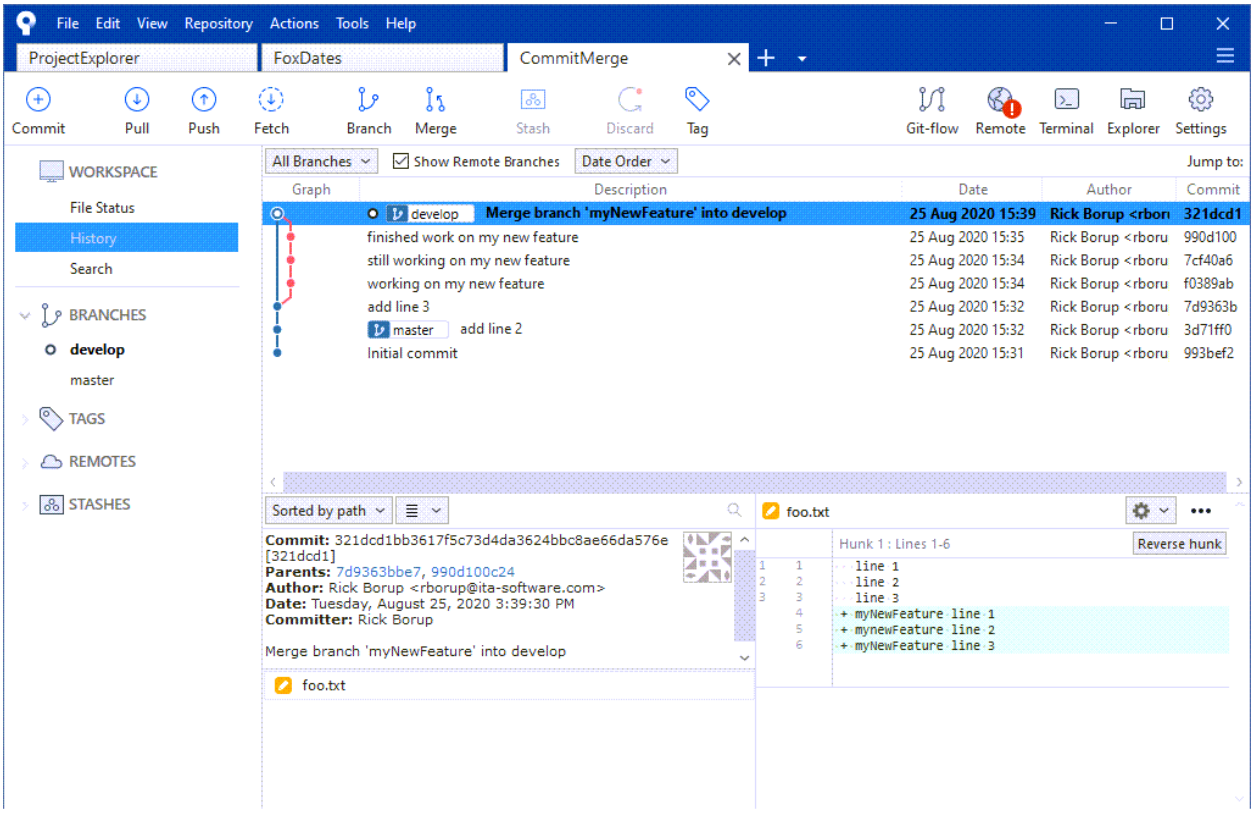

Figure 4: Sourcetree's history pane display the commit log in graphical format.

Click *File Status* in the upper left pane to see the status of the repository. This is equivalent to running *git status* from the command line. In Figure 5, Sourcetree shows that there are no uncommitted files and the working tree is clean.

