TortoiseSVN

A Subversion client for Windows

Version 1.5.5

Stefan Küng
Lübbe Onken
Simon Large

TortoiseSVN: A Subversion client for Windows: Version 1.5.5

by Stefan Küng, Lübbe Onken, and Simon Large

Published

Table of Contents

List of Figures

List of Tables

Preface

- Do you work in a team?

- Has it ever happened that you were working on a file, and someone else was working on the same file at the same time? Did you lose your changes to that file because of that?

- Have you ever saved a file, and then wanted to revert the changes you made? Have you ever wished you could see what a file looked like some time ago?

- Have you ever found a bug in your project and wanted to know when that bug got into your files?

If you answered "yes" to one of these questions, then TortoiseSVN is for you! Just read on to find out how TortoiseSVN can help you in your work. It's not that difficult.

1. Audience

This book is written for computer literate folk who want to use Subversion to manage their data, but are uncomfortable using the command line client to do so. Since TortoiseSVN is a windows shell extension it's assumed that the user is familiar with the windows explorer and knows how to use it.

2. Reading Guide

This Preface explains a little about the TortoiseSVN project, the community of people who work on it, and the licensing conditions for using it and distributing it.

The Chapter 1, *Introduction* explains what TortoiseSVN is, what it does, where it comes from and the basics for installing it on your PC.

In Chapter 2, *Basic Concepts* we give a short introduction to the *Subversion* revision control system which underlies TortoiseSVN. This is borrowed from the documentation for the Subversion project and explains the different approaches to version control, and how Subversion works.

Even most Subversion users will never have to set up a server themselves. The next chapter deals with how to set up such a server, and is useful for administrators.

The chapter on Chapter 4, *The Repository* explains how to set up a local repository, which is useful for testing Subversion and TortoiseSVN using a single PC. It also explains a bit about repository administration which is also relevant to repositories located on a server.

The Chapter 5, *Daily Use Guide* is the most important section as it explains all the main features of TortoiseSVN and how to use them. It takes the form of a tutorial, starting with checking out a working copy, modifying it, committing your changes, etc. It then progresses to more advanced topics.

Chapter 6, *The SubWCRev Program* is a separate program included with TortoiseSVN which can extract the information from your working copy and write it into a file. This is useful for including build information in your projects.

The Appendix B, *How Do I...* section answers some common questions about performing tasks which are not explicitly covered elsewhere.

The section on Appendix D, *Automating TortoiseSVN* shows how the TortoiseSVN GUI dialogs can be called from the command line. This is useful for scripting where you still need user interaction.

The Appendix E, *Command Line Interface Cross Reference* give a correlation between TortoiseSVN commands and their equivalents in the Subversion command line client `svn.exe`.

3. TortoiseSVN is free!

TortoiseSVN is free. You don't have to pay to use it, and you can use it any way you want. It is developed under the GNU General Public License (GPL).

TortoiseSVN is an Open Source project. That means you have full access to the source code of this program. You can browse it on this link *http://tortoisesvn.tigris.org/svn/tortoisesvn/* [http://tortoisesvn.tigris.org/svn/tortoisesvn/]. (Username:guest, for password hit enter) The most recent version (where we're currently working) is located under `/trunk/` the released versions are located under `/tags/`.

4. Community

Both TortoiseSVN and Subversion are developed by a community of people who are working on those projects. They come from different countries all over the world and joined together to create wonderful programs.

5. Acknowledgments

Tim Kemp
   for founding the TortoiseSVN project

Stefan Küng
   for the hard work to get TortoiseSVN to what it is now

Lübbe Onken
   for the beautiful icons, logo, bug hunting, translating and managing the translations

Simon Large
   for helping with the documentation and bug hunting

The Subversion Book
   for the great introduction to Subversion and its chapter 2 which we copied here

The Tigris Style project
   for some of the styles which are reused in this documentation

Our Contributors
   for the patches, bug reports and new ideas, and for helping others by answering questions on our mailing list.

Our Donators
   for many hours of joy with the music they sent us

6. Terminology used in this document

To make reading the docs easier, the names of all the screens and Menus from TortoiseSVN are marked up in a different font. The Log Dialog for instance.

A menu choice is indicated with an arrow. TortoiseSVN → Show Log means: select *Show Log* from the *TortoiseSVN* context menu.

