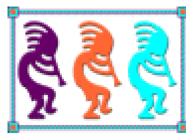
This paper was originally presented at the Southwest Fox conference
in Gilbert, Arizona in September, 2016. http://www.swfox.net



Hands-on Branching and Merging with Distributed Version Control Systems

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

Distributed version control systems: You've heard about them. You've read about them. You've attended some conference sessions. Maybe you've downloaded and installed one, and perhaps you're actually even using it. But for a lot of developers, version control systems are still a strange and scary place when you get past the basics, especially when it comes to branching and merging. What you really want is a hands-on educational experience where you can learn by doing instead of just by watching and listening. If this describes your current relationship

with version control, this pre-conference session is for you! Bring your laptop and be ready to work through a series of structured exercises designed to ease your fears and take you to a whole new level of comfort and proficiency with your version control system.

You will learn

- What distributed version control systems are all about
- The theory behind branching and merging
- How to use SourceTree and TortoiseHg
- How and when to create branches
- How and when to perform merges, and how to resolve conflicts
- How to use a remote repository for solo and/or team development
- How to use the VFPX FoxBin2Prg tool
- How to use the Hg Flow / Git Flow approach to branching and merging

Introduction

This session walks you through a series of structured exercises designed to teach the fundamentals of branching and merging using a distributed version control system. After this session you should expect to walk away with an increased understanding of and comfort level with various ways to use branching and merging in your own work.

The following section is a quick introduction / review of how distributed version control systems work. It covers the basic topics and terminology, and is intended for those who are not already familiar with it. If you are already familiar with this material, feel free to skip ahead to the next section.

How distributed version control systems work

This section is adapted from some of my earlier white papers, including *Version Control Faceoff – Git vs Mercurial*, which is available for download from http://bit.ly/1VspWTJ.

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.

Copyright 2016, Rick Borup

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

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.

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.

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. Mercurial also recognizes *checkout* as a synonym for *update*.

A *remote repository* is any repository for the same project other than your local one. One developer's local repository can serve as a remote for another developer, and vice versa.

However, it's more common for two or more developers collaborating on a project to use a *shared remote repository*.

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.

Once a merge is complete, the working copy may contain modified files that need to be committed to the local repository.

Workflows

There are many ways to use branching and merging with a DVCS. What works best for one developer or one development team may not be best for another. After some experimentation, each developer or team is likely to find and settle on the one approach that works best for them. That approach becomes their *workflow*.

The term *workflow* is often used to differentiate between ways a team can collaborate using a DVCS — examples are peer to peer, centralized, and integrator. In this paper I'm using the term *workflow* in a narrower sense, to mean the developer's or development team's standardized approach to how many branches to use, when to use them, which branches can be merged into which other branches, and when to merge from one branch into another.

By way of comparison, think about an interstate highway interchange as a type of workflow for traffic. A highway interchange workflow can be simple, like the standard cloverleaf shown in **Figure 1**…

Figure 1. A standard cloverleaf is a simple workflow for traffic.

or it can be a bit unusual, like this one in **Figure 2**…

Figure 2. This is an unusual interchange design.

or it can be quite complicated, like the one in **Figure 3**.

Figure 3. This is a more complicated interchange.

Regardless of their differences, all interstate interchanges share the common goal of enabling traffic to branch and merge among the different roads within the highway system. Similarly, all DVCS workflows share the common goal of enabling developers to branch and merge changesets among the different lines of development within a software project.

Think of the roads as branches and the vehicles on the roads as changesets in the repository. Just as vehicles cannot necessarily move directly from one road to any other road, changesets cannot necessarily move directly from one branch to any other branch. Direct access from one road to another in a highway interchange is determined by the interchange's physical design, while in a repository direct access from one branch to another governed primarily by policy.

Linear vs branchy workflows

One of the basic decisions each developer or development team makes is whether or not to use branches within a repository. A workflow that does not use branches within the repository is referred to as *linear development*. In contrast, workflows that employ branches within the repository represent a *non-linear* approach, sometimes referred to as *branchy development*.

In the following sections, we'll look at several different types of DVCS workflows and step though examples so you can see how each one works. Each has its advantages and

disadvantages. The objective is to help you learn the different approaches so you can decide which one(s) would work best for you and your team.

About the exercises

Installing the exercise files on your computer

The zip file that accompanies this session contains the files and folder structure for all the exercises. The recommended location for the exercise files is a folder named C:\SWFox2016\PreCon\Borup. To install the exercise files, simply create a folder of that name on your computer and extract the contents of the zip file into it.

I recommend that you install the exercise files to the suggested location. If you install them somewhere else, the exercises will still work but you'll have to substitute the path you used in place of the recommended path anywhere it appears in this paper.

How the exercise files are organized

This session is organized into lessons, and each lesson has one or more exercises associated with it. The files and other materials for each lesson's exercises are stored in subfolders named for the lesson number — the exercises for Lesson 1 are stored in the folder named Exercise01, the ones for Lesson 2 are stored in the folder named Exercise02, and so on.

There are two read-only zip files in each exercise folder, named *ExerciseNN_begin.zip* and *ExerciseNN_end.zip*, where NN is the exercise number. The first zip file contains the starting files for the exercise. Its contents have already been extracted into the folder for you. If you want to start an exercise over again for any reason, simply delete everything in the folder except the two zip files, then extract the files from the *_begin.zip* file and begin again. The *_end.zip* file contains the completed exercise. When you're done, you can use that file to compare your results to the expected results if you want to.

Because the root path *C:\SWFox2016\PreCon\Borup* is rather long and horizontal space on the printed page is limited, it's omitted when referring to that path in the code samples. So when you see

```
Exercise01>
```

in a code sample you'll know it's actually referring to

```
C:\SWFox2016\PreCon\Borup\Exercise01>.
```

The session code files also come with a folder named HgCentral, which will be used as a remote repository in some of the exercises. The HgCentral folder has a set of subfolders, one for each exercise. Folder HgCentral\Exercise01 is the remote repository for Exercise01, HgCentral\Exercise02 is the remote repository for Exercise02, and so on. Each HgCentral subfolder contains an empty, bare repository. Not all the exercises use a remote repository, but all are provided in case you want to explore on your own.

About the sample source code files

This session is designed to teach you about branching and merging, not about how to write Visual FoxPro code (which of course you already know how to do). For this reason, most of the exercises use a dummy VFP program file named *my.prg*, whose contents are a proxy for real source code. During the exercises you'll make simplistic one-line modifications to my.prg without needing to use valid VFP syntax. The program will never be compiled and isn't intended to do anything, so we don't have to be concerned about using valid VFP syntax. What we care about is the ability to make revisions quickly in order to generate multiple commits to the repository.

Because VFP program files are plain text files, we can use Notepad as the source code editor. Notepad launches much more quickly than VFP, so this approach enables us to move quickly though several steps in each exercise without incurring the overhead of launching or switching to the VFP IDE every time. In a later exercise, however, we will use a sample VFP app to demonstrate how to use FoxBin2Prg to work with VFP binary files in the repository.

About the figures

Mercurial assigns a new changeset ID every time you do a commit. When you work through the examples in the session or on your own, the changeset IDs assigned to your commits will differ from those shown in the code listings and figures in this paper. In some cases, the changeset IDs in the starting code in the exercise folders may also differ from those in this white paper, due to me having re-run the exercise after capturing the screenshots for the figures.

TortoiseHg Workbench

TortoiseHg Workbench is the default graphical user interface for Mercurial. We'll be spending a lot of time in it during most of the exercises, so take a few minutes to become familiar with its various parts, which are annotated in **Figure 4**.

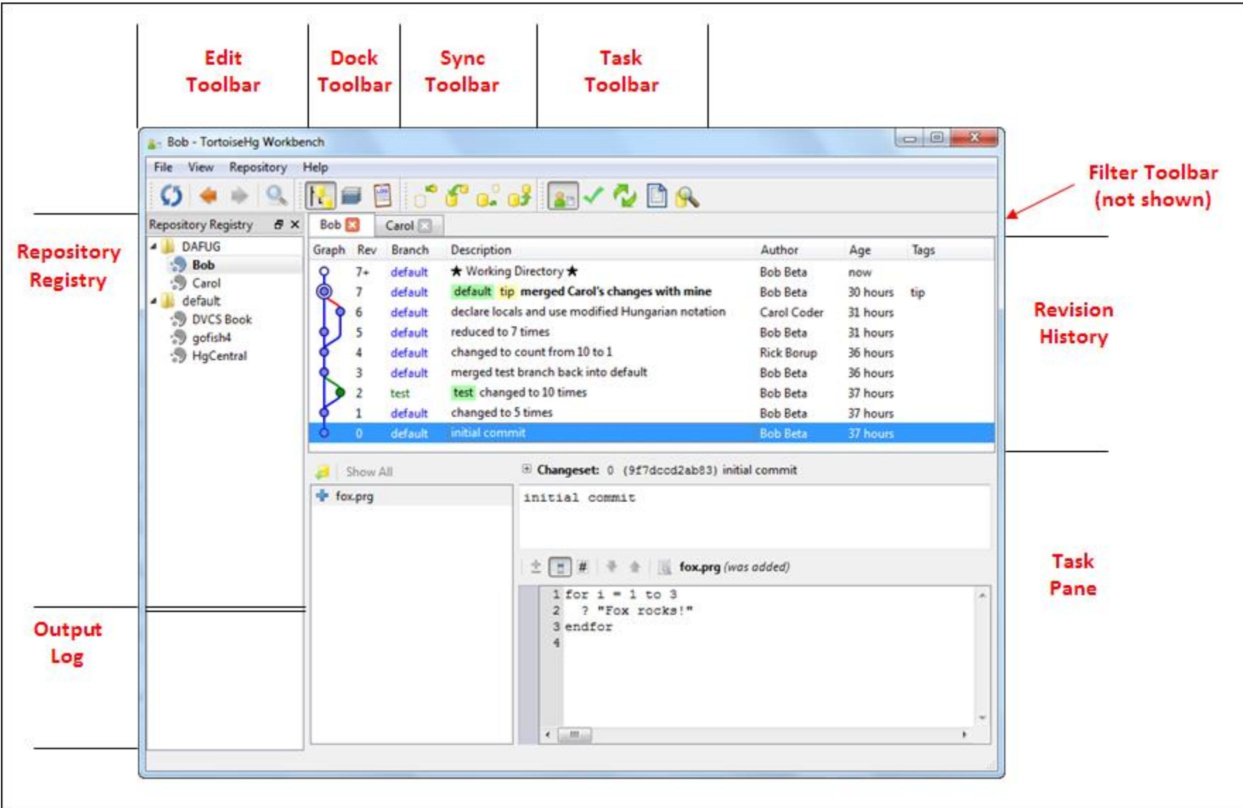


Figure 4. You'll be spending a lot of time in the TortoiseHg Workbench, so take some time to become familiar with its parts as annotated in this figure. This graphic is from an older version of TortoiseHg, but newer versions are essentially the same.

The tree view along the left side of the Repository Registry enables you to organize and group repositories for easy reference. You can add whichever repositories you want to the Repository Registry.[1] The Revision History in the upper main panel is a graphical representation of the revision history for the selected repository. The information displayed in the Task Pane in the lower main panel changes according to the task selected in the Task Toolbar (center top). The TortoiseHg workbench features a tabbed interface so you can have more than one repository open at the same time.

For more information about TortoiseHg, please refer to the section entitled *Working with the TortoiseHg shell extension* on pp. 7 – 14 of my white paper from Southwest Fox 2012, which you can download from http://bit.ly/16j5SIB.

In a later exercise, we'll use SourceTree from Atlassian as an alternative to the TortoiseHg Workbench. More about SourceTree when the time comes.

[1] Despite its name, the Repository Registry has nothing at all to do with the Windows registry. The TortoiseHg Repository Registry information is stored in a file named thg-reporegistry.xml file, located in the TortoiseHg subfolder under common application data.

Lesson 1 – Linear workflow

In a linear, or straight-line, workflow there are no branches. All changes are committed to the default branch in a straight line, each one based on the one immediately preceding it. In Mercurial, the default branch is actually named *default*. In Git it's called *master*.

The primary advantage of a linear workflow is that it's simple. With no branching, there is also no merging. When a modification or a new feature is ready to go, nothing further needs to be done in the repository — the release is simply built from the working copy corresponding to the default branch as it exists after the most recent commit.

The disadvantages of a linear workflow are that it is inflexible, it does not facilitate the concurrent development of two or more new features—especially when it's uncertain which one will be ready to release first—and it does not facilitate hotfixes. While a linear workflow can work well for a solo developer working on simple projects, it is not well suited for more complex projects or for team development.

Exercise 1 - Linear development workflow

Exercise 1 illustrates a simple linear workflow in three steps.

For this exercise, you'll be using the Windows command prompt. Working from the command prompt is the best way to learn the individual Mercurial commands required to perform each step. Although some developers work from the command prompt all the time, developers who are accustomed to working in an IDE like Visual Studio or Visual FoxPro will most likely prefer to work with Mercurial visually as well. For most of the remaining exercises in this session, we'll be using the visual tools TortoiseHg and SourceTree. While these tools enable you to work with your repositories visually, behind the scenes they run the same Mercurial commands you'll be familiar with after using command line.

Exercise 1.1

In this exercise, you initialize a repository, create a new file, add it as a tracked file, commit it to the repository, and view the log. You'll also add the *.hgignore* file, which tell Mercurial which file extension to ignore – in these exercise we want to ignore zip files, among others.

Listing 1. Initialize a Mercurial repository, add the files, perform the initial commit, and view the log.

```
Exercise01>notepad my.prg
  start
  (save the file)
Exercise01>hg init
Exercise01>hg add my.prg
Exercise01>hg add .hgignore
Exercise01>hg commit -m “initial commit”
Exercise01>hg log
changeset:   0:801d9d2bee07
tag:         tip
user:        Rick Borup <rborup@ita-software.com>
```

```
date:        Wed Jul 20 16:32:54 2016 -0500
summary:     initial commit
```

The log shows that there has been a single commit to this repository. The string 0:801d9d2bee07 identifies it as local revision 0 and shows the first twelve characters of the revision's global changeset ID. The tag named *tip* was automatically added by Mercurial. The *tip* tag always indicates the most recent revision in a repository and is automatically moved each time you do a commit.

Exercise 1.2

In this exercise, you make two modifications to the source code file and do a commit after each one.

Listing 2. Modify a file twice and commit the changes each time.

```
Exercise01>notepad my.prg
  start
  some code
  (save the file)
Exercise01>hg commit -m "some code"
Exercise01>hg log
Exercise01>notepad my.prg
  start
  some code
  some more code
  (save the file)
Exercise01>hg commit -m "some more code"
```

Observe that the log now shows the two new commits along with the original one.

Listing 3. View the log to see the new commits.

```
Exercise01>hg log
changeset:   2:87116ded32e7
tag:         tip
user:        Rick Borup <rborup@ita-software.com>
date:        Wed Jul 20 16:34:31 2016 -0500
summary:     some more code

changeset:   1:776a1a4816a8
user:        Rick Borup <rborup@ita-software.com>
date:        Wed Jul 20 16:34:03 2016 -0500
summary:     some code

changeset:   0:801d9d2bee07
user:        Rick Borup <rborup@ita-software.com>
date:        Wed Jul 20 16:32:54 2016 -0500
summary:     initial commit
```

To view the log in graphical format in the command prompt, add the --graph parameter to the log command. You probably won't do this very often, but it's good to know it's possible. From here on we'll be using the TortoiseHg graphical user interface to Mercurial, which

includes a graphical representation of the log as well as the commit messages and other information. Observe that the *default* branch is the only branch in this repository.

Listing 4. Use the --graph option to view the log in graphical format.

```
Exercise01>hg log --graph
```

Another thing to observe in the log is that all the commits are on the *default* branch. The *default* branch automatically exists in all repositories, and is the only branch that exists until you create others. You can use the 'branches' command to find out which branches exist in a repository.

Listing 5. Use the branches command to view the branches in a repository.

```
Exercise01>hg branches
default                        2:87116ded32e7
```

Exercise 1.3

In this exercise, you add a tag to the head revision. Tags enable you to associate a meaningful name with a revision. Later on, you can refer to that revision by its tag name instead of by its revision number or changeset ID.

This product is now ready to be deployed as release 1.0, so let's use the 'tag' command to associate a meaningful release name with the current revision.

Listing 6. Use the tag command to add a tag to a commit.

```
Exercise01>hg tag "Release 1.0"
```

Note that in Mercurial, adding a tag creates a new commit. You can see this by looking at the log again after adding the tag.

Listing 7. In Mercurial, adding a tag creates a new commit.

```
Exercise01>hg log

changeset:   3:7ad4b98ab046
tag:         tip
user:        Rick Borup <rborup@ita-software.com>
date:        Wed Jul 20 16:38:22 2016 -0500
summary:     Added tag Release 1.0 for changeset 87116ded32e7

changeset:   2:87116ded32e7
tag:         Release 1.0
user:        Rick Borup <rborup@ita-software.com>
date:        Wed Jul 20 16:34:31 2016 -0500
summary:     some more code

changeset:   1:776a1a4816a8
user:        Rick Borup <rborup@ita-software.com>
date:        Wed Jul 20 16:34:03 2016 -0500
summary:     some code
```

```
changeset:   0:801d9d2bee07
user:        Rick Borup <rborup@ita-software.com>
date:        Wed Jul 20 16:32:54 2016 -0500
summary:     initial commit
```

The head revision is now revision number 3. Mercurial automatically added a commit message indicating that the tag "Release 1.0" was added and refers to changeset 87116ded32e7, which is revision 2. Note that the *tip* tag has now moved to revision 3.

Exercise 1.4

In this exercise, you launch the TortoiseHg Workbench and observe the results of the previous three steps in a visual format.

1. Open the Windows file explorer

2. Navigate to C:\SWFox2016\PreCon\Borup\Exercise01

3. Right-click on the Exercise01 folder name and choose Hg Workbench from the pop-up menu.

4. Observe how TortoiseHg shows a graphical representation of the revision history along with the textual information pertaining to each commit, as shown in **Figure 5**. Read from the bottom up to track the commits in chronological order.

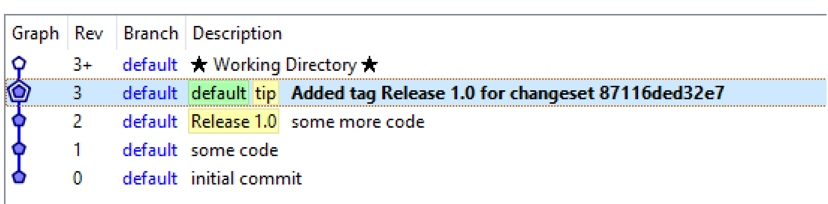


Figure 5. TortoiseHg displays the revision history in graphical format.

TortoiseHg Workbench highlights tags in yellow. Note the *tip* tag on revision 3 and the *Release 1.0* tag on revision 2. This highlighting helps make tags easier to spot when you're viewing a long history of revisions.

Lesson 2 – Using local clones as branches

Branches provide a way to work on a divergent line of development while preserving the integrity of the branch origin. Branches are often used for experimentation, for feature development, for hotfixes, and even for maintaining multiple release versions of a product.

There are many ways to create and use branches. One of the easiest is to create a clone. A clone is a complete and separate copy of the original repository located in a different folder or even on a different machine. Any modifications made in the clone are completely isolated from the original and therefore have no effect on it. If used for experimentation, a clone can simply be discarded when you're done with it. If used for feature development, any modifications made in the clone can be pulled or pushed back into the original to become part of the release version.

Within the clone, you have the same choices regarding linear development versus branchy development as you do within any other repository. Most often you'll probably use linear development within the clone, because the clone itself already represents a branch.

The primary advantage of using clones as branches is that it's simple. Like other forms of branchy development, clones facilitate the simultaneous development of two or more new features and/or hotfixes using separate clones for each. The disadvantage of using clones is that you end up with multiple (if only temporary) copies of the entire repository and the working copy in each location. Development using local clones is not ideal for teams when team members need to collaborate on modifications to the same sections of source code.

A workflow using local clones consists of four main steps:

1. Clone the original repository into a separate location.

2. Do all your work — make modifications and commit them — within the clone.

3. When the modifications are complete, push them back into the original repository or switch to the original and pull them from the clone. Skip this step if the clone was only for experimentation.

4. Delete the clone.

Exercise 2 shows how this works.

Exercise 2

In this exercise you create a clone, modify the source code file, commit it within the clone, and pull the modifications back into the original repository. The files for the beginning of this exercise are a copy of the ones from the end of exercise 1.

From now on we'll be working visually instead of from the command line. To begin, open Windows File Explorer and navigate to the *Exercise02\myapp* folder. It's not a good idea for

a clone to be a subfolder of its origin, so for this exercise the original is in *Exercise02\myapp* and you'll create the clone in *Exercise02\myclone.*

Right-click on the *myapp* folder name and choose *TortoiseHg | Clone…* from the pop-up menu. This brings up the TortoiseHg Clone dialog. A clone has a source and a destination. When opened from pop-up menu in File Explorer, both the source and destination fields in the Clone dialog are populated with the folder you clicked on, as shown in **Figure 6**. At this point, the Clone button is disabled because you cannot create a clone in its own source folder.

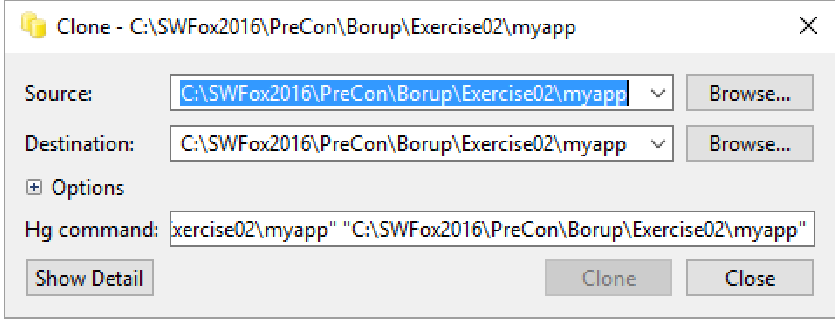


Figure 6. A clone has a source and a destination.

We want the clone to be created in the *myclone* folder. If that folder already existed, you could use the Browse button to select it. In this case, *myclone* doesn't exist yet, so edit the Destination field and replace *myapp* with *myclone*. The Clone dialog is resizable – make it wider so you can see the entire contents of all the fields, as shown in **Figure 7**. Then click the *Clone* button to create the clone.

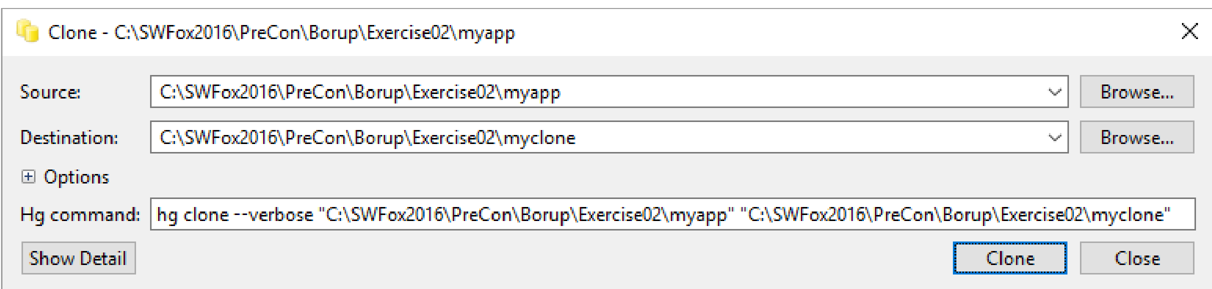


Figure 7. Change the destination to the desired location.

The *myclone* folder now contains a complete clone of *myapp*. Unless you tell it otherwise in the Options portion of the Clone dialog, TortoiseHg automatically updates the clone's working copy to the tip of the repository, so my.prg in *myclone* is the same as my.prg in *myapp*.

After the clone has been created, it's automatically opened in TortoiseHg Workbench. Here you can observe that the revision history is the same as we left it in Exercise 1 (and of course the same as *myapp*, from which it was cloned).

Now we want to add some code to my.prg in the clone. You can open any file directly from TortoiseHg. The working directory is currently clean, meaning there are no uncommitted changes, so my.prg does not show up in the TortoiseHg *Commit* pane. In any repository, you can view all version-controlled files in the *Revision Details* pane (this is called the *manifest*). Click the icon on the toolbar to switch to the Revision Details pane, then in that pane click on the upper left icon to *Show all version-controlled files in tree view* (see steps 1 and 2 in **Figure 8**). Finally, right-click on my.prg and choose *Edit Local* from the pop-up menu to open it in Notepad, assuming you've selected Notepad as the Visual Editor in your TortoiseHg Settings.[2]

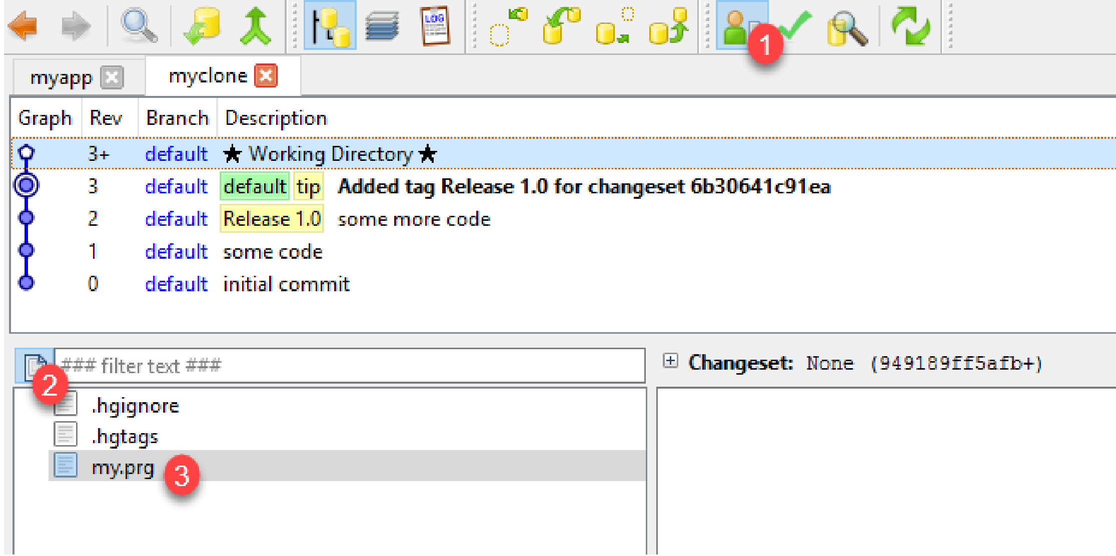


Figure 8. You can open any file directly from the Revision Details pane in TortoiseHg.

In Notepad, insert "added this code in the clone" as a new line at the bottom of my.prg. Save the file and exit Notepad.

```
my.prg - Notepad
File  Edit  Format  View  Help
start
some code
some more code
added this code in the clone
```

Figure 9. Notepad can be used a quick way to edit text-based source code files.

My.prg is now a modified file and needs to be committed to the repository. Click the icon on the top toolbar that looks like a green check-mark to change to the Commit pane. Click the

---

[2] On the pop-up menu, *Edit Local* opens the file using the Visual Editor defined in mercurial.ini, wich you can set in the TortoiseHg Settings dialog. *Open Local* opens the file in the default program as defined in Windows, which would typically be Visual FoxPro for .prg files.

Refresh button in the Commit pane to update the display. My.prg now shows up as a modified file (see **Figure 10**).

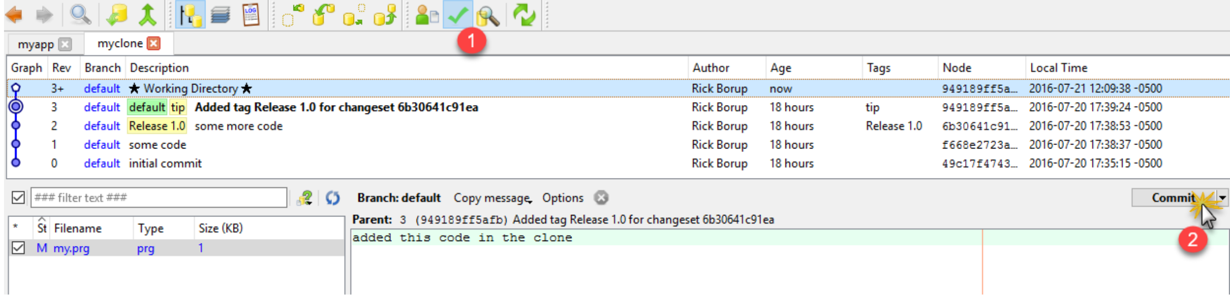


Figure 10. The Commit pane shows modified files pending commit.

Enter "added this code in the clone" as the commit message and click the Commit button to commit this change to the local repository in the clone. Note that this commit is on the *default* branch, as all other commits have also been so far.

The cloned repository now contains a changeset that doesn't yet exist in the original repository (see **Figure 11**). This is what we wanted – we're using the local clone as a branch, and the whole purpose of a branch is to be able to make modifications without affecting the original.

Graph	Rev	Branch	Description
	4+	default	★ Working Directory ★
	4	default	default tip **added this code in the clone**
	3	default	Added tag Release 1.0 for changeset 6b30641c91ea
	2	default	Release 1.0 some more code
	1	default	some code
	0	default	initial commit

Figure 11. The cloned repository now contains a revision that doesn't yet exist in the original.

Now it's time to merge the change we made in the clone into the original repository. This can be accomplished either by pushing the new changeset from the clone to the original, or by opening the original and pulling the changeset from the clone. The decision on which way to do it is governed by the relationship between the clone and its origin. In the case where both are local, as in this exercise, it makes no difference. If the origin is a shared remote repository, you'll push to it. If the origin is a repository on another team member's computer, you probably won't have permissions to push to it so you'll ask the other team member to pull your changes. If the origin is on Bitbucket or GitHub, you'll use the mechanism provided by those services to send a pull request to the maintainer of the origin.

In this exercise, because it's a local clone of a local repository, we'll simply push the change from *myclone* back to the original repository in *myapp*. To do this from TortoiseHg, first click the *Synchronize* button on the task toolbar. It's the one that looks like two green

Copyright 2016, Rick Borup

arrows in a circular arrangement (see **Figure 12**). This opens the Synchronize pane in the lower portion of the TortoiseHg window. When you created the clone, Mercurial automatically added a link back to its origin, so the clone already "knows" where it came from. In Figure 12 you can see this in the *Paths in Repository Settings* pane at the bottom left.

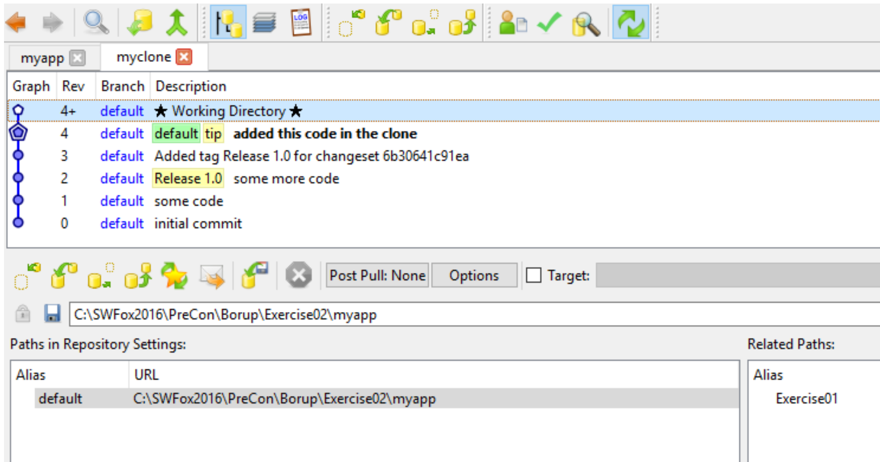


Figure 12. The Synchronize pane provides access to set up and push/pull to remote repositories.

Before pushing, it's a good idea to check what will be pushed. Mercurial provides the *Outgoing* command for this purpose. You can run the Outgoing command from TortoiseHg by clicking the *Detect outgoing changes* button, third from the left on the toolbar above the *Paths is Repository Settings* caption in the Synchronize pane. Note that the same icon and its companions — *Incoming*, *Pull*, and *Push* — are also available on the task toolbar at the top.

When you click the Outgoing icon, TortoiseHg tells Mercurial to check what's ready to be pushed and then displays it for you in the Revision History pane at the top (see **Figure 13**). We see that revision 4, the changeset we created in the clone, is ready to be pushed back to the origin. This is what we expected, so go ahead and push it by clicking the *Push all* button.

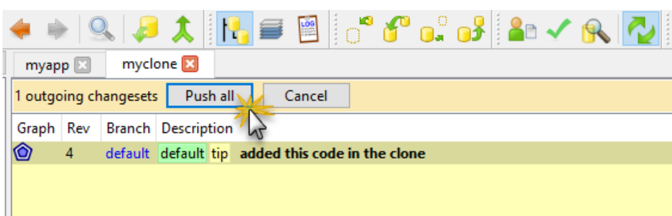


Figure 13. Mercurial's *Outgoing* command shows what is ready to be pushed. Use *Push All* to push.

To confirm that the push worked as expected, open or switch to the *myapp* folder in TortoiseHg and view the revision history. The new changeset is there, but something looks different. Compare the graph of the history of the clone in **Figure 14** to the graph of the history in the original in Figure 11.

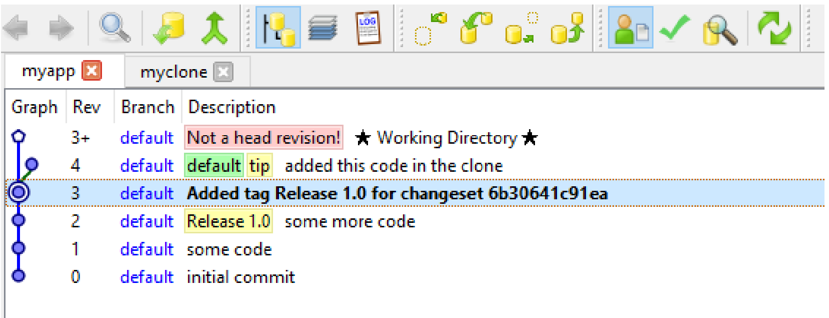


Figure 14. After pushing from the clone, the original's working copy needs to be updated.

There are two differences of note. The first and most obvious is the red warning message about "Not a head revision!" on the top line. The other is that the new changeset looks like some sort of branch in the graph, even though the Branch column shows it as being on the *default* branch just like all the others. What's going on?

The top line of the graph always references the working directory. The number in the Rev column of that line indicates the revision number on which the working copy is based, followed by a plus sign. In this example, the "3+" indicates that the parent of the working directory is revision 3. However, the head revision in this repository is revision 4 (the one that was just pushed). The "Not a head revision!" message simply alerts you that the working copy is not up to date with the head — in other words, the parent of the working directory is not a head revision.

The second issue is what looks like a branch in the graph. Revision 3 is the parent of both the working directory and of revision 4, but the working copy is not based on revision 4 so the graph cannot show it that way. Instead, the graph draws two lines from revision 3: one to the working directory, and the other to revision 4. Everything is still on the *default* branch at this point, so revision 4 is not a different named branch.

This type of branch — a divergence within the same named branch — is called an *anonymous branch*. One way anonymous branches are created is when newer changes are pushed or pulled into an existing branch of a repository, as this exercise demonstrated. They exist until the working copy is updated to the head of the anonymous branch. You can also create anonymous branches deliberately and use them as a way to do concurrent feature development without named branches, as you'll see in a later lesson.

One step remains in order to complete the process after the push: the working copy needs to be updated to the new head revision. To do this, right-click on revision 4 and choose *Update* from the pop-up menu. This triggers a confirmation dialog as shown in **Figure 15**.

