The guide to source control

SAMPLE CHAPTER

Subversion IN ACTION

Jeffrey Machols

MANNING

Subversion in Action

by Jeffrey Machols

Chapter 3

contents

3 Managing your working copy

In this chapter

- Get the state of the working copy and repository
- Keep your working copy up-to-date
- Get specific versions of files
- Add, move, and delete files in the repository

By now you should have a good understanding of the basic development cycle in Subversion. You check out the repository, make your changes in the working copy, and then commit them back into the repository. Now it's time to throw some real-world issues into the mix, such as multiple users making changes and file manipulation in the repository. In a situation with multiple developers, you need to deal with the repository and your working copy getting out of sync. In this chapter, we will explore how to check the state of files in the working copy. We will also see how to get things back in sync—or get back to a specific revision if necessary.

Part of managing your working copy and repository is managing your files. Think back to chapter 1 when we compared a version control system to a regular filesystem. Normally you would be able to copy, move, and delete files in a filesystem, which can be an important part of managing your code. When you add the extra versioning and change log capabilities of a repository, these common file-manipulation tasks can get tricky. We will examine how Subversion handles these operations and some of the side effects that come with them. For those of you coming from a CVS background, this will be pleasant change.

3.1 Checking the state of your working copy

As you know by now, Subversion uses a copy-modify-merge checkout strategy, which has the problem of the repository and working copy becoming out of sync. In chapter 2, we saw how to keep these in sync using the `checkout`, `commit`, and `update` commands. What if you want to see the state of your working copy before taking any action? Subversion provides the `svn status` command that will show you any differences between the working copy and the repository. Your first thought may be "big deal, either a file is in sync or it's not." This is not the case; a file or directory can be in multiple states. The results from this command can be a little cryptic at first, so it may take some getting used to.

3.1.1 Understanding the status codes

The `status` command returns a set of codes that will tell you exactly what the situation is with the particular file or directory. The codes consist of five columns of characters. Each column relates to a specific piece of information.

Column 1—File contents. The first column is the one you will focus on the most; it shows the status of the file contents. This will tell you whether anything inside the file has been modified. Table 3.1 lists a set of characters that can be in this field to represent the file state.

Table 3.1 File content status codes

Code	Target Object	Description
A	File or directory	The file or directory is scheduled to be added to repository from the working copy.
C	File only	The file in the working copy has been changed and is in a conflicting state with the repository. You will get this code when you run an update and you have been making changes to a file that has been changed in the repository. Subversion has failed to automatically merge the changes, so manual intervention is required before updating or committing.
D	File or directory	The object has been deleted from the working copy and will be removed from the repository on the commit.
M	File only	The contents of the file have been modified. The new revision will be put in the repository in the next commit.
X	Directory only	The directory is non-versioned, but it contains a group of checkouts from an external definition. (See chapter 6 for more details on external definitions.)
?	File or directory	The object in question is a non-versioned file or directory. It is not in the repository and is not scheduled to go in it.
!	File or Directory	The object is in the repository as a versioned file or directory but is not in the working copy. This will happen if you remove the object outside of Subversion. An update will restore the object into the working copy.
~	File or Directory	The name of the object exists in the repository, but the object type is different. For example, if you remove a file and add a directory with the specified name outside of Subversion, you will get this status. You will need to manually remove or rename the working copy object and run an update to get back in sync.

Some of these status codes may seem a little strange, but don't worry. As we move forward and talk about different operations, they will become clear to you. Also, this a comprehensive list that the `status` command can return. Some of the states are not common; for instance, you would really have to work at getting something to come back with the ~ code. For now, focus on the first four or five codes, and then you can revisit this list as needed.

Column 2—Properties status. The second column in the status output tells you the status of the properties for that file or directory. The only possible values are `M` or a blank space. If the column has the `M` character, one or more properties have

changed. Take a look at the following output to see how the different characters line up:

```
A     file.c
 M    main.c
```

As you can see, the file main.c has its second column populated with an M. This means the contents have not changed, only the properties. If the contents had changed along with the properties, the output would have looked like this:

```
A      file.c
MM     main.c
```

In this output, both the first and second columns have a status code. While you can clearly see this when multiple files are in the output of the status, it is more difficult when only one file is displayed with a properties change. Be sure to verify which field the `M` character is in before proceeding with your commit or update.

Column 3—The lock status. For the most part, you will not see this column populated by the `status` command. This field will tell you if the file or directory is locked. But how can that be—working copies do not check out with locks, right? While this is true, Subversion does need to lock the objects as they are committed to the repository. This is how it can prevent users from overwriting changes should they commit at the same time. This situation is really no different than a lock when you are writing to a relational database. The commit should happen in a timely manner so the files are locked for a very short period of time.

If the object is locked, an `L` character will be displayed in the third column. If the object remains in this state for any length of time, one of two things is likely happening. First, someone may be in the middle of editing a log message. When you run a commit and the editor starts, the files being saved are placed in a locked state. After the editor is closed and the commit finishes executing, these locks will be released. If the files are locked and there is no Subversion command running, they are in a bad state. This usually happens on a failed or interrupted commit. Remember that Subversion uses atomic commits, so these defunct locks are only in the working copy; the repository is not affected. You can use the `svn cleanup` command to remove false locks on files. We will explore this command when talk about repository administration in chapter 8.

Column 4—History is coming. The fourth column indicates whether the file or directory will have more change logs associated with the file than just the last modification. The most common way for this to happen is by using the Subversion `move` or `copy` command. While we will talk about this in more detail later in the chapter,

for now just remember that these operations will create a new object with the history of change logs attached. When an object has this additional history, a + character will be in the fourth column. An object that has the history status code will be tied to one of three content codes described in table 3.2.

Table 3.2 Additional history status codes

Code	Description
A +	The file or directory has been scheduled to be added to the repository with additional history. This will come from an `svn move` or `copy` command.
+	When no content indicator character is shown, the object itself has not changed, but it is part of directory structure that has. So if you move the directory `java`, all the files under it will have this status.
M +	First, this object is part of a directory structure that has changed as in the previous status code. In addition, the object has been modified.

Column 5—Switching paths. The fifth column in the status will tell you whether or not the object is pointing to the same URL as the rest of the subdirectory. Subversion allows you to "switch" where the working copy files point to in the repository; this is mainly used for branching. For example, if you edit the files in the `java` directory, but you want to save them to another location in the repository, you can run the `svn switch` command. When you do this, the files get a special status stating that they have been relocated. This status will be identified by the `S` character in the fifth column of the status output. We will describe the switch feature when we talk about branching in chapter 5.

3.1.2 Running the status command

Now that you're armed with enough information to decode the status symbols, it's time to run the `status` command. Like all Subversion commands, the default operation is to recursively parse through the directory tree from the current location. The command will find all the files that are not in sync with the repository. If a file or directory is not displayed in the output, it is has not been changed in your working copy.

```
$ svn status
?      .project
A      source/java/calc.java
M      source/java/hello.java
```

Out of all the files in the working copy, only three of them are different between the working copy and the repository. The `.project` file is a non-versioned file and

will not have any action taken on the next commit. Because the command is recursive, the entire path of the file is shown in the output, which is the case with `calc.java` and `hello.java`. As you can see by the indicators, `calc.java` has been added to the working copy and `hello.java` has been modified in it. The add, delete, and modify status codes will also indicate the action that will be taken on the next commit. Running this command from the top level of the working copy is great if you have only a few changes. Since it runs recursively, this command can give you massive amounts of output if you have lots of modified files.

