Certified Subversion Version Control Professional Sample Material

V-Skills Certifications

A Government of India
&
Government of NCT Delhi Initiative

V-Skills

1. FUNDAMENTAL CONCEPTS

1.1. Version Control Basics

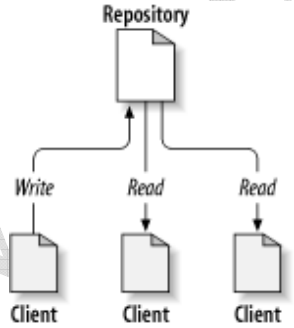
A version control system (or revision control system) is a system that tracks incremental versions (or revisions) of files and, in some cases, directories over time. Of course, merely tracking the various versions of a user's (or group of users') files and directories isn't very interesting in itself. What makes a version control system useful is the fact that it allows you to explore the changes which resulted in each of those versions and facilitates the arbitrary recall of the same.

In this section, we'll introduce some fairly high-level version control system components and concepts. We'll limit our discussion to modern version control systems—in today's interconnected world, there is very little point in acknowledging version control systems which cannot operate across wide-area networks.

The Repository

At the core of the version control system is a repository, which is the central store of that system's data. The repository usually stores information in the form of a file system tree—a hierarchy of files and directories. Any number of clients connects to the repository, and then read or writes to these files. By writing data, a client makes the information available to others; by reading data, the client receives information from others. Figure 1.1, "A typical client/server system" illustrates this.

Figure 1.1. A typical client/server system



Why is this interesting? So far, this sounds like the definition of a typical file server. And indeed, the repository is a kind of file server, but it's not your usual breed. What makes the repository special is that as the files in the repository are changed, the repository remembers each version of those files.

When a client reads data from the repository, it normally sees only the latest version of the file system tree. But what makes a version control client interesting is that it also has the ability to request previous states of the file system from the repository. A version control client can ask historical questions such as "What did this directory contain last Wednesday?" and "Who was the last person to change this file, and what changes did he make?" These are the sorts of questions that are at the heart of any version control system.

The Working Copy

A version control system's value comes from the fact that it tracks versions of files and directories, but the rest of the software universe doesn't operate on "versions of files and directories". Most software programs understand how to operate only on a single version of a specific type of file. So

how does a version control user interact with an abstract—and, often, remote—repository full of multiple versions of various files in a concrete fashion? How does his or her word processing software, presentation software, source code editor, web design software, or some other program—all of which trade in the currency of simple data files—get access to such files? The answer is found in the version control construct known as a working copy.

A working copy is, quite literally, a local copy of a particular version of a user's VCS-managed data upon which that user is free to work. Working copies[5] appear to other software just as any other local directory full of files, so those programs don't have to be "version-control-aware" in order to read from and write to that data. The task of managing the working copy and communicating changes made to its contents to and from the repository falls squarely to the version control system's client software.

Versioning Models

If the primary mission of a version control system is to track the various versions of digital information over time, a very close secondary mission in any modern version control system is to enable collaborative editing and sharing of that data. But different systems use different strategies to achieve this. It's important to understand these different strategies, for a couple of reasons. First, it will help you compare and contrast existing version control systems, in case you encounter other systems similar to Subversion. Beyond that, it will also help you make more effective use of Subversion, since Subversion itself supports a couple of different ways of working.

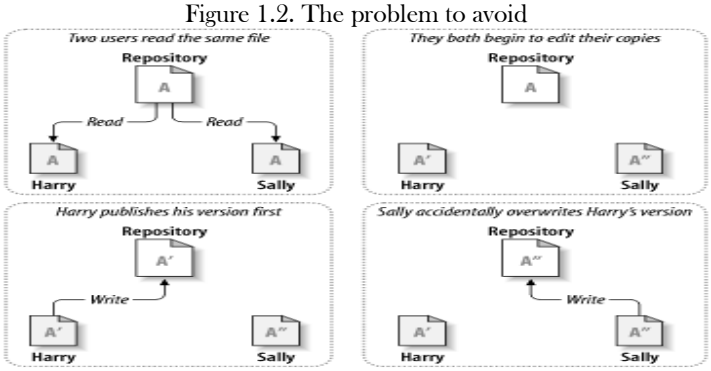
The problem of file sharing

All version control systems have to solve the same fundamental problem: how will the system allow users to share information, but prevent them from accidentally stepping on each other's feet? It's all too easy for users to accidentally overwrite each other's changes in the repository.