Copyright 2016, Rick Borup

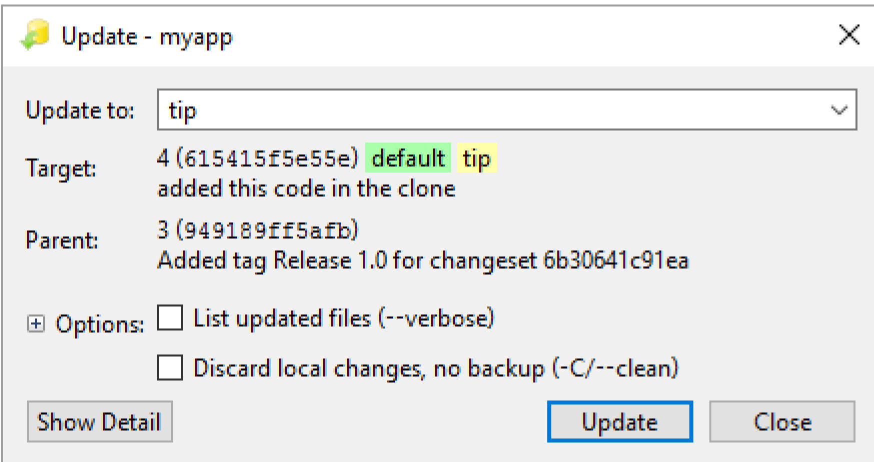


Figure 15. The Update confirmation dialog shows what is about to happen.

Click the *Update* button to complete the update. Mercurial updates my.prg in the working directory by applying the change made in revision 4. The working copy is now up to date with the head, the anonymous branch disappears, the warning message goes away, and the process is complete (see **Figure 16**).

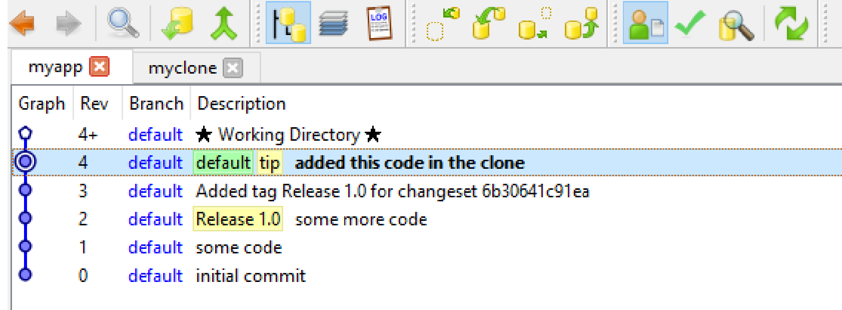


Figure 16. The original is up to date again after updating the working copy to revision 4.

If you have no other use for the clone, you can now delete it.

Lesson 3 – Non-Linear workflow with anonymous branches

Lesson 2 demonstrated how to use a clone as a branch. If there's one drawback to this method, it's the overhead involved in creating a complete clone of a project's working directory and repository if all you want is a simple branch. Mercurial provides a couple of ways to do branching within a repository. One way is to use anonymous branches, and the other is to use named branches. This lesson shows how to create anonymous branches and use them to facilitate branchy development without named branches.

In Mercurial, named branches are permanent — they persist as part of the repository's history even after they're no longer needed or used. Depending of the workflow being used, a repository's history can over time become crowded with branches that are no longer used. Some developers favor this because a particular revision can always be traced back to the branch on which it was created. Other developers, especially those coming from Git, would just as soon get rid of a branch after it's no longer needed. Anonymous branching makes this possible.

The primary advantage of anonymous branches is that they do not involve the use of named branches and can be discarded when no longer needed. One disadvantage is that anonymous branches can only be referred to by their revision number or commit hash, unless you apply a *bookmark*. The fact that anonymous branches can be made to disappear after they're no longer needed is either an advantage or a disadvantage, depending on your point of view.

In general, anonymous branches can provide an effective workflow for the solo developer but are not necessarily ideal for team development.

Bookmarks

The concept of permanent named branches is foreign to developers using Git. Git uses lightweight branches that cease to exist after they're no longer used or needed. Mercurial introduced the concept of *bookmarks* to provide a more familiar experience for developers coming from Git.

A bookmark is label associated with a commit. You can name your bookmarks whatever you like and you can add as many bookmarks as you want, but only one bookmark is active at any given time. Mercurial recognizes a special bookmark named @ to represent the "master" branch. If there is a bookmark named @ in a remote repository, that branch is automatically checked out when the remote is cloned.

A bookmark can be added to any revision, but bookmarks associated with a head revision have a special behavior: unlike tags, a bookmark that points to a head revision is automatically moved to the new head with every commit. This means that if a bookmark is applied to the latest revision in an anonymous branch, it will always point to the head revision of that branch as you make commits to it. Also unlike tags, bookmarks are local and don't get pushed to and pulled from remote repositories without special handling, which can be a disadvantage for team collaboration.

Although useful for other reasons as well, bookmarks provide a way to implement branchy development without using named branches.

Exercise 3

In this exercise you use anonymous branches to work on two new features simultaneously. When completed, the branches are merged back into the main line of development for release.

The revision history for the starting point of this exercise is shown in **Figure 17**. Version 1.0 of this product has been tagged and released, and the working copy is clean (no uncommitted changes).

Graph	Rev	Branch	Description
	3+	default	★ Working Directory ★
	3	default	default tip **Added tag Release 1.0 for changeset 6b30641c91ea**
	2	default	Release 1.0 some more code
	1	default	some code
	0	default	initial commit

Figure 17. This is the starting point for exercise 3.

To begin work on Feature 1, make a change to my.prg and commit it. At this point, the log graph doesn't show any signs of branching. The new commit is simply the new tip in default (**Figure 18**).

Graph	Rev	Branch	Description
	4+	default	★ Working Directory ★
	4	default	default tip **begin work on feature 1**
	3	default	Added tag Release 1.0 for changeset 6b30641c91ea
	2	default	Release 1.0 some more code
	1	default	some code
	0	default	initial commit

Figure 18. The revision history graph doesn't show any branching yet.

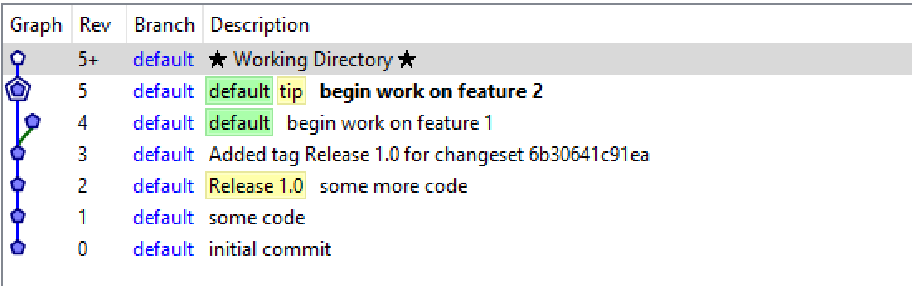
We're now ready to begin concurrent development on Feature 2. During development, we want the modifications for Feature 2 to be isolated from the modifications for Feature 1. To accomplish this, we'll use two separate branches, one for each feature.

Release 1.0 (revision 3 in the log graph) is the common parent, or base, revision for both Feature 1 and Feature 2. To begin work on Feature 2, we first need to update the working copy to Release 1.0. To do this, click on revision 3 and select *Update* from the pop-up menu. The source code file my.prg is now back to its state as of Release 1.0, and the commit we made to begin work on Feature 1 (revision 4) now looks like an anonymous branch (see **Figure 19**). It's anonymous because it's not a named branch — look at the Branch column and note that all the commits are on the *default* branch. We can also see the familiar "Not a head revision!" message on the top line, which in this case can be safely ignored.

Graph	Rev	Branch	Description
	3+	default	Not a head revision! ★ Working Directory ★
	4	default	default tip begin work on feature 1
	3	default	**Added tag Release 1.0 for changeset 6b30641c91ea**
	2	default	Release 1.0 some more code
	1	default	some code
	0	default	initial commit

Figure 19. Revision 4 looks like a branch after updating the working copy back to revision 3.

To begin work on Feature 2, make a change to my.prg and commit it. In the TortoiseHg log graph, heads are visually identifiable by the green highlight on the branch name in the Description column. **Figure 20** shows there are now two heads in the repository.



Graph	Rev	Branch	Description
	5+	default	★ Working Directory ★
	5	default	default tip **begin work on feature 2**
	4	default	default begin work on feature 1
	3	default	Added tag Release 1.0 for changeset 6b30641c91ea
	2	default	Release 1.0 some more code
	1	default	some code
	0	default	initial commit

Figure 20. There are two heads in the repository after beginning concurrent work on feature 2.

Anonymous branches don't have names, so at this point there's no good way to know which head is Feature 1 and which is Feature 2 other than by their commit messages, which were artificially constructed to make it clear which one is which. This is where bookmarks become useful.

You can name your bookmarks whatever you want, but it's a good idea to make them short and meaningful. For this example, *Feature_1* is the name of the bookmark for feature 1 and *Feature_2* is the name of the bookmark for feature 2. The underscores are not required – they're used only because *Feature_1* is easier to read than *Feature1* all run together.

To add a bookmark to any revision, right-click on it and select Bookmark. Enter a name for the bookmark and click Add. **Figure 21** shows the Bookmark dialog ready to add a bookmark named Feature_1 to revision 4. At this point, revision 4 is the head revision of the feature 1 branch.

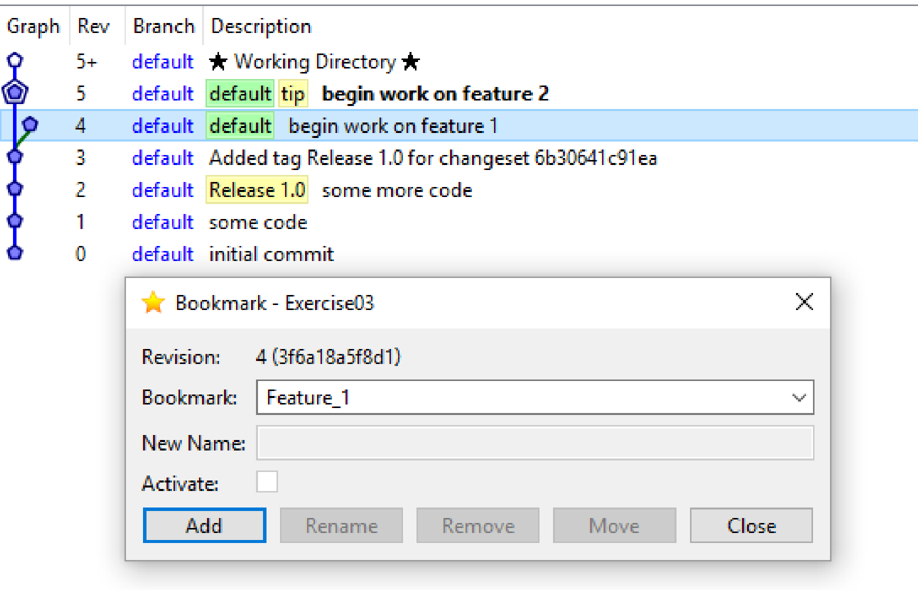


Figure 21. A bookmark is a label applied to a revision.

> ***Side note****: While you can name your tags, bookmarks, and branches anything you like, it's a best practice to adopt some kind of standard naming scheme to avoid creating duplicates. If there is a tag and a bookmark with the same name in the same repository, Mercurial may think you're referring to one when your intent was to refer to the other.*

A revision can have more than one bookmark, so you need to close the Bookmark dialog before you can add a bookmark for Feature 2. After closing the Bookmark dialog, right-click revision 5 and add a bookmark named *Feature_2*. Both of the anonymous branches are now easily identifiable by their bookmarks, as shown in **Figure 22**.

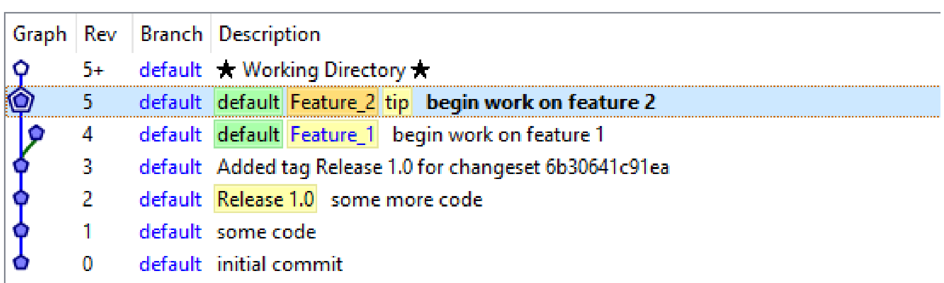


Figure 22. Bookmarks make is easy to identify the head revisions for feature_1 and feature_2.

You can now work on Feature 1 and Feature 2 independently of one another. To switch from one branch to the other, simply update the working copy to the head of the branch you want to work on, then make your modifications and commit them. Only one bookmark can be active at any given time. When you make a new commit, Mercurial automatically moves the active bookmark to the new head.

Figure 23 shows the log graph after working on Feature 1 and Feature 2 while switching back and forth between them. The most recent commit, revision 11, was made on the *Feature_2* branch so it's the tip revision and also the active bookmark. The most recent commit for *Feature_1* branch is revision 9, which is the head of that branch and easily identified by its bookmark, which is inactive at this point.

Graph	Rev	Branch	Description
	11+	default	★ Working Directory ★
	11	default	default Feature_2 tip **finish feature 2**
	10	default	more work on feature 2
	9	default	default Feature_1 finish feature 1
	8	default	work on feature 2
	7	default	more work on feature 1
	6	default	work on feature 1
	5	default	begin work on feature 2
	4	default	begin work on feature 1
	3	default	Added tag Release 1.0 for changeset 6b30641c91ea
	2	default	Release 1.0 some more code
	1	default	some code
	0	default	initial commit

Figure 23. The bookmarks move with each commit and continue to identify the head revisions.

When working concurrently on two new features, you may not know in advance which one will be ready to release first. The beauty of using branching is not only that the two features can be developed concurrently and independently, but also that whichever one is ready first can be released without affecting the other. In other words, if Feature 2 is ready to go before Feature 1, we could update to Feature 2 and build a release from there while leaving the ongoing work on Feature 1 intact.

For purposes of this exercise, we want the next release of the product to include both Feature 1 and Feature 2. As of Figure 23, the commit message indicates both features are finished and ready for release. We arbitrarily decide to update to Feature 1 first and then merge in Feature 2, but we could do it in reverse order and get the same final result.

To begin, update the working directory to the head revision of the Feature_1 branch, which yields the result shown in **Figure 24**.

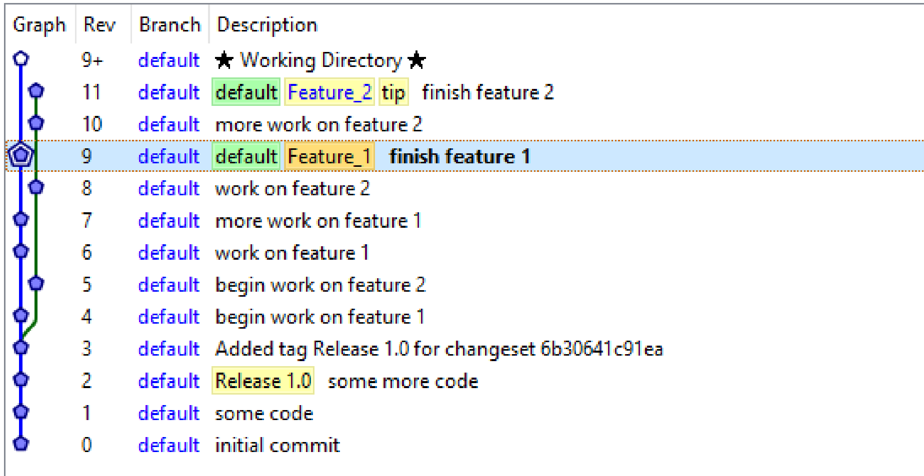


Figure 24. The working copy has been updated to the head of feature_1 in preparation for merging.

My.prg now contains all the modifications for feature 1 but none of the modifications for feature 2. We want both of these new features to be in included in the next release of this product, so we need to merge *Feature_2* into *Feature_1*. To do this, right-click on the head revision for *Feature_2* (revision 11) and select *Merge with local…* from the pop-up menu. This brings up the *Prepare to merge* dialog as shown in **Figure 25**.

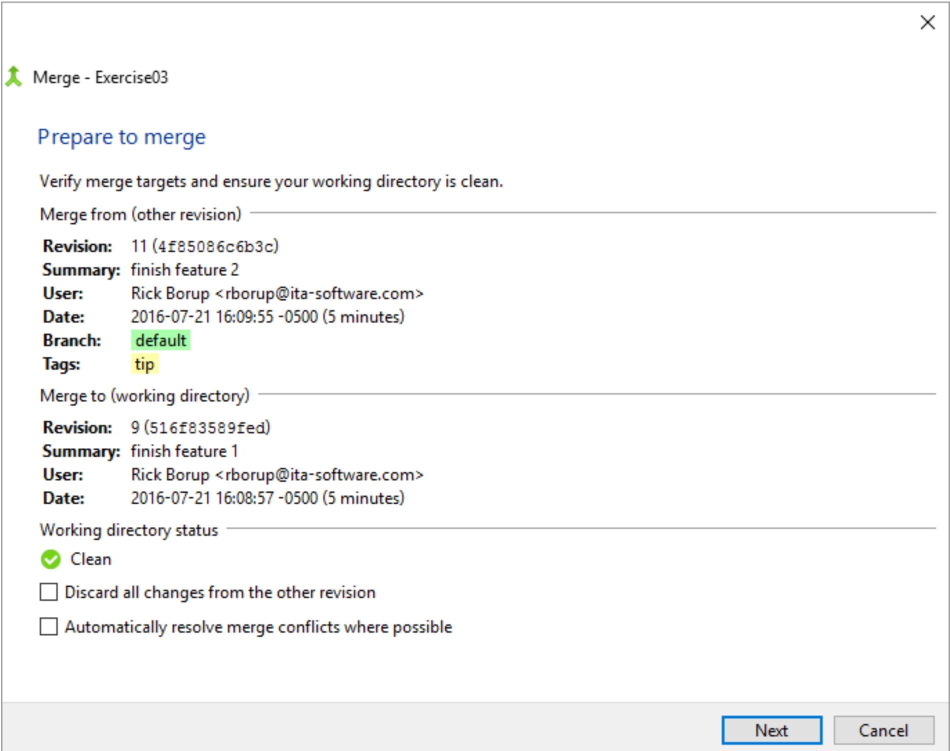


Figure 25. The first dialog in the merge process shows what's about to happen.

This dialog gives you the opportunity to confirm that the merge you are about to perform is what you want. Note the "Automatically resolve merge conflicts" check box at the bottom.

We want to handle merge conflicts manually, so remove the check mark if it's present. Click *Next* to continue.

A merge conflict occurs when there are conflicting changes to the same file. In this exercise, the modifications for Feature 1 occupy the same lines in my.prg as the modifications for Feature 2. This creates a merge conflict, which is detected by Mercurial's merge mechanism and noted in the message at the bottom of the next dialog.

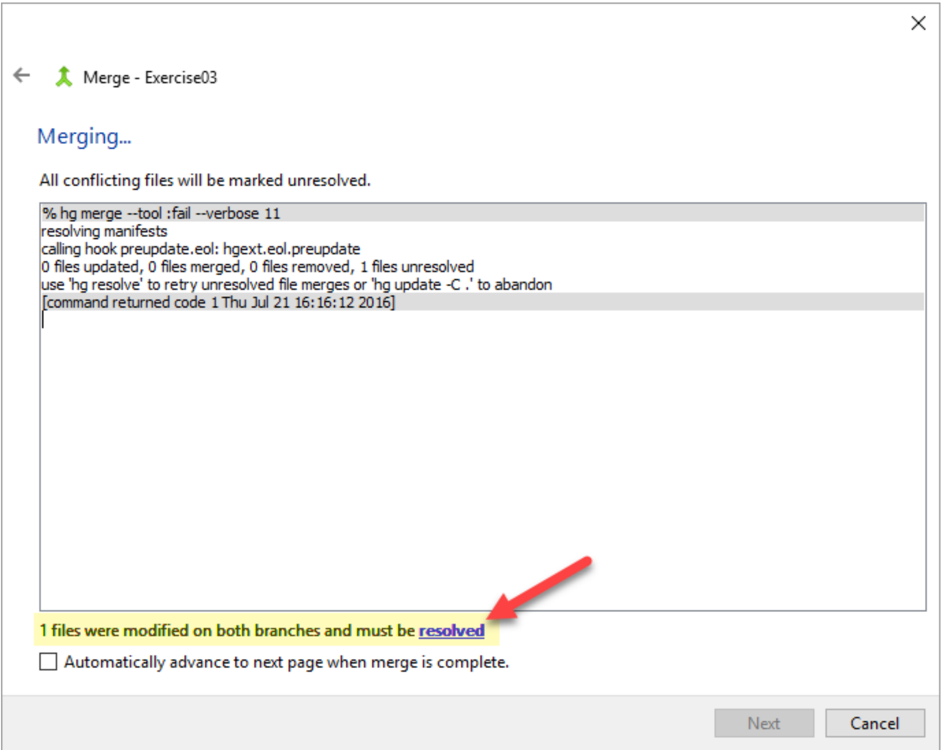


Figure 26. TortoiseHg alerts if merge conflicts are detected and provides a link to resolve them.

Note that the word "resolved" in the highlighted message in **Figure 26** is a hyperlink. Click it to launch the resolve process.

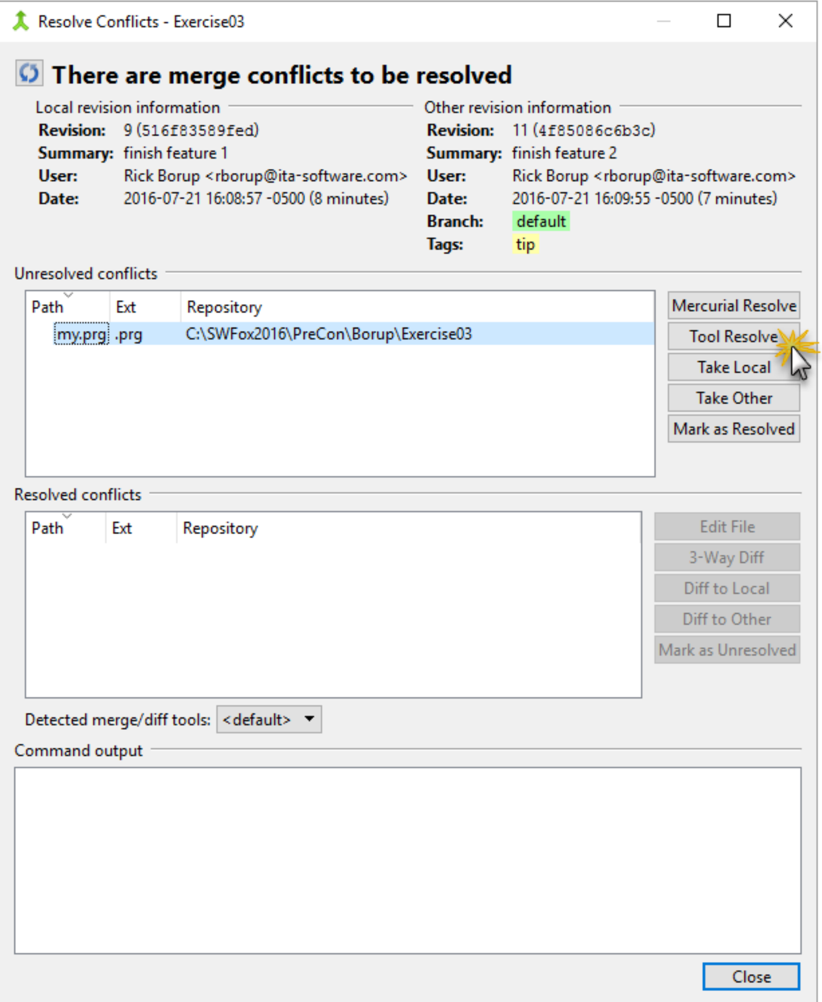


Figure 27. The Resolve Conflicts dialog lets you choose how to resolve the conflicts.

Merge conflicts can be resolved in several different ways. Note the labels on the upper five command buttons in **Figure 27**. In this exercise we want to use the 3-way diff tool to resolve the merge manually, enabling us to decide which lines to keep from feature 1 and which from feature 2, and it what order.

The default 3-way merge tool in TortoiseHg is KDiff3. Click the "Tool Resolve" button to launch that tool. This brings up the KDiff3 user interface, illustrated in **Figure 28**.

Like other 3-way merge tools, KDiff3 has two main parts. The upper panel, which has three sections side-by-side, represents the three components that are part of the merge. These are labelled A, B, and C. A is the base (unmodified) code. B is the first parent, which is feature 1 in this example, and C is the second parent, which is feature 2. The lower panel is where the output will appear after resolving the conflict. The line that says <Merge Conflict> indicates where the merge conflict occurs. In this exercise it's the last line, but that's not always the case.

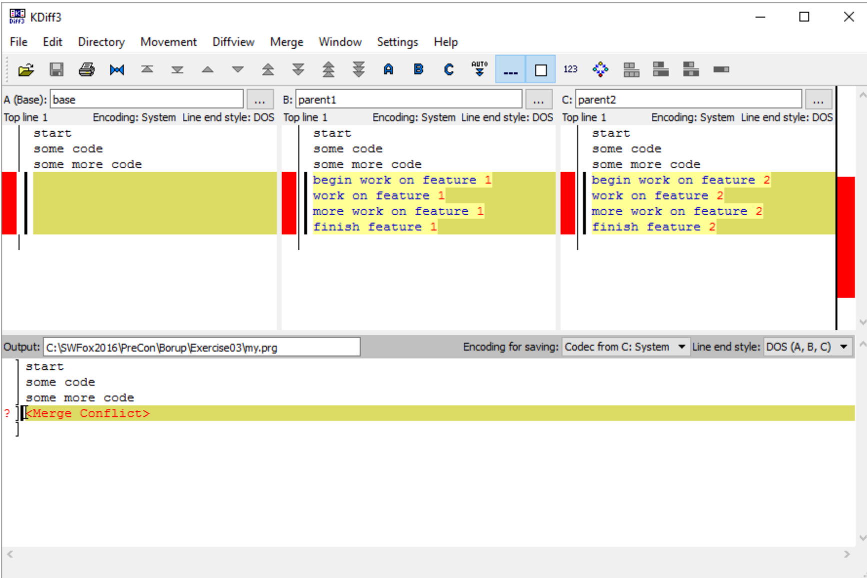


Figure 28. KDiff3 is the default merge conflict resolution tool in TortoiseHg.

In this exercise we want to keep all the lines from Feature 1 and all the lines from Feature 2. We'll arbitrarily decide to merge in the code from feature 1 first, then append the code for feature 2.

Click the *B* icon on the KDiff3 toolbar to add the highlighted lines from feature 1 to the merge conflict location in the output panel. Then click the *C* icon to append the highlighted lines from feature 2. The resulting output, shown in the lower pane, now contains the code for both features (see **Figure 29**).

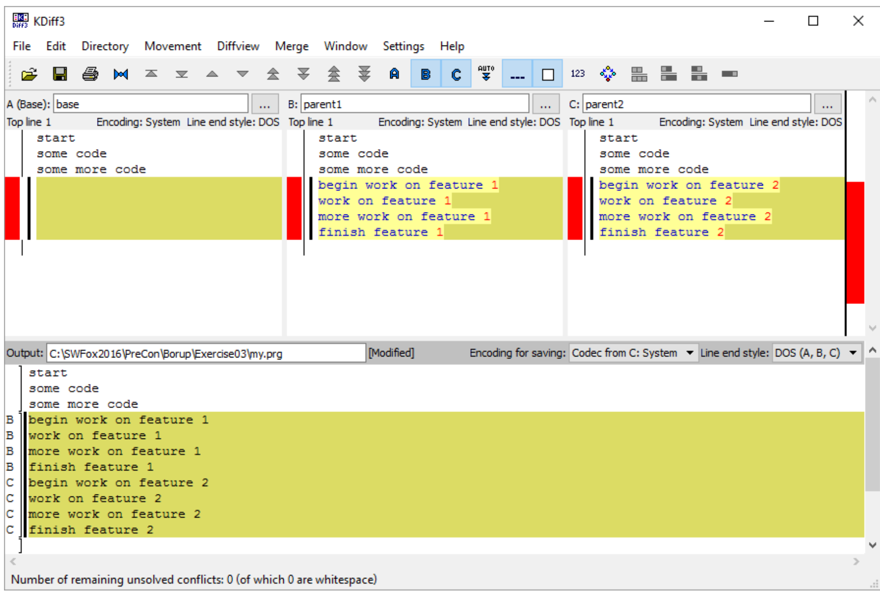


Figure 29. Use the A, B, and C toolbar icons to select the code to be resolved into the merge.

Click the *Save* button and close the KDiff3 window. The TortoiseHg Merge dialog now shows there are no more unresolved files (see **Figure 30**).

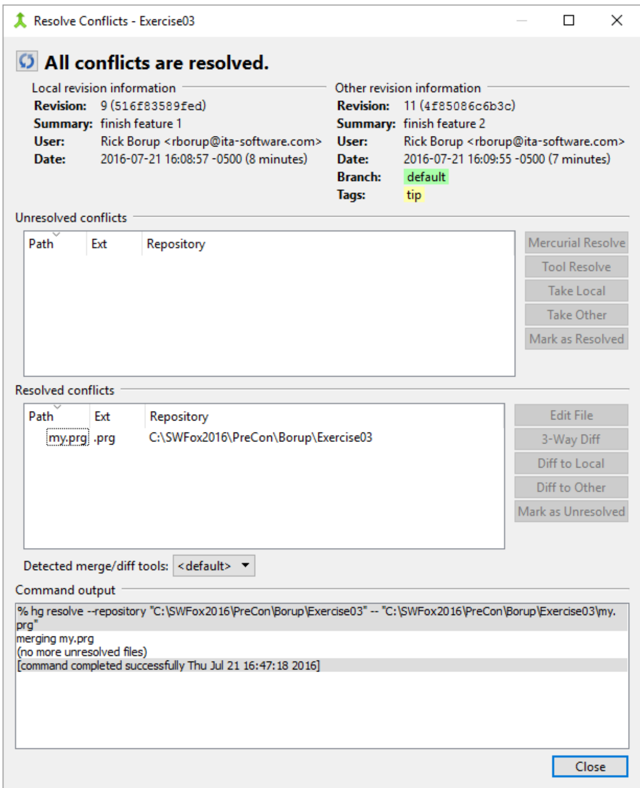


Figure 30. The Resolve Conflicts dialog now shows all conflicts have been resolved.

Click the *Close* button to finish the resolve process, then click *Next* in the Merge dialog. This brings up the *Commit merge results* window shown in **Figure 31**. The default commit message is simply the word Merge. If you prefer a more descriptive commit message, enter it in the *Commit message* area as illustrated in Figure 31.

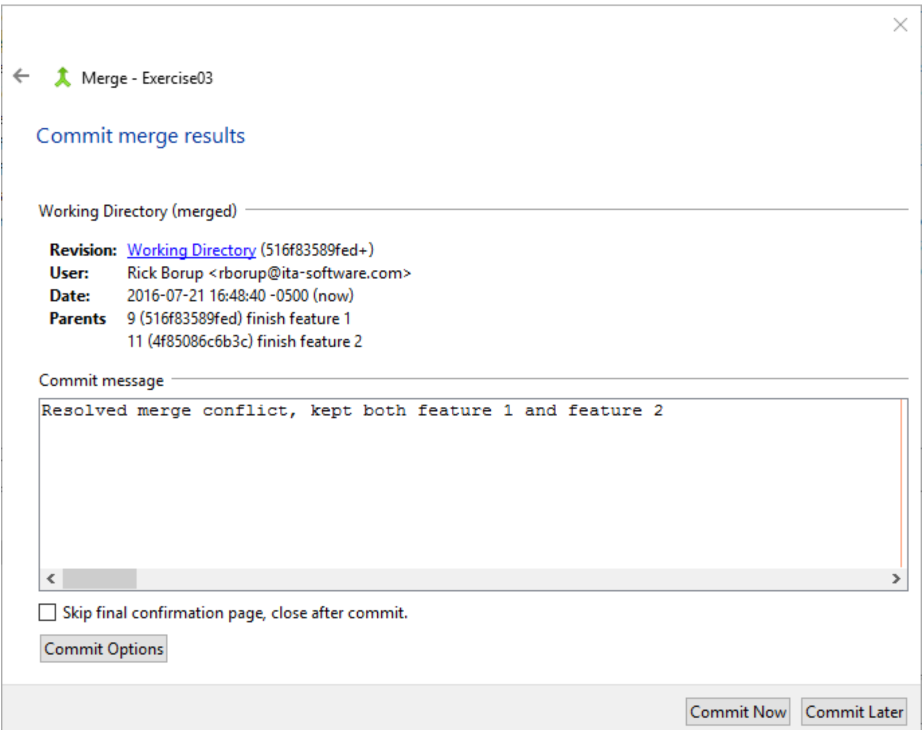


Figure 31. You can override the default merge commit message if you want to.

If you want to, you can mark the check box to skip the final confirmation page shown in **Figure 32**. Click the *Commit Now* button to commit the merged output to the repository, then click *Finish*.

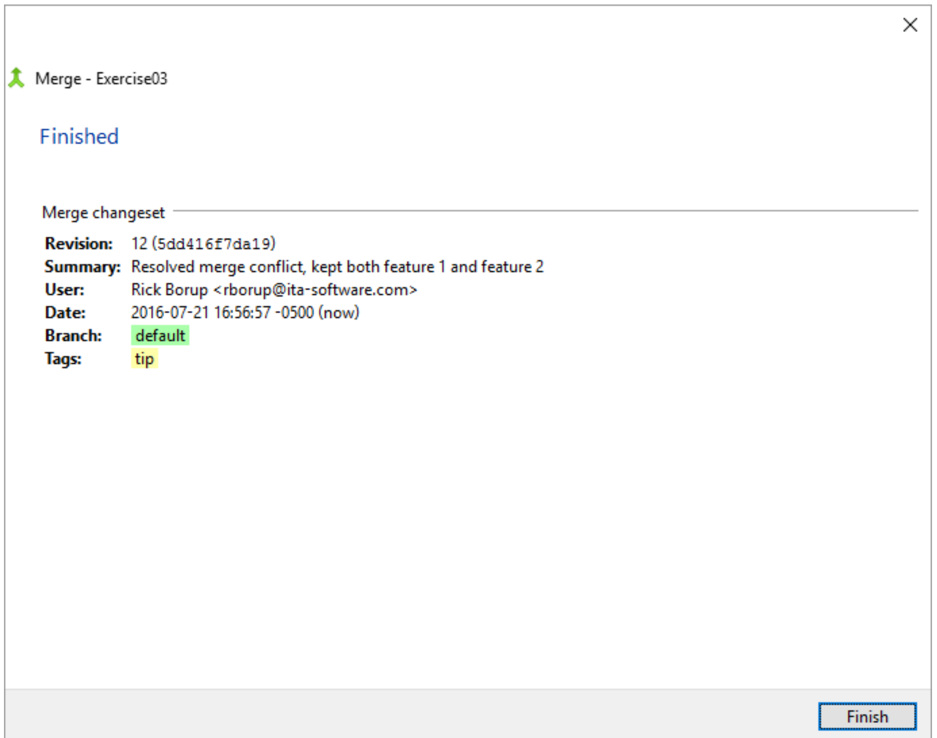


Figure 32. The optional Finished dialog can be omitted if desired.

At this point, the log graph now looks as shown in **Figure 33**. The source code file my.prg now contains both Feature 1 and Feature 2, so we can proceed to build and release.

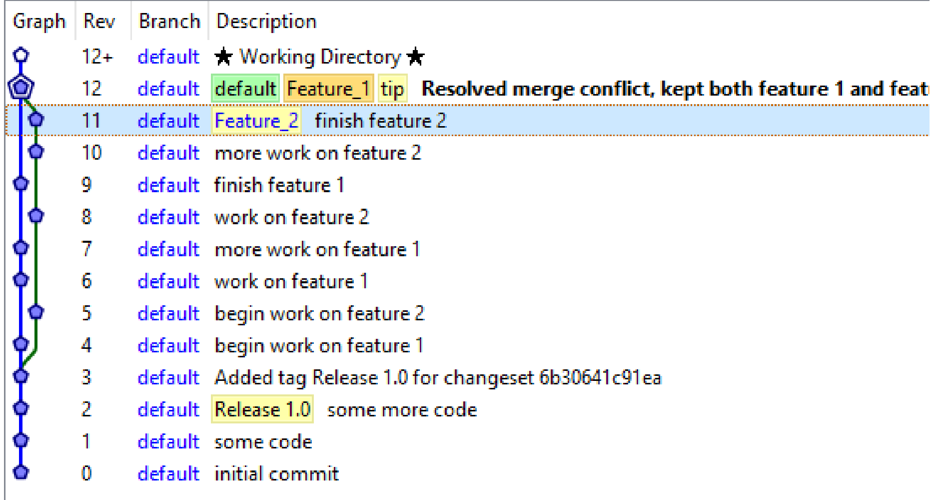


Figure 33. After the merge commit, the working directory is clean and the revision history is complete.