Where a local context menu appears within one of the TortoiseSVN dialogs, it is shown like this: Context Menu → Save As ...

User Interface Buttons are indicated like this: Press OK to continue.

User Actions are indicated using a bold font. **Alt+A**: press the **Alt**-Key on your keyboard and while holding it down press the **A**-Key as well. Right-drag: press the right mouse button and while holding it down *drag* the items to the new location.

System output and keyboard input is indicated with a `different` font as well.

Important

Important notes are marked with an icon.

Tip

Tips that make your life easier.

Caution

Places where you have to be careful what you are doing.

Warning

Where extreme care has to be taken, data corruption or other nasty things may occur if these warnings are ignored.
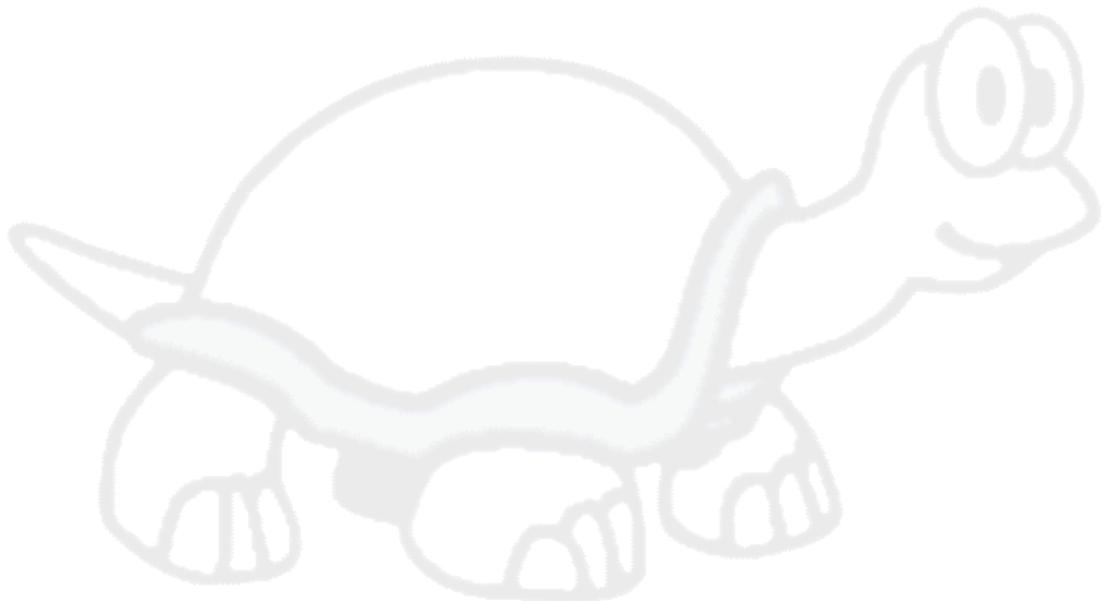
Chapter 1. Introduction

Version control is the art of managing changes to information. It has long been a critical tool for programmers, who typically spend their time making small changes to software and then undoing or checking some of those changes the next day. Imagine a team of such developers working concurrently - and perhaps even simultaneously on the very same files! - and you can see why a good system is needed to *manage the potential chaos*.

1.1. What is TortoiseSVN?

TortoiseSVN is a free open-source client for the *Subversion* version control system. That is, TortoiseSVN manages files and directories over time. Files are stored in a central *repository*. The repository is much like an ordinary file server, except that it remembers every change ever made to your files and directories. This allows you to recover older versions of your files and examine the history of how and when your data changed, and who changed it. This is why many people think of Subversion and version control systems in general as a sort of "time machine".

Some version control systems are also software configuration management (SCM) systems. These systems are specifically tailored to manage trees of source code, and have many features that are specific to software development - such as natively understanding programming languages, or supplying tools for building software. Subversion, however, is not one of these systems; it is a general system that can be used to manage *any* collection of files, including source code.

1.2. TortoiseSVN's History

In 2002, Tim Kemp found that Subversion was a very good version control system, but it lacked a good GUI client. The idea for a Subversion client as a Windows shell integration was inspired by the similar client for CVS named TortoiseCVS.