3.1.3 Trimming your file list

Subversion not only runs through your directory structure to find all the files out of sync with the repository, it also gives you all non-versioned files. This will increase the number of files in the output of the `status` command even more. Throughout the course of development, you will end up with files in the working copy that do not go into the repository. For example, you may write output of a program to a log file. Another example of a non-versioned file is a project file from Eclipse. Since you will be developing in the working copy, you may need an IDE configuration file with your code. Depending on the situation, you may not want all these files in the output of the `status` command. This is especially true if you are focusing on one area of the working copy. Since by definition a non-versioned file is not in the repository, it will never be in sync, so it will always show up. Also, files in other directories may not be of interest for your current task. Subversion gives you the ability to hone in on the area you want in order to make the output of the `status` command more readable.

Ignoring non-versioned files in the output. You may be in a situation where your working copy requires a great number of non-versioned files. This can happen if your compiled artifacts are created in a versioned directory. While this is not a problem, it can make the output difficult to view. To illustrate this, look at the following output snippet from a `status` command:

```
$ svn status
?      .project
?      source/java/SingleReplyRequest.class
?      source/java/ModifyResponse.class
M      source/java/hello.java
?      source/java/AbstractSource/java.class
?      source/java/AbandonRequest.class
...
?      source/java/Source/javaEncoder.class
?      source/java/ExtendedRequestImpl.class
?      source/java/CompareRequest.class
?      source/java/Referral.class
```

As you can see, all the non-versioned files cloud the results you are looking for. In order to see what changed for the upcoming commit, you would have to sort all the files with the `?` character. Whether the files are compiled objects, IDE configurations, or any other type of file, you can ignore them by using the `--quiet` switch. Now let's see how the output from the same directory looks:

```
$ svn status --quiet
M      message/AddResponse.java
```

This output is more concise and better for finding the changes. You do not have to sift through files that are not intended for the repository. If you have files that will always be in the working copy, such as the Eclipse `.project` files that you want the `status` command to skip, you can use the `svn:ignore` property. This will give Subversion a list of files to skip even though they are in the working copy. See section 6.5.1 for more information on the `svn:ignore` property. If the bulk of the output you are trying to clean up is from non-versioned files, the `ignore` will be all you need. However, if you still have too many files, the next step would be running the command with a narrower scope.

Refining the path. If you have made a large number of changes in your working copy, you may not want to see all the results at once. For example, if you have made a dozen bug fixes, you may want to commit them individually for tracking and back-out purposes. Since the `status` command gives you the state of your working copy, it will actually tell you what the commit will do. You can use this as a verification that the commit will do what you are expecting. If you are going to commit with a specific path and no recursion, you probably will want to verify with the same options on the `status` command. The `status` command will accept a specific path to run against, instead of the default current directory. So if we run the status from the same location, but with a path specified, we will get only the files from that starting point:

```
$ svn status source/java
M      source/java/hello.java
```

The target can be a file or a directory. If you choose a directory, the command will start there and work down, just as if you did a `cd` and ran the command without any options. When you specify a file as the target, you will get output only if that one file has changed.

If you do not wish to have the command recursively search through the directory structure, add the `--non-recursive` option. This will give you files only at the

level it is run from, so when we run it at the top level of our repository, we will only get files changed there:

```
$ svn status --non-recursive
?      .project
```

The important thing to keep in mind is that almost all Subversion options have the same behavior. Since the `commit` and `status` commands have the `--non-recursive` option and the ability to accept a specific path, you can use the same command-line arguments for both. It is a good practice to check the status before committing. When you run the status with the same arguments you are going to supply to the commit, you will see exactly what actions will be taken against the repository.

3.1.4 Getting more information

There may be times when you need to see additional information than what the default status operation gives you, which is accomplished by adding the `--verbose` option. Using this option will output all files in the working copy, even if they are in sync with the repository. It may not be wise to run the status at the top level of your working copy because of the potential mass amount of output it will generate. Instead, start the command in the directory on which you are focusing:

```
$ svn status --verbose
?                                    .project
              13       13 jeff        .
              13       13 jeff        source
              13       13 jeff        source/java
              13       13 jeff        source/java/hello.java
              13        7 alex        source/java/main.java
              13       11 jeff        source/c
M             13       10 jeff        source/c/hello.c
              13        9 alex        source/c/main.c
```

This switch gives three extra fields in addition to the status code and filename. The first number is the revision of the repository from which you are working. This will be the last time you ran a checkout or update. The second number is the last revision of the repository in which the file or directory was changed, followed by the user who made the change. For example, the file `source/java/main.java` was last changed in revision 7 by the user `alex`.

3.1.5 Checking ignored files

We talked earlier in section 3.1.3 about telling Subversion to skip certain files when running the status by setting the `svn:ignore` property. When you do this, the command will not show these files. But what if you want to view them? It

would not be efficient to remove the `ignore` just to run a status, so the command has an option called `--no-ignore`. You guessed it, this will override the `svn:ignore` property and display all files out of sync, even the ignored ones. Let's assume you added the file `.project` to the ignored list. Take a look at the output when running the command with the `--no-ignore` switch:

```
$ svn status --no-ignore
I      .project
M      source/c/hello.c
```

The ignored file now shows up the output, but notice the status code. Before you added it to the ignored list, it had a code of `?`; now there is an `I`. This indicates the file is normally in an ignored list, and nothing will be done on the commit with that object. Using this option is a one-time event, meaning that the `svn:ignore` property will not be affected. If you run the command again without the switch, the ignored file will not be displayed.

3.1.6 Repository changes

So far, everything we have seen with the `status` command has related to changes in the working copy. What if you want to check to see if a file has been modified in the repository? You can use the `status` command to do this by adding the `--show-updates` option. This will not only find local objects that have changed, it will also let you know if anything in the working copy is out of date:

```
$ svn status --show-updates
      *       13   source/java/hello.java
M             13   source/c/hello.c
M     *       13   source/c/main.c
Status against revision:     14
```

We now show the file `source/java/hello.java` in the output with an `*`. Notice that there is no content status code for that file, though. This means that the file was changed in the repository and our working copy version is out of date. The number displayed for each object in the listing is the working copy revision; in the example we were working off revision 13. The last line will display the latest revision of the repository. In the example, the repository is at revision 14, which means another user has modified the `hello.java` file and committed it since we have done an update. The file `main.c` has an `M` code as well as the `*`, which means that you have made changes to the file and someone else has committed it to the repository since your last update. Before your changes can be saved, you will have to run the `update` command to get the working copy in sync with the changes from the repository.

3.1.7 When to use the status

We have shown you many different ways the `status` command can be run. Each way will give you a slightly different view into the condition of your working copy relative to the repository. You cannot get the entire picture by running the command only one way. While you may not need all of this information all the time, it is important to understand how to get it. If you get into the habit of running the status at key points in your development, you will have a better grasp of the code and will run into fewer surprises. Let's take a look at the major points when the `status` command can help you out.

