Lessons Learned in Version Control

Doug Hennig
Stonefield Software Inc.
Email: dhennig@stonefield.com
Corporate Web sites: www.stonefieldquery.com
www.stonefieldsoftware.com
Personal Web site : www.DougHennig.com
Blog: DougHennig.BlogSpot.com
Twitter: DougHennig

Rick Borup's sessions at previous Southwest Fox conferences really helped me get started using version control for my application development. Since then, I've made a ton of mistakes and learned a lot of lessons the hard way. This document discusses what has evolved into my team's best practices for version control. It's intended for those who are familiar with version control but are looking for ideas about how to improve their processes.

Introduction

Before Rick Borup's "VFP Version Control with Mercurial" session at Southwest Fox 2011, I'd heard all the arguments for using version control but hadn't adopted it. My thinking was that I'm the only one working on my projects so I don't need version control. Rick's session convinced me I was wrong. I implemented Mercurial shortly after, with tremendous help from Rick's white paper, and haven't looked back. I now use version control for more than just application development. For example, the tables Tamar, Rick, and I use to manage speakers, topics, and scheduling for Southwest Fox and Southwest Xbase++ are in version control.

I'm not going to try to sell you on using version control. I'm also not going to provide an introduction to version control in general or certain version control tools in particular; I assume you've already been using version control or at least have enough familiarity with it to follow along. What this document is about is discussing the processes my team at Stonefield Software use and some of the traps we've run into along the way. I don't claim that these are best practices for everyone; they're processes that work for us. Feel free to adopt and adapt them as you see fit.

The examples discussed in this document use Mercurial but the concepts are almost identical in other distributed version control systems (DVCS) like Git.

Getting started

I've started a new application that both I and my wife Peggy will be working on (she's not actually a software developer; I just wanted to use her name in my examples). So far, the application just consists of a main PRG that runs a form, which just has a single button so far. The form and its controls are instances of classes in my base class library, which is in a folder outside the project folder structure called Framework.

The first step is to put the source for the application into version control (actually, the first step is to install a version control system on both machines, but I assume that's already done). Right-click the project folder, choose TortoiseHG, and select Create Repository Here (**Figure 1**). You can just accept the defaults in the dialog that appears and click the Create button. You'll see a new .hg folder and new .hgignore file created, and the project folder has a green checkmark indicating both that it's under version control and that there are no changes (in this case, because we haven't added anything to the repository yet).

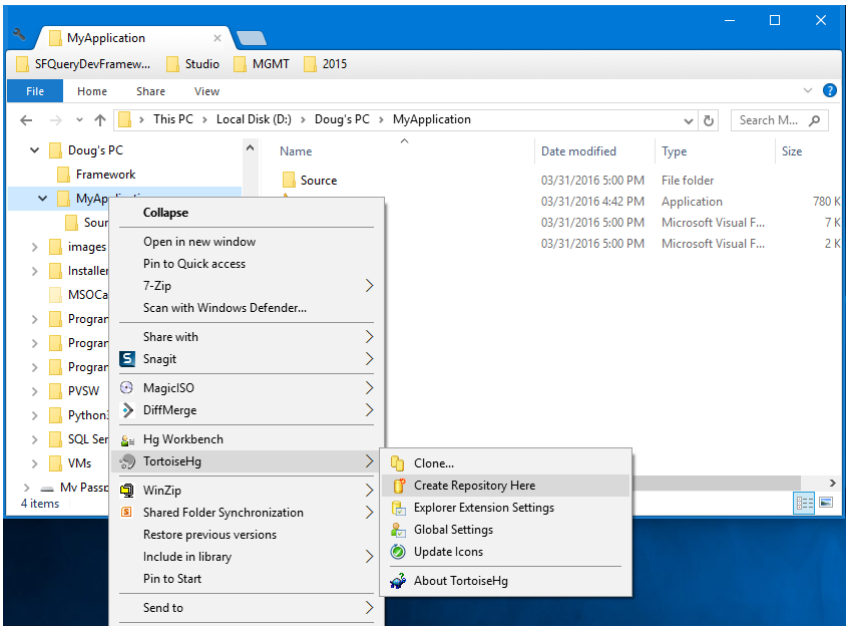


Figure 1. Creating a repository.

Right-click the folder again and choose Hg Commit. Select all files by clicking the checkbox at the top, enter a commit message (it's traditional to use "Initial commit" for the first one), and click the Commit button (**Figure 2**). Click the Add button when asked if you want to add selected untracked files. All project files now display a green checkmark.

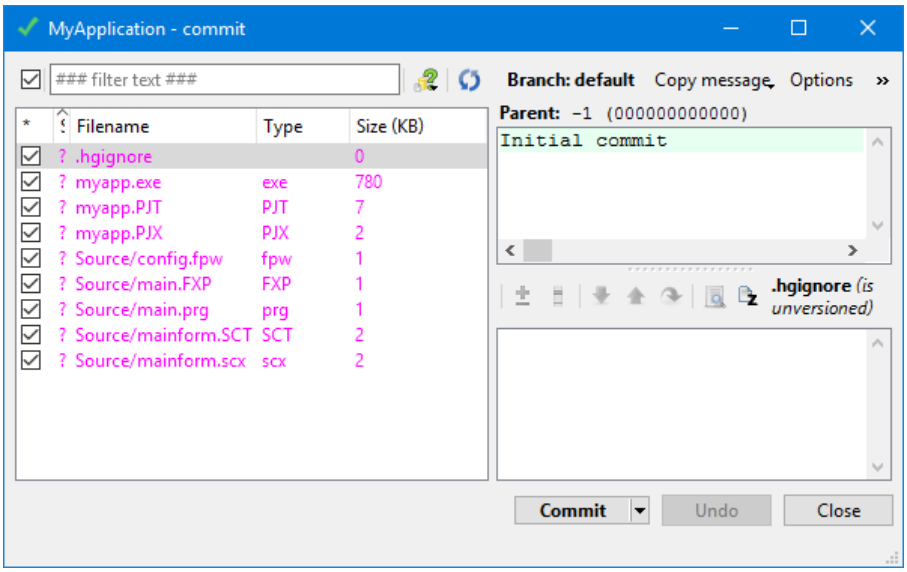


Figure 2. The initial commit for the application.

Next we need to clone our local repository to a central one where other developers can access it. For now, we'll do this on a server but later we'll use a cloud site. Create a folder on the server, right-click it, choose TortoiseHg, and select Clone (**Figure 3**). Click the Browse button for Source and select the application folder, then click Clone.

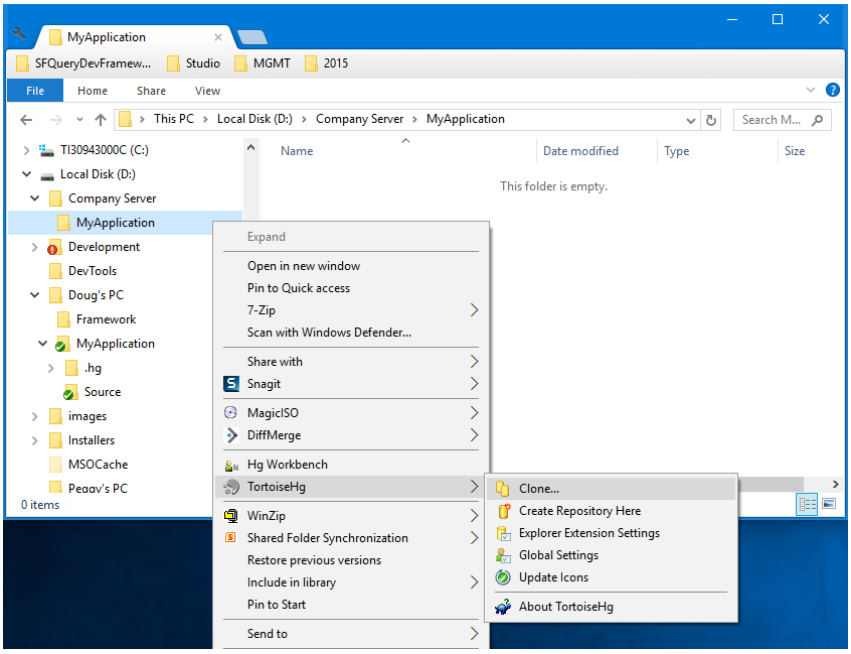


Figure 3. Cloning a repository on the server.