Tim studied the source code of TortoiseCVS and used it as a base for TortoiseSVN. He then started the project, registered the domain `tortoisesvn.org` and put the source code online. During that time, Stefan Küng was looking for a good and free version control system and found Subversion and the source for TortoiseSVN. Since TortoiseSVN was still not ready for use then he joined the project and started programming. Soon he rewrote most of the existing code and started adding commands and features, up to a point where nothing of the original code remained.

As Subversion became more stable it attracted more and more users who also started using TortoiseSVN as their Subversion client. The user base grew quickly (and is still growing every day). That's when Lübbe Onken offered to help out with some nice icons and a logo for TortoiseSVN. And he takes care of the website and manages the translation.

1.3. TortoiseSVN's Features

What makes TortoiseSVN such a good Subversion client? Here's a short list of features.

Shell integration

> TortoiseSVN integrates seamlessly into the Windows shell (i.e. the explorer). This means you can keep working with the tools you're already familiar with. And you do not have to change into a different application each time you need functions of the version control!
>
> And you are not even forced to use the Windows Explorer. TortoiseSVN's context menus work in many other file managers, and in the File/Open dialog which is common to most standard Windows applications. You should, however, bear in mind that TortoiseSVN is intentionally developed as extension for the Windows Explorer. Thus it is possible that in other applications the integration is not as complete and e.g. the icon overlays may not be shown.

Icon overlays
: The status of every versioned file and folder is indicated by small overlay icons. That way you can see right away what the status of your working copy is.

Easy access to Subversion commands
: All Subversion commands are available from the explorer context menu. TortoiseSVN adds its own submenu there.

Since TortoiseSVN is a Subversion client, we would also like to show you some of the features of Subversion itself:

Directory versioning
: CVS only tracks the history of individual files, but Subversion implements a "virtual" versioned filesystem that tracks changes to whole directory trees over time. Files *and* directories are versioned. As a result, there are real client-side **move** and **copy** commands that operate on files and directories.

Atomic commits
: A commit either goes into the repository completely, or not at all. This allows developers to construct and commit changes as logical chunks.

Versioned metadata
: Each file and directory has an invisible set of "properties" attached. You can invent and store any arbitrary key/value pairs you wish. Properties are versioned over time, just like file contents.

Choice of network layers
: Subversion has an abstracted notion of repository access, making it easy for people to implement new network mechanisms. Subversion's "advanced" network server is a module for the Apache web server, which speaks a variant of HTTP called WebDAV/DeltaV. This gives Subversion a big advantage in stability and interoperability, and provides various key features for free: authentication, authorization, wire compression, and repository browsing, for example. A smaller, standalone Subversion server process is also available. This server speaks a custom protocol which can be easily tunneled over ssh.

Consistent data handling
: Subversion expresses file differences using a binary differencing algorithm, which works identically on both text (human-readable) and binary (human-unreadable) files. Both types of files are stored equally compressed in the repository, and differences are transmitted in both directions across the network.

Efficient branching and tagging
: The cost of branching and tagging need not be proportional to the project size. Subversion creates branches and tags by simply copying the project, using a mechanism similar to a hard-link. Thus these operations take only a very small, constant amount of time, and very little space in the repository.

Hackability
: Subversion has no historical baggage; it is implemented as a collection of shared C libraries with well-defined APIs. This makes Subversion extremely maintainable and usable by other applications and languages.

1.4. Installing TortoiseSVN

1.4.1. System requirements

TortoiseSVN runs on Windows 2000 SP2, Windows XP or higher. Windows 98, Windows ME and Windows NT4 are no longer supported since TortoiseSVN 1.2.0, but you can still download the older versions if you really need them.

If you encounter any problems during or after installing TortoiseSVN please refer to *Appendix A, Frequently Asked Questions (FAQ)* first.

1.4.2. Installation

TortoiseSVN comes with an easy to use installer. Double click on the installer file and follow the instructions. The installer will take care of the rest.

> **Important**
>
> You need Administrator privileges to install TortoiseSVN.

1.4.3. Language Packs

The TortoiseSVN user interface has been translated into many different languages, so you may be able to download a language pack to suit your needs. You can find the language packs on our *translation status page* [http://tortoisesvn.net/translation_status]. And if there is no language pack available yet, why not join the team and submit your own translation ;-)

Each language pack is packaged as a `.exe` installer. Just run the install program and follow the instructions. Next time you restart, the translation will be available.