Before you commit. Running the `status` command before you commit will show you what changes you are going to save to the repository. We showed that the status will accept the same command-line arguments as the commit, so you have the opportunity to see exactly what the commit is going to do. So why is this important? Let's say you have been working on a bug fix that touched two files, `hello.c` and `main.c`. You also made an unrelated feature change to the file `hello.java` in a different location of your working copy a while back that you forgot about. If you just run a commit from the top level of the working copy, all three files will be saved to the repository in one change and one log message. Now if you are doing research on a potential problem, or have to back out of a revision, you will affect two independent sets of changes. By running the status before the commit, you will see all the files that are going to be saved and can catch extra files that you did not want to commit as one change.

Before you update. As you will see in more detail in the next section, when you run the `update` command, any changes from the repository are applied to your working copy. You can run into a potential problem if you have made changes to the same file as someone else. Assume you have been making changes the file `main.c` and still have some work left to do for a critical fix. In the meantime Alex has made some small changes to a set of files to fix style errors. It is possible that you have changed the same line in the file and may have a conflict. By running the `status` command, you can tell if there will be a conflict and make the decision to resolve it now or later. If you know the changes Alex made were only cosmetic and you are in the middle of intense coding and testing, it may be more efficient to hold off on the update. No matter what situation you are in, this will at least give you a choice as to when you want to deal with the conflict.

Before you start modifications. In the previous examples, we used the `status` command to check the state of files when the working copy has changed. Even if you

have not made any modifications to the working copy, it is a good idea to run the status before you start editing source code. This will help you find any dependency changes before going down the wrong path. For example, let's say you are implementing a feature that uses a structure defined in a C header file. You will want to see if anyone changed that structure before you start using it.

3.2 Updating your working copy

By running the `status` command, we saw situations where the repository had been changed, which made the file in our working copy "stale." Subversion will consider your copy of the file or directory out of date since there is a newer version. The `update` command is your way of keeping the working copy in sync with changes to the repository. When you run this command, your working copy will be brought to the latest version of the repository. Subversion does not copy everything, as in the case of a checkout. Only changes to the repository made since your last update will be transferred to the working copy. This makes the update much more efficient, especially if the repository is large or you are using a network protocol.

You should run the update frequently and always before you edit the working copy. If you run this command before starting to make changes, the chances of running into conflicts will be greatly reduced.

3.2.1 Running the update command

The simplest way to update your working copy is to run the `svn up` command from the top-level directory. Without any options this command will find all the changes in the repository and update the files and directories in your working copy.

```
$ svn up
A  source/java/log.java
U  source/c/hello.c
U  source/java/main.java
Updated to revision 16.
```

Here there were three changes propagated to your working copy. Just like the `status` command, the `svn up` command will display a code describing the action taken. In this example, the file `log.java` was added to the working copy, while the files `hello.c` and `main.java` had their contents updated. At this point, your working copy will be in sync with all the repository changes.

A note about the revision number. Notice in the previous example that the `update` command tells you to which revision number the working copy has been updated.

Remember, the revision number is based on the repository, not individual files. This means all the files in the working copy have been updated to revision 16. Since files that do not change between versions of the repository are identical copies (actually they are links), there is no need to transfer them. Also, you may have noticed in the output of the update that there are characters preceding the filename, similar to with the `status` command. Just when you thought we were finished with all those cryptic codes, they are back.

3.2.2 Update status codes

The status codes you see next to the filename in the output of the `update` command describe the operation taken on the working copy. With the `status` command, the codes tell you what is scheduled for the next commit. When you run an `update` command, the code displayed is what the command has just done.

The `update` command uses an abbreviated form of the status code display, showing just the first two columns. The first describes any changes in the file contents. The second field shows changes to the properties of the file or directory. There is more good news—the two columns from the update have the same set of possible status codes. Table 3.3 shows the possible status codes the `update` command can return.

Table 3.3 Update status codes

Code	Description
A	The file or directory has been added to the working copy.
D	The file or directory has been deleted from the working copy.
U	The file or directory has been changed in the working copy and you had not made any local changes to this object.
C	The file has changed locally in the working copy and in the repository. Subversion was not able to automatically combine the two changes, so manual intervention is required.
G	The file has changed locally in the working copy and in the repository, but Subversion has been able to automatically merge the changes.

3.2.3 Updating to specific revisions

Remember that the `update` command is just a specialized checkout to get only changes. So if you need to get a previous version of a file or a set of files into the working copy, you can still use `svn update` instead of checking out a whole new copy of the repository. You can apply the `--revision` option, just as we did in the checkout. The revision number, date, and the keywords discussed in section 2.4.3

are all available to specify which version you get. For example, if you wanted to get revision 15 of the repository, simply run `update` from the top of the working copy and add the `--revision` option:

```
$ svn up --revision 15
UU source/java/hello.java
U  source/java/main.java
D  source/java/log.java
Updated to revision 15.
```

The command has reported back that three files needed to be adjusted in the working copy to get it to match the repository at revision 15. The file `main.java` simply needed the file contents adjusted to the older version. The file `hello.java` not only needed to bring its contents back to an older revision, but the properties have also been restored. Remember that this is a key feature in Subversion to truly give you the ability to see what the source code looked like at a previous point in time. Notice that the file `log.java` has a `D` code, which means that the file did not exist in the repository at this revision, so it has been removed from the working copy.

Going backwards in an update is no different than updating to the latest version of the repository. Only the files that have differences will be transmitted; the rest are identical so they do not need to be updated. Instead of all the new changes being applied to the working copy, modifications are simply backed out. You can infer from this example that all the other files and directories in the working copy have not been changed since revision 15. Updating to older versions of the repository is a fairly simple and safe operation. That being said, there a few potential bumps in the road, but these can easily be avoided.

Commit before updating to older revisions. First, it is highly recommended that you commit all your working copy modifications before updating to an older revision. If you do not, things can get extremely complicated in a hurry, and you could run into constraints that will prevent Subversion from getting the old revisions. Consider the file `log.java` from the previous example. Say you have made some changes to in the working copy and you try to update to revision 15:

```
$ svn up --revision 15
UU source/java/hello.java
U  source/java/main.java
svn: Won't delete locally modified file 'log.java'
```

There is a problem because you have made changes to something that did not exist in revision 15 of the repository. Just like in the movie *Back to the Future*, we have screwed up the space-time continuum, and now Subversion is confused. You should avoid having locally modified copies when updating to older revisions.

Going back too far. Another potential issue with updating to older copies is going way back in time. Since an update operates on an existing working copy, the changes are applied to a current directory structure. If the majority of your commits were simply modifications to file contents, there should not be a problem. If you have added or moved directories, especially ones that contain non-versioned files, you may run into some issues. Let's see what happens when we try to get back to revision 10 of the repository we have been working on:

```
$ svn up -r 10
svn: Won't delete locally modified directory ''
svn: Left locally modified or unversioned files
```

When you see this message, Subversion is telling you the current working copy directory structure can't get back to the revision of the repository you specified. If you need to go way back in time, your best bet is to just check out a new working copy. The more you move files and directories around, the more difficult it will be to update your working copy backwards in time. But don't let this discourage you—if you cannot update, you can always check out a new copy.

Don't forget that the reason to do an update versus a checkout is efficiency. If you need to go back many revisions, there will likely be many changes that need to be transferred. When this is the case, you will lose some of the benefits of the update, and so a new checkout is just as good. The only caution to this concerns any non-versioned files. If you perform a new checkout on a different local directory, you will not get any of these files. To get around this situation, you will need to manually move the non-versioned files. Now that you know what pitfalls can exist, there are some other directions you can take when updating to specific revisions of the repository.

