The guide to source control

Subversion IN ACTION

Jeffrey Machols

MANNING

Subversion in Action

Subversion in Action

JEFFREY MACHOLS

MANNING
Greenwich
(74° w. long.)

For online information and ordering of this and other Manning books, please visit www.manning.com. The publisher offers discounts on this book when ordered in quantity. For more information, please contact:

Special Sales Department
Manning Publications Co.
209 Bruce Park Avenue Fax: (203) 661-9018
Greenwich, CT 06830 email: manning@manning.com

Manning Publications Co.
209 Bruce Park Avenue
Greenwich, CT 06830

Copyeditor: Linda Recktenwald
Typesetter: Gordan Salinovic
Cover designer: Leslie Haimes

ISBN 1-932394-36-2

Printed in the United States of America
1 2 3 4 5 6 7 8 9 10 – VHG – 07 06 05 04

contents

preface

When the Apache Software Foundation accepted LDAPd, an open source LDAP server, as a new project in the incubator, I was "elected" to work with the Apache Infrastructure team to bring the source code over and get it into the CVS repository. Noel Bergman, the mentor for the project, suggested to the team that we use Subversion, a new version control system, instead of CVS. No one on the team, including myself, gave Subversion a very close look. We were all very excited to be a part of Apache and just wanted to get the project going. After all, Subversion had a different revision numbering system that nobody liked, we all had our systems set up for CVS, plus Subversion had not yet even released a 1.0 version. So we took a vote on the Apache Directory project to stay with CVS.

Despite our strong resistance to change, Noel stuck to his guns that Subversion was a better tool and it would be worth the relatively small effort required to learn it. After some discussion, Alex Karasulu, the project lead and creator of the server, asked me to investigate Subversion further and see what it had to offer.

So I went to the Subversion web site and downloaded the tool. In about two minutes I had a repository created and some test files loaded. Hmm…that was easy. Then it was time to test one of the big claims: that Subversion can rename and move files and directories seamlessly. After years of struggling to do this in CVS, I had to see it to believe it. Part of the migration into Apache required switching to a different directory structure, and I knew that would be a huge headache for me. This worked without a hitch. It took only one evening of playing around, and I was singing a different tune.

After another discussion with Alex, it was time for a second vote with our new recommendations. There was some reluctance on the part of the team, but we decided to go with Subversion. I was fortunate to work with Noel and other members of the Apache Infrastructure team, who helped me migrate the code. Through this process I was able to get some great experience with Subversion and see its benefits.

Over the next few months I became a bigger proponent of Subversion. As I talked to others in the development world, both open source and corporate, I didn't get the enthusiastic response I expected. It occurred to me that developers still saw version control as a necessary evil, not a beneficial tool for software development.

It was at this point that I wanted to write a comprehensive guide on Subversion. It was important to demonstrate to the development community not only how to run the commands but also how the tool can help the software development process. The staff at Manning also saw a need for this type of book, and so we started *Subversion in Action*.

This book is not just the online help regurgitated with a few examples thrown in. While it will serve as a technical guide to using Subversion, more importantly, it will be a guide for using it as a tool in your software development.

acknowledgments

I would like to acknowledge my family for all their support and patience through this process, another one of my projects.

There are many people along the way who helped me get to the point where I could write this book. First and foremost I need to thank Alex Karasulu. He opened many doors for me in the open source and technical communities (and sometimes pushed me through the doorway). I would also like to thank Noel Bergman for introducing me to Subversion and helping me through the learning process.

None of this would have happened without the people who created, developed, and now support Subversion. The time, effort, and sacrifices it takes to create a project of this size are enormous—and this is all volunteer work.

Thanks to all the people who either formally or informally reviewed the book as it was being written and gave input. These include Darin Riedlinger, Michael A. Koziarski, Doug Warren, Michael Oliver, Bill Fly, and James Rybacki. I would especially like to thank Ryan Cox for his multiple passes through the book and great suggestions.

All of the great people at Manning Publications really helped me get through this daunting task. I owe a great deal to the following folks:

- Marjan Bace, publisher of Manning, for refining my rough idea, training me without limiting me, and finally taking a chance on a rookie
- Susan Capparelle for her help in coordinating the entire process and assiting me throughout
- Ann Navarro, the book's development editor, for all her guidance and support
- Karen Tegtmeyer, who coordinated all the reviewers and the feedback
- Mary Piergies, who managed the production of the book and kept things, including me, moving
- Linda Recktenwald for her detailed editing of the book
- Dottie Marsico and Gordan Salinovic for turning the text into a book
- Helen Trimes for getting the book out there and known
- Susan Forsyth for proofreading the book and getting me through the final stages

about this book

How the book is organized

Each chapter in the book is meant to extend your knowledge of Subversion. The first few chapters will give you a basic understanding so you can start using the software productively. The following chapters will show you how to take advantage of the power of Subversion. This means you don't have read the book from cover to cover to use Subversion. In addition to using the book as a guide, you can use it as a reference. By following the headings, you can quickly find the information you are looking for. Let's take a look at how the chapters are organized:

- Chapter 1 gives you an introduction to version control in general and shows what Subversion brings to the table.
- Chapter 2 gets you started using Subversion. Here you will install the software, create a repository, and start to access it.
- Chapters 3 and 4 go through the common Subversion commands to manipulate files and get information out of the repository.
- Chapter 5 discusses branching and tagging concepts and how Subversion implements them.
- Chapter 6 describes properties, which are Subversion's implementation of metadata.
- Chapters 7 and 8 go into repository administration, including backups and network access, and then advance to such topics as client configuration and hook scripts.

- Chapter 9 explores the "extra" Subversion clients that give you more advanced capabilities to customize your implementation.
- Chapter 10 reviews some of the common third-party tools that can enhance your development with Subversion.
- Chapter 11 wraps up by showing how you can use Subversion in areas of development that may not fit the traditional view of version control yet still be valuable.

Who should read this book

Version control is something that affects many parts of an organization, open source or commercial, that does any kind of development. There are three main groups of people who need to know about the tool and the process behind the tool:

- *Developers:* This is the obvious group who will benefit from the book. Not only will you learn how to use Subversion, but you will see ways it can help you with your software development.
- *Configuration managers:* If you are responsible for the building, deployment, and migration of source code, you will need to have a better understanding of version control than even the developers. In addition to using Subversion, you will need to create scripts, automation, and the processes around the code.
- *System administrators:* In many organizations, the people maintaining the infrastructure are separate from those who develop it. Even if you do not work with the code, you will have to install the software, set up access, configure the network, and create backups. These topics have been segmented so you can easily refer to them without having to go through all aspects of Subversion.

Conventions

The vast majority of Subversion is accessible from the command line. A command prompt will be indicated by the `$` character. This will indicate the start of a command, and you will not type the `$` when running the command. For example, if you see something like the following:

```
$ run a command
Some output
```

you will type everything on the first line, including the spaces, except the first `$`. The lines without the `$` prompt are output from the command. All the commands and the output will be in code font, which tells you that the text is part of the command or part of the output. For the sake of readability, some commands will span

multiple lines in the example. To indicate this, a \ character will be placed at the end of line:

```
$ run a command \
on two lines
Some output
```

In this example, you will type "run a command on two lines" as the actual command. The output will follow the last line that does not end with \.

Source code

While Subversion and version control in general are designed for software development, this is not a programming book. With that said, we will be working with code in many of the examples. The source code is oversimplified on purpose so you will not get caught up in figuring out what the program is doing (we will work with HelloWorld quite a bit). I intentionally switch from Java to C—and sometimes text files—throughout the book to illustrate that Subversion does not care what it is storing.

Author Online

Purchase of *Subversion in Action* includes free access to a private web forum run by Manning Publications where you can make comments about the book, ask technical questions, and receive help from the author and from other users. To access the forum and subscribe to it, point your web browser to www.manning.com/machols. This page provides information on how to get on the forum once you are registered, what kind of help is available, and the rules of conduct on the forum.

Manning's commitment to our readers is to provide a venue where a meaningful dialog between individual readers and between readers and the author can take place. It is not a commitment to any specific amount of participation on the part of the author, whose contribution to the AO remains voluntary (and unpaid). We suggest you try asking the authors some challenging questions lest his interest stray!

The Author Online forum and the archives of previous discussions will be accessible from the publisher's web site as long as the book is in print.

about the title

By combining introductions, overviews, and how-to examples, the *In Action* books are designed to help learning *and* remembering. According to research in cognitive science, the things people remember are things they discover during self-motivated exploration.

Although no one at Manning is a cognitive scientist, we are convinced that for learning to become permanent it must pass through stages of exploration, play, and, interestingly, re-telling of what is being learned. People understand and remember new things, which is to say they master them, only after actively exploring them. Humans learn in action. An essential part of an *In Action* guide is that it is example-driven. It encourages the reader to try things out, to play with new code, and explore new ideas.

There is another, more mundane, reason for the title of this book: our readers are busy. They use books to do a job or solve a problem. They need books that allow them to jump in and jump out easily and learn just what they want just when they want it. They need books that aid them *in action*. The books in this series are designed for such readers.

about the cover illustration

The figure on the cover of *Subversion in Action* is a "Hamal," or common porter. The illustration is taken from a collection of costumes of the Ottoman Empire published on January 1, 1802, by William Miller of Old Bond Street, London. The title page is missing from the collection and we have been unable to track it down to date. The book's table of contents identifies the figures in both English and French, and each illustration bears the names of two artists who worked on it, both of whom would no doubt be surprised to find their art gracing the front cover of a computer programming book...two hundred years later.

The collection was purchased by a Manning editor at an antiquarian flea market in the "Garage" on West 26th Street in Manhattan. The seller was an American based in Ankara, Turkey, and the transaction took place just as he was packing up his stand for the day. The Manning editor did not have on his person the substantial amount of cash that was required for the purchase and a credit card and check were both politely turned down.

With the seller flying back to Ankara that evening the situation was getting hopeless. What was the solution? It turned out to be nothing more than an old-fashioned verbal agreement sealed with a handshake. The seller simply proposed that the money be transferred to him by wire and the editor walked out with the bank information on a piece of paper and the portfolio of images under his arm. Needless to say, we transferred the funds the next day, and we remain grateful and

impressed by this unknown person's trust in one of us. It recalls something that might have happened a long time ago.

The pictures from the Ottoman collection, like the other illustrations that appear on our covers, bring to life the richness and variety of dress customs of two centuries ago. They recall the sense of isolation and distance of that period—and of every other historic period except our own hyperkinetic present.

Dress codes have changed since then and the diversity by region, so rich at the time, has faded away. It is now often hard to tell the inhabitant of one continent from another. Perhaps, trying to view it optimistically, we have traded a cultural and visual diversity for a more varied personal life. Or a more varied and interesting intellectual and technical life.

We at Manning celebrate the inventiveness, the initiative, and, yes, the fun of the computer business with book covers based on the rich diversity of regional life of two centuries ago, brought back to life by the pictures from this collection.

1 Introduction

In this chapter

- Introduction to version control
- Overview of Subversion
- New revision numbering paradigm

I doubt this was planned, but the dictionary definition of "subversion" happens to provide a fitting description of Subversion's social role: by *subverting* the commanding position held for years by Concurrent Versions System (CVS), it is quickly becoming the de facto standard for open source version control systems. Many open source development communities are moving away from CVS and adapting Subversion, and this trend will only accelerate. If you want to be in on this trend—and take advantage of the best versioning system available for your group-development infrastructure—this book is for you.

While other areas of software development such as IDE's progressed, version control remained relatively stagnant, especially in the open source world. But as the practice of community software development caught on, it became apparent that the glue that allows people to work together in different places and at different times is their version control system. It also became clear that the old standard, CVS, was becoming inadequate for supporting advanced open source development practices. From this need, Subversion was born as a product built by developers to make version control a seamless part of the development process instead of an impediment.

If you find yourself involved in a live discussion that touches on Subversion, you might like to know that those in the know do *not* pronounce it as a single word, with the accent on "ver." Instead, they say it as if it were two words, *Sub version*. Pronounce it as they do and your standing automatically improves.

Now, having finished with that important matter, let's turn to substance. Throughout this book, we will explore the features of Subversion that are making it the success it is and causing development communities and businesses to embrace it so quickly. Not only will we look at how the system works and what the commands do, but we will also tie it all in with your software development process.

1.1 Understanding version control

First, it will help to understand the basics of a version control system in a generic sense. Think back to elementary school and remember the bespectacled figure of your librarian. She may have been stern but she knew where every book in the library was, who had it checked out, and when a new edition was coming in. A version control system plays a similar role, but without the thick glasses and the nasty shushing, of course. Rather than tracking books in a library, a versioning system manages files in a directory. It controls the files coming in and out and keeps track of who made the change. It also provides the useful capability to get back an old revision of a file.

1.1.1 Why use version control

Have you ever been in a situation where something in a file changed and you asked, "What did that file look like before—when it worked? When did this file change? Who changed it?" I often find myself looking at an edit I made in the past and wondering why I made it. A version control system gives you the answer to these questions. It also gives you the tools to manage your code better by providing features such as the following:

- Helps you manage code by tagging specific versions as a release of the software
- Allows you to add automation to manual tasks through the use of hook scripts, which run when triggered from an action
- Allows for multiple users to work on the same file without losing any changes
- Provides the ability to start another development path from the same code base and then merge the two paths back together
- Efficiently stores multiple versions of a file by keeping only the changes to each version instead of an entire copy

Let's take a look at how some of these features are implemented by exploring some of the technical aspects of a version control system.

1.1.2 The repository

The core of any version control system is the *repository*, which is a basically a souped-up filesystem. It tracks the changes to files and allows multiple users concurrent access. Each time a modified file is saved to the repository, a new *revision* of that file is created. These revisions contain not only the contents in the file that changed but also external information, such as who changed it and when it was changed. In a standard filesystem, you access and edit files directly; not so in a repository. Instead, the system has a client interface that does the reading and writing to the repository for you. This client talks to the repository and allows for the transparent implementation of all the features we just described.

1.1.3 Checkout strategies

We know repositories do not allow direct access to the files and directories, so you need a way to get at your code. To accomplish this, you perform a *checkout*, which is a request for a copy of the file through the client. You can check out one file, a directory, or the entire repository. This is similar to checking out a book from a library. You go to a central location and get a copy of a book, or in the case of a

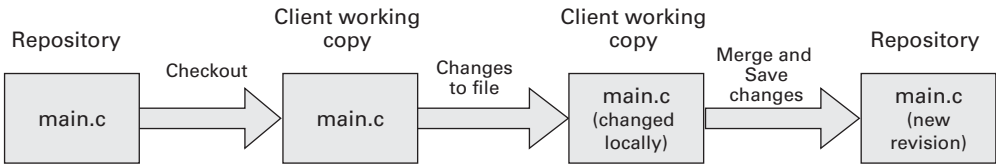

Figure 1.1 Workflow in the copy-modify-merge model

repository, a file. If a new edition or revision comes out, you go back and check out the new one.

Copy-modify-merge. Many version control systems, including Subversion, use a checkout strategy called *copy-modify-merge*. In this model, when you perform a checkout, you get a working copy of the file in a local directory on your machine. This is a snapshot of the repository where you will make your changes independently of other developers. When you have finished with your changes, you copy the file, `main.c` in this example, back into the repository. Figure 1.1 illustrates this workflow.

Then, when another developer checks out the file `main.c`, it will include your changes, and that person will follow the same process. If you are making changes at the same time as someone else, the flow will look something like figure 1.2.

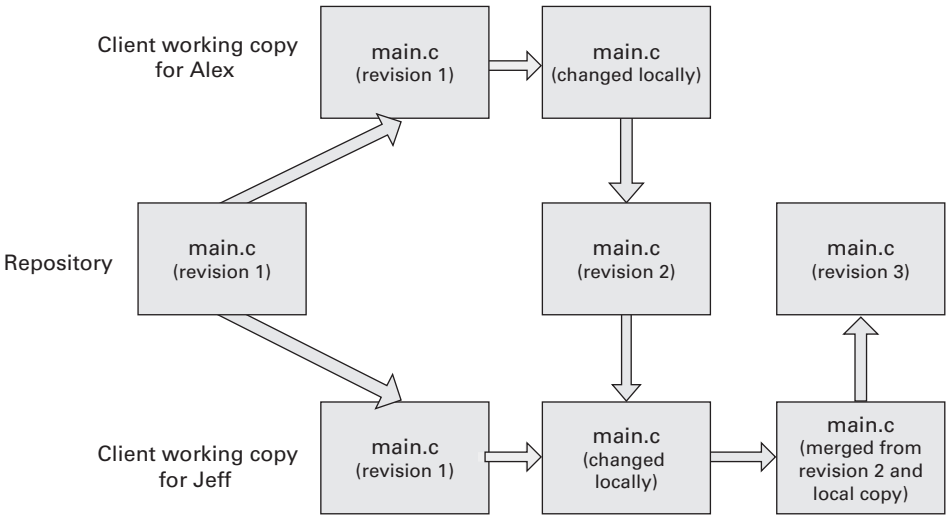

Figure 1.2 Workflow in the copy-modify-merge model with multiple users

This merge will happen automatically if the two changes do not conflict, which would happen if the users made changes to the same line in the file. It is important to understand that these conflicts are strictly based on whether or not the system can merge the data within the file. There are no logical checks that take place to ensure that you did not muck up the source code, so you are responsible for the logical validation of the change.

1.1.4 Commit and update

Version control systems that use the copy-modify-merge model create a working copy of the repository for you to make your changes. You also need a way to save your edits back to the repository. The process of applying your changes into the repository is called *committing* (some systems call this *check in*). When you perform a commit from your working copy, the system will find the files that have changed and compare them against the latest version in the repository. If there are no conflicts, a new revision of the files will be created and saved in the repository. Once you commit, the repository will be in sync with the changes in your working copy, but this is only half the battle.

If you develop in a vacuum, the checkout and commit give you everything needed to keep the repository and working copy in sync, but this does not happen in the real world. While you were editing your working copy, Alex committed his changes to the repository. So not only do you need a way to get your changes into the repository, you need a way to get any changes made to the repository back into your working copy; this is done with an update. An *update* is a specialized checkout that gives you only the changes in the repository since your checkout or last update.

You know that a commit saves your changes back to the repository, but remember that there is more to a version control system than just file contents. When the commit occurs, a record of the change is also saved. This record is called a *change log* and is one of the components that gives the answer to those who, when, and why questions we have been trying to answer.

1.1.5 What are change logs?

One of the reasons for using a version control system is to get more information about a change than just the content difference in a file between revisions. This additional information is stored in the change log and is attached to each revision of a file. Let's take a look at a typical change for the file `main.c` in figure 1.3.

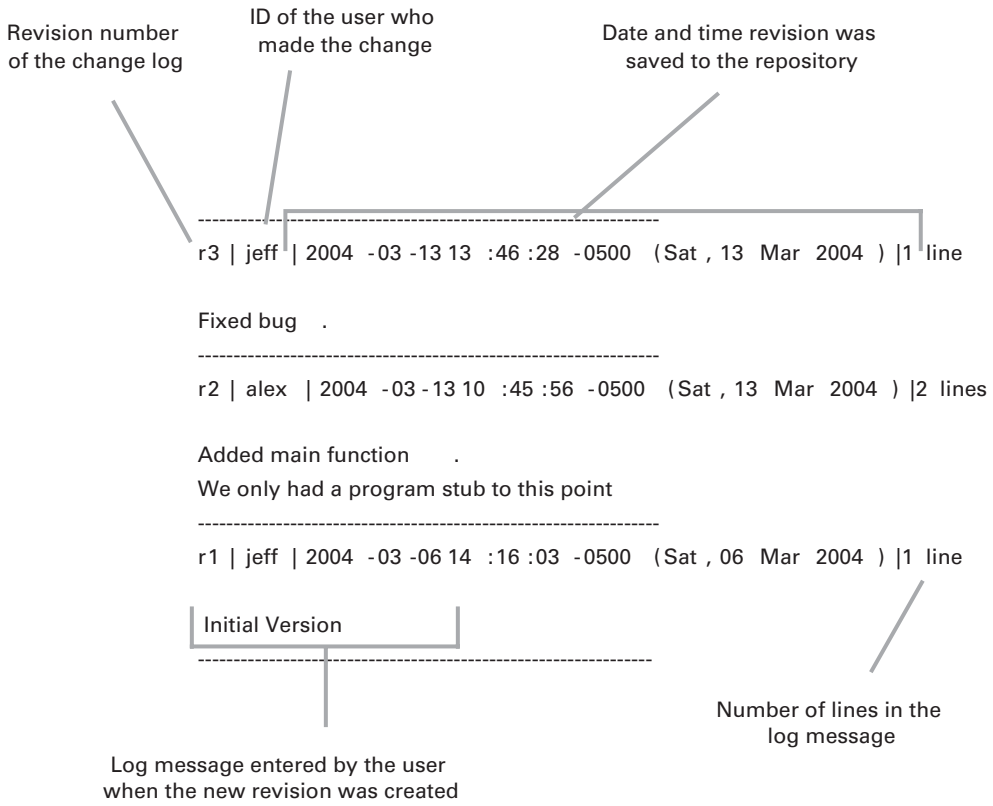

Figure 1.3 Change log information provided by subversion

Each revision's change log for the file `main.c` has the same set of information and is ordered by date in descending order. The top line of each change log is the system information. The line(s) following the system information is the log message. This is a description of the change for that revision, which the user adds as part of the commit. Since the log message is added by the user, the quality of the message will depend on that developer's attention to detail.

When developing software for the long term, it is important to be descriptive when writing the log message. It should be clear to all users why you are making the change and what the problem and fix are. To see how nondescript messages can be a problem, look at the following log message:

```
------------------------------------------------------------------------
r3 | jeff | 2004-03-13 13:46:28 -0500 (Sat, 13 Mar 2004) | 1 line

Bug Fix
------------------------------------------------------------------------
```

The log message simply says "bug fix." While this may make sense to the developer at the time of the change, it will probably not make sense in the future and likely won't help other developers who look at it. The only way to get any details about the change is to look at the content differences between revisions. This can be tedious and will not always give you the answers you are looking for. Even if you see the differences in the contents of the file, you still may not know why the change was made. Instead of just saying "bug fix," consider using something like the following message:

```
------------------------------------------------------------------------
r3 | jeff | 2004-03-13 13:46:28 -0500 (Sat, 13 Mar 2004) | 3 lines

Bug Fix number 2255.  The application is supposed to print a message
when it starts.  This message did not print because it was directed
to STDERR.  Adjusted the startup println statement to STDOUT
------------------------------------------------------------------------
```

This log message is clear. Now you can easily see what was fixed, which can drastically cut down your research time. If your development process uses a bug-tracking tool, you can include the tracker number in the log message for a quick and dirty integration. Change logs provide the basic information in a version control system, but you are not limited to this data. You can add your own customization and information to a version by using properties.

1.1.6 The role of properties

Today's version control systems are much more than simple repositories for storing old versions of source code; they help you manage your development process. For example, most software you develop will have releases. You will want a way to associate a particular version of a file, or more likely multiple files, with a particular release of the entire application. So, for example, you can say that release 2.0 of the application uses revision 10 of the file `main.c`. In order to accommodate this, version control systems have something called *properties*, also known as *metadata*. This is simply some user-defined piece of information that is associated with a file or revision of a file. If we stick with the analogy of a library, using properties

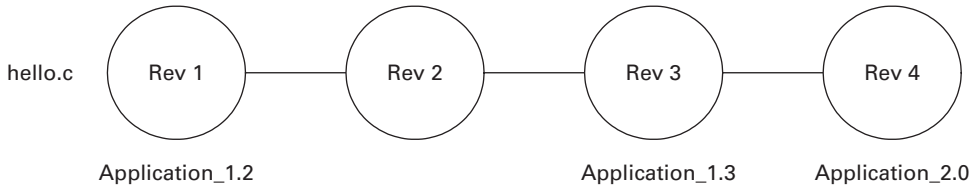

Figure 1.4 Tags in the version tree

to track releases of your software is similar to using a card catalog system. It allows you to easily track and find your files and see what state they are in.
So using properties, you can assign the name of the application release to a revision of the file to track the progress through the releases of your software. Figure 1.4 illustrates how this would look in a version tree for a file called `hello.c`.

The file `hello.c` has four revisions in the repository; three are associated with a particular release of your software. If version 2.0 of the application needs to be built, revision 4 of `hello.c` will be used. Versions not associated with a tag are either just checkpoints for a developer to save his work or not production worthy. You can use properties for any kind of custom information that is required in your development process.

1.2 30,000 foot view of Subversion

Now that you have an understanding of what most version control systems provide, we can better examine what is different about Subversion. Right off the bat, we know that Subversion uses current open source tools to fill needs that do not directly deal with version control. The following is a list of the key open source tools used:

- Berkeley DB is used for a back end, which provides an efficient storage facility in terms of speed and memory.
- The Apache Portable Runtime libraries are used, eliminating the need for operating system–specific code.
- The Apache HTTP server is used for the network protocol, which gives built-in security and web-browsing capabilities.

In addition to utilizing current technologies, Subversion has a few innovations that give it additional advantages. Let's look at some of them.

1.2.1 Atomic commits

"Atomic" means one global change to the repository, no matter how many files have changed. For example, say you make changes to five different files, instead of five new files being created in traditional systems, Subversion creates one new repository with all the changes. There is one revision number and one change log in an atomic commit, as opposed to one for each file. Don't worry if this concept is not crystal clear yet; as you move through the book, it will make sense and the reasons why this was implemented will become obvious.

1.2.2 Everything is versioned

If you have been in software development for any length of time, you have probably been in a situation similar to this. You come in to work on a Monday morning to find your mailbox full of user complaints that a function in your software doesn't work anymore, but you haven't made any changes to the source code in two weeks. After some probing, you find out that this function gets run only annually, so the last time it worked was a year ago. Of course, there have been a dozen releases and patches to the software since last year. In order to see if a code change broke the function, you will need to travel back in time to re-create the repository at that point, build the application, and see if the problem exists. This ability to "go back in time" turns out to be a very powerful feature.

Any version control system can get the file contents back easily enough, but there may be more to it than that. For instance, what if some files were moved to different directories? Then you would need to fix the paths in your build scripts. If you are using tags or labels on the files to determine what was in production, there is no way track this unless you manually record it somehow. Subversion not only tracks changes to a file's contents, it also versions directories and properties, which allows you to return the repository back to its original state, which consists of more than just file contents.

1.2.3 The three components of Subversion

Subversion consists of three components: the filesystem, the network, and the client. The filesystem component is the Subversion repository that stores the files and change logs. The client component is the set of libraries and command-line programs the user will run to access the repository. The network component can be used if the access to the repository is not local—it acts as an intermediary between the filesystem and the client. For local access, the client talks directly to the filesystem. Figure 1.5 shows the coupling points of the components.

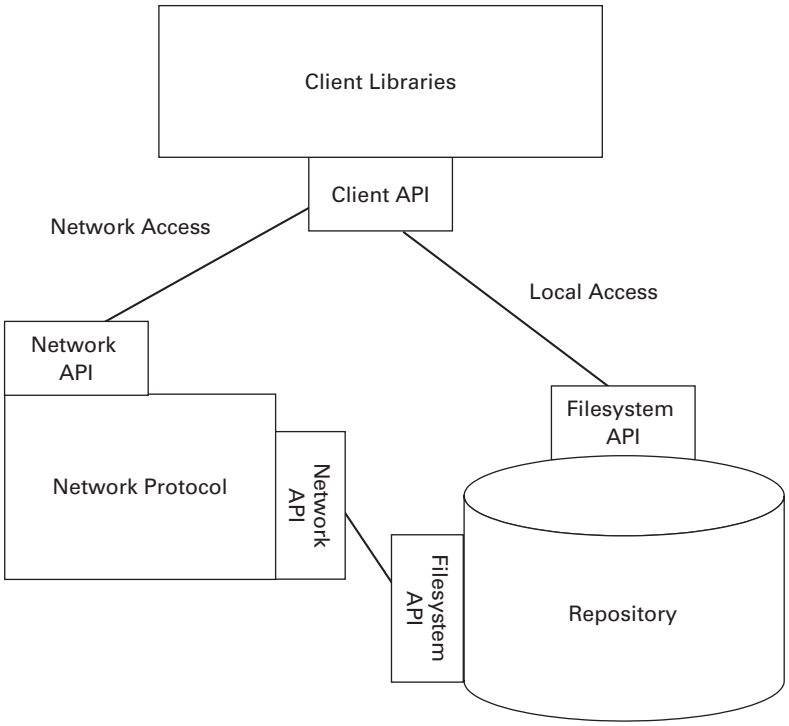

Figure 1.5 Subversion component coupling points

Each component is a standalone module and can be changed out without affecting the other two. So how does this modularity help you? First, it allows you to access the repository over the network or locally on the host. In addition, this design provides more third-party and vertical tools to sit on top of Subversion, and you can easily snap in different network protocols or client interfaces.

1.3 Revision numbers

Subversion has developed a new paradigm for identifying version numbers in a repository, one that is based on repository commits and not individual file changes. This is a little different than traditional version control systems, so it may take some getting used to.

1.3.1 The old way

In traditional version control systems, each file has its own revision number, which is an incremental decimal. So the revision numbers for a file would be 1.1, 1.2, 1.3, etc. There will not be any gaps because each time the file is changed, it is incremented off its own revision number (unless it is manually changed). To see how this can be a deficiency, consider two files: `main.c` is at revision 1.2 while `hello.c` is at revision 1.1. If a change is made to both files and committed to the repository, the output of the two logs would look something like this:

```
File: main.c
head: 1.3
----------------------------
revision 1.3
date: 2004/03/09 18:23:14;  author: alex;  state: Exp;  lines: +3 -0
Added author tags to source files that were missing it
----------------------------
revision 1.2
date: 2004/03/09 18:21:19;  author: jeff;  state: Exp;  lines: +1 -0
Added print statement so we know it started
----------------------------
revision 1.1
date: 2004/03/09 18:16:45;  author: jeff;  state: Exp;
branches:  1.1.1;
Initial revision
----------------------------



File: hello.c
head: 1.2
----------------------------
revision 1.2
date: 2004/03/09 18:23:14;  author: alex;  state: Exp;  lines: +4 -0
Added author tags to source files that were missing it
----------------------------
revision 1.1
date: 2004/03/09 18:16:45;  author: jeff;  state: Exp;
branches:  1.1.1;
Initial revision
----------------------------
```

Since the revision numbers are just sequential, `main.c` moves to 1.3 and `hello.c` is now at 1.2. There is nothing that ties the two revision numbers back to your change. The only way to tell which versions of files trace back to the same change is to look at the log file. This is fine if you are comparing files that you are aware of. What if a change caused your application to go bonkers, and you need to trace

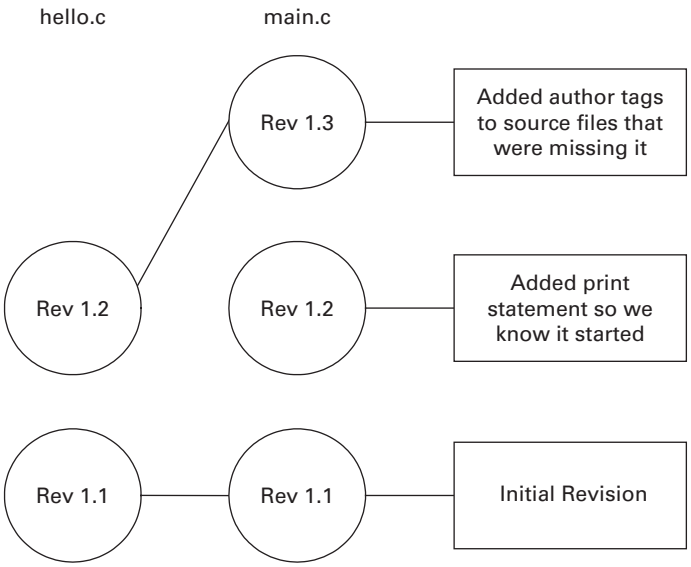

Figure 1.6 Version tree with traditional version numbering

back to see what has been modified? To see how difficult this can be to trace, look at how the changes are correlated in the version tree in figure 1.6.

As you can see, it is difficult enough to trace changes with only two files. The same commit produced revision 1.2 of the file `hello.c` and revision 1.3 of `main.c`. In order to find all the files affected by a change, you will have to look through all your files for either a time stamp or specific log message. To get around this problem with most traditional version control systems, you will need to create a label and attach it to each new file revision. Unless you create a system to tag all previous revisions of a change, it will be difficult to back out of a change.

1.3.2 Subversion revision numbers

With the atomic commits, Subversion tracks changes using a per-commit basis, not per-file. Instead of each file maintaining a separate revision number, Subversion has one global revision number for the repository. The number starts at 0 when the repository is created and is incremented by a whole number each time a commit happens. Conceptually, each commit creates a new copy of the repository with the changes applied. Files that do not change on a commit simply have identical copies created in the new repository, although they are stored more efficiently than a complete copy. Let's

take a look at what the log files would look like in Subversion going through the steps in the previous example:

```
File: main.c
------------------------------------------------------------------------
r3 | alex | 2004-03-09 22:32:34 -0500 (Tue, 09 Mar 2004) | 1 line
Added author tags to source files that were missing it
------------------------------------------------------------------------
r2 | jeff | 2004-03-09 22:30:42 -0500 (Tue, 09 Mar 2004) | 1 line
Added print statement so we know it started
------------------------------------------------------------------------
r1 | jeff | 2004-03-09 22:28:18 -0500 (Tue, 09 Mar 2004) | 1 line
Imported Sources
------------------------------------------------------------------------


File: hello.c
------------------------------------------------------------------------
r3 | alex | 2004-03-09 22:32:34 -0500 (Tue, 09 Mar 2004) | 1 line
Added author tags to source files that were missing it
------------------------------------------------------------------------
r1 | jeff | 2004-03-09 22:28:18 -0500 (Tue, 09 Mar 2004) | 1 line
Imported Sources
------------------------------------------------------------------------
```

What should stand out when you compare the two log histories is that the file `hello.c` jumps from revision 1 to revision 3. Many people (including myself) cringe the first time they see this. Once you stop and think about it, this method makes more sense. When you commit to the repository, you are applying a set of common changes. If the revision number for each file is the same, it is easier to trace and back out if necessary.

This also makes it easier to check out the repository at a specific point. The reason for the ease is that all changes are at the same level in the version tree, as figure 1.7 shows.

Unlike the traditional numbering scheme we saw earlier, in Subversion all the files that were changed by the commit to add the author tags are part of revision 3.

When you check out a specific revision of the repository, you will get the highest revision of each file less than or equal to the number specified. So using the files `hello.c` and `main.c` from our previous example, you check out revision 2 of the repository. Since `main.c` has a revision 2, you will get that copy. Because `hello.c` does not have a revision 2, it will go down to revision 1. Since the commit is considered to be at a repository basis, the change log information is stored at the repository level instead of the file level. This means that instead of each file

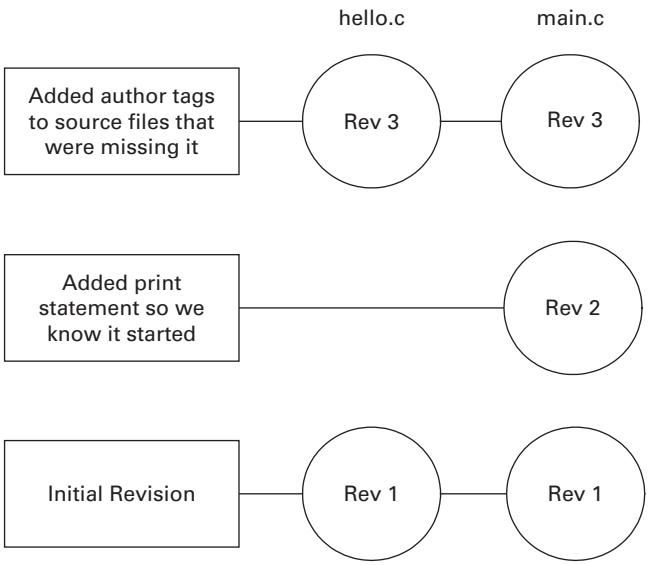

Figure 1.7
Version tree with Subversion revision numbers

revision containing the author, date, and change log, this information is kept only once in the repository. This actually allows Subversion to be more efficient in terms of storage. It also allows for easier maintenance; if you need to adjust the change log, you need to do so in only one place instead of every file.

1.4 Summary

We have discussed concepts such as atomic commits and the copy-modify-merge model at a high level, but these may not be perfectly clear to you yet. As you read through this book and explore the commands, these concepts will become second nature. Just remember that it will be up to you to keep your working copy in sync with the repository, with both your changes and those of other developers.

You are now ready to dive into Subversion! Whether you are a novice to traditional version control systems or a seasoned veteran, the best way to learn how to use this tool is to walk through the steps as we discuss them. Don't be afraid to experiment and try different things. If you create a test repository, the worst case would be that you blow it away and have to start over. If you are brand new to version control systems and are feeling a little wary, don't worry. You can easily set up and destroy test repositories to get the feel for them. We will go through Subversion sequentially, starting with creating a repository.

2 Getting started

In this chapter

- Get and install Subversion
- Get around the command-line interface
- Create a repository
- Basic work cycle—getting a file in and out of Subversion

Now that you have an understanding of the high-level features of a version control system, we can start examining the details of Subversion. By the end of this chapter, you will be making changes to your code and saving them to the repository. While the next two chapters will ease you into working with Subversion, there will still be a bit of theory thrown in. This is not intended to make you an expert in the theory behind version control, but it is important that you understand and become efficient with these concepts. The goal is to make using Subversion second nature so it will improve your software development, not hinder it.

2.1 Getting Subversion

Before you can jump into using Subversion, you will have to go out and get it. As its popularity grows, it is being bundled with other software packages such as Fedora Linux. Despite this trend, you will be better off going to the Subversion web site and downloading it. This is a fast-moving application, so a bundled distribution such as Fedora will likely be outdated by the time you install it. The software can be downloaded from http://subversion.tigris.com/project_packages.html.

2.1.1 Which package to get?

Subversion has binary distributions for multiple operating systems, including Windows and most flavors of Linux. When given the option, it is usually quicker and easier to use these, as opposed to building the source code. The install and dependencies are different for each distribution, but they use the standard install system for the particular OS. For example, the Red Hat distribution is an RPM (Red Hat Package Manager) file, FreeBSD uses pkgsrc, and Windows is just an `.exe` file. If your operating system does not have a binary package, or you need a bleeding-edge release, you will have to compile Subversion. While this may take a little more effort than running an `.exe`, it is not impossible.

You will also need to choose which release of Subversion is right for you. This is a decision that will depend on how in-depth you are going to use the tool and which features and bug fixes come out in new releases. A general rule of thumb is that if you aren't sure which version to get, you should use the latest stable release. The Subversion web site contains change logs for the releases that will tell you what has been fixed or added. You can look through these logs to see if a particular version is right for your environment or if it is time to upgrade. It is difficult to describe in great detail how to navigate through the site because it is constantly evolving. Once Subversion reaches a maturity level, the web site and distributions will likely follow.

Don't get hung on up on distributions. With all of the different operating systems and configurations, it is impossible to cover all or even most of the possibilities. If you are trying to learn Subversion, don't let getting the correct distribution or compiling become a roadblock. Most people have access to a Windows or Red Hat workstation. This is all you need to start using Subversion, and you won't get frustrated and give up because you can't get it installed.

2.1.2 Installing on Windows

The Windows install of Subversion is more straightforward than that of the other operating systems. There are no dependencies, and you need to download only one `.exe` file. Go to the Downloads link on the web site and select the Win32 distribution. After you figure out which release fits your needs, find the appropriate file in the list. The name of the file will be `svn-x-x-x-setup.exe`, where `x-x-x` is the release number. You will not need any of the files with other identifiers such as `dev` or `pdb` in their name. These are for development only.

Once you have identified the file, you can download and execute it. When you run the program, it will launch a basic Windows installer program. You will get the standard prompts such as accepting the license agreement and where to install the application. After you step through all the setup questions, the binaries will be placed in the target directory. Even though you are provided with a graphical installer, these are still command-line programs, so they must be run at a Windows command prompt or another shell such as cygwin. Some graphical tools are being developed to get around the command line; we'll discuss these later.

2.1.3 Installing on Linux

We will use Red Hat or Fedora as the Linux distribution to examine using the RPM binaries, but the process is the same for any version. In any of the Linux distributions, there are some dependencies you need to worry about, but this is still a relatively painless task. On the Downloads page, select Red Hat Linux, and you will see a list of the different versions of the OS. Select the appropriate one, and a page with a number of RPMs will open. For a normal install you will need the following packages, which need to be installed in the order listed:

- *apr-x-x-x...rpm*—Apache portable runtime libraries
- *neon-x-x-x..rpm*—DAV protocol libraries
- *subversion-x-x-x.rpm*—Subversion client
- *subversion-server-x-x-x.rpm*—Binaries to run the Subversion server

Each of the packages will have a version number appended after the filename. You must also be careful not to get the "dev" RPMs. These will have the same filename except they contain the word *dev* in between the base name and the revision number. Also, you will need only the server package if the repository needs to be accessed over the network. You can just use the default `rpm install` command. If your operating system is missing any dependencies, the install will notify you. You will find some of these package in the same location where you downloaded Subversion, so that is a good place to start looking should the need arise.

2.1.4 Subversion is installed; what now?

Unless you are accessing Subversion over the network protocol (we will discuss this setup in chapter 7), there is nothing else you need to do. There is no service or daemon that needs to run; all the access is done through clients accessing the repository layer. To see if everything is working correctly, you can run the `svn` client with the `--version` option:

```
$ svn --version
svn, version 1.1.0 (Release Candidate 1)
   compiled Aug  6 2004, 21:01:15
Copyright (C) 2000-2004 CollabNet.
Subversion is open source software, see http://subversion.tigris.org/
This product includes software developed by CollabNet (http://www.Collab.Net/
   ).

The following repository access (RA) modules are available:

* ra_dav : Module for accessing a repository via WebDAV (DeltaV) protocol.
  - handles 'http' schema
  - handles 'https' schema
* ra_local : Module for accessing a repository on local disk.
  - handles 'file' schema
* ra_svn : Module for accessing a repository using the svn network protocol.
  - handles 'svn' schema
```

This will tell you which version of Subversion you have, plus which modules are installed. You should see the three modules listed above, unless you skipped the Apache server package. If one of the modules is missing, you cannot access your repository using that method. For example, if `ra_dav` is not listed, you will not be able to set up the repository using the Apache HTTP protocol.

2.2 Command-line interface

Subversion comes with a set of command-line interfaces called clients, which are used to perform all of the tasks. Unless you have a third-party tool to provide the interface to Subversion commands (we will talk about these in chapter 10), these clients will be the way you interact with the version control system. As a result, it is important to understand how these work and how to get around them, in addition to the concepts behind them. For example, consider an application such as Word. You may know that you want to underline a section of text, but if you do not know how to get around the GUI, you will have trouble using the tool. The same holds true for Subversion; you know that you need to commit to the repository, but if you don't understand the syntax, this simple process will be more complicated. This section will guide you through the clients in Subversion, so that you can use the tool more proficiently.

There are three Subversion command-line interfaces, each with a different purpose; we'll describe these in the next section. They are simply executable commands; you pass in the operations (subcommand) and the specific options for that command. This is the same for Windows and UNIX-based operating systems. All the executables will reside in the `$SUBVERSION/bin` directory; if you set up your PATH environment variable as described in the previous section, you will be able to run these commands without specifying the entire path. While the clients will allow you interchange the order of the subcommands and options, for clarity you should use the following syntax: `client_name <subcommand> [options] [arguments]`.

2.2.1 Different clients

Subversion provides multiple interfaces for different uses. This allows for a clear separation of tasks between users and administrators.

User client. The `svn` client makes up the bulk of Subversion; this is what the users will run to perform the actual version control commands. This interface was made to feel like CVS where possible to provide an easy transition for those who are new to Subversion.

Administrator client. Subversion provides a separate client called `svnadmin` for performing administrative and maintenance functions on the repository. Having this client provides a clear separation of duties in Subversion, keeping the user client cleaner and easier to use, in addition to facilitating better security. Since the administrator client works on the repository, it must be run on the local repository machine, not over a network. As a result, all arguments referencing a repository are in the PATH format, not URL. See section 2.2.3 for more information on PATH and URL.

Browser client. The `svnlook` client is a tool that will browse a repository without making any changes. It has many of the information-gathering commands of the user client such as `diff`, `cat`, and `history`. We will see how these commands work in the user client over the next few chapters. When we discuss repository administration, these commands will be examined from the point of view of the `svnlook` client. This client is good for use in trigger or hook scripts to collect information on the repository. Just like the `svnadmin` client, this client is run on the repository host, so it accesses the repository via PATH and not URL.

2.2.2 Getting help

While all the clients provide different operations, they all parse the command-line input the same way. The first thing to note is the help for each of the clients. The help provided by the clients is a good reference, but that is usually the extent of the information provided. Running a client with help as an argument will list all available subcommands for that client. To get additional help on a specific subcommand, simply pass that subcommand as an argument. For example, let's look at the help for the `checkout` command:

```
$ svn help checkout
checkout (co): Check out a working copy from a repository.
usage: checkout URL... [PATH]

Note: If PATH is omitted, the basename of the URL will be used as
the destination. If multiple URLs are given each will be checked
out into a sub-directory of PATH, with the name of the sub-directory
being the basename of the URL.

Valid options:
-r [--revision] arg :ARG (some commands also take ARG1:ARG2 range)
A revision argument can be one of:
NUMBER       revision number
"{" DATE "}" revision at start of the date
"HEAD"       latest in repository
"BASE"       base rev of item's working copy
"COMMITTED"  last commit at or before BASE
"PREV"       revision just before COMMITTED
-q [--quiet]             : print as little as possible
-N [--non-recursive]     : operate on single directory only
--username arg           : specify a username ARG
--password arg           : specify a password ARG
--no-auth-cache          : do not cache authentication tokens
--non-interactive        : do no interactive prompting
--config-dir arg         : read user configuration files from directory ARG
```

There are a few things to understand about the syntax that will help you use Subversion effectively. First, notice the description line, specifically the text in the parentheses. Many of Subversion's commands will have multiple names; in the case of the `checkout` command, it has one alias, `co`. While this may seem confusing at first, many users prefer the shorthand or, in some cases, a "backward compatibility" feel for other version control systems such as CVS. Next, it is worthwhile to understand the usage notation if you are not familiar with it already. The usage will specify which options and arguments the command will be expecting for input. When an option or argument is enclosed in brackets, it indicates that this not required. In most cases for optional arguments, the help will tell you what the default behavior for the command will be. As you can see from the `checkout` help, if you omit the PATH argument, it will be substituted with the base name from the URL. Figure 2.1 shows an example of a Subversion command line and the different componentsIt is important to understand what the default values are for optional arguments so you know what is going to be affected by the command. Finally, all the possible options for the command are listed.

2.2.3 Paths and URLs

Throughout Subversion, you will see the different notations for referring to locations in a directory or repository. The two will we will focus on now are PATH and URL; other administrative commands have other notations and will be addressed later. A PATH is simply a standard operating system directory. This notation is used for performing operations on working copy files and also for accessing the repository with the administrative client on the local machine. The second notation is URL, which is used when referring to a repository. Just like accessing a web site, a

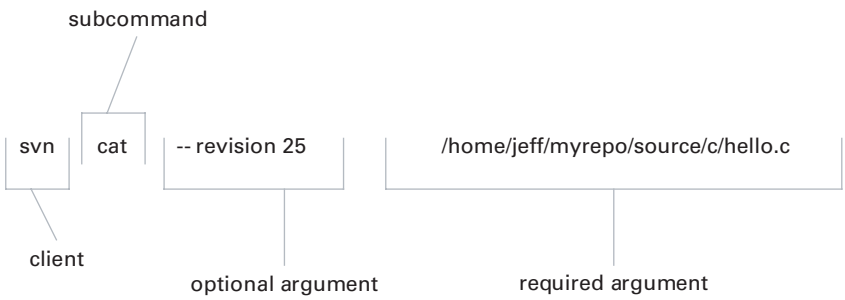

Figure 2.1 Command-line syntax

Subversion URL is a protocol followed by a path with the following notation: `protocol:///absolute/path`. Depending on how you access your repository, you will use one of the protocols described in table 2.1.

Table 2.1 Repository access protocols

Protocol	Description	Authentication	Port
`http`	Uses the `http` protocol to access an Apache web server with WEBDav/DeltaV pointing to the Subversion repository filesystem	Yes	80
`file`	Accesses a local filesystem	No	NA
`svn`	Uses the custom Subversion protocol	Yes	3690
`https`	Uses the `https` (`http + SSL` encryption) protocol to access an Apache web server with WEBDav/DeltaV pointing to the Subversion repository filesystem	Yes	443
`svn+ssh`	Uses the custom Subversion protocol but with `SSH`	Yes	3690

For example, if you are accessing a repository on the local system in the directory `/repos/testrepo`, your URL would be `file:///repos/testrepo`. If the local hostname is Pluto and you are accessing the same repository over the network, it would look something like the following URL: `http://pluto.domain.com/repos/testrepo`.

2.3 Creating a repository

Before you can start working with Subversion, you need a repository. If you have access to an existing one, great; if this is not the case, don't worry. Creating a repository is a surprisingly simple task in Subversion once you understand the basics of permissions and where the repository should be located. When you create a repository, all of the databases and configurations will be set up for you. Also, if you are creating one just for trying out some of the commands and concepts we will be discussing, getting the permissions and details is not critical. Once you are ready to create a production repository, the specifics we will be talking about come into play.

2.3.1 Finding a location for the repository

The first step to creating a repository is finding a place for it to reside. All repositories reside on local filesystems; they can be accessed locally or over one of the

network protocols listed in table 2.1. Repositories can be created anywhere on the local host with one exception; they should not be created or accessed on directories that are shared, including Windows shares, NFS, and Samba. You may be tempted to try this, thinking that you can use the local filesystem access URL (`file://`) and just mount it on a network share. Remember, Subversion uses Berkeley DB as the filesystem store; the way locking works in the database does not support access over a shared filesystem. If you try this, it may appear to work, but you will get unpredictable results and eventually put your repository in a bad state. If you need to access the repository over the network, you can use one of the protocols provided. Information to set these up can be found in chapter 7; for now we will use just local filesystem access.

Be sure the directory you use has enough space for the source code plus the back end. Subversion is a very efficient version control system in terms of disk space, but the size of the repository will be bigger than just the files you are loading. The size will depend on factors such as the number of changes, metadata attached to the files, and how your logging is set up. Unfortunately, there is no formula you can use to determine the amount of space needed. As you add content to the files and change logs, the filesystem will grow. The good news is that this will grow by only the size of the change, so it is more efficient than storing entire copies of files. Once you know where you want your repository, you will need to create the subdirectory for it. There is no standard or rule to the naming of the repository path. From here on, we will use `/repos` as the directory all Subversion repositories will be placed in. If you want your repository to be called `testrepo`, simply create the subdirectory `/repos/testrepo`.

2.3.2 Permissions

The method or protocol you use to access your repository will guide you in how the permissions will be set up for the Berkeley DB filesystem. While the examples will be in a Linux environment, the same concepts apply to a repository on a Windows machine. No matter if your users access the repository from the local machine or one of the network protocols, you should have the directory owned by a "system" user. You may want to create a user and a group called `svn`, or some derivative of that. This user should create all the repositories. You can also use this ID to perform administrative tasks on the repository. Even if there are only one or two users on a system, you should stick to this practice. If you keep the owner and administrator of the repository separate from the day-to-day users, the risk of causing problems will be greatly decreased. Just as in a UNIX environment, you should not do your work as root.

Permissions for network accessed repositories. When setting up permissions for a network repository, you have two options. The first option is to simply add the network user to the group that owns the repository. You would pursue this option in an environment where you access the repository over the network and locally. This is perfectly legitimate and can be used when security does not need to be extremely tight. The second option is to have one user with the permissions to access the repository at a filesystem level. This user would be the same ID that runs the Apache or svnserve process. Thus, the directory permissions would look the same as the previous listing, but the only user in the `svn` group would be the `apache` user, for example. This will prevent users on the repository machine from having the ability to directly write to the DB files.

UNIX users—Check your mask. No matter whether you connect to the repository locally or over a network protocol, it is critical to make sure the umask of the user is set correctly. Users, through the client interfaces, will create files in the Berkeley DB while working in Subversion. If the umask is set improperly, the files will be created with the wrong permissions. When other users try to access these files, they will get permission-denied errors. At a minimum, the umask should be set to 002. This goes not only for local users but also for the ID running the network service. You may have to set the umask for a network user in the startup script to be sure it is correct. This can easily be done in the Apache configuration file for the case of one network user. If users are accessing the repository locally, you will need to set all their umasks. If this is a security issue, you will have to write a wrapper script to `svn` that sets the umask and then calls the real `svn` command. For example, the following simple script will accomplish this for you:

```
#!/bin/bash
umask 002
/usr/local/bin/svn $*
```

Assuming Subversion has been installed in `/usr/local/bin`, this simple wrapper will change the umask for the duration of the script and then call the real `svn` command with the same parameters passed into it.

2.3.3 Svnadmin create

Once you have determined the location for your repository, the actual creation is extremely simple. The Subversion admin interface has a utility called `create`. This command is run locally from the machine on which the repository will reside. All it requires for an argument is the path to the repository directory:

```
$ svnadmin create /repos/testrepo
```

Congratulations, you have just created a Subversion repository! That's all there is to it. This will take only a few seconds and then return you right to the command prompt. There are not many options that can be used with the `create` operation. The few that do exist are advanced and do not need to be addressed now.

2.3.4 Internal repository directories and files

If you are curious and look in the `/repos/testrepo` directory, you will see a bunch of subdirectories. Table 2.2 briefly describes the files and directories in the repository directory. Most users will not have to work in these directories, especially if you are not sure about what they do. When we discuss advanced topics such as repository configuration and maintenance, we'll explain the use of these directories. Out of the box, Subversion provides a robust implementation that will prevent the need for accessing these areas.

Table 2.2 Repository directories and files

Name	Description
`README.txt`	A brief text file that indicates the directory is a Subversion repository and where some of the configuration files are located
`conf`	Directory containing configuration files for services such as the svnserve daemon
`dav`	Working area for the `mod_dav_svn` module for Apache `HTTP` server and WebDav / DeltaV protocols
`db`	The location of the Berkeley DB filesystem implementation
`format`	The file containing the Subversion filesystem layout
hooks	Directory to store all the hook or trigger scripts
`locks`	Special Berkeley DB directory to lock the back end for recovery or maintenance processes

Permissions for locally accessed repositories. For repositories accessed on the local machine, you will use the operating system group permissions to control authorization. The easiest way to do this is simply add the users who need access to the repository to the `svn` group you created previously. Make sure that the repository directory is recursively in this group and the group write permissions are turned on. Your listing should look like the following:

```
$ ls -ltr
total 28
-rwxrwxr-x    1 svn     svn          376 Mar 23 20:52 README.txt
```

```
drwxrwxr-x    2 svn     svn         4096 Mar 23 20:52 locks
drwxrwxr-x    2 svn     svn         4096 Mar 23 20:52 hooks
-rw-rw-r--    1 svn     svn            2 Mar 23 20:52 format
drwxrwxr-x    2 svn     svn         4096 Mar 23 20:52 dav
drwxrwxr-x    2 svn     svn         4096 Mar 23 20:52 conf
drwxrwxr-x    2 svn     svn         4096 Mar 23 21:36 db
```

While not all of the directories need to have group write, the one to worry about is `db`. If this directory is not set up correctly, users will be unable to check out or commit to the repository. If you have some experience with system administration, you will probably see a glaring hole in this: users will have the ability to write to these files. Depending on your user base, this may or may not be a problem. If this security setup is too loose for you, you can have your users access the repository through a network protocol and lock down the permissions more.

2.4 Common options

Now that you have your new repository, you are almost ready to jump in, but not quite yet. Before you start looking at the Subversion commands, it is a good idea to understand some of the common options and default behaviors. Since the client interfaces are a set of command-line executables, these common options are simply arguments passed into the command. This is no different than an OS command option, such as `-l` for the `ls` command. The common options we will talk about next are available in multiple commands in Subversion. An option will have the same behavior no matter what command it is in. For example, the `--username` option will pass in an ID and is available on the `commit`, `update`, and other commands.

2.4.1 Recursion

When you run a command that can operate on multiple files, by default it will recursively parse through the subdirectories. This will depend on what you use for an argument to the command: a file or a directory. If the argument is a file, then only that file will be processed by the command. If you pass in a directory, the command will process every file and directory under that. For example, consider the directory structure in figure 2.2. If you start the `add` command at the directory `/home/jeff/testrepo/source`, you will get everything under the `java` and `c` directories. While this may not seem that important in this example, consider a tree with hundreds of directories, five and six levels deep, without having recursion. You would need to write a script to add these files.

While recursion can be a great feature, there are times when you want to operate at only a single level. In our previous example, maybe you want to add only the

files in the `/home/jeff/testrepos/source` directory without going down multiple levels. In this case, Subversion commands that use recursion have a switch called `--non-recursive`. This tells the command to operate only on files at the same level in the directory tree as the target. So in the previous directory tree, the command would operate on the directories `c` and `java` and the file `config.txt`. Nothing will be done to the files under the `java` and c directories. As you will see in later chapters, there are a few exceptions where commands will operate nonrecursively by default, and you will need to instruct Subversion to be recursive.

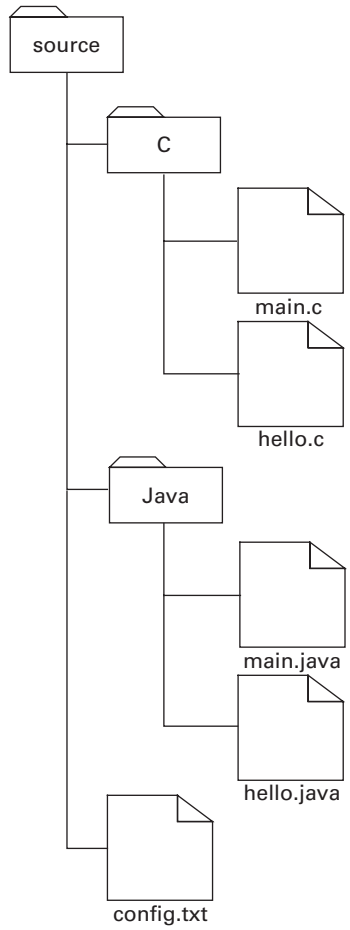

Figure 2.2 Example directory tree

2.4.2 Entering the log message

Any command that writes to the repository requires a log message to be submitted. This can be done automatically through command-line options or manually through an editor. In chapter 1, you saw the importance of entering a detailed log message. You should keep this in mind when determining how the message will be entered into a command. There are two general ways of entering a message: on a command line or through an editor. With the command line, you give the command either the text or the location of the text to use. If you do not specify this on the command line, Subversion will spawn a text editor, and you will enter the message there.