Consider the scenario shown in Figure 1.2, "The problem to avoid". Suppose we have two coworkers, Harry and Sally. They each decide to edit the same repository file at the same time. If Harry saves his changes to the repository first, it's possible that (a few moments later) Sally could accidentally overwrite them with her own new version of the file. While Harry's version of the file won't be lost forever (because the system remembers every change), any changes Harry made won't be present in Sally's newer version of the file, because she never saw Harry's changes to begin with.

Harry's work is still effectively lost—or at least missing from the latest version of the file—and probably by accident. This is definitely a situation we want to avoid!

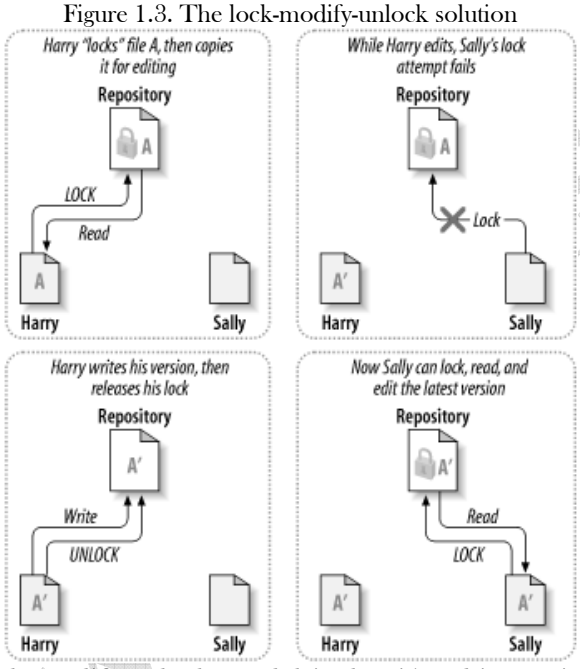
Figure 1.2. The problem to avoid



The lock-modify-unlock solution

Many version control systems use a lock-modify-unlock model to address the problem of many authors clobbering each other's work. In this model, the repository allows only one person to change a file at a time. This exclusivity policy is managed using locks. Harry must "lock" a file before he can begin making changes to it. If Harry has locked a file, Sally cannot also lock it, and therefore cannot make any changes to that file. All she can do is read the file and wait for Harry to finish his changes and release his lock. After Harry unlocks the file, Sally can take her turn by locking and editing the file. Figure 1.3, "The lock-modify-unlock solution" demonstrates this simple solution.

Figure 1.3. The lock-modify-unlock solution



The problem with the lock-modify-unlock model is that it's a bit restrictive and often becomes a roadblock for users:

- ✓ Locking may cause administrative problems. Sometimes Harry will lock a file and then forget about it. Meanwhile, because Sally is still waiting to edit the file, her hands are tied. And then Harry goes on vacation. Now Sally has to get an administrator to release Harry's lock. The situation ends up causing a lot of unnecessary delay and wasted time.

- ✓ Locking may cause unnecessary serialization. What if Harry is editing the beginning of a text file, and Sally simply wants to edit the end of the same file? These changes don't overlap at all. They could easily edit the file simultaneously, and no great harm would come, assuming the changes were properly merged together. There's no need for them to take turns in this situation.

- ✓ Locking may create a false sense of security. Suppose Harry locks and edits file A, while Sally simultaneously locks and edits file B. But what if A and B depend on one another, and the changes made to each are semantically incompatible? Suddenly A and B don't work together anymore. The locking system was powerless to prevent the problem—yet it somehow provided a false sense of security. It's easy for Harry and Sally to imagine that by locking files, each is

beginning a safe, insulated task, and thus they need not bother discussing their incompatible changes early on. Locking often becomes a substitute for real communication.

