This paper was originally presented at the Southwest Fox conference in Gilbert, Arizona in October, 2013. http://www.swfox.net



Multi-track development strategies in DVCS

Rick Borup
Information Technology Associates, LLC
701 Devonshire Dr, Suite 127
Champaign, IL 61820
Voice: (217) 359-0918
Email:rborup@ita-software.com
Twitter: @rickborup

Distributed version control systems (DVCS) such as Git and Mercurial have become the de facto standard for modern source control. The use of local repositories and the ability to create clones greatly facilitate multi-track development work, wherein developers must simultaneously manage multiple lines of development representing different release versions of a product side by side with new feature development work and hot fixes. The challenge lies in deciding how to branch, when to branch, when to merge, and how many branches to maintain for which purposes. In this session, Rick explores some of the strategies that have proven effective in his own work and in the shared experiences of others.

© 2013 Rick Borup

Table of Contents

Introduction 3
Concepts 3
What's your precious? 3
Paradigm shift 4
Branching and merging 4
An immutable record of change 4
Single-track development 5
How to branch 7
When to branch 8
Managing branches 9
Naming schemes 9
Multi-track development 10
A first example 10
The revision history 12
Closing a named branch 15
Closing a cloned branch 16
A second example 16
The revision history 18
Multi-track hotfix 20
Complex strategies 24
Mercurial vs. Git 24
Preparing a VFP Project for DVCS 25
Diff engines 25
VFP's diff problem 27
Other issues with VFP 27
Reliable workflow 31
Changes you make 31
Changes others make 32
HgTools 32
Summary 33

Introduction

This session is about using distributed version control systems (DVCS) to facilitate multi-track software development projects. The term *multi-track development* refers to situations in which two or more lines of development on the same product need to occur at the same time. One example is when work on new features for a future release of a product needs to be performed concurrently with maintenance work and bug fixes for the current release of the same product. Another is when two or more teams of developers are working separately on several new features for a product with the intent to eventually combine all those features into a single release.

Distributed version control systems facilitate multi-track development by enabling developers to create branches in the source code repository. Each branch holds the revision history for a different line of development. The revisions in one branch can later be merged with the revisions from another branch to create a stable release with the combined features of both.

The material in this session is relevant for both solo developers and for teams. Either way, the assumption is you're already familiar with the basic workings of a DVCS and understand terms such as local repository, remote repository, push, pull, branch, and merge. If not, this would be a good time to go back and get a refresher before proceeding. The examples I'm using here are mostly from Mercurial, but the concepts are applicable to other DVCS such as Git and Veracity even though the syntax and terminology in those systems may differ.

This session also includes suggestions for preparing your Visual FoxPro project for a DVCS along with recommendations for adopting a reliable workflow in order to insure the ongoing integrity of your development process.

Concepts

What's your precious?

In Tolkien's *The Lord of the Rings*, the most precious thing in the world to Gollum is the ring. What's the most precious thing to us as software developers? Our source code, of course. We'd have a serious problem if our source code became lost or corrupted. Because of its critical importance, we take every possible precaution to protect our source code, to insure its integrity, and to back it up religiously.

Without a version control system, the source code files themselves are the only place the source code exists. It sounds redundant to say that, but it has profound implications. Without a version control system, developers typically rely on a standard hierarchical backup scheme—father, son, grandson, etc.—to preserve multiple copies of the source code across several generations of modifications. If it becomes necessary to recover some or all of the source code from an earlier state, the necessary files can be restored from the appropriate backup, assuming it still exists.

Although this works, it's a very limited approach that does nothing to facilitate multi-track development. If you restore a file from a backup and overwrite the current file, you lose any modifications that may have been made since the backup. If you restore the backup copy to a separate location and make changes there, you then have no way to merge those changes back into the current file.

Paradigm shift

Using a version control system changes the picture. A version control system maintains a separate set of files called a repository in which it stores the history of modifications to the source code files made over time. A version control system can restore the source code files to any given state entirely from the history contained in the repository. Unlike a generational backup system, in which the older backups are cycled out as newer ones are created, the repository contains a complete and permanent record of every state of every source code file.

Mercurial refers to the set of source code files for any given project as the *working copy*, while Git calls it the *workspace*. If the source code files in the working copy become lost or corrupted they can be restored, in whole or in part, from the repository. In addition, an entirely new working copy can be created at any time by cloning the repository.

So here's the paradigm shift – when you use a DVCS, your working copy is expendable! The source code files themselves are no longer primary, but are now merely derivatives of the change history stored in the repository.

Branching and merging

A distributed version control system enables developers to work independently on different lines of development by creating branches within a repository or by creating an entirely new working copy called a clone. The DVCS also enables the changes from one branch or clone to be merged into another branch or clone, thereby creating a new working copy from the combined revision history.

To help keep things straight, think about actions that change the working copy, such as modifications to local files, reverts, updates, and merges, as opposed to actions that change the repository, such as commits, pushes, pulls, patch imports, and so on. Those terms are from Mercurial; the terminology may differ in other DVCSs but the ideas are the same.

An immutable record of change

One other thing to keep in mind: For better or for worse, the repository is an immutable record of the changes to a project's source code. If you commit a source code modification to the repository and then later decide to change it back to the original, the repository has a record of both actions. This can be viewed as either good or bad. Alan Stevens recently griped about this on Twitter, saying "I know best how I want the history to look, I should be able to control it!". How do you feel about this?

© 2013 Rick Borup

Single-track development

Single-track development refers to a "straight line" approach where every revision is stored in the same branch. In Mercurial this is called the default branch. In Git it's called the master branch, and it has other names in other DVCS systems. Single-track development is similar to a single line of railroad tracks, in which every mile follows in a straight line from the previous mile with no switches or deviations along the way.

Figure 1 is a symbolic representation of what a straight-line revision history could look like on the default branch in a Mercurial repository.

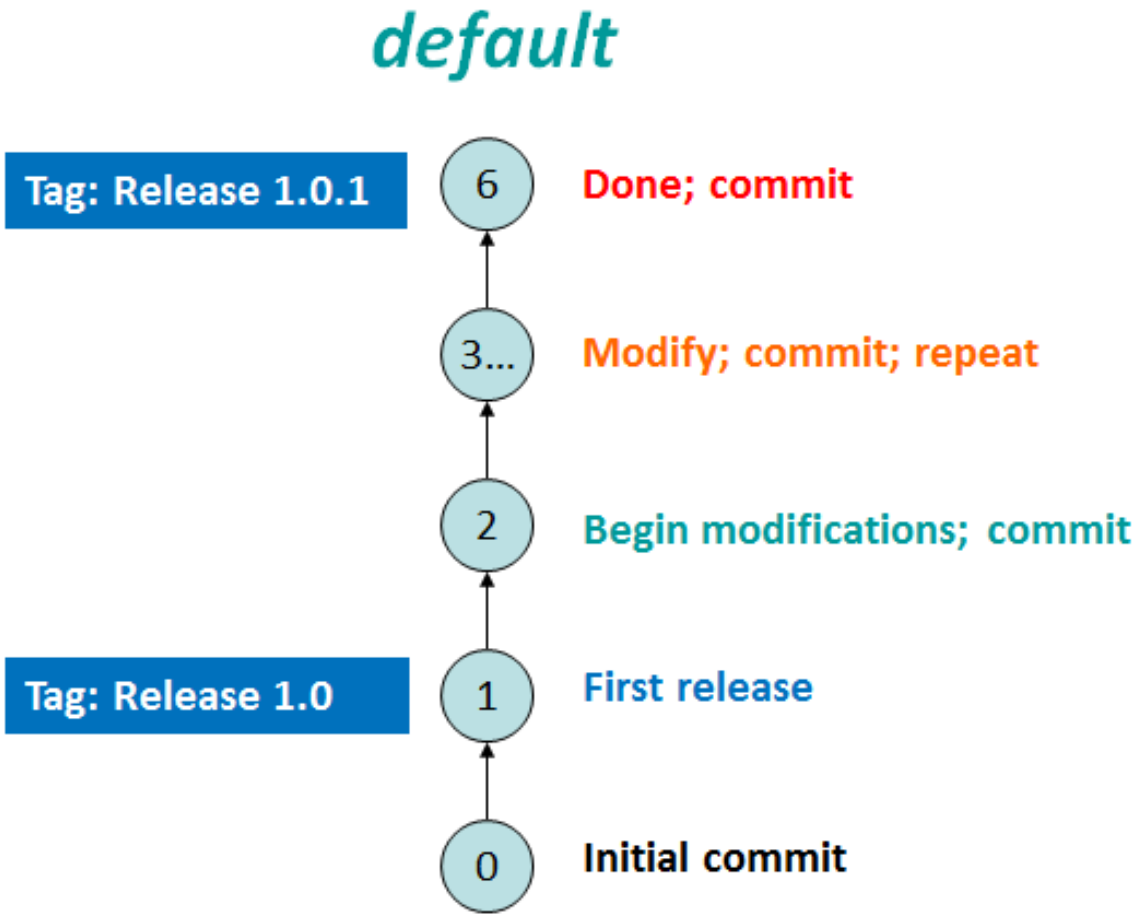


Figure 1: A symbolic diagram of single-track development. Read from the bottom up.

The main idea here is that all commits were done on the default branch. The parent of each revision is the previous revision on the same branch, so there has never been more than one head revision and therefore no need to do any merging. Tags have been assigned to revisions 1 and 6 to identify release versions of the software.

TortoiseHg is a free tool for working with Mercurial on machines running Microsoft Windows. Among many other benefits, it provides a graphical revision history for each

repository. Figure 2 shows the TortoiseHg graphical revision history corresponding roughly to the symbolic representation in the Figure 1. One thing about Mercurial is that adding a tag creates another revision, which can be seen as revision 8 in Figure 2.

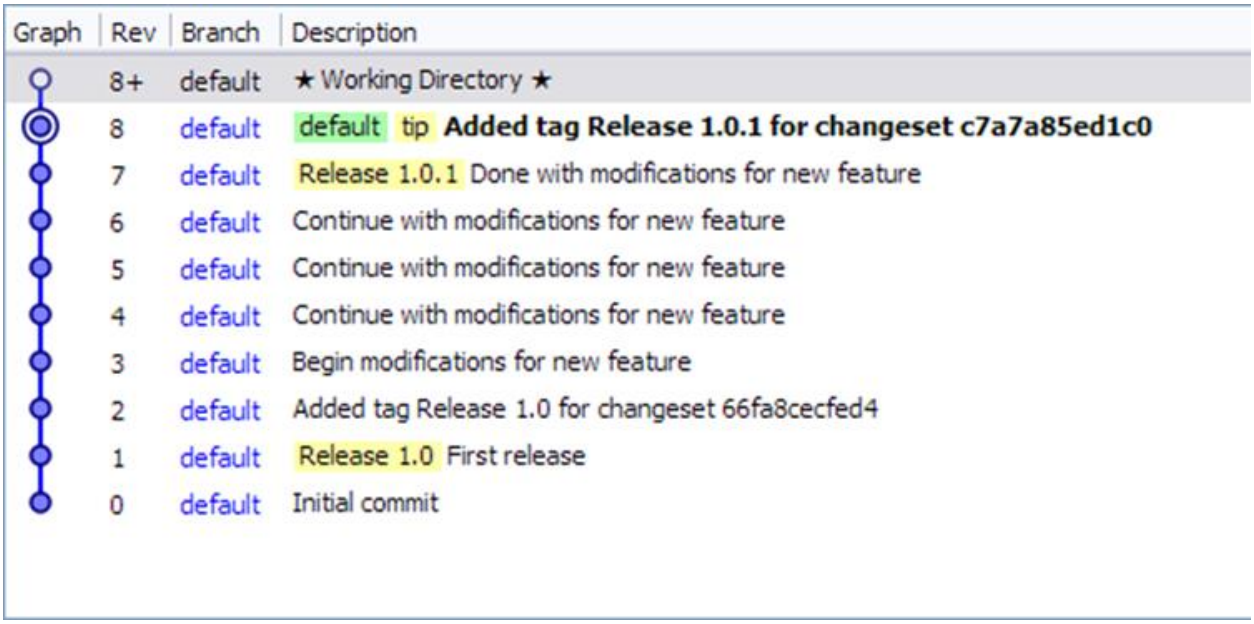


Figure 2: Single-track development results in a linear history. There is no branching.

Single-track development results in a linear history. There is no branching.

Or is there?

Mercurial creates anonymous branches whenever a new head is created. These anonymous branches exist in the default branch, and cease to exist when the new head is merged back into the main line of development. If the revision history shown above had been a little more convoluted, you might see it reflected in the graphical revision history. Figure 3 is an illustration of this, taken from some of my early work on a tool called HgTools for working with Mercurial from within VFP.