Finally, to create a copy on Peggy's machine, repeat the steps to clone the repository on the server but use the server repository as the source.

Let's see if everything worked. Open the MyApp project on Peggy's machine. Oops, we're getting a warning that SFCtrls.vcx can't be found. That makes sense: the repository only contains files from the MyApplication folder, not from Framework, so when we cloned the repository on the server and on Peggy's machine, the files in Framework weren't included.

That brings up the first topic: how we deal with common components.

Handling common components

Rarely do you start an application completely from scratch. Typically, you use a framework of some type, whether it's a commercial product such as Visual FoxExpress or Visual MaxFrame or a home-grown version like I use. You probably use other components too, again either commercial products, such as West Wind Client Tools or Stonefield Database Toolkit, or open source, such as VFPX projects or FoxyPreviewer.

Usually, rather than having separate copies of these components for every application, developers install these components in directories on their machines and then reference the components from those folders in their projects. That's what I did with MainForm.scx: it's a subclass of SFFormTLF in SFCtrls.vcx, located not in the project folder hierarchy but in a folder called Framework that's referenced by all projects.

The problem with that approach is it prevents you from including those common components used by a project in the project's repository: all the files in a repository must exist in the repository's folder hierarchy. There are a few solutions to this:

- Install the common components on each machine.
- Have copies of the component files in each project folder.
- Have a repository at a level that includes both the project and the common components folders.
- Have separate repositories for your project and common components.

Let's look at each of these options in more detail.

Install common components on each machine

For common components that weren't created in-house (commercial or open source), you could argue that you don't need to include those files in any repository. Instead, just install them on each developer's system. Of course, you need to ensure the same relative folder structure is used on each machine or you'll have issues with components not being found even though they're installed.

The pros of this approach are:

- It's straightforward: your team simply has to worry about source code under your control, not that created by other developers.
- The repository is small since it doesn't contain anything outside the project-specific files.

The cons for this approach are:

- It doesn't help with common components that were created in-house.
- It's more work setting up a new machine: instead of just cloning a repository, you have to install numerous other components the project depends on.
- There could be version problems. For example, if one developer upgrades to a new version of a component but the others don't, the application may not run properly. Everyone has to keep all components in sync with each other.

Copies of the component files in each project folder

In each project's folder structure, include one or more folders containing the common components. You could dump them all into a single folder, even the same folder as your project files, but that would be messy and difficult to manage. A better solution is to duplicate the folder structure you normally keep the components in within the project's folder hierarchy. For example, suppose you have a folder structure like shown in **Figure 4**.

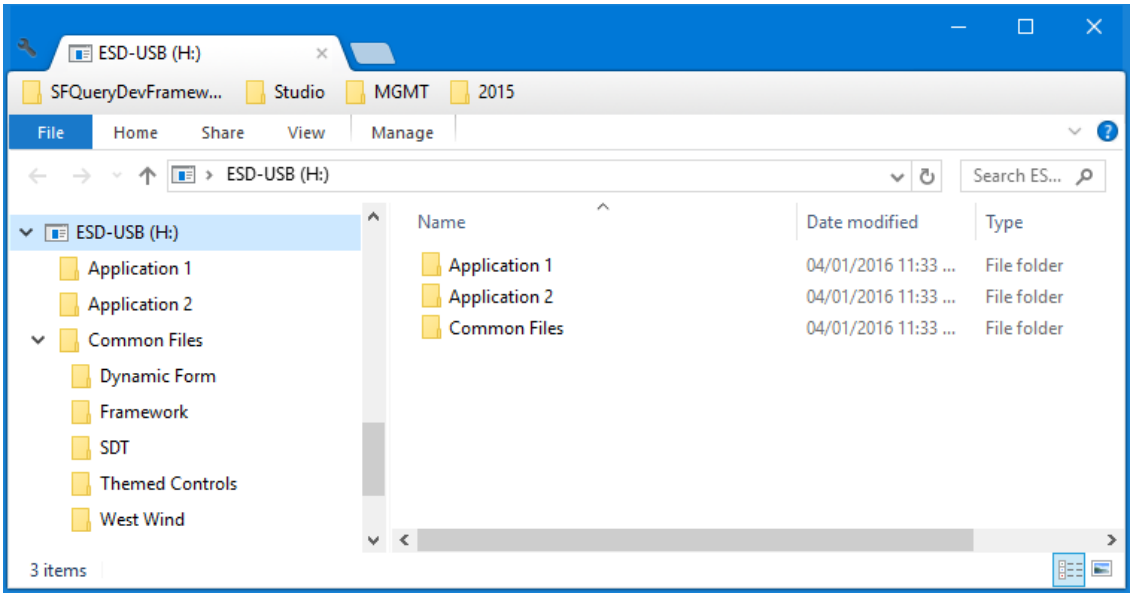


Figure 4. A folder structure with two projects and common components.

In that case, your project folder structure would look like **Figure 5** (Source is the folder containing the project-specific files). Note that Application 1 and Application 2 are under version control but Common Files is not (although it could be if desired).

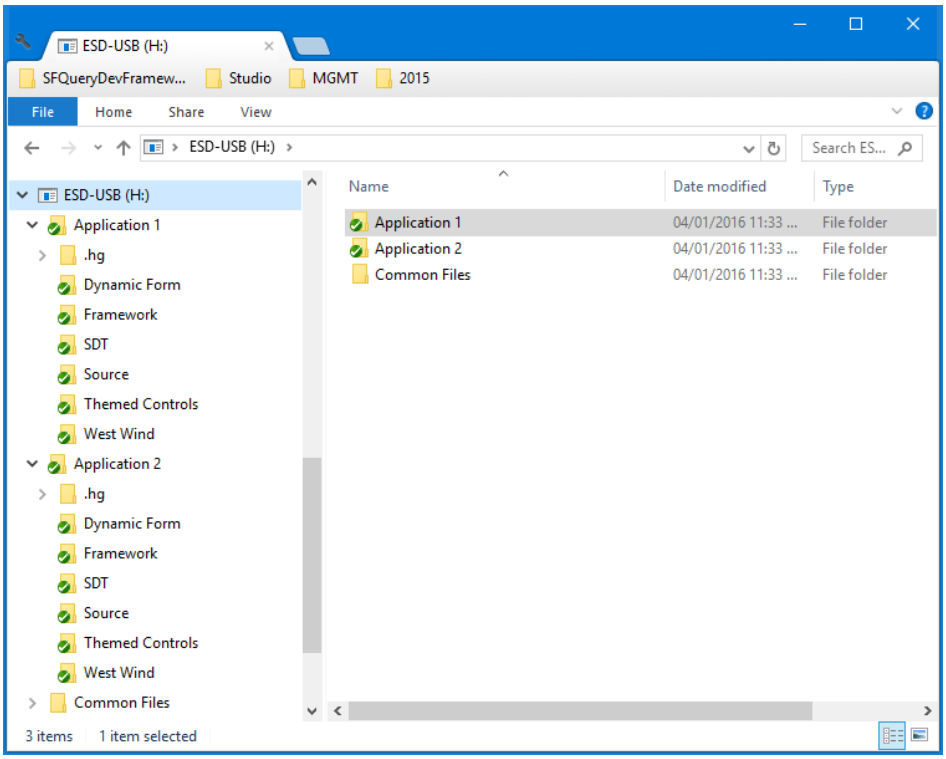


Figure 5. Copies of common components go in each project's folder hierarchy.