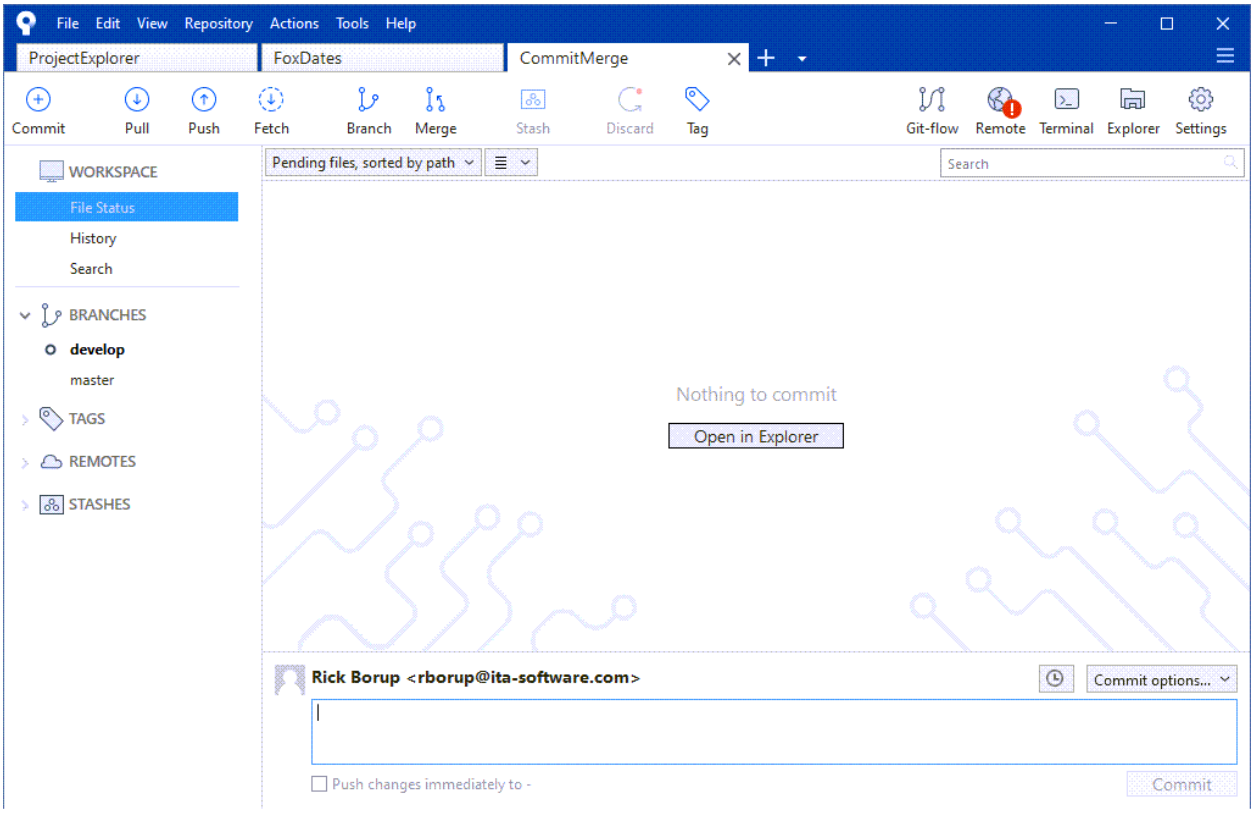

Figure 5: Sourcetree's file status pane shows modified files. In this case the working tree is clean.

After making changes to a file or files in the working tree, Sourcetree *File Status* displays several panes to help you view differences, stage files, and commit them. In Figure 6, foo.txt has been modified but not yet staged so it appears in the *Unstaged files* pane. Clicking on a file selects it. Foo.txt is selected so the diff of its changes is displayed in the upper right. The *Unstaged files* pane also shows a new file, readme.md, that is not yet tracked.

The *Stage All* button stages all unstaged files in a single step. The *Stage Selected* button is useful when you want to selectively stage and commit some but not all the unstaged files. The current branch is shown in bold on the left to help you confirm you're on the branch you want to commit to.