1.4.4. Spellchecker

TortoiseSVN includes a spell checker which allows you to check your commit log messages. This is especially useful if the project language is not your native language. The spell checker uses the same dictionary files as *OpenOffice* [http://openoffice.org] and *Mozilla* [http://mozilla.org].

The installer automatically adds the US and UK English dictionaries. If you want other languages, the easiest option is simply to install one of TortoiseSVN's language packs. This will install the appropriate dictionary files as well as the TortoiseSVN local user interface. Next time you restart, the dictionary will be available too.

Or you can install the dictionaries yourself. If you have OpenOffice or Mozilla installed, you can copy those dictionaries, which are located in the installation folders for those applications. Otherwise, you need to download the required dictionary files from *http://wiki.services.openoffice.org/wiki/Dictionaries* [http://wiki.services.openoffice.org/wiki/Dictionaries]

Once you have got the dictionary files, you probably need to rename them so that the filenames only have the locale chars in it. Example:

- `en_US.aff`

- `en_US.dic`

Then just copy them to the `bin` sub-folder of the TortoiseSVN installation folder. Normally this will be `C:\Program Files\TortoiseSVN\bin`. If you don't want to litter the `bin` sub-folder, you can instead place your spell checker files in `C:\Program Files\TortoiseSVN\Languages`. If that folder isn't there, you have to create it first. The next time you start TortoiseSVN, the spell checker will be available.

If you install multiple dictionaries, TortoiseSVN uses these rules to select which one to use.

1. Check the `tsvn:projectlanguage` setting. Refer to Section 5.17, "Project Settings" for information about setting project properties.

2. If no project language is set, or that language is not installed, try the language corresponding to the Windows locale.

3. If the exact Windows locale doesn't work, try the "Base" language, eg. `de_CH` (Swiss-German) falls back to `de_DE` (German).

4. If none of the above works, then the default language is English, which is included with the standard installation.

Chapter 2. Basic Concepts

This chapter is a slightly modified version of the same chapter in the Subversion book. An online version of the Subversion book is available here: *http://svnbook.red-bean.com/* [http://svnbook.red-bean.com/].

This chapter is a short, casual introduction to Subversion. If you're new to version control, this chapter is definitely for you. We begin with a discussion of general version control concepts, work our way into the specific ideas behind Subversion, and show some simple examples of Subversion in use.

Even though the examples in this chapter show people sharing collections of program source code, keep in mind that Subversion can manage any sort of file collection - it's not limited to helping computer programmers.

2.1. The Repository

Subversion is a centralized system for sharing information. At its core is a *repository*, which is a central store of data. The repository stores information in the form of a *filesystem tree* - a typical hierarchy of files and directories. Any number of *clients* connect to the repository, and then read or write to these files. By writing data, a client makes the information available to others; by reading data, the client receives information from others.
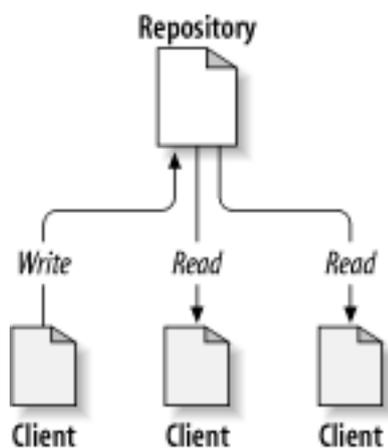
Figure 2.1. A Typical Client/Server System

So why is this interesting? So far, this sounds like the definition of a typical file server. And indeed, the repository *is* a kind of file server, but it's not your usual breed. What makes the Subversion repository special is that *it remembers every change* ever written to it: every change to every file, and even changes to the directory tree itself, such as the addition, deletion, and rearrangement of files and directories.

When a client reads data from the repository, it normally sees only the latest version of the filesystem tree. But the client also has the ability to view *previous* states of the filesystem. For example, a client can ask historical questions like, "what did this directory contain last Wednesday?", or "who was the last person to change this file, and what changes did they make?" These are the sorts of questions that are at the heart of any *version control system*: systems that are designed to record and track changes to data over time.