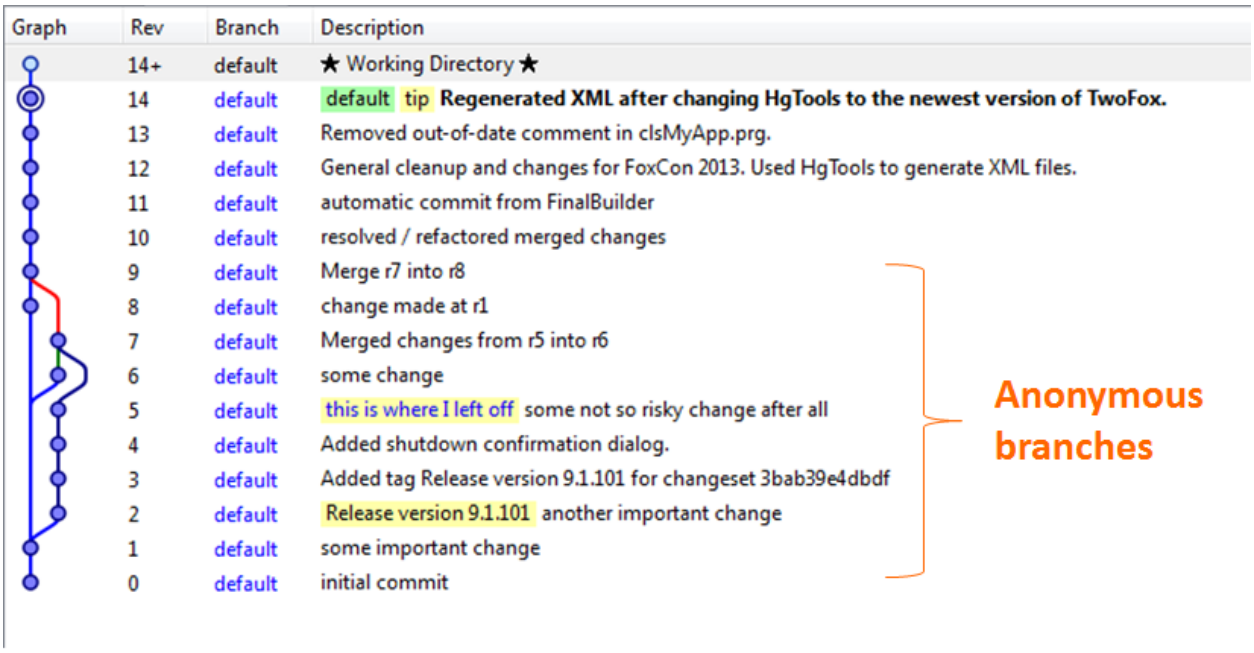


Figure 3: Mercurial creates anonymous branches when necessary, even if all commits are within the default branch.

In this history you can see that new heads were created a revisions 6 and 8. Revision 5 was merged into revision 6 to create revision 7, which in turn was merged into revision 8 to create revision 9. A series of straight-line revisions then followed from that point through revision 14.

How to branch

The fundamental technique for managing a multi-track development environment is to create branches. A branch can be thought of as “any divergent line of development”.

In a DVCS a branch can be a clone, which exists in a separate physical location with its own local repository, or it can be a named branch within an existing repository. There are some differences in the way Mercurial and Git handle named branches, but conceptually they’re the same.

Figure 4 illustrates the difference between a clone and a named branch.

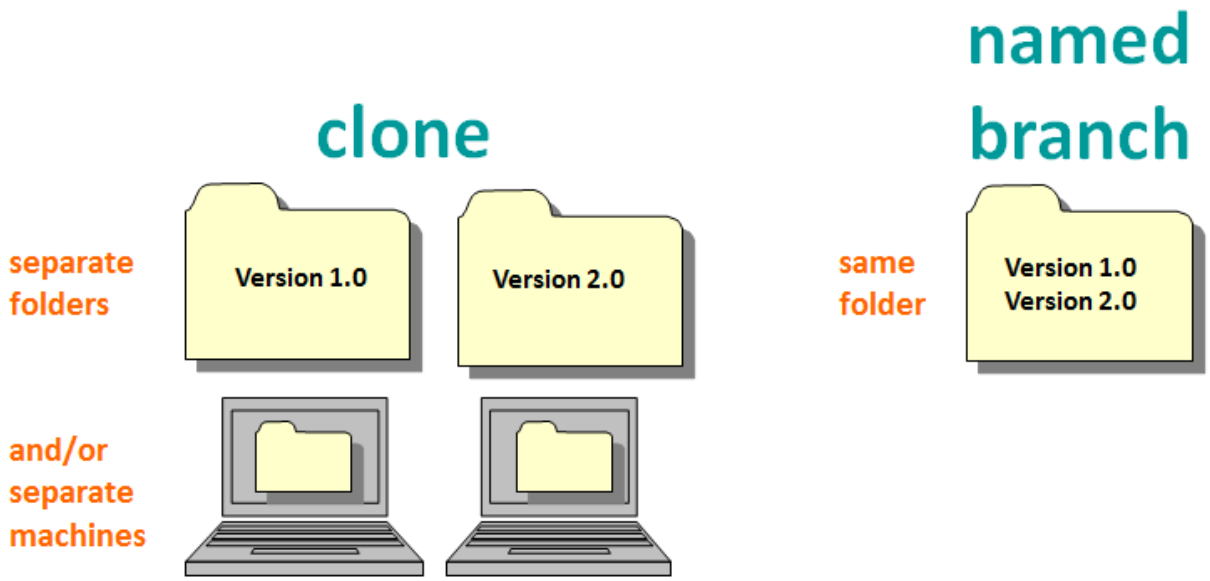


Figure 4: A clone exists in a separate folder on the same or on a different machine, while a named branch exists in the same folder as the default branch.

Although conceptually they're both branches, there are several difference between clones and named branches. Which one you choose depends on what's most appropriate for any given circumstance. The idea in both cases is to create a way to store changes for two or more divergent lines of development, most likely with the goal of eventually merging them back into the main line.

In Mercurial, a clone is created with the *clone* command while a named branch is created as part of a *commit*. Clones result in multiple physical repositories whereas named branches all exist within a single repository. Clones might be considered simpler but can also be slower to work with because you have to copy the entire revision history in order to create one. In Mercurial, modifications are exchanged among clones with the *push* and *pull* commands, while among named branches modifications are handled using *update* and *commit*.

When to branch

It's important to know how to branch, but it may be more important to know when to branch.

In some ways, knowing when to branch requires looking into the future. The basic guideline is to create a branch whenever you want to isolate a line of development. Common situations where it makes sense to create a branch include when beginning work on a new major release version, when adding a new feature while keeping the stable version available for bug fixes, when setting out on a line of experimentation that may or may not retained, and in order to facilitate the division of labor among team members.

> *The basic guideline is to create a branch whenever you want to isolate a line of development.*

Clones are created before embarking on a set of modifications. You create the clone, derive a working copy from the clone, and then begin making changes and committing to the cloned repository.

Named branches, on the other hand, are created *after* making a set of changes. This is because, in Mercurial, named branches are created as a byproduct of the *commit* command. You make a set of modifications, then specify a named branch when you commit those modifications to the repository. This is why I said branching requires looking into the future: with named branches, you start out by making modifications to the files in the working copy the same as always, but you need to remember to create a new branch when it comes times to commit those modifications to the local repository.

A parallel question is how many branches to create. In any given repository there will always be one main branch, called the *default* branch in Mercurial and the *master* branch in Git. In addition to this, some developers like to maintain a unique branch for each release version of a product, whether clones or named branches, while others might just use tags within a single branch. Clones make the most sense to me when there is more than one major version of a product, say 1.0 and 2.0, each of which has been released into the wild and needs to maintained separately.

Additional branches can be created as desired for new feature development, experimentation, hotfixes, and other reasons. It's a good idea to keep it simple, though. Multiple branches are a double-edged sword: every benefit introduces a corresponding level of complexity you'll eventually have to deal with.

Managing branches

As the number of branches grows, the job of managing them becomes more complex. It's therefore a good idea to set up a scheme for handling them in advance.

Naming schemes

One aspect of a branch management system is a naming scheme. The DVCS itself references revisions by their revision ID, in whatever form that takes for the given DVCS, but this is usually a cryptic identifier similar to a GUID and therefore of limited value to a human being. A good naming scheme helps developers understand the nature and purpose of each branch in plain language.

Branches typically fall into three groups:

- The *default* branch, which exists automatically. In Mercurial, this branch is actually named “default”.

- A *release* branch or branches. This branch might be named the “Stable” branch if there’s only one, or if there is more than one then named for the version number, such as v1.0 or v2.0.

- A *development* branch or branches. These are used for new feature work and/or experimentation and are given names representing their purpose, such as “Development” or “Feature X” or whatever.

In addition to creating named branches, Mercurial and other DVCS enable you to create a tag and associate it with a given revision. Like a branch name, a tag is a descriptive word or phrase that helps identify the revision in plain language.

In Mercurial, and probably in other DVCS as well, it’s a good idea to use a different naming scheme for tags than for branches. This helps the developer keep straight whether a name refers to a branch or to a tag, and also avoids the potential for conflicts within the DVCS itself when referencing revisions by name.

Let’s assume you maintain separate branches for each major release of a product, and within each branch you tag the revisions representing each minor release. The branches could therefore be named “Version 1.0” and “Version 2.0”, or “v1.0” and “v2.0” if shorter names are desired, while the tags within a branch could be named “Release 1.0”, “Release 1.0.1”, etc. This is the scheme I use and so far it’s worked well.

Multi-track development

Now that we’ve covered some of the basics of branching and merging, let’s take a look at how this applies to some real-life development situations.

A first example

One of the most common multi-track development scenarios arises when you want to embark on the development of a new feature while at the same time preserving the source code that supports the current release of a product, for example in case you need to make a bug fix to the current release before the new feature is ready. The examples that follow use both named branches and tags to identify and illuminate the steps in the development process.

Take a moment to look at Figure 5, representing the scenario described above.

© 2013 Rick Borup

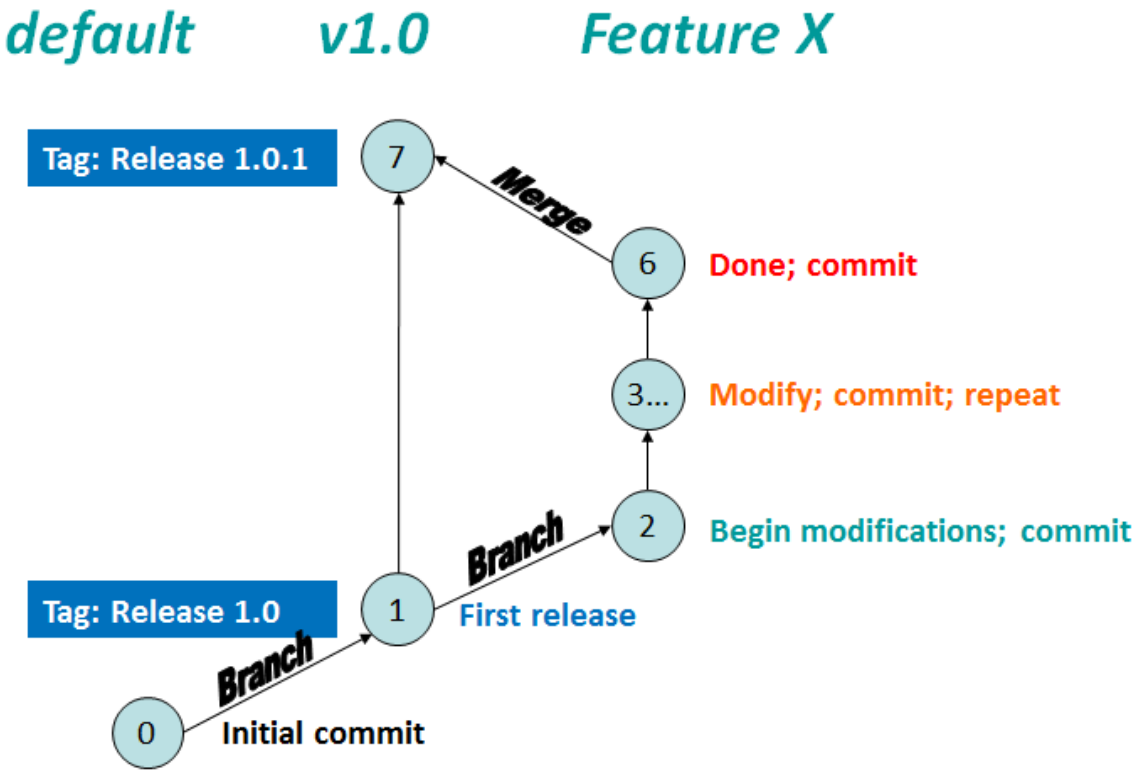


Figure 5: An example of multi-track development. Read from the bottom up.

The labels across the top are the branch names. In this example, there are three—default, v1.0, and Feature X—forming three columns in the diagram. The blue circles represent the points at which modifications were committed to the repository, with their position in one of the three columns indicating which branch they were committed to. Tags are shown in blue rectangles. The black italicized labels indicate the action that was taken, while notes and comments appear off to the right. When reading this diagram, read from the bottom up.

In this example, a product was developed and prepared for its first release. The source code was committed to the default branch of the repository at step 0. Knowing this was a stable release, a branch named "v1.0" was created and the source code was committed to that branch at step 1, which was then tagged as "Release 1.0" and became the first release of the product.

The developers then began work on a new feature planned for Release 1.0.1, but they knew it was important to keep the source code for Release 1.0 intact in case a hotfix or bug fix was required before the new feature was ready. Therefore, when the first set of modifications for Release 1.0.1 are completed, those modifications are committed to another new branch named "Feature X" (see step 2).

