THE EXPERT'S VOICE® IN SOFTWARE DEVELOPMENT

Practical Software Development Techniques

Tools and Techniques for Building Enterprise Software

Edward Crookshanks

Apress®

For your convenience Apress has placed some of the front matter material after the index. Please use the Bookmarks and Contents at a Glance links to access them.

Apress®

Contents at a Glance

Introduction

Purpose

The purpose of this book is to discuss and provide additional resources for topics and technologies that current university curriculums may leave out. Some programs or professors may touch on some of these topics as part of a class, but individually they are mostly not worthy of a dedicated class, and collectively they encompass some of the tools and practices that should be used throughout a software developer's career. Use of these tools and topics is not mandatory, but applying them will give the student a better understanding of the practical side of software development.

In addition, several of these tools and topics are the 'extra' goodies that employers look for experience working with or having a basic understanding of. In discussions with industry hiring managers and technology recruiters, the author has been told repeatedly that fresh college graduates, while having the theoretical knowledge to be hired, are often lacking in more practical areas such as version control systems, unit testing skills, debugging techniques, interpreting business requirements, and others. This is not to slight or degrade institutional instruction, only to point out that there are tools and techniques that are part of enterprise software development that don't fit well within the confines of an educational environment. Knowledge of these can give the reader an advantage over those who are unfamiliar with them.

This guide will discuss those topics and more in an attempt to fill in the practical gaps. In some cases the topics are code-heavy, in other cases the discussion is largely a survey of methods or a discussion of theory. Students who have followed this guide should have the means to talk intelligently on these topics and this will hopefully translate to an advantage in the area of job hunting. While it would be impossible to cover all tools and technologies, the ones covered in this guide are a good representative sample of what is used in the industry today. Beyond the theoretical aspects of computer science are the practical aspects of the actual implementation; it is this realm that this book attempts to de-mystify.

In short, it is hoped that this companion guide will help graduates overcome the "lack of practical experience" issue by becoming more familiar with industry standard practices and common tools. This volume we cannot create experts but it can at least provide enough cursory knowledge such that the reader can discuss the basics of each topic during an interview. With a little practice and exploration on their own, the student should realize that supplementing an excellent theoretical education with practical techniques will hopefully prove useful not only in writing better software while in school, but also translate to an advantage when out of school and searching for a job.

Overview of Topics

The following topics and tools are discussed:

- Version control
- Unit Testing and Test Driven Development
- Refactoring
- Build tools, automated build engineering, and continuous integration
- Debugging
- Comparison of development methodologies
- Design patterns and architecture
- Requirements
- Basic SQL statements and data frameworks

Prerequisites

It is assumed the reader is already familiar with many facets of languages and tools. The typical student would have used Java, .NET, C++, or some other high-level language for course assignments in a typical computer science or software curriculum and is probably at the sophomore, junior, or senior level. The reader should also be familiar with the differences between console applications, GUI applications, and service/daemon applications. The nuances of procedural, object-oriented, and event-driven program should be known at a high-level if not better. The examples will be kept as simple as possible where needed because the intent is not to teach CS topics and object-oriented design, but how to use these particular tools and concepts to assist in implementing the problem at hand.

Disclaimer

The tools and techniques discussed in this guide are not the only ones on the market. If a particular tool is mentioned in this guide it does not mean that it is the only tool for the job, is endorsed in any way by the author or publisher, or is superior in any way to any of its competitors. Nor does mention of any tool in this publication insinuate in any way that the tool owners or resellers support or endorse this work.

Java and Java-based trademarks are the property of Sun Microsystems. Visual Studio® is a registered product of Microsoft and all other Microsoft product trademarks, registered products, symbols, logos, and other intellectual property is listed at `http://www.microsoft.com/about/legal/en/us/IntellectualProperty/Trademarks/EN-US.aspx` Eclipse™ is property of the Eclipse Foundation. All other trademarks, registered trademarks, logos, patents, and registered names are the property of their respective

owner(s). We are aware of these ownership claims and respect them in the remainder of the text by capitalizing or using all caps when referring to the tool or company in the text.

Any example code is not warranted and the author cannot be held liable for any issues arising from its use. Also, the references provided for each topic are most definitely not an exhaustive list and again, their mention is not to be construed as an endorsement, nor is any resource left off the list to be considered unworthy. Many additional books, web sites, and blogs are available on these topics and should be investigated for alternate discussions. Any mentions of names anywhere in the book are works of fiction and shouldn't be associated with any real person.

Software Notes

Examples are provided in a variety of languages and with different tools, all of which have some level of free software available. Enterprise versions of these tools may exist, or similar tools with stricter licensing models and slightly different semantics may exist; it is simply assumed that tools at the educational level will be closer to the free versions. Also, most hobby developers or recent graduates will probably make use of free tools instead of starting with an expensive development tool suite. If a particular topic/example is not given in a familiar language or with a familiar tool it should be easily translated into another environment. Where possible, notes on how different platforms solve different problems in different ways will be noted. Some of these tools may already be mandated by an employer, others may be free to choose which tools to use to start a practice discussed here. The development tools are the current release available in early 2011, namely Visual Studio 2010 Express Editions, and Eclipse 3.6. Other tools used in the text were obtained around the same time.

Please note—the examples will be kept necessarily simple. In fact, most of the examples will be so short that the tools and techniques used on them will probably not seem worth it. However, in the context of much larger systems, enterprise systems, these tools and techniques are very useful.

CHAPTER 1

Version Control

Version control, sometimes referred to as the code or source repository, can serve several purposes in a typical software development organization:

1. To coordinate project source code between different developers or groups of developers.
2. To serve as the 'system of record' for code that goes into production.
3. Centralizing source storage and providing autonomy from developer's machines.
4. Allow for automated tests and builds to occur on demand or at configured times.