2.2. Versioning Models

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

2.2.1. The Problem of File-Sharing

Consider this scenario: suppose we have two co-workers, Harry and Sally. They each decide to edit the same repository file at the same time. If Harry saves his changes to the repository first, then it's possible that (a few moments later) Sally could accidentally overwrite them with her own new version of the file. While Harry's version of the file won't be lost forever (because the system remembers every change), any changes Harry made *won't* be present in Sally's newer version of the file, because she never saw Harry's changes to begin with. Harry's work is still effectively lost - or at least missing from the latest version of the file - and probably by accident. This is definitely a situation we want to avoid!
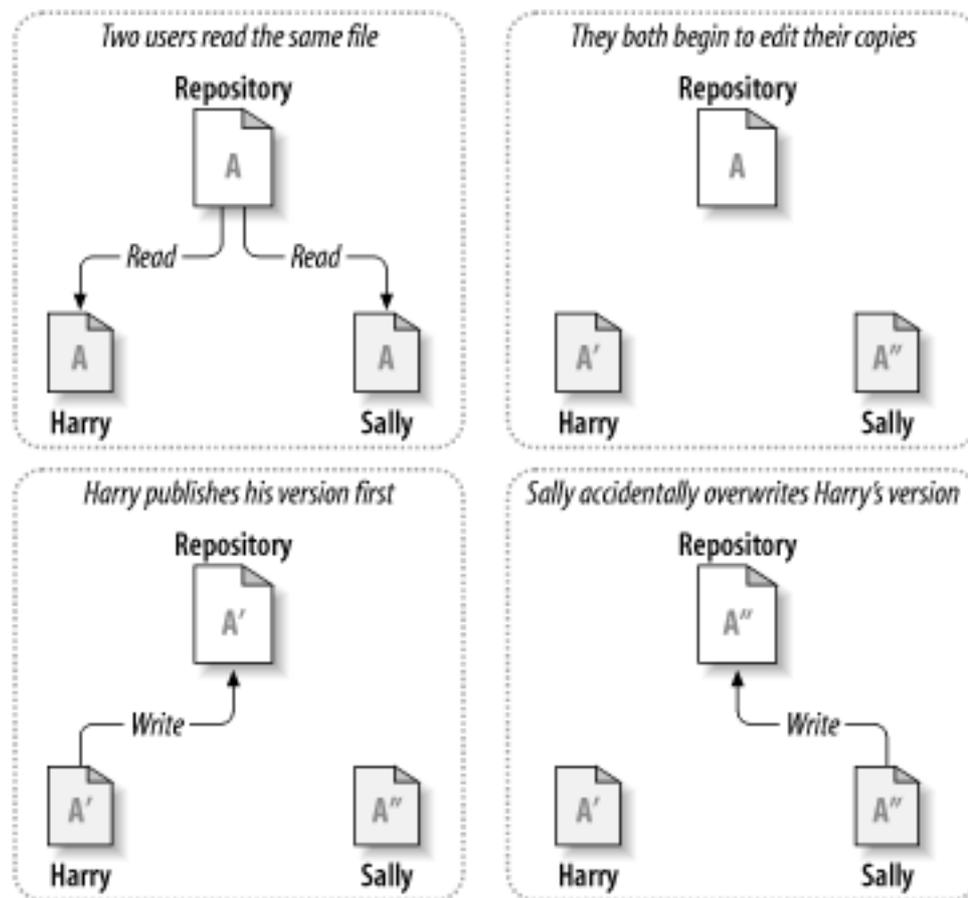
Figure 2.2. The Problem to Avoid