The copy-modify-merge solution

Subversion, CVS, and many other version control systems use a copy-modify-merge model as an alternative to locking. In this model, each user's client contacts the project repository and creates a personal working copy. Users then work simultaneously and independently, modifying their private copies. Finally, the private copies are merged together into a new, final version. The version control system often assists with the merging, but ultimately, a human being is responsible for making it happen correctly.

Here's an example. Say that Harry and Sally each create working copies of the same project, copied from the repository. They work concurrently and make changes to the same file A within their copies. Sally saves her changes to the repository first. When Harry attempts to save his changes later, the repository informs him that his file A is out of date. In other words, file A in the repository has somehow changed since he last copied it. So Harry asks his client to merge any new changes from the repository into his working copy of file A. Chances are that Sally's changes don't overlap with his own; once he has both sets of changes integrated, he saves his working copy back to the repository. Figure 1.4, "The copy-modify-merge solution" and Figure 1.5, "The copy-modify-merge solution (continued)" show this process.

Figure 1.4. The copy-modify-merge solution

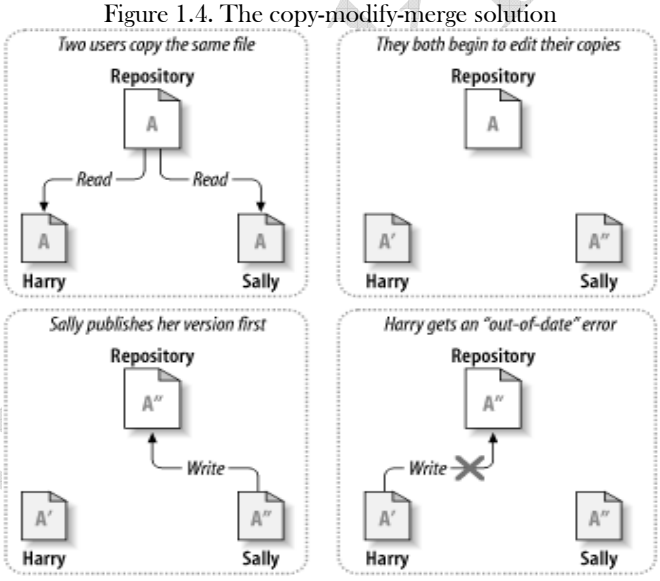
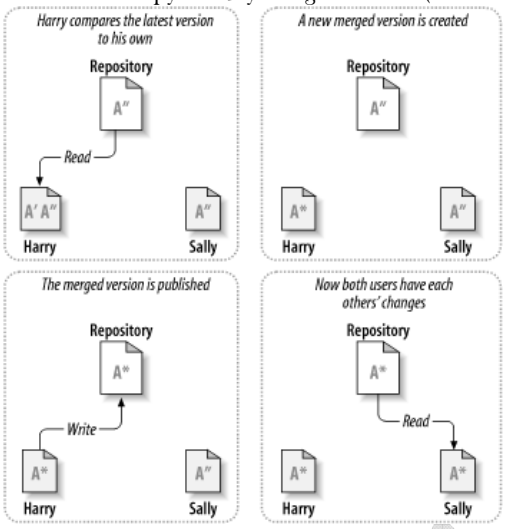


Figure 1.5. The copy-modify-merge solution (continued)



But what if Sally's changes do overlap with Harry's changes? What then? This situation is called a conflict, and it's usually not much of a problem. When Harry asks his client to merge the latest repository changes into his working copy, his copy of file A is somehow flagged as being in a state of conflict: he'll be able to see both sets of conflicting changes and manually choose between them. Note that software can't automatically resolve conflicts; only humans are capable of understanding and making the necessary intelligent choices. Once Harry has manually resolved the overlapping changes—perhaps after a discussion with Sally—he can safely save the merged file back to the repository.

The copy-modify-merge model may sound a bit chaotic, but in practice, it runs extremely smoothly. Users can work in parallel, never waiting for one another. When they work on the same files, it turns out that most of their concurrent changes don't overlap at all; conflicts are infrequent. And the amount of time it takes to resolve conflicts is usually far less than the time lost by a locking system.