Work on the new feature then continues with various commits at steps 3 through 6, all of which are committed to the Feature X branch. When the new feature is complete and ready

to go, the modifications in the Feature X branch are merged with the release version from revision 1 and committed back into the "v1.0" branch, as shown in step 7. The merged source code becomes Release 1.0.1 and is tagged accordingly.

The revision history

Figure 5 is a conceptual diagram of the development process described above. Now let's take a look at how the revision history evolved the way the developer would see it in Mercurial.

Figure 6 shows the Mercurial revision history after revision 2 has been committed. This screenshot is from TortoiseHg, the GUI interface to Mercurial on Windows. The leftmost column is the graphical representation of the history. The second column shows the revision number, while the third column shows the branch name and the fourth column is the commit message. Revision number "2+" on the top line means the working copy is derived from revision 2, or to put it another way, revision 2 is the parent of the working copy. At this point, you can look at the "Branch" column and see there are two branches in the repository, "default" and "v1.0".

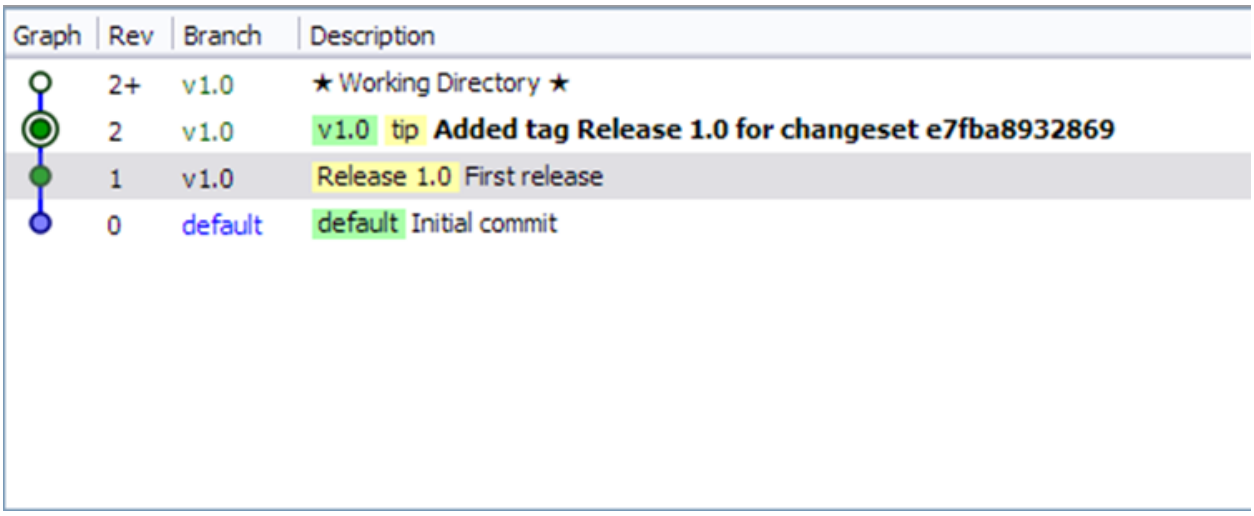


Figure 6: The revision history as of Release 1.0, as displayed by TortoiseHg.

Again referring back to Figure 5, the next step was to begin work on the new feature. This initiated a new line of development, so modifications for the new feature were committed to a new "Feature X" branch to isolate them from the stable "v1.0" branch. Figure 7 shows the TortoiseHg revision history after the first commit to the Feature X branch.

© 2013 Rick Borup

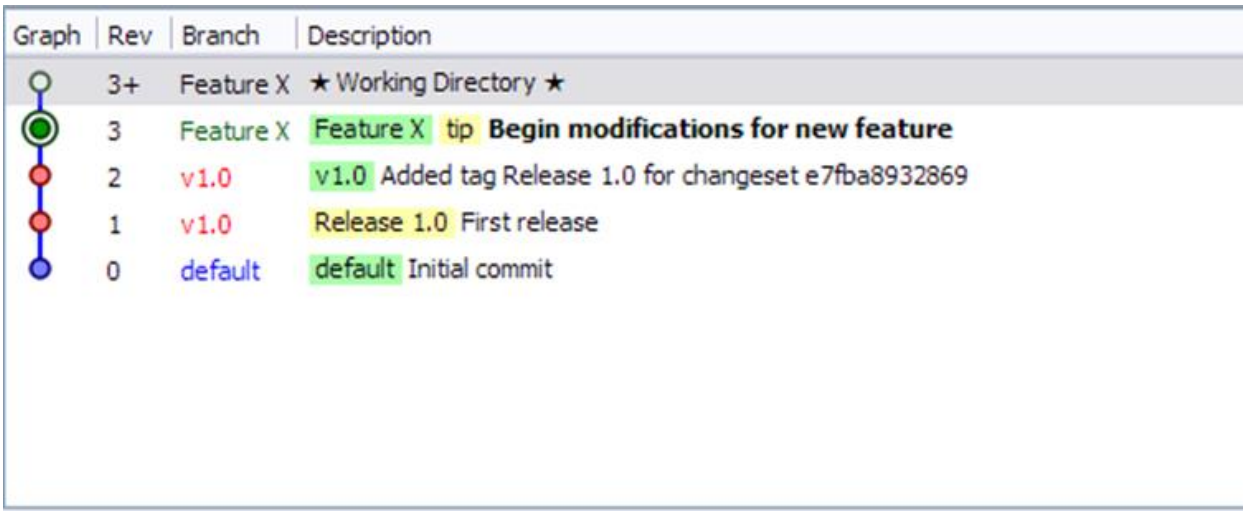


Figure 7: Modifications for the new feature are committed to the Feature X branch.

You may have noticed the revision numbers in the preceding two figures do not correlate exactly with the numbered steps in the conceptual diagram in Figure 5. This is because, in Mercurial, adding a tag creates a new revision in the repository. Looking at Figure 7, you can see that adding the "Release 1.0" tag created revision 2, which in turn means the first commit to the Feature X branch became revision 3. Revision 3 therefore corresponds to step 2 in the conceptual diagram. This variation will continue throughout the rest of this first example.

Figure 8 shows the revision history at the point where work on Feature X is complete and ready to be merged back into the main v1.0 branch. Notice the chain of revisions from 3 through 7, all of which have been committed to the Feature X branch. Also observe that the working copy (the "Working Directory" in the diagram) is based on revision 7 in the Feature X branch.

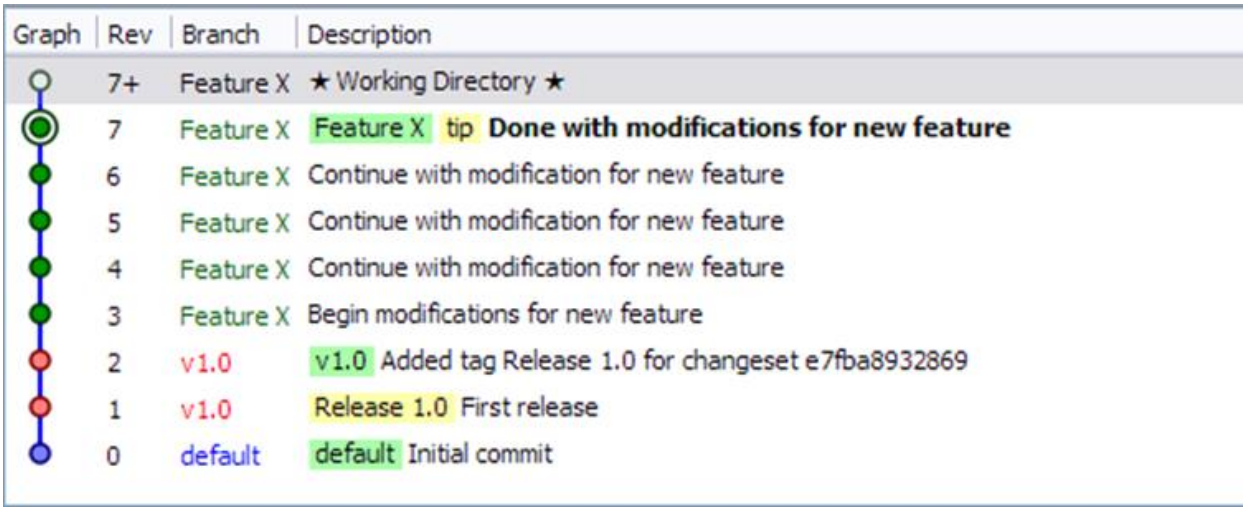


Figure 8: Work on Feature X is now complete.

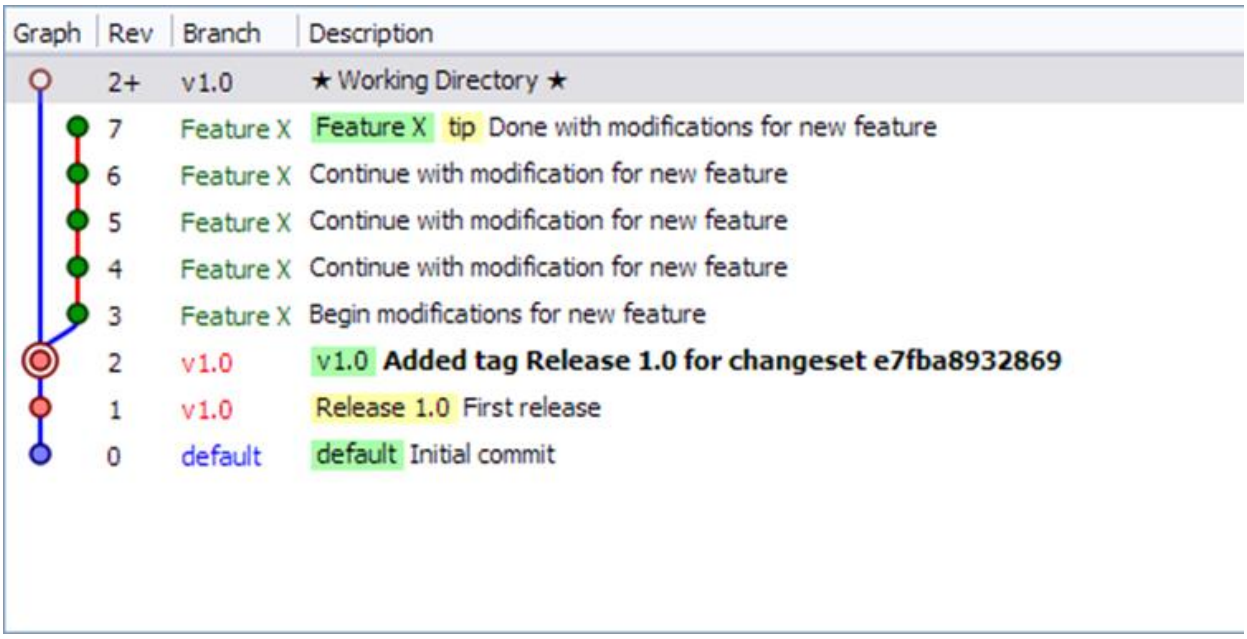
The developers are now ready to merge the Feature X branch back into the main v1.0 branch to create Release 1.0.1. When setting out to merge changes, you need to decide which is the "to" side and which is the "from" side. The merged changes will end up on the "to" side. In this example, we want the changes to end up in the "v1.0" branch, so it becomes the "to" side and the "Feature X" branch is the "from" side.

Here's where Mercurial may differ from the way other DVCS handle merging. In Mercurial, you first update the working copy to the revision you want as the "to" side, then perform the merge with the "from" side. The first step in this example is therefore to update the working directory to the most recent revision in the "v1.0" branch, which is revision 2. [1]

But wait a minute, you say! Since all of this is taking place using only one working copy of the source code files, won't this so-called update wipe out all the modifications we've just finished making to the source code files?

The answer is yes, but it doesn't matter. Remember that the source code files are expendable and can be re-created from the history in the repository at any time. Therefore, while the "update" to revision 2 does in fact wipe out the modifications we've made since that point, the merge we're about to perform brings them back in and all will be well.

Figure 9 shows the revision history after the working copy has been "updated" to revision 2. Note that the working directory is now identified as 2+, meaning it's derived from revision 2. Also note this is the first time we've seen any indication of branching in the graphical representation of the revision history, even though there have been three branches in the repository ever since revision 3, as can be seen in the Branch column.



[1] Mercurial uses the term *update* to mean the act of changing the contents of the working directory to match a specific revision, even if that revision is older than the one on which the working copy was previously based.

Figure 9: The working copy is updated to revision 2 prior to merging the Feature X branch back into v1.0,

After the update, the working copy is based on the latest revision in the "v1.0" branch, which becomes the "to" side in the merge operation. Revision 8 is created as the result of merging the modifications from revision 7 in the Feature X branch into the source code from revision 2 in the v1.0 branch, as shown in Figure 10.