In this section we'll first discuss some terminology associated with version control then move on to a technical example.

Theory

The main purpose of any version control system is to attempt to coordinate file sharing amongst multiple developers. For example, if two users want to edit the same source code file, how is that managed? In practice there are two principal modes of operation for source repositories to manage this operation. These two modes can go by many names, but we refer to them in this manual as "Lock on check out" (LOCO) and "Merge on modify" (MOM). These two methods of operation describe how the user interacts with their local copy and how that relates to the same file maintained in the repository.

Many version control tools use the LOCO methodology to address file sharing issues. In this method, when a user requests a file from the central repository, it is unavailable, or "locked" to all other users who request it. This is known as "checking out" the file and the analogy is similar to a library book. Only in the case of version control an unedited copy still exists in the repository while the user has a "working copy" they can edit. In this way the user can make changes to their local copy and constantly compare their modifications to the unedited copy in the repository. Until the user who is editing the file "checks it in" to the repository no one else can edit the file. This is a method of "forced serialization" – one user editing a file will lock *all* other users from editing that file until the file is checked in.

A different approach is the MOM method. Is this method, multiple users can get a working copy of the source file from the repository. Each user then makes changes to their local "working copy" in the course of their normal work. When the users then "check in" their local copies, the system has built in logic to smoothly merge the changes into a single file. Although this seems as if it would be more chaotic than the LOCO method, it is rare that two users are editing the same place in the same file at the same time. If so, the version control system can detect this and signal a conflict which must be resolved manually. In practice this rarely happens, and when it does it forces communication between the two (or more) team members. In addition, since multiple members can edit the same file at the same time there is no waiting for someone to check in a file.

In the technical section below, both Ron and Nancy get a copy of the source files. At this point there is an unrevised copy of each file in the repository, and both Ron and Nancy have local working copies that they each can modify. If Ron gets done quickly and checks his file back in, the repository now contains the updated version and Nancy still has the original file with her changes only. Later, when she attempts to check her file back in, the system will let her know that there has been changes to the repository version since her last check out. At that point, she must request an update from the repository. The version control system will bring a fresh copy from the repository and attempt to merge the repository version (with Ron's changes) into her local working copy. If all goes well she will end up with a file that contains both her changes and Ron's changes. She can now recompile locally and test out both her changes and Ron's changes for programming conflicts. If none are found she can now check in her version to the repository. After Nancy's check in the repository will contain both of their changes. Ron will have to update his file to get the changes done by Nancy.

Software Demonstration

Most version control systems have at least 3 types of client tools: shell commands, GUI clients, and IDE plug-ins. In this book we won't cover the shell commands but will look at two types of plug-in components.

For the first software demonstration we will use Subversion and the Tortoise SVN plug-in for the Microsoft Windows File Explorer. Subversion (also referred to as "svn") is very popular but there are many other vendors of version control software in the market. Others include ClearCase from IBM, Visual SourceSafe from Microsoft, Microsoft Team System, CVS, and Git.

In this demonstration some liberties will be taken for illustrative purposes. All files and the repository will be local to one machine, but it will be pointed out where the process would differ for a remote server. Also, instead of different users, multiple directories will be used to simulate different users.

First, we need some source code to control. A bare-bones C# Windows Forms application was created with Visual Studio 2010 Express Edition. The logic of the application simply says "Hello" to the user when the "Say Hello" button is pushed. The file structure is shown below:

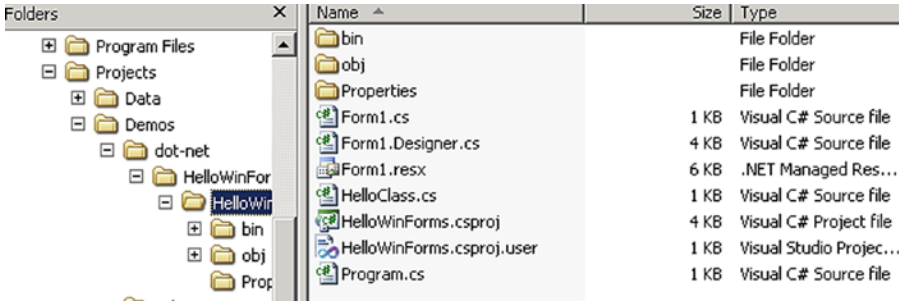

Figure 1-1. *Example Project Files*

Visually we can tell that the code is not under SVN control because the icons appear normal. After we put the code under version control the icons will be decorated with a sprite that will give us a visual clue as to the file's status.

First we need to install the Tortoise SVN tool. This is simply a download from tortoise svn site: `http://tortoisesvn.tigris.org/`. After installing with the default settings there will be a menu item for "TortoiseSVN". If you try to run the TortoiseSVN program you will get a warning message that this is a snap in and can't be run as a separate program. This warning is shown below:

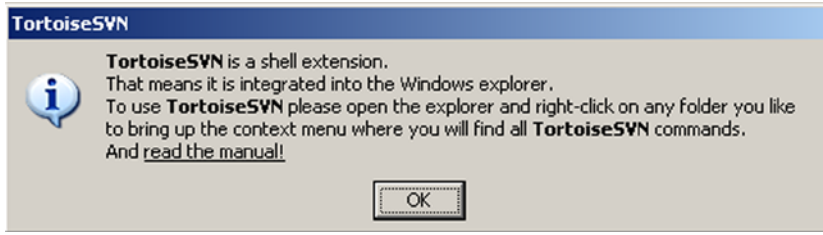

Figure 1-2. *Warning that Tortoise is a plug-in*

So if we return to our explorer window, we now have an additional context menu when right-clicking on a folder, shown in Figure 1-3. FOR THIS LOCAL DEMONSTRATION ONLY we will first create a local repository. It is usually the responsibility of a separate person or team to manage the repository, so a developer would rarely be tasked with creating an actual repository. To do this locally, we navigate in explorer to our directory that will act as our repository, right click on the folder and from the SVN context menu choose "Create repository here." This location will then become the "URL" of our repository and is discussed in more detail a little later when actually importing the code. Note that this is not our project location but a complete separate folder. From Figure 1-4 it can be seen that the repository location in this instance is "C:\Utils\SNV_Repo" and not our project directory. But just to reiterate; in a typical scenario a manager or even a completely separate team would create a repository, and the URL would be an intranet address, not a local file path.