We're also done with the Feature_1 and Feature_2 branches, so we can remove the bookmarks. It would not be a good idea to leave the bookmarks in place, because the active bookmark (Feature_1) is on a head revision and would continue to be moved to the new head every time a new commit is made, giving a false impression that we're still working on feature 1.

To remove the Feature_2 bookmark, right-click on revision 11, select Bookmark and click Remove. Repeat the process on revision 12 to remove the Feature_1 bookmark. Finally, add a tag for Release 1.1. The final result is shown in **Figure 34**.

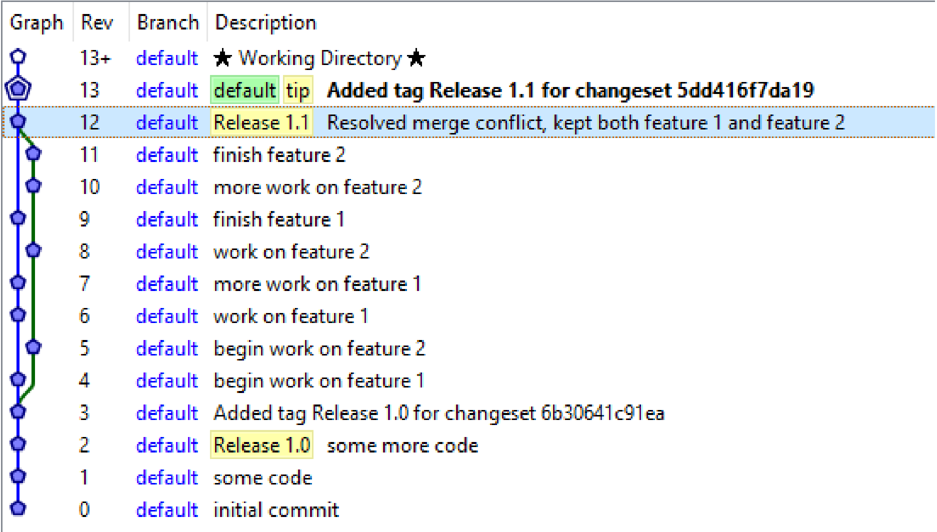


Figure 34. Tags are another type of label. They are frequently used to identify release versions.

Note that while the history shows all the commits along the way during the development of feature 1 and feature 2, it doesn't really look like there were two different branches involved. If the commit message had not been artificially constructed to identify the branch they pertain to, there would be no quick way to determine from the graph which commits belonged to which branch. This is one of the drawbacks of using anonymous branches.

Rebasing

There are two ways of getting changes from a branch back into the line from which the branch diverged. One way is to merge them in, as shown in this exercise. The other is to rebase.

Rebasing takes a series of commits from one branch and reapplies them on top of the revision that is their common ancestor in the original branch. This modifies the revision history so the sequence of commits appears as if it occurred in a straight line instead of on a branch.

Rebasing is common in Git, due in part to Git's lightweight branches. You can read about rebasing in Git at https://git-scm.com/book/en/v2/Git-Branching-Rebasing.

Rebasing is less common but still feasible in Mercurial. There's a great tutorial on how to rebase using TortoiseHg at http://thedustytome.com/2014/05/29/Mercurial-Rebase-with-TortoiseHg/.

Lesson 4 – Non-linear workflow with a development branch

Lesson 3 introduced anonymous branches and showed how to use them for non-linear (branchy) development within a repository. Lesson 4 leaves anonymous branches behind and introduces a non-linear development workflow using named branches, beginning with a single named branch for all development work.

In any workflow it's customary to designate at least one branch as the release branch. In Mercurial, the *default* branch is most often used as the release branch. As its designation implies, nothing gets committed to a release branch until the code is ready for release into production.

In the workflow in this exercise, all development work takes place on a branch other than the release branch. This branch is referred to as the development branch. The development branch (or branches, in more complex examples) can be named anything you like. This exercise uses a single development branch named *develop*.

The advantage of separating development work from the release branch is that nothing gets committed to the release branch until it's tested and ready for release into production. The code on the release branch is always stable, and any release can be built (or rebuilt) by updating the working copy to the desired revision on the release branch.

The disadvantage of using a single development branch is that it does not facilitate the simultaneous development of two or more new features. This can be particularly important when it's uncertain which of two or more new features will be ready to release first and you don't intend to wait until both are finished.

Exercise 4

Exercise 4 demonstrates a non-linear workflow using the *default* branch as the release branch and a single named branch called *develop* for development work. When completed, the code in the development branch is merged back into the release branch.

In this exercise you will

1. create a branch named *develop*
2. make changes to my.prg
3. commit to *develop*
4. repeat 2 and 3 until ready for release
5. checkout *default*
6. merge from *develop*
7. commit the merge to *default*
8. add a tag for the new release

The starting point for this exercise is shown in **Figure 35**. Release 1.0 of this product has been tagged and released into production, and the working copy is clean (i.e., there are no uncommitted changes).

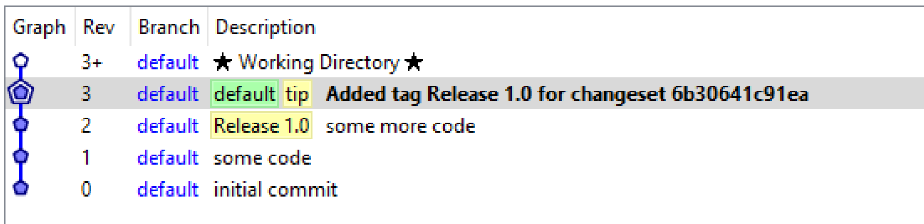


Figure 35. The starting point for exercise 4.

In this workflow, the *default* branch always represents code that is ready for release as opposed to code that is in testing or development. Each head in the release branch represents the state of the code as of a given release, along with its tag (optional).

To begin development work on a new feature, we start with the code as of the most recent release and begin to make modifications. Remember that modifying code in the working directory does not affect the repository until you do a commit.

In Mercurial, a new branch is not created until the first changeset is committed to it, so unlike Git, you don't create (or change to) the desired branch until you're ready to make a commit to it. We don't need to tell Mercurial we're working on what will become the first commit to the *develop* branch until we're ready to commit the first set of modifications to it.

In TortoiseHg, the branch to which the changeset will be committed is indicated where it says "Branch: <branch name>" at the top of the commit pane. Because we started with code from the *default* branch, this currently says "Branch: default" as shown in **Figure 36.** To create the *develop* branch, click on the word *default* to open the *branch operation* dialog, as illustrated in Figure 36. Select the "Open a new named branch" option, enter the word *develop*. This dialog can be resized to show longer branch names. Click OK when ready.

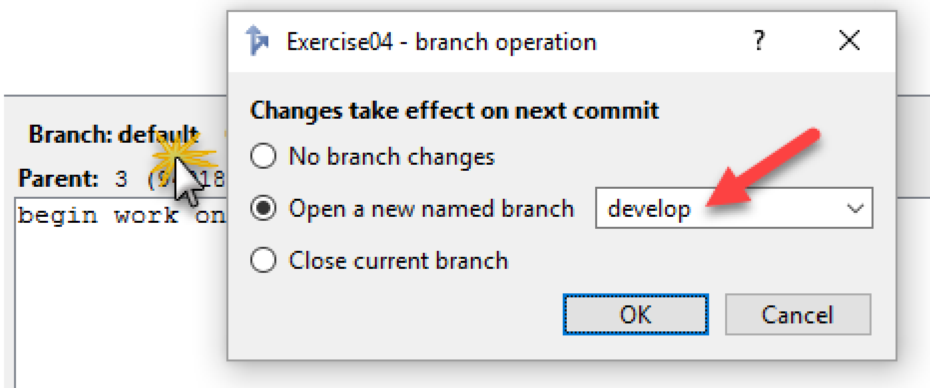


Figure 36. Use the *branch operation* dialog to open a new branch.

The branch indicator at the top of the commit dialog changes to show the new branch, as illustrated in **Figure 37**.

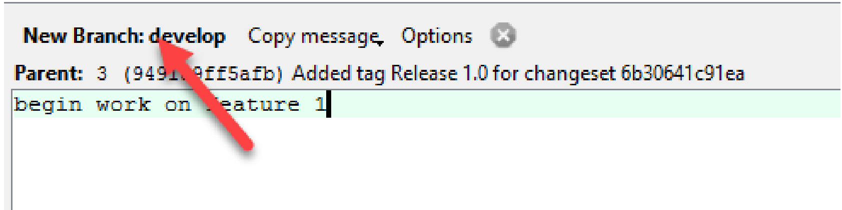


Figure 37. The new branch is indicated at the top of the *Commit* pane.

Click the *Commit* button to commit these modifications to the new *develop* branch. Mercurial asks you to confirm you want to create a new branch. In Mercurial, branches are permanent, so it's good to have a chance to cancel out if you didn't really intend to create a new branch. In this case we do want a new branch, so click the *Create Branch* button (see **Figure 38**).

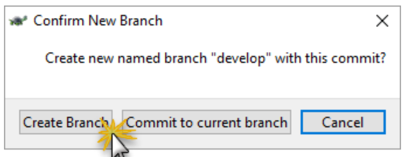


Figure 38. TortoiseHg gives you a chance to confirm you want a new branch.

Now look at the TortoiseHg revision history pane in **Figure 39**. It shows the new commit as revision 4, and the *Branch* column reveals that commit is on the new *develop* branch. Note that there are now two heads in the repository, one for each branch. The branch name for each head is highlighted in green in the *Description* column. You can see that revision 3 is the head of the default branch while revision 4 is the head of the develop branch.

Graph	Rev	Branch	Description
	4+	develop	★ Working Directory ★
	4	develop	develop tip **begin work on feature 1**
	3	default	default Added tag Release 1.0 for changeset 6b30641c91ea
	2	default	Release 1.0 some more code
	1	default	some code
	0	default	initial commit

Figure 39. The *Branch* column identifies the branch on which each commit occurred.

We can now continue to work on the new feature, making as many commits along the way as we feel are necessary. Until we tell Mercurial otherwise, all commits will go on the *develop* branch. **Figure 40** shows the revision history after a couple of work-in-progress commits, followed by a final commit with a commit message indicating we're finished with the new feature.

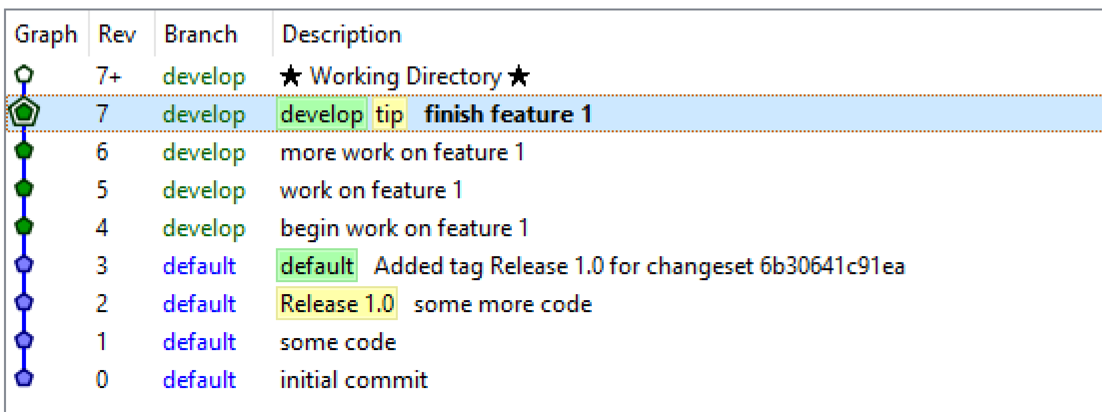


Figure 40. Four commits on the *develop* branch were made during the development of feature 1.

You may have noticed that this still looks like straight-line development even though there are two branches in the repository. This is because, so far, each commit is a child of the previous one even though we added a new named branch at revision 4. The graph will look more like branchy development in a minute, when we merge the development branch back into the *default* branch to build and release the new feature.

The first step in any merge is to update the working copy to the head of the target for the merge. The target for this merge is the head of the *default* branch, representing the code for Release 1.0, so we first need to update the working copy to revision 3. Do this by right-clicking on the line for that revision and selecting *Update* from the pop-up menu.

The Update dialog, shown in **Figure 41**, gives you a chance to confirm what's about to happen. In this exercise you do not need to make any changes to the defaults in this dialog. Do note the *Discard local changes* check box near the bottom, though. Before performing a merge, always be sure the working copy is clean, meaning there are no uncommitted changes. If there are uncommitted changes, marking the *Discard local changes* check box tells Mercurial to discard those changes. In this case the working copy is already clean, so the check box does not need to be marked. Click the *Update* button to proceed.

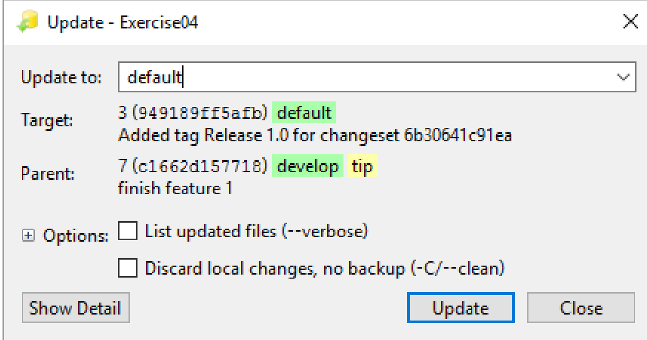


Figure 41. The first step in the merge is to update the working copy to the target of the merge.

After the working copy is updated to the head of the *default* branch, the graph more clearly shows that there are two branches (see **Figure 42**).

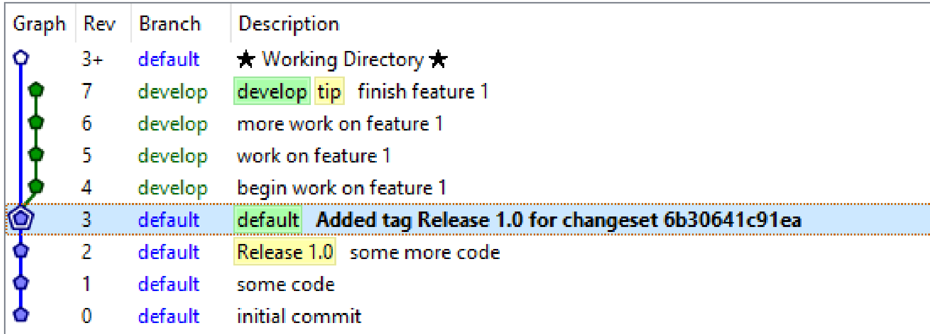


Figure 42. The revision history graph now looks more like "branchy" development.

The second step is to merge the source revision into the target. In this exercise, the source revision is the head of the development branch, which is revision 7. To complete the process of merging the development branch back into the default branch, right-click on the line for revision 7 and choose *Merge with Local* from the pop-up menu. This brings up the first of three merge dialogs (see **Figure 43**).

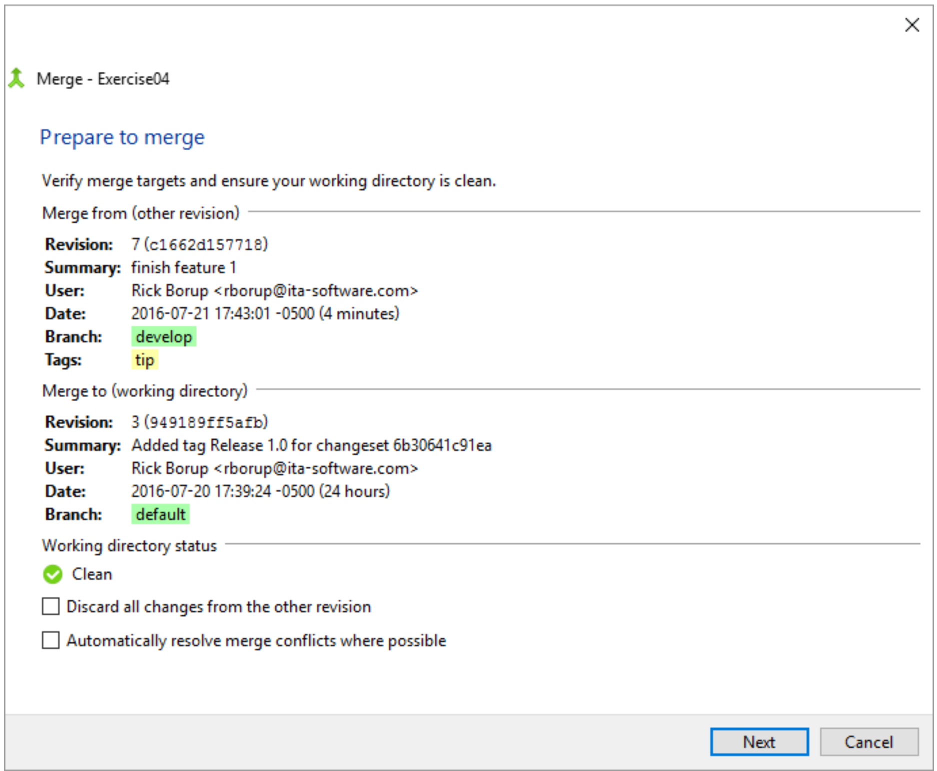


Figure 43. The *Prepare to merge* dialog is the first step.

The sentence at the top of this dialog instructs you to verify the "from" and "to" revisions are what you want, and reminds you to be sure the working directory is clean. Click *Next* when ready to proceed.

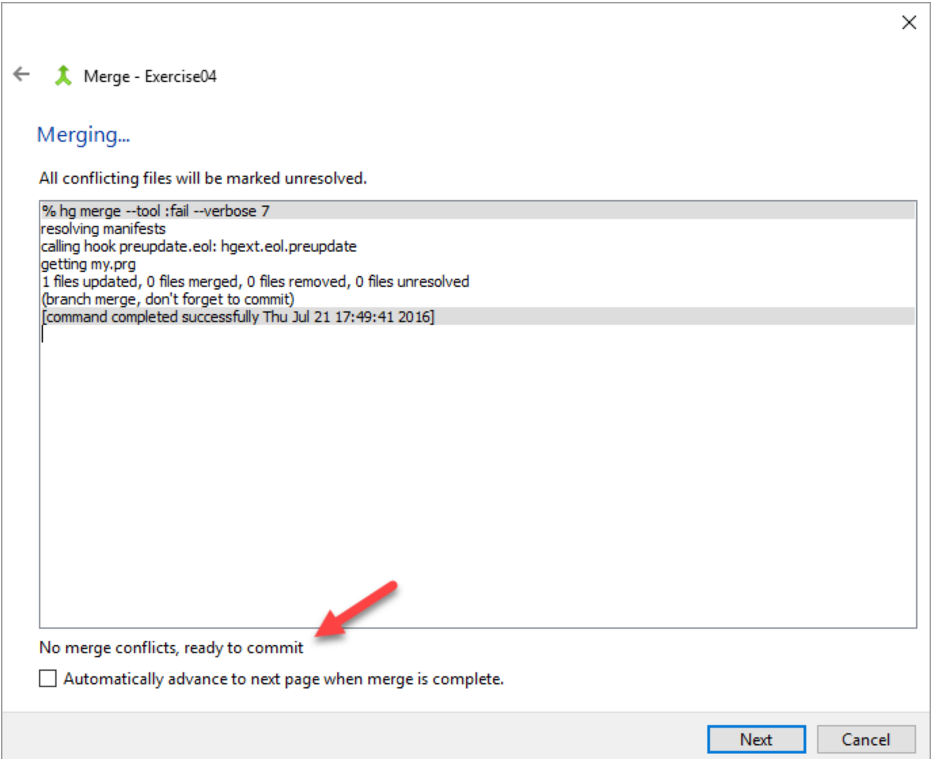


Figure 44. The second step in the merge process lets you know if there are any merge conflicts.

The message at the bottom of the next step (**Figure 44**) tells us that, unlike the earlier exercise, there is no merge conflict here, so the merge is ready to be committed. Click *Next* to proceed.

The final step is to commit the merge to the repository. Like any other commit, merge commits have a commit message. Mercurial has supplied the default message *Merge with develop*, but you can change it. If you prefer a more descriptive like *Merged Feature 1 from develop into default*, edit the commit message accordingly (see **Figure 45**). When ready, click the *Commit Now* button to finish the commit.

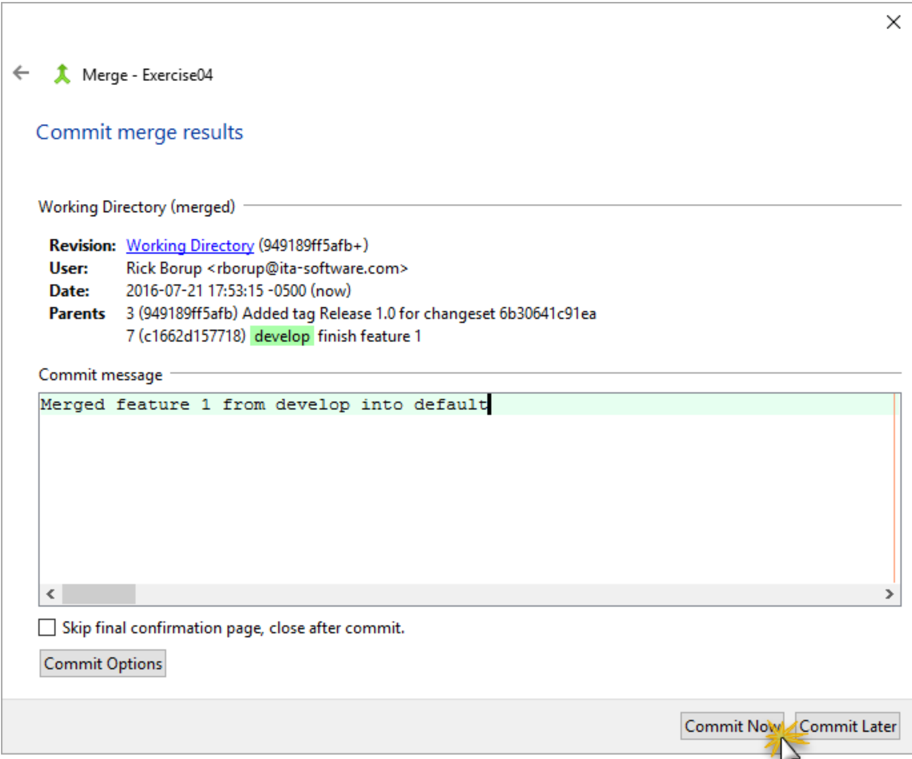
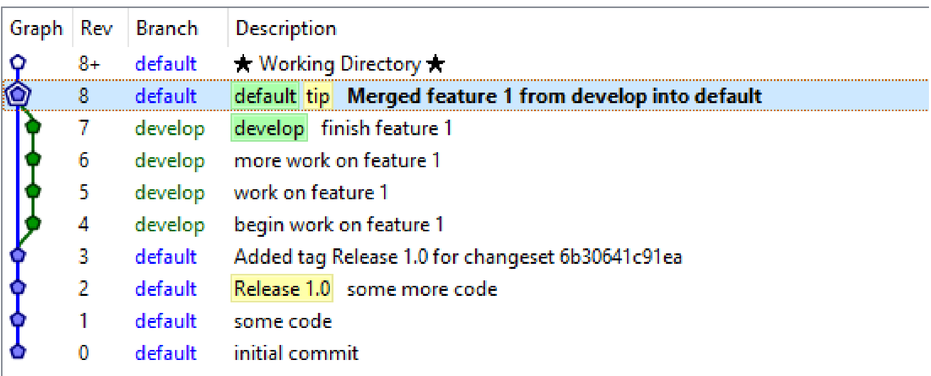


Figure 45. The last step in the merge process is to commit the merge results to the repository.

There is a final confirmation dialog (not shown) which you can skip by marking the check box near the bottom in Figure 45.

The revision history now clearly shows the branchy nature of the work we just did. The development branch was created from revision 4 of the default branch, revisions 5 through 7 were committed to the development branch as we worked on the new feature, and finally revision 8 show where the development branch was merged into the default branch (see **Figure 46**).



Graph	Rev	Branch	Description
	8+	default	★ Working Directory ★
	8	default	default tip **Merged feature 1 from develop into default**
	7	develop	develop finish feature 1
	6	develop	more work on feature 1
	5	develop	work on feature 1
	4	develop	begin work on feature 1
	3	default	Added tag Release 1.0 for changeset 6b30641c91ea
	2	default	Release 1.0 some more code
	1	default	some code
	0	default	initial commit

Figure 46. The revision history graph now clearly shows the “branchy” development history.

In this and the other workflows in this paper, tags are used to identify release versions, so the final step in the process is to add a tag for the new release. Look at the Branch column

Copyright 2016, Rick Borup

in the revision history graph and observe that we're now back in the *default* branch. Right-click on the tip revision (revision 8), choose *Tag* from the pop-up menu, enter *Release 1.1* as the tag name, and click *Add* (see **Figure 47**). A revision can have more than one tag, so the Tag dialog remains open in case you want to add another one. Click the *Close* button to dismiss the Tag dialog.

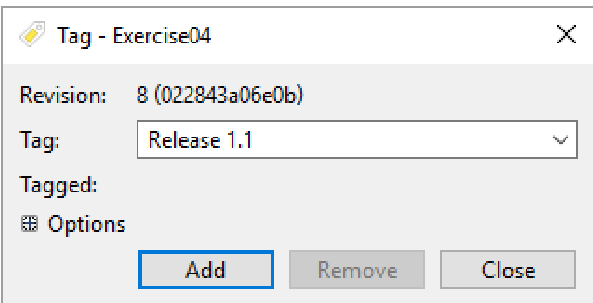
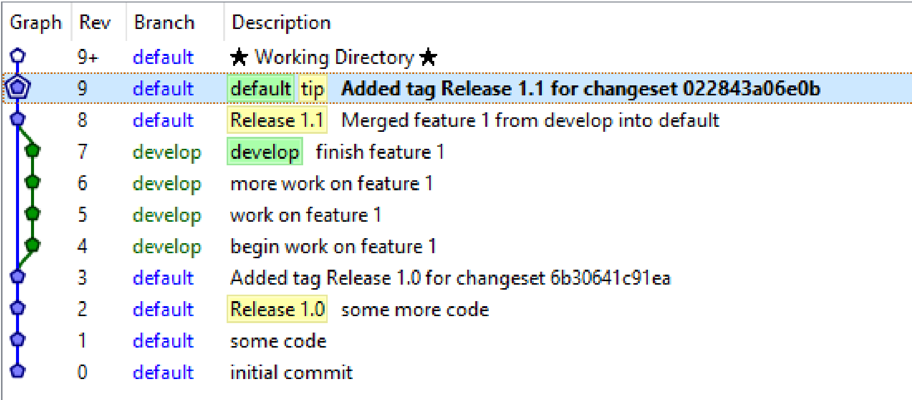


Figure 47. Use the Tag dialog to add a tag to each release version.

The final result can be seen in **Figure 48**. It's clear from both the nature of the graph and from the names in the Branch column that feature 1 was developed on the *develop* branch and merged into the *default* branch for release.



Graph	Rev	Branch	Description
	9+	default	★ Working Directory ★
	9	default	default tip **Added tag Release 1.1 for changeset 022843a06e0b**
	8	default	Release 1.1 Merged feature 1 from develop into default
	7	develop	develop finish feature 1
	6	develop	more work on feature 1
	5	develop	work on feature 1
	4	develop	begin work on feature 1
	3	default	Added tag Release 1.0 for changeset 6b30641c91ea
	2	default	Release 1.0 some more code
	1	default	some code
	0	default	initial commit

Figure 48. In Mercurial, adding a tag creates a new commit.

Lesson 5 – Non-linear workflow with a development branch and a single feature branch

Lesson 4 showed how to use a development branch to isolate development work from the release branch. The primary disadvantage of doing all development work on a single named branch is that it does not facilitate the concurrent development of two or more new features. The next step up from using a single development branch is to use feature branches, one for each feature currently under development.

The rules for this workflow are:

1. The *default* branch is the release branch.

2. The *develop* branch is the root of each feature branch.

3. Each feature gets its own named branch off of *develop*.

4. Feature branches are merged back into *develop*, never into *default.*

5. Only the *develop* branch can be merged into the *default* branch.

These rules are enforced by business practice, although it may be possible to write hooks that enforce them in Mercurial itself.

This exercise reiterates the steps taken in Exercise 4, but demonstrates using a feature branch on top of the development branch. This is a first step towards using multiple feature branches simultaneously, which is the subject of Lesson 6.

Exercise 5

In this exercise you will

1. start from *develop*
2. create branch *feature_1*
3. make changes
4. commit to *feature_1*
5. repeat 3 and 4 until ready for release
6. checkout *develop*
7. merge from *feature_1*
8. checkout *default*
9. merge from *develop*
10. commit the merge to *default*
11. add a tag for the new release

The starting point for this exercise is the same as for exercise 4 (see **Figure 49**). Release 1.0 of this product has been tagged and released into production, and the working copy is clean (i.e., there are no uncommitted changes).

At this point, the develop branch does not exist yet so we need to create it before we can create a feature branch off of it. Creating a branch requires a commit, but we don't want to start work on the new feature (feature 1) until the *develop* branch is in place so we can create the feature branch off of *develop*. What to do?

This is merely a "getting started" issue that won't ever happen again once the *develop* branch exists. To facilitate creation of the develop branch, we could make a dummy modification to an existing file or we could add a new file. In this exercise, we'll create a readme.txt file which will become part of our releases going forward. We can then commit readme.txt to the repository and create the *develop* branch in the process. Figure 49 shows the revision history after doing this.

Graph	Rev	Branch	Description
	4+	develop	★ Working Directory ★
	4	develop	develop tip **added readme.txt**
	3	default	default Added tag Release 1.0 for changeset 6b30641c91ea
	2	default	Release 1.0 some more code
	1	default	some code
	0	default	initial commit

Figure 49. The develop branch has been created.

With the *develop* branch in place, we're ready to begin work on feature 1. We make the initial modification to my.prg and commit it to a new branch named Feature_1. The revision history at this point is illustrated in **Figure 50**. Looking at the Branch column reveals there are now three named branches in the repository.

Graph	Rev	Branch	Description
	5+	feature_1	★ Working Directory ★
	5	feature_1	feature_1 tip **begin work on feature 1**
	4	develop	develop added readme.txt
	3	default	default Added tag Release 1.0 for changeset 6b30641c91ea
	2	default	Release 1.0 some more code
	1	default	some code
	0	default	initial commit

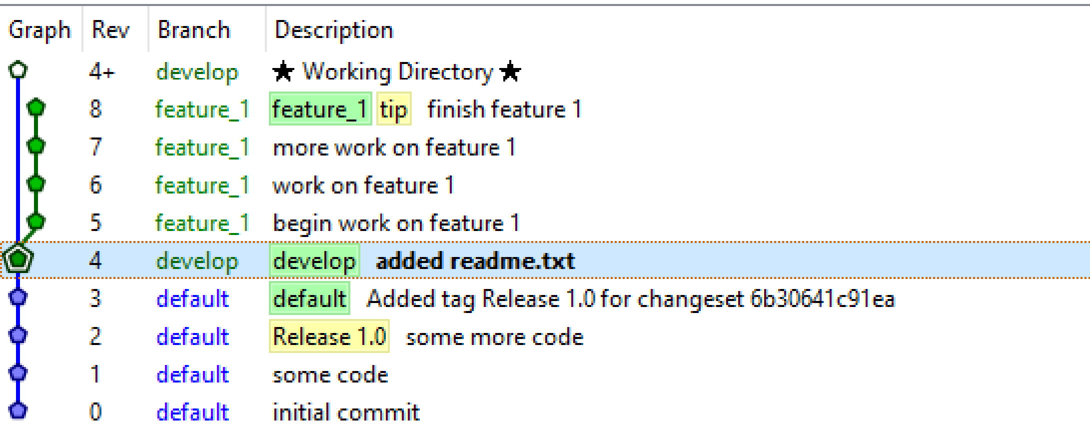
Figure 50. There are now three named branches in this repository.

Work continues on feature 1 with commits being made along the way, as in the previous exercises. All commits are on the *feature_1* branch. When feature 1 is finished, the revision history looks like **Figure 51**.

Graph	Rev	Branch	Description
	8+	feature_1	★ Working Directory ★
	8	feature_1	feature_1 tip **finish feature 1**
	7	feature_1	more work on feature 1
	6	feature_1	work on feature 1
	5	feature_1	begin work on feature 1
	4	develop	develop added readme.txt
	3	default	default Added tag Release 1.0 for changeset 6b30641c91ea
	2	default	Release 1.0 some more code
	1	default	some code
	0	default	initial commit

Figure 51. All development work on feature 1 was committed to the *feature_1* branch.

The rules for this workflow are that feature branches are always merged into the development branch, never directly into the release branch. To merge the *feature_1* branch into the develop branch, first update to the head of the *develop* branch at revision 4 (**Figure 52**).



Graph	Rev	Branch	Description
	4+	develop	★ Working Directory ★
	8	feature_1	feature_1 tip finish feature 1
	7	feature_1	more work on feature 1
	6	feature_1	work on feature 1
	5	feature_1	begin work on feature 1
	4	develop	develop **added readme.txt**
	3	default	default Added tag Release 1.0 for changeset 6b30641c91ea
	2	default	Release 1.0 some more code
	1	default	some code
	0	default	initial commit

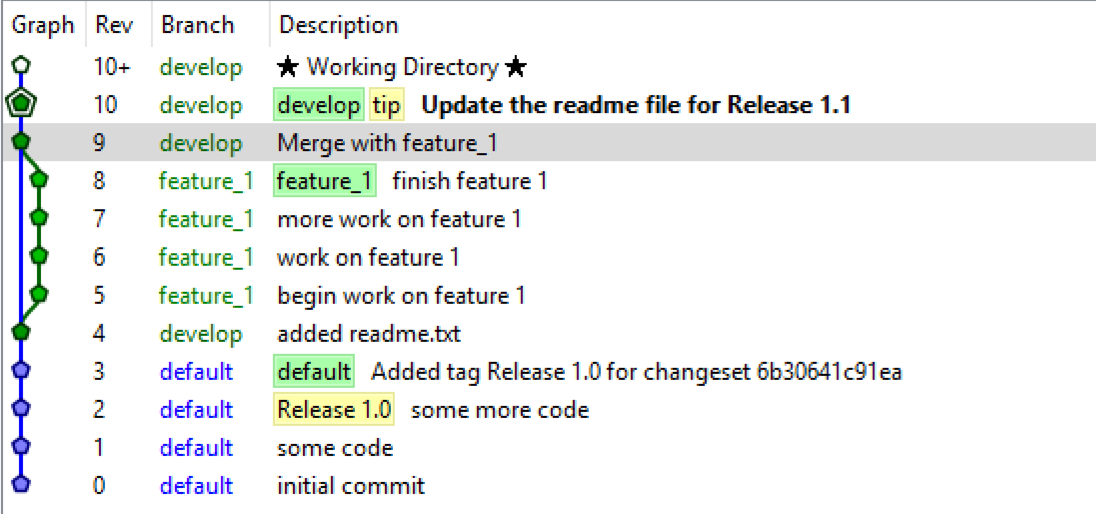
Figure 52. The working copy is updated to the head of the *develop* branch in preparation for merging.

Then merge in the head of the *feature_1* branch from revision 8 (**Figure 53**).

Graph	Rev	Branch	Description
	9+	develop	★ Working Directory ★
	9	develop	develop tip **Merge with feature_1**
	8	feature_1	feature_1 finish feature 1
	7	feature_1	more work on feature 1
	6	feature_1	work on feature 1
	5	feature_1	begin work on feature 1
	4	develop	added readme.txt
	3	default	default Added tag Release 1.0 for changeset 6b30641c91ea
	2	default	Release 1.0 some more code
	1	default	some code
	0	default	initial commit

Figure 53. The head of the *feature_1* branch has been merged into *develop*.