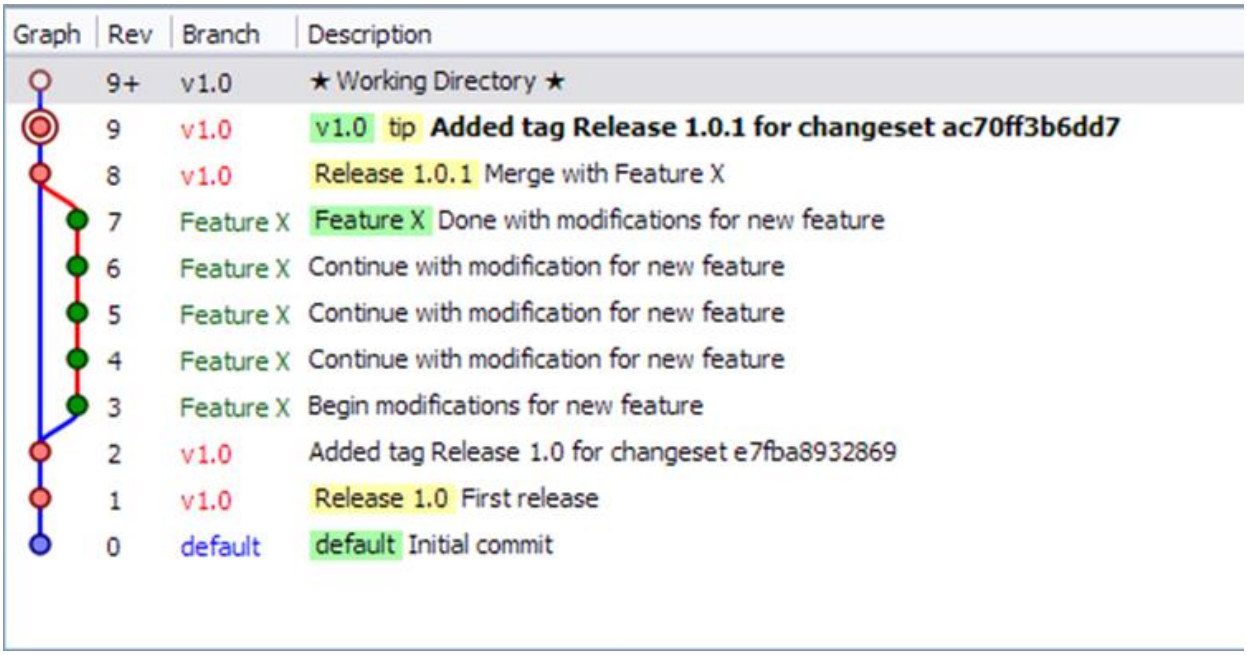


Figure 10: The revision history after merging Feature X back into v1.0 and tagging Release 1.0.1.

Figure 10 also jumps ahead one step and shows the addition of the "Release 1.0.1" tag, which appears as revision 9. This concludes the first example and brings the revision history into sync with the conceptual diagram shown in Figure 5. Note again that revision 9 corresponds to step 7 in that diagram, with the difference resulting from the addition of the two tags, each of which added its own revision to the history.

Closing a named branch

Now that work on the new feature is complete, it's doubtful the Feature X branch will ever need to be used again. Bug fixes will be done either in the v1.0 branch or in a new branch for that purpose, and work on the next new feature will be done in a new "Feature Y" branch. So, what if anything should be done with the old Feature X branch?

The answer is, it depends. It's actually not necessary to do anything, so you can leave things the way they are and get along just fine. However, there is a way to close a named branch, which is something you might elect to do if you want to make it apparent to other developers that the branch is in fact closed and should not be used again – in other words, that no future modifications should be committed to it. Marking a branch as closed causes TortoiseHg to show the "closed" symbol in its graphical revision history and tells Mercurial to no longer include the branch in response to the "hg branches" command.

In Mercurial, closing a branch is accomplished as part of the commit process. In this example, closing the Feature X branch involves updating the working copy to revision 7, which is the latest revision in that branch, performing a commit on that branch with the "close" option selected, and then updating the working copy back to the head revision on the v1.0 branch. Figure 11 shows the revision history after closing the Feature X branch.

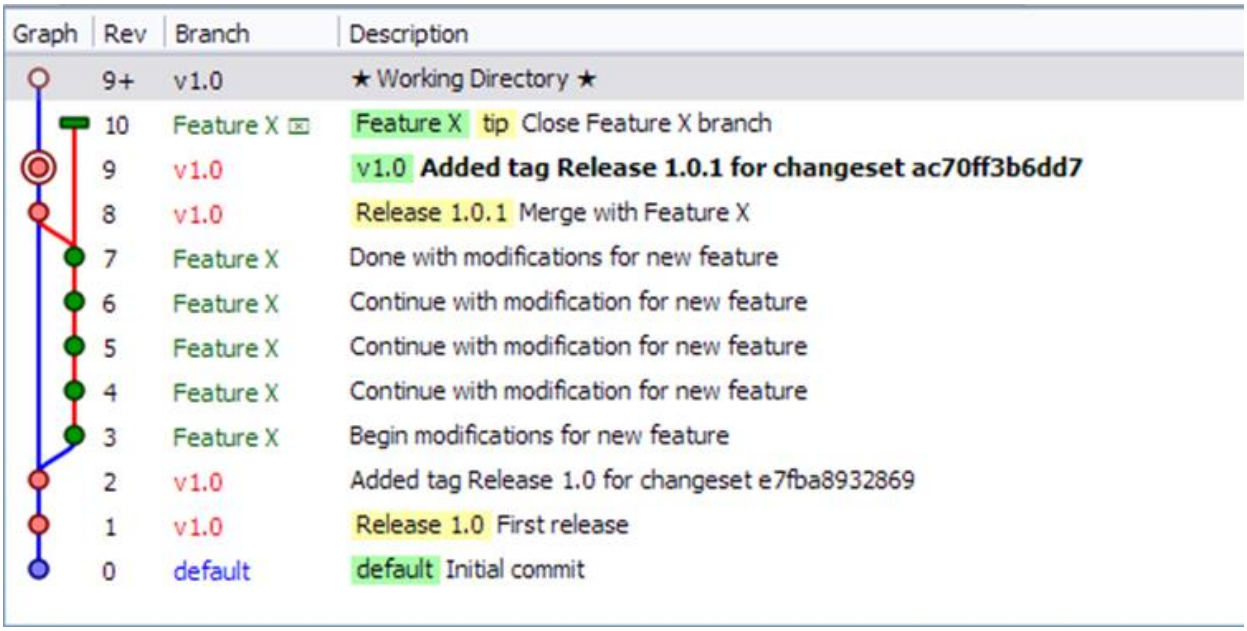


Figure 11: The TortoiseHg revision history shows a "closed" symbol when a named branch is closed.

Closing a cloned branch

If you're working with a clone instead of a named branch, the mechanics of closing it are different but the idea is the same. With a clone, you're working with a completely separate working copy with its own local repository. After making whatever modifications you want to and pushing them back up to a central repository, the clone may no longer be needed.

There is no formal mechanism for closing a clone, nor is one needed. The simplest thing is just to abandon it: stop using it, and perhaps even to delete it from your local storage. If you want to keep it around for some reason, you might want to mark it read-only as a reminder that no more modifications should be made or committed.

In either case, whether working with a named branch or a clone, the decision as to *when* to close a branch is entirely up to you or to the guidelines your development team has chosen to adopt.

A second example

The concepts illustrated in the first example can be extended to situations where more than one release branch is being maintained.

Building from where we left off, let's assume the developers now decide to create a major new version, which will be called version 2.0. It will still be necessary to maintain and

support version 1.0 since it's in use by several paying customers, but new development and new features will be targeted for version 2.0. With this in mind, a new branch named "v2.0" is created off the Release 1.0.1 revision. The conceptual diagram for this and the next several steps is shown in Figure 12.

Work on Feature Y proceeds on the v2.0 branch through steps 11 and 12. Simultaneously, customers using version 1.0 have requested some changes that the developers agree to but which won't be necessary in version 2.0. Work on version 2.0 is temporarily set aside while modifications are made to version 1.0 and committed to the v1.0 branch as step 13, resulting in release 1.0.2. Because version 1.0 and version 2.0 are in separate branches, these parallel modifications do not affect each other even though only one working copy is involved.

Release 1.0.2 is shipped and work returns to Feature Y for version 2.0. At step 15 in the diagram, Feature Y is ready to go and those modifications have been merged back into the v2.0 branch. That revision is then tagged as Release 2.0 (step 16) and the product is shipped. At this point, the Feature Y branch is no longer needed and could be closed in the same way the Feature X branch was closed in the first example.

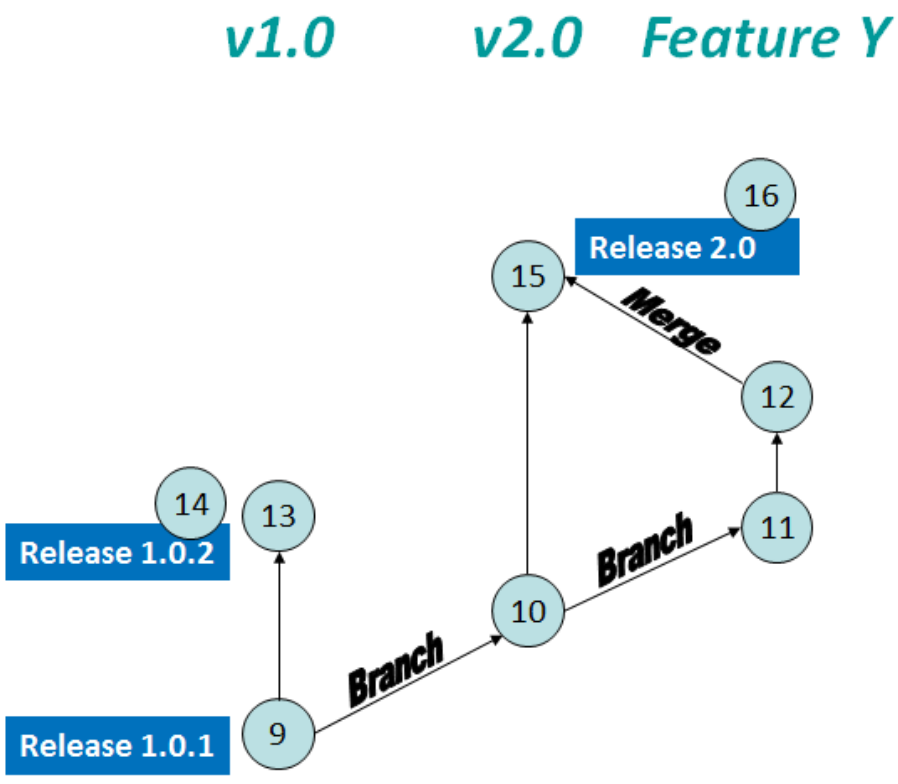


Figure 12: This second example is a conceptual diagram of two major releases, each in its own branch.

The revision history

The revision history for this second example picks up where we left off in the first example. Branch "v2.0" was created at revision 10 as the main branch for major version 2.0. At revision 11, the "Feature Y" branch was created and work on Feature Y was completed at revision 12, as shown in Figure 13.