(Of course, rather than having the folder for each component in the root of the project folder, you could have them under a Common Files folder.)

The project-specific code references the copies of the common components from within the project hierarchy rather than from the main Common Files folders and those files are included in the repository for each project.

The pros for this approach are:

- Each project stands alone with no references to anything outside the project's folder hierarchy.
- Cloning the repository is easy since it's self-contained.
- Setting up a new project is easy: simply copy the folders within Common Files the project uses into the same named subdirectories of the project folder.
- The project is insulated from changes to common components that may break the application. For example, maybe an update to a West Wind component means changing code that calls the component. Unless you upgrade a project's copy of the West Wind file, there's no need to do anything.

The cons for this approach are:

- It takes more disk space to have multiple copies. Of course, disk space is cheap and plentiful these days so this isn't much of an issue.
- Managing changes to common components is more work: you have to install updates or make changes in each of the folders where the components were copied.
- If you already have a project with the hierarchy shown in Figure 4, this is difficult to implement because all references to the common components have to change. In the case of classes and forms, this means hacking VCX and SCX files to change the path to the parent classes. For other types of code, such as PRGs, it means changing the path to the files (if you're lucky, as simple as changing a SET PATH statement).

Repository in parent of both project and common components folders

Assuming the folder structure shown in Figure 4, the repository would go at the root of the drive. In that case, there's only one copy of each common component file and the project-specific files reference the common component files in their original folders. **Figure 6** shows what the folder hierarchy looks like. Notice the .hg folder is in the root and all subdirectories are included in the repository.

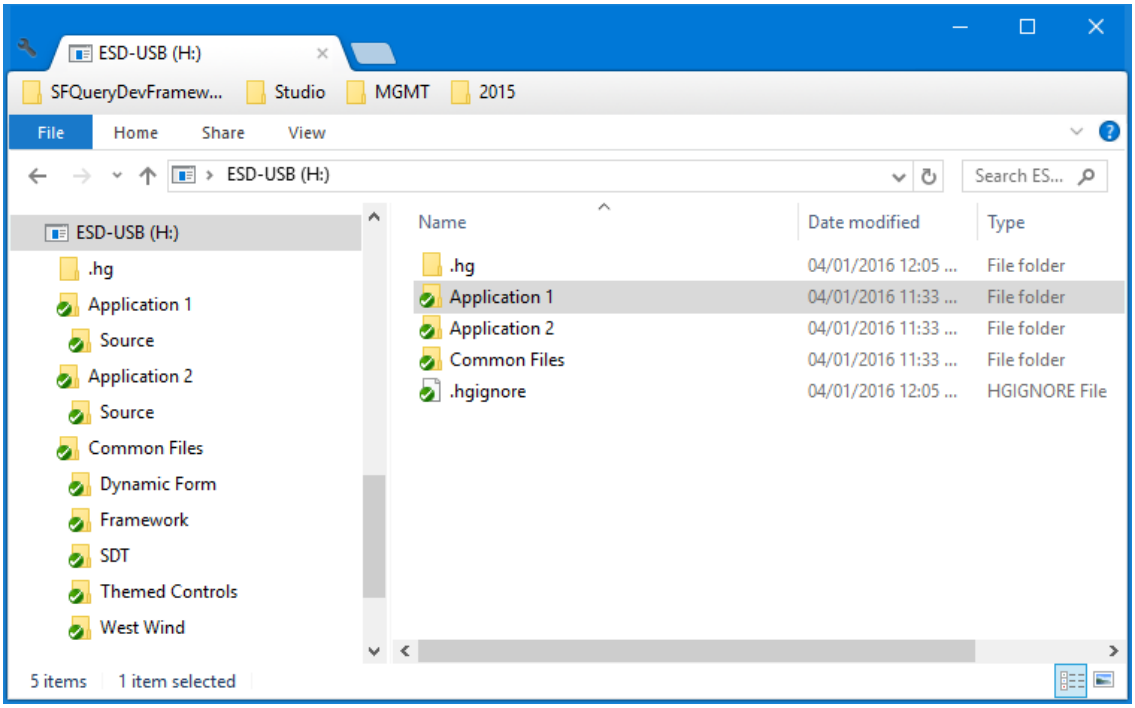


Figure 6. There's a single repository in the parent of both project and common components folders.

The pros for this approach are:

- You don't have to change anything if you're putting an existing project under version control.
- Cloning the repository is easy since it contains everything a developer would need.
- There's only one copy of each common component file, so there are no issues related to disk space or updates.

The con for this approach is that since the repository is in the parent of both project and common components folders, you can only have one repository that includes the common components. That means if you have multiple projects, they're all in one big repository. Although there's nothing strictly wrong with that, I don't think anyone would argue that it's a best practice. This really only works best when you have a single application, such as if you have a single vertical market application and that's all you do.

Separate repositories for projects and common components

As you can see in **Figure 7**, there are separate repositories for each project and for common components. In this case, there's a single repository for all common components but there could be one for each component or combinations thereof.

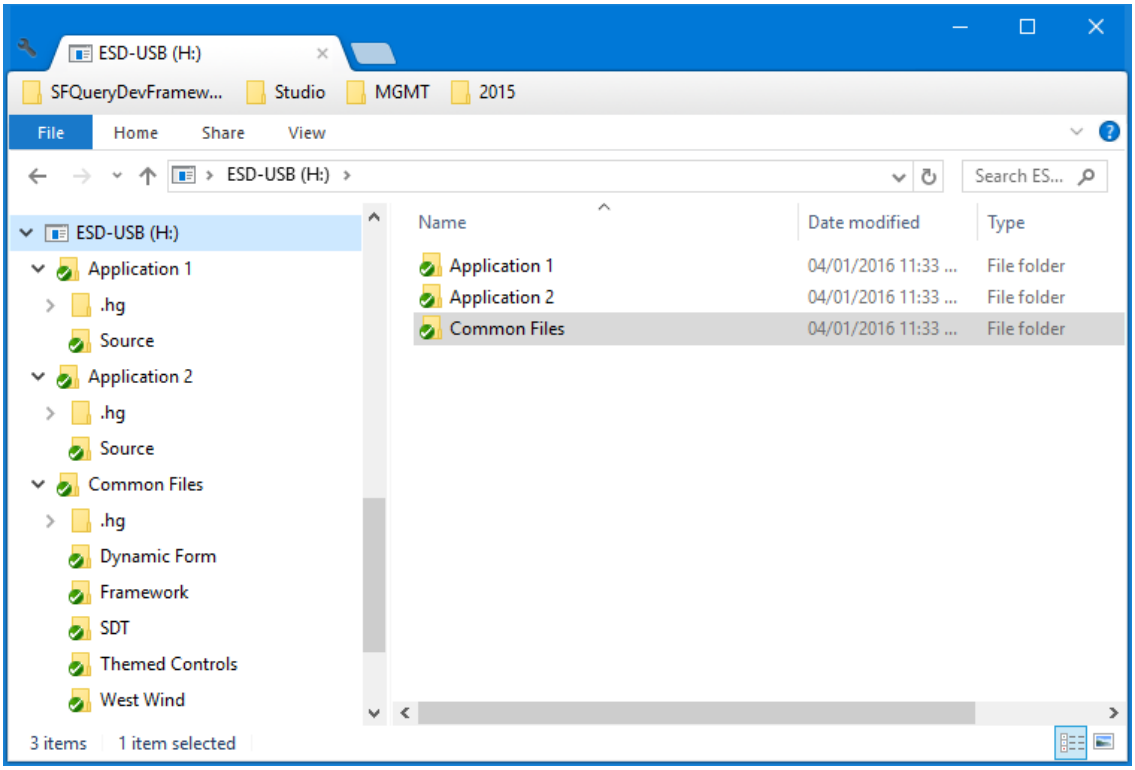


Figure 7. Projects and common components have their own repositories.