In the end, it all comes down to one critical factor: user communication. When users communicate poorly, both syntactic and semantic conflicts increase. No system can force users to communicate perfectly, and no system can detect semantic conflicts. So there's no point in being lulled into a false sense of security that a locking system will somehow prevent conflicts; in practice, locking seems to inhibit productivity more than anything else.

1.2. Version Control the Subversion Way

We've mentioned already that Subversion is a modern, network-aware version control system. As we described in the section called “Version Control Basics” (our high-level version control overview), a repository serves as the core storage mechanism for Subversion's versioned data, and it's via working copies that users and their software programs interact with that data. In this section, we'll begin to introduce the specific ways in which Subversion implements version control.

www.vskills.in

Subversion Repositories

Subversion implements the concept of a version control repository much as any other modern version control system would. Unlike a working copy, a Subversion repository is an abstract entity, able to be operated upon almost exclusively by Subversion's own libraries and tools. As most of a user's Subversion interactions involve the use of the Subversion client and occur in the context of a working copy, we spend the majority of this book discussing the Subversion working copy and how to manipulate it. For the finer details of the repository, though, check out

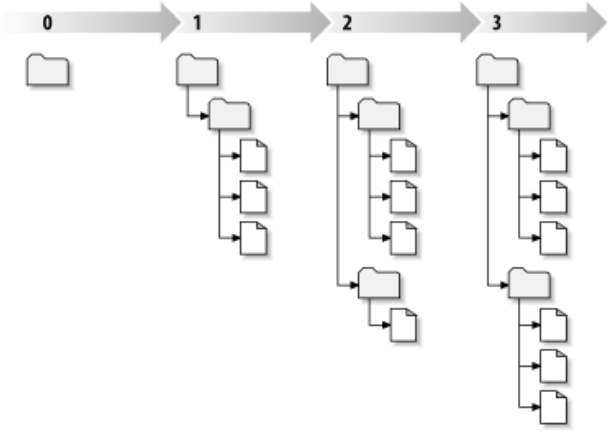
Revisions

A Subversion client commits (that is, communicates the changes made to) any number of files and directories as a single atomic transaction. By atomic transaction, we mean simply this: either all of the changes are accepted into the repository, or none of them is. Subversion tries to retain this atomicity in the face of program crashes, system crashes, network problems, and other users' actions.

Each time the repository accepts a commit, this creates a new state of the file system tree, called a *revision*. Each revision is assigned a unique natural number, one greater than the number assigned to the previous revision. The initial revision of a freshly created repository is numbered 0 and consists of nothing but an empty root directory.

Figure 1.6, "Tree changes over time" illustrates a nice way to visualize the repository. Imagine an array of revision numbers, starting at 0, stretching from left to right. Each revision number has a filesystem tree hanging below it, and each tree is a "snapshot" of the way the repository looked after a commit.

Figure 1.6. Tree changes over time



Addressing the Repository

Subversion client programs use URLs to identify versioned files and directories in Subversion repositories. For the most part, these URLs use the standard syntax, allowing for server names and port numbers to be specified as part of the URL.

- ✓ http://svn.example.com/svn/project
- ✓ http://svn.example.com:9834/repos

Subversion repository URLs aren't limited to only the http:// variety. Because Subversion offers several different ways for its clients to communicate with its servers, the URLs used to address the

repository differ subtly depending on which repository access mechanism is employed. Table 1.1, “Repository access URLs” describes how different URL schemes map to the available repository access methods. For more details about Subversion's server options, see Server Configuration.

Table 1.1. Repository access URLs

Schema	Access method
file:///	Direct repository access (on local disk)
http://	Access via WebDAV protocol to Subversion-aware Apache server
https://	Same as http://, but with SSL encryption
svn://	Access via custom protocol to an svnserve server
svn+ssh://	Same as svn://, but through an SSH tunnel

Subversion's handling of URLs has some notable nuances. For example, URLs containing the file:// access method (used for local repositories) must, in accordance with convention, have either a server name of localhost or no server name at all:

- ✓ file:///var/svn/repos
- ✓ file://localhost/var/svn/repos

Also, users of the file:// scheme on Windows platforms will need to use an unofficially “standard” syntax for accessing repositories that are on the same machine, but on a different drive than the client's current working drive. Either of the two following URL path syntaxes will work, where X is the drive on which the repository resides:

- ✓ file:///X:/var/svn/repos
- ✓ file:///X|/var/svn/repos

that a URL uses forward slashes even though the native (non-URL) form of a path on Windows uses backslashes. Also note that when using the file:///X|/ form at the command line, you need to quote the URL (wrap it in quotation marks) so that the vertical bar character is not interpreted as a pipe.