You don't have to go back. So far we have talked only about using the update to get previous revisions of the repository, but that is not the only direction you can go. Let's say your working copy is on revision 20 of the repository, and there has been a flurry of activity. When you run a status against the repository, it looks like this:

```
$ svn status --show-updates
M      *       20   source/java/hello.java
M      *       20   source/java/main.java
Status against revision:     30
```

The latest revision of the repository is now 30, so you are 10 revisions behind. But what if you don't want to take on all the changes at once? For example, you may want to work off only revision 25 for now, because anything more will be too many changes at one time. You can still use the `--revision` option to move forward in increments:

```
$ svn up --revision 25
U  source/java/hello.java
U  source/java/main.java
Updated to revision 25.
```

You have now brought the working copy up to revision 25 of the repository. From Subversion's point of view, incremental updates are not an issue, but this can give you development headaches. If you are staying behind the latest revision all the time, you are developing against stale code. You may miss dependencies changes and will increase the chances of conflicts in your updates. You will also need to be careful about saving your changes. The next section describes some of the potential problems when you commit from old versions.

Commiting from different revisions. If you are updating to noncurrent revisions of the repository to simply view source code or replicate a time point, you do not need to worry about committing issues. When you are finished with the older copy, simply run a default update to get back to the most recent revision. If you need to commit, there may be some problems. Your changes will be saved to a new revision based on the latest repository, not the one you have checked out. So from our previous example, let's try to run a status to see what has changed between the working copy, revision 25, and the latest repository, revision 30:

```
$ svn status --show-updates
      *      25   source/java/hello.java
      *      25   source/java/main.java
      *      25   source/c/main.c
Status against revision:     30
```

Between revisions 25 and 30, three files have been modified. It is important to know this before you start to make changes. If the file you need to change is not in this list, you can make changes without worrying about conflicts, since the working copy file is identical to the repository version. So if you want to make changes to the file `source/c/hello.c`, you are in the clear. After you make the changes and commit, look at what happens:

```
$ svn commit --message "Added function call to logger"
Sending       trunk/source/c/hello.c
Transmitting file data .
Committed revision 31.
```

Notice that even though we updated the working copy to revision 25, the commit produced revision 31 of the repository. This happens because a commit will always create a new revision based on the latest repository, regardless of what you have in your working copy.

Now what if you need to change one of the files in the list that were out of date. Your best bet is to update that file to the latest revision and then start the modifications. If you change the file and try to commit, you will be forced to run an update anyway. It is better to get the newest changes from the start and just work from there. If you do this, all the changes to the file will be in your version already, and you will not have to worry about resolving potential conflicts.

3.3 Conflicts

You can be the most careful person and update your working copy all the time, but inevitably you will run into a situation where two people change the file at the same time. If Subversion cannot automatically combine the changes in the file, it will be in a state of conflict.

Suppose you made changes to two files in your working copy. It turns out that another user had also committed both files to the repository. You know they are now considered out of date, so you must run an update before Subversion will allow you to commit. Take a look at what the output from the update might look like:

```
$ svn up
G  trunk/source/c/main.c
C  trunk/source/java/hello.java
Updated to revision 32.
```

The file `main.c` has the status code `G`, which means Subversion successfully merged the two changes automatically. At this point, the file is no longer out of date and can be committed to the repository. You must remember that this merge does not perform any type of logic check; all it means is that Subversion was able to move the text around in the file without any conflicts. If the changes could not be combined by the system, the file would be placed in conflict status. This is the case with `hello.java` in the example. When a file comes back with this status, manual intervention is required to resolve the conflict. We will explore how to do this in the next few sections.

3.3.1 Resolution files

Subversion tries to take some of manual intervention out the process by creating a set of four files. Each of these files represents a solution to resolve the conflict; they are actually snapshots of the contents at key time points. Look at the `java` directory after the update has run:

```
$ ls -ltr
-rw-r--r--    1 jeff     svn            3 Apr  3 14:40 log.java
-rw-r--r--    1 jeff     svn           78 Apr  3 14:40 main.java
-rw-r--r--    1 jeff     svn           59 Apr  3 15:04 hello.java.r34
-rw-r--r--    1 jeff     svn           52 Apr  3 15:04 hello.java.r33
-rw-r--r--    1 jeff     svn           65 Apr  3 15:04 hello.java.mine
-rw-r--r--    1 jeff     svn          128 Apr  3 15:04 hello.java
```

Notice that there are four files with a base name of `hello.java`. The two files with `rXX` numbers at the end contain the file contents for those revision numbers. The lower number is the original file before either change was made. The higher number is the revised file that was saved to the repository. The file with the `.mine` extension is your local working copy after your modification. Finally, `hello.java` contains the diff of both sets of changes. Figure 3.1 shows the sequence of events that each of the files represents.

3.3.2 Resolution scenarios

Now that we have seen the files produced to help resolve the conflict, we need to look at the scenarios to know which file to use when. Once you find the scenario you want to use to resolve the conflict, simply copy one of the resolution files to back to the real file, depending on which fix you use. If you need some combination of the changes, you can manually edit the file to get the exact changes you want.

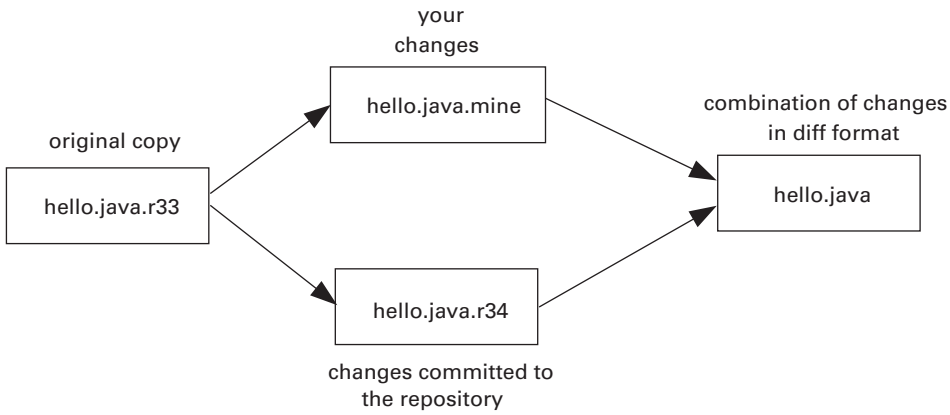

Figure 3.1 Events to get conflict-resolution files

Keep your changes. The first possibility is to keep only your changes and throw out the last edits to the repository. You local changes are stored in `filename.mine`. Using the previous example, you would just copy the file `hello.java.mine` to `hello.java`:

```
$ cp hello.java.mine hello.java
$ svn resolved hello.java
Resolved conflicted state of 'hello.java'

$ svn commit
Sending       hello.java
Transmitting file data .
Committed revision 35.
```

Even though you are "throwing out" the other users' changes, there will still be a copy of them. Remember that the current revision is 34, so when you commit your new file, it will be revision 35. You can get the changes back by checking out revision 34 again. Also notice that there is an additional command run called `resolved`. This is required before Subversion will allow you to commit and is explained in section 3.3.3.