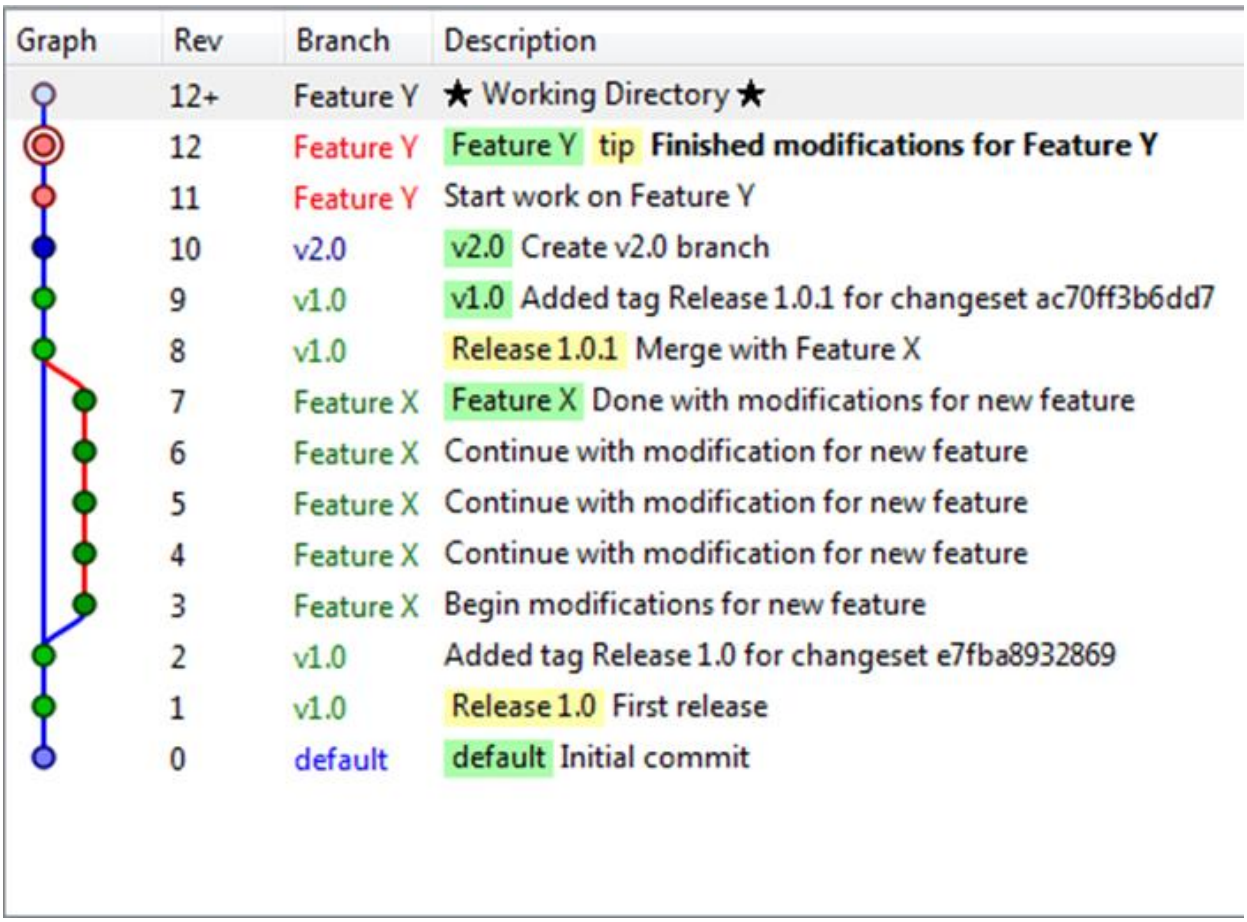
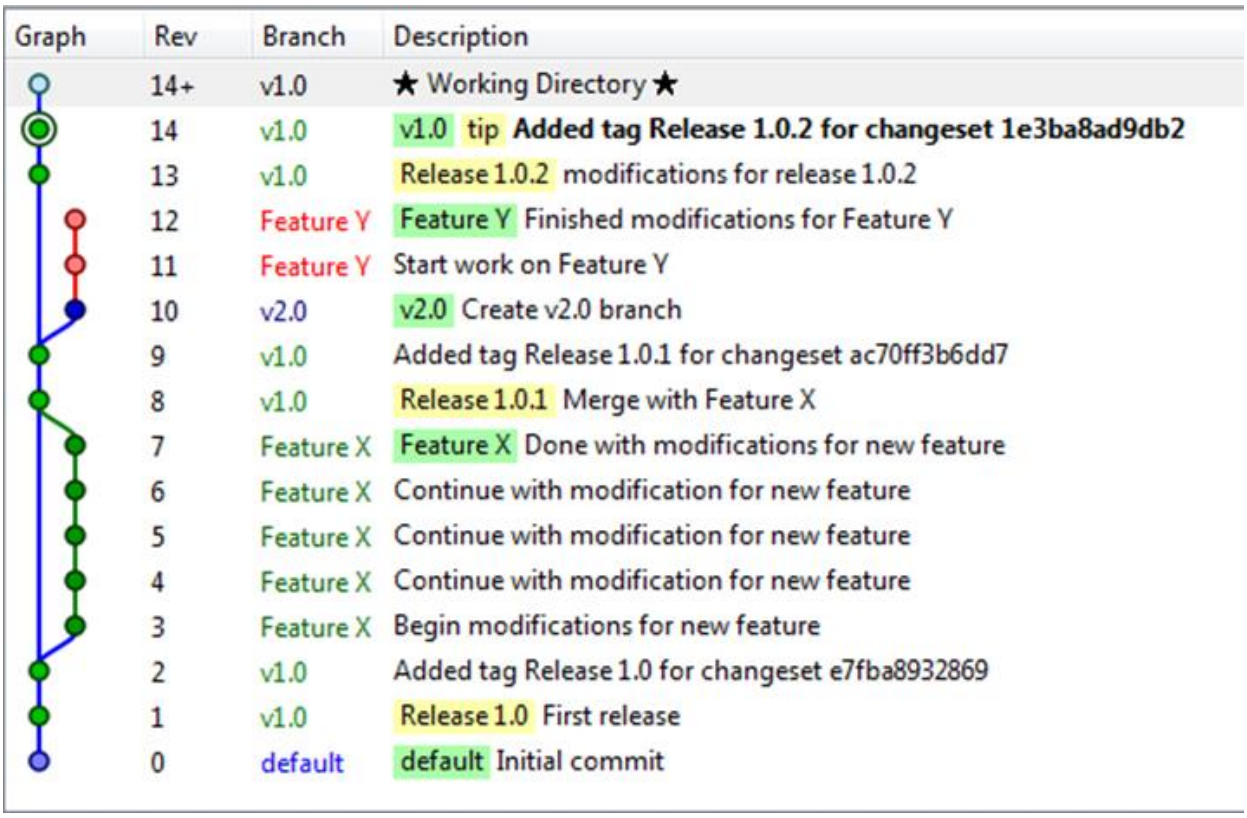


Figure 13: The Feature Y branch is created for work on a new feature for v2.0.

At this point, work was suspended on version 2.0 so the developers could make a requested change for the customers using version 1.0. The working copy was updated to revision 9, which was the latest for version 1.0. The requested modifications were then made and committed, and the updated product was released and tagged as Release 1.0.2 at revision 14 as shown in Figure 14.

© 2013 Rick Borup



Graph	Rev	Branch	Description
	14+	v1.0	★ Working Directory ★
	14	v1.0	v1.0 tip **Added tag Release 1.0.2 for changeset 1e3ba8ad9db2**
	13	v1.0	Release 1.0.2 modifications for release 1.0.2
	12	Feature Y	Feature Y Finished modifications for Feature Y
	11	Feature Y	Start work on Feature Y
	10	v2.0	v2.0 Create v2.0 branch
	9	v1.0	Added tag Release 1.0.1 for changeset ac70ff3b6dd7
	8	v1.0	Release 1.0.1 Merge with Feature X
	7	Feature X	Feature X Done with modifications for new feature
	6	Feature X	Continue with modification for new feature
	5	Feature X	Continue with modification for new feature
	4	Feature X	Continue with modification for new feature
	3	Feature X	Begin modifications for new feature
	2	v1.0	Added tag Release 1.0 for changeset e7fba8932869
	1	v1.0	Release 1.0 First release
	0	default	default Initial commit

Figure 14: Work on Feature Y for v2.0 is set aside so a change can be made to v1.0. The Feature Y branch remains open but the working copy is now derived from revision 14, which belongs to the v1.0 branch.

The developers then picked up work on major version 2.0 again. Feature Y was already complete at revision 12 and is the only feature required for version 2.0, so that version is now ready for build and release. Feature Y is merged back into the main v2.0 branch at revision 15 and that code is tagged as Release 2.0 at revision 16, as shown in Figure 15.

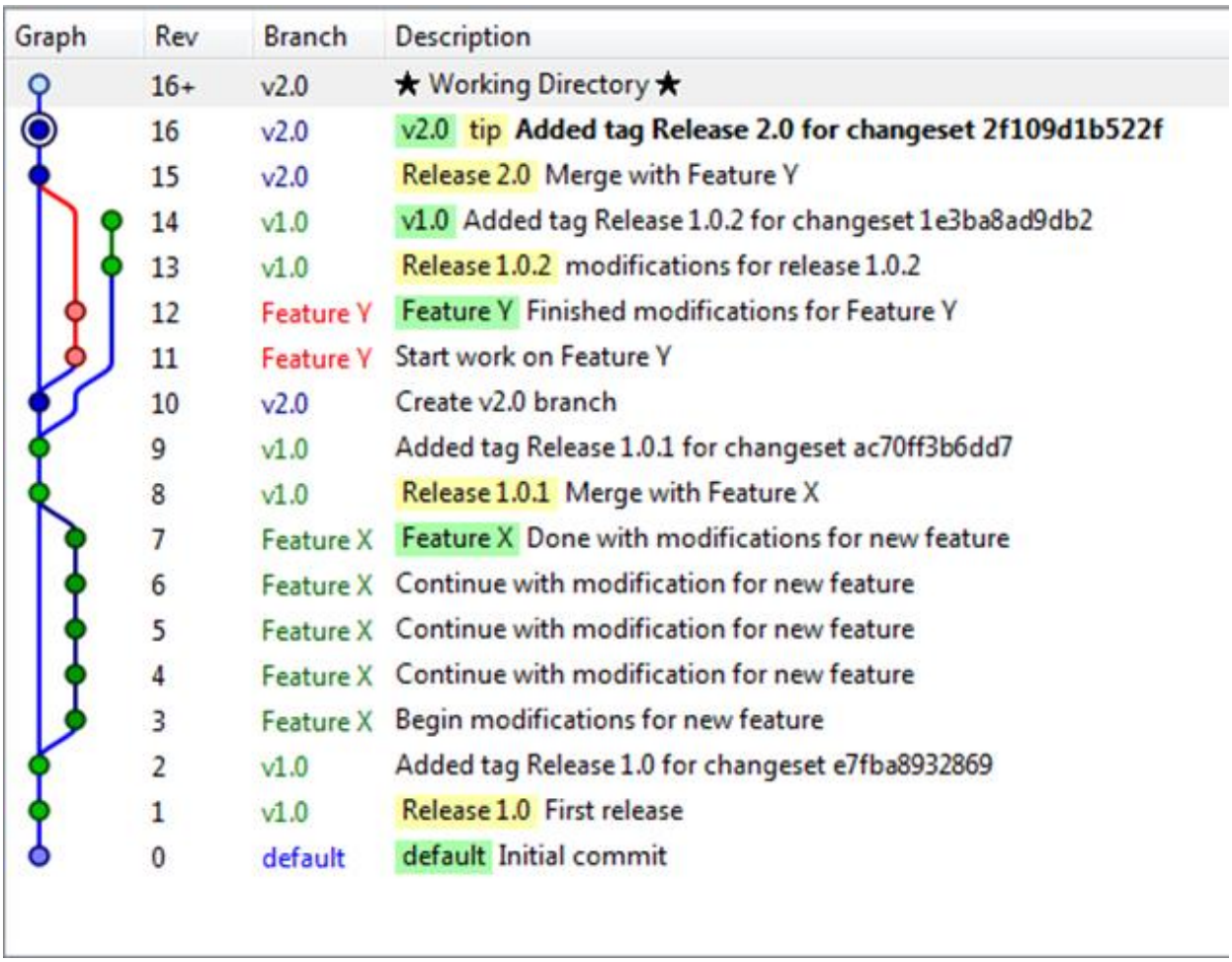


Figure 15: Feature Y is merged back into the v2.0 branch and Release 2.0 is tagged. The v1.0 branch remains open.

Note that the v1.0 branch is still open at revision 14. The developers anticipate there will be future modifications to version 1.0 because many customers are still using it, even though version 2.0 is now the latest release. For this reason, the v1.0 branch is left open.

Multi-track hotfix

In the previous example, the developers knew that the modifications they made to version 1.0 for release 1.0.2 would not be required in version 2.0, so those two lines of development remained separate. However, that's not always going to be true. If a bug is discovered in an earlier version of a product, and if there is a later version of the product that was initially derived from the earlier version, then it's likely the bug exists in the later version as well. The next example illustrates one way to deal with this situation.

Let's say that a bug is discovered in release 1.0.1. It's a bad bug that needs to be fixed ASAP and shipped to customers as a hotfix. Because it occurred in release 1.0.1, the bug is also present not only in release 1.0.2 of version 1.0 of the product, but also in release 2.0 in version 2.0 of the product.

The buggy module of code is identical in both versions, so the fix can be identical as well. However, the fix involves more than a simple line or two of code, so the developers would like to avoid having to apply it manually to both versions. How can they accomplish this?

When the need for a hotfix arises, it's conventional to create a special hotfix branch to isolate the fix from the main line(s) of development. With this in mind, the developers decide to first address the bug in version 1.0 and create a hotfix branch off release 1.0.2, as illustrated in the conceptual diagram in Figure 16.