The development branch now has stable code ready for release, so it can be prepared for merging back into the *default* branch for the build and release cycle. Preparation for merging into the default branch may include things like updating the readme file and other housekeeping tasks not directly related to the source code. See revision 10 in **Figure 54**.



Graph	Rev	Branch	Description
	10+	develop	★ Working Directory ★
	10	develop	develop tip **Update the readme file for Release 1.1**
	9	develop	Merge with feature_1
	8	feature_1	feature_1 finish feature 1
	7	feature_1	more work on feature 1
	6	feature_1	work on feature 1
	5	feature_1	begin work on feature 1
	4	develop	added readme.txt
	3	default	default Added tag Release 1.0 for changeset 6b30641c91ea
	2	default	Release 1.0 some more code
	1	default	some code
	0	default	initial commit

Figure 54. Housekeeping activities like updating the readme file can be done before merging into *default*.

All that remains at this point is to merge the development branch back into *default*. The process should be familiar by now – update to the head of the target branch (revision 3 in *default*), then merge in the head of the source branch (revision 10 in *develop*). When finished, add a tag for Release 1.1.

The final result is shown in **Figure 55**. The graph clearly shows that two named branches were used in the development of feature 1. The green highlighting also clearly identifies the head of each branch.

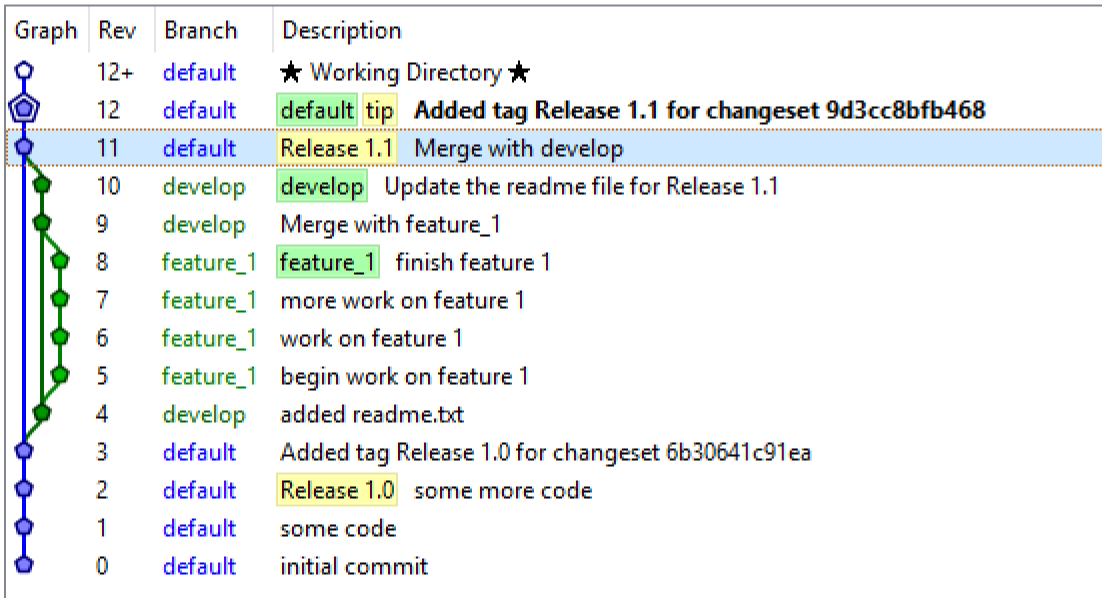


Figure 55. This is the revision history after merging *develop* into *default* and adding a tag for Release 1.1.

In this workflow, the idea is that each feature is developed on its own named branch with the name of the branch being unique to the feature. This implies that when a feature is completed, its branch is no longer needed and its name will not be used again. Unlike Git, in which feature branches are discarded when no longer needed, Mercurial maintains an immutable record of all commits, so completed feature branches will always appear in the revision history. Mercurial does provide a way to close a branch, which does not remove it but marks it as closed as an indication that it is no longer to be used. In a later exercise you'll see how and when to close a branch.

Pushing branches to remote repositories

This is a good time to talk about what happens when you push commits to a remote repository and there are branches in the local repository. When you initiate a push to a remote repository, Mercurial first builds a list of the commits that exist in the local repository but are not yet in the remote repository. By default, this list includes unmatched commits on all branches in the local repository. Sometime you do want to push all unmatched commits from all branches, but there are situations where you may want to push only one branch.

You can preview what will be pushed by using Mercurial's *outgoing* command before pushing. Once a local repository has been associated with one or more remotes, you can run the *outgoing* command from TortoiseHg by clicking on the *Detect outgoing changes* icon, third from the left on the sync toolbar.

To see this in action, we first need to link the local repo for exercise 5 with the remote repo in the HgCentral\Exercise05 folder. Click the Synchronize button on the task toolbar to open the Synchronize window in the task pane (bottom portion of the TortoiseHg window).

You can follow along with the next steps by referring in **Figure 56**. In the space provided, enter the full drive and path to the remote repository; you can either type in in by hand or use the *Browse* button (1) to select it. When the desired path appears in the field (2), click the Save button (3) and accept the alias *default*. Mercurial stores the alias and its associated path, which now shows up in the *Paths to Repository Settings* list (4).

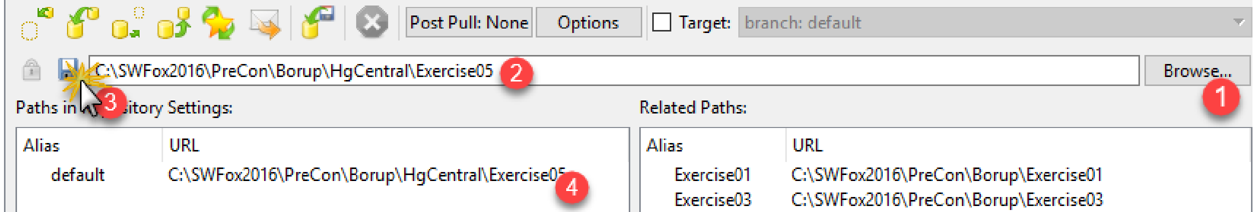


Figure 56. Follow these steps to link the local repository with a remote repository.

Now that the local repo for exercise 5 is associated with the corresponding remote, we can preview what will be pushed by clicking the *Detect outgoing changes* icon on the sync toolbar. Because nothing has yet been pushed to the remote, Mercurial tells us that all 13 commits (0 through 12) will be pushed. The preview is displayed in the upper pane, as shown in **Figure 57**.

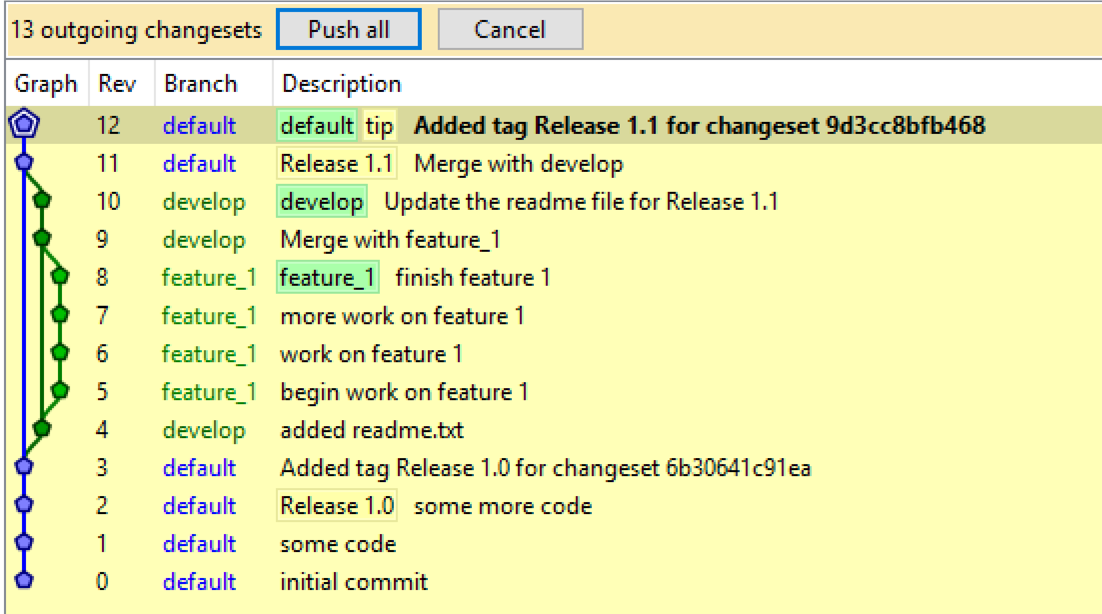


Figure 57. Use the *Outgoing* button to preview what will be pushed to the remote repository.

The default behavior is to push all changes from all branches, which you can trigger by clicking the *Push All* button. If you need to be selective about which branches and/or which commits to push, right-click on a revision and select one of the choices on the Push submenu as shown in **Figure 58**.

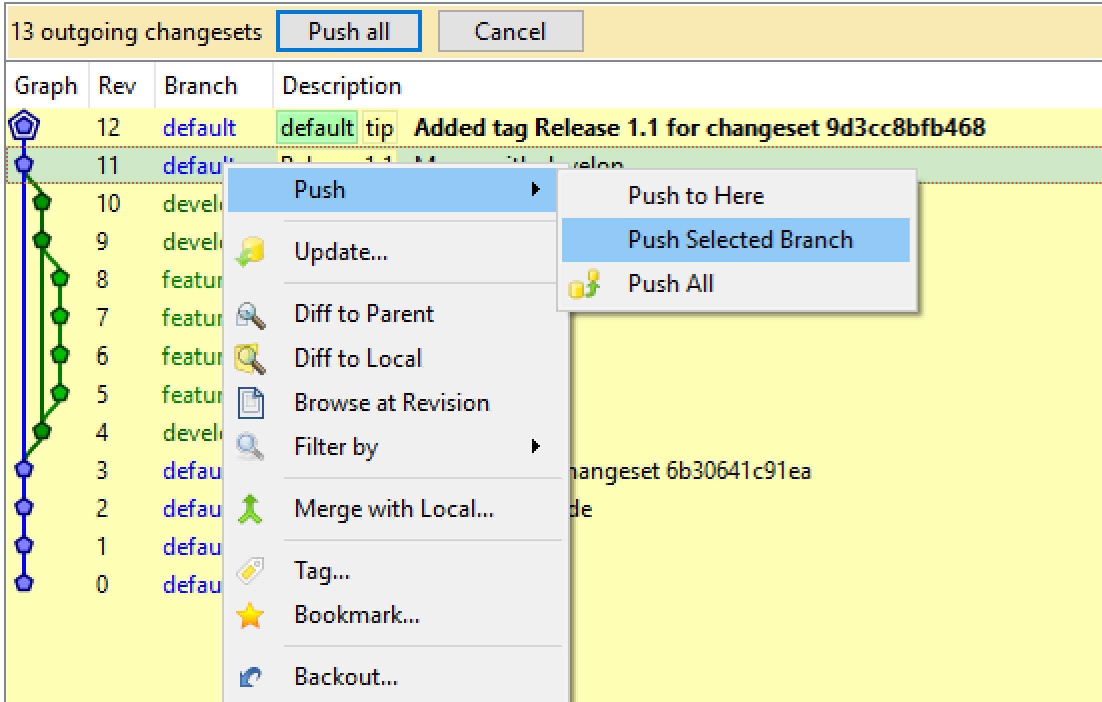


Figure 58. In TortoiseHg, the pop-up menu can be used to control which branch(es) get pushed to a remote.

Your choice of what to push will depend mostly on how your team's workflow is structured. If you're working as a solo developer, it makes sense to push all commits on all branches. If you're working with a team but are working solo on a new feature, you may or may not want to push the feature branch, at least not until it's finished. On the other hand, if you're collaborating with another team member on the development of a feature, you'll want to include the appropriate feature branch so you can push and pull revisions back and forth with your teammate.

For this exercise, we'll *Push All*. After doing so, the remote repository contains the same revision history as the local repository. The only difference is that the remote is a bare repository, meaning it has no working copy of the files. If you open the remote repository in TortoiseHg, you'll see the *Not a head revision* message at the top, as shown in **Figure 59**. This is normal and can be safely ignored.

Copyright 2016, Rick Borup

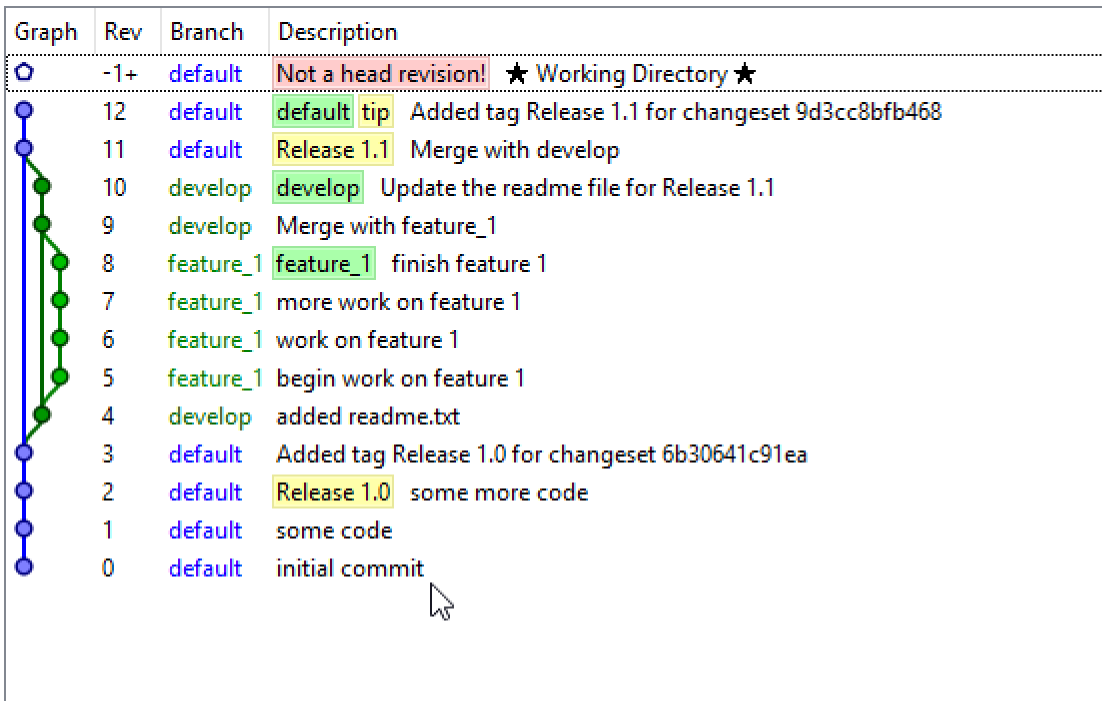


Figure 59. A shared remote repository typically has no working copy,

Lesson 6 – Non-linear workflow with multiple feature branches

Lesson 5 showed how to use a development branch and a feature branch to isolate development work from the release branch. This was a stepping stone on the way to where we really want to go, which is to be able to use multiple feature branches concurrently.

Consider the scenario in which two new features are under development at the same time. Let's suppose the company wants to release an update to its product as soon as one of the new features is ready, without waiting until the other feature is also ready. Sometimes it's uncertain which feature will be ready first. Using separate feature branches makes it possible to release whichever one is ready first.

Exercise 6

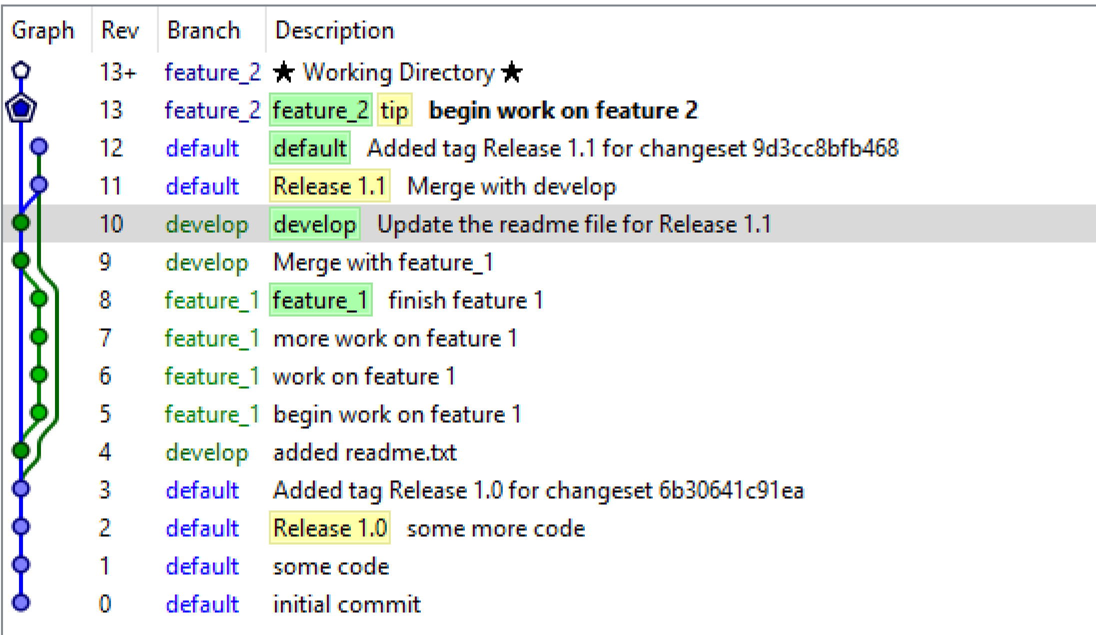
This exercise begins where exercise 5 left off. The only change is that the exercise files have been modified to reference a remote repository in HgCentral\Exercise06 instead of HgCentral\Exercise05.

In this exercise we add two new features using separate feature branches for each. Feature 2 will be developed in the Feature_2 branch, and feature 3 will be developed concurrently in the Feature_3 branch. As is common in a real life situation, development work switches back and forth between the two features as time and resources allow.

The first step is to update the working copy to the head of the *develop* branch. The rules for this workflow, as in Lesson 5, require that only the develop branch can merge into the release branch (*default*). No development takes place on the release branch, so the state of the code in the head of the *develop* branch is the same as the state of the code in the head of the release branch.

After updating to the head of the develop branch, work can begin on the new features. Let's say work begins on feature 2 before work on feature 3. The source code file my.prg is updated with the first set of modifications for feature 2, and these modifications are committed to a new branch named *feature_2* off the *develop* branch. The revision history at this point is shown in **Figure 60**.

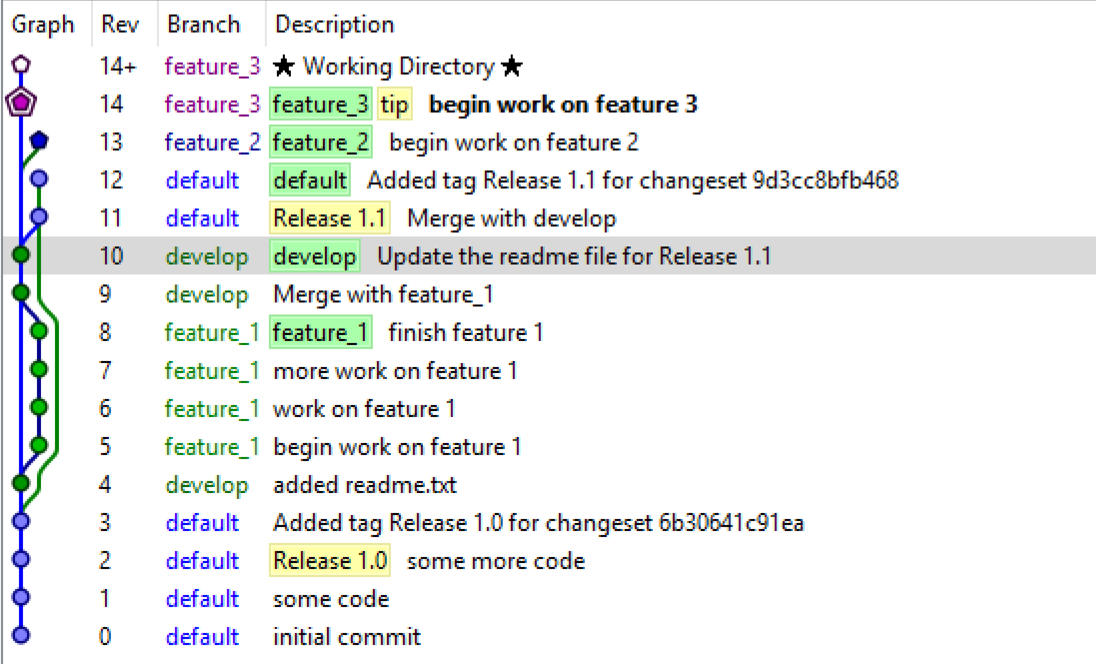
Copyright 2016, Rick Borup



Graph	Rev	Branch	Description
	13+	feature_2	★ Working Directory ★
	13	feature_2	feature_2 tip **begin work on feature 2**
	12	default	default Added tag Release 1.1 for changeset 9d3cc8bfb468
	11	default	Release 1.1 Merge with develop
	10	develop	develop Update the readme file for Release 1.1
	9	develop	Merge with feature_1
	8	feature_1	feature_1 finish feature 1
	7	feature_1	more work on feature 1
	6	feature_1	work on feature 1
	5	feature_1	begin work on feature 1
	4	develop	added readme.txt
	3	default	Added tag Release 1.0 for changeset 6b30641c91ea
	2	default	Release 1.0 some more code
	1	default	some code
	0	default	initial commit

Figure 60. A new branch named *feature_2* is created off of the *develop* branch.

The next step is to begin work on feature 3. We again start by updating the working copy to the head of the *develop* branch. We then make some modifications to my.prg for feature 3 and commit them to another new feature branch named *feature_3*, as shown in **Figure 61**. Note how TortoiseHg uses different colors to visually identify the different branch names in the Branch column.



Graph	Rev	Branch	Description
	14+	feature_3	★ Working Directory ★
	14	feature_3	feature_3 tip **begin work on feature 3**
	13	feature_2	feature_2 begin work on feature 2
	12	default	default Added tag Release 1.1 for changeset 9d3cc8bfb468
	11	default	Release 1.1 Merge with develop
	10	develop	develop Update the readme file for Release 1.1
	9	develop	Merge with feature_1
	8	feature_1	feature_1 finish feature 1
	7	feature_1	more work on feature 1
	6	feature_1	work on feature 1
	5	feature_1	begin work on feature 1
	4	develop	added readme.txt
	3	default	Added tag Release 1.0 for changeset 6b30641c91ea
	2	default	Release 1.0 some more code
	1	default	some code
	0	default	initial commit

Figure 61. Another new branch is created for the development of feature 3.

Work can now proceed concurrently on both feature 2 and feature 3 by updating the working copy back and forth between the two as needed. Work on feature is committed to the *feature_2* branch, while work on feature 3 is committed to the *feature_3* branch.

After working on both new features for a while and making a few commits along the way, feature 2 ends up being ready to be released first, before feature 3 (see **Figure 62**).

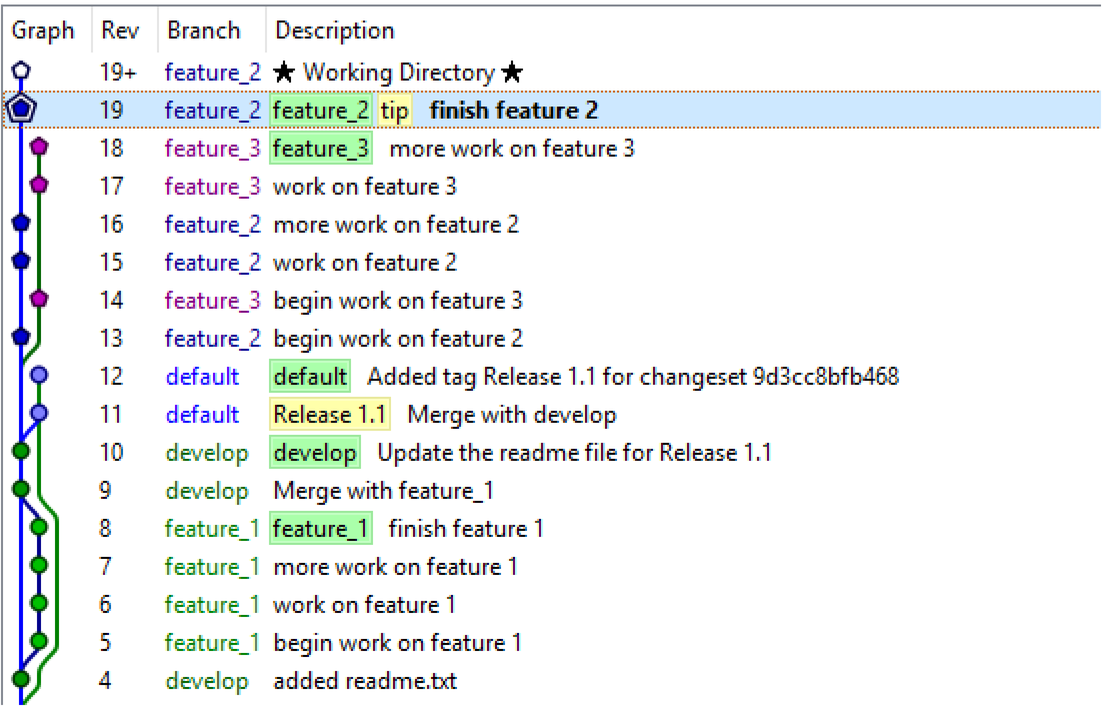
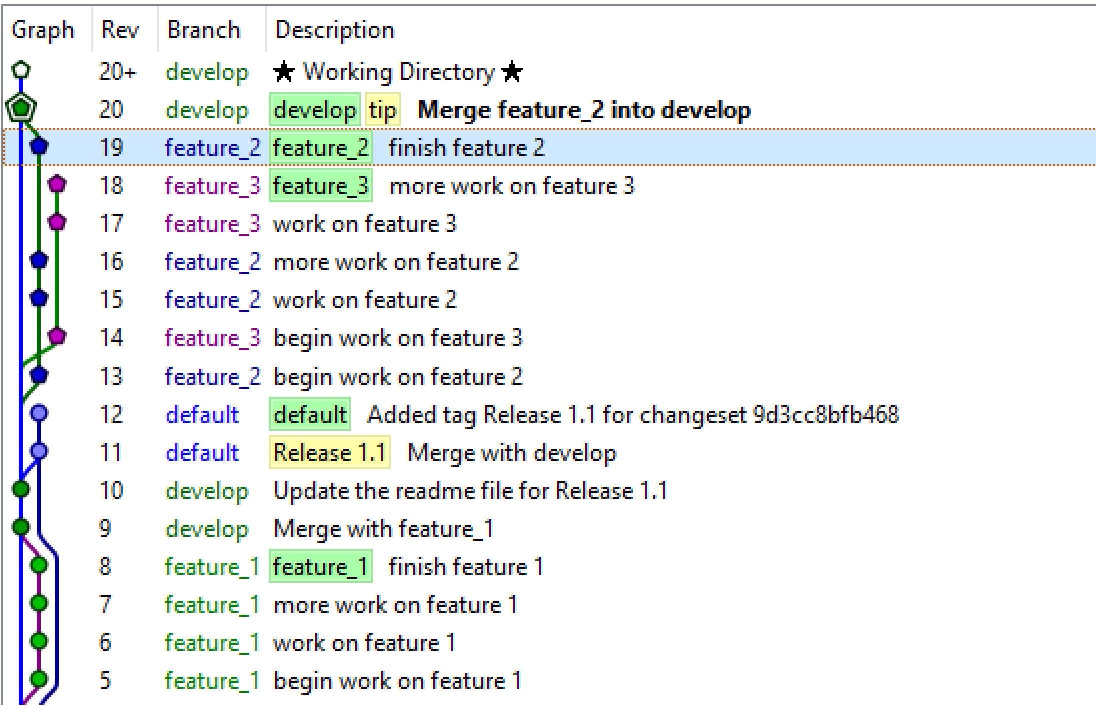


Figure 62. Feature 2 gets done before feature 3.

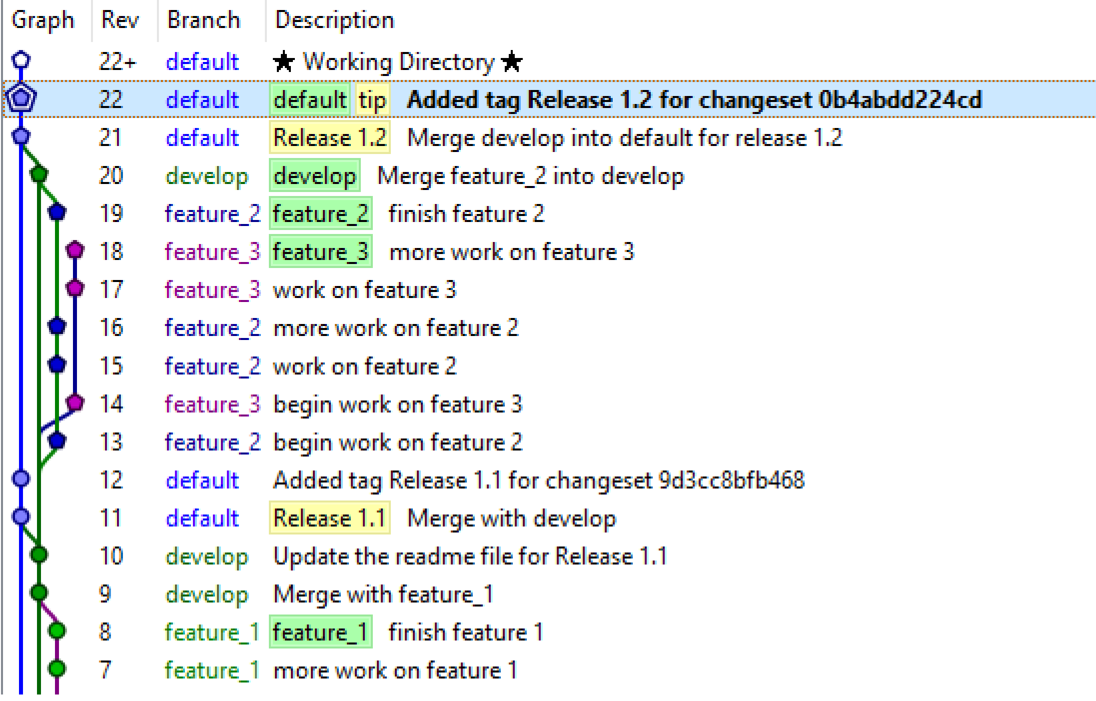
Again recalling the rules for this workflow, we can't go straight from a feature branch into the release branch. First we need to merge the *feature_2* branch back into *develop*. Follow the usual steps to do this – update the working copy to the head of the target revision (revision 10 on *develop*), then merge in the head of the source branch (revision 19 on *feature_2*). The result of this step is shown in **Figure 63**.



Graph	Rev	Branch	Description
	20+	develop	★ Working Directory ★
	20	develop	develop tip **Merge feature_2 into develop**
	19	feature_2	feature_2 finish feature 2
	18	feature_3	feature_3 more work on feature 3
	17	feature_3	work on feature 3
	16	feature_2	more work on feature 2
	15	feature_2	work on feature 2
	14	feature_3	begin work on feature 3
	13	feature_2	begin work on feature 2
	12	default	default Added tag Release 1.1 for changeset 9d3cc8bfb468
	11	default	Release 1.1 Merge with develop
	10	develop	Update the readme file for Release 1.1
	9	develop	Merge with feature_1
	8	feature_1	feature_1 finish feature 1
	7	feature_1	more work on feature 1
	6	feature_1	work on feature 1
	5	feature_1	begin work on feature 1

Figure 63. The *feature_2* branch is merged into *develop*.

The last step is to merge the head of the *develop* branch into the release branch. First, update the working copy to the head of the release (*default*) branch, which is revision 12. Then merge in the head of the *develop* branch from revision 20. Finally, add a tag for Release 1.2. The revision history now looks like **Figure 64**.



Graph	Rev	Branch	Description
	22+	default	★ Working Directory ★
	22	default	default tip **Added tag Release 1.2 for changeset 0b4abdd224cd**
	21	default	Release 1.2 Merge develop into default for release 1.2
	20	develop	develop Merge feature_2 into develop
	19	feature_2	feature_2 finish feature 2
	18	feature_3	feature_3 more work on feature 3
	17	feature_3	work on feature 3
	16	feature_2	more work on feature 2
	15	feature_2	work on feature 2
	14	feature_3	begin work on feature 3
	13	feature_2	begin work on feature 2
	12	default	Added tag Release 1.1 for changeset 9d3cc8bfb468
	11	default	Release 1.1 Merge with develop
	10	develop	Update the readme file for Release 1.1
	9	develop	Merge with feature_1
	8	feature_1	feature_1 finish feature 1
	7	feature_1	more work on feature 1

Figure 64. The *develop* branch is merged into *default* and Release 1.2 is tagged.

(You may have caught the fact that I neglected to update the readme file, which I should have done on the *develop* branch before merging it into *default*. Mistakes happen.)

We're now ready to resume work on feature 3. Looking at the revision history in Figure 64, you can see that the head of the *feature_3* branch is still sitting there at revision 18. To pick up with development on that feature again, first update the working copy to that head.

Let's say the next set of modifications to my.prg completes feature 3. To prepare for release, we follow the same sequence of steps that we used to release feature 2 – merge the *feature_3* branch back into *develop*, remember to update the readme file this time, then merge *develop* into *default* and apply a tag for Release 1.3.

Because of the way the my.prg source code file for this exercise is structured, the revisions for feature 2 occupy the same lines as the revisions for feature 3. This causes a merge conflict when we merge *feature_3* into *develop*, because my.prg already contains feature 2 at that point. The merge conflict can be resolved using KDiff3, the same way it was done back in Lesson 3. In a real world project, it's entirely possible there would be no merge conflict. For example, feature 2 might have involved changes to a different section of the source code file than feature 3, or perhaps the two features were added in different source code files altogether.

The resulting revision history is shown in **Figure 65**. Notice that with this workflow we can easily see which branch each revision belongs to because the full branch history is maintained even after merging.

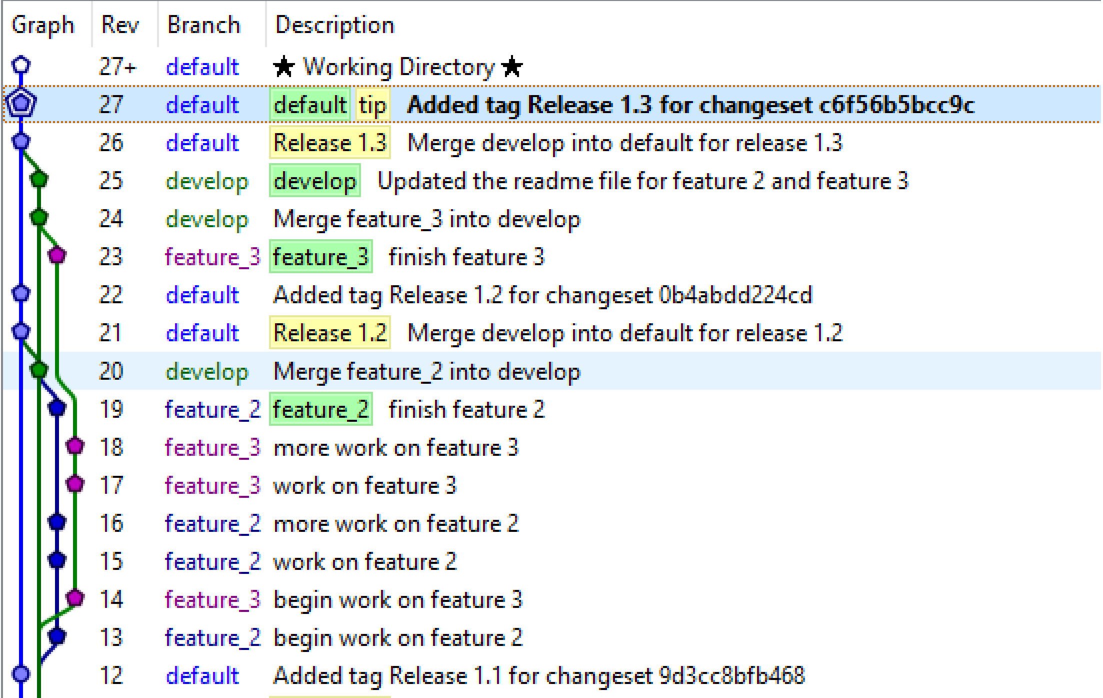


Figure 65. The revision history clearly shows the separation between the development of the two features.