The pros for this approach are:

- You don't have to change anything if you're putting an existing project under version control.
- There's only one copy of each common component file, so there are no issues related to disk space or updates.

The con for this approach is a minor one: cloning and keeping a project current is a little more work because there are multiple repositories involved. However, you can automate that by creating batch files or PowerShell scripts that use command line arguments for your version control software to do whatever is necessary. For example, the following commands pull any changes made to the Framework repository (which presumably contains all common framework components) on the server and updates the working directory with those changes:

```
cd \Development\Framework
hg pull \\MyServer\Framework
hg update
```

For our development, Peggy and I decide to go with the fourth approach, so I create a repository for the Framework folder, commit the files to it, and clone it on the server, and Peggy clones it. Peggy can now open, build, and run the project with no changes.

Dealing with VFP binary files

Now that we have version control set up, we're ready to start working on the application. We need to add buttons to the form and code in the Click method of those buttons. However, if we both work on the same form, we'll have a big issue: how to consolidate our changes?

As I'm sure you're aware, some VFP files are actually tables: VCX, SCX, PJX, MNX, FRX, LBX, and DBC (along with their accompanying memo files). Since VFP tables are binary files, they can't be compared to each other using traditional "diff" tools. As a result, you can't compare changes in two copies of a file, nor can you merge changes.

Fortunately, there are several tools available that convert VFP binary files to text files that can be compared and merged. The latest one is FoxBin2Prg, a VFPX project. There's even an extension for it, Bin 2 Text Extension, also a VFPX project, which adds support for Git and IDE integration.

Installing FoxBin2Prg is easy: simply download it from VFPX (http://vfpx.codeplex.com) and install it in any folder. If you use Thor (the most important add-on there is for VFP and also a VFPX project), it's even easier: from the Thor menu, choose Check for Updates, turn on FoxBin2Prg in the dialog that appears, and click Install Updates. I'm going to assume you're using Thor for the purposes of this document. If not, see the FoxBin2Prg documentation for instructions on how to use it.

To create text versions of all of the binary files in a project, choose Thor Tools, Applications, FoxBin2Prg, Projects, Convert All Binary Files to Text Files from the VFP system menu. This function creates text files from the binary files, using the same file name as the binary files but with "2" replacing the final "x" in the extension (for example, MyApp.pj2 for the text version of the PJX file). A similar function does the opposite: generating binary files from the text files.

To make it easier to access the functions, you can create a Thor hotkey for that using the Thor Configuration dialog.

I create a little Thor tool that both creates text versions of all binary files in a project and brings up the TortoiseHg Commit dialog, saving a few mouse clicks. I assigned a hotkey of Alt-3 to the tool so when I'm ready to commit changes in the application, I simply hit Alt-3, enter a comment, and click Commit. Creating a Thor tool is beyond the scope of this document, but all it really does is:

```
do (_screen.cThorFolder + ;
   'Tools\Thor_Tool_Repository_FoxBin2PrgConvertProjectToText.prg')
run /n0 thg commit
```

Thor_Tool_Repository_FoxBin2PrgConvertProjectToText.prg is the Thor tool FoxBin2PRG installs to convert the binary files in a project to text files.

To add my tool to Thor, simply copy Thor_Tool_Convert2PRGAndCommit.prg (included in the downloads for this document) to the Thor\Tools\My Tools folder of wherever you have Thor installed. Choose Configure from the Thor menu and assign a hotkey to this tool if you wish.

It's important to generate text files for the binaries in the project, add them to version control, push them to the central repository, and pull them into all developers' repositories only when all other changes have been synchronized in all repositories. Otherwise, merging the changes can be a mess.

To see this in action, I generate text files for the binaries in the project, add them to the repository, and push to the server; Peggy pulls from the server and updates so she has the latest version. Then I modify the code in the Click method of the existing button in the form while Peggy adds another button to the same form. I commit and push, then she commits, pulls, and merges with her local changes. Because we both made changes in the same form, there are several merge conflicts that need to be resolved (**Figure 8**). Tool Resolve handles the differences in MainForm.sc2 but it can't resolve the differences in MainForm.scx and sct because they're binary files. In this case, it doesn't matter whether we choose Take Local or Take Other since we're going to regenerate the two files from MainForm.sc2 in a moment.

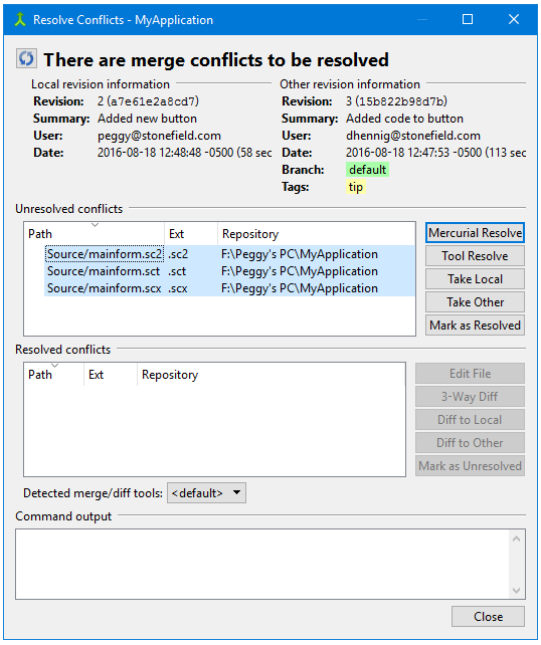


Figure 8. The Resolve Conflicts dialog shows which files have conflicting changes that need to be resolved.

After the update, Peggy uses the hotkey she assigned to generate binary files from the text files. When she then edits MainForm.scx, she sees both of our changes.

Always generate binary files from the text files after a version control update or merge.

One of the things you'll note is that when I open the project, I'm prompted to change the home directory. After I push changes and Peggy pulls them, she's prompted to change the home directory when she opens the project. This gets a little annoying after a while, so we'll see how to resolve this in the next section.

What should go in the repository?

If you look at the list of files added to the repository in Figure 2, you'll notice I added everything in the project folder hierarchy, including FXP and EXE files. Is that necessary or desirable? Not really:

- The larger the repository, the longer it takes to clone, commit, push, pull, etc.
- Even if you don't change the source code, rebuilding a project may regenerate FXP and other generated files. As a result, it looks like you have to commit and push and other developers have to pull and merge even though nothing's really changed.

The general rule of thumb is that nothing generated should go in the repository. In the case of a VFP application, that means you should exclude:

- EXE, APP, and DLL files generated by your projects
- FXP, MPX, and QPX compiled program files
- MPR and QPR files since they're generated from menus and queries, respectively
- ERR, BAK, and TBK files (you probably shouldn't have any of these anyway ☺)
- Any files generated by running your application (TXT, HTML, XML, MEM, etc.)