Entering the log message on the command line. There are two ways to enter the log message on the command line; the first is with the `--message` option. This option takes a string encapsulated in quotes as a parameter, which is the actual message:

```
$ svn commit --message "This is the log message"
```

The command line is a good tool if you are automating a process in a script and do not have a way to manually enter a message. It can also be used for messages that are short and that you want to quickly enter without having to run them through an editor. Finally, this is a good workaround if you cannot start an editor

for any reason; perhaps you are logged in through an old telnet client and your terminal session won't support it. Since you have to enter a log message for the commit to execute, using the command line for the message will allow you to continue your work even if you can't get into an editor.

If you need to enter a multiple-line message with the `--message` option, you can press the Enter key before terminating the quotes. When you have finished entering the message, you would then terminate that line with a quote:

```
$ svn commit --message "This is a
multiple line
log message"
```

You can run into some deficiencies using the `--message` option. The most obvious is the size of the message. Also, even though you can enter a multiline log from the command line, it can be difficult to use this method, because you may not be able to go back and fix typos in previous lines. As you will see with the default method, when you enter the log from the command line, you do not get a chance to see what changed.

Using a file's contents for the log message. Instead of typing in the log message on the command line, you can use a file's contents and still execute the command without manual intervention. One of the common uses for this option is reading from a separate bug-tracking or change-control tool. In most of these systems, you enter the bug and the fix in the defect-tracking tool. Instead of entering all the information a second time on the command line or in an editor, you can export the information from the defect-tracking tool into a file and use that as the log message. This gives you a good process for aligning your source code back into your defect-tracking tool and makes it easy for people to find what bugs were fixed in what version of the code. Also, most bug-tracking tools have the ability to easily export these messages. In place of the `--message` flag, you would use the `--file` option, which requires the path to the file that contains the log message:

```
$ svn commit --file /home/jeff/log_message.txt
```

Subversion will read all the text out of the file and use that for the log message on the `commit` command.

Using an editor. If you do not specify the `--message` or `--file` option on a command that writes to the repository, an editor will be started for you. In these cases, interaction with the command is required to execute. Subversion will try to use one of the following environment variables to determine the path to the editor:

SVN_EDITOR, VISUAL, or EDITOR. If you receive an error like this, it means that the system cannot find the editor or that the environment variable is not set:

```
svn: Could not use external editor to fetch log message; consider setting the
   $SVN_EDITOR environment variable or using the --message
(-m) or --file (-F) options
svn: None of the environment variables SVN_EDITOR, VISUAL or EDITOR is set,
   and no 'editor-cmd' run-time configuration option was found
```

At this point your default editor will start up and indicate what changes it will be acting on. This is one of the advantages of using an editor. You get to see exactly what is being saved to the repository before executing the command. You should always check this output and make sure that the files that are changing are what you anticipate. This will prevent unexpected commits to the repository such as adding non-versioned files or accidental modifications. Take a look at the example text that will be loaded into an editor:

```
--This line, and those below, will be ignored--
M    java/hello.java
A    c/hello.c
```

You will enter your log message in the area above the separator line and, as it says, any text below it will be ignored; this is just your sanity check. This does not mean you cannot include those lines, however. Some people like to include which files were changed in the log message. To include these lines, just delete the separator line in the editor, and everything will be saved. You may notice that the identifier characters before each file changed; this tells you what action is going to be taken with this save. Don't worry about these for now. We will discuss the identifiers when we talk about the different states a working copy can be in.

Overriding your default editor. Consider a situation where your EDITOR environment variable is set to an X Windows–based application. What if you are logged in on a session that prevents you from running a graphical program? The editor will try to start, but it will fail. If this happens, you can override the editor with the --editor-cmd option instead of changing the environment variables. For example, if your EDITOR variable is set to emacs and you want to use vi for this one commit, you could run the following command:

```
$ svn commit --editor-cmd vi
```

Using this option will start an editor; it will just ignore what you have set in your environment variable and use the argument you passed in. You will need to make sure the command you are using for the editor is in your path. To be safe, you

may want to use `/usr/bin/vi` to ensure that Subversion can find the executable for the editor.

A word about empty log messages. When you manually enter log messages, Subversion will open a temporary file in the editor. When you save and exit, that file will be used for the log message and the command will be processed. If you exit without saving the file or do not enter any text for the message, this file will be empty. When this happens, you will be prompted for action. This is good way to start over if you mess up your log message. When the editor quits, the output will look like this:

```
Log message unchanged or not specified
a)bort, c)ontinue, e)dit
```

As you can see, there are three options. The first is `abort`; this will cancel the current command without any action. The second option is `continue`; this will tell the command to continue execution and deal with the fact that there is an empty log message. Finally, if you select `edit`, your editor will start back up as if the command was run for the first time.

2.4.3 Accessing specific revisions

As we have talked about, one of the reasons for using a version control system is the ability to get older versions of a repository. More often than for checking out an older revision, you will need to access a previous revision to look at log messages or compare the contents of files. By default, all Subversion commands will also access the most recent revision of the repository. To change this, you can use the `--revision` option along with the specific revision you are accessing. You can access a revision by using one of three methods: revision number, date, or keyword.

Using the revision number. The most obvious way to access a repository version is to use the revision number. We have seen how Subversion implements these numbers; to get a specific revision, just use the number. For example, to look at the contents of revision 1 of the file `hello.java`, you would use the option `--revision 1` in the command:

```
$ svn cat --revision 1 hello.java

public class hello
{
}
```

Instead of using the working copy file, the command will access the repository to get the object. Whenever possible, Subversion will use the working copy to perform the operation to avoid going over the wire when it is not required. Since the

working copy stores only one version—the one you checked out—when you specify a different revision, the command must go back to the repository.

Revision date. There may be times when you want to access a file or the repository at a specific point. You could run through the change logs and find the revision number that correlates to the time, but that can be time consuming depending on the number of logs. To get around this, Subversion allows you to access revisions by date. When you specify a date, the command will start at that point and search backward in time until it hits a revision. You will still use the `--revision` option, but instead of a number you will enter the date inside braces, `{}`. For example, if you use the date 3/11/2004 on a checkout on the following repository, you will get revision 2. In the same example, if you select the date 3/09/2004 and a time of 15:00, you will get revision 1:

```
------------------------------------------------------------------------
r3 | jeff | 2004-03-12 22:32:34 -0500 (Fri, 10 Mar 2004) | 1 line
Added author tags to source files that were missing it
------------------------------------------------------------------------
r2 | alex | 2004-03-09 22:30:42 -0500 (Tue, 09 Mar 2004) | 1 line
Added print statement so we know it started
------------------------------------------------------------------------
r1 | jeff | 2004-03-09 06:28:18 -0500 (Tue, 09 Mar 2004) | 1 line
Imported Sources
------------------------------------------------------------------------
```

Subversion will accept multiple date formats, but we will stick to {YYYY-MM-DDTHH:MM}. This is interpreted as year-month-day; the T is a separator character between date and time, and the time is hour:minute. If you leave out the time, it will default to midnight, 00:00. Since Subversion searches backward, you will not get files on that day but the previous day instead. This happens because the Subversion checks for a revision that has the time 00:00; if it cannot find one, it moves backward to the previous day at 23:59. So if you want the files changed on 3/10, specify either a time of 23:59 or just the date of 3/11. For example, if a user reports an error that started on 3/10, you could use this for the starting point and see what the last change was before that date:

```
$ svn cat --revision {2004-03-11} hello.java
public class hello
{
        public static void main (String args)
        {
                System.out.println("Hello world");
        }
}
```

Using keywords. There is a set of keywords that represent where a revision sits in the version tree. They tend to be most useful when comparing your working copy against the repository. Just like the revision number and date, you use the `--revision` option with the keyword. The keywords listed in table 2.3 are available in Subversion.So, for example, let's say you have been making changes to

Table 2.3 Revision keywords

Keyword	Description
`HEAD`	The most recent revision in the repository.
`BASE`	The revision of the file that was checked out to your working copy. This is not the working copy you make modifications to. It gets saved in the `.svn` directory until you do a commit or update.
`COMMITTED`	This is the last committed revision of the file equal to or before the base revision. In Subversion, this will always equal `BASE`.
`PREV`	The revision just prior to the `BASE` revision.

the file `hello.java`. Now you want to see the original contents of the file before your modifications. You can simply use the `BASE` keyword to get the file as it looked when you checked it out:

```
$ svn cat --revision BASE hello.java
public class hello
{
}
```

2.4.4 Authentication

Until now, we have assumed that Subversion does not require login information to access the repository. If you are accessing the repository over one of the network protocols and do not have to provide this information, this is called anonymous access. Anonymous access means that anyone can read the repository without providing user information. If we look beyond open source, there are situations where you may not want anyone to be able to check out your repository. When authentication is required, you will need to supply a username and password to execute the command. The most straightforward way to do this is to supply the information as command-line arguments:

```
$ svn co http://pluto.domain.com/repos/testrepo --username jeff \
--password password1

A  test/file.c
```

```
A  test/source
A  test/source/java
A  test/source/java/hello.java
A  test/source/c
A  test/source/c/hello.c
A  test/config
A  test/config/hello.conf
Checked out revision 10.
```

If you are averse to supplying your password on the command line, you can leave this argument out. The `checkout` command will simply prompt you for a password before continuing. If you do not specify a username, the login ID on the local machine will be used.

2.5 Importing your files into Subversion

Before you can do anything in the repository, you must put some files in it. There are multiple ways of getting objects into Subversion. When you are adding an existing directory tree of non-versioned files for the first time, you will want to use the `import` command. Just as the name implies, it will import a non-versioned directory tree into the repository. This will add each file and directory to the repository and commit it. While this is the best way to load an empty repository, it does not have to be a one-time event. Any time you are adding a set of preexisting files or directories, you can use the `import` command.

2.5.1 A basic import

By default, the import will use the current directory as the target for the import; you need only specify the URL of the repository. The import will recursively parse the directory tree and commit the files. From the top-level directory of the structure you are importing, run the following command:

```
$ svn import file:///repos/testrepo

Adding         source
Adding         source/java
Adding         source/java/hello.java
Adding         source/c
Adding         source/c/hello.c
Adding         config
Adding         config/hello.conf

Committed revision 1.
```

Each of the files and directories imported is listed in the output of the command. The new revision number created will also be shown. Since we were loading the code into an empty repository, this is revision 1. Be aware that the current directory is not a working copy. You have simply loaded a set of non-versioned files into the repository. Before you can start making changes, you will need to check out a working copy of the repository. It is best to find a new location other than the one from which you have just imported.

2.5.2 Changing what you import

Subversion provides the ability to specify a path to import instead of just using the current directory. Use this when you are using scripts or automation, so you do have to keep changing directories and keep track of where you started. For example, if you wanted to load only the source directory of the file tree, you would add that path to the import. The path should be the argument before the URL:

```
$ svn import -message "Initial Load" \
myfiles/source file:///repos/testrepo

Adding         myfiles/source/java
Adding         myfiles/source/java/hello.java
Adding         myfiles/source/c
Adding         myfiles/source/c/hello.java
```

Even though the listing shows the absolute path of the file, the files are added to the top level of the repository. Also, the files in the directory are added, not the base directory itself. So in our example, source is not added, but each of the files and directories is. So if we make a listing of the repository after the import, it will look like this:

```
$ ls
c  java
```

In addition to changing which path you are importing, you can also limit the import to a single directory as opposed to recursively parsing the tree. You can accomplish this by adding the option `--non-recursive` to the `import` command. To see how this will change the import, look at the output from the command:

```
$ svn import --non-recursive -message "Initial Load" /
myfiles/source file:///repos/testrepo

Adding         myfiles/source/java
Adding         myfiles/source/c
```

Since there are only two objects in the directory `myfiles/source`, those are all that are imported into the repository.

2.5.3 Changing where you import

The imports we have seen so far have put the code in the top directory level of the repository. This is fine if the files are supposed to reside there and the directory structure is the same, but such is not always the case. For instance, what if you have a `php` directory that you want to import into the existing repository we created? You could make sure the directory structure in which the `php` directory exists is the repository beforehand, or you could specify this on the command line:

```
$ svn import file:///repos/testrepo/source/php
Adding         main.php
Adding         hello.php

Committed revision 5.
```

The `import` command will not only import these files, it will create the directory `source/php` in the repository.

2.6 Checking out the repository

Now that you can load some files and directories into the repository, you can check out a working copy. Think back to chapter 1 when we talked about accessing the repository. The way you interact with Subversion is through a working copy. You get this by checking out the repository. Even if you have a new repository with nothing in it, you will still need to do a checkout to start. Since all the saves to a repository happen through a commit from a working copy, you must establish the working copy. An empty repository will create just an empty directory that will be your working copy.

2.6.1 Finding your starting point

In Subversion you can check out the entire repository, a directory tree, or a single directory. This decision is based on personal preference and the setup of your repository. The main factor in how you check out is size, of both the repository and the available space on your local system. If you have a small repository, it is usually easier to check out the whole thing. When a repository is large, or if you have a limited amount of space for your working copy, getting the entire copy may not be practical. If your repository is large, it should have been broken into multiple projects, and then into modules within a project. Having multiple projects

with clear separation points in the repository allows you to check out only what you need.

As we will demonstrate later in this section, you do not have to check out the entire repository. So if each project has its own top-level directory in the repository, you won't have to check out a lot of extra code just to get your stuff. We will talk more about laying out the repository in chapter 5. Now you will need to find the base directory in the repository that will give you all the dependencies you need for development. By default, when you perform a checkout, it will be recursive. So if you get the top-level directory for a project, Subversion will also give you every subdirectory under it.

2.6.2 A simple checkout

To perform a checkout, you will use the `co` command from the `svn` interface. The only required field is the URL to the directory in the repository from where you are starting. If you are checking out the entire repository, you will use just the URL. Subversion will create a working copy in the directory from which you run the command. The directory name will be the base name from the URL specified in the command. For example, to check out the repository `testrepo` on the local box, you would use the URL `file:///repos/testrepo`:

```
$ cd /home/jeff

$ svn co file:///repos/testrepo
A  testrepo/source
A  testrepo/source/java
A  testrepo/source/java/hello.java
A  testrepo/source/c
A  testrepo/source/c/hello.c
A  testrepo/config
A  testrepo/config/hello.conf
Checked out revision 9.
```

The command has created a directory called `testrepo` in the current directory from which it was run—this is your working copy. Each of the files and directories checked out is listed, and they will all have an `A` preceding them indicating that they were added to the working directory. A new checkout will always have an `A`; when we look at updates you will see other values that can occur. In addition to listing the files, the output will also tell you which revision of the repository was checked out. Remember that revisions are based in the repository, not individual files. It is implied that each file has the same global revision number.

2.6.3 The .svn directory

If you go into any of the directories in the working copy and do a list, you will see a directory called `.svn`. For users familiar with CVS, this is equivalent to the CVS subdirectory. The `.svn` directory contains information that Subversion uses to provide efficiency. There is really no need to do anything in this directory.

2.6.4 Checking out specific versions

By default, when you perform a checkout, you get the latest version in the repository. While the majority of the time this is what you want to do, there are situations where you will need to get a previous revision of the repository. For example, if a new change is causing problems in your code, you may just want to get the previous revision of the file to see whether the error was just introduced or has always existed. To accomplish this, you will use the `-revision` switch on the `checkout` command. As with the default, you can start the checkout at any directory level in the repository. There are two ways to specify which revision of the repository to check out. The obvious way is by revision number; the other is a set of keywords that represent a revision of the repository.

Specific revision numbers. The most straightforward way to get a previous revision of a file is to simply check out the repository using the revision number. The most common use for this is when you have traced an issue back to a specific change. For example, if your most recent revision is 9 and you think a problem was introduced in revision 5, to prove this you will need to re-create the application at this revision:

```
$ svn co --revision 5 file:///repos/testrepo
A  testrepo/source
A  testrepo/source/java
A  testrepo/source/java/hello.java
A  testrepo/source/c
A  testrepo/source/c/hello.c
A  testrepo/file.c
A  testrepo/config
Checked out revision 5.
```

The output from this command is similar to the default checkout, except the revision number is 5 instead of 9. Notice that there is one other difference: the file `testrepo/config/hello.conf` does not appear in this listing. This would indicate that the file was added to repository sometime after revision 5. Even though the other files are listed in the output of both commands, it does not

mean the contents are the same. The files and directory structure will look identical to how they did when revision 5 was committed.

Other keywords. You can use any of the keywords in your checkout instead of using specific revision numbers. All the keywords described in section 2.4.3 are valid, but most are not very helpful for the `checkout` command. For example, you cannot use the `BASE` keyword because that is generated from the working copy, which you don't have yet.

A warning about mixing revisions. We have shown the flexibility of Subversion, which allows us to check out specific versions of files and also specific directories within the repository. The combination of these allows the working copy to have different revisions of files. For example, if you have two directories called `java` and `c` in your repository, you can check them out separately into your working copy:

```
$ svn co file:///repos/testrepo/source/java --revision 5
A  java/hello.java
Checked out revision 5.

$ svn co file:///repos/testrepo/source/c --revision 6
A  c/hello.c
Checked out revision 6.
```

Now we have revision 5 of `java` and revision 6 of `c` in the working copy. There may be a legitimate reason for doing this. You may wish to temporarily back out a change in Java code and not in C code. Although Subversion permits it, you should not practice this regularly. There are no guarantees that everything will work, especially if you have moved files or renamed them in the repository. This practice gets especially dangerous if you try to check out different revisions in nested directories.

2.6.5 Changing the paths

So far, we have used the `checkout` command with one URL going to the default working directory base name. Suppose you find yourself in a situation where you are testing different configuration options for the application to see the results. These options are stored in files in the `config` directory. Instead of checking out another working copy, it may be easier to create another directory in the same working copy with a different name:

```
$ ls
config  source

$ svn co file:///repos/testrepo/config config_test
A  config_test/hello.conf
```

```
Checked out revision 9.

$ ls
config  config_test  source
```

First, we did a listing of the working copy and saw two directories: `config` and `source`. In the `checkout` command, we have added a new argument: the base name of the target directory we are checking out to, `config_test`. Now when we do a list, we see `config` and `config_test`. This will allow you to change one directory and still have a clean copy of the other. Even though you have called the directory something different in your working copy, it still points to the same location in the repository. So if you make changes to the `config_test` directory and commit, it will update the original `config` directory in the repository. You can consider these renamed base names as symbolic links; even though the names in the working copy are different, they still point to the same place in the repository.

2.7 Committing changes

Thus far you have imported your current source into the repository and checked out a working copy. Now you go in and start making changes to the files. When you are ready to save your changes to the repository, simply run the `commit` command from the directory where your changes are. Remember, by default the command will recursively parse all the subdirectories so you do not have to manually do this. Let's say you made changes to two files, `hello.java` and `hello.c`. To check these in, simply start at the top level of the repository and run the commit:

```
$ svn commit --message "Added call to read configuration file
> for application specific parameters"

Sending        c/hello.c
Sending        java/hello.java
Transmitting file data ..
Committed revision 9.
```

As you can see, the commit searched the subdirectories and found the two files that changed. The other files in the working copy that did not change were left alone. The output of the command shows you which files have been transferred; while it is too late to change anything, it is a good verification. When the commit is successful, it will inform you as to what revision number your change has created.

You may have noticed that no PATH or URL was specified in the `commit` command. So how does Subversion know to which repository to send the changes? This is where our friendly `.svn` directory helps us out. When you did the

checkout, the repository URL was stored there. The commit reads this directory to find the URL, so you do not have to type it in each time you perform a commit. This directory also stores the original copy of your files, so Subversion can use this to see if your files have changed instead of running the compare over the network, which is much slower.

2.7.1 A path to commit

As you saw in the basic `commit` command, you do not need to specify the target file or directory you are committing. The command is just run from the current directory and finds everything under it. But what if you do not want to change directories to run the command? Subversion allows you to specify a target path against which to perform the commit. If you just wanted to commit the `java` directory from the previous example, you would add the path to the command:

```
$ svn commit --message "Added call to read configuration file
> for application specific parameters" /home/jeff/testrepo/java

Sending        java/hello.java
Transmitting file data ..
Committed revision 9.
```

This command can to be run from anywhere on the working copy host machine. The `.svn` in the target directory will be used, not the one from the current directory (if there even is one). Since Subversion used the target's `.svn`, you do not need to worry about getting the correct URL, even if you run this outside the working copy.

2.7.2 What and when do you commit?

This may sound like a silly question to ask, but many experienced developers have very different opinions on this matter. With the new paradigm of atomic commits in Subversion, some of the more traditional processes should be challenged. You can get more value out of the tool if you consider it a source code management tool instead of just a place to store your files. While there is no right answer to this question, there are a few guidelines that you should follow.

Do not be afraid to checkpoint your files. There is nothing worse than someone checking out a set of files and not committing for weeks at a time. For one thing, you put your changes at risk because if you need to back out or revert after three weeks of changes, you have only one way to go back—to the beginning. Your three weeks of work are gone. Another problem with not committing is that your working copy and the repository get more and more out of sync. You will not see

changes that could cause your code to be invalid. This works both ways: your changes may cause another piece of code to be changed. As this process goes on, the amount of work required to reconcile the differences increases.

Commit at bug fixes or feature completion. While you should checkpoint your changes, try not to fall into the trap of committing only before you shut down your computer. This will make the change logs sloppy and difficult to trace. You should commit at logical stopping points, for instance, when a bug fix is complete, even if you are going to fix a dozen bugs in one day. If you commit a group of unrelated changes at once, you add an unnecessary layer of dependency. Now you cannot back out or trace just one bug fix or enhancement; you have to do this as a group.

Consider each commit as a mini-release. If you go through the thought process of each commit being a release, you will have a very logical revision tree. Not only is timing important, as we discussed in the previous section, but knowing which files you group as a commit is also useful. Each time you commit, ask yourself whether you can back out the change easily. If there are too many dependencies, you are being too granular. If backing out one change will also revert a group of unrelated changes, you need to get more specific.

Remember that these are only suggestions, so you should not get hung up any of them. Ultimately, the application will have properties tied to files to determine a release. But if you keep these suggestions in mind, your development process will be smoother and the other members of your team will be thankful.

2.7.3 Out-of-date transactions

Inevitably you will run into a situation where you try to commit a file and someone else has already done so. When this happens, you will get an error back from Subversion that looks like the following output:

```
$ svn commit --message "Added hello world statement"
Sending        hello.c
svn: Commit failed (details follow):
svn: Out of date: 'hello.c' in transaction '8'
```

First, when this happens, don't panic or start swearing at your co-developers. At this point, Subversion is just telling you that someone else has made changes and you need to sync up. You will need to run an update to get the latest repository changes. Don't worry about the details of the `svn up` command; we will cover that topic in detail on the next chapter:

```
$ svn up
G  hello.c
Updated to revision 4.
```

If the transaction identifier (the character before the filename) is a `G`, you are in good shape. This means the repository revision and your working copy have been successfully merged. Now you can run your commit again, and you should get better results:

```
$ svn commit --message "Added hello world statement"
Sending        hello.c
Transmitting file data .
Committed revision 5.
```

This time the commit worked without any problems. The original file was revision 3, but someone else committed revision 4 before you could. As soon as `hello.c` was updated with the changes from revision 4, you could commit revision 5. Now you are completely in sync with the repository, and your working copy and both sets of changes are saved in the repository.

You just saw the beauty of the copy-modify-merge model, but what would have happened if the update did not come back with a `G`? This will happen if Subversion cannot automatically merge the two revisions of the file. Look at the following output from an update where this is the case:

```
$ svn commit --message "Added the argc and argv parameters"
Sending        hello.c
svn: Commit failed (details follow):
svn: Out of date: 'hello.c' in transaction 'c'

$ svn up
C  hello.c
Updated to revision 6.
```

Notice that instead of a `G`, there is now a `C`. This indicates that the changes between the repository revision and your working copy conflict. Now it might be time to start swearing. You will have to manually merge the changes or revert your changes and start over. Let's take a look the file `hello.c` and see what this update has done:

```
/*
 * Author: Jeff
 */
<<<<<<< .mine
void main(String argc[], int argv)
=======
int main()
>>>>>>> .r6
```

```
{
  printf ("hello world");
}
```

The conflicting lines are both in the file in the format of a UNIX `diff` command. You can either accept one of the two lines or combine them. Either way, this will be done by manually editing the file and removing the extra lines from the `diff`. We'll cover this process in detail when we talk about merging and explore other options such as graphical merge tools.

2.8 Summary

You now have the basic skills to at least get Subversion set up and to start using it in a single-user environment. With any open source project, Subversion will move quickly in terms of releases and patches, especially in the early stages of its life. There is no formula to determine when to get updates or new versions of the software; you will just need to keep an eye on things. If the system is doing everything you need it to, and you are not hitting any bugs, stick with distribution you have. Usually these types of software applications get easier to install and upgrade with time, so Subversion will probably follow this trend.

Before moving forward into the next chapter, make sure you have a good understanding of the syntax of the client interfaces and the common options described. Many of the commands will rely heavily on these options, so it is important that you know what the behavior will be. If you are coming from a background in version control systems, such as CVS, this should look very familiar. But if you are new to this field, it is worthwhile to spend a little extra time on these options now. Later if you find yourself worrying about what the specific options do, it will be more difficult to understand the higher concepts being addressed.

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.

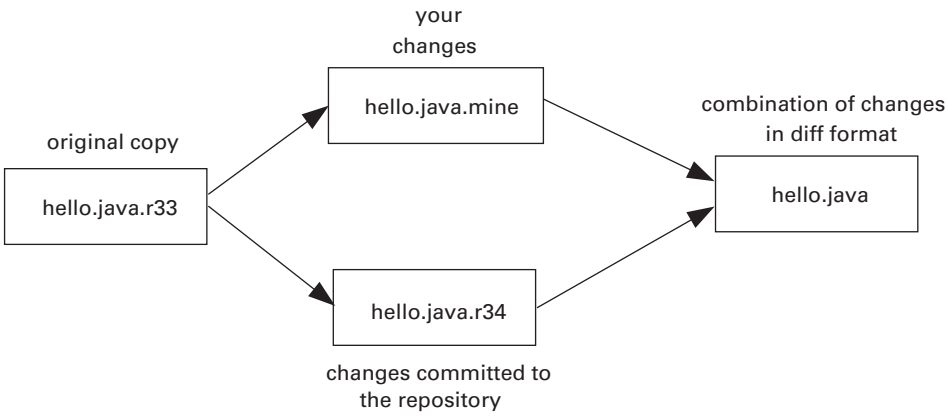

Figure 3.1 Events to get conflict-resolution files

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.

4 Getting change information

In this chapter

- Viewing revision history and change logs
- Getting the contents and system information from a file
- Finding differences in the contents of a file
- Getting detailed change information about each line in a file

We have been moving from theory to application over the first few chapters. Here we will get into the meat of Subversion and see how many of the "questions" get answered. When you are finished with this chapter, you will be able to find out the who, when, and why about changes to any of the objects in your repository.

4.1 Viewing change logs

Finally, here it is! When you think of version control systems, one of the main features that comes to mind is change logs. Subversion differs from other version control systems in that the change logs are based on the repository, not individual files. When you look at the logs on a specific file or directory (an object), only the revisions of the repository that the object changed will be displayed. This allows Subversion to base revisions on the repository but also allows you see at which points the files changed.

4.1.1 Running the log command

We have seen plenty of examples of change logs so far; now let's look at how to generate them. To examine a specific file, run the `svn log` command with the file as a parameter:

```
$ svn log main.c
------------------------------------------------------------------------
r40 | jeff | 2004-04-06 20:07:05 -0400 (Tue, 06 Apr 2004) | 1 line

Added graceful exit to main function
------------------------------------------------------------------------
r32 | alex | 2004-04-03 14:42:25 -0500 (Sat, 03 Apr 2004) | 1 line

Added license header information
------------------------------------------------------------------------
r9 | jeff | 2004-04-02 20:28:10 -0500 (Fri, 02 Apr 2004) | 2 lines

Fixed Bug (322)
Had a typo in hello world output
------------------------------------------------------------------------
r3 | jeff | 2004-04-01 22:06:50 -0500 (Thu, 01 Apr 2004) | 1 line

initial version
------------------------------------------------------------------------
```

All the revisions in the repository for the file `main.c` are listed in logs 3, 9, 32, and 40. Notice the order of the revisions; the newest revision appears at the beginning of the output. If you do not specify a file or directory as an argument to the command, the current working directory will be used. Directories are really no different than files in terms of the `log` command.

4.1.2 Change logs for directories

When you look at the change logs of a directory, you will see the repository revisions where anything under the directory has been changed or modified, and yes, it is done recursively. So, for example, consider the following directory contents:

```
$ pwd
/home/jeff/projects/testrepo/source/c

$ ls
config.xml  file.c  hello.c  main.c
```

If you change any of the files in this directory, its change logs will be updated. Let's make a change to `main.c` and then look at the logs. For the following example, the only thing we modified is the contents of `main.c`:

```
$ svn commit --message "changed the main() to return an int"
Sending        c/main.c
Transmitting file data .Committed revision 8.

$ svn log main.c
------------------------------------------------------------------------
r8 | jeff | 2004-04-17 10:05:26 -0400 (Sat, 17 Apr 2004) | 1 line

changed the main() to return an int
------------------------------------------------------------------------
r2 | jeff | 2004-04-16 20:06:26 -0400 (Fri, 16 Apr 2004) | 1 line

Fixed directory structure
------------------------------------------------------------------------
r1 | jeff | 2004-04-16 20:04:59 -0400 (Fri, 16 Apr 2004) | 1 line

Initial Load
------------------------------------------------------------------------
```

As you can see, the commit created revision 8 of the repository. When we look at the log history for the file, revision 8 is in the change log. Now let's see what the directory's change log looks like:

```
$ pwd
/home/jeff/projects/testrepo/source/c
$ svn up
```

```
At revision 8.

$ svn log .
------------------------------------------------------------------------
r8 | jeff | 2004-04-17 10:05:26 -0400 (Sat, 17 Apr 2004) | 1 line

changed the main() to return an int
------------------------------------------------------------------------
r6 | alex | 2004-04-17 09:44:11 -0400 (Sat, 17 Apr 2004) | 1 line

added hello world statement
------------------------------------------------------------------------
r3 | alex | 2004-04-17 09:40:25 -0400 (Sat, 17 Apr 2004) | 1 line

removed old GUI directories
------------------------------------------------------------------------
r2 | jeff | 2004-04-16 20:06:26 -0400 (Fri, 16 Apr 2004) | 1 line

Fixed directory structure
------------------------------------------------------------------------
r1 | jeff | 2004-04-16 20:04:59 -0400 (Fri, 16 Apr 2004) | 1 line

Initial Load
------------------------------------------------------------------------
```

The change logs for this directory also contain revision 8 because the file that was modified in that revision resides in this directory. If you move up one directory to the source, revision 8 would also be in its change log. So using this logic, if you run the `log` command on the top level of the working directory, you will see every revision. This is assuming that you have checked out the entire repository, of course.

Why doesn't the directory show the change logs after a commit? You may have noticed in the previous example that after the commit and before running the `log` command we did an update. If you modify a file and commit, and then you run the `log` command on the directory immediately afterward, the new change log will not be there. After the update is run, the change logs will be up-to-date. This happens because the commit is a push command, meaning that it writes changes to the repository only from the working directory. So in our example, `main.c` was changed, and it was the only object to get updated in the working copy. When you run the update, Subversion tells the directories which objects under them changed. Now the logs will reflect the modified children under the directory. Some people may consider this a debatable subject—since the logs of the modified file are created on the commit, the logs for the parent directories should also be created right away. Because this is an intentional

design in Subversion, it is not likely to change, so you should just be aware that an update is required to bring all the directory change logs in sync.

4.1.3 Suppressing the log messages

When we talked about running a commit, an important point was providing a descriptive log message so you will have better traceability when looking through the history. While this holds true, as your repository and history grow, you may have an extensive change output to sort through. What if you have a large change log and you want to quickly see the revisions in the repository that a particular file changed? This would be difficult when the logs look something like this:

```
------------------------------------------------------------------------
r1267 | jmachols | 2003-11-14 22:45:33 -0500 (Fri, 14 Nov 2003) | 9 lines

Modified the add and modify interfaces.  Instead of making the
add and modify operations parse out an input stream and handle
all the entries, the operations will only accept one entry at a time.
The caller will need to use the parsing class and pass the entries
in one at a time.  This will clean up the code and handling of the response
   for the caller.  It is much easier to write a loop and pass all the LDIF
   entries in than deal with a list of error codes and figure out which error
   code is associated with each LDIIF entry
------------------------------------------------------------------------
....
------------------------------------------------------------------------
r1038 | jmachols | 2003-08-30 22:43:13 -0400 (Sat, 30 Aug 2003) | 4 lines

Added shell for input parser.  This class will read through an input stream
   and separate out each LDIF entry.  THIS IS NOT AN LDIF PARSER!!  It just
   separates the entries and handles the input stream.
------------------------------------------------------------------------
r1037 | jmachols | 2003-08-30 12:49:01 -0400 (Sat, 30 Aug 2003) | 4 lines

Base embedded client package.  These classes will be used to allow an
application to integrate ldap clients.  The clients will use the protocol
   overwire to talk to any ldap server
------------------------------------------------------------------------
```

If this is but a snippet of the logs, it will be extremely cumbersome to just find the revision numbers. To get around this problem, Subversion provided the `--quiet` option on the `log` command to suppress the log messages.

```
$ svn log --quiet LdapParser.java
r1276 | jmachols | 2003-11-19 23:29:59 -0500 (Wed, 19 Nov 2003)
------------------------------------------------------------------------
r1262 | jmachols | 2003-10-31 16:00:38 -0500 (Fri, 31 Oct 2003)
------------------------------------------------------------------------
```

```
r1259 | jmachols | 2003-10-27 21:17:48 -0500 (Mon, 27 Oct 2003)
------------------------------------------------------------------------
r1038 | jmachols | 2003-08-30 22:43:13 -0400 (Sat, 30 Aug 2003)
------------------------------------------------------------------------
r1037 | jmachols | 2003-08-30 12:49:01 -0400 (Sat, 30 Aug 2003)
```

When you use this switch, the `log` command will give you only the system information from the change logs. This guarantees that each revision will take up only one line in the output. Look how much easier it is to quickly sort through this output than the previous version, which contains all the log messages. Once you figure out the revision number to look at, you can refine the `log` command to give you only a specific revision number's change log.

4.1.4 Logs for specific revisions

As we explained in the previous section, sometimes you need to search through a large set of logs quickly, so you will leave out the actual log message content. Once you know which revision should be investigated, you can use the `--revision` switch on the `log` command to get the change on a specific revision of the file instead of all of them. The following command will get the change logs for revision 62:

```
$ svn log --revision 62 main.c
------------------------------------------------------------------------
r62 | root | 2004-04-17 19:29:36 -0400 (Sat, 17 Apr 2004) | 1 line

Added return statement to main function
------------------------------------------------------------------------
```

No matter how many revisions are before or after 62, since that was the one specified, it will be the only one displayed.

Getting a range. We used an example of getting a single revision, but what if you want a range of revisions? Instead of running the `log` command once for each file, or having to parse the output of every revision, you can specify a range of revisions:

```
$ svn log --revision 35:30  main.c

------------------------------------------------------------------------
r34 | alex | 2004-04-03 18:31:47 -0500 (Sat, 03 Apr 2004) | 1 line

Fixed memory leak in malloc function
------------------------------------------------------------------------
r32 | alex | 2004-04-03 14:42:25 -0500 (Sat, 03 Apr 2004) | 1 line

Fixed check style problems
------------------------------------------------------------------------
```

We specified the range of 35:30. Note that the syntax places the higher number first. Since the higher number revisions are first by default, we specified the range in descending order to keep the output matching. By using this range, you are telling Subversion to give you only the revisions that fall between the two numbers. In this case there are two revisions that match, 34 and 32. The numbers in the range are inclusive, so the range of 34:32 would have yielded the same results. The date and other revision keywords also apply to other `log` commands, as with the `update` and `checkout` commands.

Change logs in a date range. A common situation you may find yourself in is trying to figure out what, if any, changes happened during a certain time frame. You can easily accomplish this using the `--revision` switch, but instead of using revision numbers for the arguments, you can use dates. Remember, since all options in Subversion have the same behavior and syntax, these will look like the dates we used to check out old revisions of the repository. For example, if you wanted to get all the changes made in the file `main.c` between 4/05/2004 and 4/17/2004, you could use the `log` command with the following syntax:

```
$ svn log --revision {20040417T23:59}:{20040405T23:59} main.c
------------------------------------------------------------------------
r61 | alex | 2004-04-16 22:18:04 -0400 (Fri, 16 Apr 2004) | 1 line

added header information
------------------------------------------------------------------------
r44 | jeff | 2004-04-06 20:59:20 -0400 (Tue, 06 Apr 2004) | 1 line

changed the GUI to have a menu bar
------------------------------------------------------------------------
r40 | jeff | 2004-04-06 20:07:05 -0400 (Tue, 06 Apr 2004) | 1 line

added include for time and date functions
------------------------------------------------------------------------
```

So according to the change logs, the file has been modified three times between the dates we specified. Also, notice the order of the dates on the command line. Since the higher version number will correspond to the more recent date, to keep the same order in the output we entered the dates in descending order. So far, we have been making an effort to keep the order of the revisions in descending order to match the default output. This is by no means required, and you can change this order to meet your needs.

Reversing the order of the logs. In the previous examples, we saw the output from `log` command start with the most recent revision and work backwards. In fact, we have been making sure to specify the revisions in descending order

when using a range. You may wish to have the earliest version print first and the most recent print at the end of the output. While there is no reverse option that you can use, by rearranging the order of the range on the `--revision` switch, you can achieve the desired result. Again, you can use revision numbers, dates, or keywords. Let's look at the example we used with the date, but in the reverse order:

```
$ svn log --revision {20040405T23:59}:{20040417T23:59} main.c
------------------------------------------------------------------------
r40 | jeff | 2004-04-06 20:07:05 -0400 (Tue, 06 Apr 2004) | 1 line

added include for time and date functions
------------------------------------------------------------------------
r44 | jeff | 2004-04-06 20:59:20 -0400 (Tue, 06 Apr 2004) | 1 line

changed the GUI to have a menu bar
------------------------------------------------------------------------
r61 | alex | 2004-04-16 22:18:04 -0400 (Fri, 16 Apr 2004) | 1 line

added header information
------------------------------------------------------------------------
```

As you can see, we still got back the same revisions, but the order is now reversed. The most recent revision appears last in the output. The same goes for revision numbers and keywords; if the range is specified in ascending order, the output will follow in ascending order.

One of the nice features of the `--revision` switch is the ability to mix revision numbers, dates, and keywords on the command line. Even though there is no reverse option to change the order in which the revisions are displayed, you can still use `HEAD` in the range. The following command will give the reverse order using the entire range of revision numbers in the repository:

```
$ svn log --revision 1:HEAD main.c
------------------------------------------------------------------------
r1 | jeff | 2004-04-03 09:42:25 -0500 (Sat, 03 Apr 2004) | 1 line

Initial Revision
------------------------------------------------------------------------
r32 | alex | 2004-04-03 14:42:25 -0500 (Sat, 03 Apr 2004) | 1 line

Fixed check style problems
------------------------------------------------------------------------
r34 | alex | 2004-04-03 18:31:47 -0500 (Sat, 03 Apr 2004) | 1 line

Fixed memory leak in malloc function
------------------------------------------------------------------------
r40 | jeff | 2004-04-06 20:07:05 -0400 (Tue, 06 Apr 2004) | 1 line
```

```
added include for time and date functions
------------------------------------------------------------------------
r44 | jeff | 2004-04-06 20:59:20 -0400 (Tue, 06 Apr 2004) | 1 line

changed the GUI to have a menu bar
------------------------------------------------------------------------
r61 | alex | 2004-04-16 22:18:04 -0400 (Fri, 16 Apr 2004) | 1 line

added header information
------------------------------------------------------------------------
```

All of the revisions in which `main.c` has been modified are here and in reverse order.

Revisions where no changes have been made. Now you know that when you look at an object's change log, the revisions of the repository in which that object was modified will be listed. Revisions where the file was not changed will simply not be displayed. But what happens if you specify a revision of a file using the `--revision` switch where the file was not changed?

```
$ svn log --revision 33 main.c
------------------------------------------------------------------------

$
```

All that is displayed in this situation is a separator line. This tells you that the file was not modified in the revision you specified. This also holds true when using a range; if the object was not changed within the range specified, only the separator line will be displayed.

4.1.5 Viewing the change paths

Think back to chapter 2 when we talked about entering log messages in your editor. When the editor starts, it shows you all the files that were modified and are about to be committed. These lines are ignored unless you specifically remove the `ignore` statement. Well, guess what—Subversion knows how important that information is, so it was saved anyway. We saw some examples of this in previous chapters when viewing logs. To view the changed paths in your output, add the `--verbose` switch to the `log` command:

```
$ svn log --verbose hello.c
------------------------------------------------------------------------
r30 | alex | 2004-04-03 14:10:57 -0500 (Sat, 03 Apr 2004) | 1 line
Changed paths:
   M /source/c/hello.c

Added function call to logger
```

```
------------------------------------------------------------------------
r9 | jeff | 2004-04-02 20:28:10 -0500 (Fri, 02 Apr 2004) | 2 lines
Changed paths:
   A /source/c/file.c
   M /source/c/main.c

Added hello world print statement, this is written to a file opened.  Added
   file.c to deal with file handlers.
------------------------------------------------------------------------
r1 | jeff | 2004-04-01 22:06:50 -0500 (Thu, 01 Apr 2004) | 1 line
Changed paths:
   A /source
   A /source/c
   A /source/c/main.c
   A /source/java
   A /source/java/main.java

Initial Revision
------------------------------------------------------------------------
```

What you should notice in this output compared to our previous examples is the extra information under `Changed paths`. This output lists all the files and directories that were changed in that revision of the repository, plus what action was taken on them. In revision 1, three directories and two files were added to the repository. In revision 9, `main.c` was modified and `file.c` was added to the repository. Since the `log` command was run on the file `main.c`, obviously it will be in all the paths. The log will also include any other file or directory that was changed on that commit.

Getting the object's origin. When we examined the `copy` and `move` commands, there was a problem in looking at the history and knowing whether the file was really part of the older revisions or if was copied in a later one. When you run the `log` command with the changed paths, you can clearly see if this is the case. Take a look at the following change logs to see this illustrated:

```
$ svn log --verbose HelloWorld.java
------------------------------------------------------------------------
r64 | jeff | 2004-04-17 23:26:27 -0400 (Sat, 17 Apr 2004) | 2 lines

Fixed the internal class name to match
the new file name
------------------------------------------------------------------------
r63 | jeff | 2004-04-17 22:54:55 -0400 (Sat, 17 Apr 2004) | 1 line
Changed paths:
   A /source/Java/HelloWorld.java (from /source/Java/hello.java:60)
   D /source/Java/hello.java

Made some adjustments
```

```
------------------------------------------------------------------------

r4 | jeff | 2004-04-01 23:01:15 -0500 (Thu, 01 Apr 2004) | 1 line
Changed paths:
   A /source/java/hello.java

Create a new file hello.java to print hello world
------------------------------------------------------------------------
```

Look at the changed paths for revision 63; it shows that the file `HelloWorld.java` was added, but do you notice anything different about this `add` versus other ones? In this revision, there is an extra statement after the filename that specifies a `from`. This is Subversion telling you were the file came from. In a "normal" case when nothing is there, it means the object was added from a non-versioned file, either by an `import` or `add` command. But when the `from` is in the output, the object was created from another Subversion object, either by a `copy` or a `move` command. This extra information will also tell you the exact origin of the object. Go back to the output in the example, and you can see that `HelloWorld.java` came from the file `hello.java`, specifically, revision 60.

The verbose output tells you that the object originated from another Subversion object by the `copy` or `move` command. In most cases, you can actually deduce whether it was a copy or a move. This is done by comparing the source in the `from` statement to any deletes in the same commit. Remember that a move in Subversion is a copy followed by a delete. So if you see the source file deleted in the same commit as the new object is added, chances are this is a move. If there are no deletes, it is likely a copy. Now remember, any revision of the repository before 63 will not contain `HelloWorld.java`; the file will be called `hello.java`. The change logs that show up on `HelloWorld.java` before revision 63 will just be left over from the copy and won't represent real life. We will show how to trim down that extra output next.

4.1.6 Stopping the change logs from a copy

By now you have seen plenty of examples of the issues arising with change logs and the `copy` or `move` command. So how do we get around the issue of history in the output that came from the copy operation and not from new changes to the file? If you know that the object you are getting the logs for has been copied or moved, you can tell Subversion to get only the change logs of the object since the operation by giving the `log` command the `--stop-on-copy` switch:

```
$ svn log --stop-on-copy HelloWorld.Java
------------------------------------------------------------------------
r64 | jeff | 2004-04-17 23:26:27 -0400 (Sat, 17 Apr 2004) | 2 lines

Fixed the internal class name to match
the new file name
------------------------------------------------------------------------
r63 | jeff | 2004-04-17 22:54:55 -0400 (Sat, 17 Apr 2004) | 1 line

Made some adjustments
------------------------------------------------------------------------
```

When we ran the log this time, the revisions stopped at number 63. This is because the file was created at this revision with a copy or move, and the option told Subversion that this was as far back as you wanted to go. This is essentially the life of the object as the new entity; if you omit the option, you will get the life of the object in all its forms.

4.1.7 Formatting the change logs

In many cases, you will use the change logs for more than just having developers check from time to time. You may want to send the output to web sites for updates or reports. Subversion provides a couple of extra options that let you change the formatting of the output.

Putting the logs in XML. One of the valuable features in the `log` command is the ability to output the logs in XML instead of plain text, for example, if you want to put the change logs on a web site or in some type of report. You can do this by adding the `--xml` option to the `log` command. All of the other options in the command can be used in conjunction with this one; the same behavior will result, but the output will simply be in a different format.

```
$ svn log --xml hello.c
<?xml version="1.0" encoding="utf-8"?>
<log>
<logentry
   revision="30">
<author>alex</author>
<date>2004-04-03T19:10:57.416049Z</date>
<msg>Added function call to logger</msg>
</logentry>
<logentry
   revision="9">
<author>alex</author>
<date>2004-04-03T01:28:10.470588Z</date>
<msg>Added print statement to display
hello world message to standard out
```

```
</msg>
</logentry>
<logentry
   revision="1">
<author>jeff</author>
<date>2004-04-02T03:06:50.699754Z</date>
<msg>initial version</msg>
</logentry>
</log>
```

If you have any experience with XML, this output should be straightforward. The entire output is encapsulated in the `<log>` tag, and each individual entry is contained in `<logentry>`. If you use the `--xml` option without the `quiet` or `verbose` switch, every revision will contain the following tags: `<author>`, `<date>`, and `<msg>`.

XML with verbose. You just saw what the default output of the `log` command looks like in XML, but what about in verbose mode? If you are creating reports based on the change logs, the changed paths information gained from the `--verbose` switch is likely to be important. Let's see what additional tags are generated in this method. To keep the output to a minimum, we will also add the `--stop-on-copy` option.

```
$ svn log --xml --verbose --stop-on-copy HelloWorld.Java
<?xml version="1.0" encoding="utf-8"?>
<log>
<logentry
   revision="64">
<author>root</author>
<date>2004-04-18T03:26:27.451924Z</date>
<paths>
<path
   action="M">/trunk/source/Java/HelloWorld.Java</path>
</paths>
<msg>Fixed the internal class name to match
the new file name
</msg>
</logentry>
<logentry
   revision="63">
<author>root</author>
<date>2004-04-18T02:54:55.310200Z</date>
<paths>
<path
   action="D">/trunk/source/Java/hello.java</path>
<path
   copyfrom-path="/trunk/source/Java/hello.java"
   copyfrom-rev="60"
   action="A">/trunk/source/Java/HelloWorld.Java</path>
</paths>
```

```
<msg>Made some adjustments</msg>
</logentry>
</log>
```

All of the additional information from the verbose output is in the `<paths>` element, and each individual object changed in the commit will have a `<path>` element. In the case where the object was created from a Subversion copy or move, the `copyfrom-path` and `copyfrom-rev` identifiers will be included.

Incrementally adding change logs to a report. We have looked at examples of changing the output of the log files for reporting or custom information. Subversion provides one more feature to help with this. Let's say you want to create a report that shows all the change logs related to a particular set of bug fixes. You have a script that will search the logs and give you all the revision numbers you are looking for. If you run the `log` command on each of these and `cat` the output to one XML file, you will run into an aesthetic issue. Look at the following output and see if you can find the problem:

```
$ svn log --xml --revision 32 main.c > log.xml
$ svn log --xml --revision 61 main.c >> log.xml
$ svn log --xml --revision 62 main.c >> log.xml

$ cat log.xml
<?xml version="1.0" encoding="utf-8"?>
<log>
<logentry
   revision="32">
<author>alex</author>
<date>2004-04-03T19:42:25.402288Z</date>
<msg>did some stuff</msg>
</logentry>
</log>
<?xml version="1.0" encoding="utf-8"?>
<log>
<logentry
   revision="61">
<author>root</author>
<date>2004-04-17T02:18:04.892298Z</date>
<msg>added header information</msg>
</logentry>
</log>
<?xml version="1.0" encoding="utf-8"?>
<log>
<logentry
   revision="62">
<author>root</author>
<date>2004-04-17T23:29:36.531718Z</date>
<msg>Added return statement to main function</msg>
```

```
</logentry>
</log>
```

Each time you run the command, it creates a new root node `<log>` element. This will make each entry look like a whole new set of change logs, instead of multiple revisions in one object's change log. To get around this problem, you can use the `--incremental` option to tell Subversion you are concatenating individual revision change logs into one report:

```
$ svn log --xml --incremental --revision 32 main.c > log.xml
$ svn log --xml --incremental --revision 61 main.c >> log.xml
$ svn log --xml --incremental --revision 62 main.c >> log.xml

$ cat log.xml
<logentry
   revision="32">
<author>alex</author>
<date>2004-04-03T19:42:25.402288Z</date>
<msg>did some stuff</msg>
</logentry>
<logentry
   revision="61">
<author>root</author>
<date>2004-04-17T02:18:04.892298Z</date>
<msg>added header information</msg>
</logentry>
<logentry
   revision="62">
<author>root</author>
<date>2004-04-17T23:29:36.531718Z</date>
<msg>Added return statement to main function</msg>
</logentry>
```

In this output, the top level `<log>` is left out; it is up to you to provide it in the final report. This is much easier than trying to remove a bunch of tags from the middle of the report. Just adding the following lines will accomplish this task:

```
$ echo '<?xml version="1.0" encoding="utf-8"?>' > log.xml
$ echo '<log>' >> log.xml
$ svn log --xml --incremental --revision 32 main.c > log.xml
$ svn log --xml --incremental --revision 61 main.c >> log.xml
$ svn log --xml --incremental --revision 62 main.c >> log.xml
$ echo '</log>' >> log.xml
```

This set of commands will give you the individual logs for each of the revisions but will include the header information so you can view it as one report.

4.1.8 Changing log messages

Once a log message is committed, it is in the repository and there is no getting it out, right? Technically, the answer is no; you can change a log message, but this should be done only in the most extreme circumstances. Modifying the change logs for a revision is about the only thing in Subversion that you cannot revert. Once you make a change to the log message, the old one is gone for good. This is why you should not get into the habit of making changes to them. There will come a time when you enter a log message, and just after you save it and the commit starts you see a glaring typo. You best bet is to just leave it and move on, but there may be situations where the typo may cause confusion and must be fixed.

An example situation for change the log. Consider the following situation to see where you will need to go back and fix the log message. Assume you are using some bug-tracking tool that indicates open bugs with a number. There are two open issues; the first bug is number 23 and the second is number 32. You start working on number 23, get the issue fixed, and commit the change. But when you enter the log message, you invert the numbers, so the log message is wrong:

```
$ svn log hello.c
------------------------------------------------------------------------
r66 | jeff| 2004-04-18 12:04:42 -0400 (Sun, 18 Apr 2004) | 3 lines

Fixed Bug (32) - The function is supposed to print hello world.
This was never done so we added it to the hello() function.

------------------------------------------------------------------------
```

Even though you fixed bug 23, the log message says it was number 32. If someone is comparing the change logs with the open issues, there may be confusion as to which issue was resolved. In this situation it makes sense to go back and fix the log message.

Making changes to the log. Since there is no way to get back any changes to log messages, you can fix them only through the administrator client, with the `svnadmin setlog` command. This is the same interface we used to create a new repository. Remember, all `svnadmin` commands access the repository on the local filesystem, not through the URL. You must have rights to the filesystem and be on the local repository box to run this command.

The first step is to create the new log message and write it to a file. The `setlog` command will accept the log message only as the contents of a file. To ensure that you don't miss anything, the best way to do this is to cut and paste

the log message into a temporary file and then make any modifications. Let's put the message from the previous example into a file called `/tmp/rev66.log`:

```
Fixed Bug (23) - The function is supposed to print hello world.
This was never done so we added it to the hello() function.
```

Once you have the message written how you want it and have saved the file, you can run the `setlog` command. There are three required arguments; the first is the path of the repository on the local machine. Remember, the log messages are based on the repository, not individual files. So you need only the base directory, not the entire path of a file or directory in the repository. The second argument is the `--revision` switch; you are required to specify which version of the repository is changing. As an added precaution, you can specify only one revision, even though some older releases of Subversion say in the help that you can use a range. The third argument in the command is the file in which you have saved the new message. To fix our log message on revision 66, we will use the following command:

```
$ svnadmin setlog /repos/testrepo --revision 66 /tmp/rev66.log

$ svn log --revision 66 hello.c
------------------------------------------------------------------------
r66 | jeff | 2004-04-18 12:04:42 -0400 (Sun, 18 Apr 2004) | 2 lines

Fixed Bug (23) - The function is supposed to print hello world.
This was never done so we added it to the hello() function.
------------------------------------------------------------------------
```

Notice that only the message changed; the original author, date, and revision number were left intact. Even though we are looking at the change log for `hello.c`, any file that was changed in that revision will also have the new log message. If you tried to run this command and got an error, the next section will explain what happened.

Enabling Subversion to accept log changes. If this is a repository you have just created and you try to change a log message, you will likely receive the following error:

```
$ svnadmin setlog /repos/testrepo --revision 66 /tmp/hello.c.log
svn: Repository has not been enabled to accept revision propchanges;
ask the administrator to create a pre-revprop-change hook
```

In order for Subversion to be willing to change a log, you must have a hook or trigger script defined. Since the command does not save an old version of the log message or have any way to see that it changed, Subversion will force you to run a script before the operation. The assumption is that you will have the script save

the old message or that perhaps you will send the change via email so that other developers will know that it happened. All that is required, however, is a script with the name `pre-revprop-change`; it doesn't have to do anything. We will show how to create and set these hook scripts in chapter 8, but for now you can get by with simply creating an empty one. Subversion provides templates for the hooks that you can use. In the repository directory, there will a subdirectory called `hooks`. Here you will find the templates for the hook scripts. The following is a list of files you will see:

```
post-commit.tmpl          pre-commit.tmpl          start-commit.tmpl
post-revprop-change.tmpl  pre-revprop-change.tmpl
```

If you are on a UNIX system, just remove the `.tmpl` extension and add execute permissions to the file. On a Windows system, you will need to change the extension from `.tmpl` to `.exe`. Now that you have a script with the proper name, you can run the `svnadmin setlog` command, and Subversion will process the request. The only condition is that the script must execute and exit with a return code of `0` (which is the default of any script).

4.2 Getting a list of files

We have talked about and looked at some examples of Subversion acting as a filesystem. A requirement of any filesystem is the ability to list the objects inside it. Just like the UNIX `ls` or the Windows (DOS) `dir` command, Subversion provides a similar operation that will list the objects in the repository. This command, appropriately called `svn list`, should be run on a working copy and will list all of the versioned objects. Any non-versioned file or directory will be ignored.

4.2.1 A simple list

If you run the `svn list` command in a working copy with no arguments, it will list all of the versioned objects in the directory. If you give the command a URL, you will get a list of the objects in the directory without having a working copy. Here's an example of this in the `/source/c` directory of our repository:

```
$ pwd
/home/jeff/project/testrepo/source/c

$ svn list
file.c
gui/
hello.c
main.c
```

```
$ ls -1
file.c
gui
hello.c
main.c
project.xml
```

From the working directory, the `svn list` command gave us just the objects that are in the repository. Files are listed with no extra characters, but directories have a `/` at the end of their name. Now let's compare this to the operating system list that was run also. Notice that in the second list there is an extra file, `project.xml`. Since it was in the operating system list and not the Subversion list, you can figure out that it is a non-versioned file. By default, you will get the filenames that are in version control and nothing else. You can configure the `svn list` command to give you more information if you so desire.

4.2.2 Verbose

Like most Subversion commands, `svn list` has the ability to provide more information than the default output. When you include the `--verbose` option, you will see revision numbers and system information:

```
$ svn list --verbose
      9 jeff           2453 Apr 02 20:28 file.c
     47 jeff                Apr 06 21:23 gui/
     66 jeff             42 Apr 18 12:04 hello.c
     62 jeff             95 Apr 17 19:29 main.c

$ ls -l
total 12
-rw-r--r--    1 jeff     jeff         2453 Apr 16 20:00 file.c
drwxr-xr-x    3 jeff     jeff         4096 Apr 18 14:43 gui
-rw-r--r--    1 jeff     jeff           42 Apr 18 12:03 hello.c
-rw-r--r--    1 jeff     jeff           94 Apr 18 15:49 main.c
-rw-r--r--    1 jeff     jeff          192 Apr 18 14:55 project.xml
```

First, look at the output from the Subversion list; there are four additional fields in verbose mode. The first number is the revision of the last commit, followed by the author of the commit. The next number is the size of the file, and the final number is the date of the last revision. It is important to understand that this information is based on the repository, not the working copy. This is illustrated by the second listing, which shows the working copy. Look at the file `main.c` in the two listings. Using the operating system list, you can see that the file was modified on 4/18. The Subversion listing shows the date as 4/17. This means that the working copy has been modified and not yet committed. This is different than most

other Subversion commands; even though you are specifying a working copy, the information is based on the repository.

If you run this command with no target, it will use the current directory. You can specify the directory name as an argument if you want to run the command from another location. If you are interested in seeing the output for only one file, you can use that as the target also:

```
$ svn list --verbose hello.c
     66 jeff             42 Apr 18 12:04 hello.c
```

Since this goes against the repository, by default you will be getting the `HEAD` version. If you do not want the latest version, you will need to specifically tell Subversion this by using the `--revision` option.

4.2.3 Revisions

In the `svn list` command you can specify a revision number, which will start the list at that point in the repository. In the default method, the list will start at the `HEAD` and work backwards for each file until it finds a change. When you use `--revision` option, simply substitute the revision number for `HEAD`:

```
$ svn list --revision 50 --verbose
      9 jeff           2453 Apr 02 20:28 file.c
     47 jeff                Apr 06 21:23 gui/
     30 alex              17 Apr 03 14:10 hello.c
     40 jeff              80 Apr 06 20:07 main.c
     47 jeff                 Apr 06 21:23 testgui/
```

There are a couple of differences in the output using the revision number as opposed to using just `--verbose` with no options. First, let's look at what has not changed; `file.c` and `gui` are the same in both examples, which means that these two objects have not changed between revision 50 and the latest revision of the repository. The first difference is that a directory exists in this listing that is not in the default one, `testgui`. This means that between revision 50 and the `HEAD`, this directory was removed. Also, the files `main.c` and `hello.c` have different listings, which means they also have changed in that timeframe. If you were to check out our update to revision 50 of the repository, these are the versions of the files and directories you would get.