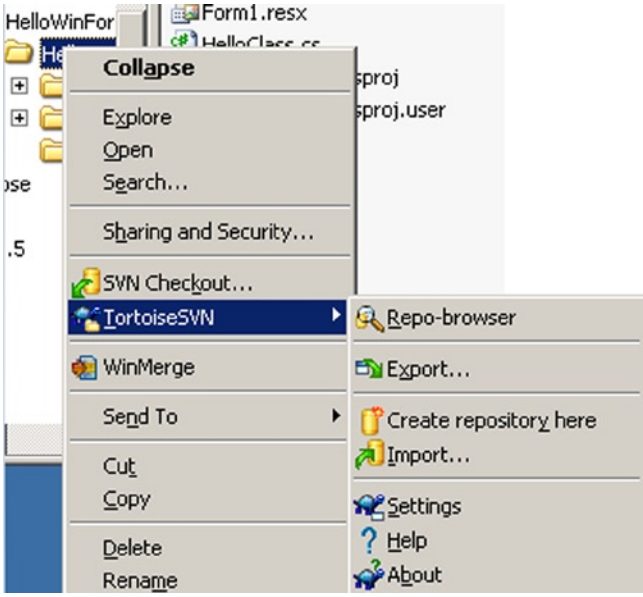

Figure 1-3. *Context menu for TortoiseSVN*

Now we are going to Import the files into the repository. Navigating back to our project location, we choose the "Import" menu item and the following dialog is displayed:

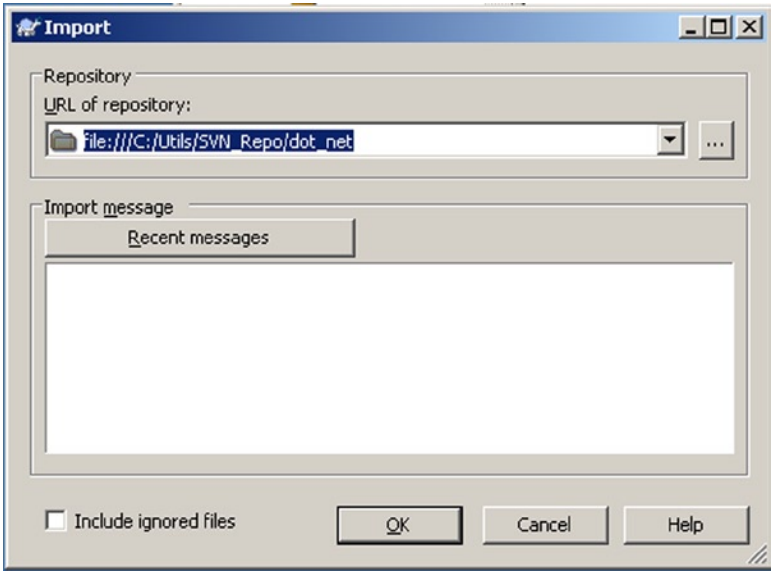

Figure 1-4. *Importing Code in SVN*

Note that in this case, as mentioned above, the URL of the repository is the local file system. THIS IS FOR DEMONSTRATION PURPOSES ONLY. In a typical distributed scenario this would be a URL of an intranet address. The "Import message" field is for annotating the files with a descriptive summary such as "Initial source code check-in." When complete a dialog with a summary of the actions is shown as in Figure 1-5.

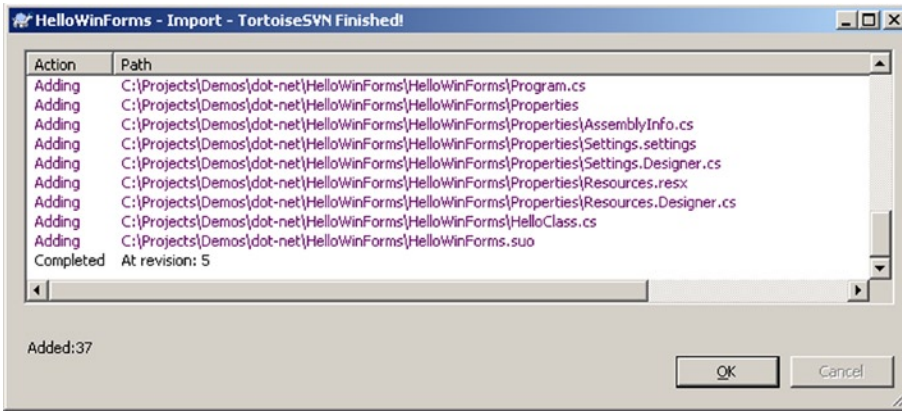

Figure 1-5. *Summary dialog for importing file*

An important piece of information is available in the below summary; "Revision 5" is the unique revision number of the code in the repository. Soon, when we check out the code, we have the ability to ask for the latest version of the code (known as HEAD), or we can ask for a specific revision. This is a neat feature of version control that allows us to "go back in time" and check out code from a previous revision. This could be useful for fixing bugs, performance testing, or simply seeing what changes were made since a certain time in the past. Viewing historical annotations and retrieving a previous version of a file is discussed in more detail later on.