Other files require some consideration.

- VFP binary files: assuming you're using FoxBin2Prg or something similar as discussed in the previous section, you can exclude the binary files. One benefit of this is that you're no longer prompted to change the home directory for the project when you open it, since you're recreating it with the current directory when you use FoxBin2PRG to generate the binary files.
- Data files: it depends: if they're used but not maintained by the application—for example, meta data such as Stonefield Database Toolkit's, lookup data, configuration settings not maintained by the user, etc.—definitely (don't forget to include any CDX or FPT files in addition to the DBF). Otherwise, probably not, or else every time you add a record to a sample copy of the tables maintained by the application, they're marked as changed and have to be committed, pushed, pulled, merged, etc. Database container files (DBC, DCX, and DCT) should be included unless you use GENDBC to

generate a program to recreate the database; in that case, include that generated program. You might want to use FoxBin2Prg to convert tables and database containers to text files and include those instead. FoxBin2PRG doesn't handle data files automatically; see its documentation for details on how to configure it to do so.

- Images: these should be included but keep in mind that they're binary files so changes made by more than one person at a time can't be merged. Fortunately, most developers I know get their images from other sources rather than creating them themselves so this is rarely an issue.
- CHM (help) file: there's no need to include the CHM file (or CHW file if it exists) since it can be regenerated from whatever source you used to create it. If you're using West Wind HTML Help Builder, you should include the HBP, FPT, and CDX files for the help source as well as WWHelp.ini and the contents of the BMP, Images, and Templates folders (you can actually skip the BMP and Templates folders since they're created when you create a new help project and don't change, but I include them to make it easier to clone the repository). The rest of the files are generated by Help Builder and so should be excluded. Note that since the HBP file is actually a VFP table, you might want to use FoxBin2Prg to convert it to a text file and include the text file instead.
- Testing files: this includes FoxUnit or other test programs, any configuration settings, and test data. These should be included in the repository. As with the other points, consider using FoxBin2Prg to create text files from any binary files and include those instead.
- Documentation: this could be Microsoft Word documents describing the feature set, Microsoft Excel documents detailing the development plans and costs, and so on. Note that these are binary files so you won't be able to merge changes. For that reason, it's best if only one person be responsible for editing these files.
- Deployment files: your setup source files (for example, the ISS file if you use Inno Setup) as well as any other files related to deployment (such as BAT or PowerShell PS1 files) should be included.
- Visual Studio files: if you're like me and have some in-house .NET components in your applications, you should include the source files for those components in the repository. You can exclude the generated files (those in the BIN and OBJ folders) and SUO (user preference settings) file.
- hgignore or gitignore file: these files contain information about which files the version control system should ignore. They should be included in the repository even though they're not strictly part of the project.

Ignore file

Speaking of the ignore file, you should use one to tell your version control software not to list certain files in the commit dialog. Otherwise, it's difficult to tell which are files you routinely exclude and which are new but haven't been added yet. See https://www.selenic.com/mercurial/hgignore.5.html for a discussion of .hgignore for

Mercurial. **Listing 1** has the content of a typical .hgignore file for a VFP application (this file is included with the files accompanying this document). Note that .hgignore is case-sensitive so file name and extensions are usually listed in multiple cases unless you use regular expression syntax.

Listing 1. Typical .hgignore file for a VFP application.

```
# use glob syntax.
syntax: glob

# VFP generated files
*.err
*.ERR
*.qpr
*.QPR
*.mpr
*.MPR

# VFP compiled files
*.fxp
*.FXP
*.mpx
*.MPX
*.qpx
*.QPX

# VFP backup files
*.bak
*.BAK
*.tbk
*.TBK

# VFP binary files
*.pjx
*.PJX
*.pjt
*.PJT
*.vcx
*.VCX
*.vct
*.VCT
*.scx
*.SCX
*.sct
*.SCT
*.mnx
*.MNX
*.mnt
*.MNT
*.frx
*.FRX
*.frt
*.FRT
*.lbx
```

```
*.LBX
*.lbt
*.LBT

# Misc backup files
*.orig
*.ORIG
*.save
*.SAVE
*.zip
*.ZIP

# Other files
*.chw
*.suo
*.docstates

# Folders to exclude
bin/*
obj/*

# Specific files
*.exe
*.EXE
```

After setting up the ignore file, remove the files you no longer want under version control by selecting them, right-clicking, choosing TortoiseHg, and Forget Files, then commit the changes. Note: one issue you might run into when you do this is that the case of the file extension somehow changes. For example, I removed "mainform.sct" from the repository but when I went to commit the change, it failed because "mainform.SCT" wasn't part of the repository. To fix this, I reverted the change to the file (which now has an SCT rather than sct extension), then forgot the file again and committed the change.

Workflow

Here's the process for development to ensure all developers keep updated with other developers' changes (see **Figure 9**):

- Make changes as necessary.
- Run any tests you have for the project and fix any problems identified by failing tests.
- Convert the binary files to text files.
- Commit the changes.
- Pull from the central repository to retrieve changes other developers made.
- If anything was retrieved:
  - Merge the changes with your local folder and resolve conflicts as necessary.
  - Convert the text files to binary files.

    - Repeat all of the above steps.
- Once a pull retrieves nothing from the repository, push to the central repository.

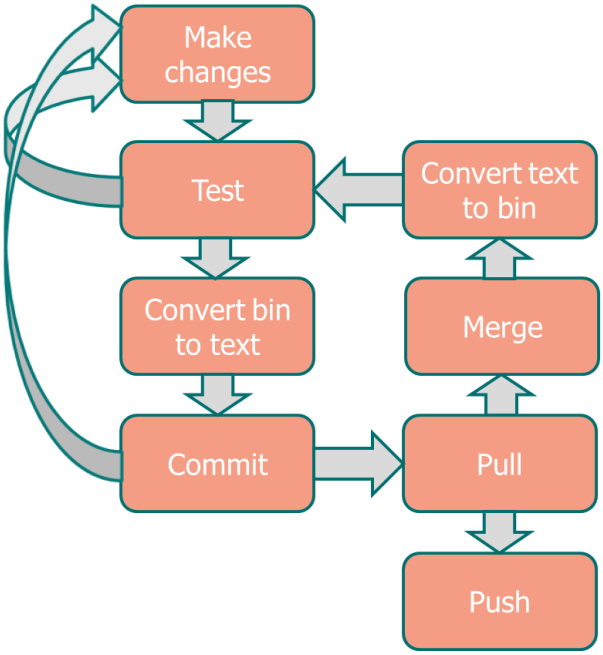


Figure 9. Version control workflow.

When to commit

How often should you commit? The nice thing about DVCS like Mercurial and Git is that there's no reason not to commit frequently. My rule of thumb is to commit after every "change," even if multiple files are involved (that is, after every atomic change). If you don't, reverting can be painful since you may lose other changes you want to keep.

Be sure to use meaningful, detailed comments. I've seen comments where the developer wrote something like "added button." I can see that from the source code change, so I don't need to see that in the comment. Include comments about why the change was made and steps to reproduce the bug if it's for a bug fix. If the reason for the change is including in your issues or bug tracking system, include the ID for the issue (we'll see an example of how this is handy later in the "Using a cloud-based repository" section). Not only does this help the next developer (or even you) understand what the change is all about, you can also output the commit comments to file so it can be used as documentation.