4.2.4 Recursive listing

You may have noticed something different about the `svn list` command compared to other Subversion commands; it is not recursive by default. This command is one of the few exceptions to the rule. If you would like to see a

recursive listing of the directory structure you are starting from, you can use the `--recursive` option:

```
$ svn list --recursive --verbose
      9 jeff           2453 Apr 02 20:28 file.c
     47 jeff                Apr 06 21:23 gui/
     47 jeff              4 Apr 06 21:23 gui/button.c
     44 jeff              0 Apr 06 20:59 gui/window.c
     66 jeff             42 Apr 18 12:04 hello.c
     62 jeff             95 Apr 17 19:29 main.c
```

In addition to the files and directories at the current level, the command also listed the files under the `gui` directory. It will keep going through all the levels from the starting point of the command.

4.2.5 List from URL

Let's say you are getting ready to do a large checkout of an old revision of the repository to look at some problems in your code. You think you need to get the code from revision 58 but are not sure. Instead of checking out revision 58 only to find out the file you need is not there, you can do a list to check this beforehand. For example, we copied the file `hello.java` to `HelloWorld.java` in revision 63, or at least that's what you think. You want to get the most recent version of the repository before we performed the copy, because you need the file `hello.c`. To see what the directory structure and files will look like before the checkout, run `svn list` from the URL:

```
$ svn list --revision 63 file:///repos/testrepo/source/java
HelloWorld.java
main.java

$ svn list --revision 62 file:///repos/testrepo/source/java
hello.java
main.java
```

You can quickly see the contents of the directory `java` at the two different revisions of the repository. This is much faster than doing a complete checkout just to see which files are in a particular revision. This is also a good mechanism for a sanity check when you are getting an old version of the repository. Subversion will let you go back as far you want as long as the target you are specifying exists.

4.2.6 Old or nonexistent files

If you try to do a listing too far back in time, you may run into a situation where the file or directory you are using as a target does not exist. When this happens,

Subversion will return an error alerting you of the situation. The following output shows what this message looks like:

```
$ svn list --revision 62 file:///repos/testrepo/source/Java/HelloWorld.java

svn: URL non-existent in that revision
```

Since `HelloWorld.java` didn't exist until revision 63, you cannot do a listing on it. The result is the same as a file not found error if you try to list a file that does not exist on a UNIX filesystem.

4.3 Comparing revisions

Anyone who has been in software development for any period of time has run into the situation like the following. You come into work and everyone is scrambling; there was a change to the application last night and now it doesn't work. Then inevitably some senior manager with no development experience will come up with the revelation, "Well, something must have changed." It is now your job to find out exactly what has been changed. Using the tools we have already explored, such as svn log and svn list, you should be able to figure out which files have changed, and from there you should be able to narrow down the list to the culprit source file. Once you have done this, you can use Subversion's `diff` command to see the differences between two revisions of a file.

4.3.1 The diff syntax

Before you can jump in and use the `diff` command, you must understand its output. This is similar in concept to the conflict file we saw in section 3.3.1, but the format is a little different. Take a look at the following example. In every diff there are two files, the old and the new. The old file is the original or baseline, and the new one is considered the modified version:

```
Index: main.java
===================================================================
--- main.java   (revision 69)
+++ main.java   (71)
@@ -1,11 +1,11 @@
-public static class main
+public class main
```

```
 {
   private int m_counter;
   private m_hello = new HelloWorld() ;

   public static void main(String args[])
   {
-    for (m_count = 0 ; m_count < 100 ; m_count ++)
+    for (m_count = 0 ; m_count < 10 ; m_count ++)
     {
       m_hello.printIt();
     }
```

The first four lines of the output are the legend. First, the index tells you which file you are examining, followed by a separator line. The next two lines tell you which character indicators represent which file. As you can see from the output, the - character on a line means it is from revision 69 of the file, while + indicates that the line is from revision 71. Any line without a character in front means it was unchanged between the two revisions. The diff will always include a few lines before and after the differences in the file to give you a better idea of the context in that area. Finally, there will be at least one line in every diff that starts and ends with `@@`. Figure 4.1 shows what the numbers on this line mean.

This summary will appear for each group of changes in a diff file. There can easily be more than one section of the file that is different. If these changes are not within a couple of lines of each other, an additional summary and diff section will be included.

Multiple change sets. Each set of differences in the two targets being compared will have its own section in the diff output. Not only will each change group have the contents displayed, there will also be an additional summary header. When you see a summary, you will know that you are starting at another point in the file, so the lines before and after it are not sequential in the real file:

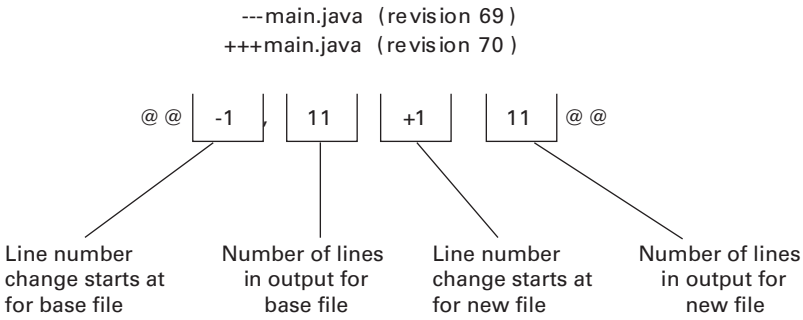

Figure 4.1 Header and legend in the diff syntax

```
Index: LdapBind.java
===================================================================
--- LdapBind.java       (revision 9843)
+++ LdapBind.java       (working copy)
@@ -256,7 +256,7 @@
   }
   catch ( IOException ioe )
   {
-    System.out.println ("IO exception connecting to the server" +
+    System.out.println ("IO exception connecting to open the socket" +
      ioe.toString() ) ;
     System.exit( -1 ) ;
   }
@@ -286,7 +286,7 @@
       + l_opts.getBindDn()
       + " failed for the following reasons provided by the server"
       + new String( l_response.getErrorMessage() ) ) ;
-    System.exit( -1 );
+    System.exit( l_response.getErrorNum() );
   }
   catch ( IOException ioe )
   {
```

In this example, there are two changes in the file, and each new set of changes starts with a new legend with the `@@` indicator. The diff gives only three lines on both sides of the change. Since the changes are farther apart in the file, the output leaves out all the lines in between.

4.3.2 Running the diff command on your local copy

You will frequently use the `svn diff` command to see what changes you have made to the working copy. Here, you are comparing the local version to the `BASE`. Remember that this is the revision of the file you checked out, and it is not necessarily the latest version in the repository:

```
$ svn diff main.java
Index: main.java
===================================================================
--- main.java   (revision 69)
+++ main.java   (working copy)
@@ -1,11 +1,11 @@
-public static class main
+public class main
 {
   private int m_counter;
   private m_hello = new HelloWorld() ;

   public static void main(String args[])
   {
```

```
-     for (m_count = 0 ; m_count < 100 ; m_count ++)
+     for (m_count = 0 ; m_count < 10 ; m_count ++)
      {
        m_hello.printIt();
      }
```

All we did was pass the filename of the working copy into the diff. This automatically uses the `BASE` as the old revision and the locally modified copy as the new one. If you look at the legend, you will see that the old target has a revision number, 69, which is the version you had checked out. Instead of showing a revision number, the new file just says "working copy." Even though we have been using the working copy in all our examples so far, you are not limited to it.

4.3.3 Specifying revisions

Think of any situation where someone would ask "What changed between revision X and Y of the file?" You can use the `--revision` option to compare specific revisions of a file. Let's try this with revisions 64 and 70 of the file `HelloWorld.java`:

```
$ svn diff --revision 64:70 HelloWorld.java
Index: HelloWorld.java
===================================================================
--- HelloWorld.java     (revision 64)
+++ HelloWorld.java     (revision 70)
```

All we did differently in this command was to tell Subversion which versions we want to compare, instead of picking the local copy and the `BASE`. The revision number does not have to be at a point where the file changed; you can use any version of the repository. For example, look at the change logs for `HelloWorld.java`:

```
$ svn log --stop-on-copy --quiet HelloWorld.Java
------------------------------------------------------------------------
r69 | root | 2004-04-18 20:18:42 -0400 (Sun, 18 Apr 2004)
------------------------------------------------------------------------
r68 | root | 2004-04-18 20:13:18 -0400 (Sun, 18 Apr 2004)
------------------------------------------------------------------------
r64 | root | 2004-04-17 23:26:27 -0400 (Sat, 17 Apr 2004)
------------------------------------------------------------------------
r63 | root | 2004-04-17 22:54:55 -0400 (Sat, 17 Apr 2004)
------------------------------------------------------------------------
```

There is no revision 70 of the file, yet the command allows you to use that number as one of the versions in the compare. Back again to the atomic commits—this can happen because revision 70 is just an identical copy of revision 69. If we had

used 65 as the original or old version, the command would have yielded the same results since there is no change for 65, so it is identical to 64. Subversion will always work backwards to find the last revision that changed from whatever starting point you give it.

Using keywords. You will find in many situations where the `diff` command is required that the keywords for revision numbers come in handy. The keywords are defined in table 2.3. Using these will let you run the `diff` command on key revisions of the repository, without having to look up numbers. These key points include the latest revision in the repository, your checked-out version, and the previous version. We will key on two relationships:

```
$ svn diff --revision PREV:BASE main.c

$ svn diff --revision BASE:HEAD main.c
```

The first one will compare the version you currently have checked out with the previsionrevision in the repository. This is useful if you run into a situation where you did an update and now the code is giving you problems. You can run the `diff` command to see what changed between the new version in your working copy and what was there previously (assuming there was only one new version between updates). The second relationship will show you the difference between the version you have checked out and the latest revision in the repository. Remember, this does not include any modifications you have made to the working copy; it is the version checked out. You can run this command if someone else has committed a file to the repository and you want to see what has changed before making your modifications. Also, you can mix revision numbers and keywords, so something like `--revision 10:HEAD` is completely valid. These are just commonly used relationships; any combination of keywords can be used to specify the old and new files for the compare. Don't take the terms *old* and *new* to heart when talking about the `diff` command; as you will see, they may not mean exactly what you first think.

Switching the order. When we look at the two files in the diff, there is an old one and a new one. While this is the terminology used in Subversion, it may not be the best description. The old and new files do not have to be based on time; it is just a matter of how you specify them in the command. The following example will reverse the order of the revision numbers we have seen so far:

```
$ svn diff --revision 10:9 main.c
Index: main.c
===================================================================
--- main.c      (revision 10)
+++ main.c      (revision 9)
@@ -6,7 +6,7 @@

   int index;

-  for (index=0;index<5;index++)
+  for (index=0;index<10;index++)
   {
     hello();
   }
```

If you compare this to the output of other `diff` commands we have seen, there is not a notable difference. The only difference is that the higher, or more recent, revision is first and is indicated by the - character. This is merely a matter of preference as to which file is shown first; there is no difference in the operation of the command.

In all the `diff` commands we have examined, we have used the working copy path even though the operation was hitting the repository in many cases. This is done by looking up the URL in the `.svn` directory in which the file resides. While using a filename instead of a URL can be a convenience, you are not forced to use the working copy as a target.

4.3.4 Running diff directly from the repository

Like most Subversion commands, the `diff` can be run directly on the repository, instead of from a working copy. This is useful if you are running the command from a script or need to quickly check the change between two files and do not want to go through the process of checking out the repository. All you need to do in this method is substitute the path of the file with the URL of the file in the repository. When you are using a URL, you must specify the revision numbers with the `--revision` option:

```
$ svn diff --revision 10:HEAD file:///repos/testrepo/source/c/main.c
Index: main.c
===================================================================
--- main.c      (revision 10)
+++ main.c      (revision 11)
@@ -11,5 +11,6 @@
     hello();
   }
   printf("Done printing hello world");
-  return 1;
```

```
+  return 0;
+
 }
```

So all we did differently here was add the URL to the file. This will allow you to run the `diff` command from anywhere, even if you have not checked out a working copy. You must specify the revisions when using the URL because Subversion does not know what to use as the default. Being able to specify the URL and revision numbers gives you the opportunity to customize the `diff`, but it is not limited to that.

Multiple URLs. What if you run into a situation where you have made a branch of a file and you want to compare that with the file on the main branch? Using URLs, you can compare two different objects in the repository. These files can be in different subdirectories, but they must be in the same repository. Also, if you use this notation, you are not forced to enter the revision numbers. In this case, Subversion will default to the latest revision of each file:

```
$ svn diff file:///repos/testrepo/source/c/hello.c \
 file:///repos/testrepo/file_io_branch/source/c/hello.c

Index: hello.c
===================================================================
--- hello.c     (.../source/c/hello.c)    (revision 73)
+++ hello.c     (.../file_io_branch/source/c/hello.c)   (revision 73)
...
```

Since we did not specify the revision numbers, Subversion used the latest version of the repository for each file. Instead of distinguishing by revision number to tell which is the old and which is the new copy, we used the file paths. You can see this in the `Addition` field in the legend output. Just like with specifying the revision numbers, the URL of the file you put first will be considered the old file and the second will be the new file. If you do not wish to use the latest version of the file, you can add the `--revision` option. The first revision number will be associated with the first file and the second revision number with the second file. Let's look at the same example, except that this time we'll compare revision 72 of the branch with revision 71 of the main:

```
$ svn diff --revision 71:72 file:///repos/testrepo/source/c/hello.c \
 file:///repos/testrepo/file_io_branch/source/c/hello.c

Index: hello.c
===================================================================
--- hello.c     (revision 71)
+++ hello.c     (revision 72)
...
```

Even though we used the URLs, we are back to distinguishing by version number and nothing else in the legend. This forces us to go back to the command and verify which revision number matched up with the appropriate file path. The only time the file path will show up is if the revision numbers are the same. So in this example, if we used `--revision` 72:72 in the command line, the paths would have shown up in addition to the revision numbers. Some will debate that any time two URLs are specified, the paths should show up, even if the revision numbers are different.

Revision number on the URL. Subversion gives you an additional format for accessing the revision numbers in the `diff` command. Instead of using `--revision`, you can use an `@` symbol at the end of the filename followed by the revision number:

```
$ svn diff file:///repos/testrepo/source/c/hello.c@71 \
 file:///repos/testrepo/file_io_branch/source/c/hello.c@72

Index: hello.c
===================================================================
--- hello.c     (revision 71)
+++ hello.c     (revision 72)
...
```

The results are identical in this method. It is simply a different notation than using `--revision`. This applies only to `diff` and will not work with other Subversion commands.

Limited keywords. When using the URL to specify the object to run the `diff` command on, you must use the `--revision` option. You might think that it would follow that you should be able to use keywords with this command as well. This is true, but only for a single keyword, which is `HEAD`. This is because `HEAD` is the only one that is based on the status of the object in the repository. The other keywords, `BASE`, `PREV`, and `COMMITTED`, are all based on the working copy. If you try to use these for revision numbers when you access the file through a URL, Subversion will give you an error:

```
$ svn diff --revision BASE:PREV file:///opt/repos/testrepo
svn: A path under version control is needed for this operation
```

When operating on a repository, you cannot use keywords that reference a revision based on a working copy.

4.3.5 Using other diff programs

If you are a longtime UNIX user, you have likely used a slightly different version of `diff` in the past. If you have a deep emotional attachment to this `diff` command,

Subversion lets you substitute this for its command. Basically, Subversion will go to the repository and get the two versions of the file you asked for and pipe them into the `diff` command of your choice. Any executable program that accepts the GNU `diff` options will work. So, for example, the following Subversion command will use the standard `diff` on a Linux machine:

```
$ svn diff --diff-cmd /usr/bin/diff  main.c
Index: main.c
===================================================================
--- main.c      (revision 73)
+++ main.c      (working copy)
@@ -6,5 +6,5 @@
 }
 void func2()
 {
-  // Do some stuff here
+  hello();
 }
```

Without any options, there appears to be no difference, but in reality the Linux `diff` command is the one that created the output. Subversion will pass in its own set of arguments that tell the command to use the unified output format. This makes the output look the same as when Subversion does the processing. Let's take a look at how the operating system command is being called in the previous example:

```
diff -u -L main.c (revision 73) -L main.c (working copy) \
.svn/text-base/main.c.svn-base main.c
```

If you were to run this command from the same current directory as the Subversion command, the output would look identical. So where is the advantage to this if the commands do the same thing? The answer lies in an additional set of options you can add to the command line. The `svn diff` command will also take an option called `--extensions`. This option requires a string as an argument, which consists of additional options that get passed to the command in the `--diff-cmd` argument. For example, the UNIX `diff` command takes the `-C` option to set the number of lines before and after the change instead of the default three lines. If we want to pass this option along to the command, we can do so:

```
$ svn diff --diff-cmd diff --extensions "-C 1 -t" main.c
Index: main.c
===================================================================
*** main.c      (revision 73)
--- main.c      (working copy)
***************
*** 8,10 ****
  {
```

```
!   // Do some stuff here
  }
--- 8,10 ----
  {
!   hello();
  }
```

Looking at the syntax above, we passed two arguments into the UNIX `diff` command; the `-C` option that we talked about was passed first. The second was `-t`, which turns tabs into spaces. Now look at the output from the command. There is only one line before and after the changed line, so we were able to use the power of the operating system `diff`. You can access all of the options the operating system `diff` provides, plus all the options from the Subversion command. In this method, all Subversion does is find the files and call the command you specify.

4.3.6 Running diff on directories

So far we have seen the `diff` command run only on individual files. But what about directories? Running the command on a directory is the same as running a `diff` on all the individual files in that directory. Since by default, the command will give you output only if there is a difference, any files that are unchanged will not have any output. And you can probably guess what's coming next—yes, the operation is recursive. The `diff` will find all the files under your starting subdirectory that have differences and show the changes in the output. Let's look at an example of a `diff` that will list all the differences in the working copy and the `BASE` of the `source/c` directory:

```
$ svn diff .
Index: hello.c
===================================================================
--- hello.c     (revision 74)
+++ hello.c     (working copy)
@@ -1,4 +1,4 @@
 void main()
 {
-  printf ("Hello World");
+  printf ("Hello World\n");
 }
Index: gui/button.c
===================================================================
--- gui/button.c        (revision 74)
+++ gui/button.c        (working copy)
@@ -1,4 +1,4 @@
 void create_button()
```

First file in output

Second file in output

```
 {
-  // Create the button here
+  XButton button = XButtonCreate();
 }
```

Second file in output

```
Index: main.c
===================================================================
--- main.c      (revision 74)
+++ main.c      (working copy)
@@ -6,5 +6,5 @@
 }
 void func2()
 {
-  // Do some stuff here
+  hello();
 }
```

Third file in output

The only difference in using a directory as the target is that multiple files are listed. Also, take a look at the second file. Since it is not in the same level of the directory as where you started, the relative path to your start directory is given.

Non-recursion. Since the default behavior of the `diff` command is to recursively parse all the subdirectories, you can use the `--non-recursive` option to prevent this. You guessed it; this will give you only the files at the same level of the start directory.

```
$ svn diff --non-recursive .
Index: hello.c
===================================================================
--- hello.c     (revision 74)
+++ hello.c     (working copy)
@@ -1,4 +1,4 @@
 void main()
 {
-  printf ("Hello World");
+  printf ("Hello World\n");
 }
Index: main.c
===================================================================
--- main.c      (revision 74)
+++ main.c      (working copy)
@@ -6,5 +6,5 @@
 }
 void func2()
 {
-  // Do some stuff here
+  hello();
 }
```

When we run the command with this switch, the file `/gui/button.c` does not show up in the output since it is not at the same level.

4.4 Looking at file contents

We just showed how to look at the differences of two revisions of the same file, but sometimes you may want to look at just the contents of one revision. If you want to see the current version in your working copy, you can use your favorite editor or operating system command to do this. What if you want to look at the contents of an older version of the file? You could check out or update the working copy to do this, but that may be overkill for what you are trying to accomplish. To view just the contents of a particular revision of a file, you can use the `svn cat` command.

4.4.1 Running the cat command

The Subversion `cat` is really no different than the UNIX version; it will just dump the contents of the file to standard output. The difference is that Subversion will get the file from the repository and allow you select the specific revision of the file you want. To run the command, just pass in the file in your working copy that you want to view:

```
$ svn cat main.c
//
int main()
{
 // do some stuff here
  return 0;
}
void func2()
{
  hello();
}
```

Without any parameters, the command will use the `HEAD` version of the file. Remember, this is the most recent copy of the file in the repository. If you want the local copy, you can just use the operating system to do this. Having the `cat` use the `HEAD` version gives you a quick way to see what the current version of the file looks like. Since the command looks at content, it can be run only on files, not directories. Of course, you are not restricted to getting the contents of the `HEAD`; you can get any revision in the repository.

4.4.2 Revision numbers

The real power of the command is apparent when you add the `--revision` option to it. This gives you a way to go back to any point in the repository and see what the file looked like without having to do an update or checkout. All of the standard revision features apply, such as keyword, dates, and numbers. Let's look at a basic example of the `cat` command using the `--revision` option:

```
$ svn cat --revision 74 main.c
//
int main()
{
 // do some stuff here
  return 0;
}
void func2()
{
  // Do some stuff here
}
```

This is how the file `main.c` looked at revision 74 of the repository. The only constraint is that the file must exist in the revision of the repository you are asking for. If the file did not exist yet, or if it was created by a `move` or `copy` command after that revision, Subversion will complain and tell you that the object is not in the repository. If you need to get the contents of a file before it was renamed, you will have to do this through the URL, not the working copy. Even though the command is hitting the repository, we are still using the working copy path. This is just like the `svn diff` command you saw earlier in the chapter. Just like with the `diff`, you do not have to have a working copy to run a `cat`.

4.4.3 URLs

Since the working copy directory structure is a snapshot of the repository at a certain point, it may not contain the files you are looking for. For example, in revision 63 of the repository we renamed the file `hello.java` to `HelloWorld.Java`. If we try to `cat` the file at revision 50, we will get an error:

```
$ svn cat --revision 50 HelloWorld.Java
svn: File not found: revision '50', path'/source/Java/HelloWorld.java'
```

Since we do not want to check out revision 50 of the repository, we can access the file through the URL with its old name and revision number. First, run the `log` command with the `--verbose` option to see what the file was named at that revision:

```
$ svn log --verbose HelloWorld.java
------------------------------------------------------------------------
r69 | root | 2004-04-18 20:18:42 -0400 (Sun, 18 Apr 2004) | 1 line
Changed paths:
   M /source/Java/HelloWorld.java
   M /source/Java/main.java
added call to print hello world 100 times
------------------------------------------------------------------------
r63 | root | 2004-04-17 22:54:55 -0400 (Sat, 17 Apr 2004) | 1 line
Changed paths:
   A /source/Java/HelloWorld.java (from /source/Java/hello.java:60)
   D /source/Java/hello.java

Made some adjustments
------------------------------------------------------------------------
r11 | jeff | 2004-04-02 20:56:30 -0500 (Fri, 02 Apr 2004) | 1 line
Changed paths:
   A /source/Java/hello.java

Initial version of hello.java
------------------------------------------------------------------------
```

By looking at the logs in revision 63, we see that the source of `HelloWorld.java` is `hello.java`. So we can use this name with the `cat` command and revision 50 to get the contents:

```
$ svn cat --revision 50 file:///repos/testrepo/source/Java/hello.java

public class hello
{
  public void printIt()  {
    System.out.println("Hello World");
  }
}
```

Now this gives us the contents of the object at revision 50 of the repository. You can use the URL method from anywhere, so you do not need to have a working copy to see the contents of any file at any point .

4.5 Annotating a file

Have you ever looked a piece or a line of code and said, "Who did that?" Think about how you would go about finding out this information. You could start with the `HEAD` of the file and keep running the `diff` command on previous revisions, but that would get a little tedious. Subversion has a very easy way to see where every line of code in a file came from, known as *annotation*.

4.5.1 Running the annotate command

The `annotate` command can be run on a working copy file. You pass the file into the command, and it will display the contents of the file. In addition, `annotate` will tell you the author and revision number of the last change for each line. If the file is large and you have a lot of changes, it may take a few seconds to generate the output:

```
$ .  svn ann main.java
    70       alex public class main
     2       jeff {
    69       jeff   private int m_counter;
    69       jeff   private m_hello = new HelloWorld() ;
    69       jeff
    69       jeff   public static void main(String args[])
    65       alex   {
    70       alex     for (m_count = 0 ; m_count < 10 ; m_count ++)
    69       jeff     {
    69       jeff       m_hello.printIt();
    69       jeff     }
    69       jeff   }
     5       jeff }
```

The output is pretty self-explanatory. The first field in the output is the number of the last revision when that line was changed. The second field is the user who made the change in that revision. So in the first line in the previous example, the user `alex` was the last person to modify that line, and it was in revision 70.

This command is great for finding out who is responsible for a change, so you know whom to contact for additional information. Also, you can look at the change logs for that revision to see the author's reasons for the change. Since the revision number is right on the line, you don't have to search all the logs to find the one you are looking for.

4.5.2 Annotating specific revisions

Just like with the `cat` command, there are numerous reasons why you may want to annotate a previous revision of a file. It will not make sense to check out a new working copy just see the contents and annotated information. If the file exists in the working copy, you can just use the path and add the `--revision` option:

```
$ svn ann --revision 68 main.java
     3       jeff public static class main
     2       jeff {
     7       jeff   public static void main()
    65       alex   {
    65       alex   }
     5       jeff }
```

Since we used revision 68 as the starting point, we will see only numbers equal to or less than that revision. Subversion is basically stepping through each revision of the repository until the most recent change to a line is found that is less than the starting point. So looking at the output above, the second line has not been changed since revision 2 of the repository. In addition to the revision numbers, you can use any keyword or date format to access a revision. Since the command operates on only one revision of the file, you cannot specify a range of revisions.

4.5.3 Directly accessing the repository

You may have noticed that the `annotate` command is almost a mirror image of the `cat` command but with additional information. And just like with the `cat` command, you can access a file in the repository without having a working copy checked out. This is helpful for troubleshooting quickly or, as we saw with the `cat` command, when the file no longer exists in the working copy. Simply substitute the path of the file with the URL to do this:

```
$ svn ann --revision 74  file:///repos/testrepo/trunk/source/c/main.c
    73       alex int main()
     9       jeff {
    40       jeff  // do some stuff here
    62       jeff   return 0;
    29       alex }
    31       alex void func2()
    31       alex {
    32       alex   // Do some stuff here
    32       alex }
```

By now, this should make perfect sense to you. We are getting a specific revision, directly from the repository, so we do not need a working copy.

4.5.4 Be cautious when using annotate

It may seem like a good idea to use this command to find out who made a change for accountability or even for measuring the amount of work each developer is doing, but you must be careful. All this command shows is that some modification was made to the line by the person specified in the revision. Let's look at an example of how the `annotate` command can point you in the wrong direction. Say you are doing a test, and the source `hello.c` seems to be giving you some problems. You have not had anything to do with the file, so first you will have to view the source code. When you run a `cat` to see what the latest version looks like, you get the following output:

```
$ svn cat hello.c
#include <stdio.h>
#include <strings.h>
void main()
{
  int l_val ;
  char l_output[10] ;

  sprintf( l_output, "This is a hello world message" ) ;
  while ( l_val > 5 )
  {
    printf ( "%s",output ) ;
  }
}
```

If you are familiar with C, you will notice some obvious errors. Let's look at two problems: first the variable `l_val` is used before it is initialized. The second problem is that we are loading more than 10 characters in the string `l_output`, even though we have defined it as 10 for the maximum size. So if you saw this, the first thing you want to do is run an `annotate` command to find out who did this and give him a lesson in C. Take a look at the `annotate` output:

```
$ svn ann hello.c
    83       adam #include <stdio.h>
    83       adam #include <strings.h>
    83       adam
     9       jeff void main()
     9       jeff {
    84       alex   int l_val ;
    84       alex   char l_output[10] ;
    83       adam
    84       alex   sprintf( l_output, "This is a hello world message" ) ;
    84       alex   while ( l_val > 5 )
    83       adam   {
    84       alex     printf ( "%s",output ) ;
    83       adam   }
    30       alex }
```

Now that you have your culprit, your next step is head over to Alex's desk and take away his commit privileges, right? Even though this looks clear-cut, you really don't know for sure what is going on. Before jumping to conclusions, look at the change logs. We know the revision number, 84, so let's see what happened:

```
$ svn log --revision 84 hello.c
------------------------------------------------------------------------
r84 | alex | 2004-04-20 22:33:20 -0400 (Tue, 20 Apr 2004) | 1 line

Fixed some checkstyle errors
------------------------------------------------------------------------
```

Well, it appears that perhaps all Alex did was to fix some style errors, so maybe it wasn't him. Let's run the `annotate` on the previous revision of the repository and see what happens. If the lines do not exist, then we know that he added them, and we have a resolution:

```
$ svn ann --revision 83 hello.c
    83       adam #include <stdio.h>
    83       adam #include <strings.h>
    83       adam
     9       jeff void main()
     9       jeff {
    83       adam   int val;
    83       adam   char output[10];
    83       adam
    83       adam   sprintf(output, "This is a hello world message");
    83       adam   while (val > 5)
    83       adam   {
    83       adam     printf ("%s",output);
    83       adam   }
    30       alex }
```

Now looking at this output we have an entirely new person to look at. It turns out that the user who created the problem is Adam. The only thing Alex did was to make the variables match the organization's style format. So the moral of the story is to use `annotate` for a guide, not as the definitive answer.

4.6 Summary

You have just seen the "guts" of Subversion. History and revision information are the basis of any version control system. While this might not be the most glamorous part of Subversion, it is critical for the day-to-day operations of your development. By mastering the tools you've seen in this chapter, you will start gaining value from version control. Even though we have mentioned it many times, it is important to remember that Subversion's change logs are based on commits to the repository, not individual files. You saw this again when we examined change logs at the directory level. In order to use all the different query methods to get change log information, you must have this concept straight in your mind.

Hopefully one of the advantages you are starting to see with Subversion is how well it fits into software development. It was created by a group of people who develop as a profession and in their free time. Benefiting from it is just a matter of using its conveniences throughout your system. Conveniences such as the XML formatting in the change log output will allow you to create custom reports and information quickly so you can focus on development.

Chapter 3 explained in more detail how Subversion is a type of filesystem with the `add`, `move`, and `delete` commands. That explanation was continued in this chapter, with the ability to look at objects in a list format. We also saw how to use the `cat` command to view the contents of a file. Keeping in line with the definition, Subversion provides the functionality of a regular filesystem, with the addition of history. Start getting into the habit of using the Subversion commands, such as `list`, instead of the operating system commands. This will start to become second nature to you, and the tool will seem transparent. It is no different than when you first starting using your favorite operating system. In the beginning, sometimes its features seemed like impediments. As you got used to using them, you wondered how you ever lived without them.

Now it is time to stop and take a deep breath. We started out at with some high-level theories about version control and just finished a detailed, technical discussion of using Subversion. At this point, you could join a development team using Subversion and be able to jump right in. Moving forward, we will start exploring how Subversion implements more complex issues such as development lifecycle processes. We will tackle such topics as properties and branches.

5 Branches and tags

In this chapter

- Branching overview
- Branching strategies
- Creating copies
- Merging files
- Tags

Now that we have explored the basic operations of Subversion, it is time to start applying some process management. Branches and tags are the first step to true configuration management. As with any process, there is no right or wrong way to use Subversion in your software development lifecycle. However, we will make some suggestions about how you can apply these tools to your projects.

5.1 Branching overview

Consider the following situation. You are a developer in a software company that provides an off-the-shelf application to your clients. The salespeople are starting to ask whether certain potential large clients can have a degree of customization in their application. This new client-specific code would be based on the current "core" application. So being the guru you now are in Subversion, you suggest using the `svn copy` command to replicate the code so that now you can make the changes to individual client code. This works great, but as time moves on and there are more and more client copies, you start to notice a problem. Since the clients came from the same base, each time a bug fix is made or an enhancement is required, it has to be done in each of the copies. Also, there may be features in the client code that you want to put back into the core development path for other clients. This is where a branch comes in—it allows you to have different development paths from the same code base.

5.1.1 What is a branch?

We have briefly discussed the idea of branches, but not in any detail up to this point. A branch is a critical aspect of any version control system. It allows different development paths to be taken from the same code base. If you think of the repository as a tree, then the main development path is the trunk. This is where the base code resides, and everything should stem from it. Sticking with the analogy of the tree, the secondary development paths are branches on the tree (hence the name). Just as with any tree, branches can have other branches on them, although that configuration starts to get a little tricky to manage. Figure 5.1 gives you an idea of what a branch looks like conceptually on a file.

In this example, the file `hello.c` was copied into a new directory and committed at revision 18 of the repository. At this point, the copies are two separate files with different development paths. If you make a change to both files on the same commit, they will be part of the same revision number. Once you create the branch, they are treated like two separate files. In our example, the client branch will remain separate from the main development path. When a change is required

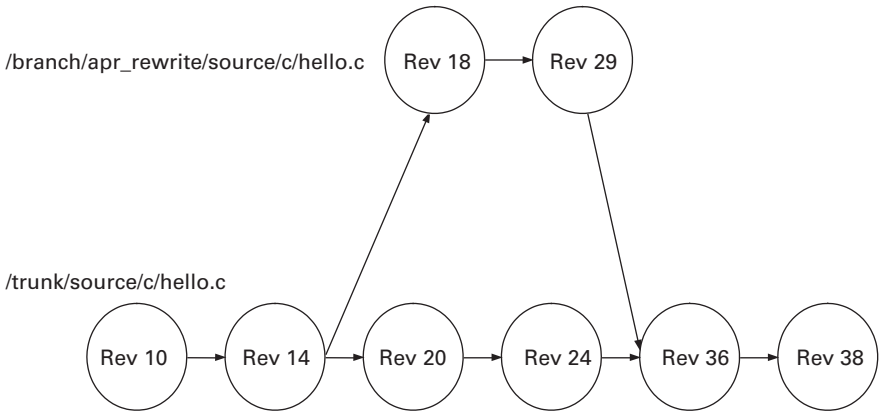

Figure 5.1 A branch in the version tree

for `hello.c` that affects the main development path and the branches, the developers will either manually edit each one or use the `merge` command. The `merge` command acts like the `diff` command we explored in chapter 4, except that it saves the changes to the local copy. We will discuss this concept in more detail later in this chapter. As we stated previously, with any tree you are not limited to one branch—or even where the branch gets initiated. Take a look at figure 5.2. Here we have branches that start from different points, plus branches on branches. While there are no limits or constraints on branching, you must make sure that you keep it manageable. If you do not understand what your structure looks like, it will be difficult to maintain your development paths.

In Subversion, a branch is just a copy of the file in another directory. In fact, Subversion does not actually have any formal branching commands or syntax. So it is up to you to understand which copies are branches and how to keep them in sync. If you use a standard naming scheme, you will be able to easily identify your branches in the repository. A little later on, we will talk about the standard that most Subversion users are adopting to lay out the repository.

The entire repo, file, or directory—what to branch? Just as with the checkout, you can branch at any point in the repository. Therefore, you can include as much as the entire directory tree or as little as one file; it all depends on what you are working on. When in doubt, make the branch bigger than you think you'll need it to be. There is no harm in having extra files. The worst case would be that you don't use them. When it is time to merge, they will simply be ignored since they haven't been changed.

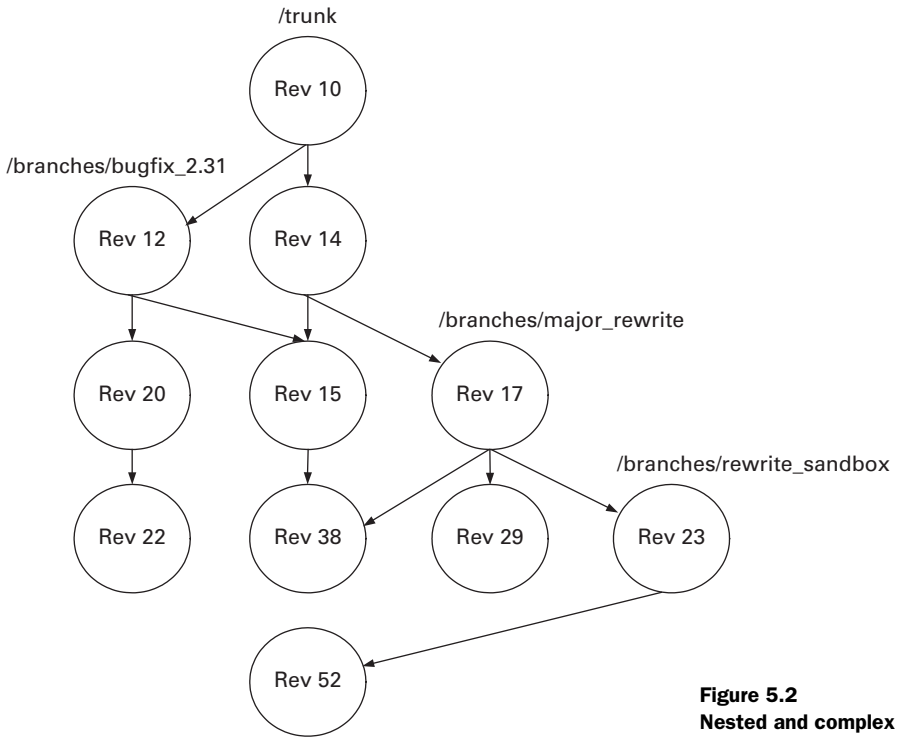

Figure 5.2
Nested and complex branching

5.1.2 When should you use branches?

We already talked about one reason to branch in the beginning of the chapter. If you need to take a new development path on your exiting code base, a branch is a good solution. This is just one use for branches. Here are a few more.

New versions of the code. In the previous example, we needed to customize the core code for certain clients. Because the clients are based on the core, they will need to accept bug fixes that were "inherited" from the copy. Presumably, new enhancements to the core will eventually be propagated to the specific client code as well. By keeping the code as a branch in the same repository, you can easily merge the changes to the files in the main path and the branch. At the same time, the paths will remain separate so that a big change in one area will not affect the other.

The ability to move changes between the trunk and branches does not have to unidirectional. For example, you may have developed a feature in one of the client branches that you determine should be part of the core code. In this case, you

will merge the changes from the branch to the trunk. This can be done as often or as infrequently as makes sense. Just remember that the longer you go between merging changes, the more effort will be required. We will discuss branching strategies in the next section.

Refactoring code. Another situation where you may want to use branches is with large changes to the code base. For example, your development team has decided to move away from using compiler directives in the code and to utilize the Apache Portable Runtime. This is a major change to the code that will take a few months to implement. While the files are being modified, they will be in interim states and likely will not compile or function properly. Since this is the case, it will be impossible to perform small bug fixes in the meantime. Let's say Adam will be doing this development; one option would be for him to just do the development in his working copy and not check in the code until the entire project is complete. This has a couple of potential problems, the first of which is check pointing the files. Any saves of the code will have to be done manually, and there will be no history or automatic versioning. Another problem with this scenario is that Adam will be working alone while the development is going on. Unless someone looks at his working copy (which is difficult if it is sitting on his desktop), other members of the team cannot see what he is doing. The change will come as one big bang, which will make the merge more difficult. Also, issues that would have been caught early in the development process will not show up until much later.

Branches can solve this problem by giving you a place in the repository to work on a copy of the file that is independent of the trunk. As long as the bug fixes and main development stay on the trunk, changes to the branch will not disturb them. As changes are made to the trunk, Adam can merge them into the branch so that they do not have to occur all at once. Also, other team members can monitor the new code and possibly take incremental changes from the branch to the trunk so that integration is more manageable. Branches do not have to be permanent, so once the new code is finalized and completely merged back into the trunk, they can be removed. Of course, like any file or directory, a branch is never truly gone.

Sandboxes. Along the lines of major changes to the code, a branch is a good place to put a sandbox to test new features or designs in the code. Again, you can start with the current code base and develop in an area that does not affect the mainline development path. You do, however, get the advantages of having your code stored in the repository. You can still merge changes from the trunk into the sandbox so that you are playing with the most up-to-date code. Finally, should you decide to use some or all of the changes in the sandbox, you can merge the modification directly

from the branch into the trunk. As you will see when we discuss the `merge` command, you can specify a subset of changes in the branch to merge—it does not have to be the culmination of all the changes made in the branch.

5.1.3 Branching strategies

The amount of branching Subversion will support is limited only by your creativity. Remember, to Subversion a branch is just a copy of the code used to create a new object. At that point, the trunk and the branch are distinct entities; the user is really the one who knows about the relationship. You can have branches off branches, keeping them forever or for just one revision; the list goes on and on. Your best bet is to find the strategy that matches your development process, team size, and level of expertise. If your team is made up of inexperienced developers, you may want to keep the branching simple. For people who are not familiar with this concept, branching can be a little overwhelming. You want to make sure that the version control software does not become an impediment. An entire book could be written on the different strategies and how to use them. We will very briefly talk about just a few of them so you can get an idea of some of the options.

All development is done in a branch. This is one of the more extreme forms of branching, but it is popular in many proprietary version control systems that have built-in integration with defect-tracking tools. In this model, each time a new task gets assigned to a developer, that person creates a branch to work on the assignment. When the task is complete, it gets merged back into the trunk as a new minor release, and the branch is removed. The advantage of this method is that the trunk stays in a production state because the changes get merged only from branches that are complete and tested. There is actually no development being done on the trunk, only merges from branches. This method also segregates the code nicely, so it is easy to tell how many tasks are being worked on and what each branch is for.

The downside is the overhead that goes along with this type of model. This extra effort is not from the tool itself but from the process of creating branches and merging changes. The developers on the team will be creating branches and merging changes all the time, even for small changes. If you are not familiar with branching and merging, this process can be very cumbersome. It can also making tracking history and change logs a little more involved, because the files containing the history are in the branches that are removed.

Stable code goes in the branch. This method is almost the opposite of the one we just discussed. In this model, the majority of development is done in the trunk,

which is not considered stable or usable for a release. When your team gets to a point in the development process when you are ready for a release, you create a branch. All your testers and developers supporting the release will work in this area. Once the release is ready for production, it is usually left alone except for minor bug fixes. This method works well in an environment where many different parts of the application are independent of one another and get released in different intervals. Once a release is no longer supported or available, the branch is removed. This will keep the code that is going into production much quieter, which will shorten your testing time.

Like the model we just saw, this model carries development process overhead. Everyone on the team, including testers, needs to understand the branching process and where things are in the repository. You will also run into the problem of supporting multiple releases for bug fixes. Let's say you have three releases you are supporting for a component of your application. You realize that the same bug is in all three releases of the code. In this model, you will be forced to change each branch. So to implement this method, you must have a good understanding of what code is in what branch and keep on top of the situation all the time. Usually you will need a dedicated configuration management team. The other alternative is to keep only one release of each component active at a time.

"Destructive" development gets branched. This is by far the most common strategy. We saw an example of this is in section 5.1.2 when the development team decided to move from compiler directives to the Apache Portable Runtime. This switch will make the code unusable while the development is going on. The main development will still be done on the trunk, but it is assumed that when you commit to the repository, your code should compile and not drastically affect the functionality of other parts of the application. Production or stable releases are defined by tags, which we will talk about later in this chapter. This method is more middle of the road compared to the previous two. The branches can be in place for a while or for the short term, depending on the timetable of the subproject for which they were created.

In order to use this model, you must be diligent about what gets checked into the trunk, especially if there is a nightly build or people are downloading the absolute latest code for use. Development is occurring in the same place where users are expecting to find relatively stable code, so everything must compile and be usable. You must also make good use of tags and metadata to provide the users of the code with a list of production-worthy releases.

5.1.4 Project root layout

We mentioned earlier in this section that Subversion has come up with a standard for laying out your branches and main development path. By no means do you have to follow this standard; after all, branches and tags are just copies of objects, so you can use any naming convention you want. However, most Subversion repositories you will see use this standard, so unless you have compelling reasons not to, it makes sense to follow this standard. To start, let's assume that you only have one project in the repository, an application called `HelloWorld`. Using the Subversion standard, you will have three directories at the top level: `trunk`, `branches`, and `tags`. These three directories at the top level of the project directory structure are called the project root. The `trunk` directory is your main development path, and it will contain the top-level subdirectories in the project. The `branches` and `tags` directories will contain subdirectories for each branch or tag created, followed by the project directories. So, for example, when you create a branch called `bug-fix.2.1`, you will have a directory with the path `/testrepo/branches/bug-fix2.1/source`. Figure 5.3 illustrates the directory layout of the repository using this standard.

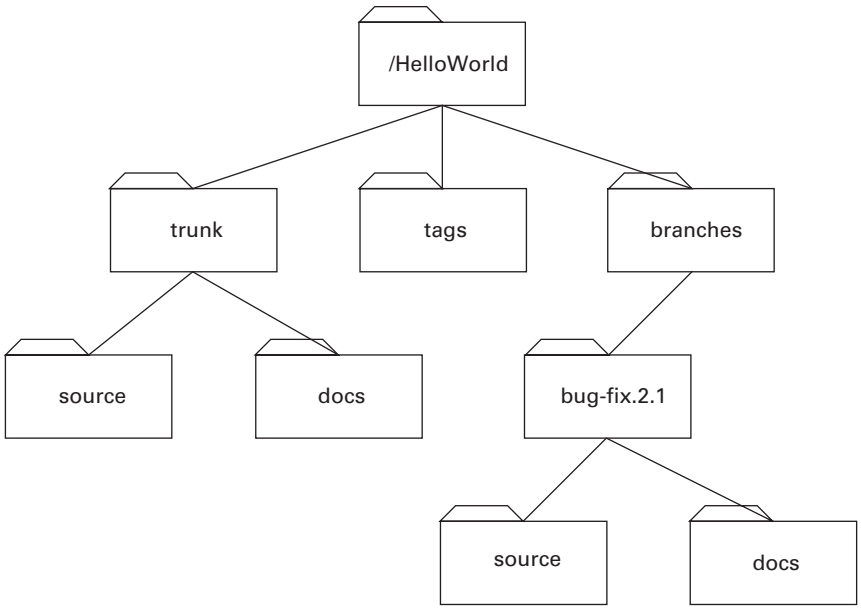

Figure 5.3 Repository layout with a single-project root layout

As you can see, by having a subdirectory dedicated to branches, it is very easy to tell where the source code is. If you name the directories properly, a user should be able to figure out what the branch is for.

Multiple projects. Many users of Subversion will need multiple projects to be in version control. If this is the case, you will have two options. The first is to store all your projects in one repository. Just as with branches and tags, Subversion does not distinguish this in any way. The project root is just another directory in the repository. The advantage of this solution is that administration is required for only one repository. Also, if there is ever a need to copy objects between projects, or perform other commands such as a `merge` or `diff`, you can do so if they all reside in one repository. Also, it helps for reporting procedures around the source code. Since all your code is one location, you need to worry about only one set of change logs.

If the projects are completely unrelated and require a different user base to access the code with different security requirements, you may need to break out the projects into separate repositories. This gives you a more granular level of security and reporting. If you need to segment the change logs between projects, you will have to place them in different repositories because change logs are based on the repository, not the project.

If you break your projects so they are in a one-to-one ratio with repositories, you need not worry about the layout. You will have just the three default directories at the top level (`trunk`, `branches`, and `tags`) and then your project files underneath. But if you are storing multiple projects in the repository, you will need to decide whether they will have their own root or if it is to be shared. For example, let's say our HelloWorld application has three projects: GUI, core, and backend. Each of these projects could have its own root directory, or they could all share one. First, let's look at an example of the layout assuming all the projects are under one project root directory, as shown in figure 5.4.

In the layout, all the projects are grouped under the `trunk`, `tags`, and `branches` directories. Since most branches are created from copies of the entire repository, any branch will have all the projects bundled together. This model works when the projects are tightly integrated and it is difficult to separate them, even during development. So when you are working on the branch `bug-fix.2.1`, you will need all the application components. If this is not the case, you may want to consider using multiple project roots, as shown in figure 5.5

When you compare this layout to the previous one, you will notice that instead of one project root containing all the projects, in this layout each project has its

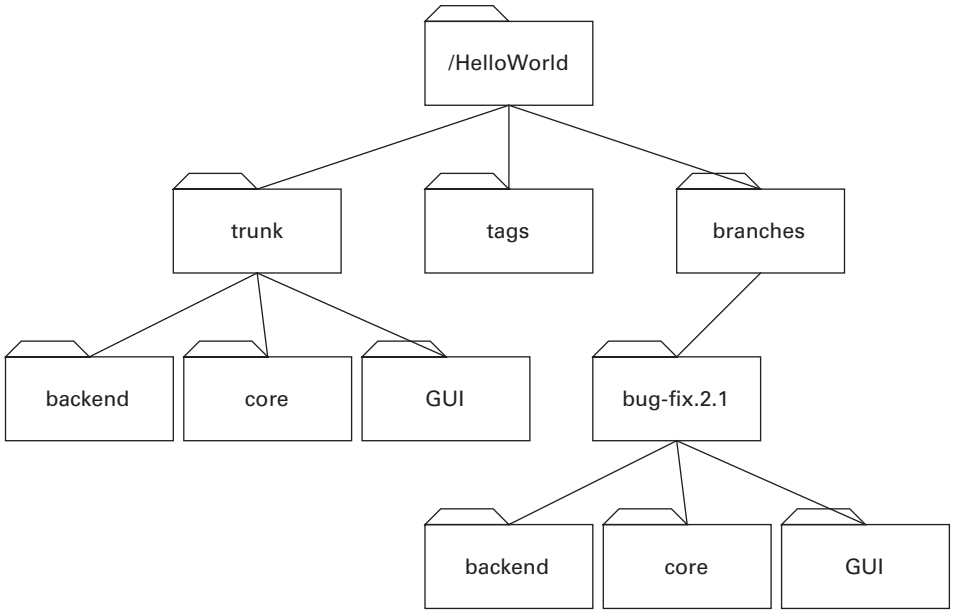

Figure 5.4 Repository layout with multiple projects in one root

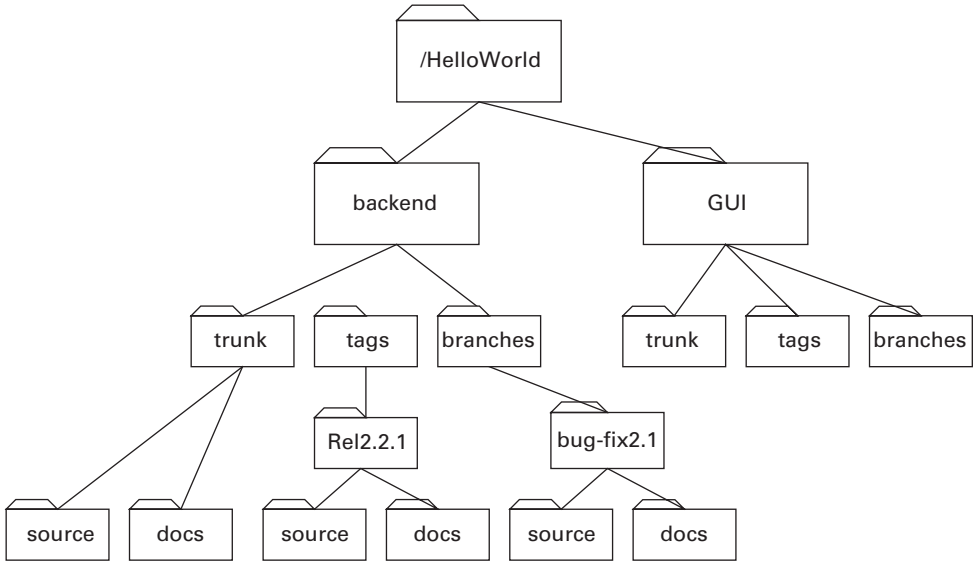

Figure 5.5 Repository layout with multiple project roots

own root. This allows you to branch at a more granular level. Again, this design has no specific meaning to Subversion, but it is a more structured layout for the user to work in. Since most users think of projects as logically starting at the root, the project should be broken out according to what makes sense to them.

Nested project roots. Since Subversion considers the project root as just another set of directories, the level in the tree in which it resides does not matter. You can have a project root at any level of the tree, and they can be mixed across different levels. Consider the layout in figure 5.6. There are a couple things to notice in this layout; the first is that the project roots are nested at multiple levels in the directory tree. Also, take a look at the path `/core/trunk/gui`. You will see another project root under this directory. You can nest project roots at as many levels as make sense to your development environment. As a general rule of thumb, you can create a project root at any point where someone will want to check out the repository. So in the last example, we can assume that some developers will check out from the `core` directory, but some will need only the `gui`. The backend project has only one project root, so that subproject cannot easily be broken into smaller components.

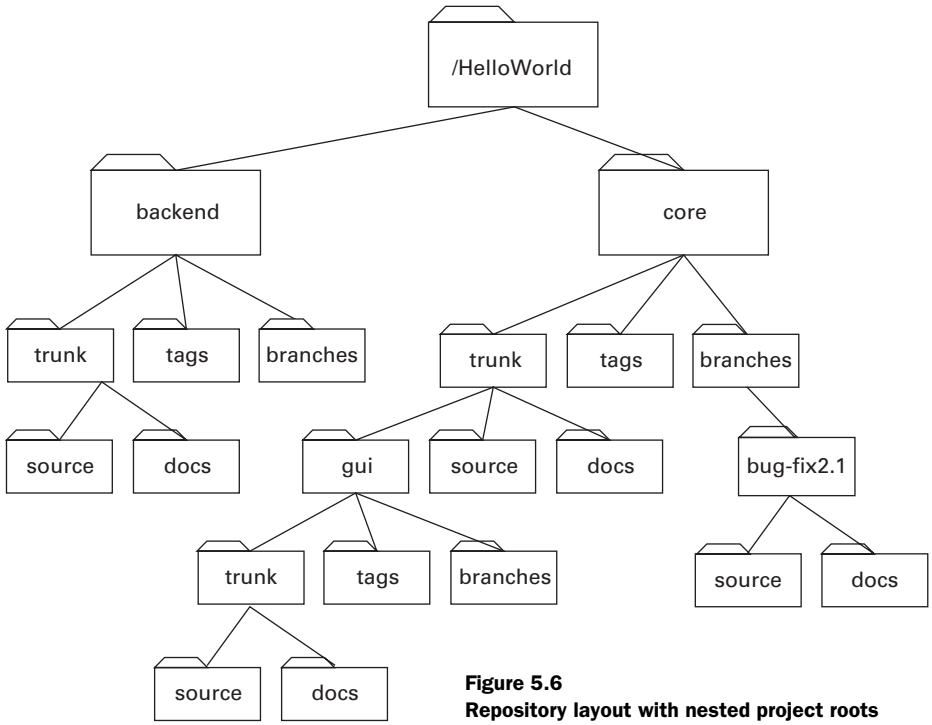

Figure 5.6
Repository layout with nested project roots

5.2 Creating branches

You already know that a branch in Subversion is just a copy of a file, a directory, or an entire project. So to create a branch, you will just use the `copy` command we touched on briefly in chapter 2. Where you make the copy is up to you, but it is recommended that you use the standard project root, as demonstrated in the previous section.

5.2.1 Creating a branch from a working copy

If you have a working copy of the repository checked out, you can just use the `svn copy` command on the local path of the directory you are branching. For example, using the directory structure from the previous section, let's branch the core project:

```
$ pwd
/home/jeff/projects/testrepo/core
$ svn mkdir branches/bug-fix2.1
A         branches/bug-fix2.1

$ svn copy trunk/source branches/bug-fix2.1
A         branches/bug-fix2.1/source

$ svn commit --message "created bug-fix2.1 branch"
Adding         core/branches/bug-fix2.1
Adding         core/branches/bug-fix2.1/source

Committed revision 28.
```

First, we created the directory for the branch called `bug-fix2.1`. The files and directories are now separate objects in the branch from the trunk. The developers working on the bug fixes will work in the new directory. The group working on the main development will stay in the trunk. Now assume that we have been making changes to a file in both directories. Take a look at the change logs of the file `main.c` first in the trunk and then in the new branch we created:

```
$ svn log --verbose trunk/source/c/main.c
------------------------------------------------------------------------
r30 | jeff | 2004-04-27 20:11:48 -0400 (Tue, 27 Apr 2004) | 1 line
Changed paths:
   M /core/trunk/source/c/main.c

Call the hello() function 10 times, not 5
------------------------------------------------------------------------
r27 | jeff | 2004-04-25 16:11:54 -0400 (Sun, 25 Apr 2004) | 1 line
Changed paths:
   D /core/gui
   D /core/source
```

```
   A /core/trunk/gui (from /core/gui:26)
   A /core/trunk/source (from /core/source:26)

moved some stuff
------------------------------------------------------------------------


$ svn log --verbose branches/bug-fix2.1/source/c/main.c
------------------------------------------------------------------------
r29 | jeff | 2004-04-27 20:10:35 -0400 (Tue, 27 Apr 2004) | 1 line
Changed paths:
   M /core/branches/bug-fix2.1/source/c/main.c

fixed main function header
------------------------------------------------------------------------
r28 | jeff | 2004-04-25 19:43:13 -0400 (Sun, 25 Apr 2004) | 1 line
Changed paths:
   A /core/branches/bug-fix2.1
   A /core/branches/bug-fix2.1/source (from /core/trunk/source:27)

created bug-fix2.1 branch
------------------------------------------------------------------------
r27 | jeff | 2004-04-25 16:11:54 -0400 (Sun, 25 Apr 2004) | 1 line
Changed paths:
   D /core/gui
   D /core/source
   A /core/trunk/gui (from /core/gui:26)
   A /core/trunk/source (from /core/source:26)

moved some stuff
------------------------------------------------------------------------
```

We know that the copy was done in revision 28, both from the output when we did the commit and by looking at the change logs of the branched version. This file will appear in the branch only because it was created, but nothing was changed in the version from the trunk. Everything before this revision will be identical because the change logs get copied along with the contents. But look at both logs after revision 28; they are different. After the copy, they can be worked on independently because they are distinct objects. It is up to you to keep changes in sync, depending on the reason for creating the branch.

5.2.2 Copying directly into the repository from a working copy

In the previous example, we used the `copy` command to create a new working copy version of the directory, which we had to commit. If you want to commit directly to the repository and not create the directory locally, you can specify a URL as the target. This is usually done for convenience, but there may be other

reasons for doing so. For example, if your working copy space is limited, you may want to create the branch in the repository but not populate the local disk. Even though a branch is cheap in terms of space in the repository, such is not the case with the working copy. The file actually gets replicated, so if you are branching an enormous amount of code, space could be an issue. The following example creates a new branch in the repository and commits it immediately:

```
$ pwd
/home/jeff/projects/testrepo/core

$ svn copy trunk/ file:///repos/testrepo/core/branches/file_io \
--message "Create a branch to write io to a file, not STDOUT"

Committed revision 34.

$ svn up
A  branches/file_io
A  branches/file_io/source
A  branches/file_io/source/test2
A  branches/file_io/source/java
A  branches/file_io/source/java/hello.java
A  branches/file_io/source/java/main.java
A  branches/file_io/source/java/goodbye.java
A  branches/file_io/source/c
A  branches/file_io/source/c/hello.c
A  branches/file_io/source/c/main.c
A  branches/file_io/gui
A  branches/file_io/gui/trunk
A  branches/file_io/gui/trunk/source
A  branches/file_io/gui/trunk/source/c
A  branches/file_io/gui/branches
Updated to revision 34.
```

From the top level of the working directory, we specified the entire trunk as the source of the copy. Notice that this time, instead of the second argument being a directory in the working copy, it is the URL of the target we want for the branch. In this example, the directory `branches/file_io` did not exist before we ran the `copy` command. Subversion automatically created the target directory for us.