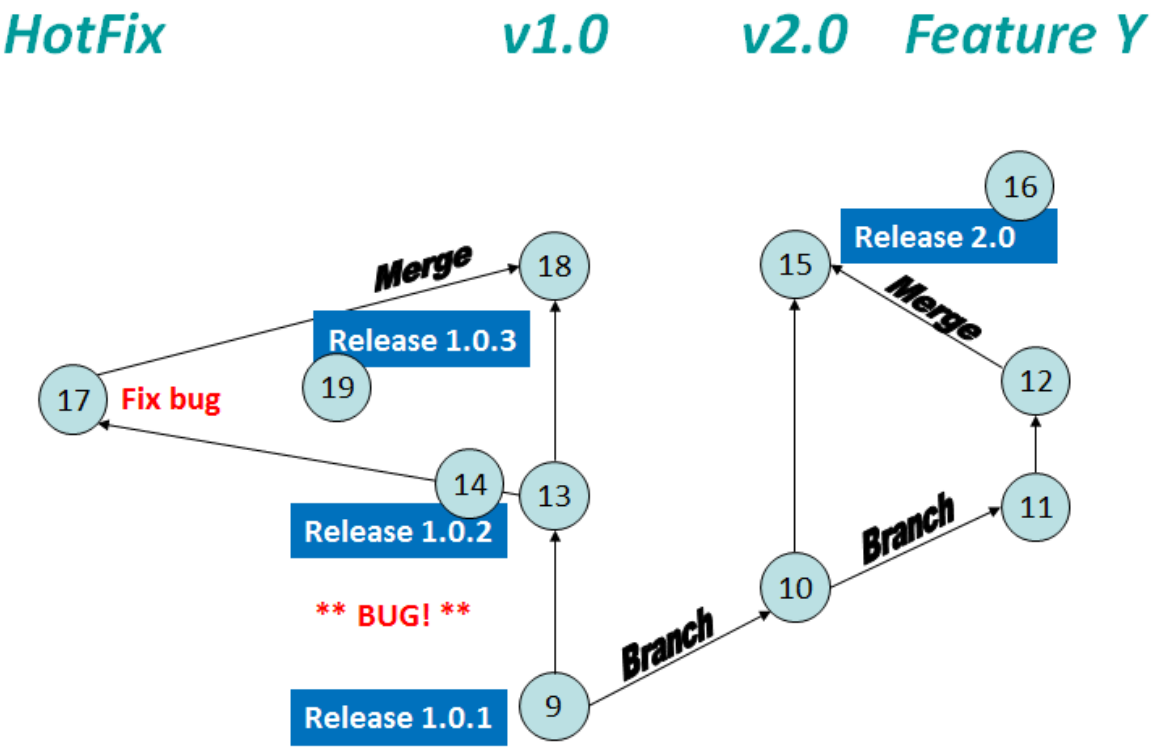
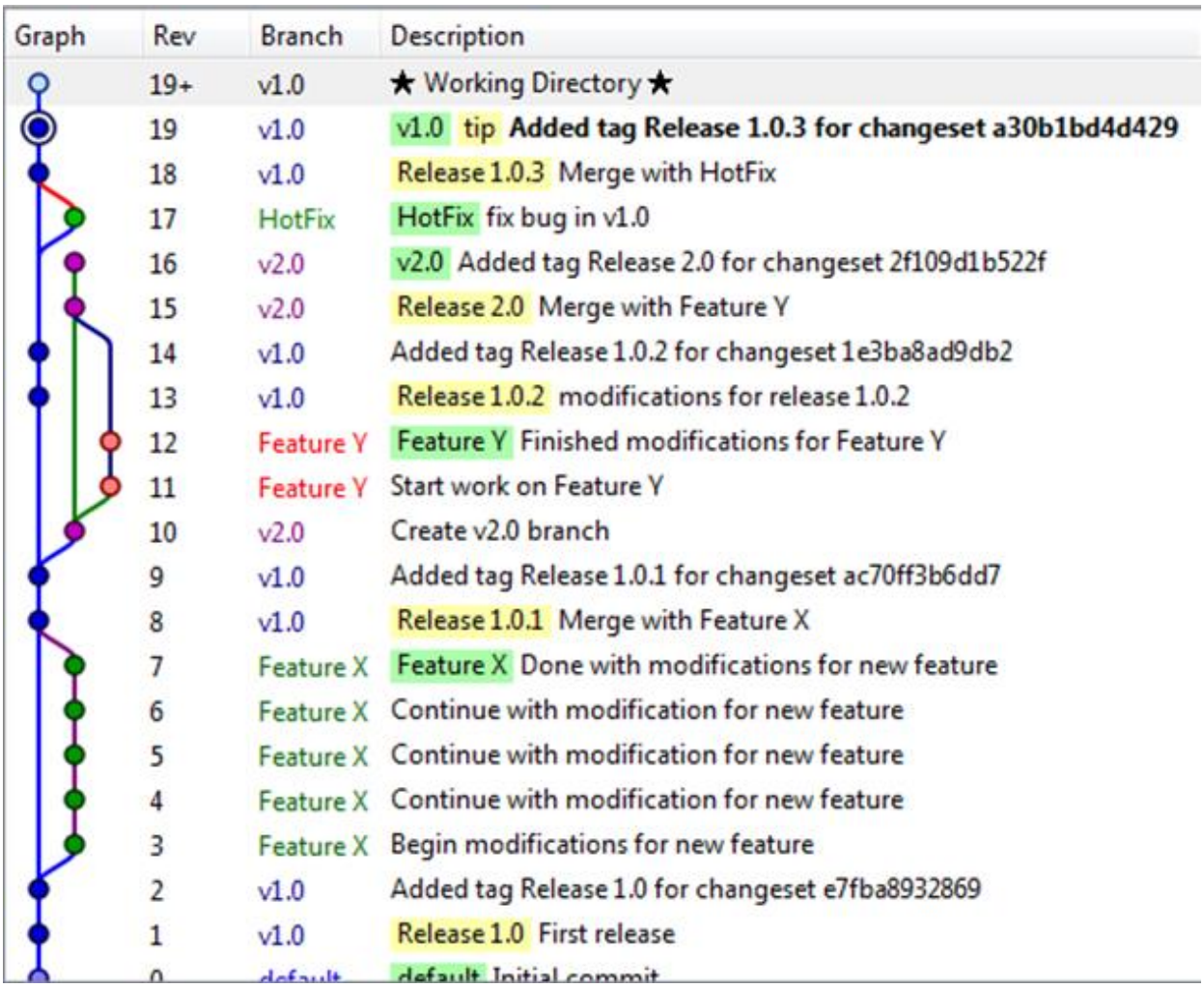


Figure 16: A hotfix branch is created to correct a bug in v1.0. The same hotfix will be needed in v2.0.

The fix is made, tested, approved by QA, and committed on the Hotfix branch, which is then merged back into the v1.0 branch to create release 1.0.3, as shown in step 18. The updated product is released and the customers using version 1.0 are happy again. The revision history at this point is shown in Figure 17.



Graph	Rev	Branch	Description
	19+	v1.0	★ Working Directory ★
	19	v1.0	v1.0 tip **Added tag Release 1.0.3 for changeset a30b1bd4d429**
	18	v1.0	Release 1.0.3 Merge with HotFix
	17	HotFix	HotFix fix bug in v1.0
	16	v2.0	v2.0 Added tag Release 2.0 for changeset 2f109d1b522f
	15	v2.0	Release 2.0 Merge with Feature Y
	14	v1.0	Added tag Release 1.0.2 for changeset 1e3ba8ad9db2
	13	v1.0	Release 1.0.2 modifications for release 1.0.2
	12	Feature Y	Feature Y Finished modifications for Feature Y
	11	Feature Y	Start work on Feature Y
	10	v2.0	Create v2.0 branch
	9	v1.0	Added tag Release 1.0.1 for changeset ac70ff3b6dd7
	8	v1.0	Release 1.0.1 Merge with Feature X
	7	Feature X	Feature X Done with modifications for new feature
	6	Feature X	Continue with modification for new feature
	5	Feature X	Continue with modification for new feature
	4	Feature X	Continue with modification for new feature
	3	Feature X	Begin modifications for new feature
	2	v1.0	Added tag Release 1.0 for changeset e7fba8932869
	1	v1.0	Release 1.0 First release
	0	default	default Initial commit

Figure 17:

This leaves version 2.0, where the hotfix has not yet been applied. Because the buggy piece of code was identical in version 1.0 and 2.0, the modifications recorded in revision 17 on the Hotfix branch are exactly what's needed for version 2.0 as well. But how do the developers apply those modifications to that version?

Mercurial provides a feature called *grafting* for situations like this. Using this feature, the developers can graft the hotfix onto the current head revision of the v2.0 branch. The conceptual diagram for this action is shown in Figure 18.

© 2013 Rick Borup

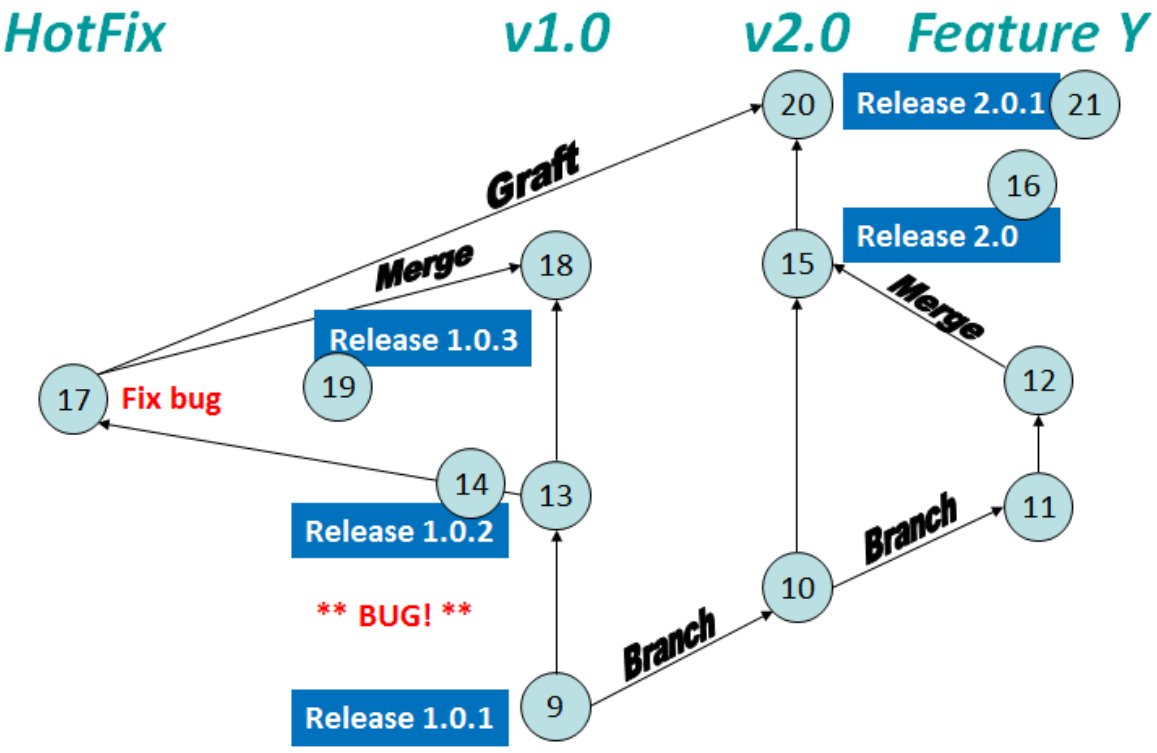


Figure 18: The hotfix from v1.0 is grafted onto the v2.0 branch to correct the same bug in version 2.0.

Once this has been done, version 2.0 now has the hotfix and release 2.0.1 can be built and released. The revision history at this point is shown in Figure 19.

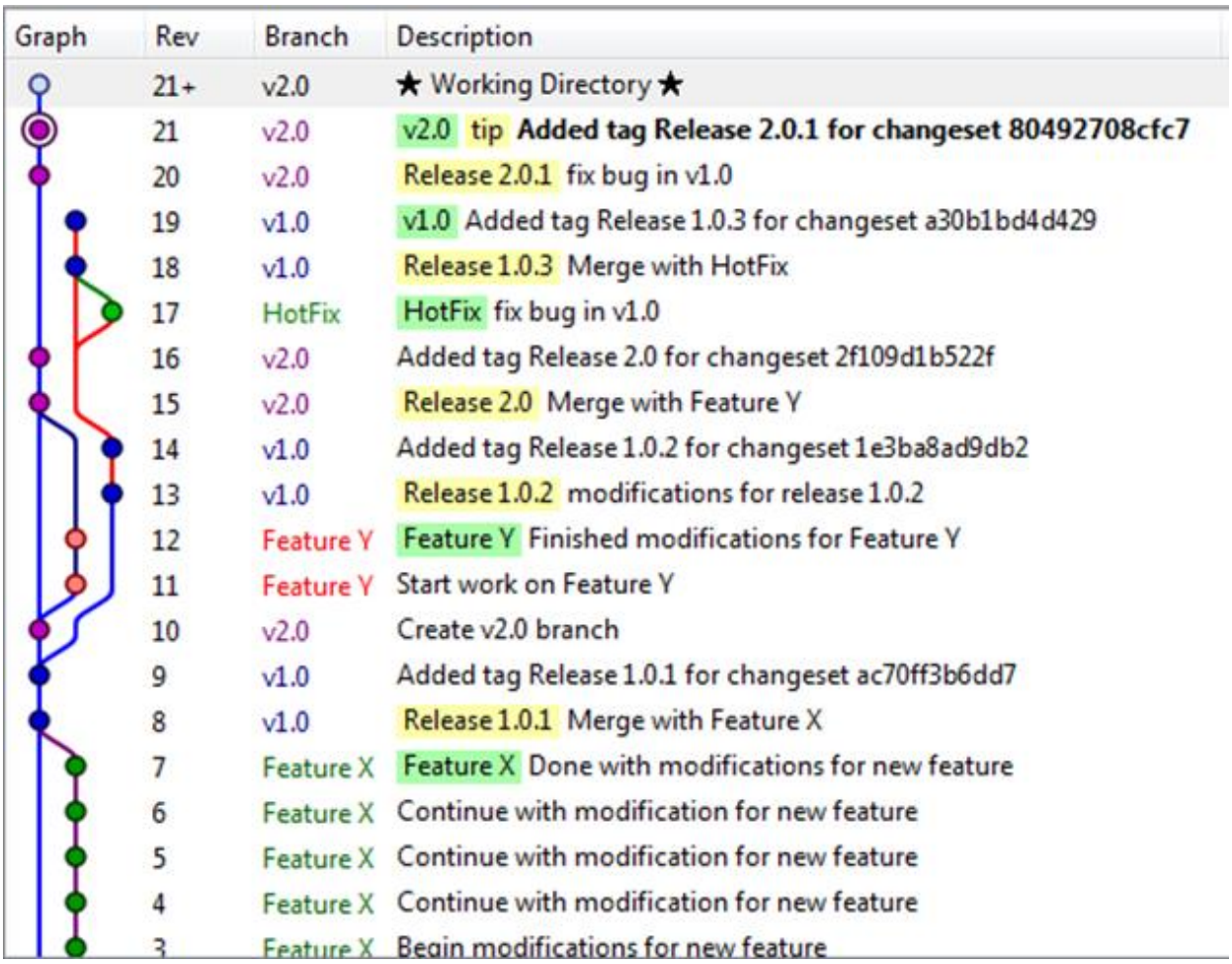


Figure 19: Release 2.0.1 is created after grafting the hotfix onto the v2.0 branch.