When to push

You can push after every commit if you wish. We tend not to: we typically push when we have a small set of features or bugs finished so other developers can test them in a batch, which can be more efficient than testing after each one (of course, the developer making the changes tests after each one). We also push if something another developer is going to work on relies on a change we made, if a customer needs an update for a bug we fixed, or when we want to create a new build.

Remember to pull before every push, or you may have more work to do in merging other people's changes with yours.

When to branch

Branching is something that made me nervous thinking about until we actually did it. Part of the reason is the fear of updating to an earlier build and losing the changes you've made. What got me over that fear was frequent commits and learning to trust the repository ("use the Force, Luke").

Another part of the reason is because there are many differing ideas about branching, including:

- In Chapter 8 of "Mercurial: The Definitive Guide" (http://tinyurl.com/d6e8oc), Bryan O'Sullivan discusses branching and makes a case for each release being its own repository: "There's a one-to-one relationship between branches you're working in and directories on your system."
- In his post at http://tinyurl.com/2vj4uhz, Vincent Driessen proposes a branching strategy that works well for his projects (**Figure 10**): separate branches for "production-ready" ("master," the Git equivalent of Mercurial's "default"), latest development ("develop"), individual features, hotfixes to previous releases, and releases.

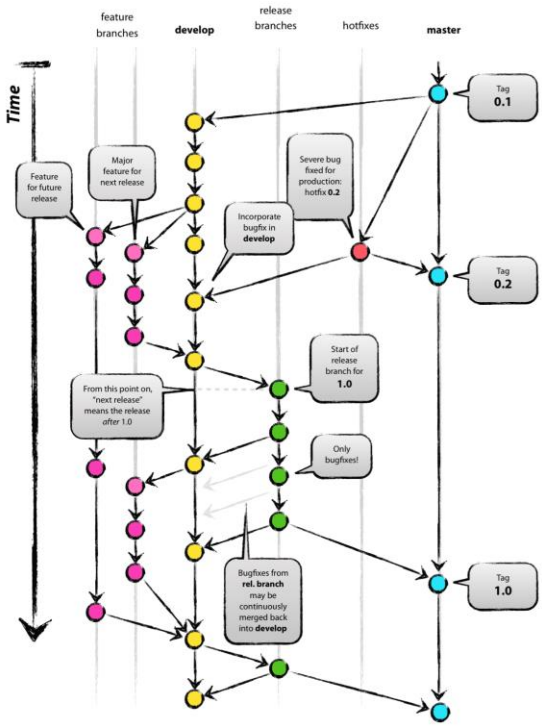


Figure 10. A complex branching strategy.

- Spencer Christensen discusses several different types of workflows in his post at http://tinyurl.com/nleeh4c, including one that uses no branching at all.
- Craig Berntson, a well-known former VFP developer, argues that branching is an anti-pattern and is wasteful, so there should be no branching (https://speakerdeck.com/craigber/branches-and-merges-are-bears-oh-my).
- Steve Losh (http://tinyurl.com/8yh3n6a) discusses a simple workflow with just two branches: "default," where development occurs, and "stable," where bug fixes for previous releases go.

The workflow we use at Stonefield Software is similar to Steve's except we have branches for each release (**Figure 11**). In detail, the workflow is:

- All new development is done in the default branch. We never add new features to previous versions; we only perform bug fixes.
- Once we've released a major version, we create a branch named for the version number. The reason for doing that is to maintain a set of source that's been deployed to customers.
- If there's a bug fix in a particular version, we switch to the branch for that version, make the changes and commit, and create a new release (or patch) for customers. Since it's likely the bug also exists in the current code (unless something unusual happened such as the code where the bug fix was made is no longer used), we switch back to the default branch and merge from the version changes.

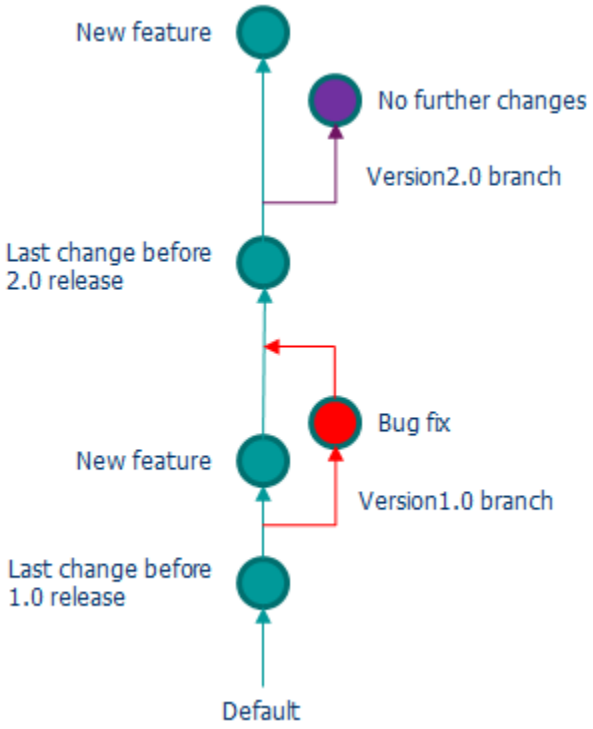


Figure 11. The workflow we use at Stonefield Software.

Always merge from an earlier version's branch into the default branch. The first time I did this, I merged the other way around and inadvertently added the new features we'd added in the new version to the old version.

The reason we use this workflow is the same one Steve outlines: most changes, and therefore commits, are done in the default branch, so it's easier on developers than having to constantly switch branches. It's a simplistic workflow that works for us.

To create a branch, do a commit (even if there are no changes) and click the Branch button (see **Figure 12**). Select *Open a new named branch* and enter the name for the branch. After clicking OK, enter a comment, such as "Created Version1.0 branch," and click the Commit button. This not only creates a new branch, it makes it the current branch. You may wish to switch back to the default branch, where new development occurs, by selecting the last revision from that branch and updating to it. Forgetting to do that means that you may inadvertently make new changes in the wrong branch; I'll discuss how to fix that situation later. **Figure 13** shows how TortoiseHg Workbench appears after creating a Version1.0 branch and switching back to the default branch (notice the Working Directory uses the default branch). Notice that the graph shows the Version1.0 branch in Rev. 7 diverges from the default branch. That'll be the case from now on.

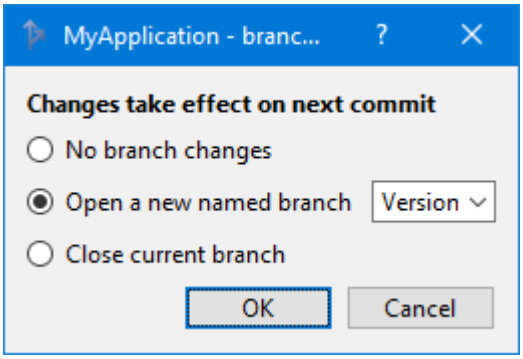


Figure 12. Create a new branch by clicking the Branch button in the Commit dialog.

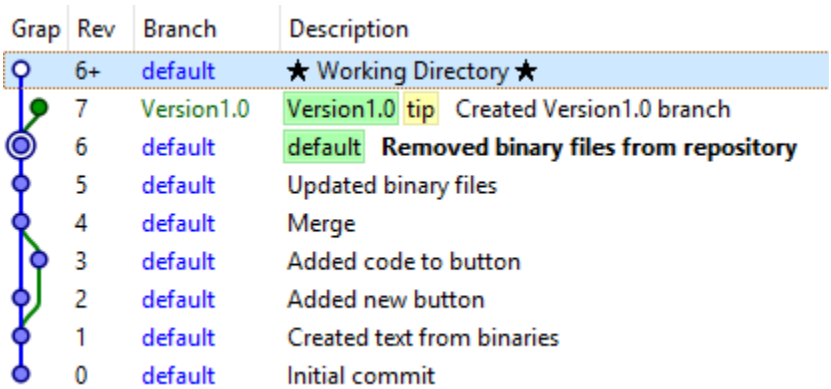


Figure 13. After creating a Version1.0 branch and switching back to the default branch.