There are a couple of other key points to look at in this example. The first is the `update` command we ran after the copy. Like all Subversion commands, when you run the `update` command against the repository, it does not affect your working copy. This is the case even though we used a working copy directory as the source of the copy. If you want the changes to be populated locally, you must run the `update` command. The good news, though, is that the changes appear immediately

in the repository. Since the command commits right away, you need to supply a log message. You can use any of the methods to enter the message that we mentioned in chapter 2. Finally, take a look at the `copy` command, and you will see a `/` character after the `trunk` in the first argument. By adding this character, you are telling Subversion that you want the contents of the directory. Let's see what would have happened if we had left this off the directory name:

```
$ svn copy trunk file:///repos/testrepo/core/branches/file_io \
--message "Create a branch to write io to a file, not STDOUT"

Committed revision 35.

$ svn up
A  branches/file_io/trunk
A  branches/file_io/trunk/source
...
A  branches/file_io/trunk/gui/trunk/source/c
A  branches/file_io/trunk/gui/branches
Updated to revision 35.
```

The difference is subtle, but if you look at the entire path created in the `branches` directory, you will see it. Instead of the contents of `trunk` being at the top level of the new branch, the `trunk` level is used. While this may not hurt anything, you might not get exactly what you expect. So just remember that inserting a `/` character will put the contents at the top level; if you omit it, the directory will be added.

5.2.3 Creating a branch on a checkout

If you can use a working copy object for the source and a URL as the target, it only makes sense that you can also reverse this process. This means that you can copy from a URL to a working copy path, which is useful in scripts or triggers from defect-tracking tools. For example, assume your development process dictates that all activities are to be worked on in branches. When you accept a task, an automated script or wrapper could get the latest code from the repository and create a branch in your working copy. This will help you keep the branch names and creation in a standard format. All we will do here is substitute the path in the first argument with a URL. Let's create a bug fix branch from the trunk:

```
$ svn copy \
file:///repos/testrepo/core/trunk/source  branches/bug-fix2.2

A  core/branches/bug-fix2.2/source/java
A  core/branches/bug-fix2.2/source/java/hello.java
A  core/branches/bug-fix2.2/source/java/main.java
A  core/branches/bug-fix2.2/source/java/goodbye.java
```

```
A  core/branches/bug-fix2.2/source/test2
A  core/branches/bug-fix2.2/source/c
A  core/branches/bug-fix2.2/source/c/hello.c
A  core/branches/bug-fix2.2/source/c/main.c
Checked out revision 36.
A         core/branches/bug-fix2.2/source
```

By looking at the output you can see what the command has done. First, the new branch has been checked out to the working copy. Just as with the `checkout` command, Subversion will tell you which revision has been checked out, in this case 36. The difference between this and a regular checkout is that the path in the working copy is different because you specified the target. Also, notice that there is an extra line after the checked out revision number. Since the branch did not exist, the directory had to be created in the working copy, which is what the added line shows. Now let's look at the `status` to see what state the working copy is in:

```
$ svn status
A +  core/branches/bug-fix2.2
```

The directory `core/branches/bug-fix2.2` is scheduled to be added to the repository on the next commit. Also, the command includes the + character, which means that extra history will be added along with the contents. Even though you got the branch directly from the repository, because your target is a working copy, you will need to run the commit to save the changes in the repository.

5.2.4 Repository to repository

There is one last combination of arguments the `copy` command will accept, as you can probably guess; both the source and target can be URLs. This allows you to create a branch and immediately commit without ever having a need for a working copy. Let's create a new branch directly in the repository:

```
$ svn copy --message "create bug-fix2.3 branch" \
file:///repos/testrepo/core/trunk/source \
file:///repos/testrepo/core/branches/bug-fix2.3

Committed revision 46.
```

Since the target is a repository URL, the command will automatically commit the change. When this happens, you will need to specify a log message. This is another good method for creating a branch if you are trying to automate the process from a script or trigger. You can do this from anywhere without checking out a working copy.

5.2.5 Creating a branch from a specific revision

Subversion will allow you to create a branch from a previous revision of the repository. This will come in handy if you need to go back and fix something permanently. For example, in our HelloWorld code, we are calling a function to print the statement. Let's say there was a problem in the function, and you implemented just a temporary workaround and committed it to the repository as revision 47. Now that the production fire is out, you can create a branch from revision 46 and work on the original problem at your leisure. Once the problem is corrected, you can merge the changes into the trunk:

```
$ svn up
At revision 47.

$ svn copy --revision 46 trunk branches/fix_hello_function
A  branches/fix_hello_function/source
A  branches/fix_hello_function/source/java
A  branches/fix_hello_function/source/java/hello.java
A  branches/fix_hello_function/source/java/main.java
A  branches/fix_hello_function/source/java/goodbye.java
A  branches/fix_hello_function/source/test2
A  branches/fix_hello_function/source/c
A  branches/fix_hello_function/source/c/hello.c
A  branches/fix_hello_function/source/c/main.c
Updated to revision 46.
A         branches/fix_hello_function
```

The first thing to notice is that the output looks like the copy when the repository URL was the source, even though we used a working copy path. Since we used an older revision as the source, Subversion had to go back into the repository. Just as with the `checkout` command, even though we are starting with an older copy, when we commit, the changes will be saved to latest revision of the repository. Let's say we made some changes. Now commit and see what happens:

```
$ svn commit --message "Fixed hello function"
Sending        branches/fix_hello_function/source/java/hello.java
Transmitting file data .
Committed revision 48.
```

The revision created is number 48, even though we started from 46. Since this was done in the branch, no changes were made on the trunk. When you are ready to move these changes into the trunk, you will have to use the `merge` command.

Common problems with previous revisions. You may be starting to see that the `copy` command, used in the correct context, can be similar to the `checkout` command. There are also similar problems in going back to a previous version of the repository.

For example, if we try the same `copy` command from revision 10 of the repository, Subversion will complain:

```
$ svn copy --revision 10 trunk branches/fix_hello_function
svn: Path 'file:///repos/testrepo/core/trunk' not found in revision '10'
```

The path of the object in the repository we are trying to access did not exist in the revision we are asking for. This means that the `trunk` directory must have been created sometime after revision 10. You can go back through the change logs to find the earliest revision of the repository the object was in, or you could use a different directory.

5.3 Changing repository URLs

For the shortcut enthusiast, Subversion provides a command to quickly change your local copy to point to a different location in the repository: the `switch` command. While this command may not be commonly run, it can save your hide in certain situations. For example, let's say you put your current bug fix development in a branch. In your working directory, it will look something like `/tes-trepo/branches/bugfix`. Now assume that you have a script that deploys patches to production based on this directory. When you are locked into something like this, it may be difficult to change the script if you need to do an interim release. Let's say you have two branches, your main `bugfix` branch and a side development path for a separate, smaller bug. When you look at the `info` for the `bugfix` directory in the working copy, it will appear like this:

```
$ svn info .
Path: .
URL: file:///repos/testrepo/branches/bugfix
Revision: 23
Node Kind: directory
Schedule: normal
Last Changed Author: jeff
Last Changed Rev: 23
Last Changed Date: 2004-05-09 12:25:18 -0400 (Sun, 09 May 2004)
```

As expected, the URL the directory points to matches the path in the working copy. But now we want to run the script from the same path using a different branch. Let's run the `switch` command and see what happens to the working copy path:

```
$ svn switch file:///repos/testrepo/branches/bugfix1.1
U  source\java\hello.java
Updated to revision 24.
```

All you need to do is specify the new URL to which the current directory will point. The output looks just like that from the `update` command. In fact, the `switch` is actually an extension of `update`; not only will it point the directory to a different URL, but the working copy will be updated. When you run the command, the only file that is different is `hello.java`, which gets updated in the working copy. When you look at the output from the `info` command, you will see the difference:

```
$ svn info Path: .
URL: file:///repos/testrepo/branches/bugfix1.1
Repository UUID: 03539556-b733-1240-bd01-78f0d22251c4
Revision: 24
Node Kind: directory
Schedule: normal
Last Changed Author: jmachols
Last Changed Rev: 24
Last Changed Date: 2004-05-09 12:26:50 -0400 (Sun, 09 May 2004)
```

Now the URL is pointing to `bugfix1.1`, even though the path is the same. You can now run the script on the original working copy path, but it will use the code from the URL `bugfix1.1`. When you commit, or update using the working copy, it will still be pointing the new location in the repository. So in this example, any commits you do will update `/bugfix1.1`, not `/bugfix`. When you are finished with the switched URL, you will need to manually change the working copy to point back to the original location.

Entering a path. As with all Subversion commands, when you do not specify a path, the current directory is used. The `switch` command does not have to be run from the working directory, you can specify the path to be used. We could have received the same results by running this command:

```
$ svn switch file:///repos/testrepo/branches/bugfix1.1 \
/projects/testrepo/branches/bugfix

U  \projects\testrepo\branches\bugfix\source\java\hello.java
Updated to revision 25.
```

5.3.1 Recursion

In the previous example, we ran the `switch` command on the entire `bugfix` directory, and it acted recursively on all subdirectories underneath it. You can run this command with the `--non-recursion` option so that it will affect only one level. Let's continue using the example we have been working from. Instead of deploying the entire branch, we'll assume that you need only the `source/java` directory from the `bugfix1.1` branch to be deployed with the rest of the `bugfix` directory:

```
$ svn switch --non-recursive \
file:///repos/testrepo/branches/bugfix1.1/source/java java

U  java\hello.java
Updated to revision 25.
```

This command will affect only the `java` directory and nothing else at that level or any other subdirectory. Take a look at the `info` from the `java` directory. Just as with the previous example, the URL will be different than the path:

```
$ svn info java
Path: java
URL: file:///repos/testrepo/branches/bugfix1.1/source/java
Repository UUID: 03539556-b733-1240-bd01-78f0d22251c4
Revision: 27
Node Kind: directory
Schedule: normal
Last Changed Author: jmachols
Last Changed Rev: 25
Last Changed Date: 2004-05-09 13:01:14 -0400 (Sun, 09 May 2004)
```

Now look at the `info` from a file that is under the `java` directory. It will still point to the original URL:

```
$ svn info java/gui/file.java
Path: java\gui\file.java
Name: file.java
URL: file:///repos/testrepo/branches/bugfix/source/java/gui/file.java
Repository UUID: 03539556-b733-1240-bd01-78f0d22251c4
Revision: 27
Node Kind: file
Schedule: normal
Last Changed Author: jmachols
Last Changed Rev: 27
Last Changed Date: 2004-05-09 14:58:13 -0400 (Sun, 09 May 2004)
Text Last Updated: 2004-05-09 14:58:01 -0400 (Sun, 09 May 2004)
Checksum: d41d8cd98f00b204e9800998ecf8427e
```

While this situation is perfectly acceptable, it can be complex and difficult to keep track of. Any changes will go to different locations in the repository.

5.3.2 Moving repositories

In previous chapters we showed that it can be cumbersome to continually check out working copies of the repository, so we used the `update` command instead. What happens if you have a well-established working copy, and the hostname or location of the repository changes? You could check out a new working copy and lose all of your non-versioned files—or possibly copy them over. Instead of dealing with this problem, you can use the `switch` command to change the URL of your

working copy to point to the new repository location. Let's say the repository we have been using has changed from `/repos/testrepo` to `/repos/svn/testrepo`. When this happens, the working directory will no longer be valid:

```
$ svn up
svn: Unable to open an ra_local session to URL
svn: Unable to open repository 'file:///repos/testrepo'
```

The `.svn` directories in the working copy still point to the original URL `'file:///repos/testrepo'`, which no longer exists. Using the `switch` command with the `--relocate` option, you can change the working copy to point to the new repository:

```
$ svn switch --relocate file:///repos/testrepo file:///repos/svn/testrepo

$ svn info
Path: .
URL: file:///repos/svn/testrepo
Repository UUID: 03539556-b733-1240-bd01-78f0d22251c4
Revision: 26
Node Kind: directory
Schedule: normal
Last Changed Author: jmachols
Last Changed Rev: 26
Last Changed Date: 2004-05-09 14:56:18 -0400 (Sun, 09 May 2004)
```

This command should be run from the top level of the working copy to ensure that it is moved entirely. At this point you can start performing updates and commits from the working copy. In the previous examples, the `switch` command was a temporary solution, and you had to switch back to the original URL. In this situation, you permanently switch the URL of the `.svn` directory.

5.3.3 Switching to an old revision

Since the `switch` is just an extension of the `update` command, it is possible to change the URL the working copy points to and update to the latest revision at the same time. For example, if you are on revision 29 of the repository, but you wanted to deploy the contents of revision 27 of the `bugfix1.1` branch, you could still use your generic deploy script that worked from the directory `bugfix`:

```
$ pwd
/projects/testrepo/branches/bugfix/source

$ svn switch --revision 27 \
file:///repos/svn/testrepo/branches/bugfix1.1/source/java

U  java\hello.java
```

```
D  java\gui
Updated to revision 27.

$ svn info java
Path: java
URL: file:///repos/svn/testrepo/branches/bugfix1.1/source/java
Repository UUID: 03539556-b733-1240-bd01-78f0d22251c4
Revision: 27
Node Kind: directory
Schedule: normal
Last Changed Author: jmachols
Last Changed Rev: 25
Last Changed Date: 2004-05-09 13:01:14 -0400 (Sun, 09 May 2004)
```

You can see that the working copy has been updated to an older revision of the repository. Also, the directory now points to the new URL. The same conditions apply to this as to the `update` and the basic `switch` command. The working directory will point to the new repository URL until you manually switch it back. You will need to be careful about switching to a revision that is too old, because the directory structure may not be correct. Also, any commits you perform will be done on the URL you have switched to and will create a new revision based on the latest number, not the one you have switched to.

5.4 Merging revisions

Since the point of using branches is to support multiple development paths, you need a way to bring the changes back into the main development path. You will also need the ability to get incremental changes in the main development into your branch. This can be done manually in various ways, but to ensure that you do not miss anything, the recommended way is to use the `merge` command. This command will take all the changes in both objects and combine them into one object.

5.4.1 A simple merge

Let's say you have created a branch at revision 35 and made changes to it at revision 36. If you want to get these modifications into the trunk, you would select the range of revision numbers to be merged into the target file:

```
$ pwd
/testrepo/trunk/source/java

$ svn merge --revision 35:36 \
file:///repos/svn/testrepo/branches/bugfix/source/java

U  hello.java
```

The command will take the differences between the revisions specified and apply them to the destination. So in the previous example, any changes in files in the directory `/bugfix/source/java` between revisions 35 and 36 will be merged with the target files in the current directory. In this example, the only file in the directory that had changes between the two revisions is `hello.java`. Subversion will attempt to automatically merge the changes into the file. If it can do so without any conflicts, you will see `U` for update. We will show what happens when there is a conflict later.

5.4.2 Moving changes from a branch

You have just seen how to merge one revision into another location. You will also need to know how to move all the changes from a branch into the trunk. When you have finished with the development of a branch, you will want to merge the changes of the branch back into the main path. This will be similar to the previous merge, except that the range will start from the point where the branch was created and end with the `HEAD`:

```
$ cd branches/bugfix3.1/

$ svn log .
------------------------------------------------------------------------
r39 | jeff | 2004-05-10 21:00:11 -0400 (Mon, 10 May 2004) | 1 line
Changed paths:
   A \branches\bugfix3.1
   A \branches\bugfix3.1\source (from \trunk\source:37)

created bugfix3.1 branch
------------------------------------------------------------------------
```

When you run the `log` command on the `bugfix3.1` directory, you will see the point where it was created. As we showed you in chapter 3, the change path will indicate when the branch was added and from where it was copied. The `bugfix3.1` branch was created in revision 39, so this will be your start point. For the end point of the range, you can simply use the `HEAD` keyword to get all the changes in the branch:

```
$ svn merge --revision 39:HEAD \
file:///repos/svn/testrepo/branches/bugfix3.1/source/java

U  hello.java
```

5.4.3 Merging a single file

So far you have seen the merge work only on directories in a branch, but you may want to merge a single file. For example, let's say you are working in a bug fix branch for the next release of the software. In the trunk, another developer has fixed a couple of typos in the file `hello.java`, which you need to merge into your branch. Instead of getting all the changes in the `java` directory of the trunk, you can specify the file alone:

```
$ svn merge --revision 34:35 \
file:///repos/testrepo/trunk/source/java/hello.java

U hello.java
```

5.4.4 Conflicts

As with the `update` command, there is a possibility in the merge that Subversion will be unable to automatically combine the changes in the two objects. When this happens, the file goes into a conflict status and manual intervention will be required. Consider the file `hello.java`, where two developers have made modifications to the same line:

```
$ svn merge --revision 35:HEAD \
file:///repos/testrepo/branches/branch1.2/source/java

C  hello.java
```

You can tell by the `C` character in front of the filename that there was a conflict during the merge. Just as in the conflict with the update, Subversion will create additional files to assist you in resolving the conflict:

```
$ ls -1
gui
hello.java
hello.java.merge-left.r35
hello.java.merge-right.r38
hello.java.working
```

The four different copies of the file `hello.java` should look familiar to you. Each of these files represents the object at a specific point during development. If you need a refresher on this, take a look at section 3.3. Once the conflict is resolved, you will need to run the `svn resolved` command to clean up the files and the state of `hello.java`.

5.4.5 Keeping track of your merges

If you simply merge all the changes of a branch into the trunk and then remove the branch, you will not have to keep a close eye on what you have merged. This is

especially true if you experienced conflicts on a merge. In the previous example, we took the changes from revisions 35–38 and merged them into the branch. Now let's say you manually fixed the conflict and then went on to make more changes in the branch. If you use the branch start revision and `HEAD` as the boundaries for the range, you will be using revisions 35–41. By doing this, you will be trying to merge changes that have already been added to the trunk. To see how this can be problem, consider the content of the file `hello.java`:

```
$ cat hello.java
public class hello
{
        String m_path = "/$HOME" ;
        public void printIt ()
        {
                String m_fileName = new String (m_path + "/config") ;
                this.read_config(m_fileName);
                System.out.println("Hello world");
        }
}
```

Now assume that you have created a branch and started to make some changes. As you were working on this, someone else made a change to the trunk. You find out that the change is a checkstyle fix, so you decide that you will merge it into your branch now. You run the `merge` command from the start of the checkstyle fix revision to the `HEAD`, and then you see that there is a conflict. After realizing the problem, you view the `hello.java` file to see which lines are in conflict:

```
$ svn merge --revision 47:HEAD \
file:///repos/svn/testrepo/trunk/source/java
C  hello.java

$ cat hello.java
public class hello
{
        String m_path = "/$HOME" ;
        public void printIt ()
        {
                String m_fileName = new String (m_path + "/config") ;
                this.read_config(m_fileName);
<<<<<<< .working
                System.out.println("Hello cruel world");
=======
                System.out.println( "Hello world" ) ;
>>>>>>> .merge-right.r49
        }
}
```

You can see by the output that there is one conflicting line between the branch and the trunk. The process to manually fix the conflict with both changes should be straightforward. Let's assume that you have made the following modification to the conflicting part of the file:

```
$ cat hello.java
public class hello
{
        String m_path = "/$HOME" ;
        public void printIt ()
        {
                String m_fileName = new String (m_path + "/config") ;
                this.read_config(m_fileName);
                System.out.println( "Hello cruel world" ) ;
        }
}
```

Now you have a combination of the changes committed to the repository, and you can go on making further changes. Let's say some more checkstyle changes are posted, but being the savvy Subversion user you are, you run a `cat` on the trunk to see what has changed:

```
$ svn cat trunk/source/java/hello.java
public class hello
{
        String m_path = "/$HOME" ;
        public void printIt ()
        {
                String m_fileName = new String ( m_path + "/config" ) ;
                this.read_config( m_fileName ) ;
                System.out.println( "Hello world" ) ;
        }
}
```

Notice that the "Hello world" print statement in the trunk remains the same because you did the merge on the branch, not the trunk. Since the trunk is not yet ready for the change, you cannot merge the branch now. So if you use the same starting point and the `HEAD` for the range on the second merge, you will still have a problem:

```
$ svn merge --revision 47:HEAD \
 file:///repos/svn/testrepo/trunk/source/java

C  hello.java
```

You are still getting a conflict because Subversion is trying to apply the same change you just fixed in the previous merge. In order to solve this you need to adjust the range of the revision numbers. If you use the revision after the merge as

the starting point, you will not be duplicating any changes. After you committed the fix for the previous conflict, the revision number of the repository created was 49, so this will be your new start point. You can still use `HEAD` as the end point because you will want to include the most recent changes:

```
$ svn merge --revision 49:HEAD \
 file:///repos/svn/testrepo/trunk/source/java

U  hello.java
```

Since you used a revision range that does not overlap, Subversion was able to merge without a conflict.

How to keep track. There is no way to automatically determine what was merged after the fact. Therefore, it is important to establish a method of keeping track of what you did. A simple way of accomplishing this is to add the information to the change log when you commit. If you get into the habit of putting this information in the message and checking the message before you run the merge, you will be able to see what has been merged so far. In our last example, we would commit using the following command:

```
$ svn commit --message \
"Merged changes from trunk/source/java/hello.java.
> Revisions 49-52 were merged containing checkstyle fixes"

Sending        java/hello.java
Transmitting file data .
Committed revision 53.
```

Now that you have added this statement to the message, it will be necessary to read the log messages before merging again.

5.4.6 Doing a dry run

Since merging updates your working copy and can potentially cause problems if there is a conflict, Subversion lets you do a dry run of the `merge` command. This will tell you which files will be updated and whether Subversion will be able to automatically merge the changes or whether a conflict will occur:

```
$ ls -l
total 12
drwxrwxr-x    3 jeff     jeff         4096 May 10 11:40 gui
-rw-rw-r--    1 jeff     jeff          290 May 14 10:00 hello.java
-rw-rw-r--    1 jeff     jeff           65 May 14 11:23 main.java

$ svn merge --dry-run --revision 45:HEAD \
 file:////repos/testrepo/branches/bugfix3.3/source/java
```

```
U  hello.java
C  main.java

$ ls -l
total 12
drwxrwxr-x    3 jeff     jeff         4096 May 10 11:40 gui
-rw-rw-r--    1 jeff     jeff          290 May 14 10:00 hello.java
-rw-rw-r--    1 jeff     jeff           65 May 14 11:23 main.java
```

Take a look at the `merge` command and notice the extra switch, `--dry-run`. This is what tells Subversion to just examine the changes and not actually update the working copy. If you look at the output, you will notice that two files require changes for the merge, `hello.java` and `main.java`. According the output, Subversion can automatically merge the changes in `hello.java` since it has a `U` character. The file `main.java` shows a `C`, which indicates that the changes conflict and cannot automatically be updated. Also, compare the output of the two directory listings before and after the merge. Nothing has changed or been updated since we added the `--dry-run` switch.

5.5 Tags in Subversion

If you are familiar with other version control systems, you are used to the fact that a tag is another form of metadata, but this is not the case in Subversion. Just like a branch, a tag is simply a copy of a directory. What is important when dealing with a tag is the way it is used. Subversion does not care how it is used, but a tag should be an important part of your development process. A tag should represent a snapshot of the repository at a critical point. The most common example of the use of a tag is for a release of the software. A tag gives you a quick way to view the code and see what is in a particular release. For example, let's say you are ready to release version 1.0 of your software, so you can create a tag just as you would a branch. First, we will create a `tags` directory at the top level, which will complement the `trunk` and `branches` directories:

```
$ pwd
/home/jeff/projects/testrepo/tags

$ svn mkdir release_1.0
A         release_1.0

$ svn copy ../../trunk/source/ .
A         source

$ svn commit -m "added release_1.0 tag"
Adding         release_1.0
```

```
Adding         release_1.0/source
Adding         release_1.0/source/c
Adding         release_1.0/source/c/hello.c
Adding         release_1.0/source/c/main.c

Committed revision 107.
```

As you can see, this process is identical to creating a branch. The only difference is the location of the copy. Because a tag represents a point in time, you really should not make changes to it since doing so would invalidate the snapshot.

5.6 Summary

Branching is traditionally one of the more difficult concepts in any version control system. Subversion has made this concept easier to grasp technically, by simply making a branch a copy of a directory in the repository and not some other piece of metadata. This is only half the battle; you will still need to figure out which branching strategy is best for your development process and software. The key—no matter how you choose to use branches and tags—is that you actually use them and are consistent. There is nothing worse for a development team than for someone to copy a chunk of code to a home directory and work on it for months before checking it back in. In Subversion, tags fall into this same category as code. It is a good idea to create a tag at each major milestone in your project's lifecycle. Since a tag is just a Subversion copy, doing so is very efficient. Remember, these copies are just links to the original locations with changes applied, and they take up little space, so don't be afraid to use them.

Once you define your processes and solidify them with branching and tags, you can build automation around them. We have already shown some examples of Subversion commands lending themselves well to scripting. We will provide more examples of this when we delve into properties, which is the next step on your road to real configuration management.

6 Properties

In this chapter

- Using properties
- Adding, changing, and removing properties
- Built-in properties
- Revision properties

Earlier in the book we talked about Subversion being able to help manage and enforce your development lifecycle process. This is done through the use of properties, which are also known to some as metadata. Typically, assigning metadata or properties is a complex undertaking, but this is not the case with Subversion. So is simple a good thing? Alex Karasulu of the Apache Directory Project is known for saying, "The more simple the implementation of a piece software, the more robust the application of it can be." Basically, this means that small, simple applications can be used in many different ways and configurations. Subversion has followed this decree with the implementation of properties, which are very simple. The use of properties allows you mold your implementation of Subversion into just about any development process. The only downfall to this is that it requires more development effort on your side. You will need to write additional scripts and wrappers to customize the properties.

6.1 Properties overview

Properties are simply a hash table, which is a name/value pair. Basically, you create a variable and assign it a value; it's no more complicated than that. Subversion treats these properties just like the contents in a file. They are versioned, so when they get modified, a commit is required. Plus, you can retrieve the value of the property at different revisions of the repository. Properties do not have to be attached to files; you can also assign them to directories. There is no difference in creating or accessing properties whether they are attached to a file or a directory. Once you add a property to an object, it stays with that object until you change or delete the property. If you check out an earlier version of the file that was created before you added the property, the property will not be attached to the object. As you will see later in this chapter, this allows you to truly retrieve the state of the repository at a point in time.

6.1.1 Name/value pairs

So what exactly is a name/value pair? It is an arbitrary variable name with some associated text string. The names and values can be any human-readable text string; other than that, there are no restrictions. You can have as many of these pairs attached to an object as you want, but they must have unique names. So, for example, you cannot have two properties with the name `label`, even if they have different values. The following is an example of some properties attached to a file:

```
LABEL                   Release_3.2.1
RELEASE                 3.2.4
```

```
Release                 3.2.1
Release_name            3.2.1
Release_3.2.1-STATE     production
```

There are a variety of characters you can use to create a name, which is case sensitive. Thus, `Release` and `RELEASE` are two different properties. Also, you can include numbers and special characters, as long as they are readable (so no EOF or Line Feed characters).

6.1.2 Properties are versioned

Since Subversion treats properties like file contents, they are versioned the same. In addition, changing or deleting a property will cause the working copy to change and the object to be in a modified state. You will have to run a commit to save the property changes to the repository. Once you have committed them, these properties will remain static until they are changed or removed. To illustrate this, let's assume that you have taken the following steps on the file `hello.java`:

1. *Revision 3*—Create a property called `LABEL` and assigned it a value of `release_1.2`.
2. *Revision 5*—Change the value of `LABEL` to `release_1.4`.
3. *Revision 8*—Remove the property from the file.

Now take a look at figure 6.1 to see what this looks like in the version tree for the file.Just like the contents of a file, a property will stay the same in new versions of the object unless you explicitly change it.

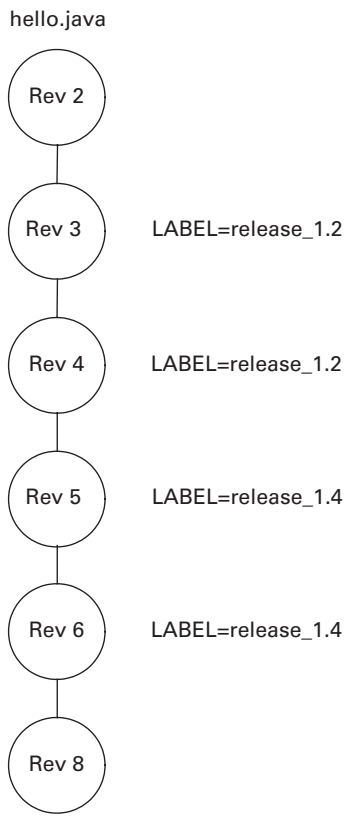

Figure 6.1 Property lifecycle in a file

6.2 Adding a property to an object

To add a property to an object, you use the `svn propset` command. Three arguments are required to run the command: the property name, the value, and the object on which you are placing the property. So, for example, to add a new property called `label` on the file `hello.c`, you can run the following command:

```
$ svn propset label production hello.c
property 'label' set on 'hello.c'
```

The first parameter is the name of the property you are setting; in this case the property name is `label`. The second is the value of the property; here we have set `label` equal to `production`. The last parameter is the file or directory to which you are attaching the property. As you can see by the output, Subversion will give you a confirmation of the operation. Just as with changing the contents of a file, you have not saved anything to the repository yet. Let's run the `status` command and see what the file `hello.c` looks like:

```
$ svn status hello.c
 M     hello.c
```

The `status` command displays an `M` character, which indicates that the object was modified. Notice, however, that the character is offset by one space. Think back to chapter 2 when we talked about the output of the `status` command. Characters in the second field represent changes to metadata such as change logs or properties. In order to save the property, you must run a commit:

```
$ svn commit --message "Added property label to the file"
Sending        c\hello.c

Committed revision 52.
```

At this point, the property has been saved to the object starting at revision 52. This command can be run on a file or a directory. But as you will see in the next section, the behavior on a directory may not be what you expect.

The format of property values. You have already seen that the value of a property can be any human-readable text. This includes multiple words and even lines, which gives the implementation of properties more flexibility because you can include very descriptive values. You should not get carried away with this, but where it makes sense, you can include multiword or multiline values. To include spaces in the value of the property, simply enclose it in quotes:

```
$ svn propset State "System Test" source/c/hello.c
property 'State' set on 'source\c\hello.c'
```

```
$ svn proplist --verbose source/c/hello.c
Properties on 'source\c\hello.c':
  State : System Test
```

Here we set the value `System Test` to the property `State`. You would enter multiline values the same way you would enter a log message with more than one line:

```
$ svn propset Release "Application release 2.3
> Bug Release 2.3.2A" source/c/hello.c
property 'Release' set on 'source\c\hello.c'

$ svn proplist --verbose source/c/hello.c
Properties on 'source\c\hello.c':
  Release : Application release 2.3
Bug Release 2.3.2A
```

You start the value with an open quote. When you get to the point where you want to start the new line, just press the `Enter` key. The command will continue on the next line, as indicated by the > character. You continue typing the value, and when you finish, you end the value with another quote. You are not quite finished yet—don't forget to add the object on which you are setting the `propset` property. When you view the property using the `proplist` command, you will see that the value is carried over to the second line.

Using these longer values can be great for providing descriptive information about an object. However, it is not such a good idea if you are using the property for queries or scripting. The spaces and newline character can make it tricky to match things up. While you can use any readable character for the value, be careful about using the colon (`:`), especially for multiline values. This is the character Subversion uses as a delimiter for the name and the value. If you include this character in the name or the value, it can make reading the output difficult.

6.2.1 Applying properties to a directory

Up until this point, almost all Subversion commands operate recursively, which should be painfully clear to you by now. The `propset` command deviates from the default behavior. When you run the command on a directory, it applies the property only to the directory object. Files in the directory are not changed, nor are subdirectories under it. This is done for safety reasons so you do not accidentally overwrite properties of files in the directory. If you do want to perform the action on all the files in a directory, you can add the `--recursive` switch:

```
$ svn propset --recursive Release 3.2.3 .
property 'Release' set (recursively) on '.'

$ svn status
```

```
M     .
M     hello.c
M     main.c
```

After you run the command recursively on the current directory, Subversion tells you that it recursively set the properties. Also, when you run the `status` command, you will see that multiple files have the property field with the modified character. Remember that the status is on the working copy in relation to the repository, so you still must commit to save the changes.

Overwriting properties. One of the reasons `propset` does not act recursively is because of the risk of overwriting another object. When you run the `propset` command with a name that already exists, it will just change the value. To see how this can be a problem, consider the file `hello.c` with the label `Release` set to `3.2.3`. Now if you run the `propset` command on the directory, you will change the value. The following commands show this dynamic:

```
$ svn proplist --verbose main.c
Properties on 'main.c':
  Release : 4.1

$ svn propset --recursive Release 3.2.3 .
property 'Release' set (recursively) on '.'

$ svn proplist --verbose main.c
Properties on 'main.c':
  Release : 3.2.3
```

Don't worry about the `proplist` command yet; we will get to it shortly. For now just understand that it displays the properties on the file. As you can see by the sequence of commands, the file `main.c` started with the property `Release` equal to `4.1`. After the directory recursively gets set to `3.2.3`, the properties of `main.c` now show that it is set to `3.2.3`. So before you use the `--recursive` switch on the `propset` command, be sure you won't inadvertently overwrite properties on files in the directory.

6.2.2 Passing in arguments as file contents

The Subversion commands such as `propset` that are geared toward process management have methods of passing in arguments other than the default. It is possible that this was done on purpose, but one way or the other it helps with scripts. Most process management commands are embedded in scripts to help automate your software management. Instead of you passing arguments on the command line, the commands can be stored in files and the contents will be used. The

`propset` command can accept the value of the property and the name of the target object as file contents.

Taking the property value from a file. One of the command-line arguments that can be augmented by using file contents is the value to which you are setting the property. Storing the value you are setting in a file has potential benefits; one of the big ones is consistency. If you are going to query against the value of a property, it is imperative that the values match. So if one developer used `release_3.3.2` for the value and another used `Release_3.3.2`, Subversion would consider them different properties. If everyone is reading from the file directly, they are guaranteed to use the same value.

Another benefit is in writing generic scripts. Let's say you have a script that moves code into production. When it moves the source, there is a property named `release_number` that gets set to the release of the software. You can insert a line in the script that reads from a common file to get the value. Before you run the script, all you have to do is update the file, and no change is required to the script. Let's take a look at an example of setting the value using the contents of a file:

```
$ echo "4.3.1" > textfile

$ svn propset Release --file textfile source/c/hello.c
property 'Release' set on 'source\c\hello.c'

$ svn proplist --verbose source/c/hello.c
Properties on 'source\c\hello.c':
  Release : 4.3.1
```

We have added an option called `--file` to the argument list. The contents of `textfile` will be used as the value of the property. Even if there are multiple words or even multiple lines in the file, they will still be attached to the same property.

Filenames. In addition to values, you can also pass in the files or directories to which you are assigning the properties. This can also be beneficial in the script we just talked about. We have a script that sets the release property to a set of source files being deployed. The developers or QA staff just need to add the objects into a file, and the script does not have to be changed. For example, consider the following lines in a script:

```
#!/bin/bash
for FILE in 'cat /source/deploy/target_files.txt'
do
  svn propset Release 4.3.4 $FILE
done
```

Instead of going through something like the loop, you can accomplish the same functionality with just one line:

```
$ cat target_files.txt
source/c/hello.c
source/c/main.c

$ svn propset Release 4.4.2 --targets target_files.txt
property 'Release' set on 'source\c\hello.c'
property 'Release' set on 'source\c\main.c'
```

The file `target_files.txt` contains two objects. When we run the `propset` command, we add the switch `--targets` and give it an argument of the file in which the objects are stored. Each of the objects in the file is individually set.

6.2.3 Editing property values

We have shown an example where the `propset` command can not only add a property to an object, it can also change the value. For example, assume that we have the property `Release` assigned to the file `main.c`. We can change its value by using the `propset` command again:

```
$ svn proplist --verbose main.c
Properties on 'main.c':
  Release : 3.2.5

$ svn propset Release 4.1 main.c
property 'Release' set on 'main.c'

$ svn proplist --verbose main.c
Properties on 'main.c':
  Release : 4.1
```

In addition to the `proplist` method, Subversion has a command called `propedit` that will also allow you to modify an existing property value. The difference between this command and the `propset` command is that an editor will be started. You may be asking yourself why you need an editor to modify a property value from `3.2.5` to `4.1`. You do not. This command is not meant for small values; it should be used when you are dealing with large, multiline values of a property.

Until now, the values of properties you have seen were small, even where they were more than one line long. What if you store something large, such as header information, in a property instead of in the file?

```
Properties on 'main.c':
  Release : 4.1
  header : Filename - main.c
```

```
Command Line Arguments - none

This is a simple main function that will be run from the
command line.  It will loop through the main function
some random amoount of times.  Each time in the loop, it
will call the hello() function.  This will print hello
world to standard out.
```

The property header is sure a mouthful. Take a look at the third line from the bottom; the word *amount* is spelled wrong. Now what should we do to fix this? Well, we could use the `propset` command to reenter the entire text, but that seems like a lot of work just to fix one typo. This is where the `propedit` command comes in. When you execute this Subversion command, an editor will be started and loaded with the current value. Once you have finished making the changes, simply save and exit from the editor, and the value will be updated:

```
$ svn propedit header main.c

...

"svn-prop.tmp" 9L, 298C written
Set new value for property 'header' on 'main.c'
```

Just as with the `commit`, Subversion will pick the editor based on the environment variable `EDITOR` or `SVN_EDITOR`. Also, you can use the `--editor-cmd` option if you want to choose the editor on the command line. Remember that when you change a value of a property, it is changed just from the revision forward. Previous revisions of a file will contain the original value. As with the `propset` command, we have changed only the working copy. If you want to save this to the repository, you will need to run a commit:

```
$ svn status
 M     main.c

$ svn commit --message "Fixed typo in the header property"
Sending        c/main.c

Committed revision 61.
```

6.2.4 Removing properties

The `propset` command will set a property to an object, `propedit` will change the value, and, finally, `propdel` will remove a property from an object. This is a pretty straightforward process: simply give the command the property name you are removing and the target object. For example, to remove the header from the previous example we just set, you can run the following command:

```
$ svn propdel header main.c
property 'header' deleted from 'main.c'.

$ svn proplist main.c
Properties on 'main.c':
  Release
```

When you do the delete, the header property is removed from the file `main.c`. Again, this is only on the working copy, so you will need to commit in order to get this changed in the repository.

Recursive. All of the property commands are exceptions to the rule that recursion is the default behavior. So if you run the `propdel` command on a directory, it will affect only that object, not any of the files or subdirectories underneath it. If you want the command to operate on everything under your target directory, just add the `--recursive` option to the command. For example, let's say you have a standard in your branches that when your code is unusable, you set a property called `Unusable` and set it to a value of `true`. You might use this when your code is being worked on and will not compile or run properly so that it doesn't mistakenly get pushed into production. You can add logic in your deploy script to ignore any files with this property set. The `proplist` command will look something like this:

```
$ pwd
/home/jeff/projects/testrepo/branches/major_rewrite

$ svn proplist --recursive .
Properties on '.':
  Unusable
Properties on 'source':
  Unusable
Properties on 'source/c':
  Unusable
Properties on 'source/c/hello.c':
  STATE
  Release
  Unusable
Properties on 'source/c/main.c':
  Release
  Unusable
```

When you are ready for your branch to be deployed into test or production, you will need to remove the property:

```
$ svn propdel --recursive Unusable .
property 'Unusable' deleted (recursively) from '.'.

$ svn commit --message "removed unusable property"
Sending        major_rewrite
```

```
Sending        major_rewrite/source
Sending        major_rewrite/source/c
Sending        major_rewrite/source/c/hello.c
Sending        major_rewrite/source/c/main.c

Committed revision 64.
```

At this point, all the files and directories in the branch will have the property removed. We also had to run a commit to get the changes saved to the repository. Now we can run the deploy script since the property has been removed.

Gone but not forgotten. You have seen that properties are versioned just like a file's contents, so do you think the properties we have deleted are truly removed from the object? Of course not; they are still attached to the revisions of the file or directory before we removed them. Let's take a look at an example of this. Suppose we have the file `main.c` with a property of `Release` attached to it. Let's remove the property and commit:

```
$ svn proplist --verbose main.c
Properties on 'main.c':
  Release : 4.1

$ svn propdel Release main.c
property 'Release' deleted from 'main.c'.

$ svn commit --message "removed Release property"
Sending        c/main.c

Committed revision 65.

$ svn proplist main.c

$
```

When you look at the previous sequence of events, the results should be fairly intuitive. The property was there, we removed it and ran a commit, and now the property is no longer attached to the file. This all happened at revision 65 of the repository, so let's check out revision 64 and see what that looks like:

```
$ svn update --revision 64
 U main.c
Updated to revision 64.

$ svn proplist main.c
Properties on 'main.c':
  Release
```

Just by looking at the output from the `update` command, you should be able to tell that something is up. Notice that `main.c` has an update character, but it is in the second column. This means that the file's metadata was changed, either the history or property information.

6.3 Listing properties on an object

We have shown some glimmers of the command used to list the properties on a file: `svn proplist`. The listing will show you all the properties that are assigned to the object. You do not need to know the names of any of the properties; the command will find all of them for you:

```
$ svn proplist hello.c
Properties on 'hello.c':
  LABEL
  Release
```

The `proplist` command takes an argument for the object on which you are running the list. Subversion will search through all the properties attached to the file or directory and list them. Notice, however, that only the names are listed; the values are not included in the default `proplist` command. You may think this is useless, but that is not true; there are benefits to listing the names only. For example, consider the `Release` property we have been assigning to objects. We assumed that there is a script that deploys the source file and adds the `Release` name along with a value. Files that are not part of a release will not have a property assigned to them at all. So now let's assume that you want to clean up all the files that are not associated with a release. You can run a `proplist` command on the objects in your working copy and find the ones without any `Release` property. In a situation like this, you will not care what the value is, only that the property exists.

6.3.1 Listing the name and value

You have seen an example of why you would want to list the property name and not the value, but a good majority of the time you will want to see both. You can do this with the `proplist` command using the `--verbose` option.

```
$ svn proplist --verbose source/c/hello.c
Properties on 'source\c\hello.c':
  Release : 4.4.2
  STATE : production
```

This object has two properties attached to it, `Release` and `STATE`, and each of the values for those properties is displayed along with the name. Remember, the list will

give you all the properties attached to the object. If you want to specify a specific property, you will need to use the `propget` command, which we'll discuss shortly.

6.3.2 Property listings on a directory

Just like most Subversion commands, the `proplist` command treats files and directories the same. In fact, if you do not specify an object, the command will default to the current working directory. Just like the `propset` command, the `proplist` command will not act recursively, so it will get only the properties attached to the directory and not any of the files or subdirectories in the target. Take a look at the `proplist` command as applied to a directory:

```
$ pwd
/projects/testrepo/branches/bugfix3.1

$ svn proplist
Properties on '.':
  Branch_Type : hot fix
```

We have listed only the properties on the directory `bugfix3.1`. If you want to see all the files under the directory in the list, you will need to use the `--recursive` switch.

Recursively listing the properties of a directory. The reason why Subversion does not default to a recursive behavior has to do with the size of the output. If you were to run the `proplist` command on the top level of your repository, the output could be difficult to sort through. If you are in a location in the tree where it makes sense, then you can run the command and get manageable output:

```
$ svn proplist --recursive --verbose
Properties on '.':
  Branch_Type : hot fix
Properties on 'source\c':
  Branch_Type : hot fix
Properties on 'source\c\hello.c':
  Branch_Type : hot fix
Properties on 'source\c\main.c':
  Release : 3.4.1
  Branch_Type : hot fix
```

Now in addition to getting the properties for the current directory, you see everything underneath it that has a property attached. Since the command lists only properties, any files or directories with nothing attached to them will not be part of the output. Since the `Branch_Type` property was set recursively, it appears on all the files. The file `main.c` has an additional property of `Release`, which is not associated with any other object in the list.

6.3.3 Listing the properties of a specific revision

So far we have listed the properties of objects in a working copy, but since this is a version control system we can also deal with specific revisions. By now, you should be able to guess the notation of running the `proplist` command on a revision of an object. That's right, you will use the `--revision` option. Since the properties can change between revisions of a file or directory, you cannot specify a range; only one revision can be used. Let's take a look at a `proplist` example that gets an older revision. First, we will look at the properties of the current revision, and then we will get the previous one:

```
$ svn proplist --verbose main.c
Properties on 'main.c':
  Release : 3.2.5
  STATE : production

$ svn proplist --verbose --revision 57 main.c
Properties on 'file:///repos/testrepo/trunk/source/c/main.c':
  Release : 3.2.4
```

The property `Release` was changed between revisions; in addition, a new property called `STATE` was added to the file. By having the properties versioned just like the content, you can get not only the source at a particular point but all the management aspects as well. For example, if you found a problem in revision 57 of the file `main.c`, you could quickly tell which release of the software was affected.

6.4 Getting the value of a property

You have seen the `proplist` command display the value of a property, but this is just a "dump" of everything attached to an object. Subversion also has a command called `propget` that will give you the value of a specific property. This is a cleaner and more elegant way of finding the value if you know the name of the property. With the `proplist` command, you will have to parse all the output to find the property you are looking for and then look through the line to find the value. While you can easily do this manually, it can be a pain if you are trying to accomplish it in a script. This is where the `propget` command comes into the picture. The `propget` command will give you only the value of the property you ask for, so no line parsing is required.

6.4.1 Running the propget

Since the `propget` command will return the value of a specific property, you will obviously need to include the name in the command line. This is the only

required argument, but as with the `proplist` command, you can also include the target object name. So to see the value of the `Release` property on the file `main.c`, you would run the following command:

```
$ svn propget Release main.c
4.4.2
```

This is fairly straightforward; you simply give the name of the property and the file or directory to run the command against. Subversion returns just the value. You can see how much easier this is for writing scripts or automating a process.

Directories. If you do not include an object as an argument, `propget` will use the current directory as the target:

```
$ svn propget Release
3.4.8
```

For the same reasons as the `proplist` command, `propget` will not recursively parse through the files and subdirectories.

6.4.2 Getting properties on multiple files

Consider a situation where you have released multiple versions of your software over some period of time. When a group of source code files is ready for a release, you will update their `Release` property. Now you want to see all the files that haven't been updated since release 3.1.1 so that they can be baselined. In a situation like this, you can use the `--recursive` switch on the `propget` command:

```
$ svn propget --recursive Release
. - 3.4.8
source/c/hello.c - 3.1.1
source/c/main.c - 3.2.5
```

In the previous example, we started in the source directory and added the `--recursive` option. We will still need to include the name of the property we are querying against. Since an object was not specified, Subversion will start at the current directory and work its way down. Each file that has the property `Release` assigned to it will be listed, followed by the value. You can now easily find all the files that have `3.1.1` as the value.

6.4.3 Getting older revisions

The `propget` command also has the ability to get a specific revision of the object from the repository as opposed to the current version in the working copy. The `--revision` switch works exactly the same as `proplist`; you specify one revision number (a range is not allowed). Since properties are versioned just like content,

this lets you see what the value of a property was for a revision. For example, consider that we are using the same process of tagging source files with the software release number as described in the previous section. Someone finds a bug in the file `main.c` that started in revision 57, and we are now at revision 61 of the repository. If you want to see all the releases of your software that are affected by this bug, you can get the value of the `Release` property for each revision of the file between 57 and 61. First, let's get the change log for the file:

```
$ svn log main.c
------------------------------------------------------------------------
r58 | jeff | 2004-05-18 20:25:49 -0400 (Tue, 18 May 2004) | 1 line

fixed checkstyle error
------------------------------------------------------------------------
r57 | jeff | 2004-05-18 20:24:40 -0400 (Tue, 18 May 2004) | 1 line

Was not releasing memory after malloc command.
------------------------------------------------------------------------
r30 | jeff | 2004-04-27 20:11:48 -0400 (Tue, 27 Apr 2004) | 1 line

Call the hello() function 10 times, not 5
------------------------------------------------------------------------
```

As you can see, we have two revisions of the file we need to look at: revision 57, which is the version the bug was introduced in, and a second version of the file at revision 58. We need run the `propget` command on both of these revision numbers to determine which release they were part of:

```
$ svn propget --revision 57 Release main.c
3.2.4

$ svn propget --revision 58 Release main.c
3.2.5
```

Revision 57 of the file is part of the 3.2.4 release of the application, and revision 58 is part of 3.2.5. This means that the bug found in the file will be a problem in these two releases of the code.

6.4.4 Removing extra characters from the output

Once again, the developers of Subversion thought about automation in the area of properties. The `propget` command has a switch called `--strict` that removes some of the formatting of the output. All newline and extra space characters are removed from the output. First, look at the output from the basic `propget` command:

```
$ svn propget --strict Release main.c
3.2.5$
```

Notice that there is no newline character placed after the value of the property. This formatting also holds for situations where you get multiple values, such as using the `--recursive` option. Let's add the `strict` option along with the `--recursive` option and see what happens:

```
$ svn propget --strict --recursive Release
3.1.13.2.5$
```

The two values are placed together because no spaces or newline characters are used between values. This is something you will need to be aware of when you are considering using this switch in a script or program.

6.5 Built-in properties

Subversion has a set of built-in properties that you can use along with your custom ones. While the properties we have discussed so far have no real meaning to the system, these built-in ones each have a specific purpose that Subversion will understand. These names of these properties all start with `svn:`, so you should avoid using this convention in your properties.

6.5.1 Ignore property

Back in chapter 3, we briefly touched on the `svn:ignore` property. This property gets attached to a directory and contains a list of files that Subversion should ignore. You will inevitably have non-versioned files in your working copy, and maybe log files, build directories, or configuration files. You will need these files for your development, but they will never go into the repository. You can tell Subversion to ignore these files if it sees them, so you will not have to manually deal with them. Consider the following directory to see how this will work:

```
$ ls -ltr
total 20
-rw-r--r--    1 jeff     svn          179 May 22 14:20 main.c
drwxr-xr-x    2 jeff     svn         4096 May 22 14:21 build
-rw-r--r--    1 jeff     svn           67 May 22 14:22 hello.c
-rw-r--r--    1 jeff     svn         1218 May 22 14:23 project.xml
-rw-r--r--    1 jeff     svn          275 May 22 14:23 helloworld.log

$ svn status
?      build
?      project.xml
?      helloworld.log
M      hello.c
```

You can see three objects in the directory that are not part of version control; two of them are files, `project.xml` and `helloworld.log`. The third object, `build`, is a directory. Let's assume that these three objects are going to be in the current working copy for the foreseeable future. Instead of having to look at these objects each time you run the `status` command, you can tell Subversion to ignore them. The easiest way to do this is to create a file and list all the objects you want to ignore. This file can be called anything you want; for now let's use `.svnignore` as the name:

```
$ cat .svnignore
.svnignore
helloworld.log
project.xml
build
```

Notice that we have included the three objects discussed earlier in the file, plus the name of the file itself. If you want to add the ignore file to version control, there is no problem in doing so. Now that we have our file set, we need to add the property to the directory. As with all properties, we do this with the `propset` command:

```
$ svn propset svn:ignore --file .svnignore .
property 'svn:ignore' set on '.'

$ svn commit --message "Added ignore to directory"
Sending        c

Committed revision 68.
```

There is nothing different between setting custom properties or built-in ones. We gave the property name, `svn:ignore`, instead of giving the value on the command line, and we passed in the contents of the file `.svnignore` using the `--file` option. Finally, we used the object ".", which is the current directory. We also remembered to commit since the changes are applied to the working copy object. Now just as with any property, we can use the `list` or `get` command to see what is attached:

```
$ svn proplist --verbose .
Properties on '.':
  svn:ignore : .svnignore
helloworld.log
project.xml
build
```

All of the objects we have included in the file are now listed as the value of the property. This means that Subversion will not consider them when viewing the status of an object. Let's see how the output of `svn status` looks now:

```
$ svn status
M      main.c
```

We have made a change to `main.c`, which rightfully shows up in the output. Notice that our non-versioned files are no longer in the output. Remember, if you want to force these to show up, you must add the `--no-ignore` option:

```
$ svn status --no-ignore
I      build
I      project.xml
I      .svnignore
I      helloworld.log
M      main.c
```

Now all the non-versioned files we were ignoring are back. Notice that they no longer include the `?` character. Since they are normally ignored, Subversion will tell you this by using the `I` character instead.

Ignoring directories. In the previous example, we added the directory `build` to the ignore list, but it was empty. Let's add some files to it and see if they are ignored also:

```
$ ls -l build
total 8
-rw-r--r--    1 jeff     svn          3173 May 22 15:26 hello.o
-rw-r--r--    1 jeff     svn          1365 May 22 15:27 main.o

$ svn status
M      main.c
```

Subversion also ignores the files in an ignored directory. This is an important point to remember: if a directory is in the ignore list, everything under that directory will be ignored also. This is really a big plus. Consider what we are using the `build` directory for. If this directory is going to contain all the derived objects from a build such as `.o` or `.class` files, we don't want to have to add them individually. By just adding the directory, all the objects that get created, even after the fact, inherit the ignored status.

Recursivly setting the ignore list. Remember that the `propset` command does not recursively add the properties by default. If you want to do this with the `svn:ignore` command, you will have to add the `--recursive` option to the command. Let's say your standard directory for storing compiled artifacts is the `build` directory, and you want to add this to be ignored in your entire repository:

```
$ pwd
/projects/testrepo
```

```
$ cat .svnignore
.svnignore
build

$ svn propset svn:ignore --recursive --file .svnignore .
property 'svn:ignore' set (recursively) on '.'
```

From the top level of the working copy, we recursively run the `propset` command with the ignore file. This will set the same ignore list on all the directories in the repository. But what about the `c` directory we set in the previous example? Let's take a look at that and see if our original ignore list is intact:

```
$ svn propget svn:ignore .
.svnignore
build
```

Just like the example in section 6.2.1, the property has been overwritten. You will need to be careful when using this switch that you do not blow away someone's ignore list. There is another potential problem with doing this. Say we add an object called `logfile` to the ignore list and apply it recursively. Now someone else comes along and adds a versioned file with the same name. They will not be able to see the file because it is being ignored. The moral of the story is that before deciding to apply a name to the ignore list globally, you must be sure no one needs to use that name for a versioned object.

Updating your ignore list. A common mistake people new to Subversion make is assuming that the file used, in our case `.svnignore`, just needs to be updated and the ignore list will be updated also. Such is not the case; the file is there to load the ignore list only when the `svn propset` command is run. This is different from CVS where merely updating the `.cvsignore` file will update the list. We will need to rerun the `propset` command for our changes to take effect. Take a look at the following sequence of commands to see how to update `svn:ignore`. We will add the file `project.xml` to the ignore list:

```
$ cat .svnignore
.svnignore
build

$ svn status
?      project.xml
M      main.c

$ echo project.xml >> .svnignore

$ svn propset svn:ignore -F .svnignore .
property 'svn:ignore' set on '.'
```

```
$ svn status
 M      .
M       main.c
```

Notice that the file `project.xml` showed up in the first status we ran. After we added it to the ignore file and ran the `propset` command, it was no longer there. Also, in the second status, the "." directory appeared as modified. For 10 points, why did this happen? Since the property of the current directory changed (the `svn:ignore` property), the metadata column showed the `M` for modified. Don't forget that you will have to commit to have your new ignore list saved to the repository.

Let's put a different spin on the previous example. We still want to add the `project.xml` file to the `svn:ignore` property, but we have not kept the original `.svnignore` file. You could re-create the file with the current value of the property plus your new object, but there is a better way. You can use the `svn propedit` command to accomplish the same thing. Simply use `svn:ignore` as the property name and "." as the path to the object:

```
$ svn propedit svn:ignore .
```

The current ignore list will be loaded in the editor, and now you can add your new object to the list. Once you have finished, save and exit the editor. You will see a message like the following, at which point you just need to commit and the ignore list will be updated:

```
Set new value for property 'svn:ignore' on '.'

$ svn status
 M      .
```

Pattern matching. So far, the `ignore` property seems to cover most of your situations easily; you can use it to block entire directories with only one entry in a list. You can ignore the same filename in your entire working copy by using the `--recursive` option. But what if you are in a situation where you want to ignore all log files, but they have different names? For example, you may use a naming convention like `module_name.date.log`. So you may have a file named something like `hello.05.17.04.15:21.log`. You obviously cannot put every possible name in the ignore list, but Subversion does have a way around this. As opposed to just using a static filename, the `svn:ignore` property can take advantage of pattern matching. So to fix the example you just saw, you could add an entry to the list like the following:

```
$ svn proplist -v .
Properties on '.':
```

```
  svn:ignore : .svnignore
build
*.log
```

Take a look at the last line in the output; it reads `*.log`. This is a standard expression, so all files that end in `.log` will be ignored. This is a powerful tool that will allow more generic ignored files, but you must be careful. If you make the expression too generic, you run the risk of excluding things that you do not want to exclude.

6.5.2 Keywords

In previous chapters, we showed how Subversion can give you information about a revision of file, such as who changed it last and when it was changed. Some people like to keep this information in the file contents, possibly in a header. This allows you to see information about the file outside of Subversion. For example, let's say you want to include the author of the last change in the header of a file. You might include something like the following:

```
$ head -10 main.c
#include <stdio.h>
#include "hello.h"

/*
 * Author: jeff
 *
 */
int main()
{
...
```

So what happens on the next revision when a different user changes the file? You will have to manually change the header or write some script to do this automatically. But you have probably guessed that since we are talking about it, Subversion provides a method for doing this. There is a set of five keywords for which Subversion will provide information inside the file. Each time the file is committed, these keywords will be substituted with actual information, such as the name of the author. The catch is that these keywords are not automatically active; you must turn them on by setting the `svn:keyword` property. Before we look at that mechanism, let's take a look at the five keywords and what they do in table 6.1.

Table 6.1 Keyword substitutions

Name	Alias	Description
LastChangedBy	Author	This will substitute the user ID of the user who committed the last revision of the file.
LastChangedDate	Date	The timestamp of the last commit.
LastChangedRevision	Rev	The revision number of the repository the last time the file was committed.
HeadURL	URL	The URL of the file in the repository.
Id	Id	All four of the previous keywords will be substituted in this one.