Keep the repository changes. It is possible that two people were fixing the same problem, in which case you do not need to save your changes. You may need a better project manager, but that's another story. You could copy the file with the later revision number extension to the real file. So in our example you would copy `hello.java.r34` to `hello.java`. Just like the previous example, you would copy the file, run the resolved command, and then finally commit. All of these steps can be done at once with another Subversion command, `revert`:

```
$ svn revert hello.java
Reverted 'hello.java'

$ ls -l
-rw-r--r--    1 jeff     svn           70 Apr  3 16:08 hello.java
-rw-r--r--    1 jeff     svn            3 Apr  3 14:40 log.java
-rw-r--r--    1 jeff     svn           78 Apr  3 14:40 main.java
```

After you run this, `hello.java` will no longer have your changes. The file will match revision 34 from the repository. Keep in mind that once you do this, your local changes are gone and cannot restored because they were not saved to the repository. Also, notice the listing that followed the `revert` command: the temporary files have been cleaned up. This will prevent you from having to manually clean up the resolution files. Finally, you may have noticed that we did not

commit; this is because there was no need to. Since you are removing all the local changes, there is nothing to save to the repository.

Throw out all the changes. You may decide to get rid of both sets of changes and start from scratch. It could turn out that the reason for the conflict is that you and the other developer had different requirements. After getting together, you will likely realize that neither change is correct, so you will have revisit the technical design or specifications. If this is case, all you have to do is copy the file with the original revision number to the true filename. So in our example, you would copy `hello.java.r33` to `hello.java`. After you do this, you have no way to get back your local changes since they are not saved anywhere. You can, however, retrieve the changes from the other user since they will still be stored in revision 34:

```
$ cp hello.java.r33 hello.java

$ svn resolved hello.java
Resolved conflicted state of 'hello.java'

$ svn commit
Sending       hello.java
Transmitting file data .
Committed revision 35.
```

Notice that when you do this, a commit is required, and it is updated to revision 35. We are actually making an exact copy of revision 33 and moving to revision 35. This is necessary because revision 34 is in the repository and there is no way to get it out. The only way is to create a new revision without the changes.

Manually editing the diff file. The last possibility for resolving the conflict is that you need some combination of the two changes. If this is the case, you will need to manually edit the diff file created. The areas in the file that do not conflict will be left alone. The parts of the file where there is a problem will look a UNIX `diff` command. Let's take a look at `hello.java` from the example to see what the contents of the diff look like. First, we will look the locally modified file:

```
$ cat hello.java.mine
public class hello
{
  public static void main(String args[])
  {
    system.out.println ("hello world");
    System.out.println ("Goodbye");
  }
}
```

Next, look at the contents of the repository file. This is the version that another developer has made changes to and checked in:

```
$ cat hello.java.r34
public class hello
{
  public static void main(String args[])
  {
    this.some_function();
    system.exit (0);
  }

  public void some_function()
  {
     system.out.println ("hello world");
  }
}
```

Last, look at the file Subversion has created that contains the diffs. As you will see, both sets of changes are incorporated into this version:

```
$ cat hello.java
public class hello
{
  public static void main(String args[])
  {
<<<<<<< .mine
    system.out.println ("hello world");
    System.out.println ("Goodbye");
=======
    this.some_function();
    system.exit (0);
>>>>>>> .r34
  }

  public void some_function()
  {
     system.out.println ("hello world");
  }
}
```

In your favorite text editor, you will need to figure out which changes to keep and which to get rid of. Remember, only the lines inside the separators are in conflict, so the rest of the file should not need to be changed. There is one caveat to this, however; think back to chapter 1 when we talked about logical checks. Look at the previous code. Revision 34 of the file added the function `some_function()`, but this does not show up in the diff section. Since logical checks are not built into Subversion, it does not know that the function call and implementation have

a relationship. This file assumes that all changes outside of the conflict will be kept, so you need understand the entire change. Suppose you made your changes to the file, and now it looks like the following output:

```
public class hello
{
  public static void main(String args[])
  {
    system.out.println ("hello world");
    System.out.println ("Goodbye");
    system.exit (0);
  }

  public void some_function()
  {
     system.out.println ("hello world");
  }
}
```

The separators have been removed and the changes combined. Because the function section was not in the diff, the developer didn't realize that this was part of the previous version and left it in. In this case it won't hurt anything; there is just a function that is not called. Even if the change would cause the code not to compile or run, Subversion will not consider this a conflict as long as the text in the file can be reconciled. When you save the final modifications to the file, you can run the resolved command and commit.

3.3.3 Cleaning up the conflict

No matter which scenario you choose to resolve the conflict, you will need to clean up the temporary files that have been created. You could manually delete them when you are finished, but the file will still remain in a conflicting state. Once you have resolved the conflict with any of the scenarios discussed previously, you will need to run the `svn resolved` command. If you try to commit without doing this, Subversion will tell you that the problem hasn't been fixed yet:

```
$ svn commit
svn: Commit failed (details follow):
svn: Aborting commit: '/testrepo/source/java/hello.java' remains in conflict

$ svn resolved hello.java
Resolved conflicted state of 'hello.java'

$ ls -l
-rw-r--r--    1 jeff     svn           70 Apr  3 16:08 hello.java
-rw-r--r--    1 jeff     svn            3 Apr  3 14:40 log.java
-rw-r--r--    1 jeff     svn           78 Apr  3 14:40 main.java
```

```
$ svn commit
Sending       hello.java
Transmitting file data .
Committed revision 35.
```

Look closely at the sequence of commands here. Even though the file was modified and the diffs were resolved, Subversion needs to be told that this happened. Once you run the command, the system removes the temporarily conflicting files and leaves just the base file you started with. If you check the status of the file, you will also notice that it is no longer in a conflicting state; this is required to be able to commit. If you run the `svn revert` command to keep the original changes, you will not have to run `svn resolved` because it updates the status and also cleans up the file. In any other scenario, however, the `resolved` command will be required.

The first three sections have shown how to keep your working copy and repository in sync, but this only one part of managing your files. Another important aspect of administering files is "traditional" filesystem operations. You will need the ability to add, delete, and move files around in your repository.

3.4 Adding files and directories

The implementation of the commands to manipulate files in Subversion is very straightforward and intuitive. The first step in any filesystem implementation is adding new files or directories; this is no different in Subversion. We have shown that the `import` command is a good tool for loading non-versioned directory structures into the repository, but what about adding just a few or even one file to an established repository? If you create a new file or directory in your working copy, you can just use the `svn add` command to get it into the repository on the next commit.

3.4.1 Adding a single file

To add a new file to the repository, you will create it in the working copy directory in which you want it to reside. You can edit the file and put in as much content as you wish before adding it. When you have your initial copy ready, there is not a lot a magic involved; just run the `svn add` command with the file as the target. Suppose you have been editing a file named `goodbye.java` in your working copy and are at the point of doing the initial commit of the file. First we will look at the status of the file before the command:

```
$ svn status
?      goodbye.java
```

Since the file had not been added to the repository yet, it will have a `?` for the status character. Now when you run the `add` command, Subversion will change the status:

```
$ svn add goodbye.java
A         goodbye.java
```

At this point you have scheduled this file to be added to the repository; it is not actually in yet. Let's take a look at the status and see where it stands:

```
$ svn status
A      goodbye.java
```

Since the file came back from the `status` command, the working copy and repository are still out of sync. The status code is `A`, which means the file is scheduled to be added on the next commit. To finally get `goodbye.java` into Subversion, you will need to run a commit:

```
$ svn commit --message "Added goodbye.java"
Adding         java/goodbye.java
Transmitting file data .
Committed revision 6.
```

If you compare this output to a commit where a modify of an existing object was done, you will notice different text preceding the file. Instead of saying `Sending`, the output displays `Adding`. This is another sanity check to for you to see what action was taken for that file on the repository.