Complex strategies

The above examples are but one way to approach multi-track development with a DVCS, and a fairly simple one at that. I've found a few more complex approaches described in books and online resources. One that I found particularly interesting is *A Successful Git Branching Model*, which you can find online at http://nvie.com/posts/a-successful-git-branching-model/. It's based on Git, but the concepts are applicable to Mercurial and other DVCS as well.

My guess is that each development team or corporate department will develop its own approach over time, so there is no one "best" solution.

Mercurial vs. Git

If you're working with remote repositories in a multi-track development environment involving multiple branches, it's worth noting one major difference between the default behavior of Mercurial versus that of Git. When pushing revisions to a remote repository, Git pushes only the *current* branch, while Mercurial pushes *all* branches.

If you want to avoid pushing uncommitted changes from other branches to a remote repository when using Mercurial, you can override the default behavior and push only the desired branch. In TortosieHg, this is done from the context menu on the Push dialog, as illustrated in Figure 20.

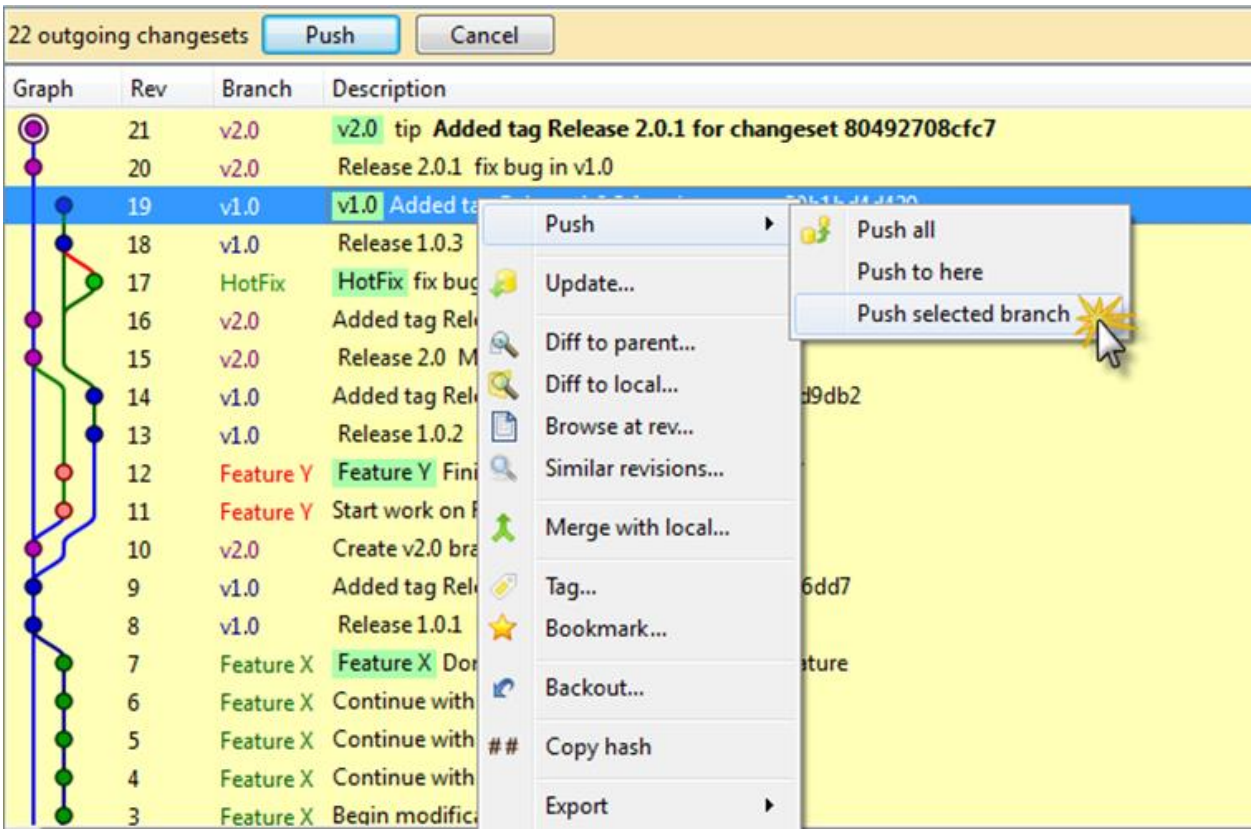


Figure 20: You can override Mercurial's default behavior for pushing branches to a remote repository.

Preparing a VFP Project for DVCS

In order to effectively use a DVCS when more than one branch is involved, you must be able to perform merge operations on the source code files. Merging is the process of integrating the modifications made in one source code file with the code in another version of the same file. Without the ability to merge, branching is essentially useless.

Diff engines

Distributed version control systems use a diff engine to facilitate merging. They're called diff engines because they identify and display the differences between two files. Diff engines are not unique to DVCS – many popular products such as Beyond Compare also ship with a diff engine. The key thing about a diff engine is that it can only operate effectively on text files – that is, files whose content consists only of the standard subset of ASCII characters for letters, numbers, symbols, and punctuation. When you use a diff engine on two text files, you see readable results and can easily tell what's different and therefore what the result of a merge would be.

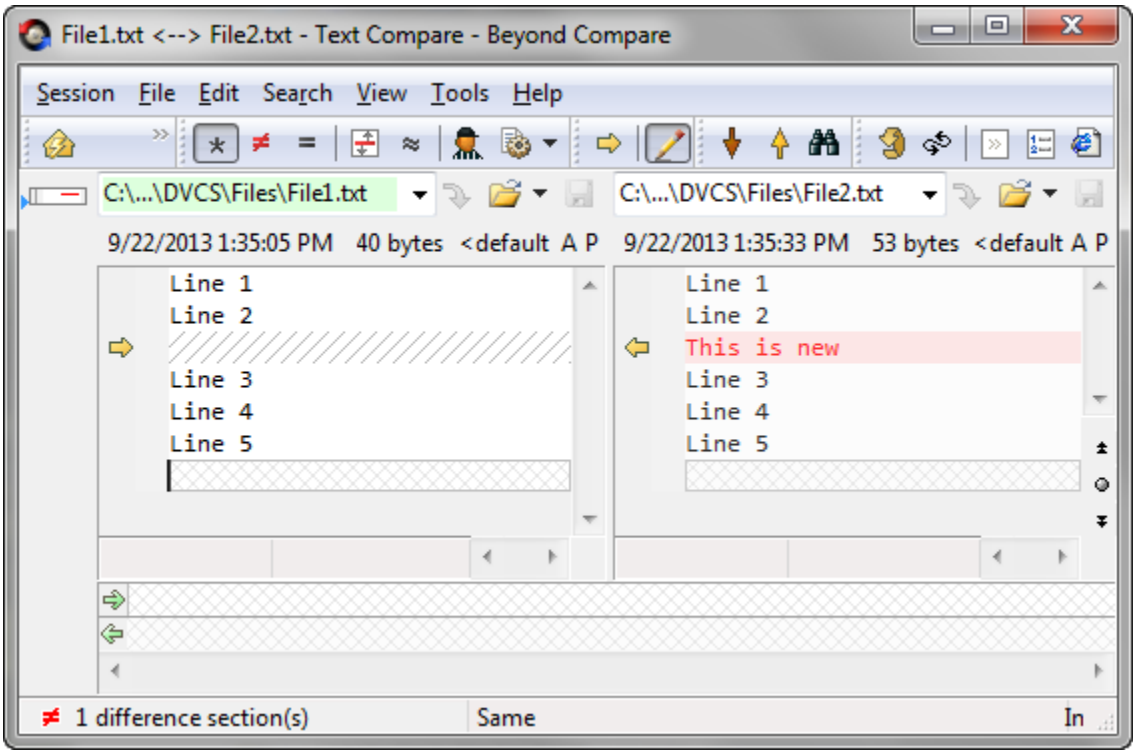


Figure 21: The visual difference between two text files is apparent when viewed in a diff engine such as this one from Beyond Compare.

While you can use a diff engine to compare two binary files, the results are essentially meaningless: a human being cannot glean any useful information about the differences between two binary files and therefore cannot make any intelligent decisions about what a merge would accomplish.

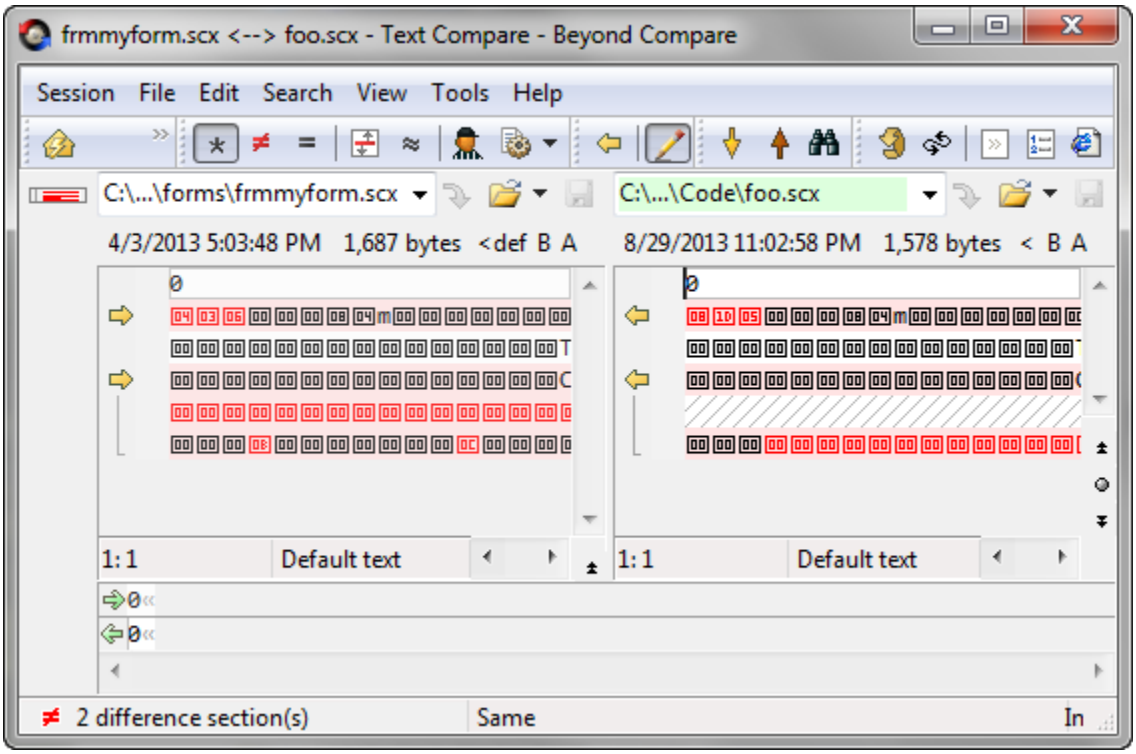


Figure 22: The visual difference between two binary files is meaningless to a human being.

VFP's diff problem

This presents a real problem for Visual FoxPro developers because many files such as class libraries, forms, menus, and reports are store in binary format. Therefore, modifications to a binary file in one branch cannot be directly merged with the file's counterpart in another branch.

The only effective solution is to use a two-way tool that enables conversion of the binary files into a text format and back again. The only tool I'm aware of that does this reliably is Christof Wollenhaupt's TwoFox, which generates XML files from the binaries and then re-creates the binaries from the XML files. XML files are text files, so the diff engine and the merge process work as expected. There are other binary to text tools for VFP but they only work in that one direction, which makes them useless for DVCS branching and merging.

Other issues with VFP

There are other issues developers face when working with VFP projects in a DVCS. Fortunately, steps can be taken in advance to mitigate the problems. For the following discussion, consider a typical VFP project as illustrated in Figure 23. This project has an assortment of files of different types, including text-based program and header files along with binary files for forms, reports, class libraries, and menus.