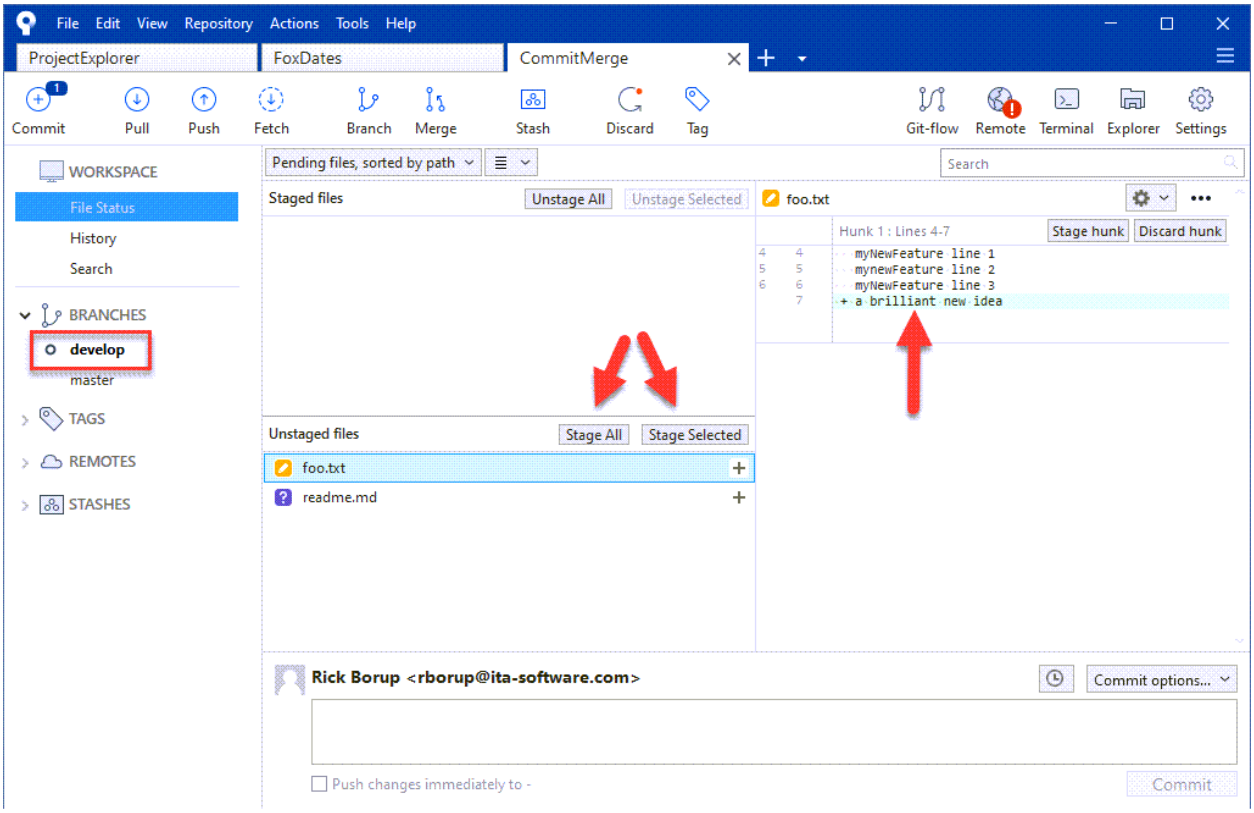

Figure 6: Use the Stage All or Stage Selected buttons to stage the desired files.

One feature of Sourcetree that's especially useful is the ability to create custom actions. VFP developers can for example create a custom action to run FoxBin2Prg from the context menu within Sourcetree when staging changes to be committed.

In my own work the need for this custom action lessened considerably after Doug Hennig integrated Git and FoxBin2Prg into his VFPX Project Explorer, but the custom action is still nice to have available. Figure 7 shows the dialog for creating a FoxBin2Prg custom action in Sourcetree.