3.4.2 Adding a directory and its contents

Now you know how to to add a file; syntactically, running this operation on a directory is no different. Since directories are versioned, you will also need to explicitly add them the same way you do a file. You can create the directory in your working copy and even create new files and other subdirectories inside it. Remember, the default behavior of most Subversion commands is that everything is recursive. So when you add a directory, everything under it will also be added. For example, if you create a new directory called `gui` and add a set of files, just run the `add` command on the directory name. This will add the directory and its contents:

```
$ svn add gui
A         gui
A         gui/button.java
A         gui/window.java
A         gui/text.java
```

Not only is the directory scheduled, but all the files under it will also be added on the next commit. It is important to remember that you must add the directory

before adding the files. If you try to add the files first, Subversion will tell you that the current directory is not a working copy and you must add it.

Using mkdir to create and add in one step. If you like shortcuts, Subversion has provided a command to shave off some steps in getting directories into the repository. Instead of manually creating the directory on the OS side, you can use the `svn mkdir` command. There are multiple ways to use the command; the first is specify a path in the working copy. When you do this, Subversion will create the directory and run the `add` command for you:

```
$ svn mkdir gui
A         gui
```

Not only does this command make the directory `gui` in the working copy, it also schedules it to be added to the repository on the next commit. If you are still troubled because you have to run the commit, there is another way to run the command. Instead of specifying a path, you can specify the complete URL of the directory in the repository, and it will commit for you:

```
$ svn mkdir file:///repos/testrepo/source/java/gui2 \
 --message "Added gui2 Directory"

Committed revision 39.
```

Notice that we used the URL of the repository and then the path to the directory we wanted to create. We also had to specify the log message because this is an actual commit. All of the other commit options, such as authentication and log message editors, are available in the `mkdir` command. Even though the command commits to the repository, it does not update your working copy. If you do a listing, you will not see it until you do an update. This happens because when you access the repository through a URL, the `.svn` directories are not used, so Subversion does not know about your working directory.

Committing the directory only. If you do not use one of the methods to commit the directory and commit in one step, you could have a directory created that contains files. When you run the `add` command, all the files will be automatically added as well. If you do not want this happen, you can add the `--non-recursive` switch. This will act only on the files you specify at the same level. So for our `gui` example, if you just wanted the directory, the switch would do the trick:

```
$ svn add --non-recursive gui
A         gui
```

This time, Subversion did not get all the files underneath `gui`. This method is useful if you want to add a group of files or directories one level at time for backout or traceability reasons.

3.4.3 Wildcards

Most Subversion commands will accept wildcards for the target, but it usually does not make sense to use them. This is because most commands are operating on a specific file, not a generic group of files. The `add` command is an exception to this. For instance, if you add a group of Java files in a directory, instead of running an add on each of them, you can use a wildcard:

```
$ svn add *.java
A         button.java
A         main.java
A         scroll.java
```

All the files that end in `.java` were scheduled to be added to the repository. You can use the wildcards in any of the regular expression formats. While this works great if you have a bunch of files that are not versioned and you want all of them to be added, there are a couple of potential problems. First, if there are already versioned files, the command will try to add them again. For example, if you added three more files to the same directory and ran the add with a wildcard, you would get the following results:

```
$ svn add *
svn: warning: 'button.java' is already under version control
A         content.java
svn: warning: 'main.java' is already under version control
A         radioBox.java
svn: warning: 'scroll.java' is already under version control
```

The files that were already in the repository will just come back with an error; the other file will be added. This will not hurt anything, and if you can deal with the errors in the output, this is perfectly fine. You can run into a problem if you have non-versioned files that you do not want added. If you had build artifacts or project files for Eclipse, these would get caught and be added to the repository also. Wildcards can help out when adding a lot of files, but be careful that you are not getting more than you bargained for.

3.5 Copying

Subversion gives you the ability to copy files and directories in the repository. This is similar to making a copy from the operating system, but in addition to getting

the file contents, the change log information and properties are also copied. We will only skim the `copy` command in this chapter because its true use is in branching. There are few compelling reasons to copy files with the history outside of branching. If you do need to use the Subversion `copy`, be sure that you understand the change log and repository version implications.

3.5.1 Where is my previous version?

Remember, all transactions are treated the same so that they create a new revision of the repository. When you make a copy of an object, you will need to commit. Any version of the repository before that commit will not contain the copy. Say, for example, you copy `hello.java` to `goodbye.java` and commit; this creates revision 40 of the repository. If you check out revision 39 or below, you will not have `goodbye.java`. Looking at the history logs, this may be confusing. Let's look at the logs with the verbose output of the two files:

```
hello.java:
------------------------------------------------------------------------
r28 | alex | 2004-04-03 13:51:05 -0500 (Sat, 03 Apr 2004) | 1 line
Changed paths:
   M /source/java/hello.java

Added hello world print statement
------------------------------------------------------------------------
r16 | jeff | 2004-04-03 12:21:32 -0500 (Sat, 03 Apr 2004) | 1 line
Changed paths:
   A /source/java/hello.java

Added file
------------------------------------------------------------------------


goodbye.java:
------------------------------------------------------------------------
r40 | jeff | 2004-04-06 20:07:05 -0400 (Tue, 06 Apr 2004) | 1 line
Changed paths:
   A /source/java/goodbye.java (from /source/java/hello.java:39)

Made copy of hello to goodbye
------------------------------------------------------------------------
r28 | alex | 2004-04-03 13:51:05 -0500 (Sat, 03 Apr 2004) | 1 line
Changed paths:
   M /source/java/hello.java

Added hello world print statement
------------------------------------------------------------------------
r16 | jeff | 2004-04-03 12:21:32 -0500 (Sat, 03 Apr 2004) | 1 line
```

```
Changed paths:
   A /source/java/hello.java

Added file
------------------------------------------------------------------------
```

First, notice that revisions 16 and 28 show identical history. Remember that more than just contents come with the file, so the source file's history gets copied also. But if you look at `goodbye.java`, the history has an extra entry, revision 40. This is the point in the repository when the file was copied and committed. You can tell this because the operation status has an `A` for add, plus Subversion also tells you which file it was copied from and in which revision of the repository it happened. So `goodbye.java` was created from revision 39 of `hello.java`. Because of the atomic commits, revision 39 is the same as 28, which was the last change.

Are you sure you didn't want to do an operating system copy? What you have to be aware of as a user is that just because the history for `goodbye.java` shows a revision at 28 and 16, it does not mean that the file was in those revisions. You have to see that there was a copy in revision 40 that created the file. Since that file did not exist in revision 39 or before, `goodbye.java` will not be in those repository revisions, even though it looks like the history says it was. You need to keep this in mind if you decide to use the copy feature.

If you just want to replicate the file contents, an operating system copy and add are your best bet. This will create a new object but use the content of the source as a starting point. This object will have its own history and will not carry all the other baggage.