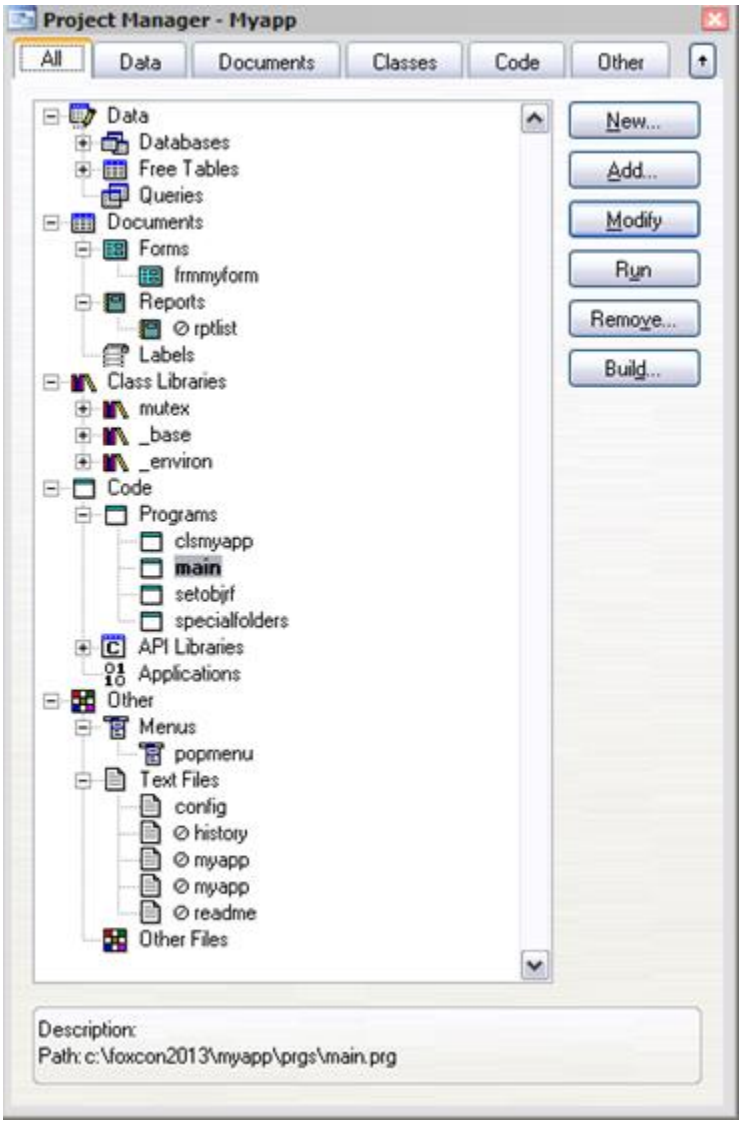


Figure 23: A typical VFP project contains a mixture of text files and binary files.

If we run a query on the project file (which is a standard VFP DBF file), we can see that not everything is as clean as it appears when viewed in the project manager. Figure 24 shows the results of a query on the Name and HomeDir columns, where you can see that not all files are stored in the VFP project folder.

Csrpjxfile

Name	Homedir
C:\FOXCON2013\MYAPP\MYAPP.PJX	c:\foxcon2013\myapp
prgs\main.prg	
..\..\program files\microsoft visual foxpro 9\foxtools.fll	
classes\mutex.vcx	
files\config.fpw	
..\..\..\..\vfptools\wlcprojectbuilder\myprojecthook5.vcx	
prgs\clsmyapp.prg	
forms\frmmyform.scx	
icons\face03.ico	
zfoxuser.dbf	
include\myapp.h	
myapp.ini	
..\..\program files\microsoft visual foxpro 9\ffc_environ.vcx	
prgs\specialfolders.prg	
..\..\program files\microsoft visual foxpro 9\ffc_base.vcx	
..\..\program files\microsoft visual foxpro 9\ffc\setobjrf.prg	
rptlist.frx	
menus\popmenu.mnx	c:\swfox2012\sessions\mercurial\myapp
data\mydata.dbc	
history.txt	
readme.txt	

Figure 24: The VFP project file can be queried to show the project's files and their relative locations.

If you want enable other developers to clone your DVCS repository and work on the app, the repository needs to include all the files required by the app. In a typical VFP app, many of these files are located outside of the root project folder, which means they cannot be included in the local repository.

You might make an exception for some of these files, such as components that ship with VFP itself, under the assumption the person cloning your repository will have their own installation of VFP and therefore also their own copy of those same files. This is a valid assumption up to a point, but consider that the path to Visual FoxPro on a 64-bit machine is different than on a 32-bit machine. That difference breaks the links to those external resources if the repository is created on a 32-bit machine and then cloned onto a 64-bit machine, or vice versa.

The best way I know of to make the repository for a VFP project as complete as possible is to create copies of common files in the VFP project's root folder and its subfolders. To more readily spot the files that may need this kind of special handling in the sample project, I revised the query above and separated the file name from the path. The result is shown in Figure 25.

Csrpjxfile

Filename	Path	Homedir
MYAPP.PJX	C:\FOXCON2013\MYAPP	c:\foxcon2013\myapp
main.prg	prgs	
foxtools.fll	..\..\program files\microsoft visual foxpro 9	
mutex.vcx	classes	
config.fpw	files	
myprojecthook5.vcx	..\..\..\..\vfptools\wlcprojectbuilder	
clsmyapp.prg	prgs	
frmmyform.scx	forms	
face03.ico	icons	
zfoxuser.dbf		
myapp.h	include	
myapp.ini		
_environ.vcx	..\..\program files\microsoft visual foxpro 9\ffc	
specialfolders.prg	prgs	
_base.vcx	..\..\program files\microsoft visual foxpro 9\ffc	
setobjrf.prg	..\..\program files\microsoft visual foxpro 9\ffc	
rptlist.frx		
popmenu.mnx	menus	c:\swfox2012\sessions\mercurial\myapp
mydata.dbc	data	
history.txt		
readme.txt		

Figure 25: This query separates the file names from their paths, making it easy to spot potential problems.

The five FoxPro Foundation Class (FFC) files used in this project are now easy to spot just by looking in the Path column. These files will be problematic if someone clones the repository onto a machine where those files don't exist along the same relative path. This will be true if the repository is created on a 32-bit machine (as shown) and cloned onto a 64-bit machine, in which case the FFC file are in *Program Files (x86)\Microsoft Visual FoxPro 9\ffc.* It will also be true if the relative path to VFP on the target machine does not begin two levels above the cloned project's root folder as it does on the source machine.

How you handle these files is up to you. You may be content to let the person cloning your repository resolve the missing link the first time they open the project, which is one way to do it. However, if you include the project file itself in the repository, as you would normally do, and if modifications to that file get passed back and forth between developers with differing environments, then every developer will need to resolve these same links every time they pull an updated copy. One solution is to copy these files to a common location that will be the same on everybody's machine. They can then be left out of the repository, where they're not really needed because you usually don't modify the FFC classes anyway.

There is one other file in this sample project that's problematic, but it's already been handled in Figure 25. The *face01.ico* icon file originally came from the *Program Files\Microsoft Visual FoxPro 9\Graphics\Icons\Misc* folder. In order to resolve a potential missing link issue in a cloned repository, I copied that file into the *icons* subfolder under the project's root and included it in the repository. The file therefore exists in the same relative location regardless of where the clone is created.

Reliable workflow

Working with any version control system introduces additional steps into your daily development process. In addition to the usual modify / build / test cycle, you now also need to include commits to your local repository and push those revisions to the remote repository (assuming one is being used). If you're a member of a team with other developers who are working on the same project, you also need to periodically pull their changes into your local repository. If any of these steps are omitted, you may cause problems for yourself or for others on your team. Therefore it's important to develop and adhere to a reliable workflow.

Changes you make

The changes you make need to be periodically committed to your local repository and pushed to the remote repository. How often you do this is either a matter of personal preference or guided by team or company policy. The basic workflow is the same regardless of the frequency, as illustrated in Figure 26.

Note that working with VFP introduces the "convert binary to text" step into the standard change / commit / push cycle. In order to enable future merging of changes, you need to include the text equivalent of all binary source files in the repository.[2]

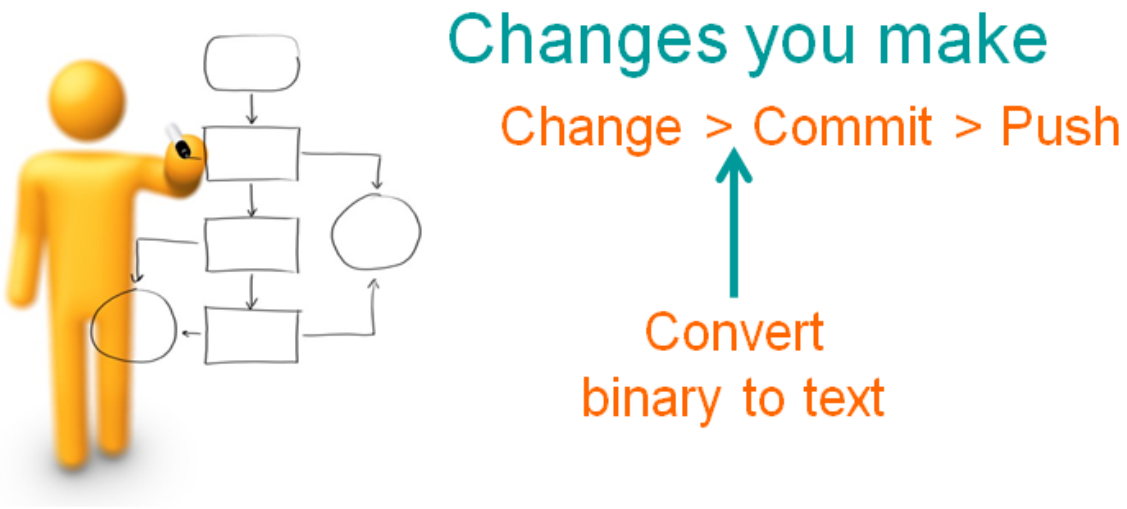


Figure 26: The standard change / commit / push workflow for changes you make also requires a "convert binary to text" step when working with VFP projects.

[2] It's a matter for debate whether you also include the binary files themselves. Once you have confidence in the binary –> text –> binary two-way conversion tool, it makes sense to omit the binaries and include only the text files. However, you may want to continue including the binary files as a backup in case the conversion to and from text files doesn't always work as expected.

Changes others make

The changes other developers make to a project you're working on need to be periodically pulled into your local repository. How often you do this depends on the situation, but the basic workflow for pulling changes is also the same regardless of frequency. This workflow is illustrated in Figure 27.

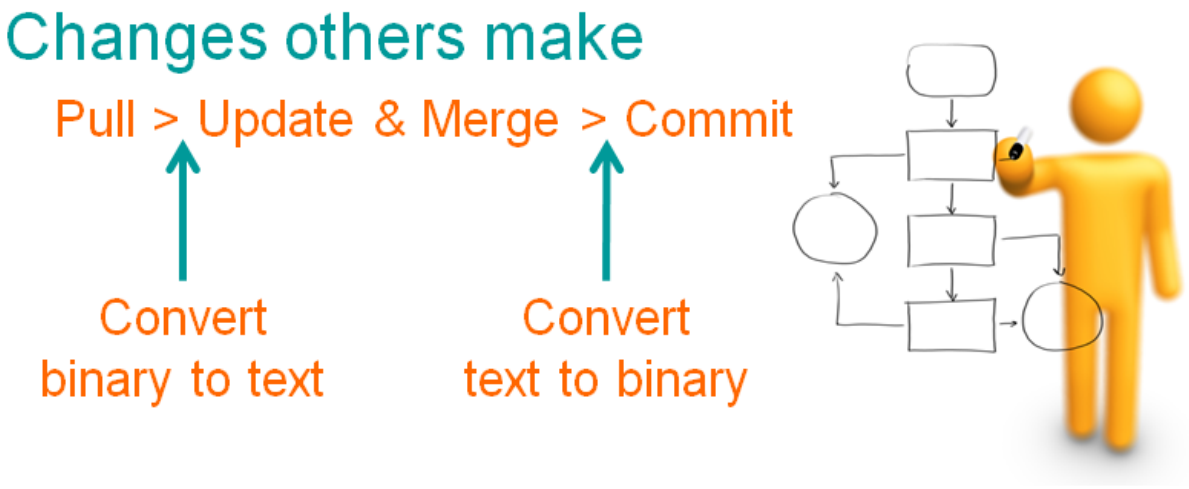


Figure 27: The standard workflow for integrating changes other developers have made also includes some special considerations for VFP binary to text and text to binary conversions.

The basic steps are to 1) pull revisions from the remote repository, 2) update your local repository with those revisions, merging where necessary, and 3) commit the updated and merged results back to your local repository.

Working with VFP again introduces a couple of extra steps that are required to deal with binary source code files and their equivalent text files. If you include the binary files in the repository and pull them into your local repository, you may want to add a "convert binary to text" step after the pull, although this is not required if you're also pulling the updated text files. In any case, you'll want to add the "convert text to binary" step after updating and merging the text files because you need the binary versions in order to work on them in the VFP IDE.

After completing this part of the workflow you're ready to begin making changes of your own. This takes you back to the start of the "Changes you make" workflow shown in Figure 26, and the cycle begins again.

HgTools

I've been working on a tool called HgTools that integrates these steps into the VFP IDE in order to help maintain a reliable workflow when using Mercurial. HgTools is based on a tool Toni Feltman developed a few years ago for working with Subversion in VFP. Like Toni's tool, HgTools uses Christof's TwoFox for the binary to text to binary conversion. I'm

not sure how many VFP developers are using Mercurial, but if there's enough interest in HgTools I'd consider making it a VFPX project. Contact me if you're interested.

Summary

Distributed version control systems are popular for many reasons, one being that they make it easier to manage multi-track development projects. This is true both for solo developers and for teams. Using a DVCS for multi-track development requires learning the concepts behind branching and merging, and how they're implemented in your DVCS of choice. It also requires some forethought and planning about when to create branches, how many to use, and for what purposes. Visual FoxPro developers face some special challenges due to binary source code files, but these can be managed. Finally, it's important to adhere to a reliable workflow to insure that no steps are omitted and the integrity of the repository remains intact.

Copyright © 2013 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.