If a remote repository were in use here, we could have pushed changes to it along the way. For the solo developer, these intermediate pushes serve as a backup if nothing else. For developers working as a team, these intermediate pushes are not only a backup but also a way to share revisions among the team members. If you want to, you can now push all the changes to HgCentral\Exercise06, which is already established a remote repository for this exercise, using the *Push All* button as shown in **Figure 66**.

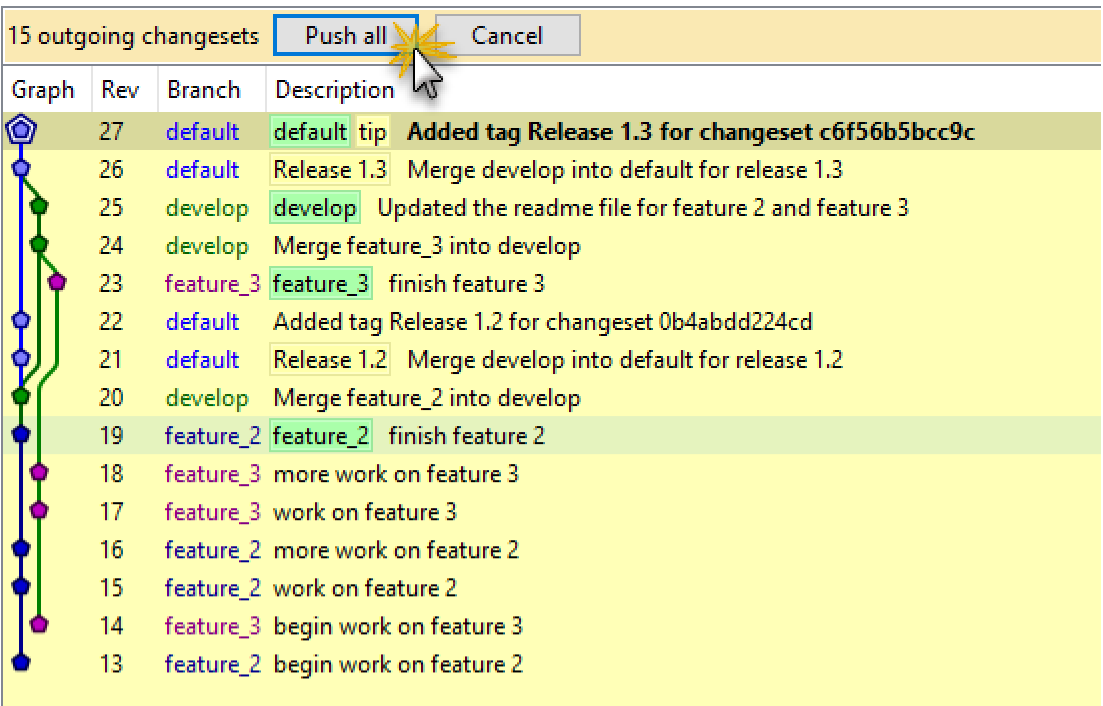


Figure 66. For the sake of this exercise, *Push All* is used to push all commits on all branches to the remote.

Before completing the push, Mercurial alerts you that the remote repository does not yet contain the feature_2 or feature_3 branches with the dialog shown in **Figure 67**. In this case we do want to create the new branches in the remote repository, so click Yes.

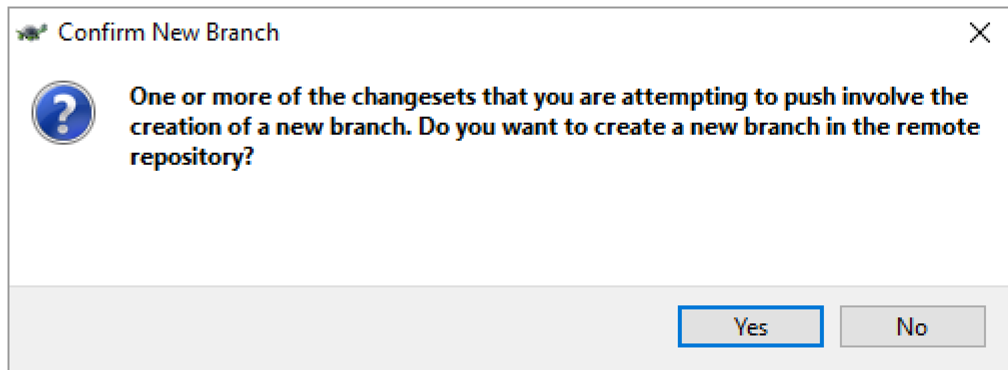


Figure 67. Mercurial alerts you that the remote does not yet contain some of the branches being pushed.

Lesson 7 – Closing a branch

The workflow in lessons 5 and 6 uses a unique feature branch for every new feature. Because each feature branch is specific to the feature for which it's created, feature branches are never reused.

Mercurial maintains an immutable record of all commits to the repository. You can't delete a branch after you're done with it, even if you wanted to.[3] So what do you do in Mercurial to indicate you're finished work on a branch and it's no longer to be used?

You close it.

Closing a branch creates a commit that marks the branch as closed. Closed branches are not included when you run the *hg branches* command to list the branches in the repository. In TortoiseHg, the commit that closes a branch is marked with a special symbol as a visual indication that the branch has been closed. Although closing a branch does not actually prevent it from ever being used again, the special symbol in the revision history lets everyone know it's been closed.

In a workflow with multiple feature branches like we've been using in lessons 5 and 6, it's a good idea to close each feature branch when you're finished with the development of that feature. That way, anybody looking at the revision history knows the feature is done and understands the feature branch is not supposed to be used again.

The only thing that makes a branch unique is its name. It's recommended that you adopt a naming convention for branches that not only distinguishes them from tags and bookmarks but also insures that each branch gets a name that will remain unique for the life of the repository. The feature branch naming convention used in the exercises in this paper – feature_1, feature _2, etc. – is simplistic but not very descriptive. A better convention would perhaps be the word *feature* followed by something briefly descriptive, such as a reference to an issue number in a bug tracking system or a task ID in a project management system.

Closing a branch is easy, as you'll see in the exercises for this lesson. The only question is whether to close a branch before or after merging it back into its parent. In the workflow we're using here, this means deciding whether to close a feature branch before or after merging it back into *develop*.

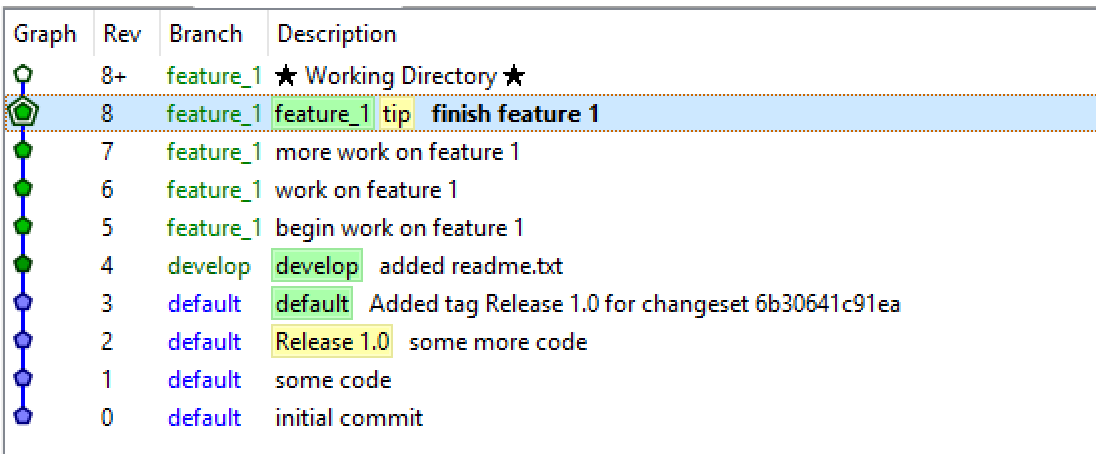
Close-then-merge is simpler and generally preferred. Merge-then-close takes more work and creates a dangling head in the repository. Dangling heads are not only a visual anomaly in the graphical revision history, but could over time lead to performance issues if the repository accumulates a large number of them.

Exercise 7 explores both approaches so you can see how they work.

[3] And why would you want to? Deleting a feature branch would make it difficult if not impossible to tell which commits were part of which features.

Exercise 7

A good starting point for this exercise is the step in exercise 5 where we're about to merge the *feature_1* branch into *develop*. At that point, the revision history looks like **Figure 68**.



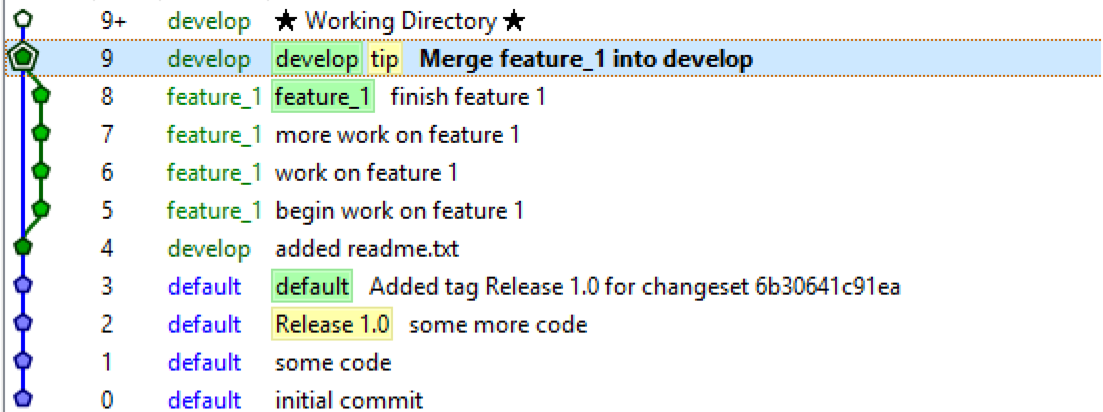
Graph	Rev	Branch	Description
	8+	feature_1	★ Working Directory ★
	8	feature_1	feature_1 tip **finish feature 1**
	7	feature_1	more work on feature 1
	6	feature_1	work on feature 1
	5	feature_1	begin work on feature 1
	4	develop	develop added readme.txt
	3	default	default Added tag Release 1.0 for changeset 6b30641c91ea
	2	default	Release 1.0 some more code
	1	default	some code
	0	default	initial commit

Figure 68. The revision history at the point where *feature_1* is about to be merged into *develop*.

This is the common starting point for both exercise 7a and exercise 7b. In both cases, we want to close the *feature_1* after we're done with it.

Exercise 7a

This exercise shows what happens under the merge-then-close approach. We start by updating the working copy to the head of the *develop* branch at revision 4, and then merge in the head of *feature_1* from revision 8, as we did in exercise 5. The revision history now looks like **Figure 69**.



Graph	Rev	Branch	Description
	9+	develop	★ Working Directory ★
	9	develop	develop tip **Merge feature_1 into develop**
	8	feature_1	feature_1 finish feature 1
	7	feature_1	more work on feature 1
	6	feature_1	work on feature 1
	5	feature_1	begin work on feature 1
	4	develop	added readme.txt
	3	default	default Added tag Release 1.0 for changeset 6b30641c91ea
	2	default	Release 1.0 some more code
	1	default	some code
	0	default	initial commit

Figure 69. The revision history after merging *feature_1* into *develop*, but before closing *feature_1*.

To close the feature_1 branch after the merge, we first have to update back to its head at revision 8. Having done that, click the *Commit* icon on the task toolbar to open the Commit pane. Confirm that the branch identifier at the top of the Commit pane says *feature_1,* then click on that name and mark the *Close current branch* option as shown in **Figure 70**.

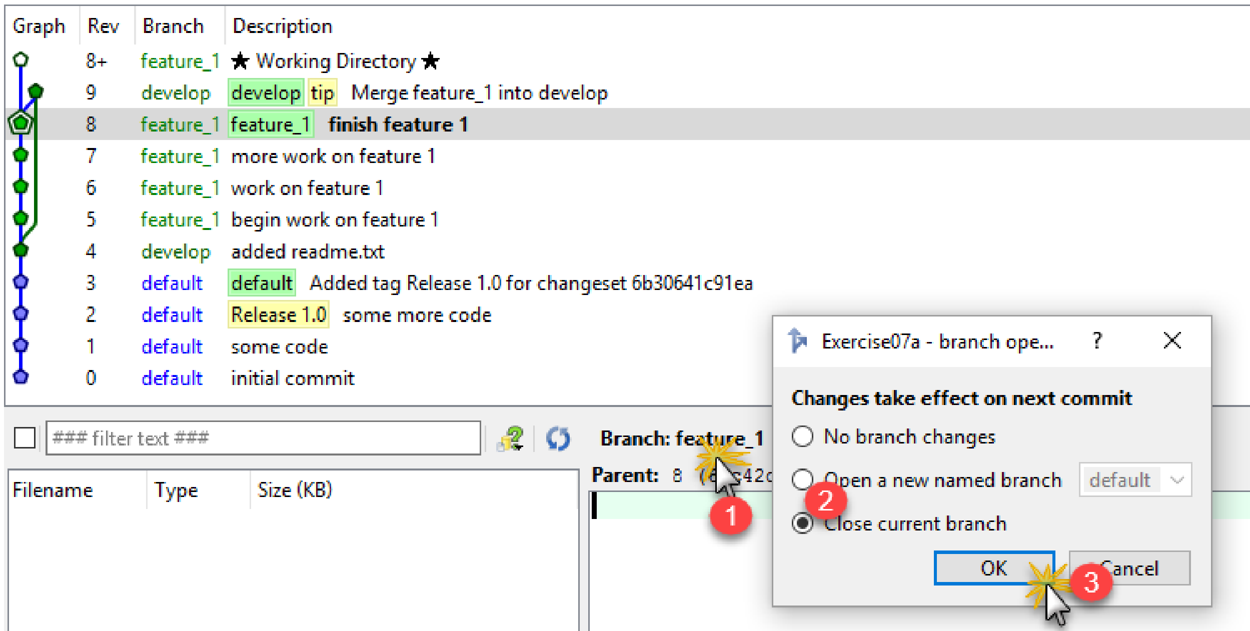


Figure 70. Use the *branch operations* dialog to close a branch.

This sets up a commit with the default commit message of "Close feature_1 branch". Click the *Commit* button to record the commit and close the branch. The result is shown in **Figure 71**.

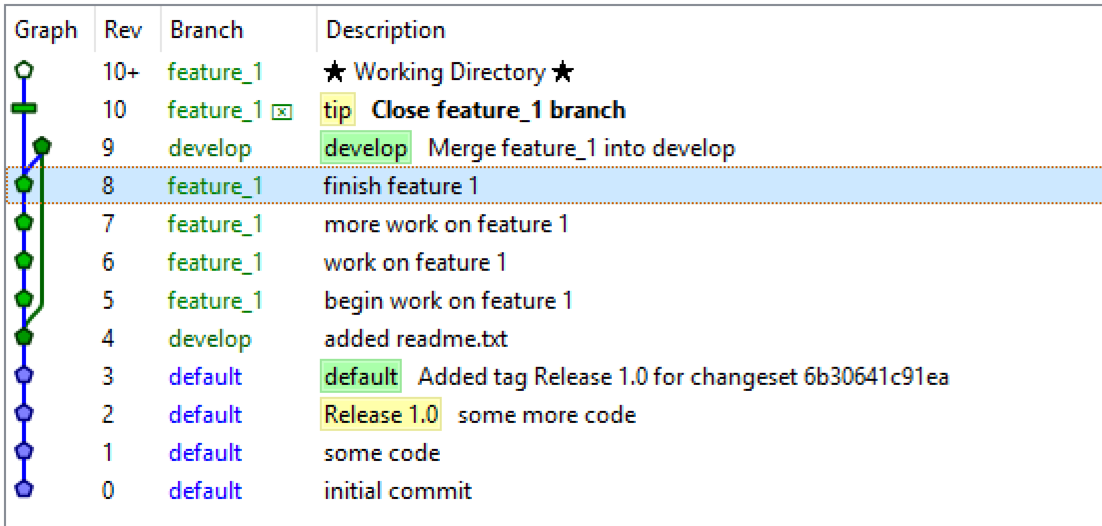


Figure 71. The *feature_1* branch has been closed but is still the parent of the working copy.

Note the block-shaped symbol on revision 10, indicating the branch is closed. At this point, *feature_1* is still the active branch in the repository, even though it's closed. To finish the release of feature 1, or to resume or begin work on another feature, we first have to update to the head of the *develop* branch. Doing so leaves the revision history with a dangling head, as shown in **Figure 72**.

Copyright 2016, Rick Borup

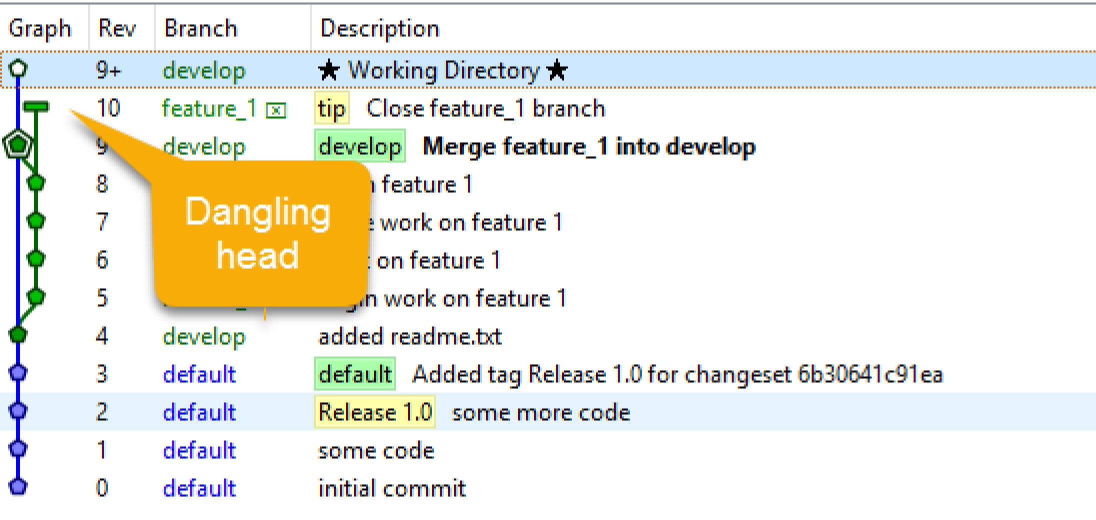


Figure 72. The merge-then-close approach results in a dangling head.

The dangling head will persist at that location in the revision history for the life of the repository.

Exercise 7b

In contrast to the above, let's now begin again from the same starting point as exercise 7a but this time use the close-then-merge approach.

In exercise 7a, we updated to *develop,* merged in the *feature_1* branch, then went back to *feature_1* and closed it, which left the dangling head. Here in exercise 7b, we first close *feature_1*, then merge it into *develop*.

The process for closing the branch and performing the merge is the same as in exercise 7a. The only difference is the sequence of steps. The result after performing the close-then-merge steps is show in **Figure 73**.

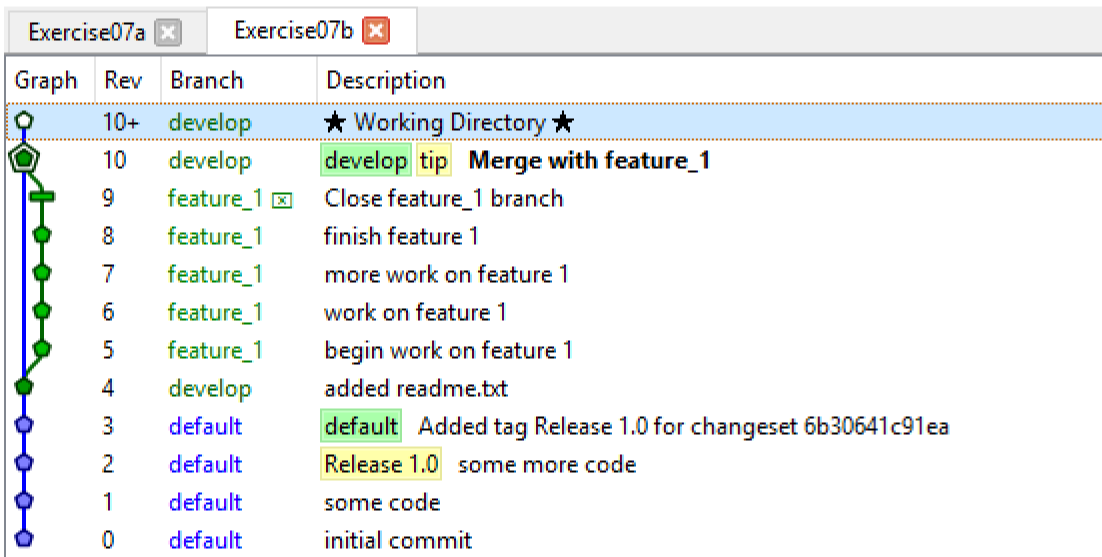


Figure 73. The close-then-merge approach is cleaner and does not leave a dangling head.

The close-then-merge approach not only requires fewer steps than merge-then-close, it also results in a cleaner revision history with no dangling heads. For this reason, close-then-merge is generally the preferred approach.[4]

[4] While researching this paper, I found many useful references to the merge-then-close vs close-then-merge debate. One of the best is on stackoverflow.com. You can read it at http://stackoverflow.com/questions/17169232/whats-the-best-way-to-close-a-mercurial-branch.

Lesson 8 – Hotfixes during feature development

A bug discovered during development is fixed during development. A bug discovered after release requires a decision –wait and fix the bug in the next release, or create a hotfix and release an update immediately?

The answer usually depends on the severity of the bug. If you can wait and fix it in the next planned release, the bug fix simply becomes part of that development cycle. On the other hand, if the bug requires a hotfix and development work on the next planned release has already begun, then a different approach is needed. That's the subject of this lesson.

Let's say version 1.3 of the product has already been released as is in use by customers. Work on release 1.4 has already begun, so the repository contains at least one feature branch being used for its development. Then a customer reports a bug in release 1.3 and it's serious enough you decide the fix can't wait for release 1.4. Instead, you need to fix the bug immediately and release v1.3.1 to your customers.

The question is, on which branch do you create the hotfix? You can't create the hotfix on the development branch and use the normal workflow to release it from there, because work on the feature currently under development on the development branch isn't finished yet and the hotfix can't wait for it to be done.

Because the bug is in the release version, it exists in the code on the release branch. One common solution is to create a hotfix branch off of the release branch, make the hotfix there, and then merge it into the release branch for the new build. This does create an exception to the rule that only the develop branch is allowed to merge into the release branch, but it makes sense if you think of the hotfix branch as a special one-time version of the *develop* branch.

There's a bit more to it, though. If the bug exists in the release version it also exists in the version under development, so the fix has to be applied there as well. One way to do this is to manually replicate the fix in the work currently under development. Another way is to graft the fix from the hotfix branch into the current development work.

Exercise 8 demonstrates how a hotfix can be created and merged into both the release version and the version under development.

Exercise 8

The starting point for exercise 8 begins where we left off in exercise 6. Release 1.3 has been completed and is in use by customers, and work on feature 4 for release 1.4 has already begun. **Figure 74** shows the revision history at this point.

Copyright 2016, Rick Borup

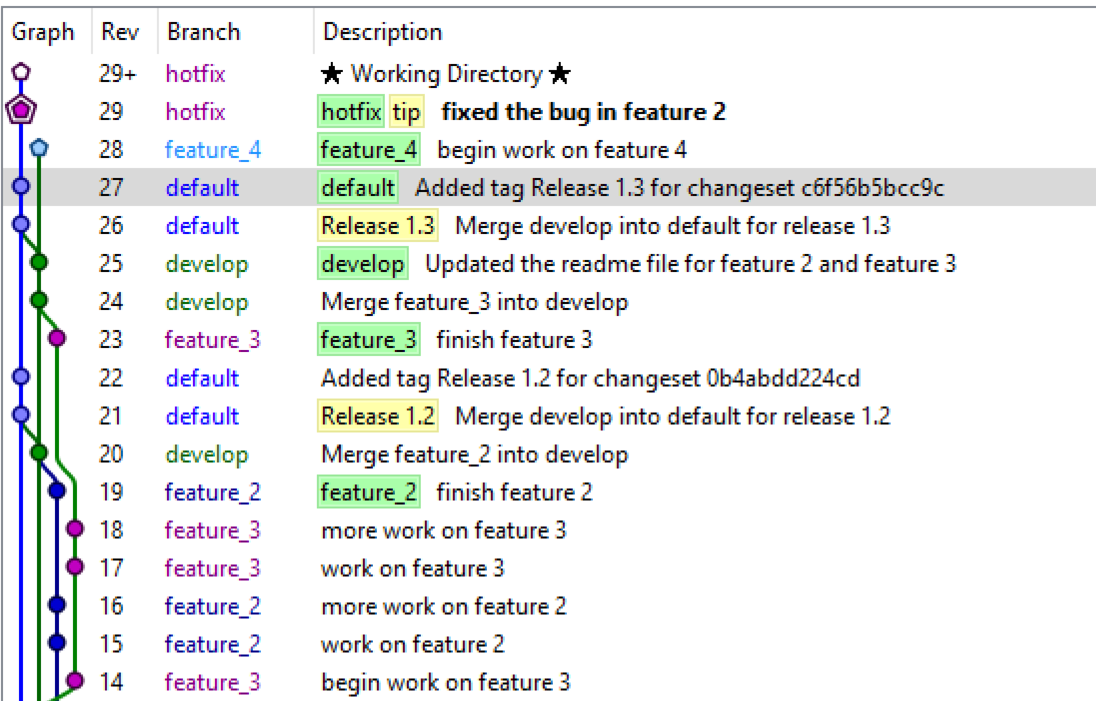
Graph	Rev	Branch	Description
	28+	feature_4	★ Working Directory ★
	28	feature_4	feature_4 tip **begin work on feature 4**
	27	default	default Added tag Release 1.3 for changeset c6f56b5bcc9c
	26	default	Release 1.3 Merge develop into default for release 1.3
	25	develop	develop Updated the readme file for feature 2 and feature 3
	24	develop	Merge feature_3 into develop
	23	feature_3	feature_3 finish feature 3
	22	default	Added tag Release 1.2 for changeset 0b4abdd224cd
	21	default	Release 1.2 Merge develop into default for release 1.2
	20	develop	Merge feature_2 into develop
	19	feature_2	feature_2 finish feature 2
	18	feature_3	more work on feature 3
	17	feature_3	work on feature 3
	16	feature_2	more work on feature 2
	15	feature_2	work on feature 2
	14	feature_3	begin work on feature 3
	13	feature_2	begin work on feature 2

Figure 74. A bug in the release version is reported after work has already begun on a new feature.

When the bug is reported by customers, we determine that it's serious enough to warrant a hotfix and a release of version 1.3.1. To begin work on the hotfix, we temporarily stop work on feature 4 and update back to the head of the release branch, which is revision 27 on the *default* branch. We then make whatever modifications to the source code are necessary to implement the fix and commit those changes to a new branch named *hotfix*. Finally, we merge the *hotfix* branch into the *default* branch, tag it with the release number, and build the new release containing the fix from there.

The first part of this process is shown in **Figure 75**, in which the *hotfix* branch has been created, the bug has been fixed, and the changes have been committed to that branch. It's not uncommon in the literature to see diagrams where the hotfix branch is shown off to the left of the release branch, as a way of indicating that it's separate from the development and feature branches on the right. While showing a hotfix branch off to the left is great for illustration purposes, TortoiseHg and other tools don't render it that way.

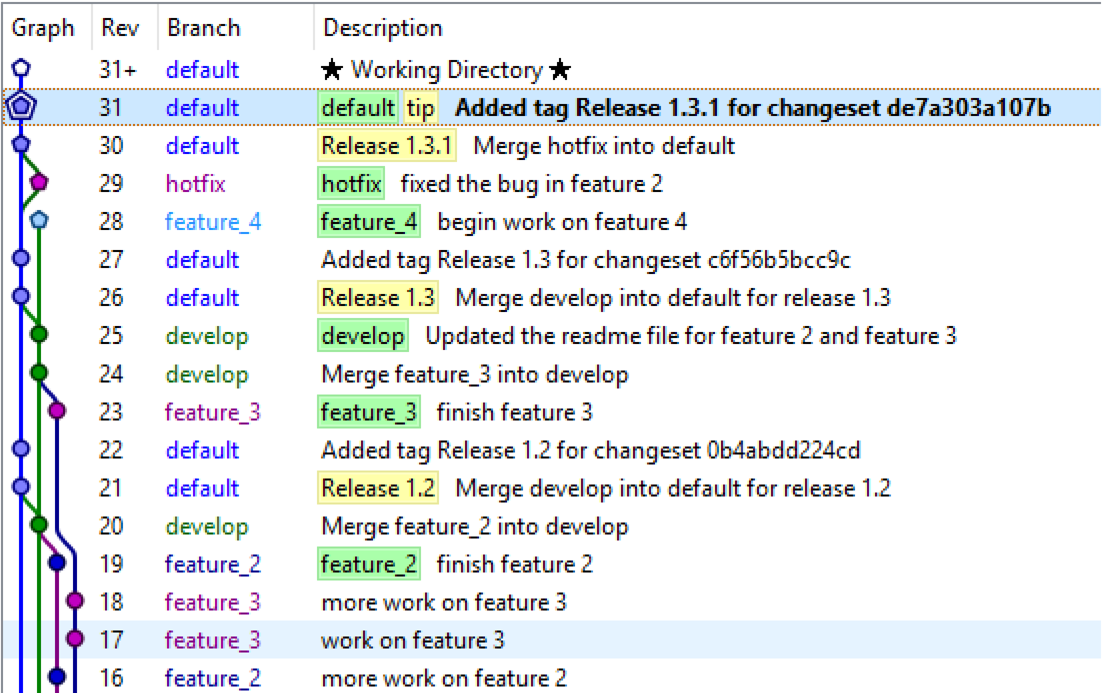
Copyright 2016, Rick Borup



Graph	Rev	Branch	Description
	29+	hotfix	★ Working Directory ★
	29	hotfix	hotfix tip **fixed the bug in feature 2**
	28	feature_4	feature_4 begin work on feature 4
	27	default	default Added tag Release 1.3 for changeset c6f56b5bcc9c
	26	default	Release 1.3 Merge develop into default for release 1.3
	25	develop	develop Updated the readme file for feature 2 and feature 3
	24	develop	Merge feature_3 into develop
	23	feature_3	feature_3 finish feature 3
	22	default	Added tag Release 1.2 for changeset 0b4abdd224cd
	21	default	Release 1.2 Merge develop into default for release 1.2
	20	develop	Merge feature_2 into develop
	19	feature_2	feature_2 finish feature 2
	18	feature_3	more work on feature 3
	17	feature_3	work on feature 3
	16	feature_2	more work on feature 2
	15	feature_2	work on feature 2
	14	feature_3	begin work on feature 3

Figure 75. To fix the bug, a *hotfix* branch is created off the release (*default*) branch.

After committing the changes to the *hotfix* branch, we update to the head of the *default* branch, merge in the *hotfix*, and apply a release tag to the head revision on *default*. The resulting revision history is shown in **Figure 76**.



Graph	Rev	Branch	Description
	31+	default	★ Working Directory ★
	31	default	default tip **Added tag Release 1.3.1 for changeset de7a303a107b**
	30	default	Release 1.3.1 Merge hotfix into default
	29	hotfix	hotfix fixed the bug in feature 2
	28	feature_4	feature_4 begin work on feature 4
	27	default	Added tag Release 1.3 for changeset c6f56b5bcc9c
	26	default	Release 1.3 Merge develop into default for release 1.3
	25	develop	develop Updated the readme file for feature 2 and feature 3
	24	develop	Merge feature_3 into develop
	23	feature_3	feature_3 finish feature 3
	22	default	Added tag Release 1.2 for changeset 0b4abdd224cd
	21	default	Release 1.2 Merge develop into default for release 1.2
	20	develop	Merge feature_2 into develop
	19	feature_2	feature_2 finish feature 2
	18	feature_3	more work on feature 3
	17	feature_3	work on feature 3
	16	feature_2	more work on feature 2

Figure 76. The *hotfix* branch is merged back into *default*.

At this point, version 1.3.1 of the product is ready to be go. The installer can be built from this code and released to production.

Having finished the hotfix, shipped the product, and made the customers happy again, we can now get back to work on the development of feature 4. To pick up where we left off, we start by updating the working copy to the head of the *feature_4* branch. But before starting to write more new code for feature 4, however, we remember that our working copy does not yet contain the hotfix. We need to apply the hotfix to the working copy so the bug doesn't crop up again during testing, or worse, in the next production release.

One way to get the hotfix into the current development work is to key those changes in again manually. If it's a quick fix, maybe even a one-line change, you might be tempted to do it that way. While that works, it leaves the revision history without any record of how it was done. Another way to do it, and one that does create a trail in the revision history, is to graft the change from the hotfix branch into the working copy.

Grafting, sometimes called cherry picking, is the process of taking a changeset from one line of development and applying it to another. That's exactly what we want to do here. TortoiseHg makes grafting easy – simply right-click on the changeset you want to graft in and choose *Graft to local* from the pop-up menu.

In this exercise we want to graft the hotfix, which is revision 29, into the code in the working copy, which is now the head of *feature_4*. Right-clicking on revision 29 and choosing *Graft to local* brings up the dialog shown in **Figure 77**.

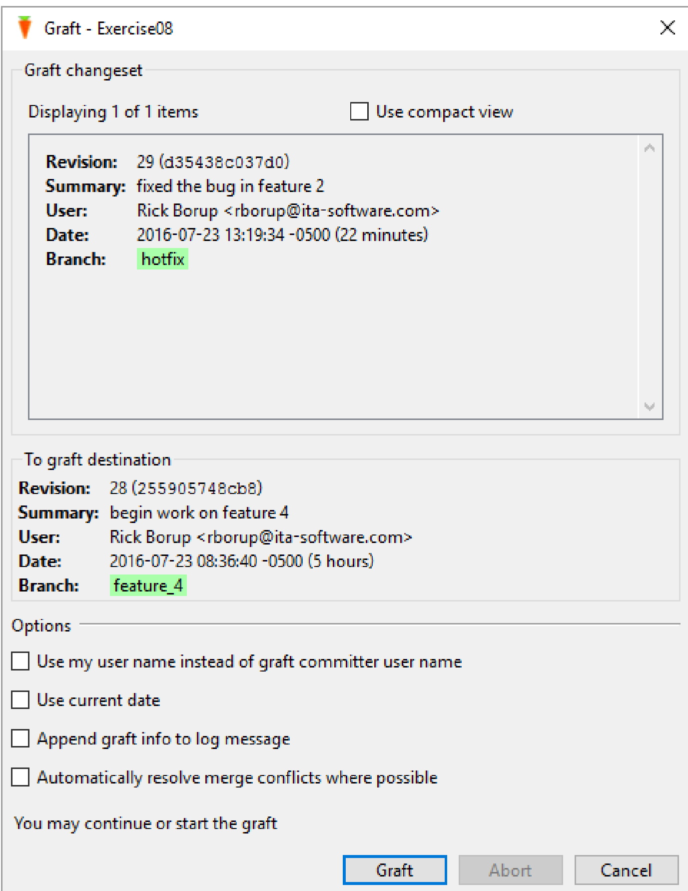


Figure 77. Grafting is a way to take a changeset from one branch and apply it to another.

A graft generates a merge followed by a commit. The merge may result in a merge conflict. If the *Automatically resolve merge conflicts* check box is marked, you may want to clear it so you can handle merge conflicts yourself. The line at the bottom of the dialog in Figure 77 tells you Mercurial has determined it's OK to continue with the graft, so click the *Graft* button to proceed.