You can use the full name of the keyword or the alias inside the file to make the substitution. To see how this applies, let's first look at the author. Instead of hard-coding the author's name in the header, we will use the keyword encapsulated by `$`.

```
$ head -10 main.c
#include <stdio.h>
#include "hello.h"

/*
 * $Author$
 *
 */
int main()
{
```

Notice that we have added `$Author$` to replace the old hard-coded name. Remember, we said that this has to be activated by setting the property `svn:keywords`. The property will contain a space-separated list of keywords that we want to be active. So to just turn on the `Author` keyword, we will run the following `propset` command:

```
$ svn propset svn:keywords "Author" main.c
property 'svn:keywords' set on 'main.c'
```

Now that we have changed the contents of the file and added the keyword, go ahead and run a commit. Once you do this, take a look at the contents of the file and see what you get:

```
$ head -10 main.c
#include <stdio.h>
#include "hello.h"

/*
 * $Author: jeff $
```

```
 *
 */
int main()
{
```

Now the header contains the ID of the user who made the last change. Subversion is also kind enough to add the label "`Author: `," so you do not need to include it. Each time someone commits, their username will be substituted in this field. Now let's set all the keywords in the header of the file and see what they look like:

```
$ svn propset svn:keywords \
"Author Date Rev URL Id" main.c
property 'svn:keywords' set on 'main.c'

$ head -10 main.c
#include <stdio.h>
#include "hello.h"

/*
 * $Author$
 * $Date$
 * $Rev$
 * $URL$
 * $Id$
 */

$ svn commit --message "added keywords"
Sending        c/main.c
Transmitting file data .
Committed revision 78.
```

Okay, now we are ready to see all the keywords in the header. Once you do the first commit after adding these keywords, they will be in the file:

```
$ head -10 main.c
#include <stdio.h>
#include "hello.h"
/*
 * $Author: jeff $
 * $Date: 2004-05-22 21:02:26 -0400 (Sat, 22 May 2004) $
 * $Rev: 78 $
 * $URL: file:///repos/testrepo/trunk/source/c/main.c $
 * $Id: main.c 78 2004-05-23 01:02:26Z jeff $
 */
```

All the change information has replaced the keywords. Each time the file is changed and committed, this information is updated.

Keywords on the diff. Let's say we make a change to the file and then run a commit. Look at the header information compared to previous output to see what has changed:

```
$ head -10 main.c
#include <stdio.h>
#include "hello.h"

/*
 * $Author: alex $
 * $Date: 2004-05-22 21:11:21 -0400 (Sat, 22 May 2004) $
 * $Rev: 79 $
 * $URL: file:///repos/testrepo/trunk/source/c/main.c $
 * $Id: main.c 79 2004-05-23 01:11:21Z alex $
 */
```

As you can see, the header information was substituted accordingly. So let's run a diff and see what the output shows:

```
$ svn diff --revision 78:79 main.c
Index: main.c
===================================================================
--- main.c      (revision 78)
+++ main.c      (revision 79)
@@ -13,7 +13,7 @@

   int index ;

-  for ( index=0 ; index<10 ; index++ )
+  for ( index=0 ; index<5 ; index++ )
   {
     hello();
   }
```

The only difference that shows up with the `diff` command is that the content changed; nothing from the keywords changed. Subversion will see these fields as the keywords, so internally they will look the same from one revision to another.

6.5.3 Executable property

When you checkout or update a working copy in Subversion, it will use the default file permissions, not necessarily what the permissions were before. Let's say we have a script called `run_main.sh` that has the `execute` permission set on the OS level. Now look at what happens when we run an update:

```
$ ls -l
total 12
-rw-r--r--    1 jeff     svn            67 May 22 21:30 hello.c
-rw-r--r--    1 jeff     svn           387 May 22 21:30 main.c
-rwxr-xr-x    1 jeff     svn            12 May 22 21:30 run_main.sh

$ svn up
U  run_main.sh
```

```
Updated to revision 82.

$ ls -l
total 12
-rw-r--r--    1 jeff     svn            67 May 22 21:30 hello.c
-rw-r--r--    1 jeff     svn           387 May 22 21:30 main.c
-rw-r--r--    1 jeff     svn            19 May 22 21:32 run_main.sh
```

Notice that after we did the update, the execute permissions are gone from the script. It can become very cumbersome to continually fix file permissions each time you run an update. To get around this, Subversion has a property called `svn:execute` that will keep the execute permission set on the file when you get the object from the repository. Subversion does not care what the value of the property is, as long as it is attached to the file:

```
$ svn propset svn:executable true  run_main.sh
property 'svn:executable' set on 'run_main.sh'

$ svn commit --message "Added executable property"
Sending        c/run_main.sh

Committed revision 85.
```

Now that this property is set on the file, each time you do a checkout or an update, the file will retain the execute permissions. Even if the permissions start out wrong, they will be corrected when you update:

```
$ svn up
U  run_main.sh
At revision 85.

$ ls -l
total 12
-rw-r--r--    1 jeff     svn            67 May 22 21:30 hello.c
-rw-r--r--    1 jeff     svn           387 May 22 21:50 main.c
-rwxr-xr-x    1 jeff     svn            21 May 22 21:48 run_main.sh
```

This time when we ran the update, the execute permissions stuck to the file. Setting the `svn:execute` property will prevent you from having to manually set the permissions.

6.5.4 End-of-line style

If you have ever worked on files in both UNIX and Windows, you have likely run into a situation where you get extra characters at the end of a line. If this happens, your output will look something like the following:

```
public class hello^M
{^M
    public hello( )^M
    {^M
    }^M
}^M
```

Subversion will allow you to force the end-of-line (EOL) character instead of letting the operating system of the client you are committing from dictate this. Table 6.2 describes the different characters you can use to set the `svn:eol-style` property.

Table 6.2 EOL characters

Value	Description
native	Use the default of the operating system from which you are running the commit.
LF	Use only the LF character. This should be used for UNIX-based operating systems.
CR	Use only the CR for the EOL character.
CRLF	Use the CR followed by the LF character. This should be used for Windows-based systems.

To set the EOL character, use the `propset` command with the property named `svn:eol-style`. This will force the EOL character for that file. Let's see if we can fix the file `hello.java` by setting the correct EOL character:

```
$ svn propset svn:eol-style LF hello.java
svn: File 'hello.java' has inconsistent newlines
svn: Inconsistent line-endings in source stream
```

If the current characters for the end-of-line character are mixed, Subversion will not be able to set the property. In this case, we must first correct the file. You can do this in your favorite editor by removing the extra character at the end of each line. Once this is done, you can run the command again:

```
$ svn propset svn:eol-style LF hello.java
property 'svn:eol-style' set on 'hello.java'
```

From this point on, no matter what operating system you commit from, Subversion will force the EOL character to be in the UNIX format.

6.5.5 Externals

As we have been working along in the `java` and `c` directories, another repository has been holding some `php` code. We'll assume that these repositories need to remain separate. Perhaps they are on separate filesystems on the OS, and there is

no room to combine them. You can actually point different directories in a working copy to separate repositories. Let's say we want to add the `php` directory to the source in our working copy, but it resides in a repository called `repo2`. You can add a property in the source directory with a name of `svn:externals`. The value will be two text strings; the first is the name of the directory that will be mounted in the working copy. The second string is the repository URL the mount point will be pointing to. The following commands illustrate how to set this up:

```
$ ls
c  java

$ svn propset svn:externals "php file:///repos/repo2/php" .
property 'svn:externals' set on '.'

$ svn commit --message "Added external definition for PHP directory"
Sending        source

Committed revision 98.

$ svn up

Fetching external item into 'php'
A  php/main.php
Updated external to revision 2.

Updated to revision 98.


$ ls
c  java  php
```

Notice that when we run the update, two revision numbers are displayed. This is because the working copy is pointing to two different repositories. When we run the `info` command on the `php` directory, we will see where it points:

```
$ svn info php
Path: php
URL: file:///repos/repo2/php
Repository UUID: bd6efcc4-0edb-0310-9b56-bc086ee89c02
Revision: 2
Node Kind: directory
Schedule: normal
Last Changed Author: jeff
Last Changed Rev: 2
Last Changed Date: 2004-05-22 23:08:02 -0400 (Sat, 22 May 2004)
```

Take a look at the URL the `php` directory points to; it is a different repository than ours. When you make changes in this directory, the directory will actually be saved to the repository `repo2`.

6.6 Revision properties

All the properties we have examined so far are attached to the entire file or directory. If you leave the property alone, it will be the same for every revision of the file. For example, if you set the property `Release` to `3.4.2` at revision 20 of the file `main.c` and don't change it, all the revisions will have the same property. Subversion has other properties that are attached not to the object but to individual revisions. To access these revision properties, you will need to add the `--revprop` option to any of the property commands. In addition, you will have to specify a revision number on which to perform the operation. Just like the regular properties, there are custom properties and default ones.

6.6.1 Default revision properties

Subversion has three default revision properties attached to each version of a file. These are `author`, `log`, and `date`. If you stop and think about this, it is all the information in the change logs. This is just another way to access the change information for a specific revision of a file. Let's take a look at an example of these default revision properties:

```
svn:log : Removed extra println on exit
  svn:author : jeff
  svn:date : 2004-05-23T03:52:01.195684Z
```

You can see all of the change log information here, along with the property names.

6.6.2 Adding properties

Since these properties are attached to a specific revision, they are not versioned. This means that when you make a change to one of them, there is no way to get things back. Think back to when we changed the log message; these properties are in the same category. In fact, the log message is actually one of these properties. Let's first try to set a custom property:

```
$ svn propset --revprop --revision HEAD Label test main.c
svn: Repository has not been enabled to accept revision propchanges;
ask the administrator to create a pre-revprop-change hook
```

Just as when we changed the log message, a script is required to be run before this command executes. This is Subversion's way of giving you an opportunity to do a

sanity check, or possibly save the old information before executing the command. This is important because there is no way to get the information back. In order for Subversion to allow you to change these properties, you must have a hook script called `pre-revprop-change`. You can use the template in the hook directory as a start:

```
$ cd /repos/testrepo/hooks/

$ cp pre-revprop-change.tmpl pre-revprop-change

$ chmod a+x pre-revprop-change
```

At this point, you should be able to set a property on the file:

```
$ svn propset --revprop --revision HEAD state production main.c
property 'label' set on repository revision '99'

$ svn proplist --revprop --revision HEAD main.c
Unversioned properties on revision 99:
  state
  svn:log
  svn:author
  svn:date
```

Now we are able to set the custom property on the file `main.c`. While the script just needs to exit with a code of `0`, it may not hurt to at least log the changes so that you have a record of what was changed.

6.6.3 *Viewing revision properties*

You can view the revision properties just like any other ones, except that you must add the same options we used in the previous example. To view all the properties and their values, you can run the `proplist` command:

```
$ svn proplist --verbose --revprop --revision 99 main.c
Unversioned properties on revision 99:
  state : production
  svn:log : Removed extra println on exit
  svn:author : jeff
  svn:date : 2004-05-23T03:52:01.195684Z
```

The only difference between running the `proplist` command against the revision properties versus regular properties is the `--revprop` switch. This is the same for the `propget` command:

```
$ svn propget --revprop --revision 99 svn:author main.c
jeff
```

6.6.4 Changing the properties

Let's say you need to change the author of a file; maybe you checked it in for your coworker and now you need to adjust the property to the correct value. This can be done with the `propset` command:

```
$ svn propset --revprop --revision 99 svn:author alex main.c
property 'svn:author' set on repository revision '99'

$ svn log main.c
------------------------------------------------------------------------
r100 | jeff | 2004-05-23 00:43:24 -0400 (Sun, 23 May 2004) | 1 line

Fixed checkstyle errors
------------------------------------------------------------------------
r99 | alex | 2004-05-22 23:52:01 -0400 (Sat, 22 May 2004) | 1 line

Removed extra println on exit
------------------------------------------------------------------------
```

Now when we run the log, the new user shows up in the history. Remember, these properties point to the same location as the change logs, so the new author is reflected.

Using the edit. You can use the `propedit` command to change any of the revision properties if they are large, instead of using `propset`. You may need to do this on something such as a log message that tends to have a large size:

```
$ svn propedit --revprop --revision 99 svn:log main.c

Removed extra println just before the exit.
This will be done in the hello() function, so we
can remove it from the main.
~
~
~
"svn-prop.tmp" 3L, 122C written

Set new value for property 'svn:log' on revision 99
```

Once you save this in your editor, the property will be set. Just remember that you will have to commit to get these changes in the repository.

6.7 Summary

After working through branches and properties, you may be noticing a pattern with Subversion's implementation of its tools. These features have always been the most complex part of other version control systems, but this is quite the opposite

in Subversion. They have been simplified, which has increased the number of ways to use them. Since these areas of version control are really ways to manage your process, Subversion will easily fit into any process you have. While this sounds great and could be used in a marketing plan, it is a double-edged sword. It requires you to actually develop a process and implement it. Properties are a typically an underused piece of Subversion—or any other version control system for that matter. You will hear things like, "I'll get to that when I have time." Using properties to manage your environment and process should not be an afterthought. If you can develop a solid process and automation around the tool, it can actually make your development better and faster. You will get the right things into production and be able to quickly troubleshoot when problems do arise.

7 Repository administration

In this chapter

- Repository backup and restore
- Configuring network access with svnserve
- Configuring network access with Apache

Just like any version control system, Subversion requires a certain amount of repository administration. The amount of work depends on how you use the repository (local or over the network), how static your environment is, the size and skill of your user base, and your backup needs. You do not need to master the material in this chapter if you are just starting out learning Subversion. However, before you start loading your source code and using it, you should understand all the concepts so Subversion will perform the way you expect.

7.1 Backing up your repository

This is one the most important administrative functions, for obvious reasons. No matter if you have one user or a thousand, you will want to back up the files you are storing in Subversion. There are a couple different ways to back up your information.

7.1.1 Dumping the repository

Subversion lets you dump the entire repository, including change information, to a file. When you do this, the contents, change logs, and properties are all part of the output. You can take this file and move it anywhere on your system or to a different system. You can then use the file to load a new repository, giving you an exact copy. To dump your repository, use the `svnadmin dump` command. This command will need only the `PATH` to the repository; the output will be sent to `STDOUT`, so you will have to redirect it to a file. Take a look at the following example to see how this is done:

```
$ svnadmin dump /repos/testrepo > testrepo.dump
* Dumped revision 0.
* Dumped revision 1.
* Dumped revision 2.
* Dumped revision 3.
* Dumped revision 4.
...
* Dumped revision 99.
* Dumped revision 100.
```

The progress will be sent to `STDERR`, so even though the dump is going to a file, you can see what the command is doing. Subversion will start with the first revision and apply changes for each revision after that. This way, each revision dumped is not an entire copy, which will make the file a more manageable size. Now you can back up this file to tape or store it on another machine. Then if you should have a disaster, you can create a new repository with the same name and load this file into it. We will show how to do this later in the chapter.

Getting a set of revisions. If your repository is large in terms of disk space or number of revisions, the `dump` command can take a good deal of time to run. If you are doing a dump nightly, you will be continually dumping revisions that you already got the night before. The `dump` command has the ability to dump a range of revisions, which will allow you to reduce the number of versions in your file. The first version dumped is a complete copy, so it doesn't matter where you start. You could keep track of the last revision dumped the night before and use that as the starting point for your range, which will end with the latest revision:

```
$ svnadmin dump --revision 80:HEAD /repos/testrepo > testrepo.dump.80-HEAD
* Dumped revision 80.
* Dumped revision 81.
* Dumped revision 82.
* Dumped revision 83.
...
* Dumped revision 97.
* Dumped revision 98.
* Dumped revision 99.
* Dumped revision 100.
```

With this command, revision 80 will be a complete copy of the repository, and then starting with revision 81, the dumps will be just the deltas. The next night you would start with revision 101 and go to the new `HEAD` for the dump. This will alleviate the need to dump revisions 1–80 over and over when you already have a copy of them.

Incremental dumps. Dumping a range of revisions will save you some time because you do not need to parse through all the revisions in the repository, but it will not save you any space. Remember, the first revision is a full copy of the repository, so each night you will get an entire copy of the repository; it will just start at a later revision. To get around this, Subversion lets you do incremental dumps. When you choose to do this, the `dump` command will assume that you have the previous revision in another dump file, so it will not make a complete copy of the first revision in the range. The only caveat is that if you need to restore, you will need to load in all the dumps in sequence. This will allow you to have a backup methodology similar to that of a database. To change the behavior of the `dump` command to be incremental, just add the `--incremental` switch to the command. Let's say we want to create three dump files to keep them small. We could run the following three commands:

```
$ svnadmin dump --quiet --revision 0:30 \
/repos/testrepo > testrepo.dump.0-30
```

```
$ svnadmin dump --quiet --revision 31:60 --incremental \
/repos/testrepo > testrepo.dump.31-60

$ svnadmin dump --quiet --revision 61:100 --incremental \
 /repos/testrepo > testrepo.dump.61-100
```

First, notice that we have included an extra option, `--quiet`. This will simply prevent the command from printing all the progress statements that show which revisions have been dumped. Since we added the `--incremental` option to revision 31 in the second command and revision 61 in the third, they will not be complete copies. This means that we cannot load `testrepo.dump.31-60` by itself; it will require the first dump file. The advantage, however, is that the space required for the three files will not be as large. This trade-off of space saved versus keeping track of multiple files is dependent on your system. The larger your repository gets, the more it makes sense to use the incremental backup.

A backup strategy. The backup strategy using the incremental dumps does not need to be complex. Consider the following template. On the first day of the week, you take a full dump of the repository. Each of the six days following, you dump only the revisions using the `--incremental` flag. If you store each of the files in a separate subdirectory and label them properly, it should be easy to keep track of them in case they need to be reloaded sequentially. At the end of the week, you start over with a new full dump. This way, in a worst-case scenario, you will need to load only seven files.

Let's say we create two scripts, one for full backups and one for incremental. We would first set up the full backup to be run once a week, maybe on Sunday. The incremental backup will run Monday though Sunday. The following scripts are very basic examples of what you could use for the backups. These scripts are written for a Linux host using the `bash` shell. Don't worry about the details of the script; just follow the overall logic. First, we will look at the full backup script that will be run on the first night of the week:

```
#!/bin/bash
DUMPDIR=/repos/svndumps

# Remove old dump files
cd $DUMPDIR/archive
rm -rf *.dump

# Archive last weeks backups
cd $DUMPDIR
mv *.dump archive 2>/dev/null

# Take a full backup
```

```
echo "Starting full backup" >> logfile
date >> logfile

svnadmin dump /repos/testrepo > testrepo.full.dump 2>>logfile
```

Since we are starting on a new week, we save all the files from the previous week, just in case. Once that is complete and the directory is empty, we run the `dump` command on the entire repository and save it into the file `testrepo.full.dump`. Now, for the rest of the week, we will run the incremental script that may look something like the following:

```
#!/bin/bash
DUMPDIR=/repos/svndumps

# Get the last revision dumped
cd $DUMPDIR
VER=`cat logfile | grep -i "dumped revision" | tail -1`
VER=`echo $VER | awk '{print $NF}' | cut -d\. -f 1`
((VER=$VER+1))

# Dump the last one plus the head
svnadmin dump --revision $VER:HEAD /repos/testrepo \
> testrepo.tmp.dump 2>>logfile

# Now get the HEAD and rename the file
# to reflect the range
NEWVER=`cat logfile | grep -i "dumped revision" | tail -1`
NEWVER=`echo $NEWVER | awk '{print $NF}' | cut -d\. -f 1`
FILENAME=`echo testrepo.${VER}-${NEWVER}.dump`
mv testrepo.tmp.dump $FILENAME
```

We are getting the last revision dumped from the `logfile`. The only trick to this method of backup is that you need to keep track of the last revision you backed up. Once we get the last revision, we run the `dump` command with a range starting at the last revision and going to the `HEAD` keyword. We then change the name of the dump file to reflect the range it holds. After a full week, our directory will look something like this:

```
$ ls -1tr
archive
testrepo.full.dump
testrepo.101-102.dump
testrepo.103-115.dump
testrepo.116-120.dump
testrepo.121-132.dump
testrepo.133-138.dump
testrepo.139-145.dump
logfile
```

Should you need to restore from these files, you would just load them starting with the full dump and then in sequence by the ranges. This process may seem very tedious, but it is a small price to pay to ensure that your repository has a solid backup.

7.1.2 Loading from a dump file

Now you have a great backup strategy in place, but this will not do you a lot of good unless you can restore the files. To restore your dump files into a repository, you use the `svnadmin load` command. You give the command the path to the repository that will be loaded, and it will get the dump information from `STDIN`. Just as we redirected the output from the `dump` command to a file, we will reverse this process and redirect the file into the `load` command. For now, let's assume that we have a complete dump in one file. First, we will create a new repository in which to load the file, and then we will run the command:

```
$ svnadmin create /repos/testrepo_restore

$ svnadmin load /repos/testrepo_restore < testrepo.full.dump
<<< Started new transaction, based on original revision 1
     * adding path : source ... done.
     * adding path : source/c ... done.
     * adding path : source/c/hello.c ... done.
     * adding path : source/c/main.c ... done.
     * adding path : source/java ... done.
     * adding path : source/java/hello.java ... done.
     * adding path : source/java/main.java ... done.

------- Committed revision 1 >>>

<<< Started new transaction, based on original revision 2
     * editing path : source/java/main.java ... done.

------- Committed revision 2 >>>

<<< Started new transaction, based on original revision 3
     * editing path : source/java/hello.java ... done.

...

<<< Started new transaction, based on original revision 100
     * editing path : trunk/source/c/main.c ... done.

------- Committed revision 100 >>>
```

This will parse the input we redirected in and create each revision, one at a time. You will be able to see for every revision what the `load` is doing for each file. Once this process is complete, you can check out the repository under the new name:

```
$ svn co file:///repos/testrepo_restore
A  testrepo_restore/trunk
A  testrepo_restore/trunk/source
...
A  testrepo_restore/trunk/source/c
A  testrepo_restore/trunk/source/c/run_main.sh
A  testrepo_restore/trunk/source/c/hello.c
A  testrepo_restore/trunk/source/c/main.c

Checked out revision 100.
```

Even though this is a different repository name and location, the contents, including versions, history, and properties, will all be present. If you have a current working copy, you can either check out a new one or use the `svn switch --relocate` command to point to the new URL. You saw an example of this in chapter 5.

Loading multiple files. The process of restoring from multiple files is a simple repetition of the process we just used. After you load the full backup, you can load each of the incremental backups in order. Just remember that they are built off each other, so the sequence is crucial. We have already loaded the full backup; now let's load one of the incremental files:

```
$ svnadmin load repos/testrepo_restore < testrepo.101-102.dump
<<< Started new transaction, based on original revision 101
     * editing path : trunk/source/c/hello.c ... done.

------- Committed revision 101 >>>

<<< Started new transaction, based on original revision 102
     * editing path : trunk/source/c/main.c ... done.

------- Committed revision 102 >>>
```

This looks just like the initial load except that the revision number starts at 101. You can repeat this step for the rest of the incremental files from the backup. Once you have them all loaded, the new repository will be up-to-date.

Starting at a new location in the repository. Until now, we have assumed that the new repository will look identical to the original, but this does not have to be the case. In fact, the dump-and-load mechanism isn't limited to backups and restores. You can use this mechanism to move the contents of one repository to another. When this is the situation, you may want to start the load in a new location, not necessarily at the top level. All you need to do for this is include the directory in an option called `--parent-dir` on the command line. This will take a directory path, starting from the top level of the new repository. For example, say you have a directory called `old_data/testrepo` that you will want to be the target for the `load`:

```
$ svnadmin load /repos/testrepo_restore \
--parent-dir old_data/testrepo < testrepo.full.dump

...
<<< Started new transaction, based on original revision 100
     * editing path : old_data/testrepo/trunk/source/c/main.c ... done.

------- Committed new rev 102 (loaded from original rev 100) >>>
```

This will run the same `load` command as before, except it will start in the new target directory. When you load into a repository that is not new, the revision numbers will start at the current number, not where the dump started. So in this example, we actually end with revision 102 instead of 100. You can see in the `commit` statement from the output how these get correlated. Without this capability, Subversion would not be able to load into a preexisting repository.

7.1.3 Creating a hot backup of a repository

Instead of running through the dump-and-load process, you can create a new repository "on the fly" in one step. The advantage of this method is that you do not have worry about keeping track of files, and it can be run while there is activity on the repository. Once executed, the command will take a snapshot of the source repository. You can check out the copy and interact with it, but once the copy is made, the two repositories are separate, so changes in one will not reflect in the other. To create this copy, you use the `svnadmin hotcopy` command:

```
$ svnadmin hotcopy /repos/testrepo /repos/testrepo_backup
```

The target repository must not exist when you execute the command. Subversion will create the repository and do the copy all at once.

When to use hotcopy. The `hotcopy` command is another tool at your disposal for backing up your repository; there is no one right way to use it. We already talked about some of the advantages of using this command over doing a dump and load. There are some potential drawbacks to using it, however. If this is your only backup, what will happen if the host machine dies? You backup will be unusable also.

One thing to remember is that these methods are not mutually exclusive; they can be used together. You can easily make your backups a two-step process. First, you can run a `hotcopy` of your production repository. This is a quick process (relative to the dump), and you do not need to worry about keeping the main repository down for long periods of time while running the dump. Once the copy is complete, the production repository can go on its merry way with the core development. In the meantime, you can dump from the backup copy, which is just

a snapshot; you would have run this from the production copy anyway, so the dump will be the same. The advantage is that you needn't care how long this process takes. Once the dump is done, you can move it to tape or a different host for redundancy. Now if there is a failure with the primary repository, you can just `hotcopy` the backup into production. If there is a total failure on the host, you can rebuild the repository with the dump files.

All of these choices depend on how critical your data is—can you afford to lose a day's work? Also, you need to ask yourself how long you can afford to be down if you have a hardware problem. Once you start answering these questions, you can apply a backup strategy that fits your needs, using some combination of the tools we just explored.

7.1.4 Exporting a non-versioned copy

There is one more method of getting content out of the repository—the `svn export` command. Notice that this is part of the user client, not the admin. The `export` command is the reverse of the `import` command we talked about way back in chapter 2. Remember that the `import` will take a non-versioned directory tree and bring it into the repository. Well, the `export` command will create a non-versioned tree from the repository. You will want to use this command when you need to provide someone a snapshot of the source code that will not have the ability to update.

Consider an organization that packages source code with its distribution. When you load the source code on the CD that will be going out the door with the binaries, you need a snapshot of the repository at the point at which you built the software. At the same time, you would not want to check out a working copy and include that because it would contain all the `.svn` directories, which would just confuse the end user. So just run the `export` command, and your problem is solved. Let's first run the command from the repository to a local directory:

```
$ svn export file:///repos/testrepo /home/jeff/testrepo_snapshot
A  /home/jeff/testrepo_snapshot
A  /home/jeff/testrepo_snapshot/trunk
A  /home/jeff/testrepo_snapshot/trunk/source
A  /home/jeff/testrepo_snapshot/trunk/source/c
A  /home/jeff/testrepo_snapshot/trunk/source/c/hello.c
A  /home/jeff/testrepo_snapshot/trunk/source/c/main.c
A  /home/jeff/testrepo_snapshot/trunk/source/c/LdapAdd.java

Exported revision 104.
```

We now have a directory created that contains the repository. Look at what happens next when we try to run the `update` command:

```
$ cd /home/jeff/testrepo_snapshot/

$ svn up
svn: '.' is not a working copy
```

Just as we suspected, this is not a working copy, so we cannot run any of the Subversion commands from it. Also, there are no `.svn` directories, so it will appear to the user that this is just a regular directory structure.

Exporting a piece of the repository. As with any Subversion command that gets information from the repository, you can specify which revision you want. Sticking with the scenario that we are including the source code on a distribution, we may not want the `HEAD`. In addition, the release we are including the code with may be a branch and not in the trunk. We will not want to include all the development branches and project layout. We just want to give the user the source code starting at the top level of the code, not the repository. Let's say we have a branch that contains the source, and we want revision 105:

```
$ svn export --revision 105 \
file:///repos/testrepo/branches/release-2.0/source \
 /home/jeff/testrepo_snapshot

A  /home/jeff/testrepo_snapshot
A  /home/jeff/testrepo_snapshot/c
A  /home/jeff/testrepo_snapshot/c/hello.c
A  /home/jeff/testrepo_snapshot/c/main.c
A  /home/jeff/testrepo_snapshot/c/LdapAdd.java

Exported revision 105.
```

All we did in this example was to add the `--revision` option and give more detail in the repository URL.

7.2 Setting up network access with svnserve

So far, the majority of the access to the repositories has been on the local filesystem, but this will not be the case in the real world. Logistically, it is difficult to get all your users on one host. Even if you could do this, many developers like to work on their own systems anyway. Also, for security purposes, you may not want users on the repository host machine. Remember, in order for them to be able to access the repository locally, they must have write permissions to the database files, which you may not want. Back in chapter 1 we talked about the different network

protocols. If you want quick and dirty, you can use the svnserve process, which is a custom protocol that ships with Subversion. While it is not feature rich, you can use this protocol to get going until you need the Apache server.

7.2.1 Choosing between svnserve and Apache

So how do you decide which server to use for the network protocol? Let's go through a few guidelines that will help you with this choice. The good news is your decision does not have to be permanent. You can switch protocols should your needs change at any time. Or, should the need arise, you could use multiple protocols. Once you answer the following questions, you should have a good idea of which server will work for you.

Do you already have Apache running? If you are currently running Apache, it may seem like a good idea to piggyback the Subversion access with your current web server. However, this can actually cause more problems for you than it provides convenience, depending on your setup. For example, if you need to run Apache as a certain user, that ID will need to have permissions to the repository directory. It is possible that other configuration options will also conflict. To get around this conflict, you will need to run a second instance of Apache, which will require a different port to listen. While this is completely viable, you will have to make sure that the port is specified in each client call. For example, if you set up the second instance of Apache on port 900, the URL in a command would have to be something like the following:

```
svn co http://wiley.mydomain.com:900/repos/testrepo
```

There is an upside to using Apache for Subversion if you are already running it, which is expertise. Chances are you, or someone in your organization, has a good understanding of configuring and running the web server. This can be an advantage by cutting through any learning curves or ramp-up time to get Subversion on the network.

What are your security requirements? Your security requirements can also help guide you to the correct network protocol. While both protocols communicate using encryption, they do it in different ways. The svnserve process uses SSH provided by the operating system. The advantage of this is that the security will be maintained at the operating system level with SSH. This would likely need to be done anyway to support users logging into the system. If you need more granularity in this area, you will need to use Apache. As you will see later in the chapter, you can create your own keys and even have individual client certificates.

In additional to encryption, you may also have constraints with a firewall or open ports. Since the svnserve process listens on its own port, you may have a hard time getting this port open on a hardened network. Since most networks already have port 80 open, network security may be easier to work with if you use Apache.

User support. Users and groups are supported both in Apache and with the svnserve protocol, but depending on how complex this arrangement is, there will be some differences. If you have a small, relatively static user base, it will not be a chore to maintain. But if you are in a large, open source community where users need to be able to change their own password, Apache has the built-in security to maintain a password file.

In addition to the number of users, the level of complexity of your access control will be a factor in your decision. If you only need to restrict user groups to read or write access for the entire repository, you can quickly set up svnserve to do this. If your environment requires directory-level access control lists, you will need to use Apache.

Let's continue looking at the svnserve protocol in more detail. When we are finished with that, we will show how to configure Apache for the network protocol.

7.2.2 Running the default configuration

When you create a repository, a standard configuration is set up for the svnserve process. This will allow only read access to the repository. We will see how to change this configuration in just a little bit, but for now let's just get the process running. First, make sure you have a user who can run this process. If you are on a UNIX-based machine, do not run this process as `root`, because it would open up far too many security holes. When we set up the repository back in chapter 2, it was suggested creating a Subversion user called `svn`. We will use this ID to run the svnserve process. While it does not matter which user ID runs the process, it must have read and write access to the repository filesystem. To start the process, log on as the user and run the following command:

```
$ svnserve -d
```

This will run the process as a daemon and bring you back to the command prompt. The svnserve daemon will use port 3690 to communicate, so if your clients are going through any kind of firewall, you will have to make sure that it is open. Now from another machine, you can run Subversion commands over the network to access this repository. Remember, since we are using the `svn` protocol, the URL will now start with `svn:` instead of `file:`. Assume that the machine name

of the repository host is `wiley` and that it is the same path we have been using. You would use the following URL to check it out:

```
$ svn co svn://wiley/repos/testrepo
A  testrepo/trunk
A  testrepo/trunk/source
A  testrepo/trunk/source/java
...
A  testrepo/tags/release_1.0/source/c/main.c
U testrepo
Checked out revision 110.
```

Notice the URL of the `checkout` command: instead of just the path, we added the hostname `wiley`. Keep in mind that we are assuming the client knows the IP address of `wiley`. You may need to add the domain name or even the IP address. The following examples of URLs are valid and are merely different ways to reference the same host we just saw:

```
$ svn co svn://wiley.mydomain.com/repos/testrepo

$ svn co svn://192.168.1.10/repos/testrepo
```

The hostname you will use depends on how your DNS is set up, but the command will accept any format your system can resolve.

Default repository access. There is no difference in the action of the command using this protocol or using the local access method with `file:` that we have been using for most of the book. If you have not modified the configuration for the svnserve process yet, you will not be able to write to the repository. If you change a file and try to commit, this is what you will get:

```
$ svn commit --message "Added debug statements"
svn: Commit failed (details follow):
svn: Connection is read-only
```

Since the svnserve process will not allow write access by default, you cannot commit anything to the repository. If you wish to change this, you will need to change the authorization component of the configuration.

7.2.3 Svnserve configuration file

Before we start looking at the various tuning options, you must understand the configuration file for svnserve. It is a relatively small file and does not have a ton of options, so you should be able to get through it pretty quickly. First, each repository has its own configuration for svnserve in the directory `conf`, under the repository directory. Let's look at the configuration location for our repository:

```
$ cd /repos/testrepo/conf
$ ls -l
total 12
-rw-r--r-- 1 svn svn 1078 May 31 13:07 svnserve.conf
```

In order to make modifications to the configuration, you must have write access to this file. Again, you can see the advantage of having an ID for repository ownership and administration. Let's take a look at the contents of the file on a new repository:

```
$ cat svnserve.conf
### This file controls the configuration of the svnserve daemon, if you
### use it to allow access to this repository.  (If you only allow
### access through http: and/or file: URLs, then this file is
### irrelevant.)

### Visit http://subversion.tigris.org/ for more information.

# [general]
### These options control access to the repository for unauthenticated
### and authenticated users.  Valid values are "write", "read",
### and "none".  The sample settings below are the defaults.
# anon-access = read
# auth-access = write
### This option controls the location of the password database.  This
### path may be relative to the conf directory.  There is no default.
### The format of the password database is:
### [users]
### USERNAME = PASSWORD
# password-db = passwd
### This option specifies the authentication realm of the repository.
### If two repositories have the same authentication realm, they should
### have the same password database, and vice versa.  The default realm
### is the path to the repository, relative to the server's repository
### root.
# realm = My First Repository
```

This looks like any INI configuration file. The lines that start with # are comments and not part of the actual configuration. As you can see, the default implementation of the file has everything commented out. If you look carefully, you will see that some lines have three # characters and some have only one. The lines with just one are default configuration settings; the other lines are meant to give additional information. Let's take out all the extra lines and see what we are really dealing with:

```
[general]
anon-access = read
auth-access = write
password-db = passwd
realm = My First Repository
```

```
### [users]
### USERNAME = PASSWORD
```

There are two types of entries in the configuration file; the first is section headers. These are enclosed in [], and there are two possibilities for svnserve. The first is `general`, which will basically include all the configuration settings about the behaviors of the process. The second section is `users`. This will allow you to define Subversion users inside the configuration file. The second type of entry in the file is a setting. This is where you define the values for specific settings. For now, we will assume that user information is not stored in this file, but we will address this topic soon. Table 7.1 describes what each of the settings means. The sections following describe how to apply the settings.

Table 7.1 Svnserve configuration settings

Name	Value	Description
anon-access	none, read, write	Determines the level of authorization an anonymous or guest user has without any authentication.
auth-access	none, read, write	Determines the level of authorization a user has after authenticating to the svnserve process.
password-db	filename	Points to the file on the repository host that stores the Subversion user's ID and password.
realm	string	Tells the svnserve process what realm it is a part of. This allows multiple repositories to share one user database.

This configuration is dynamic, so when you save the changes to file, they will immediately take effect without you restarting the process.

7.2.4 Setting up authorization

We just showed that, by default, the svnserve process will not allow writing to the repository. Most people will need to change this access, but there are different ways to do so, and you will have to choose which method works for you. Table 7.1 showed the two types of authorization categories; the first is anonymous (`anon-access`). This setting will tell the process what authorization it should allow for users who are not authenticated. The second type is authenticated (`auth-access`), which is the level of access that users who are logged in get. The svnserve process has its own user database, so when we talk about a user logging in with this method, we are referring to the custom svnserve authentication.

Anonymous access. The first step in setting up authorization is deciding what you want to do with anonymous access. The default setup is to allow anyone to read from the repository but not write to it. Remember, Subversion was built with open source projects in mind, so this setup might not work for you. If you need to lock this authorization down more, you will need change the `anon-access` setting:

```
[general]
anon-access = none
```

With this setting, you will not be able to even check out the repository unless you have an `svn` user ID. Since we have not configured users yet, if you try to run a checkout, you will get an access-denied message.

```
$ svn co svn://wiley/repos/testrepo
svn: No access allowed to this repository
```

This will also happen to any user who does not have an ID once the security is set up. Now that we have examined tighter security, let's take a look at the other direction. You can set up svnserve to allow anonymous users to write to the repository. All you need to do is change the entry from `none` to `write`.

```
[general]
anon-access = write
```

Now when you run the checkout again, you will be able to run the `commit` or any other command that writes to the repository. While this is easy to set up and administer, there is a problem. Since the repository does not need any authentication, it will not ask for credentials, so the user will be unknown. Let's make a change and commit to the repository to see what will happen:

```
$ svn commit -m "Added hello world output statement"
Sending    c/run_main.sh
Transmitting file data .
Committed revision 111.

$ svn log run_main.sh
------------------------------------------------------------------------
r111 | (no author) | 2004-06-01 20:33:24 -0400 (Tue, 01 Jun 2004) | 1 line

Added hello world output statement
```

Notice the user in the change log. Since Subversion does not ask for a user, it does not know what to use for the ID. Even if security is not an issue, if you set up anonymous write, you will lose the traceability in your change logs. Therefore, you will likely want to set up users in Subversion.

Setting up user authentication with svnserve. Setting up users in Subversion is not a difficult task, but it will add some administration hurdles that you will need to deal with. First, we will discuss how to set up users with svnserve, and then offer some suggestions about administration. The first step is configuring the authorization level for users. This is done the same way we set up the anonymous access in the previous section, but now we will use the `auth-access` setting. In almost all cases you will want these users to have write access to the repository:

```
[general]
anon-access = read
auth-access = write
password-db = passwd
```

In this setup, svnserve will allow anyone to read from the repository, but users must have an ID to write to it. The last line in the configuration will point to the file that holds the Subversion user list. The filename is relative to the `conf` directory, so if there is no directory attached, it will look for the file in `/repos/testrepo/conf`. The format of this file is pretty simple: each user ID will be listed, followed by the password in clear text. Take a look at the following password file with two users:

```
[users]
jeff = password
alex = hello123
```

First, notice that the top line in the file is the `[users]` header. This is required to tell Subversion that the following lines are users and not configuration settings. Each line contains the user ID first, followed by the = character, followed by the user's password. Only users listed in this file will have access to write to the repository.

It is important to note that this is completely independent of user IDs on the operating system. You do not need an account on the OS to have a Subversion account, and just because you have an account on the system does not guarantee that you have a Subversion ID. Also, the access we set up in the configuration file is the same for all users. There is no way to limit access by user; if users have an account, they can write to the repository.

Using one user database for multiple repositories. If you have broken your projects into multiple repositories but you have the same user base for each, you can share the password file. This is done by pointing the `password-db` setting to the same file for all the repositories. For example, if we have another repository called `repo2`, the `svnserve.conf` file would look something like the following:

```
[general]
anon-access = read
auth-access = write
password-db = /repos/testrepo/conf/passwd
realm = test
```

Notice that the authorization settings are the same; we just added a path to the password file to point to the one from `testrepo`. Also notice that we set the `realm` setting. You can use this setting to group repositories into one authorization group. While this is not required for sharing a password file, it can be helpful for users who cache their authorization credentials. We will explore this situation in chapter 9. The more repositories we add to the mix, the more security becomes a problem, and so we will need a more secure way to maintain this password file.

7.2.5 Changing the URL

We have shown that the URL the svnserve process uses is just the hostname plus the path on the local host. If all your repositories are in the same directory, you can tell Subversion to leave off the path. For example, if your repositories are in `/svn/repos`, you can omit this path and specify only the base name. This is done by using the `--root` option and giving the portion of the directory that you want to omit from the URL:

```
$ cd /svn/repos

$ ls
repo2  testrepo


$ svnserve -d --root /svn/repos
```

Now look at what we use for the URL on the `checkout` command:

```
$ svn co svn://wiley/testrepo
```

Now the users simply need to know the name of the repository and not the entire location on the host machine. This will allow you to move the repository on the host machine without having to tell the users of the change. Just rerun the svnserve process with the new root directory. Another advantage is that users have a shorter URL to deal with when running commands.

7.2.6 Changing the svnserve network settings

If you have a compelling reason to change the port Subversion uses, you can do so by adding the `--listen-port` option. Keep in mind that the clients and other third-party tools use 3690 as the default port for svnserve:

```
$ svnserve -d --root /svn/repos --listen-port 2233
```

Now Subversion will be listening on 2233 instead of 3690. But when we run a checkout with the svnserve protocol without any additional information, look what happens:

```
$ svn co svn://wiley/repo2
svn: Can't connect to host 'wiley': Connection refused
```

Since the client is trying to hit port 3690, it is failing because nothing is running on that port. You will have to change the hostname in the client commands to look for the following:

```
$ svn co svn://wiley:2233/repo2
```

We have added the port number to the end of the hostname using `:`. This will tell the client to use the port you send it, not the default. Keep in mind that you will have to use this syntax each time you access the repository by URL in the client commands.

Using a different host or IP. If you have multiple IP addresses or hostnames on the repository machine, you can tell the svnserve process on which interface to listen. When you run the svnserve process, just add the `--listen-host` option and give it the host or IP address. For example, let's say the host `wiley` has two network cards in it, and we want svnserve to run on the second one:

```
eth0: 192.168.1.10
eth1: 192.168.1.11

$ cat /etc/hosts
192.168.1.10 wiley
192.168.1.11 svn
```

We have been running everything on the primary IP address, which has the alias `wiley`. To tell svnserve to run on the second IP address, we will use the following command:

```
$ svnserve -d --root /svn/repos --listen-host svn
```

Now we can use `svn` as the hostname in the client commands, as in the following checkout:

```
$ svn co svn://svn/repo2
```

7.2.7 Running svnserve from the Linux xinetd

If you are running Subversion on Linux, you may want to make this process part of the `xinetd` so the process runs as a UNIX service. When you do this, the server will

listen on port 3690 and spawn svnserve when a request comes in. The first thing you will need to do is add the port in the `/etc/services` file to look like the following:

```
svnserve        3690/tcp                        # Subversion
svnserve        3690/udp                        # Subversion
```

When you have these entries in the `services` file, you will need to create a file in the directory `/etc/xinetd.d` called `svnserve`:

```
service svnserve
{
        disable = no
        socket_type     = stream
        wait            = no
        user            = svn
        server          = /usr/bin/svnserve
        server_args     = --inetd
        log_on_failure  += USERID
}
```

The only parts of the file that may be different for you are the `user` and `server` settings. `User` is the ID of the account that has been running the process and has read/write access to the repository. The `server` setting is the path to the svnserve process and will be wherever you installed Subversion on the machine. Now all you have to do is restart the `xinetd` daemon and you will able to hit the repository.

7.2.8 Running svnserve with ssh

we saw how to create users with the simple svnserve protocol, but what if you want to use operating system IDs? If this is the case, you can simply use the svnserve protocol on top of SSH to let the OS do the authentication. In addition to letting the OS take care of logging on the user, all of the traffic is encrypted. This is ideal for organizations with sensitive data going across the Internet. We will still use the same authorization settings, but the `password-db` setting is irrelevant because the IDs are handled by SSH. To use this method, set up the svnserve the same way we did in the previous sections by either running it in the daemon mode with the `-d` switch or with the UNIX `xinetd`. Now on the client side, instead of using `svn:` for the protocol, use `svn+ssh:`.

```
$ svn co svn+ssh://wiley/repos/repo2
The authenticity of host 'wiley (192.168.2.24)' can't be established.
RSA key fingerprint is 08:a4:2c:f7:82:a3:8d:43:76:e2:4b:c0:c2:78:cf:99.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'wiley,192.168.2.24' (RSA) to the list of known
   hosts.
jeff@wiley's password:
```

```
A  repo2/file
Checked out revision 2.
```

If this is the first time you hit the repository host machine from the client, SSH will ask you to accept the key. After you do this once, it will no longer ask you each time you connect. Now you will enter the operating system password, not the one we created in the `password-db` file in the previous section.

Why don't the username and password options work? If you try to run the `checkout` command with the `--username` and `--password` options, they will be meaningless when using `svn+ssh`. This is because the authentication happens at the operating system level before the options in the Subversion command are even parsed. This doesn't mean you can't send the username and password information, though. Instead of using Subversion to do this, you will have to use SSH. To send a different username, you must attach it to the hostname. For example, if you are logged in as `jeff` on the client, but your ID on the repository box is `jmachols`, you will use the following command:

```
$ svn co svn+ssh://jmachols@wiley/repos/repo2
```

All you have to do is add `username@hostname` in the URL, and it will try to connect as that user. While you cannot send the password in the command line, you can share the SSH private keys between the two machines so that a password is not required.

7.3 Setting up access with the Apache HTTP server

If you need a more full-featured network protocol, you can use the Apache HTTP server to connect to your repository. There is a little more to the setup and configuration than you saw in the svnserve protocol, but it is not impossible.

7.3.1 Getting the modules set up

Before you can start configuring Apache, you must ensure that the required components are in place. First, you must have Apache 2.0 or later, since Subversion will not work with anything previous to this release. You will need a module called `mod_dav.so`; this will be in the `modules` directory of your Apache home directory. If it is not there, you can download it from the Apache web site: `http://modules.apache.org`. Now you will need to locate the Subversion module called `mod_dav_svn.so`. This should be in the Subversion `install` directory under `httpd`; if it is not there, see the appendix for instructions on building it. Once you have located both modules, place them in the Apache home directory:

```
$ cd /etc/httpd/modules

$ ls -1 mod_dav*.so
mod_dav_fs.so
mod_dav.so
mod_dav_svn.so
```

In this example, the Apache home directory is `/etc/httpd`, but this may not be the case for you. No matter where it is installed, the Apache directory will contain modules. You can just copy your `mod_dav_svn.so` module from its current location into this directory. Just placing the modules in this directory is only half of the process, however. In order for the HTTP server to load them, you must use the `LoadModule` directive in the configuration file. In the beginning of the `httpd.conf` file, you will see a group of these loads. If `mod_dav` is not already in the file, you will need to add it yourself. The file should look like the following snippet:

```
LoadModule dav_module         modules/mod_dav.so
LoadModule dav_svn_module     modules/mod_dav_svn.so
```

Now that you have the modules loaded, it is time to configure the server for your repositories. As with svnserve, there are different authorization and authentication methods you can configure. Before we jump into that, let's just get a simple setup working.

7.3.2 A basic setup

Apache 2.0 has a little different setup for the configuration files than previous versions. Instead of all the settings going into one file, there is a group of files that will get read in. This will allow you to create a file for Subversion, separate from other configurations. First, to make sure the files will be read in, you should see a line in the `httpd.conf` file like the following:

```
Include conf.d/*.conf
```

This is what tells the main configuration to include everything in the directory `conf.d` that ends with `.conf`. Now let's take a look at the directory and see what is in there:

```
$ ls -1
perl.conf
php.conf
python.conf
README
ssl.conf
subversion.conf
welcome.conf
```

Depending on how you installed Subversion, there may already be a file called `subversion.conf`. If not, don't worry—you can add it easily enough. Let's examine a simple configuration:

```
$ cat subversion.conf

<Location /testrepo>
  DAV svn
  SVNPath /repos/testrepo
</Location>
```

We will use the `Location` directive to tell Apache that when it sees a URL that matches the path in the element, in this case `testrepo`, it should redirect the request to Subversion. The `SVNPath` setting will tell Subversion the location of the repository on the local machine. Before we can try this out, we need to ensure that the permissions on the path are set up correctly.

File permissions and server users. Since the repository will be accessed by the `httpd` process, the user ID that runs the process will actually be the one accessing the repository. By default, this will be the ID `apache` with the same group name. Chances are this user will not have permissions to the repository filesystem, which will cause a problem. There are a couple of solutions to get around this. First, you can ensure that the `apache` user ID has permissions to read and write to the local repository directory. The second method of correcting permissions is to use the `User` or `Group` settings in the Apache `httpd.conf` configuration file. This will run the process with a specific user and group. For example, if you have been using the user `svn` as the owner of the repository filesystems, you could set the `User` directive to this:

```
User svn
Group svn
```

If the permissions on the directory allow the group `svn` to read and write, you will need to set only one of these settings. This solution works great if you are running Apache only for Subversion. In situations where you are attaching to an established HTTP server, it may not be possible to change the "run as" user. When this is the case, you can either give the `apache` user the correct permissions or run a second instance of the web server. At this point you are ready to restart the server and try doing a checkout.

Accessing the repository through http. Once you have a simple configuration set up, you can test the connection by running a checkout of the repository. Since we

are using Apache as the protocol, the URL will be `http:` followed by the hostname and the path we set in the `Location` directive:

```
$ svn co http://wiley/testrepo
```

This is just like the svnserve we saw in the previous section except we are using `http` instead of `svn` for the protocol identifier. We need to use only the alias for the repository established in the `Location` directive. While this works for a single repository, you may want the Apache server to have access to more than one repository.

7.3.3 Multiple repositories

You could be in a situation where you are supporting multiple repositories. When this is the case, you will not want to create multiple HTTP instances for each one. In this situation, we will configure the web server in a slightly different manner. Instead of creating an alias to the path of the repository, we will create an alias to the parent directory. In our example where the repositories are in `/repos`, the configuration would look like this:

```
<Location /repos>
  DAV svn
  SVNParentPath /repos/
</Location>
```

Notice in this configuration that instead of using the `SVNPath` setting, we use the `SVNParentPath` setting. This tells Apache that anything with a URL of `/repos` should be redirected to Subversion. Along with `/repos`, you will need to provide the repository name, since we are referencing only the parent. So to check out the repository that sits in `/repos/testrepo` on the local box, you would use the following command:

```
$ co http://wiley/repos/testrepo
```

If there is another repository in the directory called `repo2`, you can check that out with the same configuration, just by changing the repository name:

```
$ co http://wiley/repos/repo2
```

As long as all the repositories are created in the directory `/repos`, they will be accessible with the current configuration, and no change will be required.

7.3.4 Users and authentication

Just as with the svnserve process, you can set up anonymous or authenticated access to the repository. If you are going to require users to log in, you need to set up the IDs, similar to what we did with svnserve. The advantage of doing this with

Apache is that it has a built-in mechanism to add or change passwords for users. Before we look at that, let's see what the configuration file looks like:

```
<Location /svn/repos>
  DAV svn
  SVNParentPath /repos/
  AuthType Basic
  AuthName "Authorization Realm"
  AuthUserFile /repos/passwd
  Require valid-user
</Location>
```

The first new line you see is the `AuthType` directive; this tells Apache what type of passwords it will be sent. We will use Basic in our examples; if you want an added layer of security, you can use Digest instead. The next setting is `AuthName`, which is similar to the `Realm` setting we used with svnserve. This will allow you to group multiple Apache instances together in the same authentication realm. But remember, unlike svnserve where each repository has its own configuration, there is only one configuration for all the repositories when using Apache (unless you set up multiple instances of the web server). `AuthName` is followed by the `AuthUserFile`, which points to the user password file. This file is similar to the `passwd` file for a UNIX operating system. The user IDs and passwords will be stored here. Now let's look at the mechanism to add users to this file.

Adding users and passwords. The Apache web server comes with a command called `htpasswd` that will allow you to maintain the user password file without having to manually edit the file. We will show a way to allow users to run this command to maintain their own account information. For now, we will run the command as the system user who will be running the `httpd` daemon we configured in section 7.3.2. The `htpasswd` command will take two arguments, the path of the file and the username on which you are running the operation. It will then prompt you for a password and finally add the entry to the file. The first step we need to take is to create the password file, which is done with the `-c` option on the command:

```
$ htpasswd -c /repos/passwd svn
New password:
Re-type new password:
Adding password for user svn
```

This option tells the command to create the file as well as add the user. While it is not required, we will add the `svn` ID as a user just to get going. Once the command is run, it will ask for the user's password and a confirmation. When this is complete, you will have a password database created in the directory `/repos`. At this point you can start adding other users, or we can set up a way for users to add themselves.

Setting htpasswd for users' self maintenance. Since all the `htpasswd` command does is work on the file you pass in, you can allow anyone to read or write the file based on the operating system permissions of the file. Since the file will reside on the machine, the command must be run on that machine. If you trust all the users who have accounts on this machine, you can just open up the permissions on the file, and then anyone can run the command and add themselves to the user file or change their password. The other side of this is that a user can change another user's password. If this is not a problem for you, just make the permissions like the following on the `/repos/passwd` file:

```
$ ls -l /repos/passwd
-rw-rw-r--  1 svn svn 37 Jun  5 10:17 /repos/passwd
```

Now anyone in the `svn` group can operate on this file. If this security does not work for you, change the permissions on the file so that only the `svn` user can update the file. You can also create a copy of the `htpasswd` command and give it special permissions so that any user can run the command as `svn`. This will allow you to write a wrapper script that users can run so that they can still update the file on their own but only with their own ID.

7.3.5 Configuring authorization

In the previous example, user authentication is required for all operations on the repository, both read and write. This is indicated by the `Require valid-user` directive in the base configuration. If you want to set up access so that anyone can read from the repository, but you still require a login for writing, use the following configuration:

```
<Location /svn/repos>
  DAV svn
  SVNParentPath /repos/
  <LimitExcept GET PROPFIND OPTIONS REPORT>
    AuthType Basic
    AuthName "Authorization Realm"
    AuthUserFile /repos/passwd
    Require valid-user
  </LimitExcept>
</Location>
```

By using the `LimitExcept` directive, you can configure Apache to allow certain operations to bypass the authentication. Since Subversion uses the DAV protocol, the operations will be based on that protocol. If you are not familiar with this, don't worry; by using the previous configuration, you will have all read operations open and all write operations will require authentication. You saw this type of total coverage for authorization with the `svnserve` process. When you set up access, it is

either none, read only, or write for everything. The Apache protocol gives more granularity than this.

Detailed authorization. In the Apache setup, you can actually create access control down to the directory level and by user. First, you will need one more module loaded by the server called `mod_authz_svn.so`. We will load this module the same way as the other modules. Just be aware that it is important to follow the order shown:

```
LoadModule dav_module modules/mod_dav.so
LoadModule dav_svn_module modules/mod_dav_svn.so
LoadModule authz_svn_module modules/mod_authz_svn.so
```

This module will be in the same location as `mod_dav_sv`. If you had to build that module or install it separately, `mod_authz_svn` will be part of it. Once you have the module loaded, you will need to turn it on by setting the `AuthzSVNAccessFile` directive in the `subversion.conf` file. Just like the password file, this configuration will point to a file that contains a list. In this case, the list will be a set of directories in the repository along with a list of users and their permissions. When setting up the control file, you need to be aware that by default access is not allowed. So if a directory or user is not represented in the file, there will be no authorization. Let's take a look at a simple configuration to see how the file is organized:

```
$ cat svn.auth

[testrepo:/]
jeff = rw
```

Each directory control list starts with the directory name encapsulated inside `[]` and is translated to `repository_name:directory`. Everything under the directory given will inherit the permissions, so in our example, starting the list with `/` for `testrepo` will include the entire repository. Under the directory heading will be the list of users with access. We have one user listed, `jeff`, who is assigned read/write access. There are only two values that can be assigned to a user, `r` and `rw`. An `r` will give the user read access, and `rw` will allow read and write.

Now let's say we have a slightly more complex situation, with two repositories and two users. The first repository is `testrepo` and is a sandbox for the user `jeff`. We will allow him to have full rights to `testrepo`, but the user `alex` will not be working in here, so he does not get access. The second repository, `repo2`, holds the production code. If the user `jeff` is new, we may not want him to have access to the trunk where we deploy code. The only user who can write to the trunk is `alex` since he is charge of the production source files. Even though `jeff` cannot work on the trunk, we will allow him to submit patches via a branch. Remember

that in order to create a branch, you must be able to read the trunk. So `jeff` will have read access to the trunk and write access to the `branches` directory. These permissions are shown in the following configuration:

```
[testrepo:/]
jeff = rw

[repo2:/trunk]
jeff = r
alex = rw

[repo2:/branches]
jeff = rw
alex = rw
```

You can get as detailed as you want on the access control lists, but as you can imagine, they will become more difficult to manage as the number of users and directories increases. In order to help compensate for this, there are a couple of features you can take advantage of to ease the administration of authorization.

Wildcards. Since the default permission on directories is no access, you may need to just give everyone access to a location. Instead of adding every user in your system, you can use * to indicate this. For example, if you decided that all users should have complete access to the `testrepo` repository, you could use the following configuration:

```
[testrepo:/]
* = rw
```

Now all users will have read and write access to the `testrepo` repository. Also, when you add new users, they will automatically have permissions. In most cases, since the point of access control lists is to limit permissions, you will not want to globally open up something. You can still ease the administration burden by placing users into groups.

Group access control. Think back to chapter 5 when we talked about branching strategies; one of the methods was to put all the releases into a branch. Let's assume that we are using this method in the development process and we create branches called `release_x.x`. These branches will be created when the software is ready for testing, at which point the QA staff will need access. The rest of the developers will not be able to write once the code is there. Whenever you have distinct segmentation of users like this, you can use groups in the authorization file. Using the requirements we just saw, let's see what the configuration will look like:

```
[groups]
dev = jeff, alex, adam
QA = paul, patti

[testrepo:/]
* = r

[testrepo:/trunk]
@dev = rw
@QA = r

[testrepo:/branches/release-2.0]
@QA = rw
@dev = r
jeff = rw
```

The first thing to notice is a new header called `[groups]`; this section defines the groups in the authorization and which users are in the group. You can see in our example that there are two groups defined: `dev` and `QA`. Each group is followed by a comma-delimited list of the users who belong to the group. Now when you reference the groups in the directory headings, simply add the `@` character in front of the name to tell Subversion that this is a group, not a user. The permissions you set on the group are the same as for a user. You may also notice in the last line that we made an exception with the user `jeff`. Even though this user is in the `dev` group, which gives him only read access to the `release-2.0` directory, we added an additional entry for him. Since we did this, he will have permissions to write to the directory. So even though you are using groups, you can still manage individual users where exceptions occur.

In addition to securing your repository through limiting user access, you can protect the data on the wire.