3.5.2 Making a local copy

When you do decide that the Subversion copy is the way to go for you, there are multiple ways of doing it. The "safe" way is to do this in your working directory, and then when you are happy with the changes, you can commit to the repository. In order to make a local copy, you use the same syntax and a regular UNIX copy, which is simply passing the source file and the target file. The source file must be in a working copy to use this method. To create the copy of `goodbye.java` in the previous example, run the following `copy` command:

```
$ svn copy hello.java goodbye.java
A         goodbye.java

$ svn status
A  +   goodbye.java
```

Just as with the `add` command, a new working copy file will be created; this is now a completely separate file. Notice the output from the `status` command that was run after the copy. It includes the + character, which indicates that there is additional history attached to the object. This should clue you in to the fact that the file was created with a copy or move. Since all changes to the repository are considered, a commit is required for this change to take effect.

Directories. This should not come as a big shock to you: when you copy a directory, it recursively copies all the files under it. The new files created are completely new entries but start with the contents and change logs from the respective sources:

```
$ svn copy gui testgui
A        testgui

$ ls testgui/
button.c  window.c

$ svn status
A  +   testgui
```

Even though the output shows only that it copied the directory, when we do a listing, all the files have been created. Also, take a look at the output from the `status` command. The only thing that shows up is the directory. Look at the difference when we run `status` in verbose mode:

```
$ svn status --verbose
A  +           -       58 jeff        testgui
   +           -       59 jeff        testgui/button.c
   +           -       59 jeff        testgui/window.c
```

The files appear in the list with the history column marked, but nothing else. This indicates that these files were created as a side effect of a copy. They were not explicitly added but are coming along for the ride with the parent.

3.6 Moving and renaming

In many version control systems, once a file or directory is the repository, it is difficult if not impossible to rename or move it. For CVS users, this has been a longtime complaint. By versioning directories, Subversion easily lends itself to renaming and moving files and directories, just as you would in a regular filesystem. When you move a file, the change logs go along with the contents. If you are new to version control systems, you may be thinking, "what's the big deal?" Just ask anyone who has tried to refactor code in CVS, and you will get a different response.

The `move` and the `rename` commands are the same command; in fact, you can move and rename at the same time. You start with a source file or directory and tell Subversion what its new name is. If the name is in a different directory, then you have also moved it.

3.6.1 Moving files in the working copy

Most users will use the default method of moving files: run the command on objects in your working directory. This is a safer way to move files because you can look at the actions in the working copy and make sure everything is right before committing the changes to the repository.

Rename only. To rename a file, simply run the `svn move` command from the directory in which the file resides. The two arguments will be the current name and the new name, respectively. Lets see what the command would look like to rename the file `hello.java` to `HelloWorld.java`:

```
$ svn move hello.java HelloWorld.java
A        HelloWorld.java
D        hello.java

$ svn status
A  +   HelloWorld.java
D      hello.java
```

By looking at the output, you should be able to figure out what Subversion is doing in a move operation. The original file `hello.java` is being removed, and the target file `HelloWorld.java` is being added. This is just in the working directory; you will have to commit to get the changes in the repository. Also, you can see from the `status` command that the file is being created with additional history because of the `+` character. This will indicate that the file was based on a preexisting repository object:

```
$ svn commit --message "moved a file"
Adding        java/HelloWorld.java
Deleting      java/hello.java
Committed revision 49.

$ svn log HelloWorld.java
------------------------------------------------------------------------
r49 | jeff | 2004-04-06 22:09:06 -0400 (Tue, 06 Apr 2004) | 1 line

moved a file
------------------------------------------------------------------------
r38 | alex | 2004-04-03 16:48:06 -0500 (Sat, 03 Apr 2004) | 1 line
```

```
added clean exit statement
------------------------------------------------------------------------
r4 | jeff | 2004-04-01 23:01:15 -0500 (Thu, 01 Apr 2004) | 1 line

Added hello.java file
------------------------------------------------------------------------
```

Even though the file was created at revision 49, the change logs were moved with it, so it appears to have been around since revision 4. As you will see in the next chapter, you can view the change logs in verbose mode, which will give you details about the operation. You also saw an example of this in section 3.5.1. This will give you the additional information to determine that the file was generated from a move or copy, and you can deduce that it will not be in the repository prior to the revision in which it was created.

Moving a file. To move a file, you will still use the `svn move` command, but instead of specifying a new filename, you will give a target directory. This target will be the new location of the file, which will keep the same name. Say you want to move the file `main.java` from `/testrepo/source/java` to the directory `/testrepo/code/java`:

```
$ cd /testrepo

$ svn move source/java/main.java code/java
A         code/java/main.c
D         source/java/main.java
```

Subversion will see that the directory `/testrepo/code/java` exists and will understand that you want to move the file into that directory. Be careful not to specify a directory that does not exist, because the command will think you are renaming the file. In the `move` command, you can use absolute or relative paths, so the command can be run from anywhere. Again, you need to note only that the working directory has been changed; a commit will need to be run before the repository is updated.

You can rename and move at the same time by combining these two methods. You will specify just the target directory of the new location and tack on the new filename.

3.6.2 Directories

So far, you have seen the `move` command operate only on files, but you can perform these same actions on a directory. The effect is the same: the directory will be renamed or moved, along with the contents.

3.6.3 Moving directly in the repository

You can move files in the repository without ever having a working copy. This is useful in scripts or for administrative tasks. The main difference is that instead of using a path to a working copy file or directory, you will use the URL of the repository. Since this is a repository transaction, the commit will immediately take place. Whenever this is the case, you will need to provide a log message. Just as with the commit, this can be done using any of the methods discussed in chapter 2. The following example will move the file `log.java` to a different directory:

```
$ svn move file:///repos/testrepo/source/java/log.java \
file:///repos/testrepo/code/java \
--message "Moving from source to code subdirectory"

Committed revision 52.
```

Since the command writes to the repository, we get a new revision number from the commit. When the URL is used like this, the command can be run from anywhere on the system. If by chance you did run the move from a working directory, the changes will not be reflected there. In order to get the changes into the working copy, you will need to run the `update` command.

If you are thinking that this command is a great way to move files between repositories, unfortunately that won't work. You can move files only in the same repository. Remember, when you move a file, the history comes along with it, so it would be impossible to get the two repositories matched up with revision numbers and log messages. Also, in the `move` command, the source and target must be the same type. You cannot move between working copies and repositories.

3.6.4 Implications of moving files on previous revisions

The reason why Subversion can easily handle moves and copies is the atomic commit concept. Since each revision is based on the repository and not on individual files, you can get the exact state of the repository for any revision. This can cause some confusion when you're looking at the history and checking out older revisions. Either your log messages need to specify that the file was moved or you need to view the logs in verbose mode (see chapter 4 for more information on this). Let's take a look at the change logs in verbose mode for the file `log.java`:

```
------------------------------------------------------------------------
r52 | jeff | 2004-04-09 09:38:29 -0400 (Fri, 09 Apr 2004) | 1 line
Changed paths:
   A /code/java/log.java (from /source/java/log.java:51)
   D /source/java/log.java
```

```
Moving from source to code subdirectory
------------------------------------------------------------------------
r28 | alex | 2004-04-03 13:51:05 -0500 (Sat, 03 Apr 2004) | 1 line
Changed paths:
   M /source/java/hello.java
   M /source/java/log.java
   M /source/java/main.java

Moved logging functionality to centralized location (log.java)
------------------------------------------------------------------------
r16 | jeff | 2004-04-03 12:21:32 -0500 (Sat, 03 Apr 2004) | 1 line
Changed paths:
   A /source/java/log.java

Added file log.java
------------------------------------------------------------------------
```

Look at revision 52, and you will see the move we just did; the log tells us what the original file was and which revision it came from. In this case, it came from revision 51, which is the transition point. Any revision of the repository before 52 will put the file in the original `source/java` directory. Starting at revision 52 and moving up, the file will be in the new location we just moved it to, `code/java`.