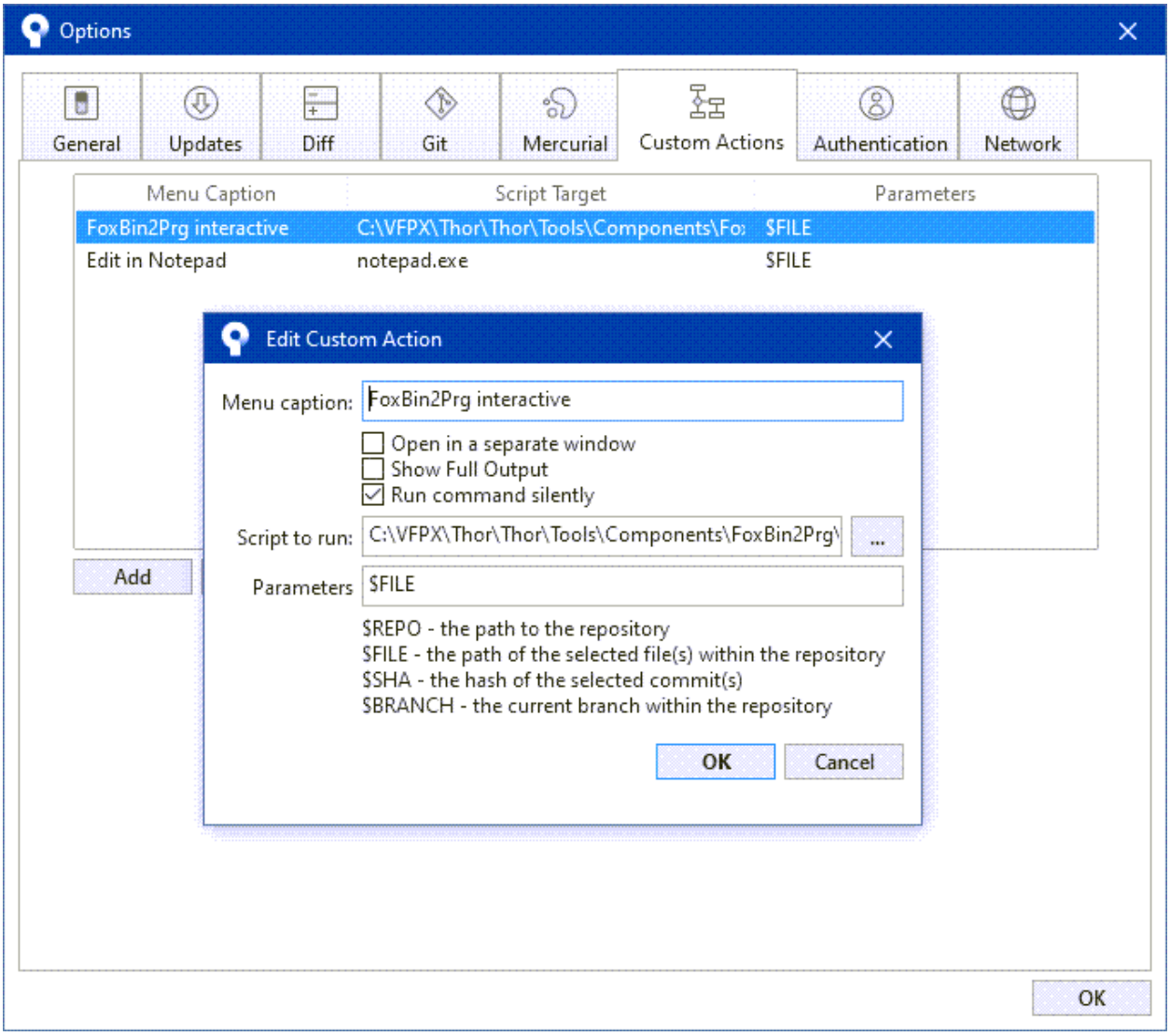

Figure 7: The custom action for running FoxBin2Prg in interactive mode from the Sourcetree context menu.

Sourcetree is available for download from https://www.sourcetreeapp.com/.

Summary

Git can be complicated but it need not be scary. Whether you prefer command line Git or a Git GUI, the key is to use it as often as you can. Appendix B summarizes the Git commands presented in this paper, the ones you'll use most often in your daily workflow. Master these basic operations and you'll be well on your way to getting comfortable with Git.