Fixing problems

There are lots of different kinds of problems you can run into with version control. Here are some of the ones I've encountered and the solutions to them.

Undoing a change

You make a change to a program, save it, and then realize you made a mess and want to go back to the way the file was before. Or maybe you were experimenting with a change and found that the experiment didn't work. If the change was to a VFP binary file such as a form and you haven't generated the text file from it yet, the fix is easy: regenerate the binary from the text file to put it back the way it was. If you did generate the text file or the change was to a PRG or other non-binary file, Revert is your friend: right-click the file and choose Revert Files from the TortoiseHg menu, or use the hg revert command. This restores the file to its committed version (yet another reason to commit often). Remember that if it's a VFP binary file, you actually revert the text version of the file so you have to regenerate the binary file.

If you committed the change, you can revert to a previous revision or rollback the commit.

Always commit before starting new work. Otherwise, if you have to revert, you'll lose more than just the new changes you made.

Committing to the wrong branch

You're working away in the default branch, working on a new version of the application. Then a user reports a bug. You run the application, reproduce the bug, track it down in the source code, and fix it. Then you realize you fixed the bug in the default branch, not the Version1.0 branch the application the customer is working with is built from. You have a few choices:

- If you haven't committed yet, you can apply the same changes in the desired branch using these commands:

  ```
  hg diff --git > mypatch
  hg update --clean Version1.0
  hg import --no-commit mypatch
  ```

  (Replace "Version1.0" with the name of the desired branch.)

- If the change is simple (say a change in one line of code), the easiest thing to do is to back out the change (right-click the revision in Tortoise Workbench and choose Backout), switch to the correct branch, redo the change, commit, switch to the default branch, and merge with the change to fix the bug in both versions.

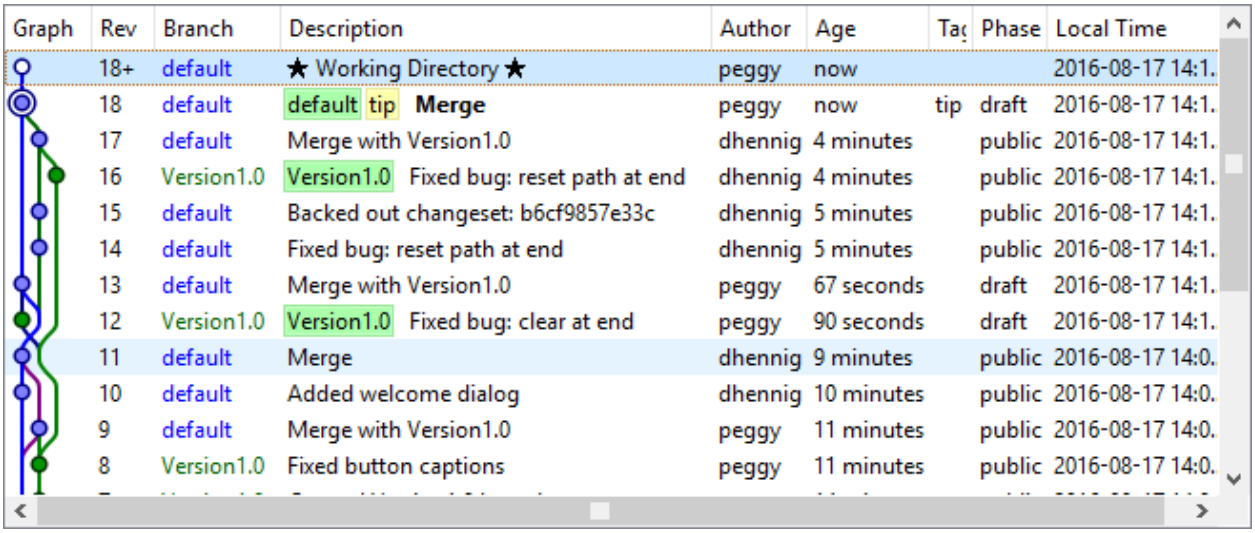
- When the situation is more messy, such as when I have some uncommitted new features plus the bug fix, I've resolved the problem using a mechanism that sort of feels like cheating from a version control perspective: making a copy of the source file the bug fix is in, committing the changes so I don't lose them in the current branch (you could also shelve the changes and then later unshelve them), updating to the desired branch, opening both the current source file and the copy I made, and manually copy and paste the changes from the copy into the current file.

Multiple developers changing multiple branches

The first time another member of my team and I both made changes to more than one branch and merged them together, I was surprised at what we had to do to make everything work.

Here's a simplified version of this scenario: I make a change in the Version1.0 branch and merge the changes with the default branch. Peggy does the same thing (a different change, obviously). I push my changes and Peggy does a pull.

Figure 14 shows the situation we're in. Her change was in Rev. 12 and mine was in Rev. 16. After pulling and merging my Rev. 17 (which merges my Version1.0 change with the default branch) with her working directory (which contains her Rev. 13, which merges her Version1.0 change with the default branch), the default branch has both sets of changes. However, you can see that her Version1.0 branch (Rev. 12) has her change and my Version1.0 branch (Rev. 16) has my change. If she wants to fix another bug in Version1.0, she can't update to Rev. 16 because it doesn't have her changes and she can't update to Rev. 12 because it doesn't have my change.



Graph	Rev	Branch	Description	Author	Age	Tag	Phase	Local Time
	18+	default	★ Working Directory ★	peggy	now			2016-08-17 14:1.
	18	default	default tip **Merge**	peggy	now	tip	draft	2016-08-17 14:1.
	17	default	Merge with Version1.0	dhennig	4 minutes		public	2016-08-17 14:1.
	16	Version1.0	Version1.0 Fixed bug: reset path at end	dhennig	4 minutes		public	2016-08-17 14:1.
	15	default	Backed out changeset: b6cf9857e33c	dhennig	5 minutes		public	2016-08-17 14:1.
	14	default	Fixed bug: reset path at end	dhennig	5 minutes		public	2016-08-17 14:1.
	13	default	Merge with Version1.0	peggy	67 seconds		draft	2016-08-17 14:1.
	12	Version1.0	Version1.0 Fixed bug: clear at end	peggy	90 seconds		draft	2016-08-17 14:1.
	11	default	Merge	dhennig	9 minutes		public	2016-08-17 14:0.
	10	default	Added welcome dialog	dhennig	10 minutes		public	2016-08-17 14:0.
	9	default	Merge with Version1.0	peggy	11 minutes		public	2016-08-17 14:0.
	8	Version1.0	Fixed button captions	peggy	11 minutes		public	2016-08-17 14:0.

Figure 14. Two developers made independent changes in the Version1.0 branch which must be merged.

To fix this, we need to merge the Version1.0 branches as well as the default branches. She updates to her last Version1.0 revision (Rev. 12) then merges my last Version1.0 revision (rev. 16).

Using a cloud-based repository

If all of your developers always have access to a file server on your network, you can use that as the location for your common repository. However, that doesn't work as well when you need remote access to the repository, such as when you're on the road or if you have a developer who works remotely. In that case, using one of the cloud-based repositories, such as GitHub or BitBucket, is a better solution.