You cannot use Subversion's file:// URLs in a regular web browser the way typical file:// URLs can. When you attempt to view a file:// URL in a regular web browser, t reads and displays the contents of the file at that location by examining the file system directly. However, Subversion's resources exist in a virtual file system (see the section called “Repository Layer”), and your browser will not understand how to interact with that file system.

The Subversion client will automatically encode URLs as necessary, just like a web browser does. For example, the URL http://host/path with space/project/españa — which contains both spaces and upper-ASCII characters — will be automatically interpreted by Subversion as if you'd provided http://host/path%20with%20space/project/espa%C3%B1a. If the URL contains spaces, be sure to place it within quotation marks at the command line so that your shell treats the whole thing as a single argument to the program.

There is one notable exception to Subversion's handling of URLs which also applies to its handling of local paths in many contexts, too. If the final path component of your URL or local path

contains an at sign (@), you need to use a special syntax—described in the section called "Peg and Operative Revisions"—in order to make Subversion properly address that resource.

In Subversion 1.6, a new caret (^) notation was introduced as a shorthand for "the URL of the repository's root directory". For example, you can use the ^/tags/big sandwich/ to refer to the URL of the /tags/big sandwich directory in the root of the repository. Note that this URL syntax works only when your current working directory is a working copy—the command-line client knows the repository's root URL by looking at the working copy's metadata. Also note that when you wish to refer precisely to the root directory of the repository, you must do so using ^/ (with the trailing slash character), not merely ^.

Subversion Working Copies

A Subversion working copy is an ordinary directory tree on your local system, containing a collection of files. You can edit these files however you wish, and if they're source code files, you can compile your program from them in the usual way. Your working copy is your own private work area: Subversion will never incorporate other people's changes, nor make your own changes available to others, until you explicitly tell it to do so. You can even have multiple working copies of the same project.

After you've made some changes to the files in your working copy and verified that they work properly, Subversion provides you with commands to "publish" your changes to the other people working with you on your project (by writing to the repository). If other people publish their own changes, Subversion provides you with commands to merge those changes into your working copy (by reading from the repository).

A working copy also contains some extra files, created and maintained by Subversion, to help it carry out these commands. In particular, each working copy contains a subdirectory named .svn, also known as the working copy's *administrative directory*. The files in the administrative directory help Subversion recognize which of your versioned files contain unpublished changes, and which files are out of date with respect to others' work.

Prior to version 1.7, Subversion maintained .svn administrative subdirectories in every versioned directory of your working copy. Subversion 1.7 offers a completely new approach to how working copy metadata is stored and maintained, and chief among the visible changes to this approach is that each working copy now has only one .svn subdirectory which is an immediate child of the root of that working copy.

While .svn is the de facto name of the Subversion administrative directory, Windows users may run into problems with the ASP.NET Web application framework disallowing access to directories whose names begin with a dot (.). As a special consideration to users in such situations, Subversion will instead use _svn as the administrative directory name if it finds a variable named SVN_ASP_DOT_NET_HACK in its operating environment. Throughout this book, any reference you find to .svn applies also to _svn when this "ASP.NET hack" is in use.

www.vskills.in

How the working copy works

For each file in a working directory, Subversion records (among other things) two essential pieces of information:

- ✓ What revision your working file is based on (this is called the file's working revision)
- ✓ A timestamp recording when the local copy was last updated by the repository

Given this information, by talking to the repository, Subversion can tell which of the following four states a working file is in:

Unchanged, and current

The file is unchanged in the working directory, and no changes to that file have been committed to the repository since its working revision. An svn commit of the file will do nothing, and an svn update of the file will do nothing.

Locally changed, and current

The file has been changed in the working directory, and no changes to that file have been committed to the repository since you last updated. There are local changes that have not been committed to the repository; thus an svn commit of the file will succeed in publishing your changes, and an svn update of the file will do nothing.