Because all the work in these exercises is being done in my.prg, and because the hotfix inserted a line between two previously existing lines of code, Mercurial detects a merge conflict. Resolve the conflict in the usual manner and then complete the graft.

Completing the graft creates a new commit on the *feature_4* branch. As you can see in **Figure 78**, the revision history now indicates that a graft was performed by showing a dotted line from revision 29 to the new head of the *feature_4* branch at revision 32.

Graph	Rev	Branch	Description
	32+	feature_4	★ Working Directory ★
	32	feature_4	feature_4 tip **fixed the bug in feature 2**
	31	default	default Added tag Release 1.3.1 for changeset de7a303a107b
	30	default	Release 1.3.1 Merge hotfix into default
	29	hotfix	hotfix fixed the bug in feature 2
	28	feature_4	begin work on feature 4
	27	default	Added tag Release 1.3 for changeset c6f56b5bcc9c
	26	default	Release 1.3 Merge develop into default for release 1.3
	25	develop	develop Updated the readme file for feature 2 and feature 3
	24	develop	Merge feature_3 into develop
	23	feature_3	feature_3 finish feature 3
	22	default	Added tag Release 1.2 for changeset 0b4abdd224cd
	21	default	Release 1.2 Merge develop into default for release 1.2
	20	develop	Merge feature_2 into develop
	19	feature_2	feature_2 finish feature 2
	18	feature_3	more work on feature 3
	17	feature_3	work on feature 3

Figure 78. TortoiseHg uses a dotted line to show a graft.

The advantage of using a graft over simply keying in the hotfix again as part of the next commit on *feature_4* is that the revision history carries a clear record of how the changes got there.

Having grafted in the hotfix, we can now proceed with the development of feature 4 using the standard workflow. When finished, we merge *feature_4* into *develop* and then *develop* into *default* for release, as in the previous exercises.

Copyright 2016, Rick Borup

Lesson 9 – Managing multiple release versions of the same product

This lesson is about how to handle version control when there are two, or perhaps even more than two, versions of the same product in production at the same time.

Let's say a product starts out with version 1.0. It's in use by several customers and evolves over time from v1.0 to v1.5 with point releases along the way, which can collectively be referred to as version 1.x. Many customers say they love v1.x and are content to keep on using, but some customers have started asking for new features that will require a major reworking of much of the code. You decide to begin work on a brand new version, version 2.0, to meet the needs of those customers while still maintaining and releasing updates to version 1.x for the customers who are happy to stick with it.

Let's also say that a version control system has been in use for v1.x all along, and you want to use version control for the development of version 2. The starting point of the code base for version 2.0 is the code base for whatever the current release of version 1.x is at the time development of version 2.0 begins. Version 2.0 therefore becomes a divergent line of development – a branch, if you will – off the current release of version 1.x.

Over time, the ongoing development of version 2 will generate releases 2.0, 2.1, 2.2 and so on. Because version 1.x will continue to be actively maintained and enhanced as well, there will also be releases 1.6, 1.7, etc. of that product. Future releases of version 1.x and the initial work on version 2.0 both begin from the same starting point, but permanently diverge going forward.

How to you handle the version control for this situation? What are the alternatives? Is one way better than the others?

Some of the literature on distributed version control systems talks about the "big picture" vs the "little picture" view of how a DVCS is structured. Decisions about how many branches to use within a repository, what to name them, and which ones can be merged into which other ones are "little picture" views. Decisions about how to separate major lines of development, such as maintaining version 1.x and 2.x of a product independently of one another, require thinking about the "big picture".

Let's look at some of the "big picture" alternatives as they apply here.

Create a brand new repository

A simple way to begin separate development on version 2.0 is to copy the source code files from the current state of version 1.x into a new, unrelated folder. If the intent is to create a brand new repository, do not copy the .hg subfolder – i.e., do not copy the repository. Instead, initialize a new repository in the new folder and use it for version 2.0. The initial commit to the new repository is the initial state of the files in the copy of the working directory, before any modifications are made.

This approach creates a complete break between version 1.x and version 2.x. Because the two repositories are no longer related, you won't be able to push or pull changes between

them. This could be the preferred choice if revisions to version 1.x will never by directly applied to version 2.x or vice versa.

Create a clone

Another way is to create a clone of the current release of version 1.x and use it as the starting point for version 2.0. The difference between creating a clone and creating a brand new repository is that when you create a clone you get a complete copy not only of the working directory but also of the entire revision history. Because the new repo is a clone of the original, the two repositories remain related. This means you can push and pull revisions between them if it ever becomes necessary or expedient to do so.

Creating a clone might be the preferred choice is (a) it's important that the repository for version 2.x carries all the history from the development of version 1.x, and (b) it's likely you may want to exchange revisions between the two repositories.

Create a separate branch within the repository

Another alternative is to create a new branch for version 2.x with the same repository used for version 1.x. This approach is the most complex but offers the tightest possible integration between the two versions going forward.

The complexity arises from the number of branches that result. The main branch for version 1.x already has release, development, and feature branches, and the new main branch for version 2.x will require the same set of branches for its own use. Furthermore, a single repository can only ever have one branch named *default*, so if two release branches are needed the workflow must be revised to use something other than *default* as the release branch.

On the other hand, this approach affords the tightest possible integration between the two versions because revisions from one can be grafted onto the other. It might be the preferred choice for a simple project without too many revisions, but is probably not a good choice for larger or more complex projects.

How to decide

There is no "correct" or "best" choice here. Each situation is likely to be unique. When making a decision about which approach to take, each developer or development team needs to weigh the factors in play at the time.

One question to consider when thinking about creating a clone for a situation like this is, at what point does ancient history become irrelevant? If you create a brand new repository for version 2.0, does giving up the history of v1.x really matter? After all, that history is still in the version 1.x repository. If dragging all the baggage of the v1.x revision history over to v2.0 is not important, then a brand new repository may be the better choice.

Copyright 2016, Rick Borup

Exercise 9

There are no structured exercises for this lesson. The Exercise09 folder contains a repository from a previous exercise. You can use it as a starting point if you want to experiment with one or more of the approaches to maintaining multiple release version discussed here.

Copyright 2016, Rick Borup

Lesson 10 – Team development workflows

Lessons 10 is about team development workflows, exploring different ways teams can choose to collaborate on a project. These are "big picture" decisions and are not directly related to branching and merging within an individual repository. Within whatever constraints may be imposed by other factors, each team can adopt the "little picture" branching and merging scheme that works best for them within their repositories.

Remember that in a distributed version control system, each developer has his or her own local copy of the repository. Team collaboration works best if all the developers working on the same project use the same branching and merging scheme within their local repository for that project. However, even with that discipline in place, the process of exchanging revisions with other developers can create anonymous branches and require developers to merge and commit in order to stay in sync with one another.

There are essentially three workflows for team development with a DVCS – peer to peer, centralized, and integrator.

Peer to peer

Peer to peer is the simplest of the three team workflows. In a peer to peer workflow, team members exchange revisions directly with one another, either by pushing and pulling among each other's local repositories or by exporting and importing patches exchanged via email or some other mechanism. No central repository is involved. Because of the absence of any central authority, peer to peer workflows are sometimes referred to as creative anarchy.

Peer to peer collaboration is easy to implement and can work well for a small team, but it may not necessarily work well for a larger team. Sharing revisions directly with each of the other team members can become burdensome as the number of developers grows. I'm not aware of any statistics on how many teams use peer to peer, but my guess is that it's not as common as the other two methods.

There is no exercise in this session for peer to peer workflow. It works very much like the local clones workflow described in lesson 2, except with two or more developers each working from their own local clone.

Centralized

Outside of open source projects, which tend to favor the integrator workflow described later, the centralized workflow is perhaps the one most commonly used for team development. In a centralized workflow, there is a shared repository in a central location to which all team members have access. Instead of exchanging revisions directly with one another as in peer to peer, developers interact with one another indirectly by pushing changes to and pulling changes from the central repository.

Copyright 2016, Rick Borup

The central repository is a bare repository – it has no working copy, because nobody does any development work on the machine on which it resides, or at least not within the folder structure that contains the shared repository.

The central repository can be located on any resource to which all team members have access. It could be a server on a local area network that's accessible through the file system, or it might be on a remote server accessible over the Internet with ssh or http/https. The workflow is the same in either case – only the mechanics of access differ.

The basic workflow when using a centralized workflow is as follows:

- Each developer begins with a clone of the central, shared repository. Creating the clone established a link between the developer's local repository and the origin from which it was cloned. The first developer on a new project might be the one to create the central repo, in which case that person does not need to create a clone.

- If two or more team members are collaborating on the same feature branch, each one should check to see if another member has pushed any revisions to that branch on the shared repository before beginning to make changes to the files in their local working copy. If any incoming changes are detected, the developer pulls them into his or her own local repository and updates the working copy. If a merge conflict is detected, it must be resolved and the result of the merge committed to the local repo before proceeding.

- At some point after working on a feature locally, the developer decides it's time to push those changes to the central repository in order to share them with the rest of the team. Here the developer must decide which branch or branches to push, remembering that in Mercurial the default to push all. If two team members are collaborating on a feature while other team members work on other features in other branches, they will most likely want to push only the branch they are working on.

- If a developer forgets to check for incoming changes before making changes to her local working copy, or if another developer has pushed revisions to the same branch after the first developer pulled and merged but before she tries to push her changes back up, Mercurial responds with a "Push would create a new head" message. If this happens, the developer who's trying to push needs to again pull, merge, and commit to their local repository before pushing to the remote.

Integrator

The integrator workflow is similar to the centralized workflow in that revisions from all developers working on a project ultimately end up in a central repository. The difference is that in the integrator workflow one person or one group of people is designated as the integrator of all modifications with authority to review and approve or reject them.

The integrator workflow is commonly used in open source development, where commercial services like Bitbucket and GitHub provide a mechanism for developers to send pull requests to the integrator, aka the maintainer, of the central repository. In this workflow, developers contributing code to the project do not push their changes directly to the central repository. Instead, they request the integrator to pull their changes into the central repo to become part of the product. The integrator has the final say as to whether a modification or enhancement makes its way into the code base.

The integrator workflow could be implemented by a team without using a commercial service simply by implementing access restrictions to the central shared repository. Services like GitHub and Bitbucket, however, not only host the central repository but also provide additional features to facilitate the collaboration between developers, reviewers, and the project's owner(s).

From the developer's point of view, the workflow begins by cloning or forking the central repository. The developer can then proceed to work on bug fixes, make changes, and/or add new features while working on their copy of the code. When ready, the developer creates a pull request to notify the integrator that modifications are ready for code review.

From the integrator's point of view, the workflow begins when a pull request is received. This triggers the code review process, during which the integrator makes an initial assessment of the proposed modifications. If they look good, the integrator can approve the request and pull the modifications into a branch of the central repository for testing before integrating them into the stable code base for release. During code review and testing, the integrator may need to communicate with the originator to request further revisions or refinements of the code. The integrator can also reject the pull request outright if the modifications are flawed or simply not wanted in the first place.

There is no exercise in this session for the integrator workflow. Anyone interested in details is encouraged to check out the documentation for Bitbucket (Mercurial or Git) and GitHub (Git only) to see how this workflow is implemented by those services.

Exercise 10

Exercise 10 focuses on the centralized workflow. In this exercise you'll act as both developers on a two-person development team. Both of these people work for a small software company named Megasoft. Carol Coder is the company founder and lead developer. She created the first code for the product and has worked on it exclusively on her local machine since development first began.

At some point, Carol realizes there's a need for a second developer. After an extensive search, she hires Bob Beta as employee number two. Bob's job is to work with Carol on the development of the next big feature for Megasoft's product.

To facilitate their working together, Megasoft adopts a centralized workflow using a shared central repository. In real life this centralized repo would be located on a file server or other resource accessible to both Carol and Bob, but for purposes of this exercise it's

located in the HgCentral\Exercise10 folder on your machine. The root Exercise10 folder has two subfolders, Carol and Bob, enabling you to work as each one in turn.

Every commit carries the name and email address of the person who made it. This is important information in a team development environment because it lets each team member know who made commits other than their own.

To distinguish Carol's commits from Bob's in this exercise, Carol's local repository needs to be configured with her name and email address and, when the time comes, Bob's needs to be configured with his. In TortoiseHg, click File | Settings on the main menu to open the Settings dialog. Select the page for the local repository settings, which is titled *Carol repository settings*, as illustrated in step (1) in **Figure 79**. In the left-hand column, select the Commit options (2) and then enter Carol's name and email address as shown in step (3). Click the *Save* button to save these changes. The changes are recorded in the hgrc configuration file for this local repository and do not affect your global settings, so you don't need to worry about this affecting anything else on your machine.

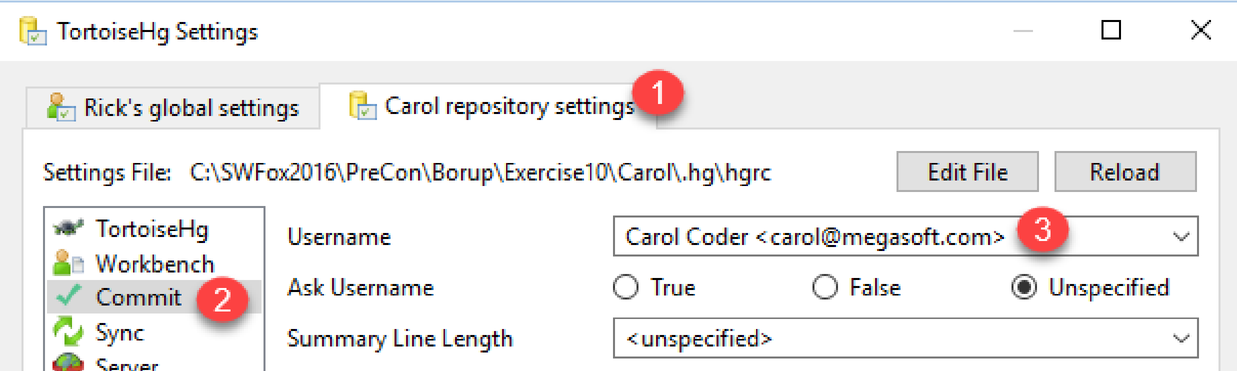


Figure 79. Repository-specific settings override the corresponding global settings.

Because she was the only developer until Bob was hired, Carol's machine already has the latest code. The central repository in the HgCentral\Exercise10 folder has been initialized, but nothing has ever been pushed to it so it's empty. To make the current state of her code available to Bob, Carol pushes all the branches from her local repository to the shared remote. Her local repository, as well as the remote, both look as shown in **Figure 80**.

Graph	Rev	Branch	Description	Author
	8+	default	★ Working Directory ★	Carol Coder
	8	default	default tip **Added tag Release 1.0 for changeset 2c24bafbe41c**	Carol Coder
	7	default	Release 1.0 Merge develop into default	Carol Coder
	6	develop	develop updated readme for release 1.0	Carol Coder
	5	develop	Merge feature_1 into develop	Carol Coder
	4	feature_1	feature_1 finish feature 1	Carol Coder
	3	feature_1	work on feature 1	Carol Coder
	2	feature_1	begin work on feature 1	Carol Coder
	1	develop	added readme file	Carol Coder
	0	default	initial commit	Carol Coder

Figure 80. Carol's local repository shows all the commits have been made by her.

Notice the Author column included in Figure 80. All the commits at this point are clearly identified as having been made by Carol. That's why it was important for Carol to put her name and email address in her settings for this project before making any commits to her repository.

The next step in this exercise, and the first step for you to perform, is to switch hats and act as Bob. To begin collaborating on this project with Carol, Bob first needs to create a clone from the central repository. Navigate to the Exercise10\Bob folder in File Explorer and create the clone, using what you learned in lesson 2.[5] Note the Bob is cloning from the central repository, not directly from Carol's local repository (see **Figure 81**).

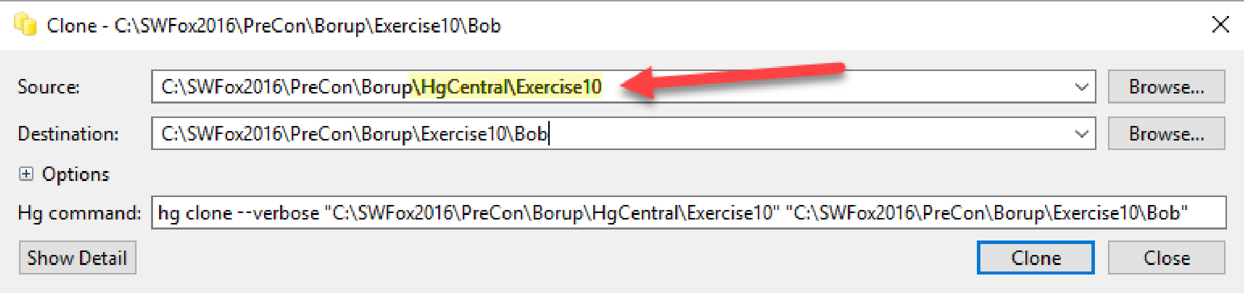


Figure 81. Bob clones the project from the central shared repository.

Bob now has a local clone of Carol's work from the central repository. As a brand new repository, this clone does not yet have any repo-level settings of its own, so Mercurial will use the global settings on your computer. For this exercise we want Bob's commits to be identified in the revision history with his name instead of whatever the global settings are. Refer back to Figures X and use the same steps to update Bob's repository settings with username to *Bob Beta* bob@megasoft.com.

[5] Hint: right-click on folder Bob, select TortoiseHg | Clone… from the popup menu, use the Browse button in the Clone dialog to select HgCentral\Exercise10 as the source, and click the Clone button.

If you created Bob's clone from the pop-up menu as described in the hint above, TortoiseHg automatically opens it. If not, open it in TortoiseHg yourself. Bob's local repository should look like **Figure 82**. Note that TortoiseHg features a tabbed interface and can have two or more repositories open at the same time. As expected, Bob's local repository is identical to Carol's because it's a clone.

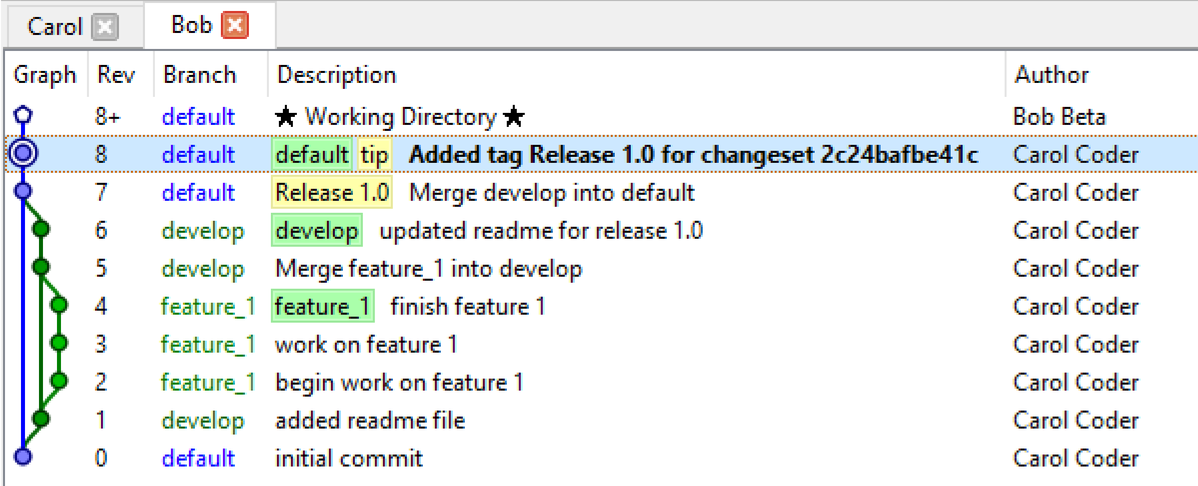


Figure 82. After cloning, Bob's local repository is identical to Carol's.

Bob and Carol now begin to work on feature 2 for this product. Both will be working in a feature branch named feature_2. If they communicate with one another and let each other know when they've pushed commits to the central repository, each of them will know to look for incoming changes before beginning work on their own local copy.[6]

If they don't communicate, Carol may have pushed a commit that Bob doesn't know about, or vice versa. In that case, whoever tries to push next runs into the "Push creates a new remote head" situation. Let's see how that might happen.

Carol updates her working copy to the head of the *develop* branch at revision 6, begins work on feature 2, commits it to the new *feature_2* branch, and pushes it to the central repository. Since the commit involves changes on only one branch, she uses *Push all*. Mercurial alerts Carol that a new branch will be created in the remote, which she confirms is OK. Carol's repo now looks like **Figure 83**.

[6] In a team environment, checking for incoming changes before beginning to make your own modifications is a best practice in any event, but it doesn't always happen.

Carol | Bob

Graph	Rev	Branch	Description	Author
	9+	feature_2	★ Working Directory ★	Carol Coder
	9	feature_2	feature_2 tip **begin work on feature 2**	Carol Coder
	8	default	default Added tag Release 1.0 for changeset 2c24bfabe41c	Carol Coder
	7	default	Release 1.0 Merge develop into default	Carol Coder
	6	develop	develop updated readme for release 1.0	Carol Coder
	5	develop	Merge feature_1 into develop	Carol Coder
	4	feature_1	feature_1 finish feature 1	Carol Coder
	3	feature_1	work on feature 1	Carol Coder
	2	feature_1	begin work on feature 1	Carol Coder
	1	develop	added readme file	Carol Coder
	0	default	initial commit	Carol Coder

Figure 83. Carol begins work on feature 2.

Bob, being new, forgets he's supposed to check and see if Carol already pushed any feature 2 changes to the central repo before beginning his own work. He goes ahead and starts writing code for feature 2 himself. Like Carol, he first updates his working copy to the head of the *develop* branch at revision 6. Unlike Carol, he makes a different set of modifications than Carol made. He commits them to a new *feature_2* branch in his local repository and then tries to push the revision to the central repo. The push is aborted and Bob sees the message shown at the top of **Figure 84**. An abort message along with a return code is also displayed at the bottom of the TortoiseHg window (not shown).

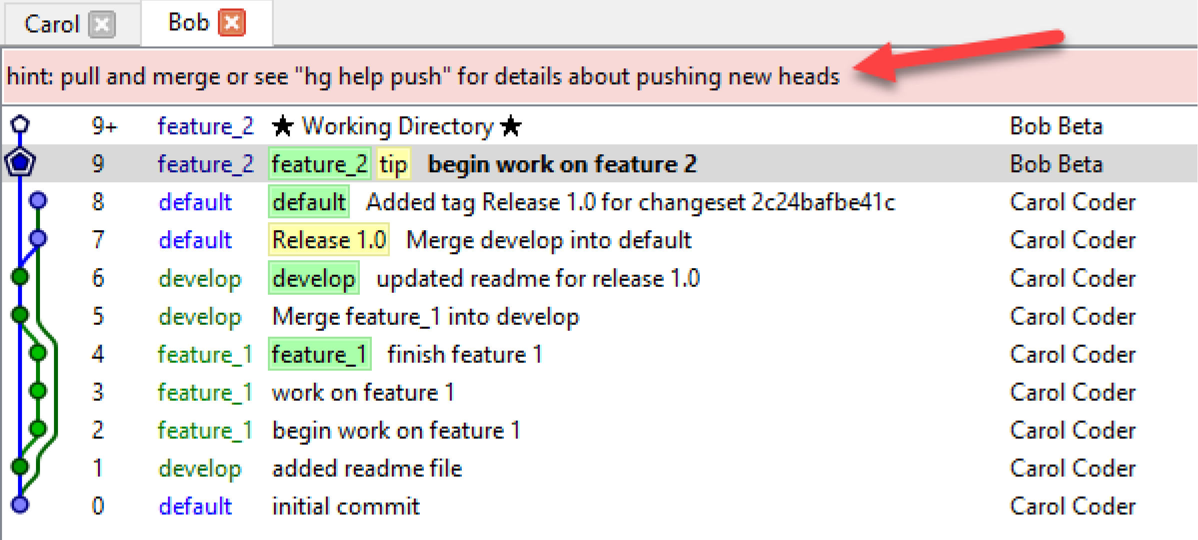


Figure 84. When Bob tries to push, he gets a message telling him he needs to first pull and merge.

The reason this push would create a new head in the remote repository is that both Carol and Bob have made modifications with revision 6 as the parent.

Copyright 2016, Rick Borup

In order to proceed, Bob must abandon the push and check for incoming changes from the remote repository. When he does so, he sees that there is one from Carol (see **Figure 85**).

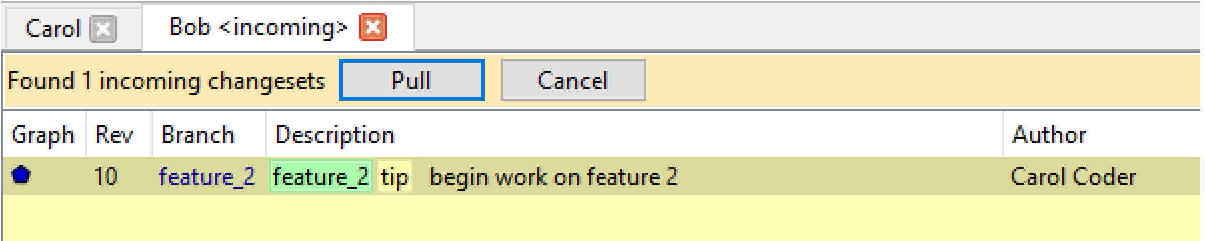


Figure 85. Bob runs the *incoming* command and finds Carol has pushed a commit he doesn't yet have.

Bob pulls the incoming changeset into his local repository. His local repository now has two heads on the feature_2 branch – one from his modification at revision 9 and the other from Carol's medication at revision 10 (see **Figure 86**). Remember that revision numbers are scoped to the local repository; revision 10 in Bob's local repo is revision 9 in Carol's. Mercurial uses the global changeset ID assigned to every commit to uniquely identify them across repositories, even if they end up with different revision numbers.

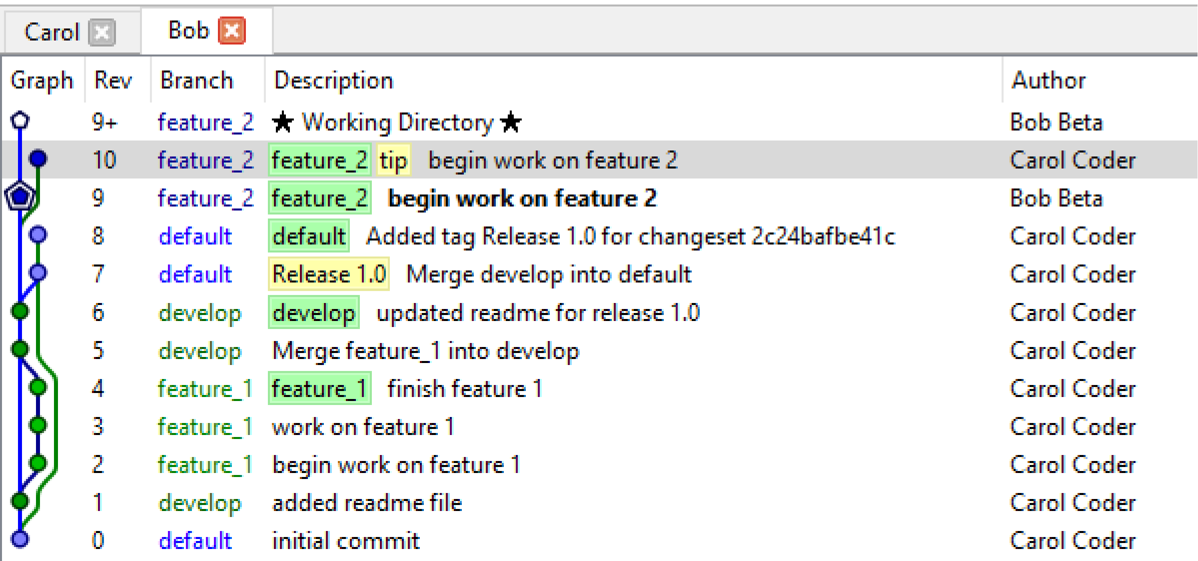


Figure 86. Bob pulls Carol's changes from the central shared repository.

Resolving a condition with two heads in a branch is a straightforward process of merging one into the other. Bob merges Carol's changes from revision 10 into his changes at revision 9. Because of the artificial nature of the sample source code file my.prg, this creates a merge conflict, but that won't always be the case. Bob resolves the conflict and commits the merged file to his local repo, creating revision 11 as shown in **Figure 87**.

Copyright 2016, Rick Borup

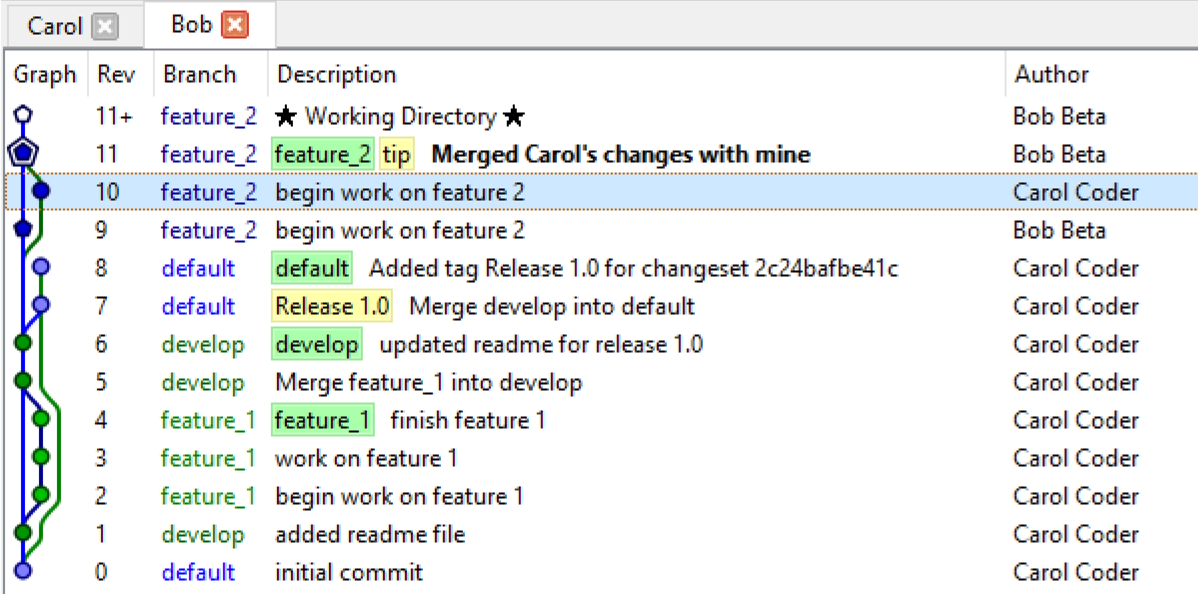


Figure 87. Bob merges Carol's changes with his own work.

Bob is now ready to push his feature 2 changes to the remote repository. This will no longer create two heads in the remote, because Carol's changes are now part of Bob's revision history. Bob checks outgoing and finds there are two changesets to be pushed (**Figure 88**).

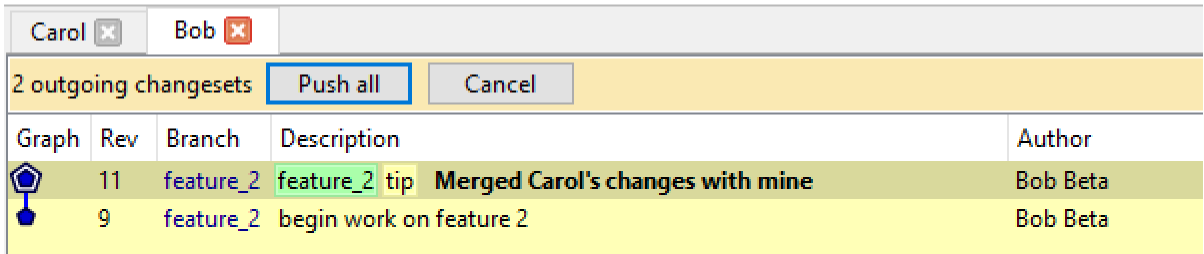


Figure 88. Bob runs the outgoing command to see what will get pushed.

As Carol did, Bob can use *Push all* here because no changes to other branches involved.

Bob emails Carol and lets her know he merged her changes and pushed some of his own. Carol checks for incoming changes from the remote and finds the two commits Bob pushed (**Figure 89**).

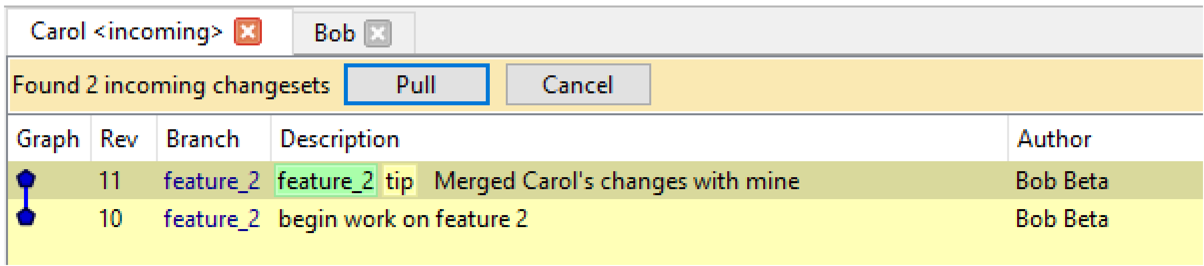
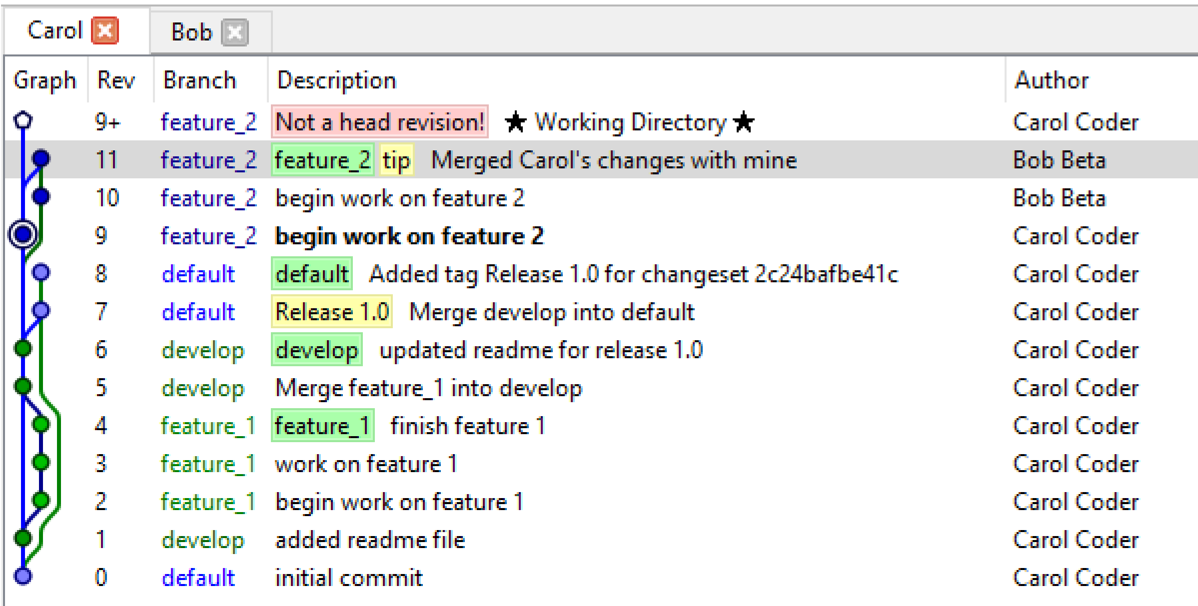


Figure 89. Carol checks incoming to see what Bob pushed.

After pulling those changes from the remote, Carol's is local repository is up to date but her working copy, which is still based on her revision 9, is not a head revision (see **Figure 90**).