We use BitBucket because it was more popular with Mercurial users when we started using Mercurial five years ago (BitBucket also works with Git). Some developers have told me that GitHub is better, but BitBucket has all the features we need (and then some) so we have no reason to switch. So, I'll discuss BitBucket but GitHub is a fine choice and works in a similar manner.

To start with BitBucket, create an account at BitBucket.org. BitBucket is free for up to five users; for larger teams, it's currently $1 US per user per month. After logging in, you can create a repository by choosing Create Repository from the Repositories menu (see **Figure 15**). Fill in the settings (click Advanced Settings to see the full set) and click Create Repository.

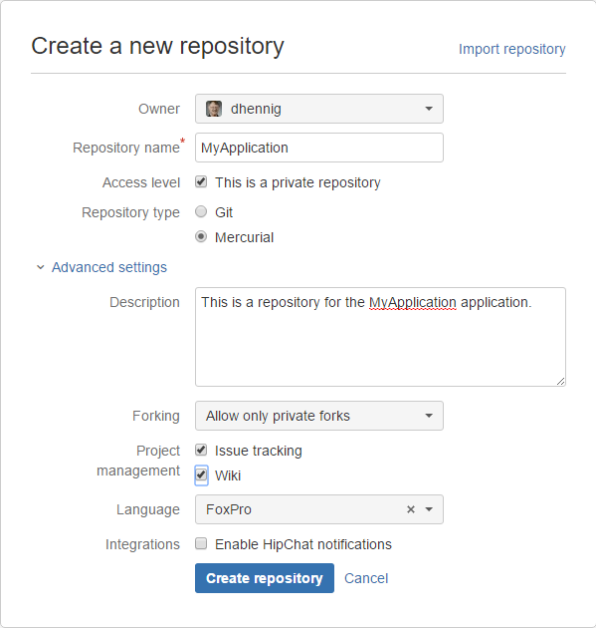


Figure 15. Creating a BitBucket repository.

To push an existing repository to it, click the "I have an existing project link" in the repository's control panel page, shown in **Figure 16**. The page shows the commands to push the repository or you can use Tortoise Workbench to do it visually:

- Copy the URL displayed in the page, edit the hgrc file in the .hg folder using a text editor, and paste the URL into the default setting in the [paths] section. Alternatively, right-click the project folder, choose Synchronize from the TortoiseHg submenu, and paste the URL in as the default path.

- Push the repository. The files in the project then appear in the Source page of the repository in BitBucket.

After creating the repository, you can invite other team members to access it by clicking the Send Invitation button in the Overview page (they need to have BitBucket accounts as well). You can also control access to the repository on the Access Management page under Settings.

A couple of nice features BitBucket provides are a wiki and issue tracking. The wiki can be used for anything you wish, such as documentation or wish list items from customers (if you give them access to it). Issue tracking is really helpful; we use it to track bugs and to-do items. You can automatically close an issue if you use a comment that includes the issue number when you commit (such as "Resolves issue #75").

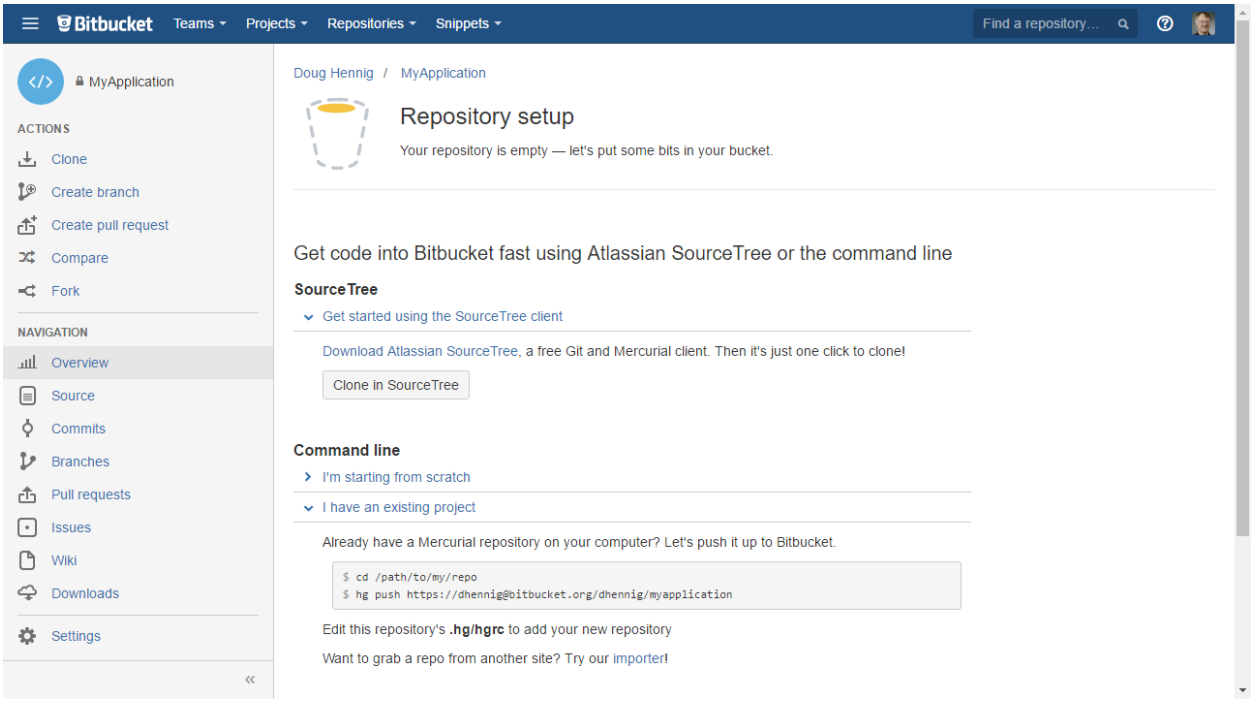


Figure 16. The repository control panel.

Summary

Version control is an essential part of every developer's toolkit. I hope you found some useful tidbits of information in this document. Feel free to adopt and adapt the processes I described to fit your needs.

Biography

Doug Hennig is a partner with Stonefield Software Inc. He is the author of the award-winning Stonefield Database Toolkit (SDT); the award-winning Stonefield Query; the MemberData Editor, Anchor Editor, and CursorAdapter and DataEnvironment builders that

come with Microsoft Visual FoxPro; and the My namespace and updated Upsizing Wizard in Sedna.

Doug is co-author of *VFPX: Open Source Treasure for the VFP Developer*, *Making Sense of Sedna and SP2*, the *What's New in Visual FoxPro* series, *Visual FoxPro Best Practices For The Next Ten Years*, and *The Hacker's Guide to Visual FoxPro 7.0*. He was the technical editor of *The Hacker's Guide to Visual FoxPro 6.0* and *The Fundamentals*. All of these books are from Hentzenwerke Publishing (http://www.hentzenwerke.com). He wrote over 100 articles in 10 years for FoxTalk and has written numerous articles in FoxPro Advisor, Advisor Guide to Visual FoxPro, and CoDe. He currently writes for FoxRockX (http://www.foxrockx.com).

Doug spoke at every Microsoft FoxPro Developers Conference (DevCon) starting in 1997 and at user groups and developer conferences all over the world. He is one of the organizers of the annual Southwest Fox and Southwest Xbase++ conferences (http://www.swfox.net). He is one of the administrators for the VFPX VFP community extensions Web site (http://vfpx.codeplex.com). He was a Microsoft Most Valuable Professional (MVP) from 1996 through 2011. Doug was awarded the 2006 FoxPro Community Lifetime Achievement Award (http://tinyurl.com/ygnk73h).