Unchanged, and out of date

The file has not been changed in the working directory, but it has been changed in the repository. The file should eventually be updated in order to make it current with the latest public revision. An svn commit of the file will do nothing, and an svn update of the file will fold the latest changes into your working copy.

Locally changed, and out of date

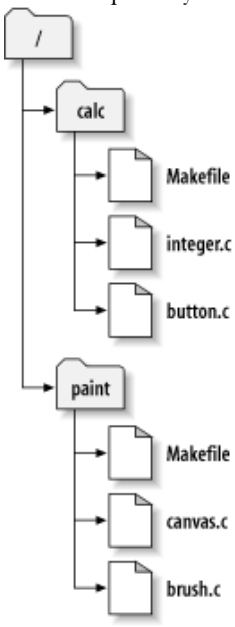
The file has been changed both in the working directory and in the repository. An svn commit of the file will fail with an "out-of-date" error. The file should be updated first; an svn update command will attempt to merge the public changes with the local changes. If Subversion can't complete the merge in a plausible way automatically, it leaves it to the user to resolve the conflict.

Fundamental working copy interactions

A typical Subversion repository often holds the files (or source code) for several projects; usually, each project is a subdirectory in the repository's filesystem tree. In this arrangement, a user's working copy will usually correspond to a particular sub tree of the repository.
For example, suppose you have a repository that contains two software projects, paint and calc. Each project lives in its own top-level subdirectory, as shown in Figure 1.7, "The repository's file system"

www.vskills.in

Figure 1.7. The repository's filesystem



To get a working copy, you must check out some subtree of the repository. (The term check out may sound like it has something to do with locking or reserving resources, but it doesn't; it simply creates a working copy of the project for you.) For example, if you check out /calc, you will get a working copy like this:

```
$ svn checkout http://svn.example.com/repos/calc
A    calc/Makefile
A    calc/integer.c
A    calc/button.c
Checked out revision 56.
$ ls -A calc
Makefile  button.c integer.c .svn/
$
```

The list of letter As in the left margin indicates that Subversion is adding a number of items to your working copy. You now have a personal copy of the repository's /calc directory, with one additional entry—.svn—which holds the extra information needed by Subversion, as mentioned earlier.
Suppose you make changes to button.c. Since the .svn directory remembers the file's original modification date and contents, Subversion can tell that you've changed the file. However, Subversion does not make your changes public until you explicitly tell it to. The act of publishing your changes is more commonly known as *committing* (or *checking in*) changes to the repository.
To publish your changes to others, you can use Subversion's svn commit command:

```
$ svn commit button.c -m "Fixed a typo in button.c."
Sending        button.c
Transmitting file data .
Committed revision 57.
$
```

www.vskills.in

Now your changes to button's have been committed to the repository, with a note describing your change (namely, that you fixed a typo). If another user checks out a working copy of /calc, she will see your changes in the latest version of the file.

Suppose you have a collaborator, Sally, who checked out a working copy of /calc at the same time you did. When you commit your change to button's, Sally's working copy is left unchanged; Subversion modifies working copies only at the user's request.
To bring her project up to date, Sally can ask Subversion to *update* her working copy, by using the svn update command. This will incorporate your changes into her working copy, as well as any others that have been committed since she checked it out.

```
$ pwd
/home/sally/calc
$ ls -A
Makefile button.c integer.c .svn/
$ svn update
Updating '.':
U    button.c
Updated to revision 57.
$
```

The output from the svn update command indicates that Subversion updated the contents of button.c. Note that Sally didn't need to specify which files to update; Subversion uses the information in the .svn directory as well as further information in the repository, to decide which files need to be brought up to date.