■ **Note** Revision numbers may be handled differently by other version control products. In SVN, the revision identifies the version of the entire project tree. In other products, such as CVS, revision numbers can be applied directly to individual files. Please be sure and read the documentation for whichever product you use.

Note that for this example, I have chosen to import the entire tree, which includes the compiled executable and/or dll files. We will discuss the impact of this a little later on. This policy can differ per organization however; with some including *only* source files and resource files and not the executable. This philosophy stems from the belief that the repository is only for code, not the compiled application itself.

Versioning logic is also what gives us the ability to "Branch" and "Tag" source code, which we discuss later on. In short, branching allows for two different streams of code to co-exists, and tagging is used to further identify a revision of code.

Now if from the context menu we choose to "Browse Repository" there should be a structure like the following figure:

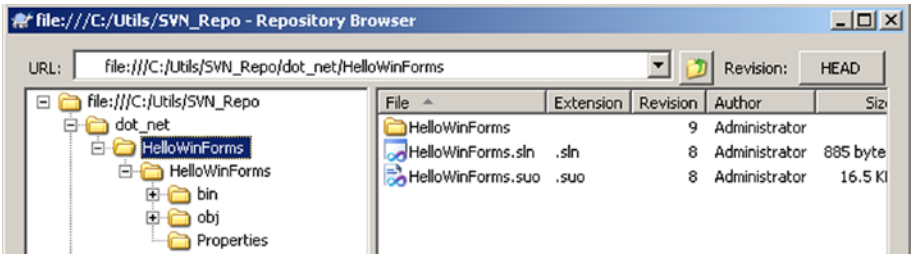

Figure 1-6. *Repository after importing code*

So we have the source in the repository. Now what? Next we need to have our two users, Ron and Nancy, check out the source code and make their own changes. Again, we are simulating two distinct users by using two separate directories, cleverly named "Ron" and "Nancy".

For TortoiseSVN, we navigate to the folder we want to contain our local working copy of the code, right-click, and from the context menu choose "SVN Checkout...", which brings up the dialog box shown in Figure 1-7. Again, the URL in our case is for a local repository. The checkout directory defaults to the one we right-clicked on, and Checkout Depth gives us several options for handling the recursion of the file structure we want to get. Notice also that we have the ability to get the "HEAD" revision or a particular revision number. Using the "Show log" button will bring up a screen for showing revision details such as dates, actions, and messages.

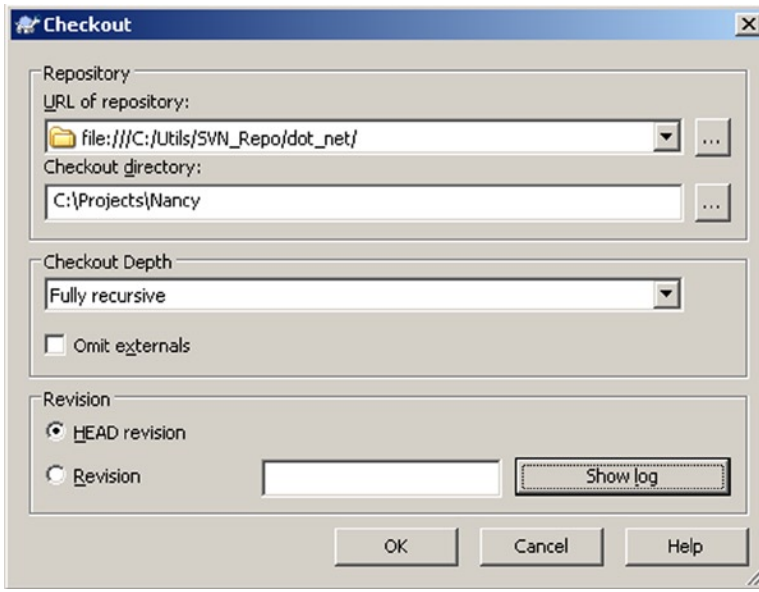

Figure 1-7. *SVN Checkout Command Options*

Since we chose "Fully Recursive" the entire tree is replicated in our local working folder and the folders and files are decorated with special icons. In addition, a special ".svn" folder is added to the directory tree. This is used be the server and SHOULD NOT be modified. The snapshot of the file structure in Figure 1-8 illustrates both directories (Ron and Nancy) after both have checked out the code.