This process will be straightforward if you are checking out an older version of the repository to a new working copy. It will be built from the repository at that point in time. This may not be so easy if you are doing an update, however. In a simple case of moving a file, there should be no problem. For example, if you wanted to update the working copy back to revision 51, it would look like this:

```
$ svn up -r 51
A  source/java/log.java
D  code/java/log.java
Updated to revision 51.
```

This isn't a problem. The file has been deleted from the new location and added back into the original directory. Since that was the only change in the repository for revision 52, we now have the exact copy as revision 51.

3.7 Removing files from the repository

In some other version control systems, you can actually remove an object from the repository. If you could do this in Subversion, the extent of this section would be "do not remove files," and we would move on to the next chapter. Taking objects out of the repository defeats the purpose of using version control. Since Subversion takes a snapshot of the repository each time a commit happens, removing a file takes on a little different meaning. When you remove a file in

Subversion, it gets removed from that revision forward. If you check out a previous revision, the file will be restored. This has a couple of advantages. First, no matter how sure you are that you will never need the file again, you will run into situations where you want it back. It may be as simple as figuring out how something was done, but there will come a time you want to retrieve the file. Another plus of being able to retrieve files that have been removed is recovering the repository at a previous revision. You have seen this in many examples so far, and it is the same with the delete. If you need to rebuild the code at a specific point in time, you will presumably need all the files. If you could permanently remove files, this process would be impossible.

3.7.1 Running the remove command

Just as with the `move` command, you can run the `remove` command from a working copy or directly against the repository. And just as with those commands, it is safer to do this in the working copy first and then commit the changes to the repository. This will allow you to look at your changes and test if necessary, before saving them.

Removing files from a working copy. When you run the `remove` command from the working copy, the local version is removed. The modification will be propagated to the repository on the next commit:

```
$ svn remove HelloWorld.java
D        HelloWorld.java

$ svn commit --message "removed HelloWorld.java"
Deleting       java/HelloWorld.java

Committed revision 55.
```

If you do a listing in the working copy, the file will not be there, but it is still in the repository. When you run the commit, you will see the revision number where the file will no longer be present, 55. In order to restore the file, you will need to check out the repository at revision 54 or before.

Removing directly from the repository. Once again, you can run file-manipulation commands directly against the repository. This will automatically commit the change, so a log message is required from this operation:

```
$ svn remove file:///repos/testrepo/source/java/main.java \
--message "removing main.java"

Committed revision 56.
```

If you run this command from a working directory, it will not make any changes to the directory. If you do a listing, the files will still exist. You will need to run an update to have the remove reflected in the working copy.

3.7.2 Directories

Unlike removing a directory in UNIX, Subversion will allow you to do this while the directory has other files in it. The good news is that if you do this from the working copy, you will be alerted and you can get the directory back. To remove the directory and its contents, just specify the directory in the command:

```
$ cd /testrepo/source/c

$ svn remove gui
D         gui/window.c
D         gui/button.c
D         gui
```

You can see that not only is the directory marked for deletion, so are the files in it. All these files are removed from the working copy, and on the next commit they will be removed from the repository. Just as with a file, you can also specify a URL instead of a path to a working copy if you want to commit right away. You can also use an absolute path so that you are not forced to change into the local directory to run the command.

Using the force. In the previous example, we showed that when a directory is removed, all the files are also scheduled for removal. This holds true when all the files are versioned, but it is not the case when non-versioned files exist in the directory. For instance, if the `gui` directory contained an Eclipse project file, the `remove` command would not have worked:

```
$ svn remove gui
svn: Use --force to override this restriction
svn: 'gui/project.xml' is not under version control
```

Since there is a file that's not under version control in the directory you are removing, the `--force` switch is required. So why is the `force` required for removing non-versioned files? Since these files are not in the repository, if you remove them from the working copy, there is no getting them back. You can always get back a file from the repository by checking out an older version:

```
$ svn remove --force gui
D         gui/window.c
D         gui/button.c
D         gui
```

Now with the `--force` switch, the command will remove all the versioned and non-versioned files in the working copy.

3.8 Summary

One of the biggest hurdles you will face when using a version control system that implements the copy-modify-merge model is keeping things in sync. Your working copy will likely be set up for long-term development, which will require the use of non-versioned files. You cannot be constantly checking out the entire repository each time someone else saves a change. Also, as you work in larger groups, the chance of simultaneous changes increases. It is vital to understand the different ways a working copy and repository can be out of sync. The `status` command and the update will be your best friends in keeping your working copy fresh. If some of these statuses do not quite make sense yet, try to force them to happen. If you can get a file in each of the statuses we discussed, you will have mastered these concepts. Once you have this knowledge, you will be able to fly through the rest of Subversion.

In any development process, you will find yourself in a situation where file manipulation is required. Let's face it; we all move, copy, add, and delete files in a filesystem, and your code will not be any different. As it develops, there obviously will be a need to rearrange the layout and add new files and directory structures. But remember, Subversion is more than just a filesystem; it is a group of filesystem snapshots of particular points in time. You must always keep this in your head when you are running these commands. Also, these objects have more than just their contents; they have all the baggage of a version control system. You are operating on properties and the change history, not just what's inside the file.

JAVA

Subversion IN ACTION

Jeffrey Machols

A new-generation version control tool, Subversion is replacing the current open source standard, CVS. With Subversion's control components you can simplify and streamline the management of your code way beyond what's possible with CVS. For example, with just one powerful feature, Subversion's atomic commit, you can easily track and roll back a set of changes.

Subversion in Action introduces you to Subversion and the concepts of version control. Using production-quality examples it teaches you how Subversion features can be customized and combined to effectively deal with your day-to-day source control problems. You'll learn how to do practical things you cannot do with CVS, like seamlessly renaming and moving files.

The book covers branching and repository control, access control, and much more. It is written not just for release engineers, but also for developers, configuration managers, and system administrators.

What's Inside

- Integrate Subversion into your development environment
- Repository creation, backup, and options
- Svnadmin and svnlook client interfaces
- Change logs and comparing versions
- Advanced administration and configuration commands
- Lifecycle development with Subversion

A system administrator and developer with over ten years of experience, **Jeffrey Machols** is a co-founder of the Apache Directory Project and an early adopter of Subversion. He lives in Jacksonville, Florida.

"… the best book on Subversion."
—Michael Oliver
CTO, Matrix Intermedia Inc.

"… crammed with easy-to-follow examples on how to use Subversion."
—Alex Karasulu, Apache Directory Project co-founder

"… an indispensable tool."
—Mark Maimone
Rochester Institute of Technology

"Without this book, any environment using Subversion is incomplete."
—Paul Bonkowski
Senior Architect, LB Industries

"... gets you up and running quickly."
—Lübbe Onken
RA Consulting GmbH, and TortoiseSVN Developer

www.manning.com/machols

ISBN 1-932394-36-2

$39.95 US/$53.95 Canada