7.3.6 Encrypting the transmission

If you are accessing the repository over the Apache network protocol, especially over the Internet, you may want to consider using SSL to encrypt the data. Instead of the client talking to the server over port 80 over `http`, it will use port 443 with `https`, just like a secure web page. To turn this on, all you will need to do is add the setting to the `subversion.conf` file. Let's take a look at our configuration with the new setting:

```
<Location /svn/repos>
  DAV svn
  SVNParentPath /repos/
  AuthzSVNAccessFile /repos/svn.auth
  SSLRequireSSL|                    <——— Setting for using SSL
```

```
  <LimitExcept GET PROPFIND OPTIONS REPORT>
    AuthType Basic
    AuthName "Authorization Realm"
    AuthUserFile /repos/passwd
    Require valid-user
  </LimitExcept>
</Location>
```

As you can see from the new configuration file, only one line is required, and it need not be set to any value. Now all you need to do is restart you web server and it will be switched to `https`. If you then try to connect using `http`, you will get an error:

```
$ svn co http://wiley/svn/repos/testrepo
svn: PROPFIND request failed on '/svn/repos/testrepo'
svn: PROPFIND of '/svn/repos/testrepo': 403 Forbidden (http://wiley)
```

By using `https`, the client will talk to the Apache server using SSL. If you are connecting to the server for the first time, you will get a warning from the protocol informing you that there is no trusted certificate between the two machines:

```
$ svn co https://wiley/svn/repos/testrepo
Error validating server certificate for 'https://wiley:443':
 - The certificate is not issued by a trusted authority. Use the
   fingerprint to validate the certificate manually!
 - The certificate hostname does not match.
Certificate information:
 - Hostname: localhost.localdomain
 - Valid: from May 29 03:20:26 2004 GMT until May 29 03:20:26 2005 GMT
 - Issuer: SomeOrganizationalUnit, SomeOrganization, SomeCity, SomeState, --
 - Fingerprint: 6f:3d:f2:5d:29:1b:9d:18:f9:d5:b9:21:b3:32:15:72:22:72:fa:e9
(R)eject, accept (t)emporarily or accept (p)ermanently? p

A  testrepo\trunk
A  testrepo\trunk\source
A  testrepo\trunk\source\java
A  testrepo\trunk\source\java\hello.java
...
Checked out revision 120.
```

If this is the case, you will be prompted to accept or reject the certificate. You can temporarily accept it, which will give you a key for only this command. In most cases you will want to accept the key permanently. When you do this, the certificate will be stored on the client machine and used each time you run a command. Now you can run commands without having to accept the key or any manual intervention. In the svnserve process, when we switched to SSL, the authentication was handled by the operating system. With Apache, the account information is still handled by the user database we set up with the `htpasswd` file. This built-in

authentication is one of the big advantages of using Apache. Another one we have talked about is the ability to browse the repository via a browser.

7.3.7 Repository browsing

Since Subversion uses the DAV protocol, the Apache web server lets you browse the repository "out of the box." All you need to do is point your web browser to the same URL you will use to check out the code. As with any of the other Subversion commands, you can use any path in your repository to start. Let's take a look at the `java` directory in the trunk. Figure 7.1 shows what the browser will look like. Since we have turned on SSL in the Apache configuration, the URL in the web browser will be `https://wiley/svn/repos/testrepo/trunk/source/java/`.
You can see that there is nothing fancy about the web page; it simply lists the files and directories in the path you are pointing to. The Apache protocol will show only the latest revision in the repository, and the version number will appear at

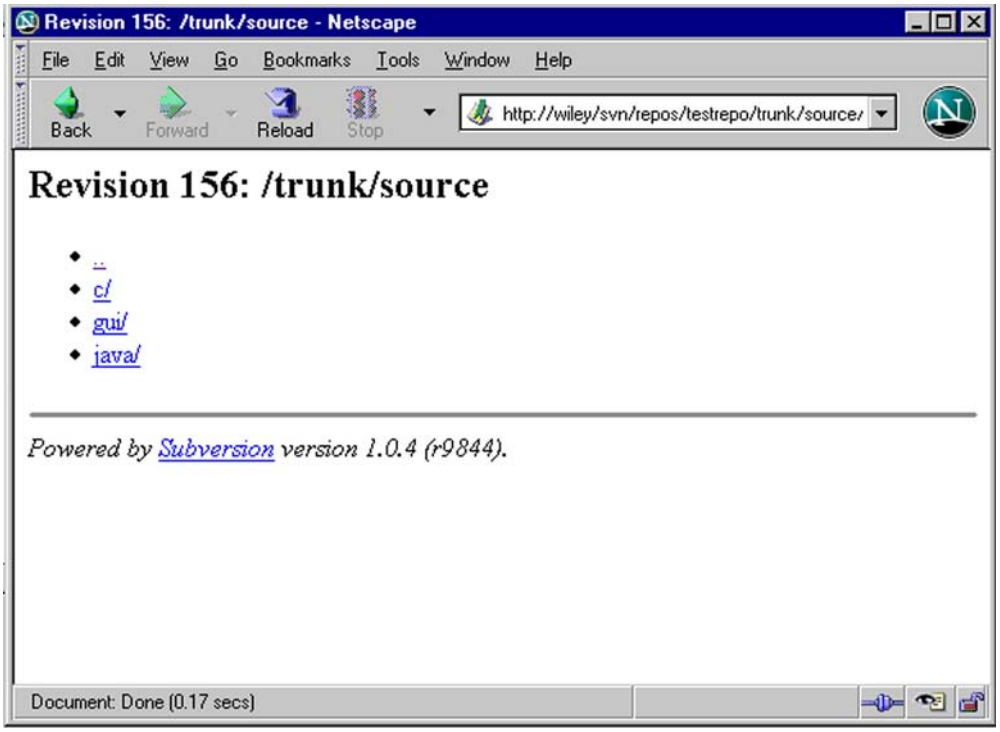

Figure 7.1 Directory listing in the Apache web browser

the top. If you select a file instead of a directory, its contents will be shown, as in figure 7.2.

While this may not provide a lot of functionality, Apache will give you a quick solution to viewing the repository. If you want more functionality such as log viewing, there are some third-party tools with additional features, which we will cover in chapter 10. While you may not be able to get more features, you can change the appearance of the web page.

Adding stylesheets. You can use the XSLT stylesheet to liven up the web page that is produced when browsing the repository. The Subversion distribution comes with a stylesheet called `svnindex`, and it will be in the `$SUBVERSION/tools/xslt` directory. It contains two files, and you will need to copy them somewhere that Apache can see them. For now, we will put them in the directory root of the HTML files:

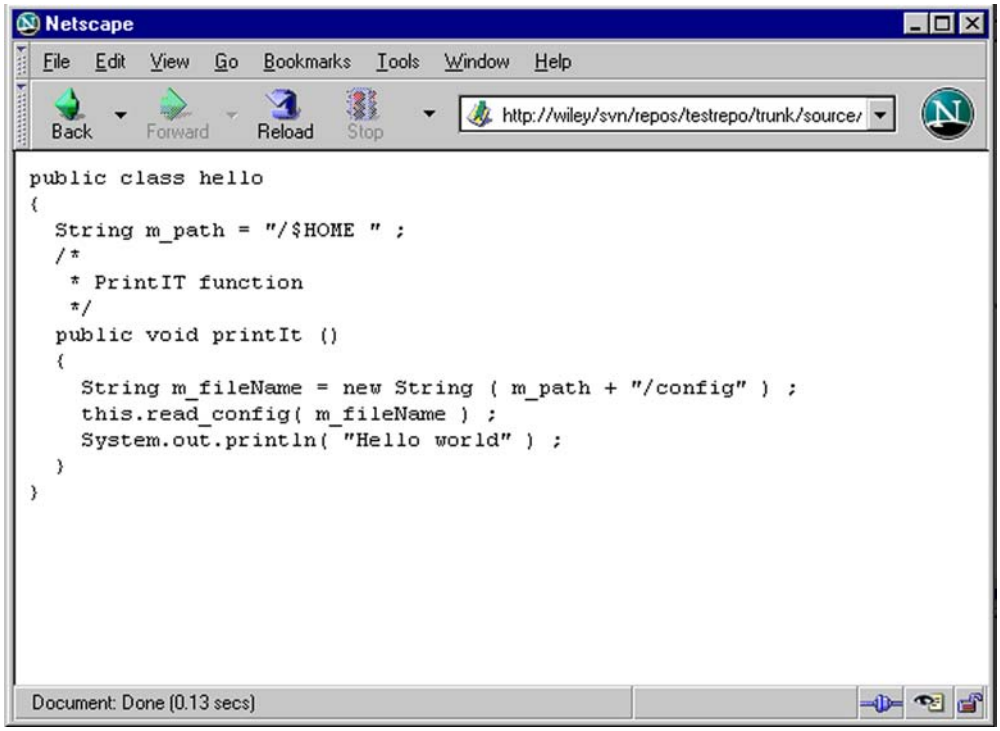

Figure 7.2 File contents from Apache

```
$ cd /usr/share/doc/subversion-1.0.2/tools

$ ls
svnindex.css  svnindex.xsl
$ cp * /var/www/html/
```

Now need to turn on a directive in the `subversion.conf` file to point to the stylesheet. To do this, we will add the following line to the configuration:

```
SVNIndexXSLT "/svnindex.xsl"
```

This will apply the filename of your stylesheet to the web pages. Now take a look at the browser shown in figure 7.3.

As you can see, this gives a little more life to the page. You can add to this stylesheet or create one of your own to get a different look or feel for your repository.

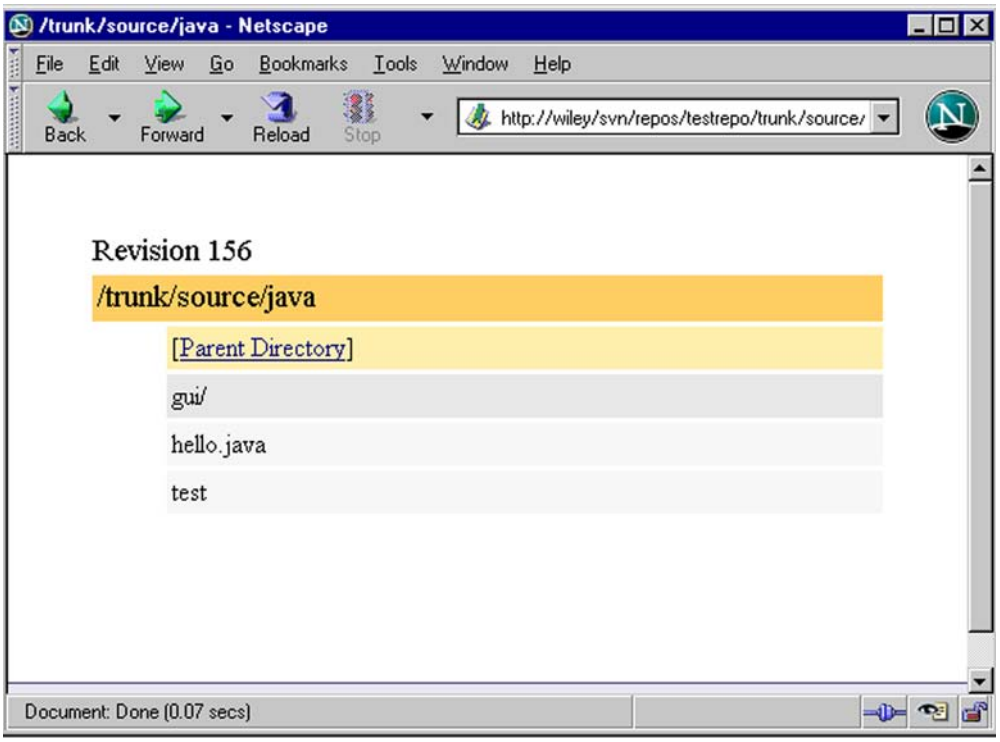

Figure 7.3 Apache web browser with a stylesheet

7.4 Summary

Repository administration is a necessary task, but the amount of work required depends on your individual needs and setup. We have looked at setting up and maintaining Subversion, but some of the tools it uses can be further customized. For example, using SSH with the svnserve protocol, you can set up trusted keys so that a user's ID and password will not be required. Also, there are numerous configurations in Apache that could be applied to your implementation of Subversion; in fact, there are plenty of books dedicated to this topic. While not all implementations will require you to use all of the tools discussed in this chapter, it is important to at least understand the capabilities and then decide which ones are important to you.

8 Advanced administration and configuration

In this chapter

- Configuring the client
- Maintaining and configuring Berkeley DB
- Hook scripts

In the last chapter we explained how to do some of the more common administration tasks for the repository. Now we can start looking at fine-tuning Subversion on both the client side and the server side. These are the areas where you can begin to see exponential benefits of using Subversion in your development lifecycle. We will start by looking at some conveniences you can add to the client side that will relieve you of some of the tedious operations. We will then move into creating automation with triggers and hook scripts, which we have touched on throughout the book. These hook scripts can be a large factor in enforcing your development process rules.

8.1 Client-side configuration

Throughout the examples of commands we have looked at ways to customize the interaction of the client and the Subversion filesystem. These ranged from ignoring certain files via the `svn:ignore` command to accepting SSL certificates when communicating over `https`. While all these things can be done through switches, separate commands, or even manual intervention, Subversion provides a means to set up some of these behaviors on the client automatically. This is done through configuration files in the home directory of the user on the client box. When you run a Subversion command for the first time, it will create a directory called `.subversion` in your home directory and place certain files in it. Modifying these files will allow you to change the behavior of your client without having to constantly add options to the command line.

8.1.1 Configuration files

Two files and one directory make up the configuration of the client side in Subversion. By default, any command will look in the `$HOME/.subversion` directory on a UNIX-based system. If your client is a Windows system, the files will be in the `%APPDATA%` directory and will have the same layout and names. Let's examine this directory and see what is in there:

```
$ cd /home/jeff/.subversion/
$ ls -l1R
.:
total 24
drwx------    5 jeff     svn          4096 Apr  1 21:34 auth
-rw-r--r--    1 jeff     svn          4224 Apr  1 21:34 config
-rw-r--r--    1 jeff     svn          3328 Jun 10 22:13 servers

./auth:
total 12
```

```
drwxr-xr-x    2 jeff     svn          4096 Jun 11 11:21 svn.simple
drwxr-xr-x    2 jeff     svn          4096 Apr  1 21:34 svn.ssl.server
drwxr-xr-x    2 jeff     svn          4096 Apr  1 21:34 svn.username
```

First you will notice a directory called `auth`, which contains the three subdirectories shown in the second part of the listing. In chapter 7, we saw how the svnserve process caches authentication information; this is where that data is stored. You will not likely add anything here manually, although you will see later how you can change some of this information. Next you will see a file called `config`, which will store information about how the client commands should behave. For example, you can set up the `ignore` property automatically on certain types of files. Finally, there is a file called `servers`, which will allow you to configure how to talk to Subversion repositories over the network. We will look at each of these configurations in detail shortly, but for now you need to understand the structure. By default these files will reside in `.subversion`, but you can move them elsewhere or create additional configuration directories if you need to do so.

Creating multiple client configurations. For the most part, there are ample customization capabilities in the files to support multiple repositories and multiple servers, but sometimes these may not be enough. For example, let's say you have a workstation containing the Subversion client you use to work on both an open source repository and the source code for your job. You may want completely different behaviors depending on which set of code you are working on. In this case, you can set up an additional configuration directory and then specify which one to use on the command line. We will use the default `.subversion` directory for the source code and create a new one for the open source project. First, copy the original directory to a new one and call it `.svnconf_open`:

```
$ cp -pr .subversion .svnconf_open

$ ls .svnconf_open/
auth  config  servers
```

You will need to ensure that you get the entire directory structure by using the `--recursive` switch in your `copy` command. Once you have this structure, you can make any customizations in the new configuration. Then all you need to do is add the `--config-dir` switch on any of the `svn` client commands you want to use with the new settings:

```
$ svn commit --config-dir /home/jeff/.svnconf_open --message "Bug Fix"
```

While this works great and is straightforward, it can be tedious to add this string each time you run a command. To get around this problem, you can use an alias.

For example, the following alias could be used for all commands that use the open source configuration:

```
alias svn_open='svn --config-dir /home/jeff/.svnconf_open'
```

Now all you need to do is run this command instead of `svn` with all the same subcommands and options:

```
$ svn_open commit --message "Bug Fix"
```

This will yield the same results as adding the `--config-dir` option and requires much less typing. Now that you have seen the structure and know how to access the configuration options, let's look at what these options are.

8.1.2 Configuring the server connection information

If you are connecting to your repository over `http` or `https`, the `servers` file in the configuration directory will allow you to customize the connection settings. Let's take a look at the possible settings in this file:

```
###   http-proxy-host            Proxy host for HTTP connection
###   http-proxy-port            Port number of proxy host service
###   http-proxy-username        Username for auth to proxy service
###   http-proxy-password        Password for auth to proxy service
###   http-proxy-exceptions      List of sites that do not use proxy
###   http-timeout               Timeout for HTTP requests in seconds
###   http-compression           Whether to compress HTTP requests
###   neon-debug-mask            Debug mask for Neon HTTP library
###   ssl-authority-files        List of files, each of a trusted CA
###   ssl-trust-default-ca       Trust the system 'default' CAs
###   ssl-client-cert-file       PKCS#12 format client certificate file
###   ssl-client-cert-password   Client key password, if needed
```

This is copied from the `servers` file, to provide a reference of all the possible settings. You can apply these settings to specific servers individually or globally by placing specific servers into groups and then setting the values for the configurations of each group. Take a look at the following example:

```
[groups]
work = *.manning.com
open = svn.apache.org

[work]
neon-debug-mask = 4

[open]
ssl-authority-files = /home/jeff/ssl/svn_server.crt
```

1 Defines the groups

2 Configures all servers in the workgroup

```
[global]                                    ❸ Sets the global settings
http-compression = yes <
```

❶ In the first part of the file we declare the groups and place servers in them. The section starts with the `[groups]` header. Each line in this section has a group name followed by the servers in the group. You can simply list the server name, such as `svn.apache.org`, or you can use wildcards, such as `*.manning.com`. In this case, any server in the `manning.com` domain will be in the group `work`.

❷ Once we have the groups defined, we configure the specific settings for them. First, the group we are setting up is declared in the header, for example, `[work]` and `[open]`. Under each of these headings we list the settings to change. Any setting not specified will be set to the default.

❸ We use the `[global]` header to set configuration options for all servers to which we connect. This is used as a catchall for any servers not defined in groups.

Proxy settings. If you are required to go through a proxy server to connect to your repository, you must configure it in the `servers` file. The settings for proxy are fairly standard. Consider the following example:

```
http-proxy-host = proxyserver.mydomain.com
http-proxy-port = 7000
http-proxy-username = proxyuser
http-proxy-password = proxypasswd
```

You need to include these entries for all the server groups that need to go through a proxy to talk. You can build exceptions into the configuration by using the `http-proxy-exceptions` directive to list the servers that do not need to go through a proxy, for instance, machines inside the network. So if we had a repository on an internal server called `svn-server.mydomain.com`, we could use the following configuration:

```
[global]
http-proxy-host = proxyserver.mydomain.com
http-proxy-port = 7000
http-proxy-username = proxyuser
http-proxy-password = proxypasswd
http-proxy-exceptions = svn-server.mydomain.com
```

All servers except the one defined in the `exceptions` directive will go through the proxy and use the specified settings. Of course, if we had multiple internal servers on our domain, we could use the following statement instead:

```
http-proxy-exceptions = *.mydomain.com
```

Just like setting up the servers in a group, using a wildcard will specify all servers in the domain. In addition to setting proxy information, you can configure communication settings.

Communication settings. There are two options that will allow you to tweak the way the client communicates with the server. The first is fairly trivial: `neon-debug-mask`. This will turn on a higher level of debugging based on an integer value you set. The higher the number, the more debugging messages will be displayed. Let's use the following setting and see what we get:

```
neon-debug-mask = 6
```

Now that we have this option set, Subversion commands that talk over the network protocol will print additional messages. Look at the results when we run a `checkout` command:

```
$ svn co https://wiley/svn/repos/testrepo
Creating request...
Running request create hooks.
Request created.
Doing DNS lookup on wiley...
Running pre_send hooks
Sending request headers:
PROPFIND /svn/repos/testrepo HTTP/1.1
Host: wiley
User-Agent: SVN/1.0.4 (r9844) neon/0.24.6
Keep-Alive:
Connection: TE, Keep-Alive
TE: trailers
Content-Length: 300
Content-Type: text/xml
Depth: 0
Accept-Encoding: gzip
Accept-Encoding: gzip

Sending request-line and headers:
Connecting to 192.168.2.24
```

The `neon-debug-mask` option is good to use if you are having problems talking to the Apache server. If there is a problem, you can see what the command is trying to do at that time.

The second communication setting you can adjust is whether or not Subversion will compress data before sending it over the wire to the server. This option is useful if you have a slow connection between the client and the server. The compression will add processing time on both ends, but if you have large amounts of

data and a small pipe, this processing may be offset by the faster transfer time. You turn on compression by setting `http-compression` to `yes`.

```
http-compression = yes
```

While this option can help if you are transferring large quantities of data, you need to remember that only changes are sent across the wire. So if you change only a few lines in a file, only those differences are sent, not the entire file. Therefore, you may not be sending as much data as you think. The best way to see whether this option helps is to turn it on and try it.

So far, we have examined settings that affect how the client talks to the server. If the method of communication is over `https`, there are a few more settings that can help you out.

8.1.3 SSL settings

When you use SSL with your communication, some form of certificate exchange will be required. We previously saw this when we first connected to the `https` protocol in chapter 7 and had to accept the certificate. Since we don't have the time or space for a full discussion on SSL and certificates, we will just briefly describe these settings. In the appendix we will cover the steps for creating and sharing certificates for authentication. There are two forms of certificate authentication; the first is *server* based.

Server certificates. The first time you establish a connection from a client to the Subversion server, you will be asked to accept or reject the certificate. If you are in a small, trusted community, you can get around this by sharing the certificate up front. For example, if you are using a certificate called `server.crt` in the Apache configuration for SSL authentication, you can copy it to the client machine. Then, add the following line to the `servers` file:

```
ssl-authority-files = /home/jeff/ssl/server.crt
```

Now when the client connects and the server passes down the certificate, the client will know about it. No manual intervention will be required to accept a certificate. In this example, we are using a custom certificate, but you may just want to use the default certificate that ships with openssl. If this is the case, you need only specify the following directive in the `servers` file:

```
ssl-trust-default-ca = yes
```

This will tell Subversion to accept all the default openssl certificates without having to copy files back and forth. What we have shown so far is only half of what

you can do with SSL. In some cases, communication may be set up to require a client certificate.

Client certificates. Some systems with an extra layer of security require a client certificate. This is a certificate created on the client machine, signed by a Certificate Authority (CA), and trusted by both systems. If the Apache server is set up to require a certificate from the client, you will be prompted when running a command to provide a valid certificate. It can be cumbersome to manually enter this information each time you run a command, so you can configure the `servers` file to pass in the certificate automatically. This is done with the `ssl-client-cert-file` setting:

```
ssl-client-cert-file = /home/jeff/ssl/mort_svnclient.p12
```

All you do is set the value equal to the name of the certification file. It is important to note that the file extension is `.p12`, which is the standard for files in PKCS#12 format. Only certificates in this format will be accepted by Subversion for automatic transfers. Files in the old PEM format won't work.

In many cases when a server requires a client certificate, it will issue a password challenge. Even if you have set up the file to be sent automatically, manual intervention will be required for the password. You can automate this process also by adding an entry to the `servers` file:

```
ssl-client-cert-password = cert1passwd
```

Now when the server asks for a certificate, the client will send `mort_svnclient.p12`, and if a password challenge comes, `cert1passwd` will be transferred. If either of these fails the server authorization, the command will prompt you.

8.1.4 Command configurations

The second file in the client configuration directory, `config`, contains instructions for the `svn` client on how to behave. Just like all the configuration files we have seen in Subversion, the `config` file uses the standard INI format and has multiple sections identified by `[]` characters. This file is broken into several parts; the first is authentication.

Authentication caching. The authentication section is defined by the `[auth]` header and has only one setting. As discussed in chapter 7, the Subversion client will cache the authentication credentials so that the next time you run a command, you will not need to provide a password. We also showed that you can turn off this behavior by using the `--no-auth-cache` switch. You can permanently turn it off by setting the value `store-auth-creds` to `no` in the `config` file:

```
[auth]
store-auth-creds = no
```

This setting will prevent commands from saving the credentials when they are run. It is important to note that setting this switch will not go in and remove previously stored information in the `auth` directory. In section 8.1.5 we will show how to manually remove this information.

Custom SSH tunnels. Using the `svn+ssh` protocol, you can connect to the svnserve process over an SSH tunnel. Remember, you will have to change the URL if you want to specify a user ID or a different port from the default. For example, let's say your user ID on the client machine is `jeff` but your ID on the repository host is `jmachols`. You would have to use the following URL to pass in the server ID:

```
svn co svn+ssh://jmachols@wiley/repos/testrepo
```

Instead of changing the URL, you can create a custom SSH tunnel that specifies any set of parameters you want. This section starts with the `[tunnel]` header, and then you define your own scheme. Let's take a look at an example:

```
[tunnels]
ssh = /usr/bin/ssh -l jmachols
```

This will give the Subversion client the command to substitute when SSH is in the protocol. Now we can use the URL without having to change the ID on the command line:

```
$ id
uid=1000(jeff) gid=1000(svn) groups=1000(svn)

$ svn co svn+ssh://wiley/repos/testrepo
jmachols@wiley's password:
```

As you can see, the ID on the client box is `jeff`, but since we added a custom tunnel that changes the name with the `-l` option, the command will try to connect using `jmachols` as the user. You are not limited to one tunnel; you can create multiple tunnels and name them whatever you like. For example, assume that in addition to the previous tunnel we created, you have another repository that has a different ID and talks over a different port. Your `config` file might look something like the following:

```
[tunnels]
ssh = ssh -l jmachols
ssh_mort = ssh -l jjm -p 1212
```

Now you have a scheme that will use an ID of `jjm` and connect to port 1212. To use this, simply replace the `ssh` with `ssh_mort` for the protocol of the URL:

```
svn co svn+ssh_mort://mort/repos/repo2
jjm@mort's password:
```

You can use any command to connect as long as it can connect and authenticate on the server side. Specifying an SSH connection executable is not the only external program you can define in the configuration. Other Subversion commands that use external programs can also be set up in the `config` file.

External command declarations. The next section in the `config` file sets up which external commands Subversion should use. You have seen multiple instances of this, for example, the editor used when you are entering a log message for commands such as `commit` and `copy`. This section is defined under the `[helpers]` header, which stands for helper applications. There are four settings you can use here:

```
[helpers]
editor-cmd = vi
diff-cmd = /usr/bin/diff
diff3-cmd = /usr/bin/diff3
diff3-has-program-arg = true
```

The first setting is `editor-cmd`, which we have explored in detail. This tells Subversion which editor to use for entering log messages. This setting also overrides anything you have as environment variables such as `EDITOR` or `SVN_EDITOR`. The next setting is `diff-cmd`, which we saw in chapter 4 when looking at the `svn diff` command. This provides a system executable to perform the operation instead of Subversion doing it. This setting can be used to replace the `--diff-cmd` switch in any Subversion command.

The next two settings look similar to `diff-cmd`, but they are used for the `diff3` command replacement. Commands such as `svn merge` use `diff3` since they are dealing with multiple files. The `diff3-cmd` setting provides an external command to any Subversion operations that use `diff3`. Simply set the value of the setting to the path of the executable you want to use. The last setting goes along with the `diff3-cmd` setting; it tells Subversion whether the external command you are specifying for `diff3` accepts command-line options. Some external `diff3` programs take the same options as a `diff` command and some do not. This value can be set to `true` or `false` depending on whether it can receive these options.

Miscellaneous options. In keeping with INI file format and having settings grouped together under a heading, the next set of configuration options is unrelated to anything else, so they are grouped together. This section is identified with the `[miscellany]` header. The first setting here is called `global-ignores` and it provides a systemwide list of files to ignore in the `svn status` command. This can be set to a space-delimited list of files, and it will accept wildcards:

```
global-ignores = *.o .project *.class
```

In this list, Subversion will ignore all files named `.project` and anything that ends with `.o` or `.class`. These will be in addition to the ignore list set on a directory with the `svn:ignore` property.

Moving on, the next setting in this section is `log-encoding`. By default, Subversion will use the locale of the system from which you are entering the log message. If you wish to override this setting, you can set the encoding in the `config` file:

```
log-encoding = UTF8
```

The text entered as the log message will be transferred as UTF8 encoded, so you will need to make sure you are actually using the locale specified or you may get corrupt data.

When you run any command that updates the working copy such as `checkout` or `update`, the files get a timestamp when they are created. So when you check out a repository, all its files will have the same timestamp. Let's do this with the `testrepo` repository and look at one of the directories:

```
$ svn co http://wiley/svn/repos/testrepo
A  testrepo/trunk
...
Checked out revision 129.

$ ls -l testrepo/trunk/source/java
total 12
-rw-r--r--    1 jeff     svn           193 Jun 15 21:41 goodbye.java
-rw-r--r--    1 jeff     svn           187 Jun 15 21:41 hello.java
-rw-r--r--    1 jeff     svn           236 Jun 15 21:41 main.java
```

As you can see, all the files looked like they were created at the same time, but this is not the case. If you want Subversion to use the last time the file was changed, you can set `use-commit-times` to `yes`.

```
[miscellany]
use-commit-times = yes
```

Now when a file is updated in the working copy, it will show the time of the last commit. Let's remove the working copy and check it out again:

```
$ rm -rf testrepo/

$ svn co http://wiley/svn/repos/testrepo
A  testrepo/trunk
...
Checked out revision 129.

$ ls -l testrepo/trunk/source/java
total 12
-rw-r--r--    1 jeff     svn           193 Jun  3 21:51 goodbye.java
-rw-r--r--    1 jeff     svn           187 Jun  7 19:15 hello.java
-rw-r--r--    1 jeff     svn           236 Apr 29 22:13 main.java
```

Now look at the timestamp of the files; they will indicate the last time the file changed in the repository, not when it was checked out. There is one more setting in the miscellaneous section we will look at. This setting is called `enable-auto-props` and should be set to `yes` if you want to have properties set automatically when you add files:

```
enable-auto-props = yes
```

Turning this on by itself will not do anything, but when you use it in conjunction with the next section, `[auto-props]`, the properties you define will automatically be set on all new files.

Automatic properties. In the final section of the `config` file you can set up properties that will be attached to all files or to types of files when they are added to the repository with the `add` or `import` command. This section starts with the heading `[auto-props]` and will contain a list of file types followed by the property to set. Let's see some examples:

```
[auto-props]
*.c = svn:eol-style=native
*.sh = svn:executable
*.jpg = svn:mime-type=image/jpeg
```

By using wildcards, you can ensure that all files that end with `.sh` will have the property `svn:executable` set. You can have as many files or file types using wildcards as you want. If you need more than one property set on a file, just separate them with `;`.

In the beginning of section 8.1.4, we looked at the authentication section, where you can turn off caching. We will now look at where that information is stored and what can be done with it.

8.1.5 Authentication caching

The Subversion client stores login credentials from the svnserve process and keeps this information in the `auth` directory under `.subversion`. Depending on how you connect, svnserve will dictate which subdirectory the credentials are stored in. If you just connect using `svn`, they will be stored in `svn.simple`. For connections using `svn+ssh`, the credentials are stored in `svn.ssl.server`. Let's look at the file structure for a client box that has a credential stored for each connection method:

```
./svn.simple
./svn.simple/77b88bdccc344a3b09200ad7d5abfa69
./svn.ssl.server
./svn.ssl.server/58321f1091283cff60c1bfe92086f2e2
```

As you can see, the filenames in the directories are not very meaningful to the user. Let's see what's inside the file in the `svn.simple` directory:

```
$ cat svn.simple/77b88bdccc344a3b09200ad7d5abfa69
K 8
username
V 4
jeff
K 8
password
V 5
hello
K 15
svn:realmstring
V 23
<svn://wiley:3690> test
END
```

❶ User ID

❷ User password

❸ Authentication realm

❶ In each section of the file, there is a line telling you the piece of the credential, in this case the username, and the actual value. For this file, it is storing the ID `jeff`.

❷ This is the password for the user; notice that it is stored in clear text. The permissions of the files stored in this area will be read only by the user who owns them, and they should stay this way. If you don't want this type of storage, you can set the client not to cache, as shown in the previous section.

❸ When we set up the svnserve authentication in the previous chapter, we showed how to set up multiple repositories with the same user database using realms. The last part of this file specifies the authentication realm of which this credential is a part.

Once you create these files, you can edit them if required. For example, if you change your password, you can modify the file to reflect this change. Let's say that on the svnserve host, you changed the password from "hello" to "password123."

Not only do you change the password string, but line above it that contains the length of the string also changes. Therefore, you will need to change the number to reflect the new size of your password. Let's look at the new entries:

```
 8
password
V 11
password123
```

After making this change you will be able to run commands that require authentication without manual intervention. If you no longer want to store this information, simply remove the files and set `store-auth-creds` to `no`.

8.2 Resolving deadlocks

There are two types of locks you may encounter while using Subversion: repository and working copy. Each of these problems has a "magic" command to clean up any problems. To help you even further, when you run into these issues Subversion will tell you which command to run. First, let's examine the working copy locks.

8.2.1 Working copy locks

When we looked at the `status` command and the possible states of a working copy file, we saw the lock field. Normally the only time you will see a file in this state is if someone is entering a log message and hasn't finished yet. Remember, when you run the `commit` command, the file goes into a locked state until the command is complete. If the command writing to the repository dies ungracefully, it is possible that the file will remain in a locked state. There are two ways to determine that this is going on; the first is receiving an error message when you try to write to the repository:

```
$ svn commit
svn: Working copy '/home/jeff/projects/testrepo' locked
svn: run 'svn cleanup' to remove locks (type 'svn help cleanup' for details)
```

This is pretty straightforward; the command will tell you that there is a working copy lock and even let you know how to fix it. We will get there in a minute. The second way to tell whether there is a lock is by running the `status` command:

```
$ svn status
  L    trunk/source/java
?      trunk/source/.svn-commit.tmp.swp
M      trunk/source/java/hello.java
  L    trunk/source/c
```

The lock field of two directories contains an `L`, which means they are locked. Once you have identified this situation, you can run the `cleanup` command.

Cleanup command. As the error on a write operation will tell you, in order to fix a lock on the working copy you will need to run the `svn cleanup` command. The best way to run this is from the top level of the working copy. It will recursively parse the directory and clean up all locks and resume any unfinished operations:

```
$ svn cleanup

$ svn status
?      trunk/source/.svn-commit.tmp.swp
M      trunk/source/java/hello.java
```

Now you can see that the files with the lock status are gone, so your working copy has been cleaned up—almost. While it is completely normal to have non-versioned files in your working copy, as indicated by a `?` in the status, in this case the file is left over from the lock. When you see a file named `.svn-commit.*`, it is an artifact from the failed command. Depending on which editor you use for the log messages, you may have the autosave copy hanging out there. You can remove this if you like. Or, the next time you run a command that writes to the repository, anything that was in the editor will show up again, which may be good if you want to get that text back.

The locks on the working copy are easy to get out of, and they will not impact other users because the problem is contained in your copy. If things get really bad, you can remove the working copy and start over. The worst case would be that you'd lose any work since the last time you committed, which is usually not the end of the world. The next type of lock can be a little scary because it affects the repository.

Repository locks. You may run into a situation where the repository host is not shut down cleanly or some other event happens that causes the Berkeley database to become corrupted. When this happens, you will see an error like the following:

```
svn: Unable to open an ra_local session to URL
svn: Unable to open repository 'file:///repos/testrepo/source/c/main.c'
svn: Berkeley DB error while opening environment for filesystem /repos/
   testrepo/db:
DB_RUNRECOVERY: Fatal error, run database recovery
```

When you see this message, it is time to thank the Subversion designers for using Berkeley, which has a good recovery mechanism. The command to fix this will have to be run on the repository host and by a user who has write permission to all the files. The best thing to do is run it as the `svn` user, or whichever ID is the

repository owner. Let's see how to fix our current problem by using the `svnadmin recover` command:

```
$ svnadmin recover /repos/testrepo
Please wait; recovering the repository may take some time...

Recovery completed.
The latest repos revision is 129.
```

Since the command is from the `svnadmin` client, it is run on the repository path, not a URL. Once the command is complete, it will tell you which revision the repository is at. Now you should be able to resume normal activity.

8.3 Berkeley DB

Since Subversion uses the Berkeley DB for the repository filesystem, it is possible to customize the back end. Even though this is entirely possible, the databases created with the `svnadmin create` command will be set up for Subversion and should not be toyed with unless you are sure you understand what the impact will be. While we will not get deep into the configuration of the DB, Subversion provides a couple of commands that allow cleanup and administration. We already saw one example of this with the `svnadmin recover` command; now let's look at transaction logs, which are a key component of the Berkeley DB.

8.3.1 Transaction logs

In the early chapters we spent a good deal of time talking about atomic commits. We saw that Subversion will actually build the new tree, and once everything is okay, it will be added as a new revision of the repository. All of these actions are done through transaction logs, which can get quite large. Thankfully, the default configuration is set to automatically remove unused logs, so the system will be kept optimized in terms of disk space. You can see this setting in the `/$REPO/db/DB_CONFIG` file. The size of each log file is also set in this file:

```
set_lg_bsize     262144
set_lg_max      1048576
set_flags DB_LOG_AUTOREMOVE
```

These settings, in order, are the memory buffer size of the logs, the maximum size a file can reach before rolling to a new log, and finally the switch to turn on the autoremove option for unused log files. You can change these values, but `lg_max` must be at least four times larger than `lg_bsize`. If you wish to turn off the automatic log remove option, comment out the `set_flags DB_LOG_AUTOREMOVE` line

and run the `svnadmin recover` command to force the change. This will prevent the logs from being removed, and now you will be responsible for managing disk space at that point.

Listing the transaction logs. Subversion offers a command called `svnadmin list-dblogs` to list all the transaction logs in the Berkeley DB. The command needs only the path to the repository as an argument:

```
$ svnadmin list-dblogs /repos/testrepo
/repos/testrepo/db/log.0000000096
/repos/testrepo/db/log.0000000097
```

This is not all that exciting—or useful, for that matter. You could have just as easily run an operating system `list` command in the `db` directory of the repository. There is a variation to this command that is more useful called `svnadmin list-unused-dblogs`. This will give you all the log files that are not being used by the back end. Let's run this in the same repository:

```
$ svnadmin list-unused-dblogs /repos/testrepo
/repos/testrepo/db/log.0000000096
```

This output tells you that the transaction log `log.0000000096` is no longer needed and can be removed. If you do not have the automatic removal turned on, you will need to delete this file or it will sit forever on the system. Even when you have this option turned on, there will almost always be one unused file. The old files get cleaned up when a new one hits the size limit and "rolls over." You can remove this file if you like, but since they are set to only 1MB, there will not likely have a large impact.

8.3.2 Old transactions

Just as you can have deadlocks in the working copy, you can have old or unfinished transactions in the repository. For example, earlier we explained that the DB can require you to run the `recover` command to fix any issues it may encounter. In either case, there can be unfinished transactions sitting around that need to be cleaned up. To check for these, you can run a command from the `svnadmin` client that will list them:

```
$ svnadmin lstxns /repos/testrepo
9c
```

The `lstxns` command will search through the database and find any outstanding transactions. If there are any, the transaction number will be displayed. If you want additional information, you can use the `info` command from the `svnlook`

client. We will talk more about `svnlook` later. For now, take a look at the following command to see what the transaction information looks like:

```
$ svnlook info /repos/testrepo --transaction 9c
alex
2004-06-17 14:23:45 -0400 (Thu, 17 Jun 2004)
35
Fixing a bunch of checkstyle errors
```

This command will tell you who was executing the transaction and when it was started. The last line is the log message of the operation; this should give you enough information to know which command failed so that you can reexecute it. Before you do this, however, it would be prudent to clean up the defunct transactions.

Cleaning up. When you do come across old transactions in the database, you will need to remove them. Subversion gives you a command to do this, so you do not have to work on the Berkeley DB directly. This command is called `svnadmin rmtxns`, and it requires the path to the repository and the name of the transaction you are removing:

```
$ svnadmin rmtxns /repos/testrepo 9c
Transaction '9c' removed.
```

You should remove an old transaction only after you have examined it and are sure you no longer need the contents.

8.4 Hooks

Subversion provides the ability to run hook scripts or triggers on certain actions on the repository; this is the same concept as a trigger in a database. You match a script to an action, and when the action happens, the script is executed. If the script fails, the next action will not happen, so you have a way to control whether an operation is valid, based on any executable you want. Before we start writing scripts, we need to identify the different types of actions in Subversion that can trigger a hook script.

There are five types of actions in Subversion for all commands. Each of these actions will have one hook associated with it. The script will have the same name as the action, and we will look at this in the next section. All the actions are based on write operations to the repository and fall into two categories: commit and revision properties.

8.4.1 Commit actions

Any command that changes the contents of a file or directory in the repository is considered a commit from the hook standpoint. This can be a `commit`, `import`, or something else such as a `copy` done using URLs. An operation that writes to the repository has three main steps, as we have mentioned throughout the book. The repository receives the request, the transaction is processed in a "temporary" space, and finally the transaction is moved into the repository. Each of these steps can have a hook action associated with it, however unlike the revision property actions, the hook is not required. If the hook does not exist for one or all of the commit actions, Subversion will just move along. Let's look at each of them in order of processing.

Start-commit. The start-commit action happens when the request is received and before the transaction is processed. The script will receive two arguments: the name of the repository and the name of the user running the command. Since all you have is these two pieces of information, this hook is mostly limited to checking for authorization. If this script does not give back a zero for the return code, the transaction will be aborted. So, for example, if you are checking for a user's ability to write to the repository, returning `-1` will prevent the operation from happening.

Pre-commit. After the transaction is processed, the operation is in the pre-commit phase. Even though the transaction is complete, Subversion has not saved it to the repository. The script will accept the transaction number of the name of the repository. By passing the transaction number into the `svnlook` command, you can get a great deal of insight into the revision about to be added. We will look at the `svnlook` client in detail in the next chapter, but we can examine its potential uses at a high level now. Remember, each transaction creates a new revision, which is a snapshot of the repository, so you can use the commands in `svnlook` on the transaction and previous revisions to do any checks you want. This action probably provides more potential options than any other. Some common uses for hooks here are granular access controls, log message verification, and commit approval mechanisms. This is your last chance to stop the commit; if your `pre-commit` hook returns a non-zero value, the operation will not go through.

Post-commit. After the transaction has been written to the repository, Subversion will process the `post-commit` hook. At this point it is too late to stop the operation from happening, so the scripts here are used for notification or logging. Since the repository has been written to, Subversion will pass this hook the name of the repository and the revision number this operation created. You can use the `svnlook`

command to get information about the revision or just use the `svn` client commands with a URL using the `file://` protocol on the repository server. The most common use of this hook is for an email notification for each commit. In many development groups, a distribution list is set up for commit notifications so that everyone can see what changes are being made. Another use of the post-commit action is to update a tracker software entry. For example, you can set up a script to update a bug tracker entry to complete based on the bug number in the log message. Also, if you have strict backup needs, you can trigger a backup after each commit.

The commit actions happen when the operation is acting on the contents or properties of the file. The caveat to this is that the properties changes do not include versioned properties. There is a different pipeline when the operations are working on properties.

8.4.2 Revision property actions

In chapter 6 you saw the difference between properties attached to an entire object and versioned properties that are attached to a specific revision of the repository. Examples of these are the log message and the author of a specific revision. Since these are non-versioned, you cannot get the original data back after you make a change. Because of this difference, Subversion assumes that you will want a different set of hook scripts to handle this type of transaction. As mentioned previously, for commit actions, a hook script is not required for processing. If the script does not exist, the operation will continue. This is not the case with revision property actions; they will not run unless the hook script runs and returns a zero for the return code of the script. Another difference with these actions compared to the commit actions is the number of steps. The revision property changes have only two actions: `pre-revprop-change` and `post-revprop-change`.

Pre-revprop-change. The `pre-revprop-change` hook is required for Subversion to allow you modify a version property. Since a change to a versioned property is final, there are two main uses for this hook. The first is for permission. You may want to set up some sort of access control list for who is allowed to make these changes. The second suggested use of this hook is for logging changes. Since there is no getting the information back once you change it, for auditing purposes you may want to keep a log of the modifications.

Post-revprop-change. The `post-revprop-change` action should be fairly obvious to you by now. This action will run after the property has been successfully changed and will mainly be used for notification, similar to the post-commit email we mentioned earlier. You may need to have a message going out to all

developers, or this might go just to administrators of the repository so that they can see what happened. You can also use this hook for logging, but the original data will be gone by now, so it makes more sense to do the log messages on the `pre-revprop-change` hook.

8.4.3 Scripts

There is not a lot of magic to creating hook scripts; in fact, Subversion provides templates you can start from. All the hooks need to be in the `$REPO/hooks` directory. As we said earlier in the section, the names of the scripts are critical; they must be named after the action. In UNIX, there will not be an extension, so the script for the pre-commit action will be called `pre-commit`. If you are in a Windows environment, you will need to be able to call `pre-commit` as an executable, so it can be named either `pre-commit.exe` or `pre-commit.bat`. Let's take a look at a directory with all hooks in it:

```
$ cd /repos/testrepo/hooks

$ ls
post-commit               pre-commit               start-commit
post-commit.tmpl          pre-commit.tmpl          start-commit.tmpl
post-revprop-change       pre-revprop-change
post-revprop-change.tmpl  pre-revprop-change.tmpl
```

First, you may notice that each action has a file that ends in `.tmpl`; these are the hook templates we talked about earlier. You can just copy these to the action name with no extension to get started. These templates provide a good working example of what you can do inside a hook script. Once you have these hook scripts created, they must exit cleanly for any operation to execute.

Where to do the work. The hooks can be any executable, but in most cases you will use a script, either a shell script in UNIX or a batch script in a Windows environment. While you can do all the processing in these scripts, a better way to manage your scripts is to have the hook script call other programs to do the work. For example, in the `post-commit` hook, we might be sending an email notification and logging the commit. Instead of doing all this in `post-commit`, it will call two separate scripts for each task. This will keep your administration cleaner, and if you keep them in a common location, multiple repositories can use the same functionality.

When the hook gets executed, the action will send it a set of arguments that is different for each one. These arguments are the means by which the script will get the details about the operation. In the next few sections we will take brief look at each of the possible hook scripts, including the arguments that get passed in. You

can use these hooks for just about anything you can dream up, so the following will serve as an example just to get you started.

Start-commit script. The `start-commit` hook will get only the name of the repository the operation is working on and the name of the user running it. Let's look at a simple example of a custom authorization script for checking a user's privilege to write to a repository:

```
#!/bin/sh
REPOSITORY_NAME="$1"    # argument1 - Path to the repository
USER_NAME="$2"          # argument2 - User running the operation
SCRIPT_DIR=/usr/local/svn_local

RETVAL=`$SCRIPT_DIR/check_commit_write_access \
  $REPOSITORY_NAME $USER_NAME`
if [ $RETVAL -ne 0 ]
then
  echo "You do not permissions to perform this operation" > /dev/stderr
  exit $RETVAL
fi

exit 0
```

The `check_commit_write_access` script would just compare the username to a list of approved users for the repository in the operation. Notice the `echo` statement in the script. If a hook script fails, all the text that went to standard error will be displayed, so you can tell the user if, and why the action was aborted. If the hook script exits cleanly, no output will be displayed.

Pre-commit script. The `pre-commit` script gets only two arguments: the repository name and the transaction number that was processed. By using the `svnlook` command, you can see all the information about the revision that is going to be added to the repository. Let's assume we are using a bug tracking system in addition to Subversion. Any commit made to the repository must include the tracker number somewhere in the log message. We can create a hook to check for this that will fail if it does not exist. First, let's see what the `pre-commit` hook looks like:

```
#!/bin/sh
REPO="$1"       # argument1 - Repository Path
TXN="$2"        # argument2 - The transaction number of the operation

SCRIPT_DIR=/usr/local/svn_local
$SCRIPT_DIR/check_log $REPO $TXN
RETVAL=$?
if [ $RETVAL -ne 0 ]
then
  echo "The log message must include a tracker number" > /dev/stderr
```

```
  echo "In the form of tracker #xxx" > /dev/stderr
  exit $RETVAL
fi
exit 0
```

This looks similar to the `start-commit` hook we looked at earlier. Basically, we call the `check_log` script, which will verify the log message. If that comes back okay, we will exit with a zero so the transaction can move on. If it does not return with a zero, the hook will exit with the error code from the `check_log` script and the action will fail. Let's take a look at what the `check_log` script might contain:

```
#!/bin/sh
REPO="$1"
TXN="$2"

### Get the log message of the transaction
LOG_MESSAGE=`/usr/bin/svnlook log --transaction "$TXN" "$REPO"`

### See if the Bug Tracker number is there
COUNT=`echo $LOG_MESSAGE | grep -i "tracker #[0-9]*" | wc -l`
if [ $COUNT -eq 0 ]
then
  exit 1
else
  exit 0
fi
```

Now when you commit to the repository, your log message must contain the text "tracker #xxx" or the action will fail. Look at what happens when we try to run a commit:

```
$ svn commit --message "changed number of times we loop through"
Sending        c/main.c
Transmitting file data .svn: Commit failed (details follow):
svn: MERGE request failed on '/svn/repos/testrepo/trunk/source/c'
svn:
'pre-commit' hook failed with error output:
The log message must include a Bug Tracker number
In the form of tracker #xxx
```

The log message did not contain the proper text, so our script caught the problem. Also notice that since we had the error, output was displayed to inform the user of the reason for the error. In order to fix this problem, we will need to use the proper log message:

```
$ svn commit --message "Fix tracker #435 changed number of times we loop
   through"
Sending        c/main.c
```

```
Transmitting file data .
Committed revision 143.
```

Since the text "tracker ###" is in the message, the hook script will pass and the commit will finish successfully. While this feature may not be of use to you, it shows you how to dig into a transaction for validity before allowing it to save to the repository. Also, you are not limited to one check; you could call as many other scripts as required for your development process.

Post-commit script. The `post-commit` script is used for notification only. Even if the script fails, the commit has already happened, so this isn't the place to check for valid transactions. Since the transaction has been completed and attached to the repository, a new revision number will be created. This number along with the path to the repository are passed in as arguments to the hook. Using this information, you can get all the details about the last operation and any previous ones you may need for comparison. Probably the most common use for this hook is email notification of a commit. In fact, Subversion comes with a perl script to send out the email for you. Look at the following example `post-commit` script:

```
#!/bin/sh
REPO="$1"     # argument1 - Repository Name
REV="$2"      # argument2 - Number of the revision this operation created

commit-email.pl "$REPOS" "$REV" svn-dev@mydomain.com
```

The first thing you may notice about this script is that there is no exit condition. Since the transaction is complete and the status of the post-commit action does not matter to the operation, we don't need to check the return code. Each time a command writes to the repository, an email will go out the user, or in this case the distribution list, with information about the change. The following is an example of what the email will look like:

```
Author: jeff
Date: 2004-06-19 17:05:34 -0400 (Sat, 19 Jun 2004)
New Revision: 145

Modified:
   trunk/source/java/main.java
Log:
Used function to print output statements instead of doing it in the main

Modified: trunk/source/java/main.java
===================================================================
--- trunk/source/java/main.java 2004-06-19 20:28:50 UTC (rev 144)
+++ trunk/source/java/main.java 2004-06-19 21:05:34 UTC (rev 145)
@@ -2,9 +2,12 @@
```

```
 //
 public class main
 {
-
+  public m_hello = new hello() ;
+  public m_bye = new goodbye() ;
+
   public static void main( String args[] )
   {
-    System.out.println("Hello World");
+    m_hello.printIt() ;
+    m_bye.printIt() ;
   }
 }
```

The default email that Subversion provides will give you all the change log information, plus the diff of the last revision, so you can see what has changed. You can find the `commit-email.pl` script in the `/tools/hook-scripts` directory of your Subversion installation directory. In addition to this one, there are some other scripts you can use for various hooks. There are always more scripts being added, so it won't hurt to check the new releases of Subversion occasionally. Chances are if you have a need for a hook script, so does someone else. They may have already submitted theirs as a patch into the code base, so you don't have to write it yourself—the beauty of open source!

Pre-revprop-change script. Of all the hook scripts, you should take `rep-revprop-change` the most seriously. If fact, this hook must exist in order for any command that changes a versioned property to run. Since this is your last line of defense before changing a revision property that you cannot get back, there are a couple of things you may want to consider: authorization checks and logging. We will look at a simple example. First, consider the hook itself:

```
#!/bin/sh
REPO="$1"       #arg1 - Path to the repository
REV="$2"        #arg2 - Revision number the operation is working on
USER="$3"       #arg3 - User running the operation
PROPNAME="$4"   #arg4 - Name of the property being changed
SCRIPTDIR=/usr/local/svn_local

### Check access to change properties for this user
$SCRIPTDIR/check_revprop_access "$REPO" "$USER" "$PROPNAME"
RETVAL=$?
if [ $RETVAL -ne 0 ]
then
  echo "User: $USER does not have access to change" > /dev/stderr
  echo "the property $PROPNAME in the repository $REPO" > /dev/stderr
  exit $RETVAL
```

```
fi

### Log the change.  We will make sure this works, so the
### operation will not run if we can't log it
$SCRIPTDIR/log_revprop_access "$REPO" "$REV" "$USER" "$PROPNAME"
RETVAL=$?
if [ $RETVAL -ne 0 ]
then
  echo "Failed to log property change so action has been aborted"
  exit $RETVAL
fi

### All scripts passed, exit with a zero so the operation will run
exit 0
```

This script looks just like the other hooks we have seen so far, except that here we have multiple checks instead of just one. Notice that we have more command-line arguments compared to other actions as well. This is because a transaction is not built ahead of time like the commit actions. Because of this, we must get all the information up front. Let's take a look at what the `check_revprop_access` script might look like:

```
#!/bin/sh
REPO="$1"       #argument1 - Path to the repository
USER="$2"       #argument3 - User running the operation
PROPNAME="$3"   #argument4 - Name of the property being change
REPO_NAME=`echo $REPO | awk -F/ '{print $NF}'`
ACLDIR=/usr/local/svn_local/etc
ACLFILE=$ACLDIR/revprop_acl_${REPO_NAME}

echo "ACLFILE is $ACLFILE"
### The ACL File will have a line for each type of property.
### Following the property name will be a comma-separated list
### of users who are allowed to change the access
###
### svn:log = jeff,alex
### svn:author = alex

### Get the line containing the property name
### and see if the username is in it
LINE=`grep $PROPNAME $ACLFILE`
COUNT=`echo $LINE | grep $USER | wc -l`
if [ $COUNT -eq 0 ]
then
  exit 1
else
  exit 0
fi
```

Again, this script is not very robust, but it should give you an idea of what a `pre-revprop-change` script can be used for. If this script does not return a zero, the operation will fail and the client command will be notified, just as we saw in the `pre-commit` hook earlier. If the hook exits cleanly, the property will be changed and the `post-revprop-change` action will run.

Post-revprop-change script. The final action that will trigger a hook script is the `post-revprop-change` script, which will run after the change to the repository is written. At the point, this hook is executed, the change has been written, and there is no way to stop it or even see what the data looked like previously. This leaves notification as one of the only uses for a hook here. The `post-revprop-change` action sends the same arguments to the hook as the `pre-revprop-change` action. Similarly to the email script for the `post-commit`, Subversion provides a script for this action. We will look at the `post-revprop-change` script, which should seem familiar to you:

```
#!/bin/sh
REPO="$1"       #arg1 - Path to the repository
REV="$2"        #arg2 - Revision number the operation is working on
USER="$3"       #arg3 - User running the operation
PROPNAME="$4"   #arg4 - Name of the property being changed
SCRIPTDIR=/usr/local/svn_local

$SCRIPTDIR/propchange-email.pl "$REPOS" "$REV" "$USER" "$PROPNAME" \
   svnadm@domain.com
```

Remember, we do not need an exit status here because it is a post script, so no matter happens, the operation is complete.

8.5 Summary

In this chapter we started to reach some of the "far corners" of Subversion. While many people may not use these features, if you can master them, the repository can really start working for you. Subversion was designed by software developers to be used by development teams. Much of it can be automated or set up to streamline development, so the tool is not an impediment. Since these tools and configurations have a great deal of power, you need to make sure you understand them before using them for real. You may want to have a test repository as a sandbox for these types of changes so you do not corrupt or get your production instance into a bad state.

9 Subversion utility clients

In this chapter

- Examining transactions
- The svnlook client
- Repository revision information
- Filtering the dump and load

We have looked extensively at the `svn` and `svnadmin` clients, which will probably be the majority of your Subversion use. There are a couple of additional clients that may not be used frequently but can be very powerful in certain circumstances. For example, we will explore the `svnlook` command, which can examine transactions that are not yet in the repository. This will allow you to completely examine an operation in one of the hook scripts before it is saved to the repository. Since much of the content in this chapter may not be used frequently, it is more important that you understand what the utilities can do. When you come across a situation where you need some of the functionality, you can come back to review the details.

9.1 The svnlook interface

The `svnlook` client gives you the ability get information about the repository or specific revisions. This command is run on the path of the repository from the local host and cannot be used with a URL. In addition to looking at the repository, the client can also look at a transaction in progress as if it were the latest revision of the repository. All of this makes the `svnlook` client a good candidate for use in hook scripts.

9.1.1 Transactions and revisions

As we stated, the `svnlook` client can work off the repository or a transaction that is in process and not yet saved. Since this transaction will be the next revision saved to the repository, the `svnlook` client just assumes it is already there. This will allow you see the repository as it will appear after the operation is done. As we explained in the previous chapter, this is ideal for hook scripts to see if the operation will meet all the requirements you have for writing to the repository (authorization, log message constraints, etc.).

Just about all the subcommands in the `svnlook` client can work off a revision of the repository or a transaction not yet committed. You make this distinction by adding a switch to the client. If you want to look at an object already in the repository, you would use the `--revision` option. This is just like the `--revision` option you have worked with in the `svn` client interface. So if you wanted to get the log message of revision 49, you would use the following command:

```
$ svnlook log --revision 49 /repos/testrepo
fixed checkstyle error
```

As with the switch in the client interface, you can use any of the keyword or date functionality with the `--revision` option. If you are writing a hook script and want

to work on the operation that is in progress, you will need to work off the transaction. To do this, use the `--transaction` switch instead of `--revision`. You will need to know the transaction number when using this option. If this is a hook script, the transaction number will be passed in by the action; otherwise, you will need to use the `svnadmin lstxns` command to find it. Let's assume we ran this command before revision 49 was committed, and it was transaction 9c:

```
$ svnlook log --transaction 9c /repos/testrepo
fixed checkstyle error
```

The results of the command will be the same; the only difference is where you tell Subversion to look for the information.