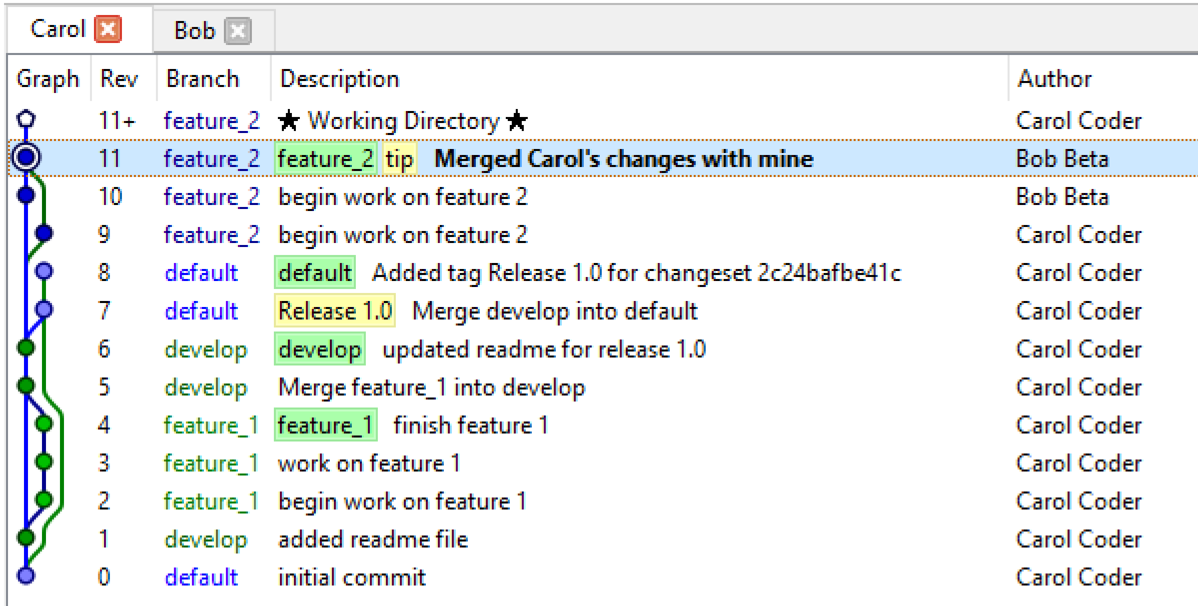


Carol | Bob

Graph	Rev	Branch	Description	Author
	9+	feature_2	Not a head revision! ★ Working Directory ★	Carol Coder
	11	feature_2	feature_2 tip Merged Carol's changes with mine	Bob Beta
	10	feature_2	begin work on feature 2	Bob Beta
	9	feature_2	**begin work on feature 2**	Carol Coder
	8	default	default Added tag Release 1.0 for changeset 2c24bafbe41c	Carol Coder
	7	default	Release 1.0 Merge develop into default	Carol Coder
	6	develop	develop updated readme for release 1.0	Carol Coder
	5	develop	Merge feature_1 into develop	Carol Coder
	4	feature_1	feature_1 finish feature 1	Carol Coder
	3	feature_1	work on feature 1	Carol Coder
	2	feature_1	begin work on feature 1	Carol Coder
	1	develop	added readme file	Carol Coder
	0	default	initial commit	Carol Coder

Figure 90. Carol pulls Bob's changes from the central shared repository into her local repository.

To get Bob's changes into her own work, Carol simply needs to update her working copy to the new head of the *feature_2* branch at revision 11. Because Bob already merged Carol's changes, this does not create any conflicts for Carol (assuming her working copy is clean). Carol's working copy and local repository are now in sync with Bob's, as is the shared remote repository (see **Figure 91**).



Carol | Bob

Graph	Rev	Branch	Description	Author
	11+	feature_2	★ Working Directory ★	Carol Coder
	11	feature_2	feature_2 tip **Merged Carol's changes with mine**	Bob Beta
	10	feature_2	begin work on feature 2	Bob Beta
	9	feature_2	begin work on feature 2	Carol Coder
	8	default	default Added tag Release 1.0 for changeset 2c24bafbe41c	Carol Coder
	7	default	Release 1.0 Merge develop into default	Carol Coder
	6	develop	develop updated readme for release 1.0	Carol Coder
	5	develop	Merge feature_1 into develop	Carol Coder
	4	feature_1	feature_1 finish feature 1	Carol Coder
	3	feature_1	work on feature 1	Carol Coder
	2	feature_1	begin work on feature 1	Carol Coder
	1	develop	added readme file	Carol Coder
	0	default	initial commit	Carol Coder

Figure 91. Carol updates her working copy to the new head of the feature_2 branch.

A centralized workflow like this can be extended to more than two developers working concurrently on two or more feature branches. Changes are not usually pushed to the central repository until they're ready to be shared with other developers working on the same feature. Before making any changes of their own, each developer should remember to check and see if there are new revisions to their feature from other developers.

On any team, one developer is probably going to be tasked with creating the build. In a large organization there might even be a separate build team, and perhaps even a QA team ahead of the build team. Regardless, whoever is doing the QA and the build-for-release pulls the release-ready code from the central repository.

Copyright 2016, Rick Borup

Lesson 11 – The Hg Flow workflow

A workflow called Git Flow has become increasingly popular among developers using Git. This workflow stemmed from an approach to branching and merging first introduced, to the best of my knowledge, by Vincent Driessen in a blog post entitled *A successful Git branching model*, published in 2010.[7] Today, many references to the Git Flow model can be found across the Internet.

Atlassian, the company that owns Bitbucket and created the SourceTree program, has integrated the Git Flow model into SourceTree, along with a corresponding workflow called Hg Flow for developers using Mercurial. Exercises 4 through 8 in this paper are area based on a workflow similar to Hg Flow, so although the full Hg Flow model introduced here is a bit more complicated, it should already look familiar.

This lesson demonstrates how to use SourceTree and the Hg Flow model of branching and merging. Although Hg Flow could be implemented manually by creating the branches and doing the merges manually, as we've been doing in all the exercises up to now, it's much easier to implement Hg Flow by using the tools that are built into SourceTree.

Figure 92 is a snapshot of a revision history created using Hg Flow, as seen in SourceTree. This is not the same revision history we'll develop step-by-step in exercise 11, but it will help to give you an idea of what Hg Flow looks like.

In addition to the *develop* and *feature* branches we've been using in earlier exercises, Hg Flow uses distinct *release* branches. As you look at the commits in Figure 92, note the naming scheme Hg Flow uses – feature branches are named *feature/<description>* and release branches are named *release/<description>*. The development branch is just called develop, but a hotfix branch is named *hotfix/<description>*.

Also note that many of the entries in the *Description* column are prefixed with *flow:*. These are the commits generated by Hg Flow, which does a lot of the work for you.

Exercise 11 begins by creating a new repository and walks you through the steps involved to implement and use Hg Flow to develop and release a new feature. We will be using SourceTree and its built-in Hg Flow tools for this exercise.

[7] http://nvie.com/posts/a-successful-git-branching-model/

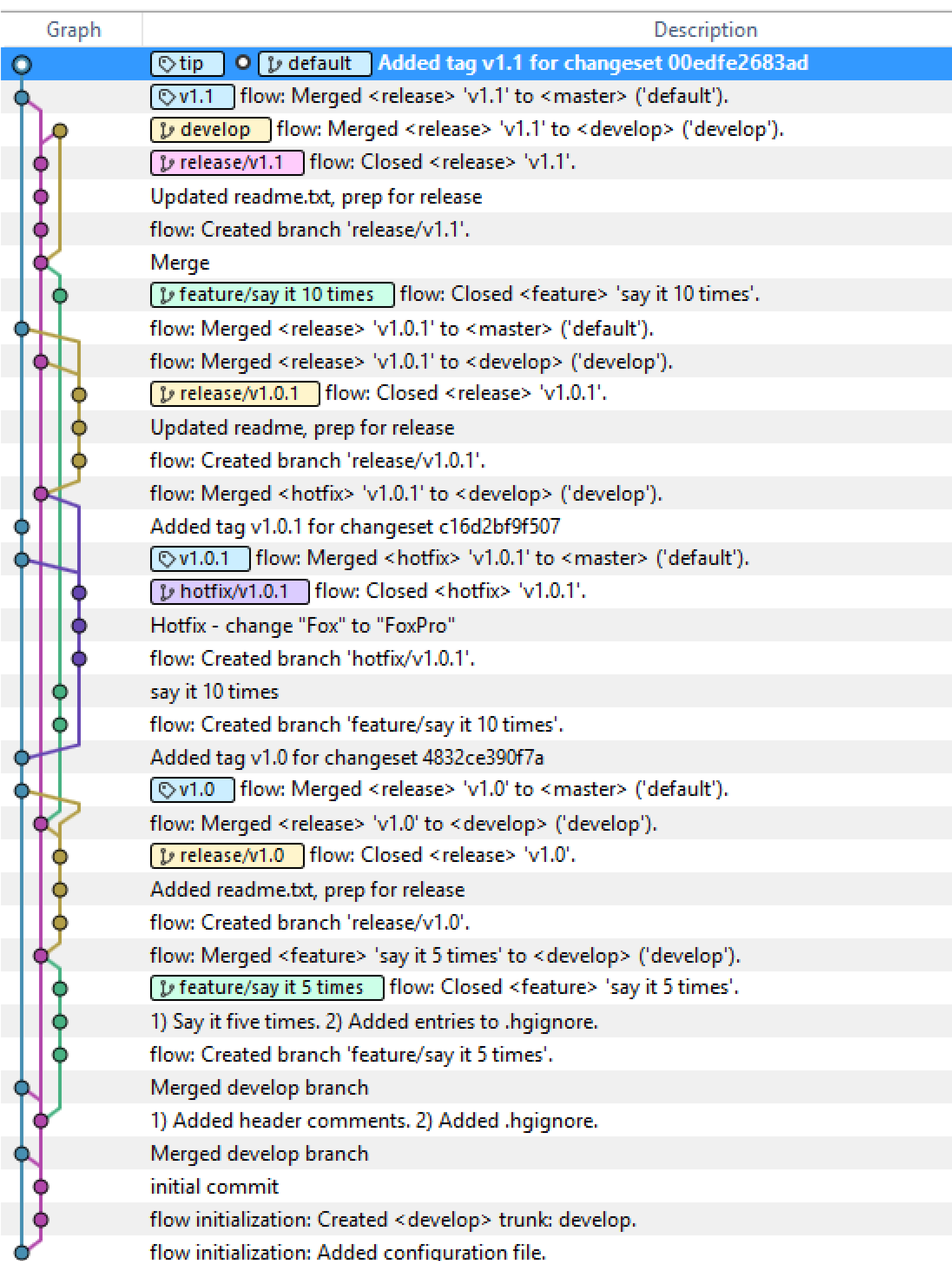


Figure 92. A sample revision history using Hg Flow.

Exercise 11

Begin by initializing a new repository in your Exercise11 folder, which starts out with a couple of files – the my.prg source code file and the .hgignore file – but which does not yet have a repository. To do this from SourceTree, click the *Clone / New* button on the toolbar

and select the *Create New Repository* page (see **Figure 93**). Select Mercurial as the type of repository (1) and insert the path to the Exercise11 folder as the destination (2). Leave the default of *Exercise11* in the Name field, and choose the folder (if other than *[Root]*) where you want this repo to appear in the SourceTree tree-view. Click *Create* to create the repository.

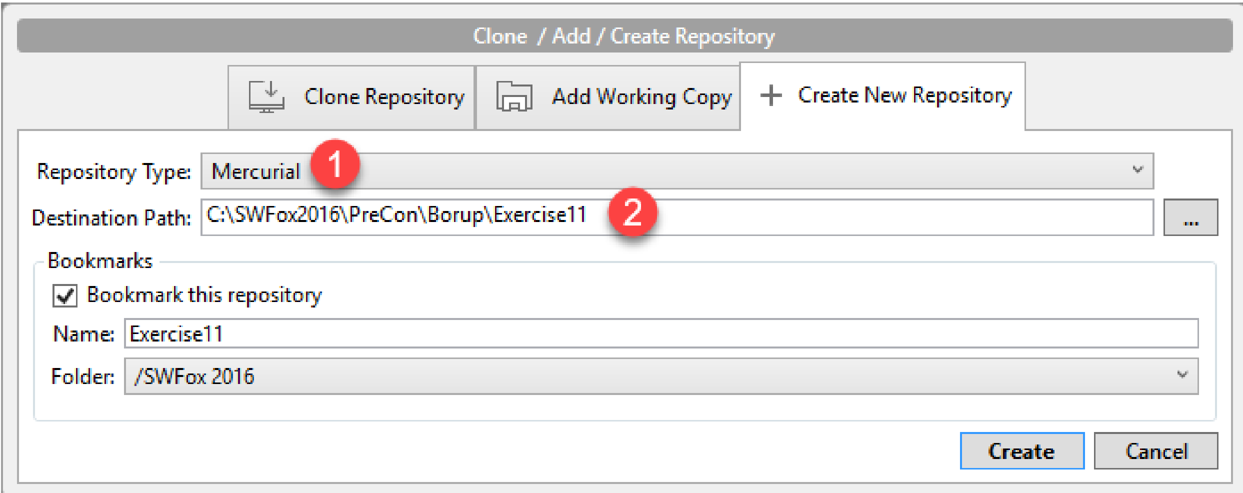


Figure 93. A new Mercurial repository is created in SourceTree.

To create the initial state of the repository, select the File Status page from the tabs along the bottom of the SourceTree window. Select both files, right-click and choose Add. Mark the check box next to each file to include it in the upcoming commit (see **Figure 94**). In the Commit pane in the lower half of the window, enter *initial commit* as the commit message and click the Commit button. This commit occurs on the *default* branch, which at this time is the only branch in the repository.

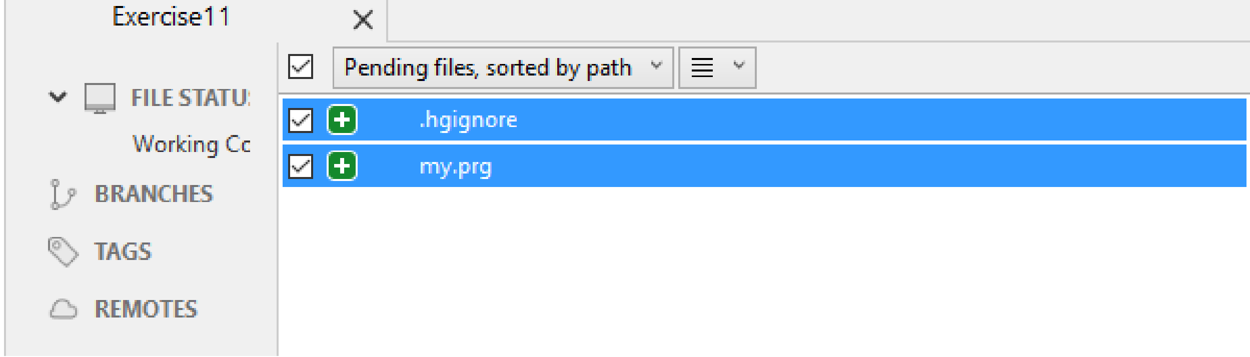


Figure 94. The File Status pane in SourceTree shows the files pending commit.

Before beginning to make any modifications to the source code, we want to initialize this repository for Hg Flow. To do this, click the *Hg Flow* button in the upper right portion of the SourceTree toolbar at the top of the window. This opens the *Initialise repository for Hg Flow* dialog shown in **Figure 95**.

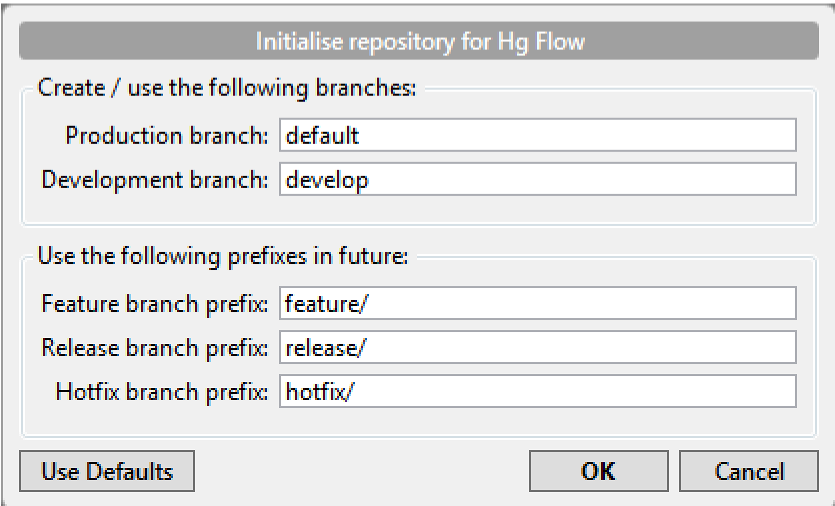


Figure 95. SourceTree initializes Hg Flow for this repository.

For this exercise, accept the default values for the branch names and branch prefixes. Later on, if you decide to adopt Hg Flow for you own use, you can use a different naming scheme, which you could specify in this dialog. Click *OK* to initialize Hg Flow on this repository.

Initializing a repository for Hg Flow applies changes to the repository. SourceTree gives you a chance to confirm you really want to do this (**Figure 96**).

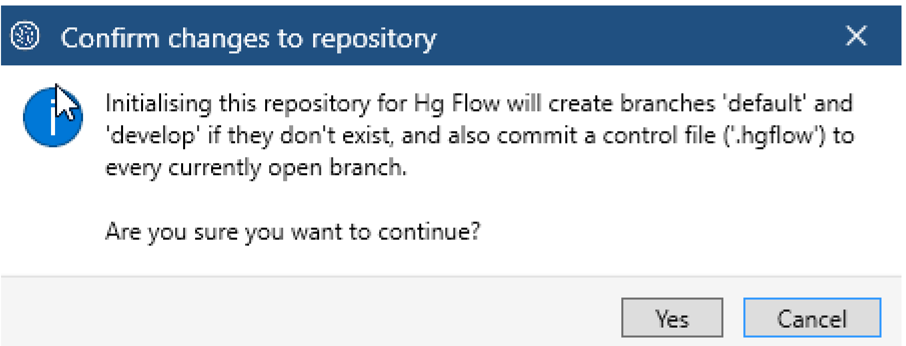


Figure 96. SourceTree gives you a chance to confirm what it's going to do when you initialize Hg Flow.

Click *Yes* to proceed. This process creates a commit, and revision history now appears as shown in **Figure 97**.

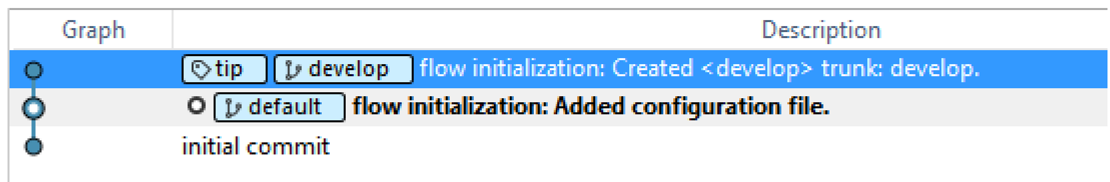


Figure 97. The revision history after initializing Hg Flow.

Note that the *develop* branch has been automatically created but *default* is the active branch at this point. You can tell because its description is bold and its icon in the graph has a heavier outline than the others.

We're now ready to begin development work. Our job is to add feature 1 and release version 1.1 of this product. When using Hg Flow, new features, new releases, and new hotfixes are initiated by clicking the Hg Flow button on the SourceTree toolbar. This brings up the *Choose Next Flow Action* dialog shown in **Figure 98**.

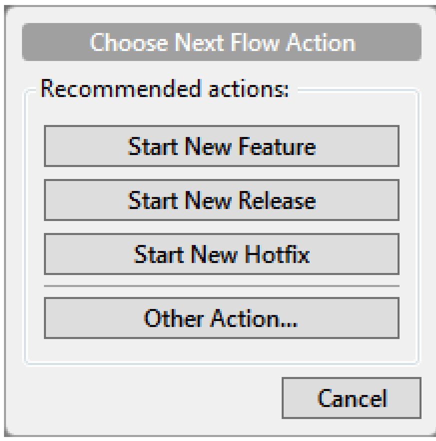


Figure 98. The *Choose Next Flow Action* dialog guides lets you control the steps when using Hg Flow.

Click the *Start New Feature* button. The next dialog prompts for the feature name and lets you specify the branch from which it is to start. The default Hg Flow naming scheme for feature branches, which we selected earlier, is word *feature,* followed by a slash, followed by the name of the feature. To be consistent with earlier exercises, we'll call this feature *feature_1,* so in the repository, the branch for this new feature will be named *feature/feature_1*. The default starting point is the latest development branch. This is what we want, so leave that setting as is. Click *OK* to start the new feature (**Figure 99**).

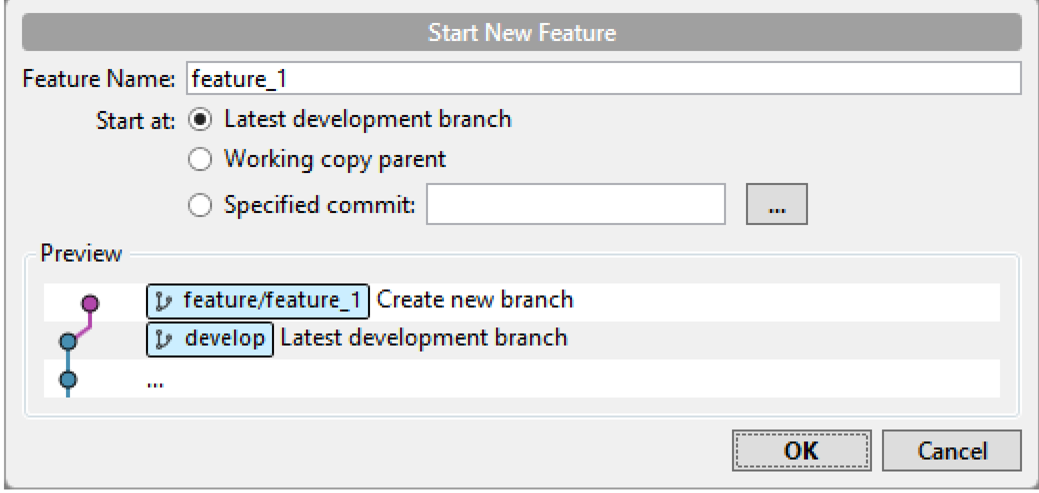


Figure 99. Start a new feature using Hg Flow's *Start New Feature* dialog.

The revision history now reveals that a new feature branch has been created off the develop branch, as shown in **Figure 100**. Note that the description of all the commits since the initial commit are prefixed with *flow:* indicating they were performed by Hg Flow. We did not have to create any of these branches or perform any of these commits manually because Hg Flow did it for us.

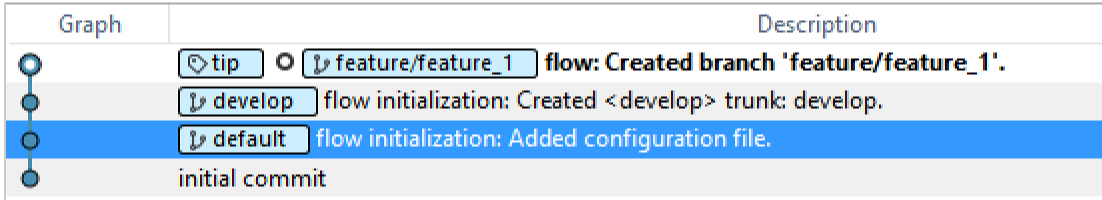


Figure 100. Hg Flow has created a new branch named *feature/feature_1*.

Note that the active branch is now the *feature/feature_1* branch. We're now ready to begin making modifications to the source code for the new feature.

There are a couple of ways to edit the source code. If you select a file in SourceTree's File Status pane and choose *Open* from the pop-up menu, SourceTree tells Windows to open the file in the default editor for its file type. Since my.prg is a Visual FoxPro file, Windows will launch a new instance of VFP (even if one is already running) and open the file there. Unlike TortoiseHg, there is no way to tell SourceTree to open it in Notepad or another simple file editor. If that's what you want to do, run your preferred file editor and open the file from there.

After making a modification, return to the File Status pane in SourceTree. Select Pending Files to display files in modified status pending commit and mark the check box for the file(s) you want to commit (see **Figure 101**). Enter a commit message in the lower pane (not shown), and click the *Commit* button.

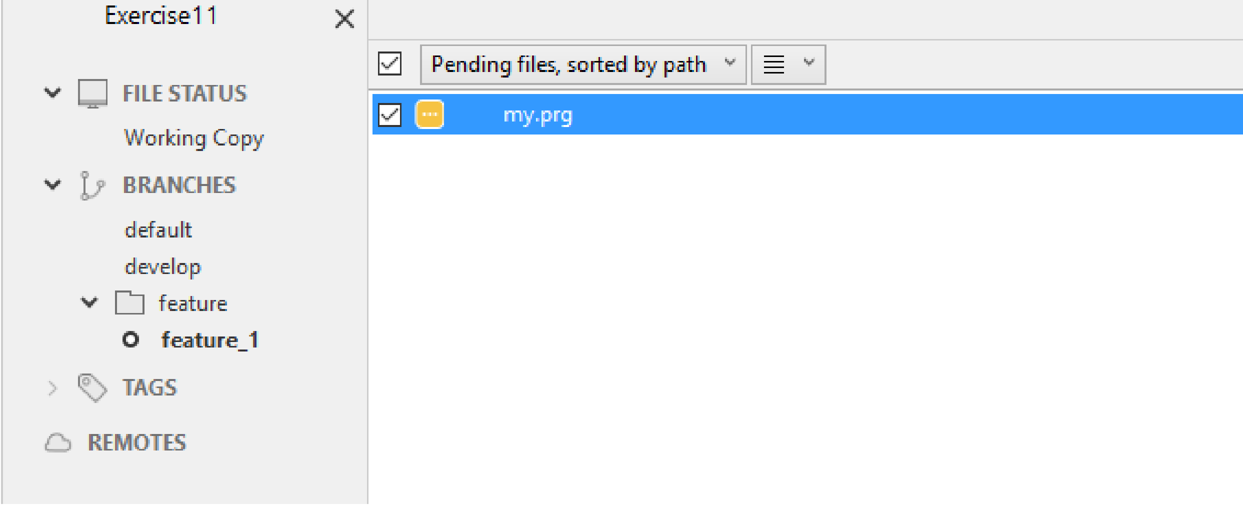


Figure 101. SourceTree shows the pending file after starting modifications for feature 1.

After making a few modifications while working on feature 1 and committing them to the feature branch, the revision history looks like **Figure 102**.

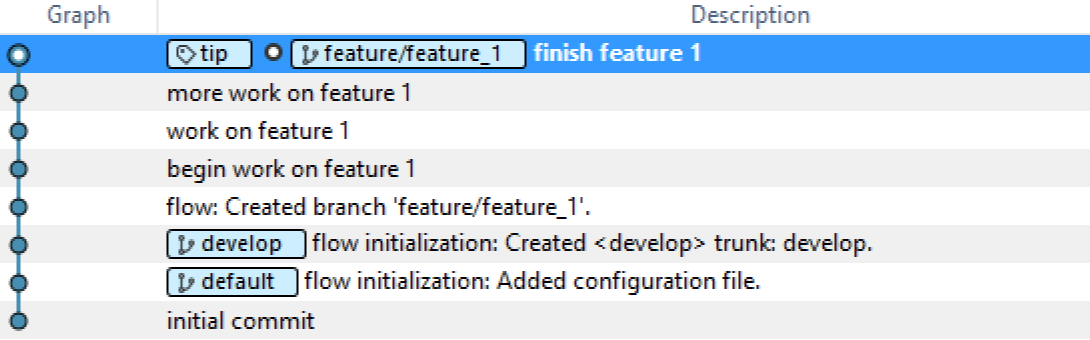


Figure 102. Several commits have been made to the *feature/feature_1* branch while working on feature 1.

Having finished development work on feature 1, we're now ready to finish the *feature/feature_1* branch in the repository. Hg Flow is again able to do a lot of the work for us. Click the *Hg Flow* button on the SourceTree toolbar, then click *Finish Feature* as shown in **Figure 103**.

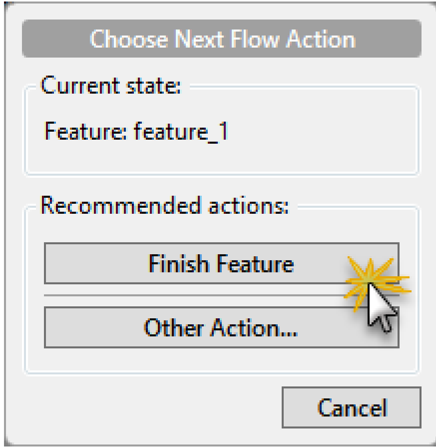


Figure 103. Use the Finish Feature flow action to complete the feature.

Hg Flow displays a preview of what's about to happen, showing that it will merge the *feature/feature_1* branch into the *develop* branch with the commit message shown. Click OK to proceed (**Figure 104**).

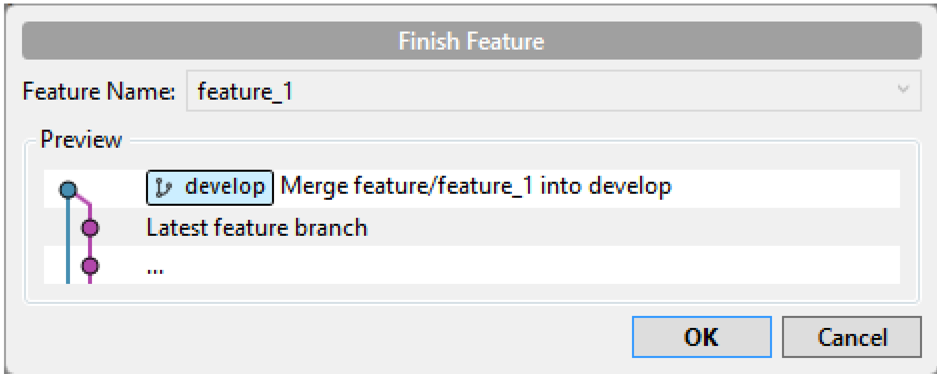


Figure 104. SourceTree shows what is about to happen when the feature is finished.

Hg Flow makes two commits for us. First of all, it closes the feature branch, using the close-before-merge paradigm we looked at in lesson 7. Then it merges the feature branch into the develop branch. Both commits have a meaningful commit message and both carry the *flow:* prefix indicating they were performed by Hg Flow (see **Figure 105**).

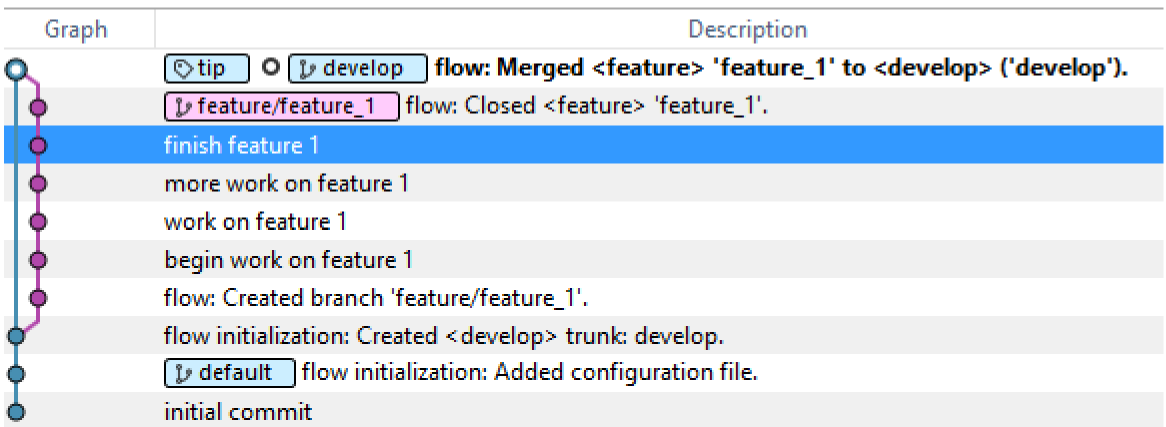


Figure 105. The revision history after finishing the feature in Hg Flow.

Note that the repository is now back on the *develop* branch. If at this point we are ready to release an update to the product, Hg Flow can help us do that, too.

Click the Hg Flow button to select the next action and click Start New Release (**Figure 106**).

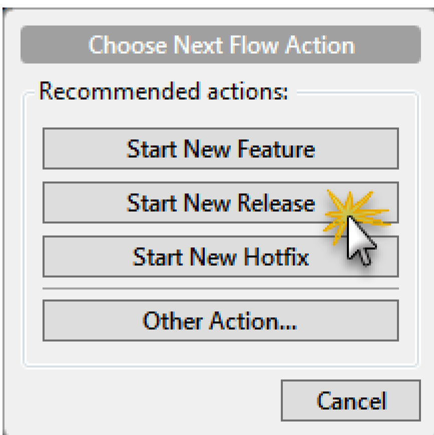


Figure 106. The next flow action is to start a new release for the feature that was just finished.

Unlike our earlier exercises, in which we merged develop into default for release, Hg Flow creates a new branch for each release. Give the release branch a name, for example “v1.1” as shown in **Figure 107**, and click OK to proceed.

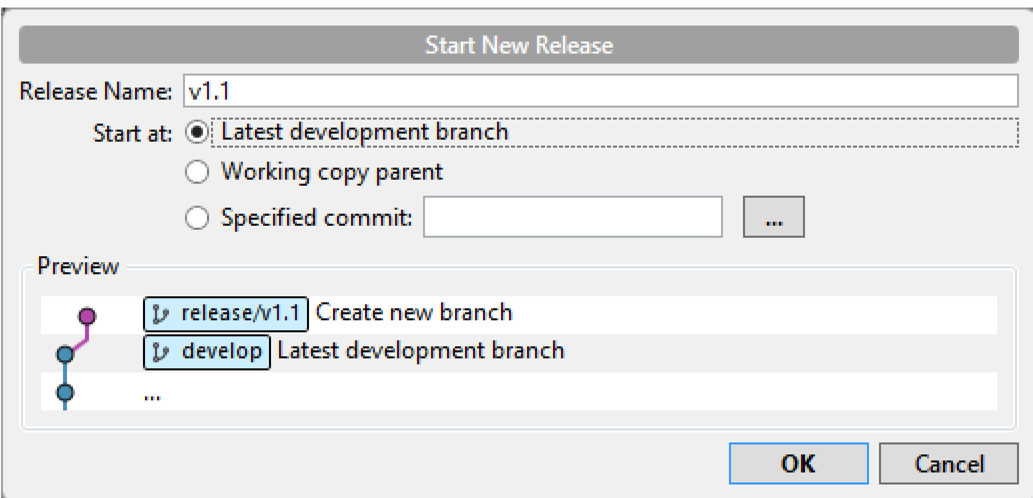


Figure 107. Give the new release a name in the *Start New Release* dialog.

The revision history now appears as shown in **Figure 108**.

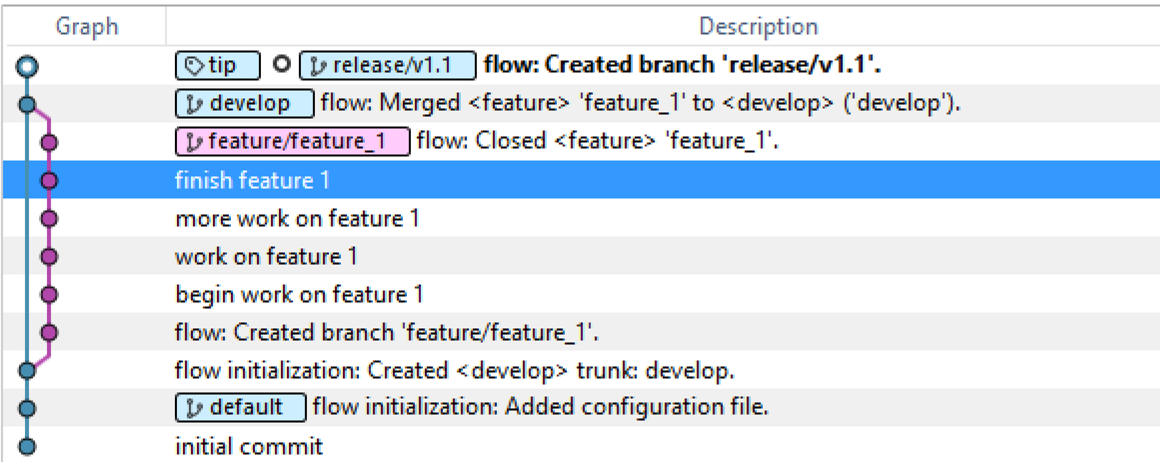


Figure 108. A new branch named release/v1.1 has been created.