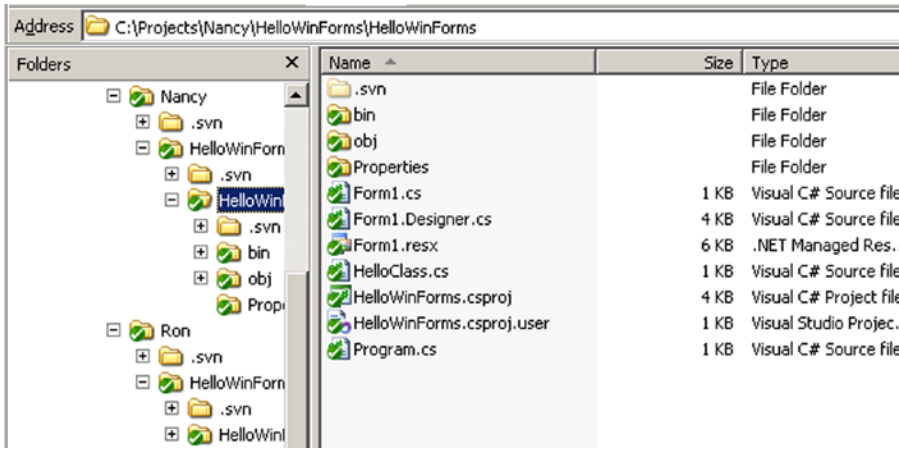

Figure 1-8. *File Structure with decorated icons*

Now we can start our modifications. We'll open the solution from the "Nancy" folder and make some changes to the code. For this example the changes themselves aren't important. After these changes have been saved and we go back out to our File Explorer, we can clearly see that the tortoise plugin has changed the icons to signal that we have some changes made to the files that have not been committed to the repository. Figure 1-9 on the next page shows the state of the local working folder. Notice some of important points in this figure. We can see that not all files have changed; only the ones decorated with the red exclamation point are those that have changed. Also, the folder at the very top of the tree is decorated in this manner so if we are coming back later we can tell at the 'root' level of the tree when a project has some changes to it. Finally, we can see from the figure that Ron has not made any changes yet.

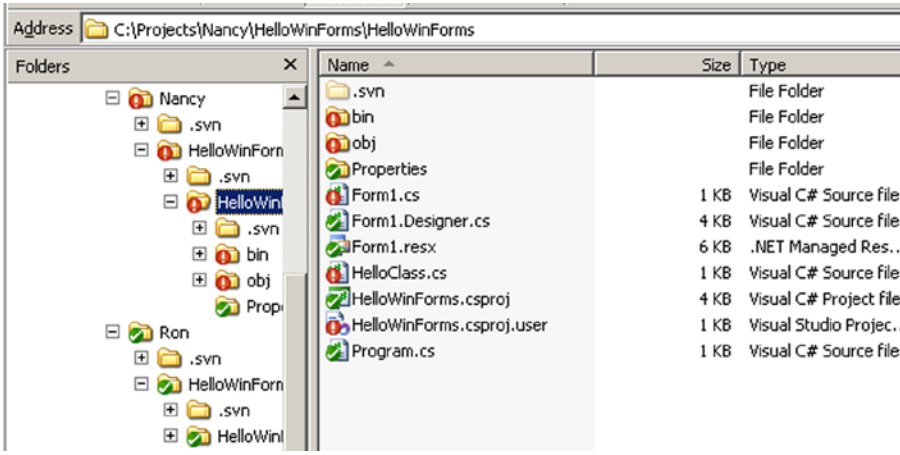

Figure 1-9. *Local files after Nancy's changes*

So let's open the project from the "Ron" folder and make some code changes. Again, the details of the changes are not important, only that we have two different working folders and both working folders have changed code in them. The picture in Figure 1-10 shows the state of the file system after changes to Ron's folder.

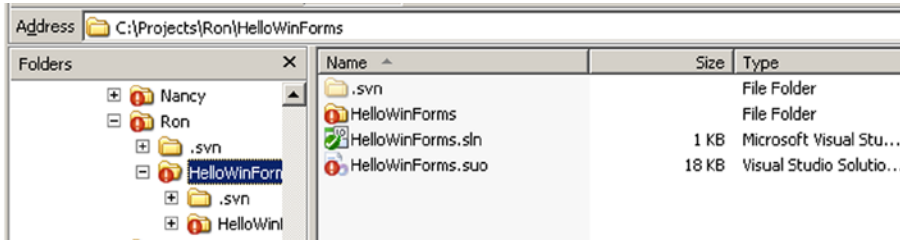

Figure 1-10. *After Ron's changes*

So now we begin to check in source files and see what happens. First, "Ron" checks in his files. This can either be done one file at a time, by highlighting each file and choosing "SVN Commit...". While this might be desirable in some circumstances, here we will make use of the recursive nature of the system and make that choice on the "Ron" folder. Once we make the section the dialog shown in Figure 1-11 appears. Here we can enter our message for this revision ("Ron's changes to the Winforms app") and if needed we can deselect files that we don't want to check in. In this example we've chosen only the modified source files. One neat feature is the ability to see the difference between the working file and the repository file. As it says in the "Changes made" section, double clicking on a filename will launch an application called Winmerge to see the differences in the local file and the repository. We will talk more about that in a little while. But for now, simply choosing the files to check in, typing a message, and clicking "OK" is all that needs done.