No options. Think back to earlier chapters and remember the golden rule about specifying revisions. If you do not specifically tell Subversion which revision you want in the repository, it will default to the `HEAD`. This is the same for the commands in the `svnlook` interface. If you do not give the commands a revision number or transaction number, they will work off the `HEAD` revision on the repository.

9.2 Getting change log information

The majority of the time you will likely use the `svnlook` interface to get change log information about a revision or transaction. Whether it is for a hook script or administrative functions, you will want to see such things as who made the change and when it was done. Each piece of information in the change log can be accessed with a specific command. When you use the commands in this section, you will need to provide a revision number or transaction number. Since the change information is associated with a revision of the repository and not specific files, the only other argument you will need to give is the path to the repository.

9.2.1 Getting the author

Let's say you want to develop a more granular set of access controls in a `pre-commit` hook script. The most critical piece of information you will need from the transaction is the user ID running the command. You can determine the user responsible for the transaction or a previous revision by using the `svnlook author` command:

```
svnlook author --transaction $TXN $REPO
```

This will return the author of the transaction passed in. If you want to run this from the command line or on a known revision, you can use something like the following:

```
$ svnlook author --revision 51 /repos/testrepo
alex
```

The output shows the user who created the revision. Whether you are using this command on transactions or revisions, the user ID will be the one on the server side. So in our example, the user `alex` was defined in the network protocol used to access this repository, whether Apache or svnserve.

9.2.2 Finding the date of a revision

Similar to the `author` command, you can pull the date of a revision or transaction by running `svnlook date`. This will return the timestamp of the revision as it was (or will be) saved to the repository:

```
$ svnlook date --revision 51 /repos/testrepo
2004-05-13 21:56:04 -0400 (Thu, 13 May 2004)
```

The date will be sent to standard out in the format shown in the example. There are no formatting options with this command, so you will have to be content with the default format.

9.2.3 Viewing the log message

The last part of the change information is the log message associated with a revision or transaction. In chapter 8, we used an example of checking for a valid log message where it was necessary to enter a tracking number. You may want to get the log message for a validation, for logging, or possibly in some kind of search capacity. No matter what the use, the `svnlook log` command will give you a log message:

```
$ svnlook log --revision 55 /repos/testrepo
Added property label with a value TEST
to the file main.java
```

Each line in the log message will be displayed for the revision or transaction.

9.2.4 Getting all the change information

We have shown how to get the change information from a revision or transaction in individual commands, but what if you want to get everything at once? Subversion provides a command to get all three pieces of information at the same time. This is done with the `svnlook info` command and is basically the same as running the `svnlook author`, `log`, and `date` commands:

```
$ svnlook info --transaction 9 /repos/testrepo
jeff
2004-05-14 21:06:10 -0400 (Fri, 14 May 2004)
60
```

```
Added property label with a value TEST
to the file main.java
```

The output of this command should be obvious to you. It shows the author of the revision followed by the date on the second line. The third line is the number of characters in the log message. The remaining output is the log message itself. These commands have no other options such as `verbose` to get additional information. One key piece of information that you may want is the path of the files that changed, which is not available from the `info` command. While you cannot do this with extra options, there is an additional command that will provide this information.

9.2.5 Finding the files that changed

In you need to find the files that changed in a particular revision, you can run the `svnlook changed` command. This is basically the same as adding the `--verbose` option to the `svn log` command we used in chapter 4.

```
$ svnlook changed --revision 9 /repos/testrepo
A   c/hello.c
U   java/hello.java
```

This output should look familiar. All the files and directories that were modified in the commit of that revision will be listed. In addition to the filenames, the operation will be shown as well. In this example, `hello.c` was added and `hello.java` was changed. If a property was changed, the second column will be populated with a character.

9.3 Getting information on properties

If you ran the `svnlook changed` command and saw a character in the second column, you would know that a property on the object changed. Suppose in the example in the previous section that we received output that looked something like this:

```
$ svnlook changed --revision 146 /repos/testrepo
_U  trunk/source/java/main.java
```

You can see that the `U` character is in the second field of the output. In fact, in the `svnlook` interface it is obvious because there is an underscore in the first field to indicate that it is empty. We know from the `status` command in chapter 2 that the second column indicates that properties were modified. To see what the property is in this version, you can use the `proplist` and `propget` commands.

9.3.1 Listing the properties

If you know that there are properties attached to the file but are not sure what they are, you can use the `proplist` command to show everything that is attached to the file or directory. This command is just like the `svn proplist` command but with fewer options. You must provide the path to the repository and the path to the object you are looking for inside the repository. For example, to list all the properties on the file `main.java` that changed in the previous section, you would use the following command:

```
$ svnlook proplist --revision 146 \
/repos/testrepo trunk/source/java/main.java

  STATE
```

There is only one property attached to this file and it is named `STATE`. Like the other commands we have used so far, `proplist` can be used on transactions before they are committed to the repository.

Getting the properties on a transaction. Remember, these properties are attached to the object and can span multiple versions of the file or directory. Since we added them with the `propset` command and used the `propedit` command to change their value, they need to be committed in order for the change to be saved in the repository. So when you are setting up a hook to check properties, it will be in the `pre-commit` action and a transaction will be created. This will allow you to check for specific properties before the change is saved to the repository. For example, let's say you are using a branching strategy where all changes that are saved to the trunk must have a release property associated with them. The property name must be `release` and it needs to be on every file and directory in the trunk. You could write a little hook script that runs the `proplist` command on the transaction before it is saved. Let's look at a simple example of this:

```
$ cat check_release
#!/bin/sh
REPO="$1"
TXN="$2"

for file in `svnlook changed --transaction $TXN $REPO  | grep trunk | awk
   '{print $2}'`
do
  COUNT=`svnlook proplist --transaction $TXN $REPO $file | grep -i release |
   wc -l`
  if [ $COUNT -eq 0 ]
  then
    echo "$file does not have release tag on it"  > /dev/stderr
```

```
    exit 1
  fi
done
exit 0
```

In this script, we are simply looping through all the objects that were changed, and for each one that is in the trunk, we get the list of properties attached to it. If any of them does not have the release property name associated with it, we fail by exiting with a non-zero. All you need to do is call this from your `pre-commit` hook (see chapter 8), and if someone tries to commit without the file containing a release property, the command will fail:

```
$ svn commit --message "changed state to test on"
Sending        trunk/soujava/hello.java
Transmitting file data .svn: Commit failed (details follow):
svn: MERGE request failed on '/svn/repos/testrepo/trunk/source/java'
svn:
'pre-commit' hook failed with error output:
trunk/source/java/hello.java does not have release tag on it
```

You will have to either add the release property to this file or create a branch and work there (since you can check for files only in the trunk). The `proplist` command is one of the few commands in the `svnlook` interface that has options for changing its behavior.

Getting the property name and value with proplist. If you want to get the name and the value of all the properties on an object, you can add the `--verbose` option to the `proplist` command. This acts just like the `verbose` option on the `svn` client version of the `proplist` command we used in chapter 6. You can use this option in conjunction with revisions or transactions and get the same results. Let's run this command on a revision of the repository:

```
$ svnlook proplist --verbose --revision 150 \
 /repos/testrepo trunk/source/java/hello.java
  release : 1.0
  STATE : test
```

Now in addition to printing the name of the properties on this file, the command will give you the values. You can have multiword and multiline values on the properties. Each property name has the ":" after it so you will know when a new property starts.

9.3.2 Getting the value of a property

In the previous section we used an example where we wanted to check for a property on an object and we did not care what the value was. Now consider a little different example. Let's say you have a property called `STATE` on all the files in your repository.

This represents where that revision is the development lifecycle (development, test, QA, and production). You are trying to keep the code in the trunk pristine, so the only things that can get checked in there are files with the `STATE` property set to `production` (we will assume that there is another hook that allows only the QA tester to set the value to `production`). When a user tries to save something to the trunk, we will need a hook that checks the value of `STATE` before allowing the action to complete. We could use the `proplist` command and just parse out the value of the property, but there is cleaner way. We can use the `svnlook propget` command, which will ask for the name of the property as an argument. It will return only the value of the property, so there is no parsing of extra characters required. First, let's look at an example of this on the command line, specifying a revision number:

```
$ svnlook propget --revision 151  /repos/testrepo \
STATE  trunk/source/java/hello.java

test
```

Along with the `--revision` switch, we pass the path of the repository, the name of the property, and the path of the object we are checking. The command will output the value of the property we asked for. Now let's tweak the hook script in the previous section to meet the requirements of having a specific value on the property:

```
$ cat check_trunk_prod
#!/bin/sh
REPO="$1"
TXN="$2"

for file in `svnlook changed --transaction $TXN $REPO  | grep trunk | awk
   '{print $2}'`
do
  VAL=`svnlook propget --transaction $TXN $REPO STATE $file`
  RETVAL=$?
  if [ $RETVAL -ne 0 ]
  then
   echo "$file must have STATE set to production to enter trunk"  > /dev/stderr
    exit 1
  fi
  if [ $VAL != production ]
  then
   echo "$file must have STATE set to production to enter trunk"  > /dev/stderr
    exit 1
  fi
done
exit 0
```

This script just takes the property checking one step further than the one in the previous section. Not only does `STATE` need to be attached to the object, it needs a specific value or it will fail and the action will come back with an error.

9.4 Finding out what changed

So far, everything we have seen with the `svnlook` interface will tell us about the properties or change log information, but there may be instances when you need more details on the change. You may want see the changes in a file that are being saved or perhaps the entire contents of a file. We will look at some subcommands in the interface that allow you look inside a file or directory to get more information.

9.4.1 Getting the differences between revisions

One of the common options for a `post-commit` hook script was to send out an email notification of all writes to the repository. Something you may want to include in the email is the changes to the file(s) so everyone can see what has been modified. The `svnlook` interface provides a `diff` command like the one you saw in chapter 4. It will accept a transaction or revision and compare all the files that changed to their previous versions. Let's take a look at the command working against a revision:

```
$ svnlook diff --revision 152 /repos/testrepo

Modified: trunk/source/java/hello.java
===================================================================
--- trunk/source/java/hello.java        2004-06-22 00:49:04 UTC
(rev151)
+++ trunk/source/java/hello.java        2004-06-22 02:55:14 UTC
(rev152)
@@ -1,6 +1,6 @@
 public class hello

 {
-  String m_output ;
+  String m_output = new String() ;
   public hello()
   {
     m_output = new String( "hello world" ) ;
Property changes on: trunk/source/java/hello.java
```

Differences in the file contents

```
------------------------------------------------------------------
Name: STATE
   - test
   + production
```

Changes in value of the file properties

In this example, only one file, `hello.java`, has changed. In addition to the contents being modified, you can also see that a property has changed. The name of the property is given first; in this case it is `STATE`. The string with the - character is the previous value, and the + is the new value. So in revision 152 the value of `STATE` was changed from `test` to `production`. If you wanted to put this in an email script, you would use `--transaction` instead of `--revision`. When you do this, the transaction files will be compared to the `HEAD` of the repository. So the command will show you the changes should the transaction be successfully saved. With the `diff` command, you will see only the lines in the file that have changed, not the entire file.

9.4.2 Viewing the entire contents of a file

We just showed how to view the changes in a file in one revision from its predecessor. What if you need to view the contents of a file? The `svnlook` interface has a `cat` command to simply output the contents of a file. Let's say you want all your source files to be run through a checkstyle parser before being committed to the repository. This program will read all the lines in the file and make sure the file fits your requirements for how the code should look, variable name formats, etc. You will need to pass all the lines into a checkstyle parser to accurately check the file. The following script utilizes `svnlook cat` and could be used in a `pre-commit` hook:

```
#/bin/sh
REPO="$1"
TXN="$2"
for file in `svnlook changed --transaction $TXN $REPO | awk '{print $2}'`
do
  svnlook cat --transaction $TXN $REPO $file > /tmp/svn.$TXN.$file
  /usr/local/bin/check_style_app /tmp/svn.$TXN.$file
  RETVAL=$?
  if [ $RETVAL -eq 0 ]
  then
    echo "$file did not pass checkstyle program"  > /dev/stderr
    exit 1
  fi
done
exit 0
```

The `svnlook cat` command takes the repository name and the path to the file you want to view. You can use the `--transaction` or `--revision` option depending on how you are running the command. Since `cat` dumps the contents of a file, you

cannot run this command on a directory. The next command we will discuss shows you how to look at directory information.

9.4.3 Finding the directories that changed

You may have a situation where detailed access control lists are required in your system, perhaps at a directory level. For each file changed in a transaction, you will need to find out which directory it is in and check that against your ACL. Subversion provides you an easier way to do this. The `svnlook dirs-changed` command will give you a list of directories that have changed in the revision or transaction. Let's take a look at the command against a revision first:

```
$ svnlook dirs-changed --revision 135 /repos/testrepo
trunk/source/c/
trunk/source/java/
```

In revision 135, files in the two directories listed have changed. Running this command on the transaction to see what the user is trying to change can make implementing access control very easy.

9.4.4 Looking at the entire tree

Most of the commands we have used in the `svnlook` client have a counterpart in the `svn` interface. We use the `svnlook` client for administrative tasks and scripting. One command that is not in the `svn` client is `tree`. By running this command, you will get a hierarchical layout of your entire repository. This can be used for design diagrams, documentation, layout integrity checks, or any other use you may have. You can run `tree` against a revision or a transaction if you want to use it in a hook script. By giving a revision number, you will get the layout at that point, which may not be the same as the `HEAD`. The only thing you need to pass besides the `--transaction` or `--revision` option is the path to the repository:

```
$ svnlook tree --revision 150 /repos/testrepo
/
trunk/
  source/
   java/
    hello.java
    main.java
    goodbye.java
   c/
    run_main.sh
    hello.c
    main.c
    LdapAdd.java
 branches/
```

```
  major_rewrite/
   source/
    java/
     hello.java
     main.java
     goodbye.java
    c/
     hello.c
     main.c
tags/
  release_1.0/
   source/
    java/
     hello.java
     main.java
     goodbye.java
    c/
     run_main.sh
     hello.c
     main.c
     LdapAdd.java
```

There is really nothing complex about the output; all the children of a directory are indented underneath it. All directories will have a `/` character at the end of the name, so you can distinguish between a file and an empty directory.

9.4.5 Getting change information on a directory

A common question you may see in the source control arena is how much activity there is in a particular section of the code. Maybe you want to see where your resources are being used, or which area has the most bugs. For example, let's say a development manager wanted to see the differences in activity between the C and Java code. You could run through all the change logs and search, but `svnlook` provides a `history` command that will show all the revisions of a file or directory that changed. If any file or directory changed, all the parents are considered changed also. Therefore, the top level of the repository will be in all the revisions, since everything is under that. Let's run the command on the two directories in the trunk that the manager wanted and see what the results look like (we start this at a lower revision so the output is manageable):

```
$ svnlook history --revision 70 /repos/testrepo trunk/source/java
REVISION   PATH
--------   ----
      70   /trunk/source/java
      53   /trunk/source/java

$ svnlook history --revision 70 /repos/testrepo trunk/source/c
```

```
REVISION   PATH
--------   ----
      70   /trunk/source/c
      69   /trunk/source/c
      68   /trunk/source/c
      67   /trunk/source/c
      65   /trunk/source/c
      61   /trunk/source/c
      58   /trunk/source/c
      57   /trunk/source/c
```

So from revision 70 and lower, there have been two changes in the `java` directory and eight in `c`. You can tell by looking at this that some file or subdirectory under the path we looked at changed in the revision indicated in the left-hand column. You may be wondering why the command needs to include the path since they are all the same; after all, you gave the path as an argument in the command. But think back to chapter 3 when we talked about copying and moving and the same object getting a different name. You could actually get different names in the PATH column.

Different names in the PATH column. If the directory of file you are looking at with the `history` command was created with a `copy` or `move` command, you will see different names in the path for different revisions. Let's look at an example of this with the `java` directory:

```
$ svnlook history --revision 70 /repos/testrepo trunk/source/java
REVISION   PATH
--------   ----
      13   /trunk/source/java
       7   /source/java
```

In this situation, the directory `trunk/source/java` used to be named just `/source/java` until revision 13. This gives you a quick and easy way to see all the names of the target you are looking at.

9.4.6 Getting repository information with svnlook

There are a couple of commands that will give you information about the repository as a whole and not individual revisions or transactions. These can be used as utilities or sanity checks to make sure you are where you think you are. First, we will look at a way to get the identifier of the repository, independent of the path.

Getting the ID of the repository. When a repository is created, a universal unique identifier (UUID) is created. This is a number that sticks with the repository and not its contents. So, for example, if you do dump and load into a new repository,

the UUID will not change for either. Even if you rename a repository, the UUID will stay the same as when you first ran the `svnadmin create` command. To get this information, you can run the `svnlook uuid` command. All that you need to pass in is the path of the repository you are checking:

```
$ svnlook uuid /repos/testrepo
1d60fc7f-69d7-0310-85f1-d6054c0b3515
```

You can use this ID to make sure that you are not working from a backup or that someone renamed the path to a different repository.

Getting the current revision number. When creating hooks and automated scripts, you will likely come across instances where it would be helpful to know what the latest revision of the repository is. This can simply be done with the `svnlook youngest` command with an argument of the repository path:

```
$ svnlook youngest /repos/testrepo
152
```

This will return just the revision number of the repository that is the `HEAD`, or most recent one.

9.5 Other Subversion programs

Subversion comes with couple of other programs besides the ones we have seen: `svn`, `svnadmin`, and `svnlook`. First is the `svnversion` command, which gives you a fast and concise way of getting revision information about your repository. The second interface is `svndumpfilter`, which allows you to adjust the contents of a dump from the `svnadmin` interface.

9.5.1 The svnversion tool

When you run the `svnversion` command against your working copy, it gives you the revision you are working from. This is a convenient way to see at a high level where your working copy is. The command accepts a path to a working copy as an argument and gives back the revision number. Let's say we just checked out a working copy and ran the command from the top level of the repository:

```
$ svnversion .
154
```

So now we know that the working copy was created from revision 154 of the repository. Let's make a change to one of the files and see what differences show up:

```
$ cd testrepo/trunk/source/java

$ vi hello.java

$ svnversion .
154M
```

Now when we run the command, the revision number has an `M` at the end of it. This indicates that your working copy has been modified in some way and that you need to run a commit in order for it to be in sync with the repository. If you want to see the details of the change, you can run the `svn status` command. So let's commit and see if that clears the character:

```
$ svn commit --message "removed tags and used spaces for indentation"
Password for 'jeff':
Sending        java/hello.java
Transmitting file data .
Committed revision 155.

$ svnversion .
154:155
```

Now the extra character has been removed, but we have a range of revisions in the output. When you see this, Subversion is telling you that the working copy has mixed revisions in it. The numbers shown will be a range of the oldest and youngest revisions in the working copy. In this case, we did a commit from the `testrepo/trunk/source/java` directory, which was the only one that updated to the revision. The rest of the working copy is still on revision 154, so you will need to run the `svn update` command at the top of your working copy to get everything at the same revision. This will tell only which revisions you have checked out; you can change this around and ask for the revision range of the last time any files were modified if you like.

Getting the last changed revision. You can put a slightly different spin on looking at the revision range in your repository by basing the search from the last time a file was changed and not the last time it was refreshed. For example, say the file `main.c` has not been updated since revision 60, but each time you do an update, it comes across as the latest revision of the repository because of the atomic commits. If you were more interested in seeing the range of revisions of the file since the last time it was modified, you could add the `-c` option to the `svnversion` command. Let's see how it changes our results:

```
$ svnversion -c .
27:155
```

So in the repository, we have a range of files that have changed, with the oldest being revision 27 and the youngest being 155. You can look back at the `svn status` command for a detailed view on which files or directories were changed in specific revisions.

9.5.2 The svndumpfilter interface

The last of the applications provided by Subversion that we will look at is `svndumpfilter`. This interface lets you adjust the contents created from the `svnadmin dump` command. Let's say, for example, that you have a branch in your repository that was made for a major rewrite of the code. After a long development process, the code and requirements have changed, and you end up with a completely different application. You decide to put this code into a new repository so it can be maintained separately. You could use the `export` command, but all the history and previous revisions would be gone. If you use the `hotcopy` command from the `svnadmin` interface, you will get the new code, but you'll still have to deal with the other code by removing it. By using the `svnadmin dump` and `load` commands, you can populate a new repository. When you add in the `svndumpfilter` command, the contents of what gets dumped will be tweaked. So to fix our problem, we will dump the `testrepo` repository and filter out the `branches/major_rewrite` directory. We can load that into the new repository to get the desired results. The first thing we will do is a regular `svnadmin` dump of the original repository and create a new one (although we will not load it yet). Since the `major_rewrite` branch was created at revision 50 of the repository, we will start there and go to the `HEAD`:

```
$ svnadmin dump --revision 50:HEAD /repos/testrepo > /tmp/dumpfile
* Dumped revision 50.
* Dumped revision 51.
...
* Dumped revision 155.

$ svnadmin create /repos/newrepo
```

Now, before loading the new repository with `dumpfile`, we will run the file through the filter and just take anything in the `major_rewrite` branch:

```
$ svndumpfilter include branches/major_rewrite < /tmp/dumpfile \
> /tmp/rewrite_dump

Include without drop called for prefixes:
'branches/major_rewrite'
```

❶ **Confirmation of what are we including**

```
Revision 50 committed as 50.
...
Revision 154 committed as 154.

Dropped 0 revisions, 291 nodes

Dropped nodes list:
core
core/trunk
...
trunk/source/java/hello.java
trunk/source/c/main.c
```

❷ Revisions read and revisions after the filter

❸ Revisions excluded and objects ignored

❹ Revisions excluded and objects ignored

This interface has two subcommands; we used the first one called `include`. This told Subversion to include anything in the path specified. We used the dump file as input for the command and redirected the output to a new file. While the actual repository information is going into `/tmp/rewrite_dumpfile`, there will be some messages coming to the screen.

❶ The first message just tells you what the command is doing; basically this is a sanity check that confirms that what the command is doing matches your intentions. We will include some options later that will slightly change the behavior and thus the message.

❷ As with the `svnadmin dump` command, the filter will list each of the revision numbers as they are parsed. You will see the original number and the new revision number. Certain options will condense the dump so that the revisions will not always be the same.

❸ After all the revisions have been parsed, the filter will tell you how many revisions were left out (again, we will show this shortly), plus which objects or nodes were excluded. Since we are including only items under `branches/major_rewrite`, all other files and directories will not make it into our new dump file. We have removed 291 files and directories from the `testrepo` repository after running the dump through the filter.

❹ After the summary, the filter will list all the files and directories that were left out. This is just another sanity check that you are getting what you expect in the new dump.

Once you have run the dump through the filter and created the new file, you can use this to load the new repository:

```
$ svnadmin load /repos/newrepo < /tmp/rewrite_dump 2>/tmp/stdrr
<<< Started new transaction, based on original revision 50
------- Committed new rev 1 (loaded from original rev 50) >>>
```

```
...
<<< Started new transaction, based on original revision 154

------- Committed new rev 105 (loaded from original rev 154) >>>
```

Now you can check out the new repository, and it will contain only the files from the `major_rewrite` branch.

Running the filter with an exclude. We previously ran the `svndumpfilter` command and included a specific path; now let's look at an example where we want to exclude something. In the previous example we created a new repository with only the `/ranches/major_rewrite` directory. Let's say management does not want that code stored with the original code in the `testrepo` repository. They decide to create a new repository and do not want anything under the `branches` directory, only the trunk and tags. We can run the `svndumpfilter` command on the same original dump file, only this time excluding the `branches` path, and we will get everything else:

```
$ svndumpfilter exclude branches < /tmp/dumpfile >/tmp/nobranch_dump
Exclude without drop called for prefixes:
'branches'
```

Notice the summary from the output; the filter has excluded `branches`, so we got everything else. Remember, the command assumes that we are starting at the top level of the repository when specifying the path to exclude. Now we can run the load into the new repository, just like before:

```
$ svnadmin create /repos/original_repo

$ svnadmin load /repos/original_repo < /tmp/nobranch_dump
```

Now when we check out a working copy, take a look at the paths in the repository:

```
$ svn co file:///repos/original_repo
A  original_repo/trunk
A  original_repo/trunk/source
A  original_repo/trunk/source/java
A  original_repo/trunk/source/java/hello.java
A  original_repo/trunk/source/java/main.java
A  original_repo/trunk/source/java/goodbye.java
A  original_repo/trunk/source/c
A  original_repo/trunk/source/c/run_main.sh
A  original_repo/trunk/source/c/hello.c
A  original_repo/trunk/source/c/main.c
A  original_repo/trunk/source/c/LdapAdd.java
A  original_repo/tags
A  original_repo/tags/release_1.0
A  original_repo/tags/release_1.0/source
```

```
A  original_repo/tags/release_1.0/source/java
A  original_repo/tags/release_1.0/source/java/hello.java
A  original_repo/tags/release_1.0/source/java/main.java
A  original_repo/tags/release_1.0/source/java/goodbye.java
A  original_repo/tags/release_1.0/source/c
A  original_repo/tags/release_1.0/source/c/run_main.sh
A  original_repo/tags/release_1.0/source/c/hello.c
A  original_repo/tags/release_1.0/source/c/main.c
A  original_repo/tags/release_1.0/source/c/LdapAdd.java
 U original_repo
Checked out revision 85.
```

Under the top level of the repository, there are only two directories: `tags` and `trunk`. While all the code will be in place, you should be aware of some side effects the `svndumpfilter` command can cause with revision numbers. We will look at what they are and how to adjust them.

Empty revisions. When we run the `svndumpfilter` command, it tells us about the nodes (files and directories) that were left out of the new dump. So what happens to revisions of the repository where only those files were changed? For example, let's say revision 85 changed only the file `branches/release-2.0/source/c/main.c`. Since that object was excluded, there is no need to keep that revision. So by default, Subversion will create an empty revision for padding. Take a look at the log messages in the `original_repo` repository and you can see this:

```
$ svn log original_repo | more
------------------------------------------------------------------------
r85 | (no author) | 2004-06-22 21:54:45 -0400 (Tue, 22 Jun 2004) | 1 line

This is an empty revision for padding.
------------------------------------------------------------------------
r84 | jeff | 2004-06-22 21:53:26 -0400 (Tue, 22 Jun 2004) | 1 line

initialized int
------------------------------------------------------------------------
r83 | jeff | 2004-06-21 22:55:14 -0400 (Mon, 21 Jun 2004) | 1 line

called construction for m-output
------------------------------------------------------------------------
r82 | (no author) | 2004-06-21 20:49:04 -0400 (Mon, 21 Jun 2004) | 1 line

This is an empty revision for padding.
------------------------------------------------------------------------
r81 | jeff | 2004-06-21 20:48:35 -0400 (Mon, 21 Jun 2004) | 1 line

changed state property to test on main.c
------------------------------------------------------------------------
```

As you can see, revisions 85 and 82 have been padded with no information because they are no longer relevant. If you want to keep the revision numbers the same on the new repository, the default behavior does this for you. This could be important if you are tracking bugs and revision numbers to keep Subversion in sync with your tracking tool. The log message and author in these revisions are left out so you can easily see that these are padded revisions.

Keeping change log information on empty revisions. If you wish to keep the empty revisions but retain the original log message and author, you can add the `--preserve-revprops` switch on the `svndumpfilter command`. You will still have the padded revisions, but they will look like the original one. Let's run through the filter and load process again:

```
$ svndumpfilter exclude --preserve-revprops \
 branches < /repos/dumpfile > /tmp/nobranch_dump

$ svnadmin load /repos/original_repo < /tmp/nobranch_dump
```

Here we added the switch to preserve the original information. All the output will look the same, but when we check out a new revision of the repository and look at the log messages, we will see something a little different:

```
------------------------------------------------------------------------
r85 | alex | 2004-06-22 21:54:45 -0400 (Tue, 22 Jun 2004) | 1 line

initialized int
------------------------------------------------------------------------
r84 | jeff | 2004-06-22 21:53:26 -0400 (Tue, 22 Jun 2004) | 1 line

initialized int
------------------------------------------------------------------------
r83 | jeff | 2004-06-21 22:55:14 -0400 (Mon, 21 Jun 2004) | 1 line

called construction for m-output
------------------------------------------------------------------------
r82 | jeff | 2004-06-21 20:49:04 -0400 (Mon, 21 Jun 2004) | 1 line

bugfix 324
------------------------------------------------------------------------
r81 | jeff | 2004-06-21 20:48:35 -0400 (Mon, 21 Jun 2004) | 1 line

changed state property to test on main.c
------------------------------------------------------------------------
```

Now instead of having an empty log message and author in revisions 85 and 82, the original information is in place. If we run the log in verbose mode and look at the path, we should be able to confirm that this is from an excluded path:

```
$ svn log --verbose original_repo
------------------------------------------------------------------------
r85 | alex | 2004-06-22 21:54:45 -0400 (Tue, 22 Jun 2004) | 1 line
Changed paths:

initialized int
------------------------------------------------------------------------
r84 | jeff | 2004-06-22 21:53:26 -0400 (Tue, 22 Jun 2004) | 1 line
Changed paths:
   M /trunk/source/c/main.c

initialized int
------------------------------------------------------------------------
```

Notice that even though the original information is in the change log, still nothing shows up in the changed path when we turn on the `--verbose` switch. This happens because the object it points to does not exist. If you have no need for this padding in the revision numbers, you can turn it off by adding an option when you run the filter.

Leaving out the empty revisions. The `svndumpfilter` command has an option called `--drop-empty-revs`, which will not add in the extra padded revisions. If a particular version has only changes for the nodes excluded, it will be skipped altogether. Let's run the `svndumpfilter` command again on the same dump file and add this option:

```
$ svndumpfilter exclude --drop-empty-revs \
branches < /tmp/dumpfile > /tmp/nobranch_dump

Exclude with drop called for prefixes:
'branches'
Revision 0 committed as 0.
Revision 1 committed as 1.
...
Revision 81 committed as 81.
Revision 82 skipped.
Revision 83 committed as 83.
Revision 84 committed as 84.
Revision 85 skipped.

 Dropped 5 revisions, 20 nodes


Dropped nodes list:
branches
...
branches/release-2.0/source/java/main.java
```

If you look at the output from this command, you will see a couple of differences. The first line in the output explicitly says that the "drop revision" was added to the command. Again, this is just a sanity check that shows you and Subversion are on

the same page. Now when you look at the revision list, you will see that some of the revisions are labeled as skipped. These are the versions that contain the excluded files. At the end of the list, you will see the summary, which includes the number of skipped revisions and nodes. In our example, five revisions were dropped from the new dump file. Now we can run the load the same way into the new repository and check out a new working copy. After this, run the `log` command and you will not see any padded revisions:

```
$ svn log original_repo
------------------------------------------------------------------------
r80 | jeff | 2004-06-22 21:53:26 -0400 (Tue, 22 Jun 2004) | 1 line

initialized int
------------------------------------------------------------------------
r79 | jeff | 2004-06-21 22:55:14 -0400 (Mon, 21 Jun 2004) | 1 line

called construction for m-output
------------------------------------------------------------------------
r78 | jeff | 2004-06-21 20:48:35 -0400 (Mon, 21 Jun 2004) | 1 line

changed state property to test on main.c
------------------------------------------------------------------------
```

Notice that the youngest revision is now revision 80, not 85. This is because there were five revisions left out, so there will not be as many. All the information is the same; the revisions numbers have just been mapped differently.

9.6 Summary

The "extra" Subversion commands we have looked at all stemmed from very specific needs and are consequently narrow in scope, especially compared to the `svn` and `svnadmin` command interfaces. Despite this, they can really get you out of a jam since the needs they fulfill can be complex to do by hand. We have demonstrated a common thread through our exploration of Subversion that the tool lends itself well to automation and scripting. While this might not seem like a big deal to an entry-level user, it is a powerful feature. In many development communities, there are no resources for performing configuration management tasks on the repository and code. If you can utilize these automation tools that Subversion provides, much of this administration can be scripted, and you will not tax developers' time by having them deal with version control issues.

10 Third-party tools

In this chapter

- Importing a CVS repository into Subversion
- Subversion plug-in for Eclipse
- Browsing a Subversion repository with a web browser
- Windows interface for Subversion commands

As Subversion matures and becomes more engrained in contemporary software development, more third-party tools will appear supporting it. Since the majority, if not all, of the tools will be open source, you must carefully examine them before deploying them. Take the time to see if the software has had any production releases and if the community that is developing the software is strong. Open source projects that fit these criteria will tend to be more stable and provide better support. In this chapter, we will discuss four tools that are becoming popular as well as stable. We will look at an application to convert a CVS repository to Subversion, a plug-in to a popular IDE, a tool for browsing your repository over a web browser, and finally a Windows interface for Subversion commands.

10.1 Importing a CVS repository

Many people who will start using Subversion will be coming from a CVS repository and will need to bring over the source code. You could simply do an export of CVS and then an import into Subversion, but you would lose any history, in addition to your branches and tags. If you want to keep all this information, there is a tool called `cvs2svn` for Linux/UNIX that will export a CVS repository either into a Subversion dump file or directly into a repository.

10.1.1 Getting cvs2svn

The Subversion source code and some distributions come with cvs2svn. While you can use this version, it is better to get a newer version. This tool is maintained separately, and the version that comes with Subversion is likely to be outdated. Since cvs2svn is getting constant improvements, you are much better off with the standalone version. This application is stored in a Subversion repository on the Internet, so you can just check it out. The URL for this is `http://svn.collab.net/repos/cvs2svn/trunk`. Let's check this out to a directory in the `svn` user's home directory:

```
svn co http://svn.collab.net/repos/cvs2svn/trunk cvs2svn
```

This will create a directory called `cvs2svn` that will contain all the scripts and files to convert your repository. Before we start looking at the conversion process, let's look at a few system requirements.

Prerequisites. The `cvs2svn` script is written in Python and calls RCS (Revision Control System) commands, so you will need these utilities on your system. The following is a list of all the prerequisites:

- Python 2.0 or greater
- A recent version of RCS
- A GNU version of the sort utility

All of these come standard on any version of Linux or can be obtained by going to your favorite RPM provider. Finally, make sure the directory you are running the script from has plenty of disk space available. A set of temporary files will be used when running the conversion, and they can be large depending on the size of your data set.

Installation. Since cvs2scvn is a Python script, there is no actual installation required. Once you have it checked out, the best way to run the command is right from that directory since all the required files are there.

10.1.2 Creating a Subversion dump from CVS

When the script runs, it will basically check out each revision of the CVS repository and load them one at a time into the Subversion repository. Depending on the size of your repository and the speed of your machine, this may take a while. Some people have attempted to use cvs2scvn as a replication/synchronization tool between CVS and Subversion, but this is not a good idea. The tool was designed for a one-time conversion and should be limited to this use.

To run the conversion, you will need to pass the script the path to the CVS repository; a Subversion dump will be created in the current directory. Let's try to convert the repository in `/repos/cvs/ldapd`. If you are running the script and generating a dump file, you will need to include the `--dump-only` option:

```
$ ./cvs2svn.py --dump-only /repos/cvs/ldapd
----- pass 1 -----
Examining all CVS ',v' files...
/repos/cvs/ldapd/editinfo,v
/repos/cvs/ldapd/avail,v
...
/repos/cvs/ldapd/ldapd-common-test/src/java/ldapd/common/message/spi/
  TestProvider.java,v
Done
----- pass 2 -----
Re-synchronizing CVS revision timestamps...
Done
----- pass 3 -----
Sorting CVS revisions...
Done
----- pass 4 -----
Finding last CVS revisions for all symbolic names...
```

```
Done
----- pass 5 -----
Mapping CVS revisions to Subversion commits...
Creating Subversion commit 2 (commit)
Creating Subversion commit 3 (commit)
Creating Subversion commit 4 (commit)
...
Starting Subversion commit 1305 / 1305
Done.
------------------
pass 1: 13 seconds
pass 2: 0 seconds
pass 3: 0 seconds
pass 4: 0 seconds
pass 5: 4 seconds
pass 6: 0 seconds
pass 7: 0 seconds
pass 8: 96 seconds
total: 116 seconds
```

This will parse your entire CVS repository, and all the files and directories will be listed as the output of the script. Next, you will see all of the versions being committed into Subversion revision numbers. Although the script is not yet loading the code into a repository, it must create revisions so it can generate a valid Subversion dump file. This file will be in the current directory and will be named `cvs2svn-dump`. It will be in the same format as a file created from the `svnadmin dump` command.

Now you can create a new Subversion repository and run the `svnadmin load` command using the dump file created with the `cvs2svn` script. Also, since this is in the correct format, you can run the `svndumpfilter` command we introduced in the last chapter to indicate whether to include or exclude any of the directories:

```
$ svnadmin create /repos/svn/ldapd

$ svnadmin load /repos/svn/ldapd < ./cvs2svn-dump
<<< Started new transaction, based on original revision 1
     * adding path : trunk ... done.
     * adding path : branches ... done.
     * adding path : tags ... done.

------- Committed revision 1 >>>
...
```

Take a look at the output from the `load` command, and you will see that the first thing it does is create the standard project root. Since all the branches and tags get imported with the conversion from CVS, Subversion assumes you want to put them in the `trunk`, `tags`, and `branches` directories at the top level. We will later

show how to change this behavior if you do not the follow the standard project root layout.

Changing the dump file. In most cases, you will want to specify the name and location of the dump file. This is done by simply adding the `--dumpfile` option to the script. Let's say we want to run the conversion and use the file `/tmp/ldap_cvs_dump`. We would change the execution of the script to look something like the following:

```
$ ./cvs2svn.py --dump-only --dumpfile=/tmp/ldap_cvs_dump \
/repos/cvs/ldapd
$ ls -ltr /tmp
total 93992
-rw-rw-r--  1 svn  svn  94472558 Jun 25 19:47 ldap_cvs_dump
```

The file specified in the command line was created as the dump file. You should note, though, that the temporary files will still be created in the directory from which you run the command, not where the dump file will reside.

Empty directories. If you are looking through the new repository and notice that certain directories are missing from the conversion, it is because those directories were empty in CVS. During the conversion, if the script comes across a directory that does not contain other subdirectories or files, it will be omitted. This is done to give you a clean start in the new repository, but you can override this behavior. Let's look at the CVS repository working copy to see the empty directory:

```
$ cd $HOME/projects/cvs/ldapd/ldapd-clients/

$ ls
build.xml  default.properties  lib        project.properties  src
CVS        empty_dir           maven.xml  project.xml

$ ls empty_dir/
CVS
```

The only thing in the `empty_dir` directory is CVS, which is the equivalent of `.svn`, so you can see that there are no other files here. Now let's look at the repository we just loaded and see if the directory exists:

```
$ cd $HOME/projects/svn/ldapd/trunk/ldapd-clients/

$ ls
build.xml           lib        project.properties  src
default.properties  maven.xml  project.xml
```

In the new Subversion repository, the `empty_dir` directory is gone. If you want to keep the empty directories, simply add the `--no-prune` option when you run the `cvs2svn.py` script. Let's run the conversion with this option:

```
$ ./cvs2svn.py --dump-only --dumpfile=/tmp/ldap_cvs_dump \
 --no-prune /repos/cvs/ldapd
```

Now you can create the Subversion repository and run the load the same way as before. When you check out the working copy and look at the `ldapd-client` directory, the directory will be there.

10.1.3 Changing the project root information

In many conversions, users want to bring only the main development path into the new repository. Not only can this decrease the size of the code base you are starting with, it can also simplify the process. If you want the script to ignore all the tags and branches, add the `--trunk-only` option to the script:

```
$ ./cvs2svn.py --dump-only --dumpfile=/tmp/ldap_cvs_dump \
 --trunk-only /repos/cvs/ldapd

$ svnadmin load /repos/svn/ldapd < ./cvs2svn-dump
<<< Started new transaction, based on original revision 1
     * adding path : trunk ... done.

------- Committed revision 1 >>>
...
```

Now when you do the load, only the trunk directory gets created. This will also decrease the number of revisions since the conversion script omits any version that changed only a branch or a tag. Even though you left out the branches and tags, the top level will still start with `trunk/`. You can change the name of the top level, or any of the project root directories for that matter, by adding some additional switches.

Changing the project root directories. If you are using a different design for the project root directories, you can do some customization in the conversion script to put the branches and tags in your format. Let's assume you are using the following layout for your project root:

Table 10.1 non-standard project root layout

Standard Design	Your Design
/trunk	/main
/branches	/activities
/tags	/releases

Looking at this design, you need to map the directories in your layout using the command-line switches on the `cvs2svn` script. These switches are called `--trunk`,

`--branches`, and `--tags`. You will set each equal to the name of the path you want it to be. Using the previous list, we would change the conversion script execution to look like the following:

```
 ./cvs2svn.py --dump-only --dumpfile=/tmp/dump --trunk=main \
--branches=activities --tags=releases /repos/cvs/ldapd

$ svnadmin load /repos/svn/test < /tmp/dump
<<< Started new transaction, based on original revision 1
     * adding path : main ... done.
     * adding path : activities ... done.
     * adding path : releases ... done.

------- Committed revision 1 >>>
...
```

Compare this output to the first load we ran in section 10.1.2, and you will see the new names for the three directories in the project root.

10.1.4 Converting directly into a Subversion repository

You may not want to create a dump file but instead load the code directly into a Subversion repository. The dump file works well if the repositories are on different machines and you need to move the dump from one server to another. Let's say you have a huge repository and want to migrate from CVS to Subversion over a weekend. You can go right from CVS to Subversion without having to wait for the dump file to be created before you run `svnadmin load`. This is all done in one step so no manual intervention is required, and your weekend will not be ruined by having to check to see if the dump is complete so that you can run the load. By default, the command will also create the repository, so you will need to make sure the Subversion repository does not exist. To load into the repository instead of creating the dump file, you must remove the `--dump-only` option and add `-s` with the repository path. For example, to load the CVS code into the Subversion repository you want in `/repos/svn/ldapd`, you would run the following command:

```
$ ./cvs2svn.py -s /repos/svn/ldapd /repos/cvs/ldapd
```

Before running this command, be sure there is no existing repository at `/repos/svn/ldapd`. This command will create that Subversion repository, get the contents from CVS, and then do the load. All the options in reference to renaming the project root directories are still valid in this method of doing the conversion.

Loading into an existing repository. If you try running the `cvs2svn.py` script with the `-s` option on an existing repository, it will fail. Let's say we ran `svnadmin create` before running the script in the previous example:

```
$ svnadmin create /repos/svn/ldapd

$ ./cvs2svn.py -s /repos/svn/ldapd /repos/cvs/ldapd
Error: the svn-repos-path '/repos/svn/ldapd' exists.
Remove it, or pass '--existing-svnrepos'.
```

If you look at the error message, Subversion will tell you how to get around this issue. When you add the `--existing-svnrepos` option, the script will append the CVS code to your Subversion repository:

```
$ ./cvs2svn.py --existing-svnrepos -s /repos/svn/ldapd \
 /repos/cvs/ldapd
```

The CVS repository conversion will start at the next available version of the Subversion repository. So, for example, if we had three revisions in `/repos/svn/ldapd`, `cvs2svn` would start at revision 4. Look at the following log output for the repository after the conversion has run:

```
$ svn log --revision 1:HEAD
------------------------------------------------------------------------
r1 | jeff | 2004-06-26 13:30:18 -0400 (Sat, 26 Jun 2004) | 1 line
add directory
------------------------------------------------------------------------
r2 | jeff | 2004-06-26 13:30:38 -0400 (Sat, 26 Jun 2004) | 1 line
add file1
------------------------------------------------------------------------
r3 | jeff | 2004-06-26 13:30:50 -0400 (Sat, 26 Jun 2004) | 1 line
made a change to file 1
------------------------------------------------------------------------
r4 | (no author) | 2002-10-16 17:11:58 -0400 (Wed, 16 Oct 2002) | 1 line
New repository initialized by cvs2svn.
------------------------------------------------------------------------
r5 | root | 2002-10-16 17:11:58 -0400 (Wed, 16 Oct 2002) | 2 lines
initial checkin
------------------------------------------------------------------------
```

Take a look at revision 4, and you will see the message the `cvs2svn` script uses to indicate that it is starting the conversion. If you look closely, you will notice that there is an anomaly created by running the conversion on an existing repository. As the revision numbers get larger, the dates should move forward in time, but this jumps back in time between revisions 3 and 4. If the dates on the CVS repository are older than those on the Subversion repository, you will see this jump.

Since the log message on the revision where the dump starts clearly states that it is an initialization from cvs2svn, users should be able to understand this peculiarity.

10.2 Eclipse plug-in

The Eclipse IDE is quickly becoming one of the more popular tools for software development, especially in the Java community. When you use this tool in conjunction with a version control system, there can be problems trying to manage the files with two different systems. There are plug-ins to Eclipse that allow you to run most of the Subversion control commands through the IDE. You can browse through the directory structure with the explorer in Eclipse instead of having to use a command-line interface. The most popular Subversion plug-in for Eclipse is called Subclipse.

10.2.1 Getting Subclipse

Subclipse is a separate project that is also maintained by Tigris. You will need to download the software from this web site at `http://subclipse.tigris.org`. On the main page, select Documents and Files, and you will see a list of distributions. The version you select will depend on which version of Subversion and Eclipse you are using. This software is moving very quickly, so new releases are coming out rapidly. You will need to match up the version of Subclipse with the version of Eclipse you are using. Select the version you want and save the ZIP file to your local machine. Once this is complete, you can extract it to your Eclipse directory. For example, if Eclipse is located in `c:\program files\eclipse`, that is where you will extract the ZIP file. Instead of manually downloading and extracting the plug-in, you call also use the update manager to obtain the plug-in. Now you can start up Eclipse and look at the SVN plug-in. Under the Window menu bar, select Perspectives and Other. A window will appear with a list of perspectives, as shown in figure 10.1. One of the choices will be SVN Repository Exploring.In the SVN repository perspective, you will be able to add new URLs and check out a repository into an Eclipse project.

Figure 10.1
Perspective selection dialog

10.2.2 Using Subclipse

Once you have the SVN perspective open, you can start running commands and set up a project. Figure 10.2 shows what this looks like.

Now you will need to add a repository. Right-click in the SVN explorer pane on the left, select New from the context menu, and then choose New Repository Location. Figure 10.3 shows the Add SVN Repository dialog.

You will need to add the URL of the repository; this is the same one you would use to check out a working copy from the command line. If you have any user authentication in the repository, you can add it here. Should any of the commands require credentials to complete, Subclipse will send them to the server.

In this example, a couple of repositories are already set up. You can browse through the directories just as you would in any explorer. If you right-click on a file or directory, you will see an option to show the history for that object. The change log information will show up on the tab labeled Resource History. You

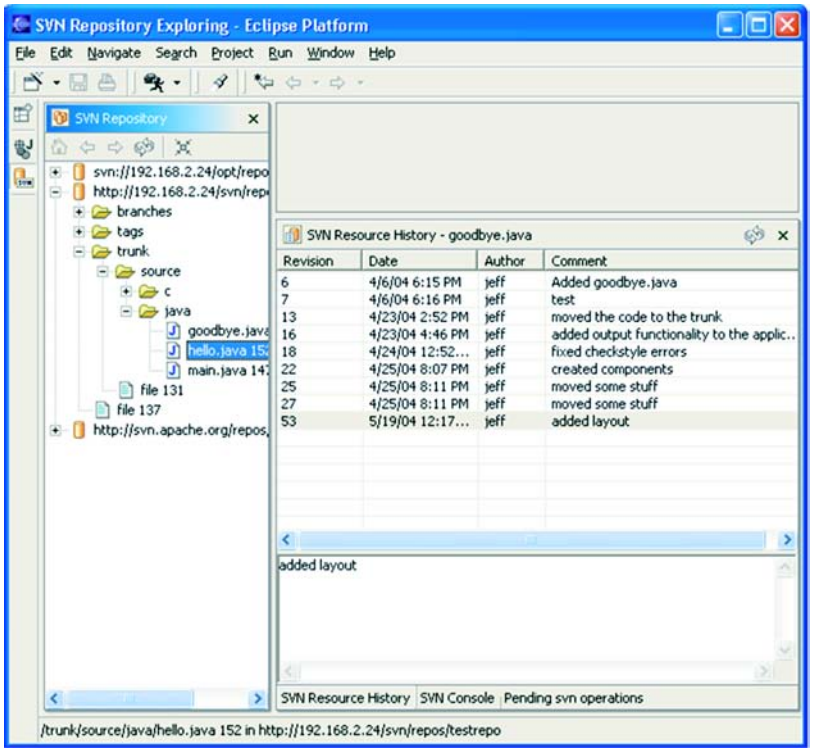

Figure 10.2 SVN perspective

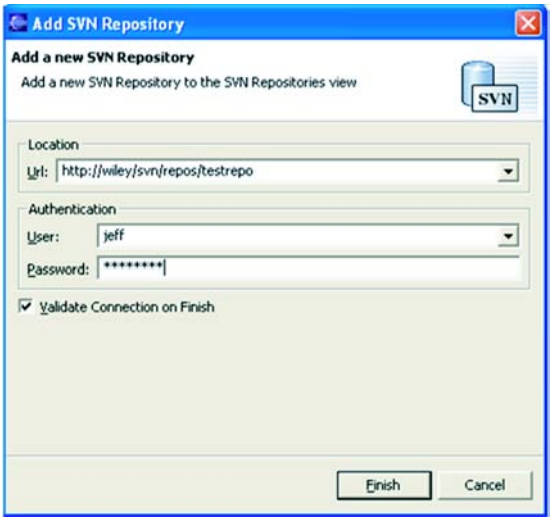

Figure 10.3
Add SVN Repository dialog

cannot make any changes to the files in this view because you have not yet checked out the repository.

Checking out the repository. When you are ready to start working with the code in the repository, you will need to check it out. To do this, right-click on the directory where you want to start the checkout, and a context menu will appear. From this menu, select Checkout as Project. Now when you switch to your main development perspective, you will see the project. Let's say we checked out the trunk of `testrepo`; figure 10.4 shows how this will look.

Notice on the icon for each file and directory that there is a cylinder; this indicates that it is part of the repository. Also, after the filename, you will see a date and user ID. This indicates the time of the last change and the user who made that change. If you look at the file `test.java` in the Package Explorer, you won't see a cylinder; in fact, it has a `?` instead. This means the file is not in the repository, so it is a non-versioned file located in the local filesystem.

Let's say you go on with your development, make some changes, and save the files in Eclipse. You are now ready to commit the modifications.

Committing from Eclipse. Once you are ready to save your changes, you can right-click on the file or directory where you want to start the commit. Remember, commits are recursive, so everything under the directory you start at will be committed in the same transaction. Think back to when we talked about committing in

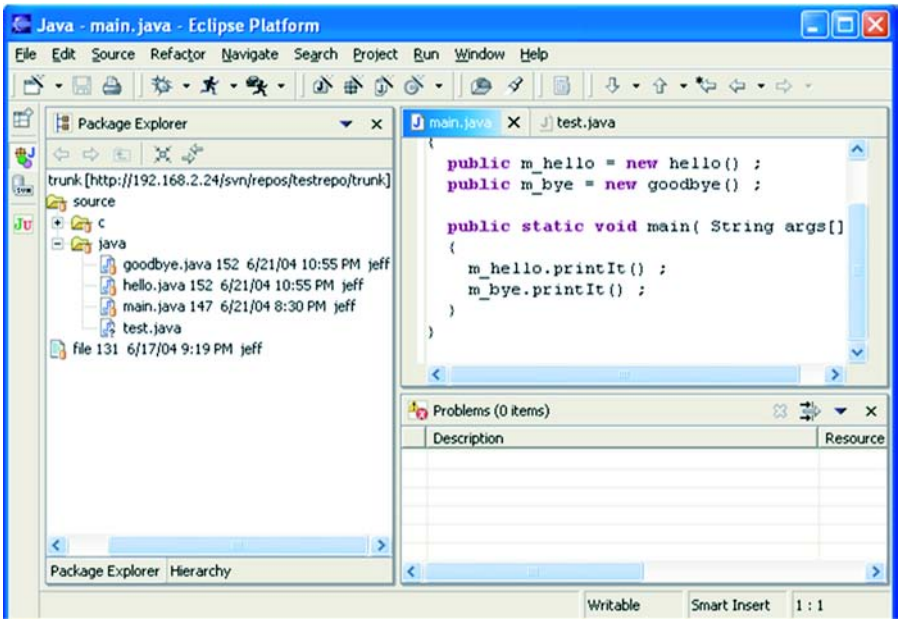

Figure 10.4 SVN layout in the Package Explorer

chapter 2; whatever current directory you started from will include everything beneath that point.

All the commands for Subversion will appear under the Team menu in the context menu. So to run a commit, right-click on the directory you want for the top level of the commit. Click the Commit button on the Team menu, and a Commit dialog will open. You can enter your log message in this window, as figure 10.5 illustrates.

Once you have the comment entered, press the OK button for the commit to start. This is how you can save your changes in the repository but still get the modifications in your working copy.

Updating the working copy. Updating the working copy is very simple in Eclipse. As we said before, each of the commands for Subversion is located under the Team menu. To run the update, right-click on the directory on which you want to start the update, and select the Team menu. Now just click the Update button, and the modifications from the repository will be populated in the working copy.

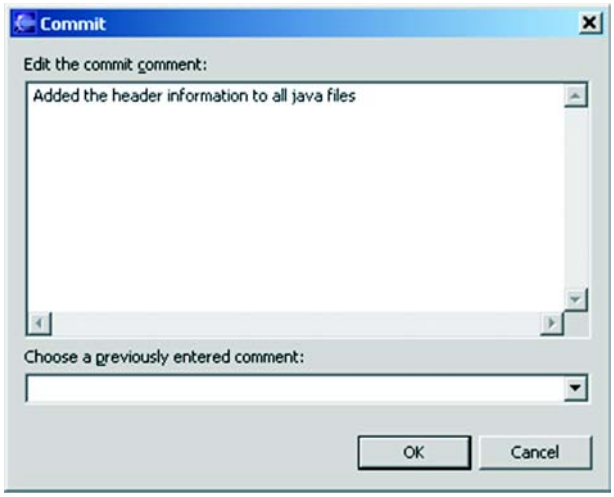

Figure 10.5
Log message editor dialog

10.2.3 Using the command line versus the plug-in

The Subclipse plug-in does not provide a complete set of Subversion commands, nor is it meant as a replacement for the command line. It does, however, give you quick access to the most common things you do while working with Subversion. By supporting commits, updates, and history browsing, it lets you tightly integrate your software development with most of your version control system needs. If you use Eclipse for development, your best option will likely some combination of the Subversion command-line interface and the Subclipse plug-in.

10.3 Subversion browsing over Apache with a browser

In chapter 7, we examined the basic ability to view the repository with a browser, but the features were limited. If you want more functionality in your browsing such as history and log message viewing, you can use a software product called WebSVN. This is an open source project also hosted on Tigris. You can find the distribution and documentation at `http://websvn.tigris.org`. WebSVN is a set of PHP files that you will point to browser, and it will read the repository for you. This tool is used only for reading and will not take the place of setting the Apache server to handle all of the client commands. WebSVN requires that you have PHP on your repository host, and you must be using Apache as the network protocol. Once you have the ZIP file downloaded, extract it to a location where your Apache server can read it.

Let's assume you are using the default `DocumentRoot` directive for Apache, so your HTML files are stored in `/var/www/html`. You can place the entire WebSVN directory there, so it will look like the following listing:

```
$ cd /var/www/html

$ ls
index.htmlsvnindex.css  svnindex.xsl  WebSVN
```

Once this directory is in place, be sure the user ID that runs the `httpd` daemon has read access to the directory and the files it contains. When you have done this, you are ready to configure WebSVN.

10.3.1 Configuration

The default distribution of WebSVN does not point to any repositories, so you will have to set this up first. If you look under WebSVN, you see a directory called `include`, which is where the configuration file resides. The software comes with a default template you can use, called `distconfig.inc`. Copy this to a new file named `config.inc`; this will be your configuration file. To get started, you need to change only one setting. Add the following line based on the location of your repository:

```
// --- REPOSITORY SETUP ---

$config->addRepository("Test Repository","/repos/testrepo");
```

Because this is PHP, the comments in the configuration will be indicated by `//`. In this line, we are telling WebSVN where our repository is, similar to when we set up Apache. Now, point your web browser to WebSVN, for example, `http://wiley/WebSVN`, and you will see something like the screen shown in figure 10.6.

You can see the repository we set up on the main page. Since you set up the location in the WebSVN configuration, you need only point your browser to the WebSVN directory, and it will take you to the filesystem.

Setting up multiple repositories. There are two ways to set up multiple repositories in WebSVN. The first method is to add each repository you want to be browsed using the `$config->addRepository` command. For example, if you wanted to add a second repository, your configuration file would look something like the following example:

```
$config->addRepository("Test Repository","/repos/testrepo");
$config->addRepository("LDAPd","/repos/ldapd");
```

The repositories do not need to reside in the same directory since you are explicitly pointing to each of them. The second method to set up more than one

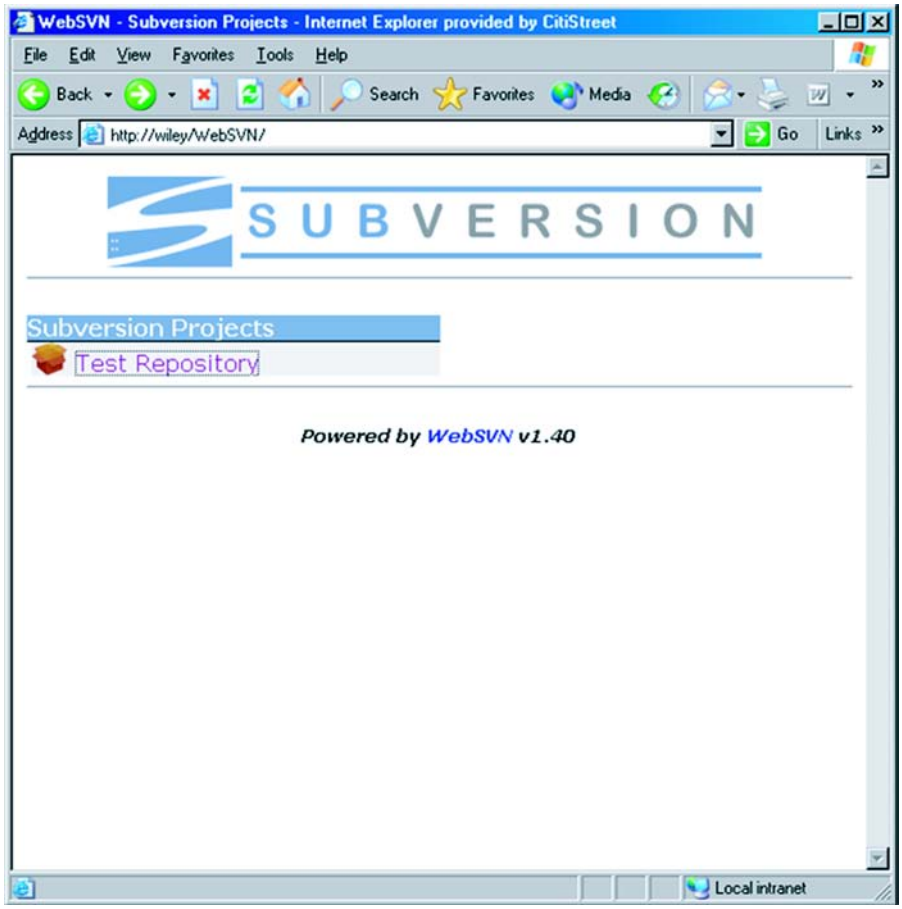

Figure 10.6 WebSVN repository list

repository uses the concept of the parent path we used in svnserve and Apache Setup. You can tell WebSVN that the repositories are in a path, and it will find them for you. This method works better if you add repositories frequently and do not want to be forced to change the configuration file each time a new one is added or removed. We will use a little different command in this configuration. Look at the following example to see this:

```
$config->parentPath("/repos");
```

The `$config->parentPath` command will accept the path in which all the repositories reside. When you bring up the browser now, these repositories will all be in

the list. Also in this method, the names will be the same as the base name of the directory, so you lose the ability you had to label the repositories when you added them individually. If you are on a UNIX-based machine, this configuration is ready to go. However, if the repository is on Windows, you will need to go through one more step.