Resources

Papers

Multi-track Development Strategies in DVCS, Rick Borup, Southwest Fox 2013
http://bit.ly/NHgonB

Version Control Face-off – Git vs Mercurial, Rick Borup, Southwest Fox 2015
http://bit.ly/1VspWTJ

Migrating to Git from Mercurial, Rick Borup, Southwest Fox 2019
http://bit.ly/foo

Online

Pro Git (online version of the book by Scott Chacon and Ben Straub)
https://git-scm.com/book/en/v2

Git Documentation
https://git-scm.com/docs

Git Manual (MAN) pages – local hard drive
file:///C:/Program%20Files/Git/mingw64/share/doc/git-doc/

Atlassian Git Tutorials
https://www.atlassian.com/git/tutorials

Git Branching – Branches in a Nutshell
https://git-scm.com/book/en/v2/Git-Branching-Branches-in-a-Nutshell

Git Do's and Don'ts
https://blog.axosoft.com/git-dos-donts/

15 Git Commands You May Not Know
https://dev.to/zaiste/15-git-commands-you-may-not-know-4a8j

An explanation of all the possible responses to the stash prompt
https://stackoverflow.com/questions/1085162/commit-only-part-of-a-file-in-git/1085191#1085191

What is a hunk?
https://stackoverflow.com/questions/37620729/in-the-context-of-git-and-diff-what-is-a-hunk

http://www.gnu.org/software/diffutils/manual/html_node/Hunks.html

Books

Version Control with Git, Second Edition by Jon Loeliger and Matthew McCullough (O'Reilly), Copyright 2012 Jon Loeliger, ISBN 978-1-449-31638-9

Git Pocket Guide by Richard E. Silverman (O'Reilly), Copyright 2013 Richard Silverman, ISBN 978-1-449-32586-2

Jump Start GIT by Shaumik Daityari, Copyright 2015 SitePoint Pty. Ltd., ISBN 978-0-9943469-2-6 (ebook)

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 published articles in FoxTalk, FoxPro Advisor, and FoxRockX 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.

Appendix A

To configure a global (per-user) exclusion file, create a *.gitignore* file with the usual syntax and store it in your user profile folder, which is C:\Users\<user name> on a Windows machine. This is the same folder where your global Git configuration file is stored.

For example, you can tell Git to ignore ZIP files in any working tree with a *.gitignore* file as follows. Because it's global, you no longer need to include *.zip and *.ZIP in each project's exclusion file.

Listing 14: A .gitignore file to ignore all ZIP files.

```
*.zip
*.ZIP
```

Tell Git where to find the global exclusion file by running the following command from the command prompt:

```
>git config --global core.excludesFile "C:\Users\<username>\.gitignore"
```

The generic format of that command from the Windows command prompt is

```
>git config --global core.excludesFile "%USERPROFILE%\.gitignore"
```

or, if you're working from PowerShell,

```
>git config --global core.excludesFile "$Env:USERPROFILE\.gitignore"
```

Thanks to https://stackoverflow.com/questions/7335420/global-git-ignore for some of this information.

Appendix B

A list of the essential Git commands and when you use them.

```
git init     && to create a new repository

git config   && to tell Git who you are and to configure how it works

git add      && to add a new or modified file to the repository

git status   && to see the current state of the working tree

git commit   && to record a set of changes in the repository

git ls-files && to list files in the index and the working tree

git log      && to see the history of changes

git diff     && to see what's different between two versions

git show     && to see information about a Git object

git branch   && to list branches, create a new branch, or delete an existing branch

git checkout && to switch to another branch

git switch   && to switch to another branch or create a new one

git restore  && to restore a file to a previous state

git merge    && to merge one branch into another

git stash    && to save your changes without committing them

git rm       && to remove a file from the working tree and the index

git clone    && to make a copy of another repository

git remote   && to configure or list remote repositories

git push     && to upload changes to a remote repository

git fetch    && to download changes from a remote repository

git pull     && to downoad and merge changes from a remote repository in one step

git help     && to see the Git manual page for a command
```