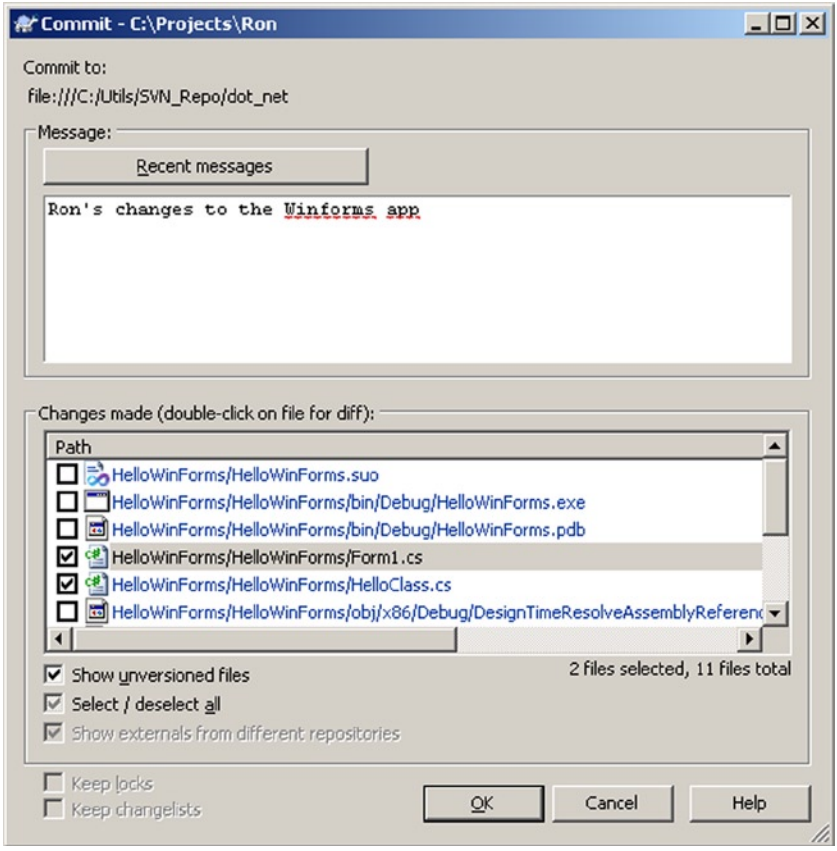

Figure 1-11. *SVN Commit options screen*

Once the files are checked in we get a summary dialog, similar to the summary dialog when importing files shown in Figure 1-5. Ron's folder will finally look like Figure 1-12. Note that we are beginning to see the downside of our choice to put all files, even binary and IDE-generated files, in version control. Ron's code changes have been completely checked in but his "root" folder, "Ron", still shows the red exclamation point of a folder with changes. We'll discuss more about this a little later.

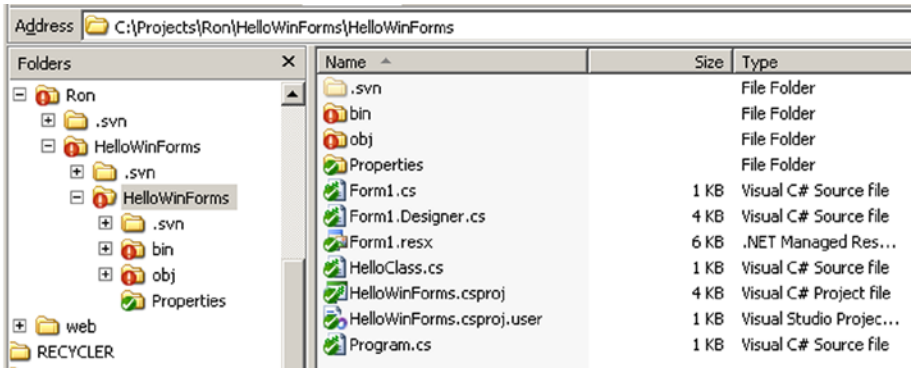

Figure 1-12. *Ron's Folder after his commit*

Now let's move back to the "Nancy" and her directory. Since Nancy is more careful than Ron she will check in files one at a time. Also before Nancy checks in her files, she plans to perform two very important steps. She wants to verify her changes and update from the repository. Why are these important?

Verifying changes will help ensure the changes being committed are the proper ones. Updating from the repository will pull down any changes that have been made in the files and attempt to merge them with the local working copy. This will allow Nancy to see any changes made by other users before she commits her changes. This is important both on LOCO systems and MOM systems. On MOM systems, since all files are inherently unlocked, changes can be made to the very files that we've been working on (as is the case with this example). On LOCO systems, changes could have been made to other files not locked by us but being used by us. As a matter of integration testing we should make sure our changes work in the context of the entire application, not just in the module or component we are changing.

So let's get back to Nancy. Reference Figure 1-9 for the state of Nancy's folder. From that we can see that only two true source files have changed, `Form1.cs` and `HelloClass.cs`; the other changes are to binary folders or IDE-generated files. First she wants to verify her changes so she right-clicks on the `Form1.cs` file and from the context menu selects "TortoiseSVN ➤ Diff." This will open the side-by-side file comparison window, shown in Figure 1-13.

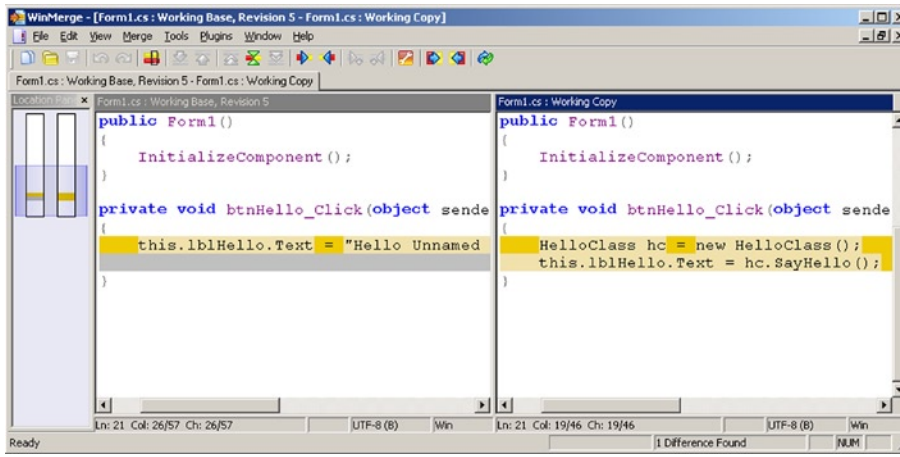

Figure 1-13. *Diff compare before committing*

Again, this is a very simple example and her changes are minor; the important concept is verifying code changes before committing. She is satisfied with this comparison, so she closes this application, right-clicks and chooses "SVN Commit..." and follows the same process as Ron. However in her rush, she forgot to do a "SVN Update" before committing. The result is shown in Figure 1-14.