The last step is to finish the release. Before doing so, this is a good time to do anything else that might be needed as part of the release, such as updating a readme file. One doesn't exist yet at this point, so we add one and commit it on the *release/v1.1* branch (**Figure 109**).

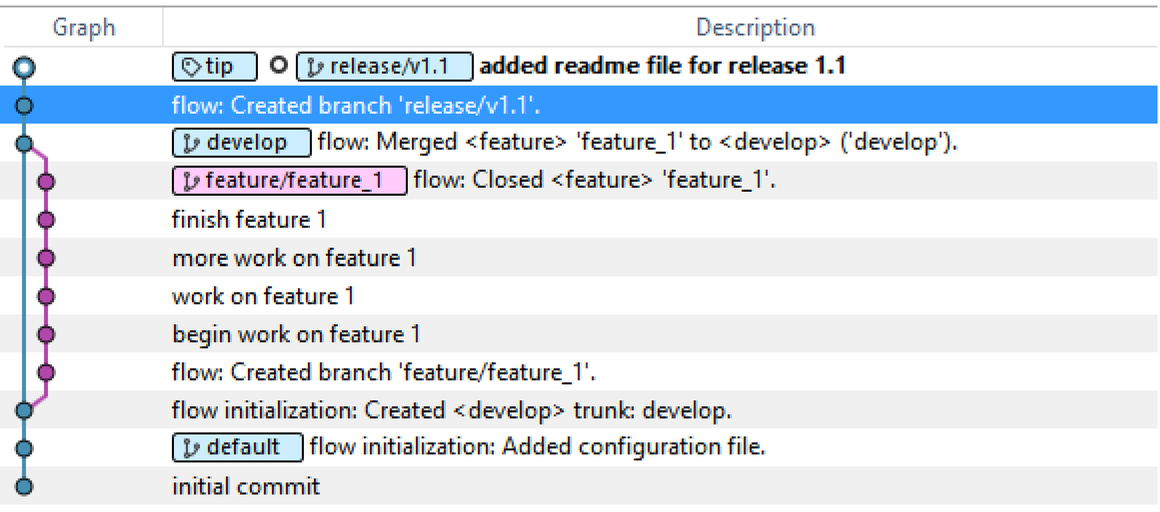


Figure 109. The readme file can be updated and other housekeeping done before finishing the release.

When everything is ready to go, click the Hg Flow button and select *Finish Release* (**Figure 110**).

Copyright 2016, Rick Borup

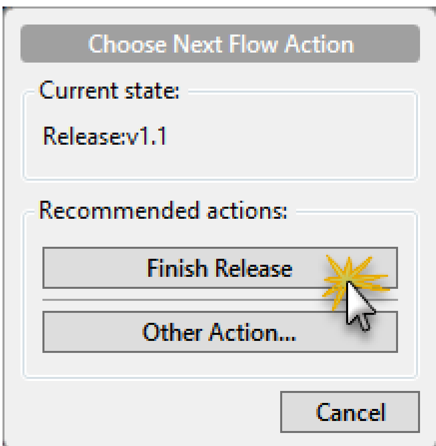


Figure 110. Use the Choose Next Flow Action dialog to finish the release.

The action is again previewed, as shown in **Figure 111**. Click OK to proceed.

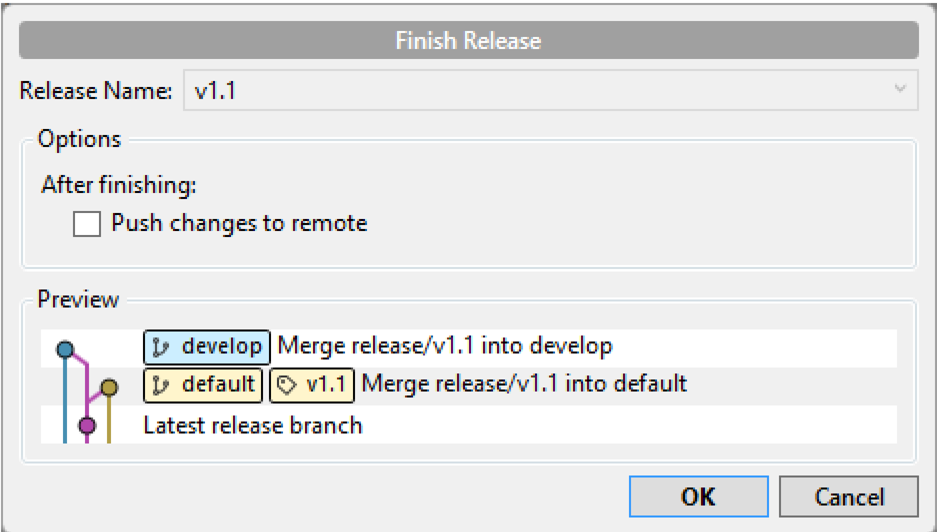


Figure 111. The *Finish Release* dialog previews what will happen.

Hg Flow again performs several steps for us. First, it closes the release branch. Next, it merges the release branch into *develop* and also into *default*. Finally, it adds a tag with the name of the release as the final commit on the *default* branch.

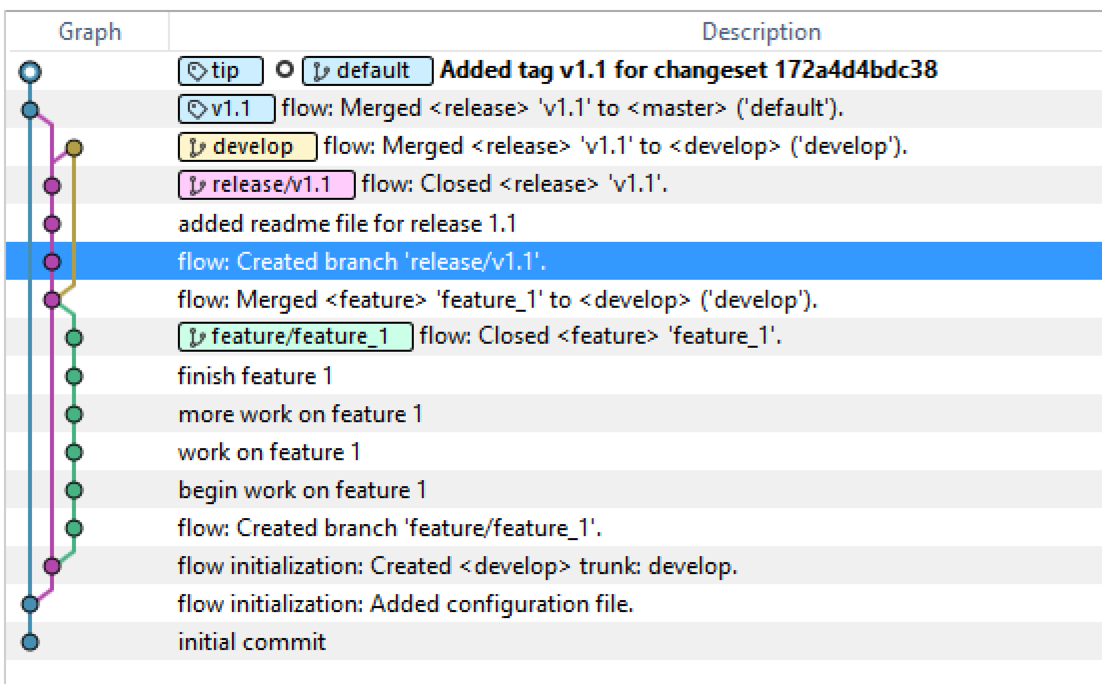


Figure 112. The new release has been finished and tagged, and the working copy is again on *default*.

Compare the end result from **Figure 112** in this exercise to the result in Figure 55 from a similar revision history in lesson 5. While SourceTree does not have columns for local revision numbers and branch names, you can see how the two workflows result in similar revision histories.

Lesson 12 - SourceTree, FoxBin2Prg, & Git and Hg Utilities

SourceTree

SourceTree is a desktop program that provides a visual interface to Git and Mercurial repositories. It was created by Atlassian, the company that also created and operates Bitbucket.org. SourceTree is free but requires you to set up an account with Atlassian (also free).

SourceTree can be used in place of or in addition to TortoiseHg to work visually with Mercurial repositories. The two are similar but the SourceTree interface is different enough to require a bit of a learning curve.

TortoiseHg works only with Mercurial repositories but has the advantage of having been around for a long time and therefore being used by many developers. SourceTree is something of the new kid on the block. It has the advantage of being able to work with both Git and Mercurial repositories. SourceTree also provides tools to facilitate using the Hg Flow workflow, which is covered in lesson 11.

Download and install SourceTree from https://www.sourcetreeapp.com/. Unlike TortoiseHg, SourceTree does not provide context-menu integration with Windows File Explorer, so you need to launch it in the conventional manner.[8]

The SourceTree interface is shown in **Figure 113**. Like TortoiseHg, SourceTree features a tabbed interface, so you can work with more than one repository at the same time. The tabs across the top are the repositories currently open in SourceTree. In Figure 113, both exercise 11 and exercise 12 are open, with exercise 12 the selected tab. The tabs across the bottom are where you select the view you want – File Status, Log / History, or Search.

Use the File | Open menu to open a repository in SourceTree. The leftmost column shows the repositories that have been bookmarked. Like TortoiseHg, you can create folders to organize your bookmarked repositories. Figure 113 shows the repositories bookmarked in the *SWFox 2016* folder.

The second column from the left displays information about the repository currently selected. The space to the right of that is where the desired pane is displayed. In Figure 113, the Log / History pane is shown, where you can see there has been only one commit to the repository for exercise 12.

Column widths and pane heights are resizable. Unlike TortoiseHg, SourceTree does not have columns for the local revision number or the branch name associated with each commit. Other than that, the revision history looks about the same in both programs.

[8] Shout-out here to the venerable SlickRun utility (https://bayden.com/SlickRun/), which saves me a ton of time every day and still works great on Windows 10.

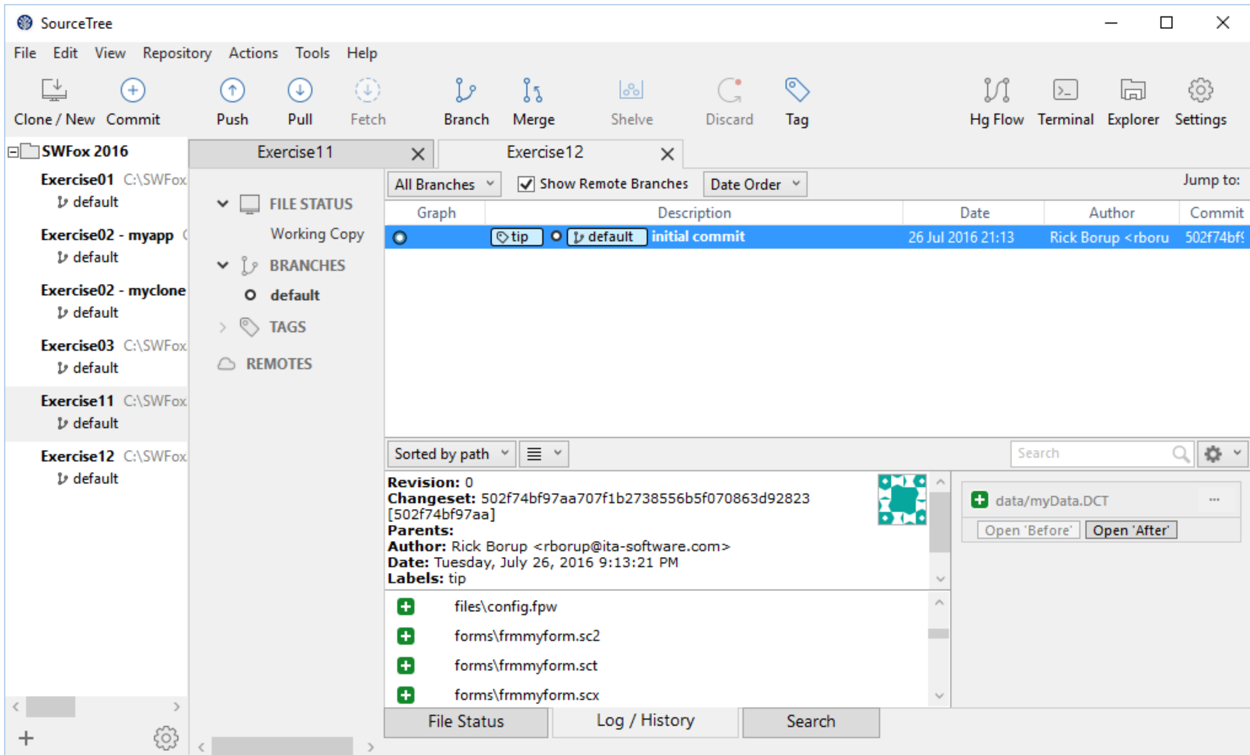


Figure 113. The SourceTree interface is similar to TortoiseHg. Select the File Status, Log/History, or Search page from the tabs at the bottom.

The File Status pane, shown in **Figure 114**, is where you can see a list of the individual files in the working directory. This is most often used to see and select new and modified files pending a commit, but the drop-down control at the top enables you to set a filter to view pending, conflicts, untracked, ignored, clean, modified, or all files as well as how you want them sorted.

Figure 114 shows the File Status pane with the *Pending files* filter selected. There are no modified files, so SourceTree displays the message *Nothing to commit*.

The lower portion of the File Status pane is for commits. The commit message goes in the edit box provided. Clicking the *Commit* button then commits the revisions to the selected files to the repository.

Copyright 2016, Rick Borup

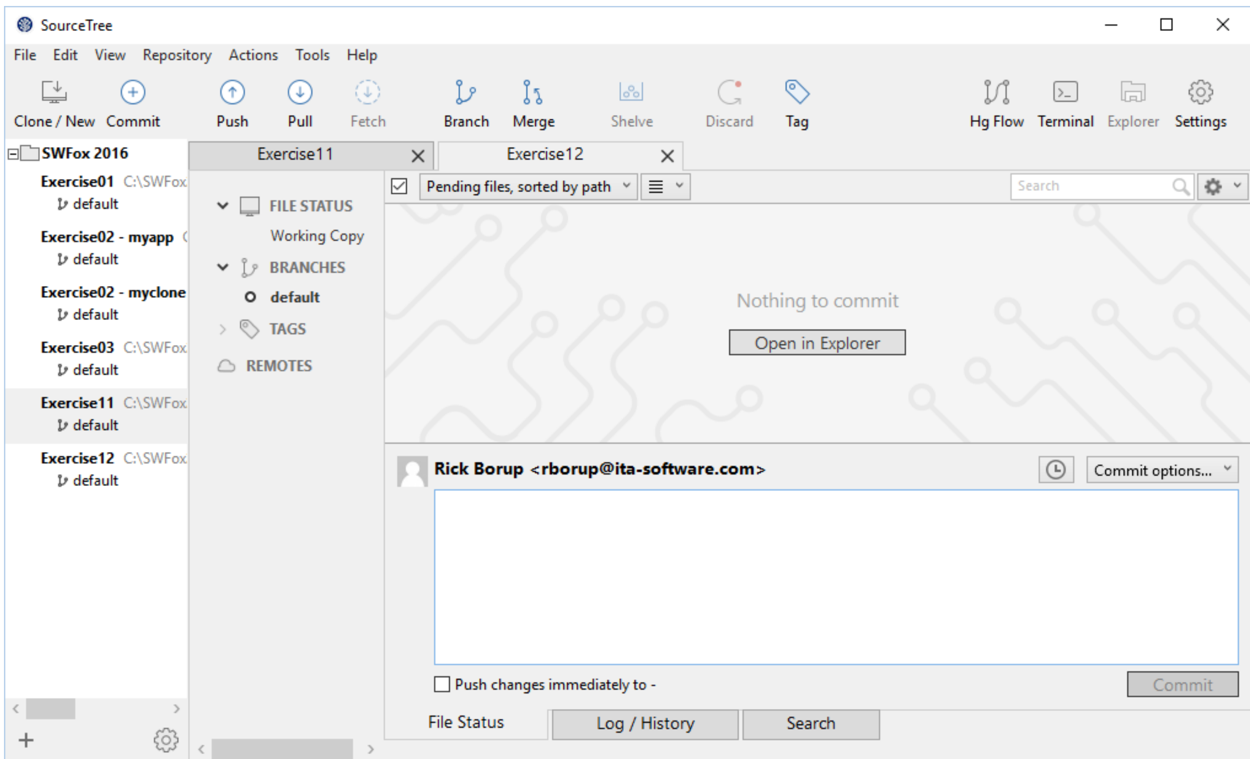


Figure 114. The Commit pane is the lower portion of the File Status page.

For Visual FoxPro developers, SourceTree has one distinct advantage over TortoiseHg: it provides for custom actions, which enables VFP developers to use FoxBin2Prg directly from the SourceTree window.

FoxBin2Prg

Because they cannot be diff'd and merged, VFP's binary source code files – screens, visual class libraries, reports, etc. – have always been a challenge for VFP developers when it comes to working with a version control system. FoxBin2Prg is a VFPX project created by Fernando D. Bozzo to help VFP developers meet this challenge. 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.

FoxBin2Prg enables two-way conversions between VFP's binary source code files and the text-equivalent files it creates. Not only can FoxBin2Prg generate PRG-style files from VFP binaries, but it can also regenerate the binaries from those files. This is a critical feature because without it, merging two revisions to a VFP binary file would not be possible.

FoxBin2Prg's PRG-style text files provide a familiar look for VFP developers, making it easy to see and understand the differences between two revisions of the same file during a merge. Developers can even make changes directly to the text-equivalent files before regenerating the binaries.

FoxBin2Prg is a VFPX project. It can be installed by itself, or via Thor. The FoxBin2Prg homepage at http://vfpx.codeplex.com/wikipage?title=FoxBin2Prg on VFPX includes basic information for getting started. At a minimum, read that one before installing and using FoxBin2Prg. The *FoxBin2Prg Internals and Configuration* page has more extensive documentation. Information can also be found in my whitepaper on *Git vs Hg* from Southwest Fox 2015, which is available for download from http://bit.ly/1VspWTJ.

If you're a VFP developer using Git or Mercurial for version control, using FoxBin2Prg is a no brainer. About the only question is, what's the best way to use it?

Using FoxBin2Prg from Windows File Explorer

One way is to hook FoxBin2Prg up to the Windows File Explorer context menu via "Sent to" shortcuts. Instructions are provided with FoxBin2Prg. If you installed it using Thor, there is a menu option to create the "Send to" shortcuts for you.

With the shortcuts in place, right-clicking on file with a recognized Visual FoxPro file name extension enables you run FoxBin2Prg and convert binary to text or text to binary, as needed. There is a separate menu item for each of these two functions, plus a third "smart" choice that performs the correct action based on the type of file. The drawback to using File Explorer integration is that it requires you to perform those steps from the Windows File Explorer. The context menu hooks do not work from the list of files in the VFP Project Manager.

Using FoxBin2Prg from Thor

If you installed FoxBin2Prg using Thor, it can also be run from the Thor Tools menus. Because you don't have to leave the VFP IDE, this may be preferable to using it from File Explorer, but you have to go four levels deep in the menu to get to the desired action items, so it's a bit cumbersome. This approach does have the advantage of enabling you to work with groups of files based on their status instead of having to work with each file individually.

Using FoxBin2Prg from SourceTree

Another way to use FoxBin2Prg is via a custom action in SourceTree. To set up a custom action for FoxBin2Prg, go to Tools | Options on the main menu and select the Custom Actions page. Fill in the information as shown in **Figure 115**, click OK to close the *Edit Custom Action* dialog and OK again to close the *Options* window.

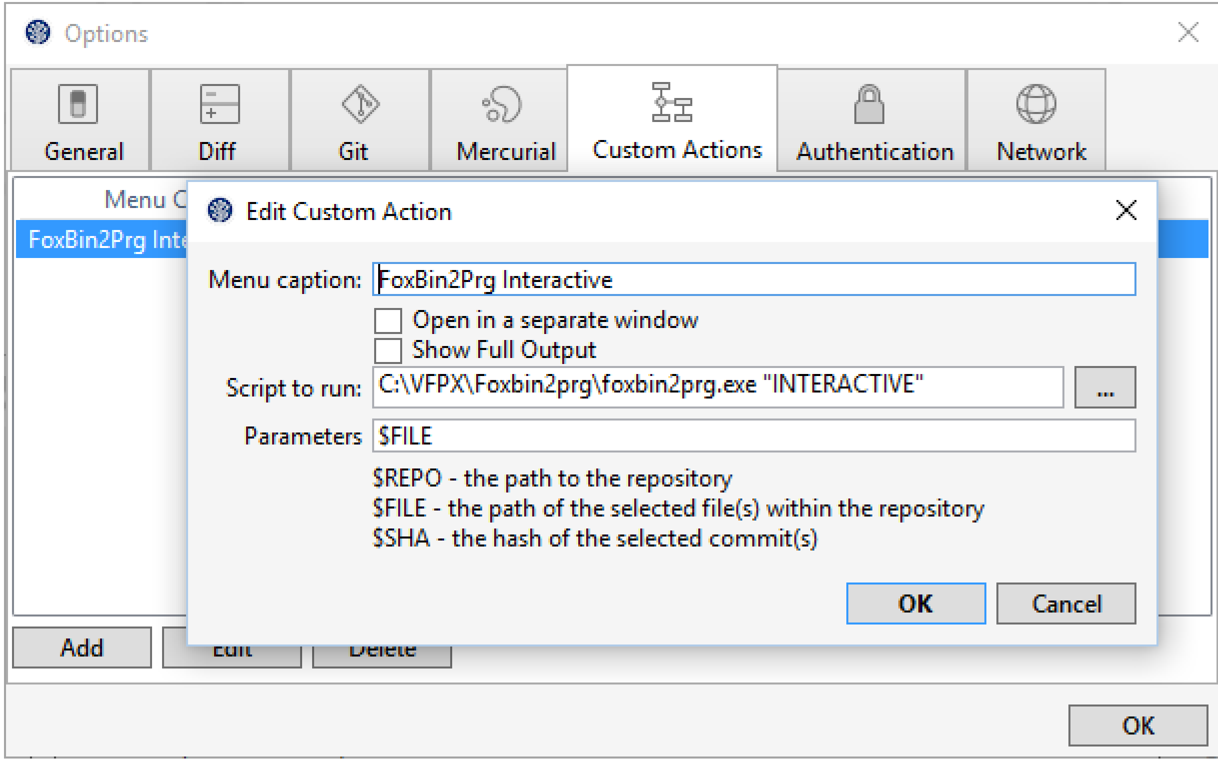


Figure 115. Set up a custom action to invoke FoxBin2Prg in its "interactive" mode.

The "INTERACTIVE" parameter tells FoxBin2Prg to perform the appropriate action depending on the file type. With this custom action in place, you can now access the power of FoxBin2Prg for any individual file from the list in the File Status pane. Simply right-click on a file and choose *FoxBin2Prg Interactive* (the name assigned to this custom action) from the Custom Actions item on the pop-up menu.

I find this to be a highly convenient way to work with FoxBin2Prg. You do have to handle each file individually, but if your commits are frequent and granular enough there are typically going to be only a handful of files that need binary-to-text or text-to-binary conversion.

Using the FoxBin2Prg API

Fernando has equipped FoxBin2Prg with an API so it can be driven programmatically. The API is explained in the FoxBin2Prg documentation. If you want to, you can certainly roll your own programmatic interface, but thanks to Mike Potjer's Git and Hg Utilities you don't have to.

Git and Hg Utilities

During my presentation on *Git vs Hg* at Southwest Fox in 2015, I mentioned I had recently become aware of a cool tool from Mike Potjer called Git Utilities. This tool integrates Git with FoxBin2Prg and makes it easier for VFP developers to use them.

At that time, Mike's utility worked only with Git. However, Mike released an update in late July, 2016, so it now works with both Git and Mercurial. The utility has appropriately been renamed Git and Hg Utilities.

The recommended way to install Git and Hg Utilities is via Thor. The project is hosted on Bitbucket, where you can read all about it on the Overview page at https://bitbucket.org/mikepotjer/vfp-git-utils.

In exercise 12, we'll be using all three of these tools – SourceTree, FoxBin2Prg, and Git and Hg Utilities – to step through a workflow involving revisions to a simple VFP app. As noted in the Git and Hg Utilities documentation, you should be committing both the binary files and their corresponding text equivalents to the repository in order for the utility to work as intended.

Exercise 12

The beginning point for exercise 12 is a simple VFP application located in the Exercise12 folder. This app, named *myVFPApp*, comprises two program files, a form, a report, a menu, a database container with one table, and four supporting files. All classes are VFP base classes, and everything needed to compile the app is contained within the project folder. FoxBin2Prg text files have been created on all the binaries except the database container and table. A Mercurial repository has already been created and the initial state of the project has been committed, as shown in SourceTree in Figure 113 above.

You can compile the app and run it if you want to. It's a top-level form with a button to preview a report listing the speakers at this year's conference. The form is shown in **Figure 116**.

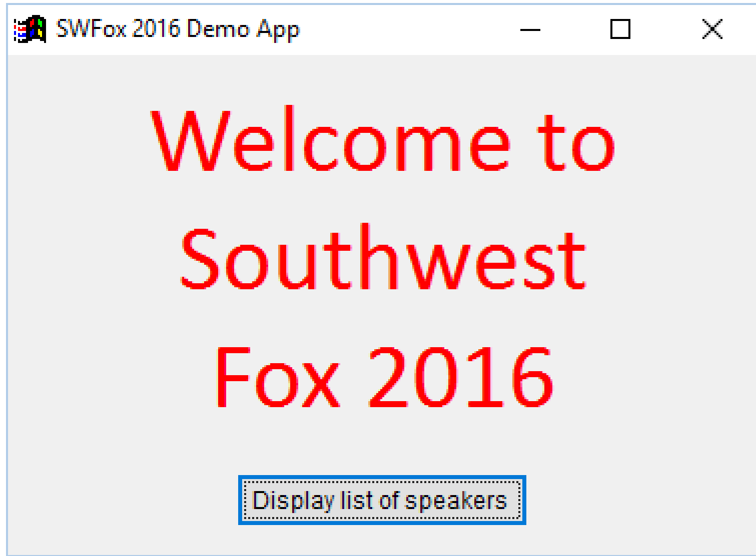


Figure 116. The demo app is a top-level form.

For this exercise, we are going to add an image to the form. The image file is named kokopelli.png, located in the *images* subfolder. The image file has not yet been added to the

Copyright 2016, Rick Borup

repository, so Mercurial will detect it as a new file. Forms are binary source code files, so by modifying the form we also create the need to generate an updated text-equivalent file for it before committing.

Open the project in VFP and open frmMyForm for modification. Add an image control somewhere on the form – the upper left area is suggested. Set the dimensions of the image control to width 80 and height 50, then select *images\kokopelli.png* as the value for the picture property. Move the *Welcome to Southwest Fox 2016* textbox so it does not overlap the new image control, save your changes, and build myVFPApp.exe. Compiling the EXE automatically adds the new image file to the Project Manager, meaning the PJX and PJT also become modified files.

In preparation for committing these changes to the repository, we need FoxBin2Prg to generate new text-equivalent files for the form and for the project file. Here's where Git and Hg Utilities makes things easier.

In this exercise, as in my own work, the PJX and PJT files are included in the repository along with their corresponding text-equivalent files. FoxBin2Prg won't be able generate a new text file for the project if it's open. SourceTree, and for that matter also TortoiseHg, will complain if the project file is open when you go to commit. If you do not include the PJX and PJT files in the repository (and Mike's documentation gives a good reason not to), you can leave it open. But for this exercise, close the VFP Project Manager before proceeding.

On the Thor Tools menu, choose Applications | Git-Hg Utilities | Prepare for Git-Hg Commit. In the scope dialog, mark the check box for *Current Dir*, confirm it's set to the Exercise12 folder, and click *Go* (**Figure 117**).

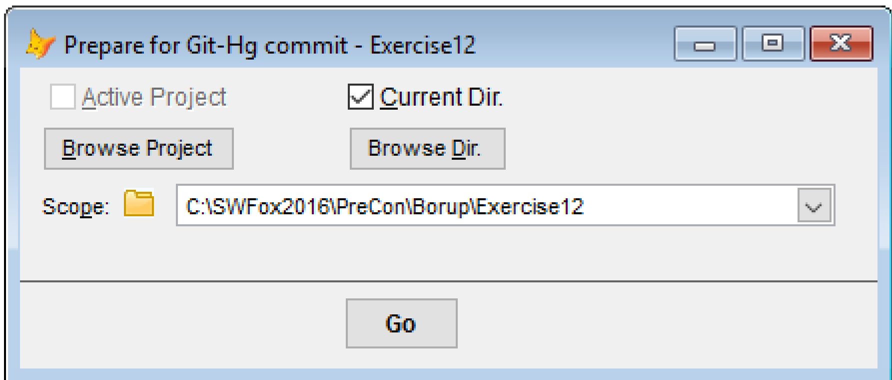


Figure 117. Use Mike Potjer's *Git-Hg Utilities* to prepare for commit.

Now open the repository for exercise 12 in SourceTree and select the File Status pane to see what's pending. It should show that the binaries and the text-equivalent files for the frmMyForm for and the myVFPApp project are now detected as modified files that need to be committed. The image file is not listed because the .hgignore file in use with this repository tells Mercurial to ignore .png files.

The list of pending files may also show a file named .hgtimestamp file as a new, untracked file (see **Figure 118**). If the option is set to do so, Git and Hg Utilities creates and uses this file to track the most recent timestamps of tracked files so it can restore them after a *rebuild all* for files that weren't actually modified. Figure 118 shows the files pending commit, before any action has been take on them. The popmenu.mpr file is listed because it got updated by the build.

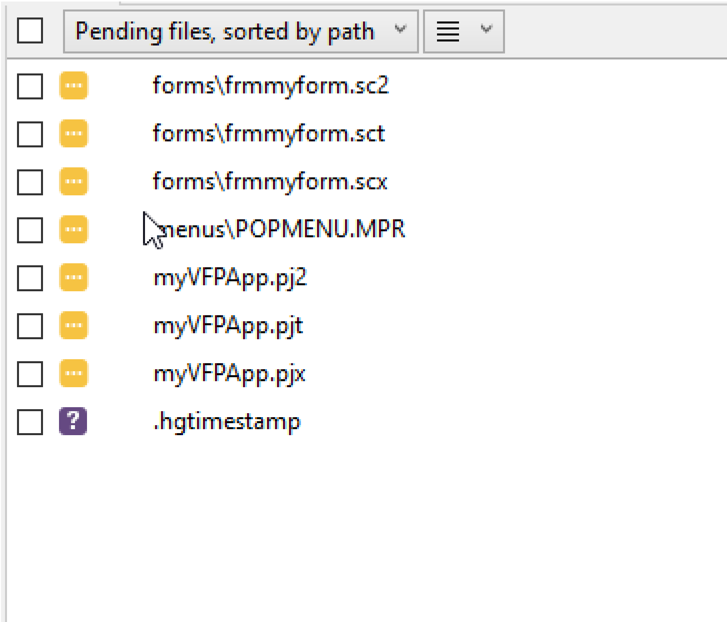


Figure 118. SourceTree displays the files pending commit after running *Prepare for Hg-Git Commit*.

Right-click on .hgtimestamp and choose Add to add it as a tracked file. SouceTree changes its icon to show that it's now a newly added file. Now mark the check boxes for all the files in the list to indicate they are to be included in the commit. Clicking the check box in the upper left corner marks the check boxes for all listed files in one step. **Figure 119** shows the list after these actions have been taken.

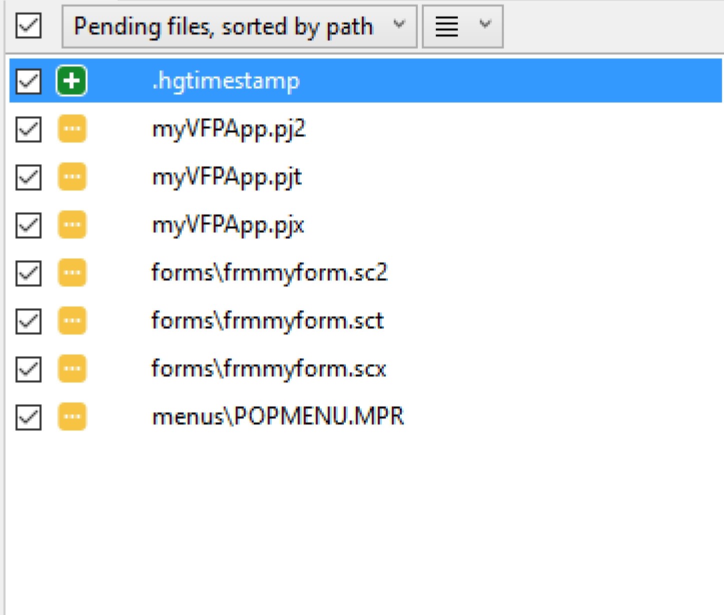


Figure 119. Add the .hgtimestamp file as a tracked file and mark all files to be committed.

Enter a commit message in the lower pane and click the Commit button to finish the commit. Then switch to the Log / History pane and review your work, as shown in **Figure 120**.

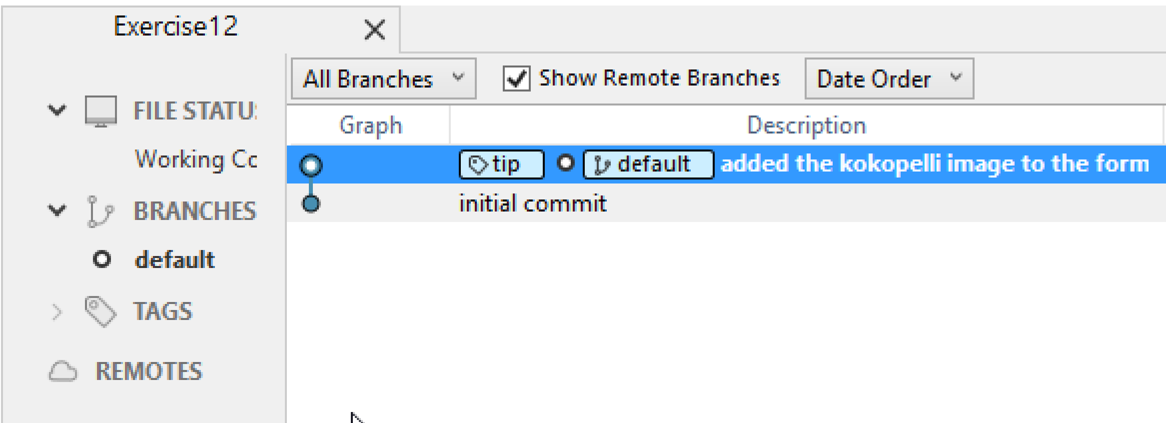


Figure 120. The revision history, after committing the changes to the demo app.

Git and Hg Utilities helps you out going the other direction, too. After pulling changes made by others into your repository, it may be necessary to merge the text-equivalent files and regenerate the binaries from the merged output. Git and Hg Utilities provides a post-checkout file synchronization process for this purpose.

These tools – SourceTree, FoxBin2Prg, and Git and Hg Utilities – can be used with any of the workflows we've used in this session. FoxBin2Prg and Git and Hg Utilities can also be used with TortoiseHg. SourceTree is recommended if you want to use the Hg Flow workflow, and also provides integration with FoxBin2Prg via a custom action. However, if you use Mike's Git and Hg Utilities you do not need a custom action.

Copyright 2016, Rick Borup

Summary

Distributed version control systems like Mercurial and Git are powerful tools. They make life much easier for development teams and benefit even the solo developer. Branching and merging are fundamental to the effective use of a DVCS, enabling developers to work independently and concurrently on separate features in the same project, and to release updates and hotfixes whenever they're ready. There are many different approaches to branching and merging. This paper explored several of them with the goal of helping each developer or development team choose one that works best for them.

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.

Copyright © 2016 Rick Borup. Windows® is a registered trademark of Microsoft Corporation in the United States and other countries. All other trademarks are the property of their respective owners