Configuring Windows. If you are running WebSVN on a Windows machine, you will need to set this in the configuration. This one line needs to be uncommented, as shown in the following snippet:

```
$config->setServerIsWindows();
```

In terms of basic configuration, this is all you need to do. As this open source product develops, there will likely be more advanced settings available all the time.

10.3.2 Using WebSVN

The web interface to WebSVN is straightforward. Let's look at the summary of a directory listing. In figure 10.7 you can see the change information from latest repository revision and the contents of the directory.

If you select a file from the listing, the contents will be displayed in a new window. If you select the log for a file, each of the revisions will be displayed along with the change information for that revision.

WebSVN is a slightly more robust browsing utility than the default Apache one. It is important to remember that it is just for browsing, however. This is not meant to be a fully functional graphical interface. If you want something along this line, you will be interested in the next section, which describes a tool that can provide more functionality.

10.4 Windows graphical user interface

If you are using Subversion on Windows or UNIX, the command-line interface will work just fine, but you are not limited to it. For Windows users who want a graphical user interface, an open source tool called TortoiseSVN will fill this need. This is actually a plug-in to Windows Explorer that will give you an additional pull-down menu for Subversion commands.

10.4.1 Installing TortoiseSVN

TortoiseSVN is a separate project that you can get from the Tigris web site. Along with the download, there is some addition documentation on the project site at `http://tortoisesvn.tigris.org`. The first thing you will need to do is go to the

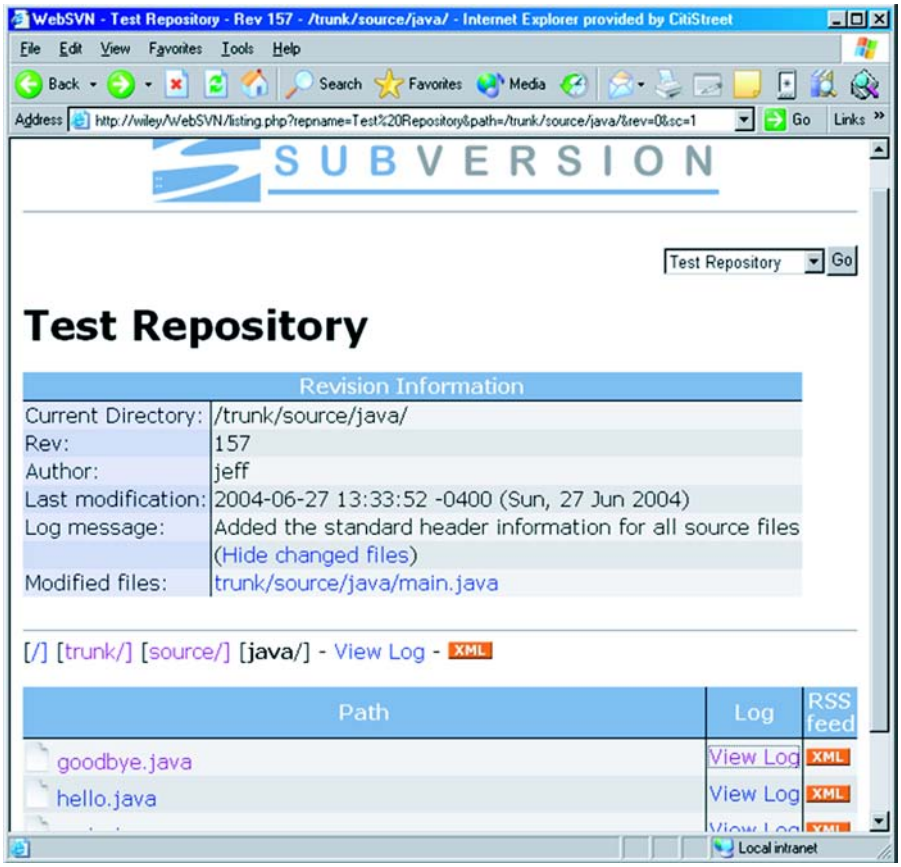

Figure 10.7 WebSVN directory browsing view

project site and click the Download link. This will take you to a page where you can select the version of Windows you are running. At this point, an `.msi` program will be downloaded, and you can simply open the file (there is no need to download it locally unless you want to use the file for multiple installs). You will need to answer a few installation questions, and then you will be finished. Now when you bring up Windows Explorer, you will see a couple of additional options in the context menu, as shown in figure 10.8.

We now have an option called Checkout and one called TortoiseSVN. Now let's see how to use this new tool.

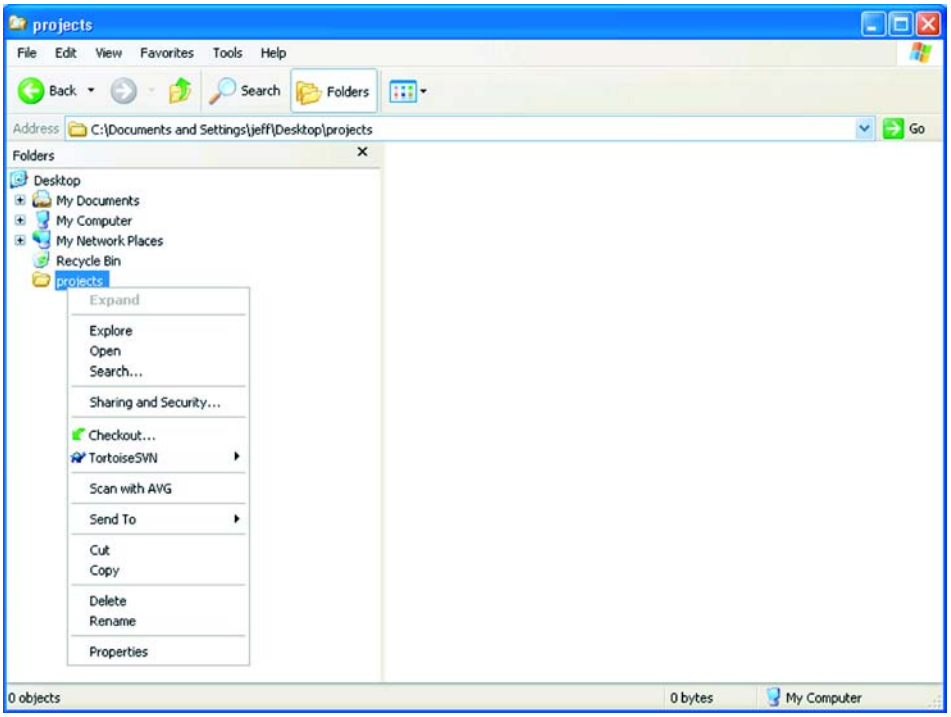

Figure 10.8 TortoiseSVN menu plug-in

10.4.2 Checking out a repository

Before you check out a working copy, you should probably create a directory in which to store it; in our example we created a directory called `projects` and created a shortcut to the desktop. We also created a directory with the name of our repository, in this case `testrepo`. Once you have the top-level working copy directory created, select it in Explorer and right-click. Then select the Checkout option from the menu. You will see a pop-up window like the one in figure 10.9, which will ask for the repository information.

You will need to enter URL of the repository, which is no different than what you used for the checkout from the command line. There is also a place to enter a specific revision number to check out if you do not want the `HEAD`. When you click the OK button, the working copy will be checked out, and the output will be displayed, showing what was checked out and the progress. Figure 10.10 shows this dialog box.

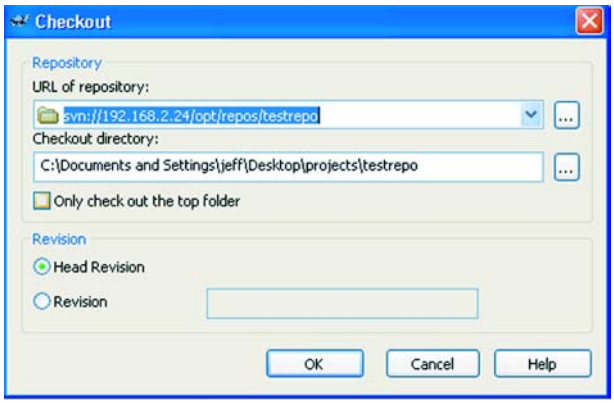

Figure 10.9
TortoiseSVN Checkout output dialog

Now when you look at the `testrepo` directory, you will see all the files in the repository. Along with the name of the file, you will see an icon that will indicate the status of that file or directory. To start, everything should have a green check mark, indicating that it is unchanged and part of the repository.

10.4.3 Checking the status

One of the advantages of using a GUI is the ability to add information to the display. You know the old adage: a picture is worth a thousand words. When you look at files inside an explorer that are part of a working copy, an icon will quickly tell you the status of each object. Let's take a look at figure 10.11 to see this.

Files with a green check mark have not been changed in the working copy. We ran an `add` command on `newfile.java`, which is represented by a plus sign. The file `goodbye.java` has been changed locally, so it has an explanation point for its icon.

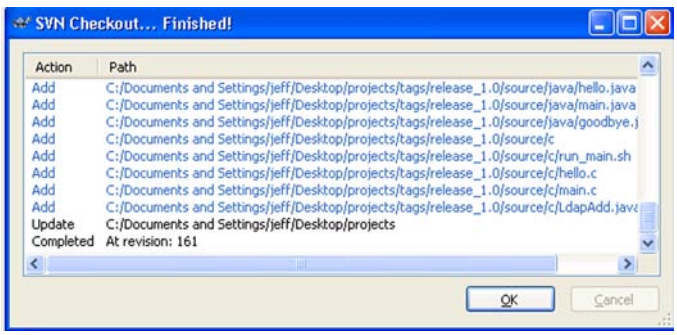

Figure 10.10
Checkout output dialog

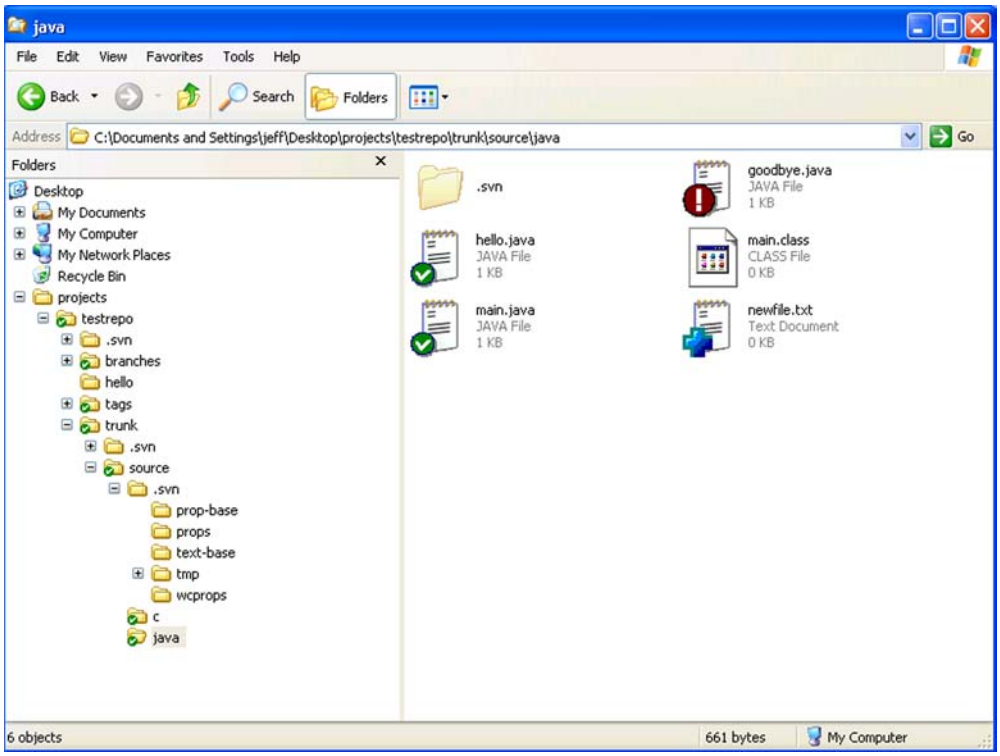

Figure 10.11 TortoiseSVN file list

10.4.4 Committing to the repository

When you want to save a change to the repository, simply select the directory from which to start the commit and right-click. From the Tortoise menu, select `Commit`, and you will see a pop-up window like the one in figure 10.12, where you can enter the log message.

A list of the files that will be saved is displayed in the dialog box. You can see the changes made to a file by double-clicking it. Once you are ready to save the changes, click the OK button, and the commit will be processed.

10.4.5 Running other commands

We will not go through every command and option that TortoiseSVN provides. If you understand the command line and all the options, the GUI should be pretty easy to use. You can see the list of possible commands under the TortoiseSVN menu in figure 10.13.

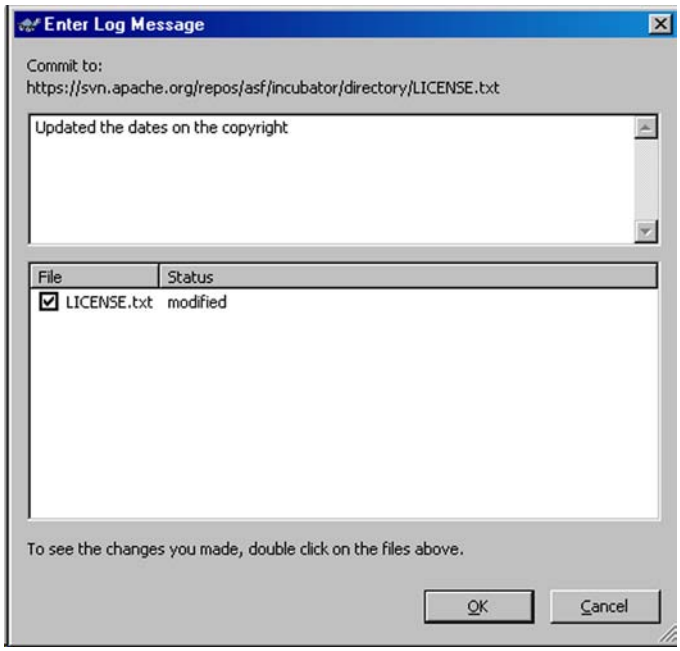

Figure 10.12
TortoiseSVN commit dialog

Your best bet is to run through the screens and get a feel for the different commands. You will probably like some better with the GUI and some better through the command-line interface, depending on your environment.

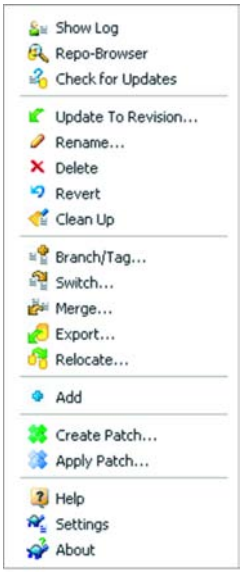

Figure 10.13
Command list in the TortoiseSVN menu

10.5 Summary

This is by no means a comprehensive list of the third-party tools available for Subversion. New open source and commercial tools appear all the time. Your challenge will be to find the ones that not only fit your development and configuration management needs but also are easy to use. It is also important to follow which tools are the standards in communities using Subversion, especially if you have an influx of developers, as in an open source group. By doing this, you will make their transition easier, which will allow them to spend more time writing code and less time learning the environment.

11 Subversion in a development lifecycle

In this chapter

- Client setup
- Developing with Subversion
- Code migration and testing

You now have a wealth of Subversion knowledge not only about how to use the commands, but also in which situations to use them. We will wrap up our discussion by looking at an example development process and show how Subversion can fit in the picture. While every process is different, the goal of this chapter is to show how Subversion can improve your development efforts.

11.1 Setting up your environment

With Subversion providing the client-side configuration, you can set up your local machine to remove certain tedious manual tasks. If you take the time to configure these settings from the start, your development process will not be interrupted with these adjustments. Let's look at some of the common situations where Subversion can streamline the development cycle.

11.1.1 Ignored files

You will almost definitely have certain file types in your working directory that are not part of the repository. In chapter 6, we showed how to set the `svn:ignore` property, which is good for exceptions, but you will not want to have to do this for all new directories you create. Let's assume your repository will have C and Java code. We know that these languages will create `.class` and `.o` files when they are built at a minimum. Since you will not want to store these files in the repository, you can set up the ignore list from the start in the `config` file:

```
[miscellany]
global-ignores = *.o *.class
```

By adding this code to your global settings, you will not need to worry about adding the `svn:ignore` property to subdirectories later.

11.1.2 Using autoprops

In addition to the global ignore list, in chapter 8 we explained how you can set certain properties on all files with the same extension. The properties we looked at included `svn:eol-type` and `svn:executable`. While this may seem like a minor task in your development process, it can actually have a large impact. One of the most common problems that users post on the Subversion IRC and mailing lists has to do with the end-of-line characters getting messed up in their editors.

When we looked at the `svn:executable` property, we saw developers having to reset permissions after an update so a script would run properly. If you are in the middle of a large coding effort, you do not want to waste time fixing permissions

on some script every time you do an update. These settings are also located in the `config` file in the Subversion client configuration directory:

```
[auto-props]
*.java = svn:eol-style=CRLF
*.c = svn:eol-style=native
*.sh = svn:executable
```

We explained how to do all this in chapter 8, so why bring it up again as part of the development process? The reason is that these little things that Subversion has implemented can really smooth out your development. The goal is to make the tool transparent so that it helps you with your work and does not become an impediment. You can accomplish this goal by tweaking these settings from the start to match your environment.

11.1.3 Getting plug-ins and other tools

Throughout the book we have mentioned third-party tools that can plug into Subversion or that Subversion can plug into. You will be better off if you can identify which tools you need up front and configure the environment before you start developing. We have already shown a few examples of these tools, such as TortoiseSVN for a GUI interface and WebSVN for web browsing. If you know which tools you will be using, you can gear the development process to take advantage of them. It is also a good idea to keep your eyes open for new tools entering the market. These include not only open source tools but also commercial products that are emerging to integrate with Subversion or embed Subversion commands. For example, if your development process requires frequent or complex merges, you can purchase a tool called Guiffy that will provide a graphical merge utility, as shown in figure 11.1.

If you are in a corporate development environment, don't be afraid to mix open source products with commercial ones. As Subversion becomes the standard open source version control system, both free and licensed vertical tools will become abundant. Many of these, such as Guiffy, provide free licenses to open source communities to help foster software development.

11.2 Writing code

The whole point of version control is to assist with writing and storing your source code. Let's walk through some typical situations you will run into in a development process that Subversion can handle.

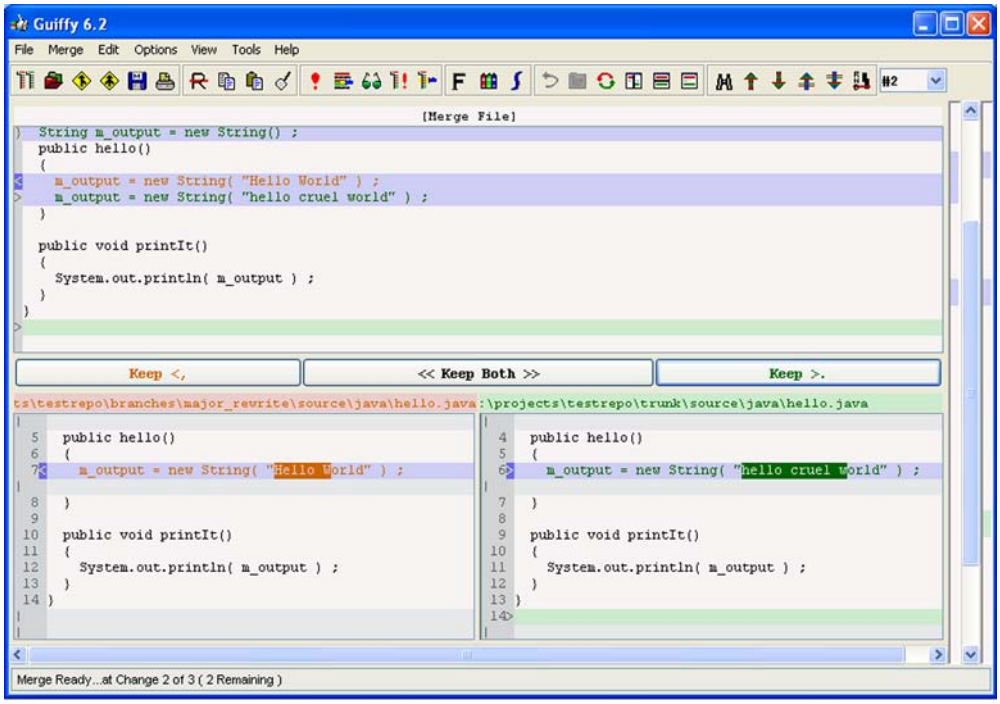

Figure 11.1 The Guiffy merge tool

11.2.1 Bug fixes

One of the major parts of any development process is fixing bugs. When you roll out Subversion, or any version control system for that matter, you need to decide where this work will be done. This may sound simple, but if don't you think it through all the way, you could run into situations where developers working on different tasks start to interfere with each other. For example, if you are working on an enhancement in the same file where Adam is fixing a bug, you both will be trying to hit a moving target. Take a look at the following code example; there is a typo in the `String` declaration that Adam has been assigned to fix:

```
public class hello
{
  String m_output = new String() ;
  public hello()
  {
    m_output = new String( "hell world" ) ;
  }
```

```
    public void printIt()
    {
      System.out.println( m_output ) ;
    }
}
```

In the meantime, you are working on a new constructor for `hello.java` so that the string to be printed will be passed in, not hard-coded. Your change will be in the next release, but the bug fix cannot wait that long and must be patched right away. To avoid any conflicts, Adam can create a branch to work on his bug fix. Once that fix is complete, Adam can create the patch from the branch to install in the release. If this change also needs to go into the new feature you are developing, you can

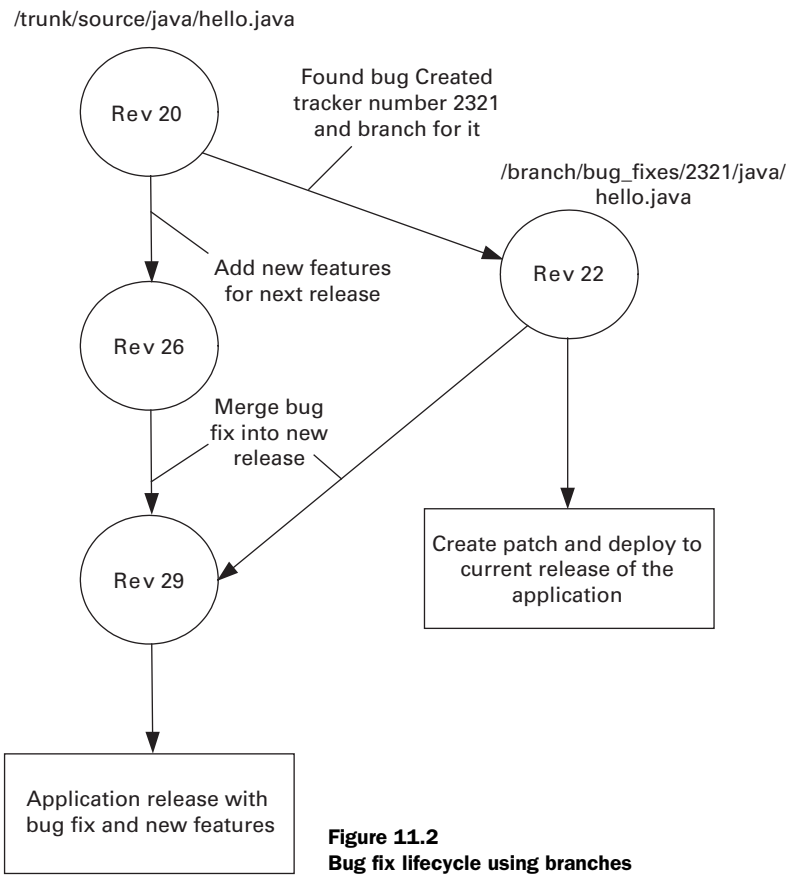

Figure 11.2
Bug fix lifecycle using branches

just merge it into the trunk (or wherever this development is taking place). When you are happy with the results, you can simply remove the branch and move on. Figure 11.2 shows how this looks in the revision tree, assuming you are doing new development in the trunk.

Using branches to develop your bug fixes allows you to easily track and monitor them. You can merge the changes to any development paths that require them, including the trunk or other branches that are for future releases.

11.2.2 Patches

Depending on the permissions of the repository you are working with, there may be a need to create patches. This allows developers without commit permissions to contribute code changes. This practice is common in many open source communities where developers need to earn their "karma" (commit privileges) by submitting patches. Let's look at the Eclipse IDE for creating and applying patches.

Creating a patch. If you do not have commit privileges, you can still check out code and make changes. Instead of using a commit to save to the repository, you will need to submit a patch to someone in the development community with the proper permissions. Let's say you have made some changes to the file `main.java` and tested them on your local system. In Eclipse, you can select the file, and under the Team menu you will see an option called Create Patch. Select this option, and you will get a dialog box asking where the patch should be created. Figure 11.3 shows an example of creating a patch file.

If you are going to submit the patch, it makes sense to save it directly to a file. Eclipse will save the path as a regular text file with a `.txt` extension. Now that patch is created, let's take a look at the contents of the file we created:

```
Index: C:/projects/testrepo/trunk/source/java/main.java
===================================================================
--- C:/projects/testrepo/trunk/source/java/main.java    (revision 167)
+++ C:/projects/testrepo/trunk/source/java/main.java    (working copy)
@@ -5,7 +5,7 @@
public static void main( String args[] )
   {
-    System.out.println( "Hello world" );
-    System.out.println( "bye" );
+    m_hello.printIt() ;
+    m_bye.printIt() ;
   }
 }
```

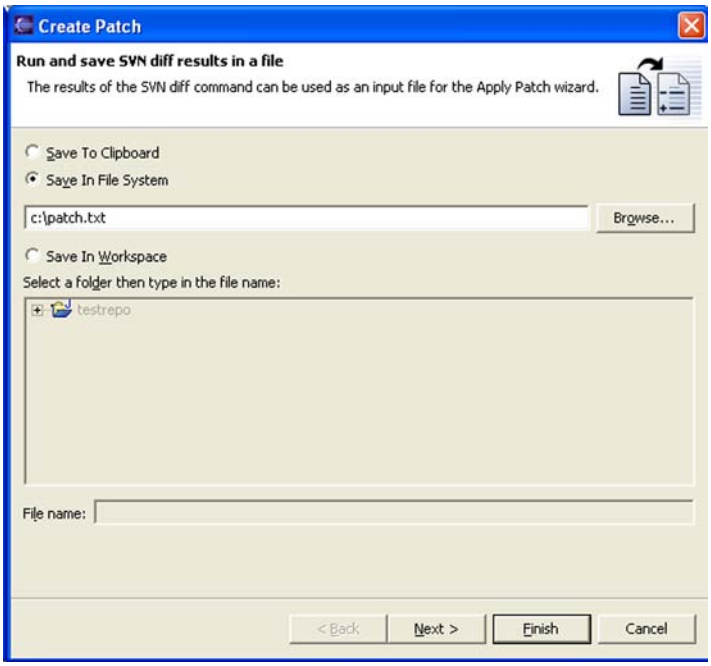

Figure 11.3 The Create Patch dialog

The patch is just the output from a `diff` command from your changes. The Apply Patch mechanism will use this information to apply the new or changed lines to the new file. You can send this file to the developer who will be applying the patch.

Applying the patch. When you get a patch file, you will first need to check the `Index` line to make sure the directory structure is the same. Many people use different paths for a working copy, so you must match up the filenames before trying to apply the patch. To find out the exact path Eclipse is using for the target file, right-click it and select Properties. You will see a dialog like the one shown in figure 11.4.

You are interested in the value of the Path label; in this case the absolute path to `main.java` is `/testrepo/source/java/main.java`. You will need to change the `Index` value in the patch file to match this path:

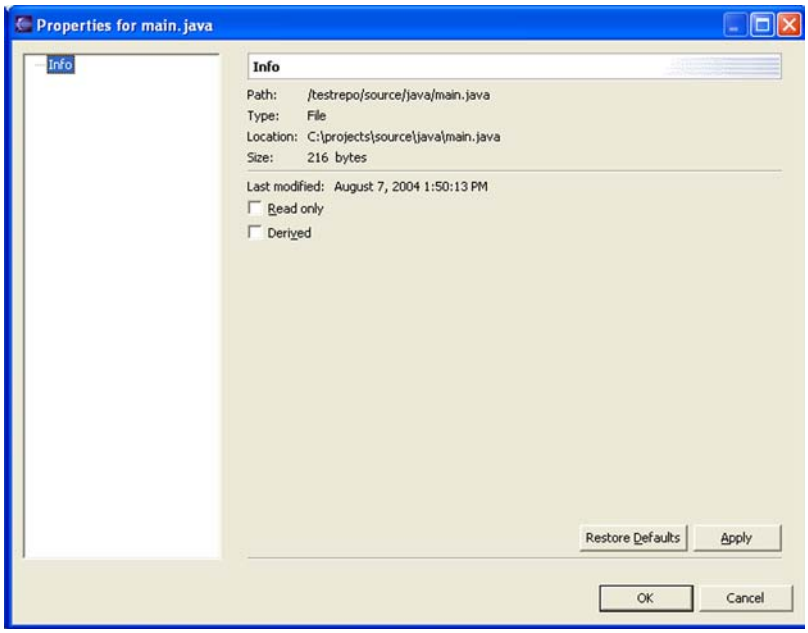

Figure 11.4 Finding the target path in Eclipse

```
Index: /testrepo/source/java/main.java
===================================================================
--- C:/projects/testrepo/trunk/source/java/main.java(revision 167)
+++ C:/projects/testrepo/trunk/source/java/main.java(working copy)
@@ -5,7 +5,7 @@

   public static void main( String args[] )
   {
-    System.out.println( "Hello world" );
-    System.out.println( "bye" );
+    m_hello.printIt() ;
+    m_bye.printIt() ;
   }
 }
```

You need to change only the `Index` value because the values in the legend are for display purposes and are not required to change. Once you have saved the change to the patch file, you can apply the patch. To start the process, right-click the target file and select Apply Patch from the Team menu. You will then be prompted for the name of the file.

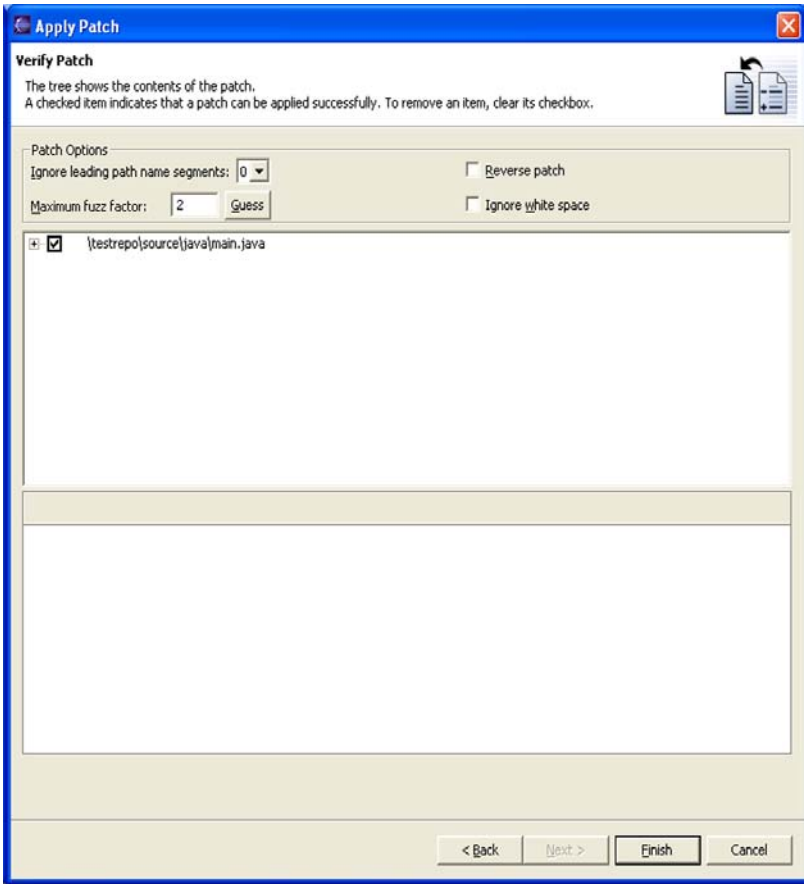

Figure 11.5 Applying a patch

Once you have selected the file, click OK, and you will see a check box next to the target file, as shown in figure 11.5. Make sure the box is checked, and click the Finish button. The changes from the patch will be applied to your working copy version. Now you just need to commit, and the changes will be saved to the repository.

11.3 Promoting code

Along with developing your code, another critical task Subversion can help you with is moving your source code. No matter what you call the different phases of your development lifecycle, it is important that you move the correct code to the

correct places. There are a number of different ways to move the source code around; let's take a look at a few of them.

11.3.1 Properties

You can use properties to tell where a specific revision of a file is in your development cycle if you don't need to run a lot of queries or have distinct segments based on the stage. The basic premise here is that each time a revision of the code gets promoted, the `STATE` property will get a new value. This value will be one of the predefined strings that match the stages in your development process. The following snippet from a promotion script demonstrates this concept:

```
#!/bin/bash
# promote.sh
#
# Arg 1 - revision of repository to promote
# Arg 2 - New State - Test, QA, Production
# Arg 3 - Target Files
svn propset --revprop --revision $1 --targets $3 STATE $2
...
```

Notice that we are using revision properties when setting the state. This is important because you do not want the `STATE` property to be moved with a new revision. Let's look at how this can be a problem by running through an example. If we set the `STATE` property of `main.java` to `production` at revision 168 of the repository, it will stay with future revisions:

```
$ svn propget STATE main.java
Production
```

Now if we need to make changes to the file, it will no longer be the version in production, but look at the `STATE` property:

```
$ svn commit --message "Changed function calls"
Sending        java/main.java
Transmitting file data .
Committed revision 168.
```

Now this code should go through the normal procedure of development, test, QA, and production. Let's examine the properties of the new revision of the file:

```
$ svn propget --revision 168 STATE main.java
Production
```

This revision starts with the production version because normal properties remain with the object on new revisions. If you use revision properties, they will stay with only the version to which you attached them. Let's try the same thing with revision 168:

```
$ svn propset --revision 168 --revprop STATE production main.java
property 'STATE' set on repository revision '168'
```

Now make a change to the file, and then check the properties to make sure the new revision does not have the `STATE` property set to `production`:

```
$ svn proplist --verbose --revision 168 --revprop main.java
Unversioned properties on revision 168:
  STATE : production
  svn:log : Changed to function calls
  svn:author : root
  svn:date : 2004-08-08T00:23:32.985822Z
$ svn proplist --verbose --revision 169 --revprop main.java
Unversioned properties on revision 169:
  svn:log : fixed some syntax
  svn:author : root
  svn:date : 2004-08-08T00:42:01.588360Z
```

Revision 168 still has the property set, but it did not carry over to the new revision you created. If you are using properties for the segmenting the code by development cycle, make sure you are using revision properties. There is one caution to this—remember that revision information is based on the repository, not individual files. If you attach the `STATE` property to a revision, it will show up for all files that were modified. Be sure all the objects in the change path are supposed to be in the state you are setting.

11.3.2 Tags

While properties can work great for identifying whether a version of a file is associated with a specific state in the development lifecycle, they do not work well for viewing the code in one state altogether. If you need to have all the code grouped together in one easy view, you are better off using a tag, as we explained in chapter 5.

An example of this would be a view for your QA team to see a release candidate all together for final testing. In the meantime, you would not want to slow down your future development. If you just use properties to determine the state of the code, there will be no way for the testers to check out all the files with a specific property set. By using tags, all this code can easily be grouped together. We will use the format of `release_number.state` to name the tag. So if you want to put release 1.3 of the code into a QA state, you would create the following tag:

```
$ svn copy trunk tags/release1.3.QA
A         tags/release1.3.QA

$ svn commit --message "added tag for release 1.3 in QA"
Adding         tags/release1.3.QA
Adding         tags/release1.3.QA/file
```

```
Adding         tags/release1.3.QA/source

Committed revision 170.
```

Now you can have your QA group check out the tag, and they will see all the code that is being tested for this release. You can even give permissions to change the files if necessary. When they make changes here, they will affect only the release and not be in the main development path. This will ensure a separate environment for your testing and future development. Any bugs that are found that need to go back into other development paths can just be merged in.

This method may seem like a lot of overhead, but remember that each tag or branch is not a complete copy. They start off as links, and only the deltas get saved to the repository. This makes tags very efficient in terms of disk space. They are also easy to create, so don't be afraid to use them throughout your development process.

11.3.3 Automating code promotion

Interestingly enough, one of the most common problems faced in moving code is getting the correct files moved. This is especially true if you have a large organization with a separate configuration management team. If files need to be individually pushed, inevitably something will be skipped or there will be a typo in creating the list. To get around this problem, you can create a script to migrate your code. The question is how to have the script recognize the files it should act on. If you create a list, the same risk of missing or wrong files exists. If you go back to using properties, you can avoid some of the risk.

You can create a specific property that tells the script to pick up the file. We will call the property `MIGRATE` and will give it a value of the revision number to be moved. For example, if you want to migrate revision 168 of `main.java`, you would set the `MIGRATE` property on that file:

```
$ svn propset  MIGRATE 168 main.java
property 'MIGRATE' set on 'main.java'
```

Now you should leave the property attached to the object until the migrate script picks it up. This will allow you to still make changes to the file but not change the status of the revision to be migrated. In the migrate script, you can use the following command to get the files that have the `MIGRATE` property still attached:

```
$ svn propget --recursive MIGRATE
trunk/source/java/main.java - 168
```

This command will list the files that have the property, plus give you the revision number of the file you need to pick up.

11.4 Testing and Subversion

When most people think of version control, they usually think of it as being used by developers and possibly configuration managers. Subversion can also be used to facilitate your testing process. This can be done by segmenting code to be tested in a branch or using properties to keep track of what has been tested and what the results are. Let's take a look at these key points of using Subversion to test.

11.4.1 Maintaining the testing environment

In any testing environment, you need to maintain a separate set of files such as test cases, test scripts, and results. Since these are just files, they can be stored in Subversion, which gives them the same traceability and recovery options as your source code. As your code develops, you will need to change the test scripts accordingly. If you need to restore the code to a previous revision, the test cases will need to be restored also, so being able to get back to previous revisions can be critical.

If you have multiple projects in different repositories, you can get creative in setting up test scripts by using the `switch` command to switch to your `testcases` repository. For example, if you have a directory in the `testcases` repository, you can point it to the development repository you are looking at so it will appear to part of the testing working copy. This will allow you to maintain a level of consistency in your test scripts and result reporting. Let's take a look at this process. First we will check out the `testcases` repository into a working copy:

```
$ svn co file:///opt/repos/testcases/
A  testcases/testscripts
A  testcases/code
A  testcases/testcases
Checked out revision 1.
$ cd testcases/code
$ ls

$ svn info .
Path: .
URL: file:///opt/repos/testcases/code
Repository UUID: d3ed68af-24e1-0310-b136-b8ceca19af8e
Revision: 1
Node Kind: directory
Schedule: normal
Last Changed Author: jeff
Last Changed Rev: 1
Last Changed Date: 2004-08-08 09:47:05 -0400 (Sun, 08 Aug 2004)
```

This previous set of commands looks very familiar; we checked out the repository and looked at the `code` directory, which is empty. If all of the test scripts point to

this directory, we can link in a development repository so that the code appears to be part of the `testcases` repository. Let's use a repository that contains the organization's financial application as an example:

```
$ svn switch --relocate file:///opt/repos/testcases \
file:///opt/repos/financial_repo/
```

Now the code directory will contain the objects from the `financial_repo` repository so that you can run the test scripts locally in the `testcases` working copy. When you have finished with the tests, you can switch back to the default URL or move on to the next repository.

11.4.2 Locking down properties

As previously mentioned, you can migrate or keep track of the state of code with properties. If you are in a large organization or have very distinct roles between developers and testers, you may want to lock down these properties. For example, if you go through a change control and code review process before any new code goes to production, you may not want all the developers to be able to change the properties. If you are using the `MIGRATE` property, for instance, only people in the QA group should be able to set this property once the change is approved. In chapter 8 we looked at some examples of access control scripts. What is important when developing the process is determining who will have access to change the properties for migration and code promotion. Once you have that figured out, you will be able to plug that restriction into the access hook script.

11.5 Summary

Throughout the book, you have seen not only how to use Subversion from a mechanical standpoint but also how it fits in your overall software development. This final chapter is by no means a complete guide to the software development process, but it should start you thinking down the path of process. The real power of Subversion does not come from the efficient storage of files, directory versioning, or any other "marketing" point on the project's web site. It comes in the form of integration with any development lifecycle model. Subversion lends itself to automation and easy integration. If you can start using the tool for this purpose and not just as a place to save files, you are on your way to better software development.

A SSL certificates

When we set up the Apache server, you saw how to configure it to use SSL for securing the transmission. We also explained how to configure Subversion to require a client certificate and set up your runtime area to automatically pass this in. Now let's see how to create a certificate and sign other certificates for authentication. The following example of SSL certificate creation will be based on `openssl`, which comes with most Linux distributions or can be obtained at `http://www.openssl.org/`.

A.1 Creating a server certificate in Apache

First, let's create a certificate that we can place on the repository host for secure transmission. We can use this in the Apache web setup, and we can also send a copy of it to the clients so they will automatically trust the server. The first thing we need to do is create a key:

```
$ openssl genrsa -des3 -out svn_server.key 1024
Generating RSA private key, 1024 bit long modulus
.......................++++++
.........++++++
e is 65537 (0x10001)
Enter pass phrase for svn_server.key:
Verifying - Enter pass phrase for svn_server.key:
```

This will create the key and prompt you for a pass phrase. It is critical to remember this phrase since you will need it each time you use the key.

Now we must create the certificate with the key. You will need the pass phrase used to create the key, plus some additional information, as you will see shortly:

```
$ openssl req -new -x509 -days 365 -key svn_server.key \
-out svn_server.crt
Enter pass phrase for svn_server.key:
You are about to be asked to enter information that will be incorporated into
   your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [GB]:US
State or Province Name (full name) [Berkshire]:Florida
Locality Name (eg, city) [Newbury]:Jacksonville
Organization Name (eg, company) [My Company Ltd]:none
Organizational Unit Name (eg, section) []:none
Common Name (eg, your name or your server's hostname) []:wiley.mydomain.com
Email Address []:svn@wiley.mydomain.com
```

We will use this as the host certificate for the Apache web server. Next, we need to configure the server to see this certificate.

A.1.1 Configuring Apache with the certificate

Now that we have the certificate created, we need to set this up in the Apache configuration. If you look at the `conf.d` directory where we set up the Subversion configuration, you will see a file called `ssl.conf`. If you are using the default installation for Apache, the file will be in the `/etc/httpd/conf.d` directory. Let's examine the file to see what the configuration should look like:

```
<VirtualHost _default_:443>
...
SSLCertificateFile /etc/httpd/conf/ssl/svn_server.crt
SSLCertificateKeyFile /etc/httpd/conf/ssl/svn_server.key
...
</VirtualHost>
```

You will need to add the two lines in the `VirtualHost` section of the configuration file. The values should point to the certificate and key we just created. Once you have these values in place, you must restart the web server. Notice that you will have to manually enter the pass phrase when the Apache server starts:

```
# /etc/init.d/httpd restart
Stopping httpd:                                            [  OK  ]
Starting httpd: Apache/2.0.49 mod_ssl/2.0.49 (Pass Phrase Dialog)
Some of your private key files are encrypted for security reasons.
In order to read them you have to provide the pass phrases.

Server wiley:443 (RSA)
Enter pass phrase:

OK: Pass Phrase Dialog successful.
                                [  OK  ]
```

Once the web server is back up, you can connect to the repository host using the `https` protocol. Don't forget to turn on the `SSLRequireSSL` directive, as discussed in chapter 7, so that non-encrypted transmissions will not be allowed. Now when you try to check out the repository, you will see the certificate you created for the client machine:

```
$ svn co https://wiley/svn/repos/testrepo
Error validating server certificate for 'https://wiley:443':
```

```
  - The certificate is not issued by a trusted authority. Use the
    fingerprint to validate the certificate manually!
Certificate information:
 - Hostname: wiley
 - Valid: from Jul  3 14:53:05 2004 GMT until Jul  3 14:53:05 2005 GMT
 - Issuer: none, none, Jacksonville, FL, US
 - Fingerprint: 10:2c:7b:9e:47:7b:41:97:07:a0:79:f6:34:01:71:62:b1:38:28:e7
(R)eject, accept (t)emporarily or accept (p)ermanently?
```

Once you accept this certificate, the command will continue. You can also change the setup in your client configuration to automatically trust this certificate, as discussed in chapter 8.

Setting up the trust on the client. If you have a small group of trusted client machines, you can copy the certificate to each of them so they will not need any manual intervention. The first thing you need to do is get the certificate to the clients. Let's create a directory in which to store them, called `/etc/svn_ssl`, on the client machine `mort` :

```
$ mkdir /etc/svn_ssl

$ scp svn@wiley:/etc/httpd/ssl/svn_server.crt /etc/svn_ssl
```

Once you have the certificate in the proper location, you will need to point the client configuration there. The `servers` file in the client configuration would need the following line:

```
[global]
ssl-authority-files = /etc/svn_ssl/svn_server.crt
```

Now you can hit the repository over `https` and the certificate will be accepted, so no manual intervention will be required.

A.2 Creating a client certificate

You may need more security than encrypted transmissions and user IDs. In the previous section, we explained how to use encryption, but anyone can get the certificate and read from the repository. You can set up the server so that it will not allow anonymous access, but you can also add another layer of security if desired. You can force each client to provide a certificate signed by a common certificate authority (CA) in order to communicate with the server. This will allow you to have user and client machine–level authentication. The first step is creating a certificate request on the client machine.

A.2.1 Creating a certificate request

Since we are going to require a signed certificate, we need to create a certificate request. This is a little different than the certificate we created in the previous section. We will not get a complete certificate until it is signed by a CA, which we will discuss shortly. To start, we still need to create a key, and this should be done on the client machine that will hold the certificate:

```
$ openssl genrsa -des3 -out svn_server.key 1024
Generating RSA private key, 1024 bit long modulus
.....................++++++
.................................++++++
e is 65537 (0x10001)
Enter pass phrase for svn_server.key:
Verifying - Enter pass phrase for svn_server.key:
```

You will need to enter a pass phrase for the key, just as you we did in creating the certificate. Now you will create a certificate signing request. You will need to fill in the location and host information as prompted:

```
$ openssl req -new -key svn_server.key -out svn_server.csr
Enter pass phrase for svn_server.key:

Country Name (2 letter code) [GB]:US
State or Province Name (full name) [Berkshire]:Florida
Locality Name (eg, city) [Newbury]:Jacksonville
Organization Name (eg, company) [My Company Ltd]:none
Organizational Unit Name (eg, section) []:none
Common Name (eg, your name or your server's hostname) []:mort.mydomain.com
Email Address []:svn@mort.mydomain.com

Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []:
An optional company name []:
```

At the end of the processing, you can enter a challenge password. This will require the user to enter a password each time the certificate is accessed. For now we will leave this field blank. Notice also that the extension on the file is different. Instead of `.crt` for certificate, we use `.csr` for a request. The extension name is not mandatory, but it will help you keep track of what state the file is in. Once you have the request, you can send it off to be signed. There are commercial organizations such as VeriSign that will do this for you. As an alternative, you can self-sign your certificates, which does not work well for public use but is fine if you have a private set of clients who will be connecting.

Creating a CA. A CA is an entity that will verify that a certificate request is valid and add a signature to it. This signature will tell everyone using the certificate that it is trusted. We will assume that you have a network of client machines to which you want secure transmissions. You would want to do this in a situation where you had a repository server and a bunch of development and test servers that people check out the code to and work from. By forcing a client certificate, you can prevent other systems from accessing the repository. We will use the same steps to create the CA as we did when we made the server certificate:

```
$ openssl genrsa -des3 -out ca.key 1024

$ openssl req -new -x509 -days 365 -key ca.key -out ca.crt
```

Now that you have a request and a CA, you can create a signed certificate. The best way to do this is to use a script called `sign.sh`, which comes with the `mod_ssl` package. If you do not have this, you can download it from `http://www.modssl.org/`. Once you get the distribution, you'll find `sign.sh` in the `pkg.contrib` directory. You will need to edit the script and adjust the settings that point to the location of your certificates; we have been using `/etc/httpd/conf/ssl` for this location. Once you have changed the script to match the location of your CA files, you will need to copy all the client certificate requests to the host with the CA. Let's say we copied the file `mort_svn.csr` to our repository server. We would run the following command to generate the signed certificate:

```
$ ./sign.sh mort_svn.csr
CA signing: mort_svn.csr -> mort_svn.crt:
Using configuration from ca.config
Enter pass phrase for /etc/httpd/conf/ssl/ca.key:
Check that the request matches the signature
Signature ok
The Subject's Distinguished Name is as follows
countryName           :PRINTABLE:'US'
stateOrProvinceName   :PRINTABLE:'FL'
localityName          :PRINTABLE:'JAx'
organizationName      :PRINTABLE:'mort client cert for SVN'
organizationalUnitName:PRINTABLE:'non'
commonName            :PRINTABLE:'mort'
emailAddress          :IA5STRING:'nobody@localhost'
Certificate is to be certified until Jul  3 19:49:12 2005 GMT (365 days) Sign
   the certificate? [y/n]:y


1 out of 1 certificate requests certified, commit? [y/n]y
Write out database with 1 new entries
Data Base Updated
```

```
CA verifying: mort_svn.crt <-> CA cert
mort_svn.crt: OK
```

This will create a new file called `mort_svn.crt`, which is your signed certificate. Subversion requires you to take one more step and convert the file into a different format. You must use files in the PKCS#12 format for client certificate authentication. To do this, run the following command on the certificate you just signed:

```
$ openssl pkcs12 -export -certfile ca.crt -inkey mort_svn.key \
-in mort_svn.crt -out mort_svn.p12

Enter pass phrase for mort_svn.key:
Enter Export Password:
Verifying - Enter Export Password:

$ scp mort_svn.p12 jeff@mort:/etc/svn_ssl/
```

This command will create the file in the correct format. Now you need to copy the file `mort_svn.p12` back to the client machine. The other files can and should be removed.

Next, you need to configure the Apache server to ask for the client certificates:

```
SSLVerifyClient require
SSLVerifyDepth  1
SSLCACertificateFile /etc/httpd/conf/ssl/ca.crt
```

The first two directives will tell the server to ask for a certificate from the client. The third directive will point to the CA that was used to sign the client certificate. Restart the web server, and then any `.p12` file that was signed by this CA will be able to authenticate. Let's run a checkout from the client machine `mort` and see how this works:

```
$ svn co https://wiley/svn/repos/testrepo
Authentication realm: https://wiley:443
Client certificate filename: /etc/svn_ssl/mort_svn.p12
A  testrepo/trunk
...
```

You will be prompted to provide a certificate when you connect to the repository server. At this point you must find the filename of the certificate that was signed by the CA we set up in the Apache configuration. Remember, you can set up your client configuration area to automatically provide the certificate each time you run a command.

A.2.2 Setting up a client certificate to be sent automatically

If you do not want to manually enter the client certificate each time you run a Subversion command that connects to the repository, you can set up your client configuration area to get around this. All you need to do is provide the name and path to the certificate that you want to send in the `servers` file of the client configuration. Let's take a look at an example using `mort_svn.p12`:

```
[global]
ssl-client-cert-file = /etc/svn_ssl/mort_svn.p12
```

Now when you run commands that hit the repository, the certificate will be automatically sent and no intervention will be required.

B Building Subversion

If there is not a binary distribution for your operating system, or if you need the absolute latest code before creating a package, you will need to build the source. It is impossible to walk through all the variations of compiling Subversion here, so we will explore just the basics of building on a UNIX-based system. Most of the library dependencies are included with the source. The ones not included are pretty much standard for any operating system. You will need to have a C compiler and a make utility; the GNU versions of these are just fine.

B.1 Building Subversion from the source

Go to the Subversion web site and select the download page; you will see a link to get the source distribution. There will be a couple different packaging methods such as `gzip` and `bzip`; select the best one for you and download it. Once you have the package saved to your local system, unzip it and you will get a tar file. Now extract the tar file; this will create a directory with the same name as the tar file. There will be a file called `configure` in the top level; you will need to run this script to generate the `Makefile`. This is no different than any standard open source project: the configuration script will search through your system and decide how to compile. This script will also be used to determine where the Subversion binaries and libraries will be installed. If you need to make changes to the default, it is important to do this in the script, not after you build and install the software. Moving objects around after the install could result in the executables failing to run. The default install path for the binaries and the dependent libraries is `/usr/local`. To change this path, run the configuration script with the `--prefix` switch. For example, if you want your copy of Subversion to be in your `home` directory, run the following command:

```
$ mkdir /home/jeff/svn

$ ./configure --prefix=/home/jeff/svn
```

If you are building the source on the repository server and plan on using the Apache web server, you will need to include the source directory of Apache. If you have installed Apache in the default location, `/usr/local/apache2`, the configuration will find this for you automatically. Use the following switch to include this in the `Makefile`:

```
$ ./configure --with-apache=/home/jeff/downloads/apache-2.0.50

$ make

$ make install
```

The script will now run through a long process of checking requirements and system settings. If you are a missing a required piece, the script will stop and let you know. You may see the script report that certain items are not found, and then it will just move on. Don't worry about these; as long as the configure script does not stop, you should be all set. We have shown only one switch for this script, but there are many more. We will not go through all the options, but you can view them by running `configure` with the `--help` option.

Once the configure script is finished, you will have a set of make files that will be used to compile the code. From the top-level directory, just run `make` with no options. This will create the binaries in the working directories for each component. There is quite a bit to build, so if you are compiling on a slow machine, go get a cup of coffee and come back in a while. Once the build is done, you will need to install the binaries and the libraries. This can be done using the command `make install`. Or, if you would like to do this in one step, you can run the `make` command with the `install` argument first, and it will build and install in one step. When this command finishes execution, Subversion will be installed in `/usr/local/bin`, or wherever you specified if you used the `--prefix` option.

B.1.1 Environment variables

Before you can start to use Subversion, there are a couple of environment variables that need to be set. The first one is your `PATH`, which will tell your system where the binaries are so that you do not have to specify the absolute path each time you run a command. The value will be set to the directory used as the prefix when you installed, with the subdirectory `bin` added at the end. So, for example, if you used the default location, you would add the following lines to your profile:

```
PATH=$PATH:/usr/local/bin
export PATH
```

In addition to the `PATH`, you will also need to make sure you have an `EDITOR` environment variable set. Any command in Subversion that writes to the repository, such as `commit`, will require the user to enter a log message. This is entered through an editor, which is automatically started by Subversion. In order to figure out which editor to start, the command will use the `EDITOR` environment variable. If this is not set, the Subversion command will not execute. You can use any editor; the following example uses vi:

```
EDITOR=vi
export EDITOR
```

Subversion will allow you to use a different variable if you wish. Since `EDITOR` is a standard for other tools in UNIX, you may want to have a separate editor for entering log messages in Subversion than you do for command-line shells. To resolve this, Subversion will also accept the variable `SVN_EDITOR`. The commands will use this environment variable first; if it does not exist, then they will try `EDITOR`.

B.1.2 Apache modules

The `make install` command should place the Apache modules for Subversion in the correct location, but you may need to manually copy them in. In your Apache install, there will be a directory called `modules`. Make sure the following files are in this directory after the `make`:

```
mod_dav_svn.so
mod_authz_svn.so
```

If these files are not in the `modules` directory, you can find them in the source distribution. Go to the directory you installed and where you built the source code. The files will be in the following locations:

```
./subversion/mod_dav_svn/.libs/mod_dav_svn.so
./subversion/mod_authz_svn/.libs/mod_authz_svn.so
```

All you need to do is copy the files to the `modules` directory of your Apache install and restart the server. You can now enter the `LoadModule` directives we discussed in chapter 7.

B.1.3 Verifying all the access modules

Once you have finished and have installed Subversion, you should verify that all the access modules have been properly installed. Doing so will also verify that you have the correct version and that the install went okay. Let's assume we are trying to upgrade from version 1.0.1 to 1.1.0. Run the `svn --version` command to check the install:

```
[root@wiley subversion-1.1.0-rc1]# svn --version
svn, version 1.1.0 (Release Candidate 1)
   compiled Aug  8 2004, 17:58:08

Copyright (C) 2000-2004 CollabNet.
Subversion is open source software, see http://subversion.tigris.org/
This product includes software developed by CollabNet (http://www.Collab.Net/
   ).

The following repository access (RA) modules are available:
```

```
* ra_dav : Module for accessing a repository via WebDAV (DeltaV) protocol.
  - handles 'http' schema
* ra_local : Module for accessing a repository on local disk.
  - handles 'file' schema
* ra_svn : Module for accessing a repository using the svn network protocol.
  - handles 'svn' schema
```

The output shows that we got the correct version of the software, but we are missing a module. There is no access to repositories over `https`. When this happens, you need to go back to the configure and rerun the command with the following option added:

```
$./configure --with-ssl
```

In addition to the `--with-ssl` switch, you should include any other options you used in the original script. Once this is done, you can run the `make install` command again, and the updated binaries with SSL compatibility will be installed.

B.1.4 Utilities

When you have finished the Subversion build, you will have a directory called `tools`. Let's examine its contents:

```
[root@wiley tools]# ls
backup  client-side  cvs2svn  dev  examples  hook-scripts  test-scripts
   xslt
```

Each release of Subversion contains more tools and utilities than the previous one. We have the cvs2svn program distributed here, but you are still better off getting the latest version. In the `backup` directory you will see a script that will give a backup solution right out of the box. Another key set of files will be in the `hook-scripts` directory. Some of the common scripts, such as email notification, will already be written for you. It is a good idea to look here before deciding to write something from scratch. It also will not hurt to download the latest Subversion distribution from time to time and see if there are any updates to the utilities. To create these tools, you need only build the source and not necessarily install it; thus you can get the latest tools without having to upgrade the core Subversion applications.

index

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

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