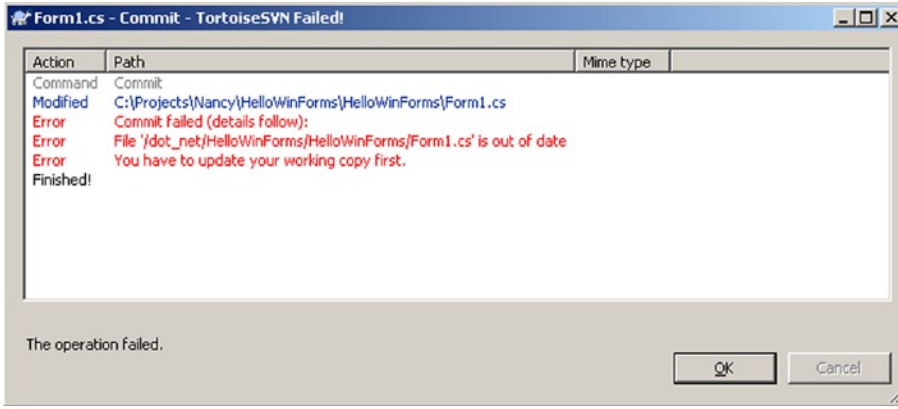

Figure 1-14. *Attempt to commit when working base is old*

When committing, the version control system is smart enough to make sure that her "Working Base" copy is the same as the latest revision in the repository. So this is more than just a "best practice", the repository actually enforces this rule. Confused? Here is a textual explanation of what just happened.

When Ron and Nancy each got their copies of the code from the repository, they both had the same "Working Base" version. That is, they both had exact copies of what came from the repository. The .svn folder we mentioned earlier keeps track of each user's "base" version. This base version is not updated until the user requests it to be. More explicitly, the local copy is not notified of changes in the repository until the local user asks for an update. Once Ron made his changes and checked them in, the repository now had a new base version; one that contained his changes. However Nancy's base version was still the original she received when she first checked out the code.

So how does Nancy fix this problem? She simply right-clicks on the `Form1.cs` file and chooses "SVN Update." Once that is done she gets the message shown in Figure 1-15.

Figure 1-15. *Success Update from the repository*

Now that she has successfully updated *from* the repository, she is free to commit her changes *to* the repository. After committing `Form1.cs` she follows the same process with `HelloClass.cs`. Once this is complete the repository will contain changes from both Ron and Nancy.

Resolving Conflicts

In our case, the `HelloClass.cs` file is a little more complex. Most of the time in large team development changes will be isolated to different sections of a file or even different files altogether. This example was so simple and small that Ron and Nancy can hardly keep from stepping on each other code toes.

When Nancy follows procedure to do an "SVN Update" on her `HelloClass.cs` file, she gets a warning message that there is a conflict. After this warning her local working directory has some funny looking icons in and some additional files, shown in Figure 1-16.

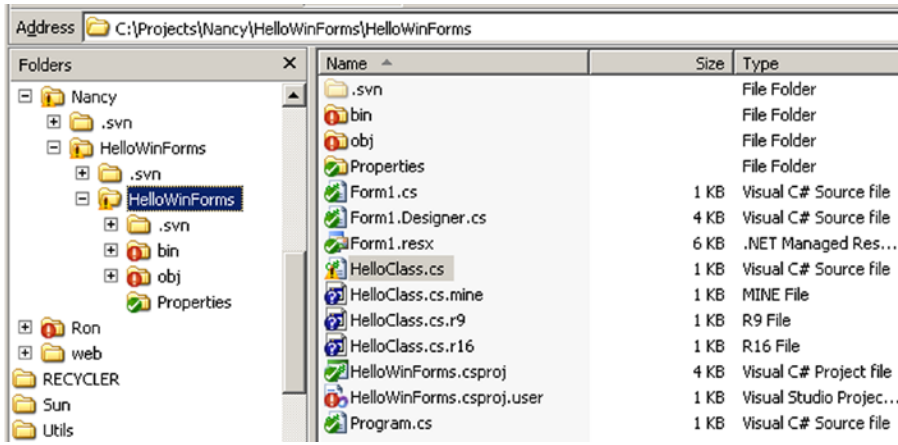

Figure 1-16. *Nancy has a conflict*

To fix this conflict, Nancy right-clicks on the `HelloClass.cs` file and from the context menu chooses "TortoiseSVN ➤ Edit Conflicts" opening the TortoiseMerge screen. Although this screen looks rather busy (and colorful!) it clearly separates Nancy's changes ("Mine") from the repository file ("Theirs") and the attempt at merging them done by the system ("Merged"). The following figures show Nancy's process of merging the changes into a single file.

First, by right-clicking on a conflict line, a context menu with choices of where to pull the resolved source from is shown. Nancy makes her choices, maybe having to consult with Ron while doing so. The final version of her resolution, shown in Figure 1-19, contains both her and Ron's changes. Once this is complete Nancy can save the merged file and mark the conflict as resolved by choosing "Edit ➤ Mark as Resolved" from the menu. Nancy's local directory will now look normal, as shown in Figure 1-20.