Mixed-revision working copies

As a general principle, Subversion tries to be as flexible as possible. One special kind of flexibility is the ability to have a working copy containing files and directories with a mix of different working revision numbers. Subversion working copies do not always correspond to any single revision in the repository; they may contain files from several different revisions. For example, suppose you check out a working copy from a repository whose most recent revision is 4:

```
calc/
Makefile:4
integer.c:4
button.c:4
```

At the moment, this working directory corresponds exactly to revision 4 in the repository. However, suppose you make a change to button.c, and commit that change. Assuming no other commits have taken place, your commit will create revision 5 of the repository, and your working copy will now look like this:

```
calc/
Makefile:4
integer.c:4
button.c:5
```

www.vskills.in

Suppose that, at this point, Sally commits a change to integer.c, creating revision 6. If you use svn update to bring your working copy up to date, it will look like this:
calc/

Makefile:6
integer.c:6
button.c:6

Sally's change to integer.c will appear in your working copy, and your change will still be present in button.c. In this example, the text of Make file is identical in revisions 4, 5, and 6, but Subversion will mark your working copy of Make file with revision 6 to indicate that it is still current. So, after you do a clean update at the top of your working copy, it will generally correspond to exactly one revision in the repository.

Updates and commits are separate

One of the fundamental rules of Subversion is that a "push" action does not cause a "pull" nor vice versa. Just because you're ready to submit new changes to the repository doesn't mean you're ready to receive changes from other people. And if you have new changes still in progress, svn update should gracefully merge repository changes into your own, rather than forcing you to publish them. The main side effect of this rule is that it means a working copy has to do extra bookkeeping to track mixed revisions as well as be tolerant of the mixture. It's made more complicated by the fact that directories themselves are versioned.
For example, suppose you have a working copy entirely at revision 10. You edit the file foo.html and then perform an svn commit, which creates revision 15 in the repository. After the commit succeeds, many new users would expect the working copy to be entirely at revision 15, but that's not the case! Any number of changes might have happened in the repository between revisions 10 and 15. The client knows nothing of those changes in the repository, since you haven't yet run svn update, and svn commit doesn't pull down new changes. If, on the other hand, svn commit were to automatically download the newest changes, it would be possible to set the entire working copy to revision 15—but then we'd be breaking the fundamental rule of "push" and "pull" remaining separate actions. Therefore, the only safe thing the Subversion client can do is mark the one file—foo.html—as being at revision 15. The rest of the working copy remains at revision 10. Only by running svn update can the latest changes be downloaded and the whole working copy be marked as revision 15.

Mixed revisions are normal

The fact is, every time you run svn commit your working copy ends up with some mixture of revisions. The things you just committed are marked as having larger working revisions than everything else. After several commits (with no updates in between), your working copy will contain a whole mixture of revisions. Even if you're the only person using the repository, you will still see this phenomenon. To examine your mixture of working revisions, use the svn status command with the --verbose (-v) option (see the section called "See an overview of your changes" for more information).
Often, new users are completely unaware that their working copy contains mixed revisions. This can be confusing, because many client commands are sensitive to the working revision of the item they're examining. For example, the svn log command is used to display the history of changes to a file or directory (see the section called "Generating a List of Historical Changes"). When the user

invokes this command on a working copy object, he expects to see the entire history of the object. But if the object's working revision is quite old (often because svn update hasn't been run in a long time), the history of the older version of the object is shown.

Mixed revisions are useful

If your project is sufficiently complex, you'll discover that it's sometimes nice to forcibly backdate (or update to a revision older than the one you already have) portions of your working copy to an earlier revision; you'll learn how to do that in
Perhaps you'd like to test an earlier version of a sub module contained in a subdirectory, or perhaps you'd like to figure out when a bug first came into existence in a specific file. This is the "time machine" aspect of a version control system—the feature that allows you to move any portion of your working copy forward and backward in history.

Mixed revisions have limitations

However you make use of mixed revisions in your working copy, there are limitations to this flexibility.
First, you cannot commit the deletion of a file or directory that isn't fully up to date. If a newer version of the item exists in the repository, your attempt to delete will be rejected to prevent you from accidentally destroying changes you've not yet seen.
Second, you cannot commit a metadata change to a directory unless it's fully up to date. You'll learn about attaching "properties" to items in
A directory's working revision defines a specific set of entries and properties, and thus committing a property change to an out-of-date directory may destroy properties you've not yet seen.
Finally, beginning in Subversion 1.7, you cannot by default use a mixed-revision working copy as the target of a merge operation. (This new requirement was introduced to prevent common problems which stem from doing so.)

www.vskills.in