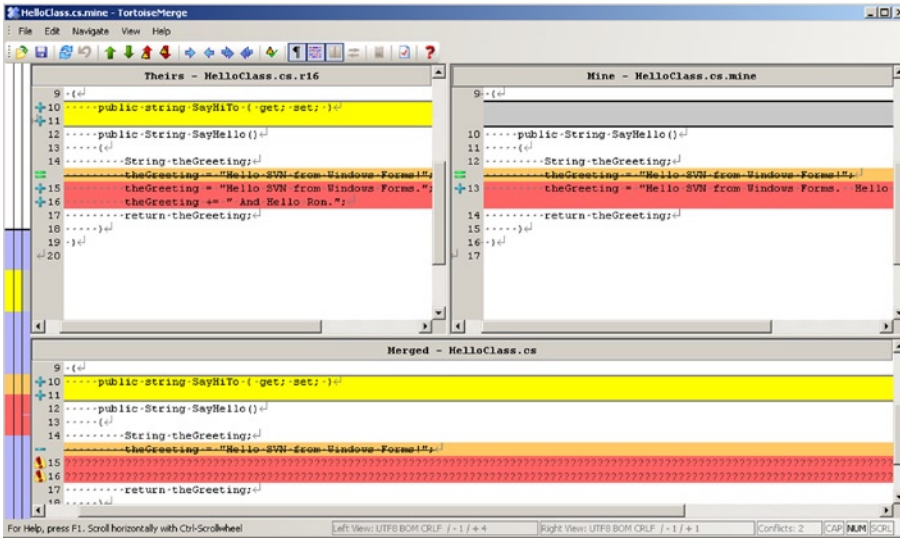

Figure 1-17. *Original merge screen*

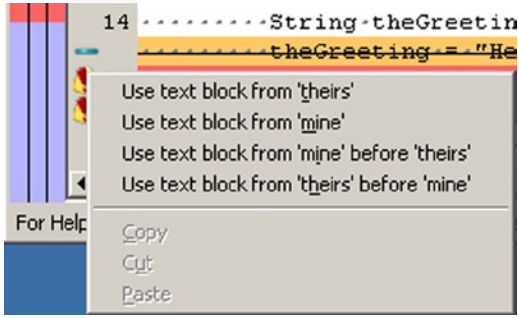

Figure 1-18. *Context menu for a conflict line*

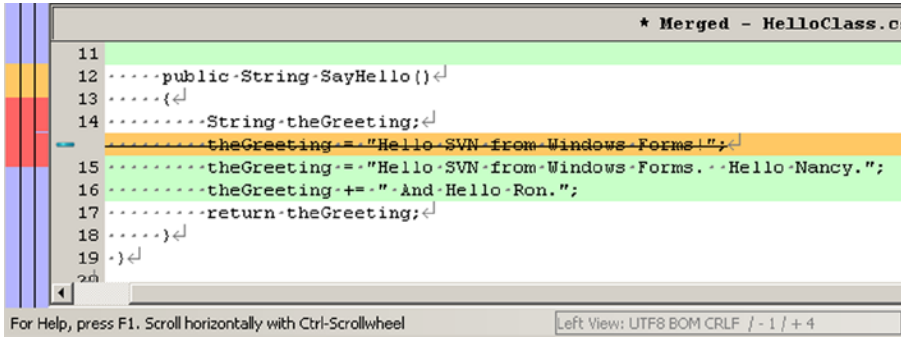

Figure 1-19. *Manually merged file results*

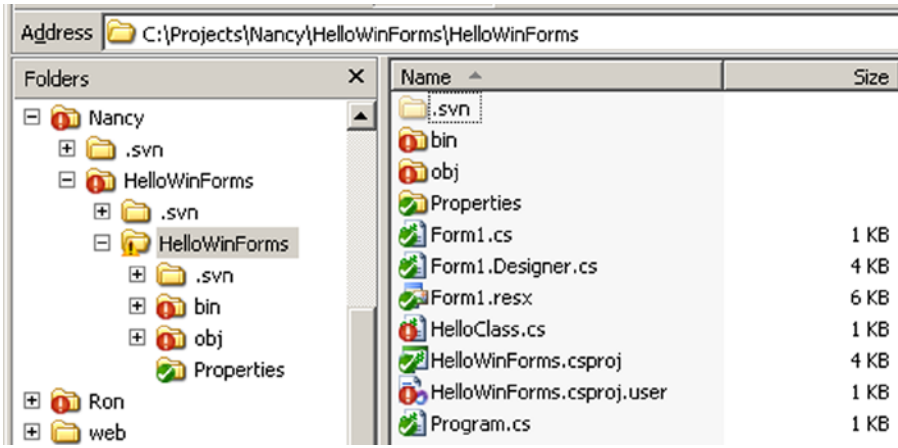

Figure 1-20. *After conflict resolution*

Although she has resolved the conflict, she still has to commit her changes. This is a simple matter of following the normal commit procedure and should work smoothly now that see has merged her changes with the repository.

So our repository now contains both Ron's and Nancy's changes, but there is one final issue to discuss; that of Ron's local working source status. Although he doesn't need Nancy's changes to make his code work correctly, he doesn't want to make new changes to stale source code either. Not only could that make his life miserable by having several conflicts when committing, he could potentially miss changes that are an important prerequisite to his changes. So as a best practice, one should always perform an update before modifying any code. That goes for both LOCO and MOM systems.

Tagging and Branching

Keeping track of changes and allowing for concurrent editing is an important feature of version control. But other important capabilities exist as well. Tagging (or labeling) provides a mechanism to identify a particular revision of code as important, say for a particular milestone in the development process or at a release point.

Branching allows some code to be isolated into a separate area. That area can be worked on independently without disturbing the main line of code. When the "branch" is complete it can be merged back into the main line. The main line is often referred to as the "trunk."

Different tools handle these concepts in different ways. Some apply tags directly to source files and allow for simply retrieving all files based on a tag. SVN has a convention of creating a directory structure with a separate folder for tags.

Branching is also handled in different ways; sometimes it is even transparent to the developer if retrieving and editing a previous version of a file. In those instances, when a previous version of a file is retrieved, a branch is created and later can be merged with the main trunk. In SVN a folder convention is followed, similar to tags. In fact, tagging and