2.2.2. The Lock-Modify-Unlock Solution

Many version control systems use a *lock-modify-unlock* model to address this problem, which is a very simple solution. In such a system, the repository allows only one person to change a file at a time. First Harry must *lock* the file before he can begin making changes to it. Locking a file is a lot like borrowing a book from the library; if Harry has locked a file, then Sally cannot make any changes to it. If she tries to lock the file, the repository will deny the request. All she can do is read the file, and wait for Harry to finish his changes and release his lock. After Harry unlocks the file, his turn is over, and now Sally can take her turn by locking and editing.
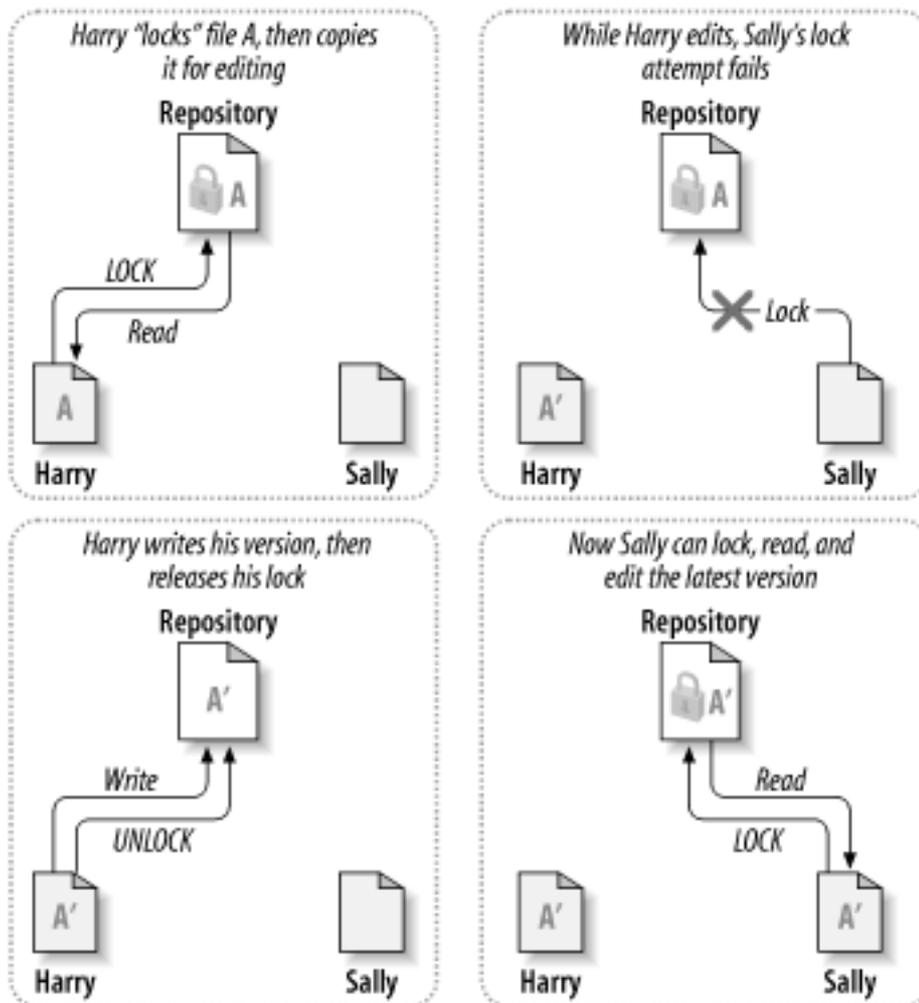
Figure 2.3. The Lock-Modify-Unlock Solution

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

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

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

- *Locking may create a false sense of security.* Pretend that Harry locks and edits file A, while Sally simultaneously locks and edits file B. But suppose that A and B depend on one another, and the changes made to each are semantically incompatible. Suddenly A and B don't work together anymore. The locking system was powerless to prevent the problem - yet it somehow provided a sense of false security. It's easy for Harry and Sally to imagine that by locking files, each is beginning a safe, insulated task, and thus inhibits them from discussing their incompatible changes early on.

2.2.3. The Copy-Modify-Merge Solution

Subversion, CVS, and other version control systems use a *copy-modify-merge* model as an alternative to locking. In this model, each user's client reads the repository and creates a personal *working copy* of the

file or project. Users then work in parallel, modifying their private copies. Finally, the private copies are merged together into a new, final version. The version control system often assists with the merging, but ultimately a human being is responsible for making it happen correctly.

Here's an example. Say that Harry and Sally each create working copies of the same project, copied from the repository. They work concurrently, and make changes to the same file A within their copies. Sally saves her changes to the repository first. When Harry attempts to save his changes later, the repository informs him that his file A is *out-of-date*. In other words, that file A in the repository has somehow changed since he last copied it. So Harry asks his client to *merge* any new changes from the repository into his working copy of file A. Chances are that Sally's changes don't overlap with his own; so once he has both sets of changes integrated, he saves his working copy back to the repository.
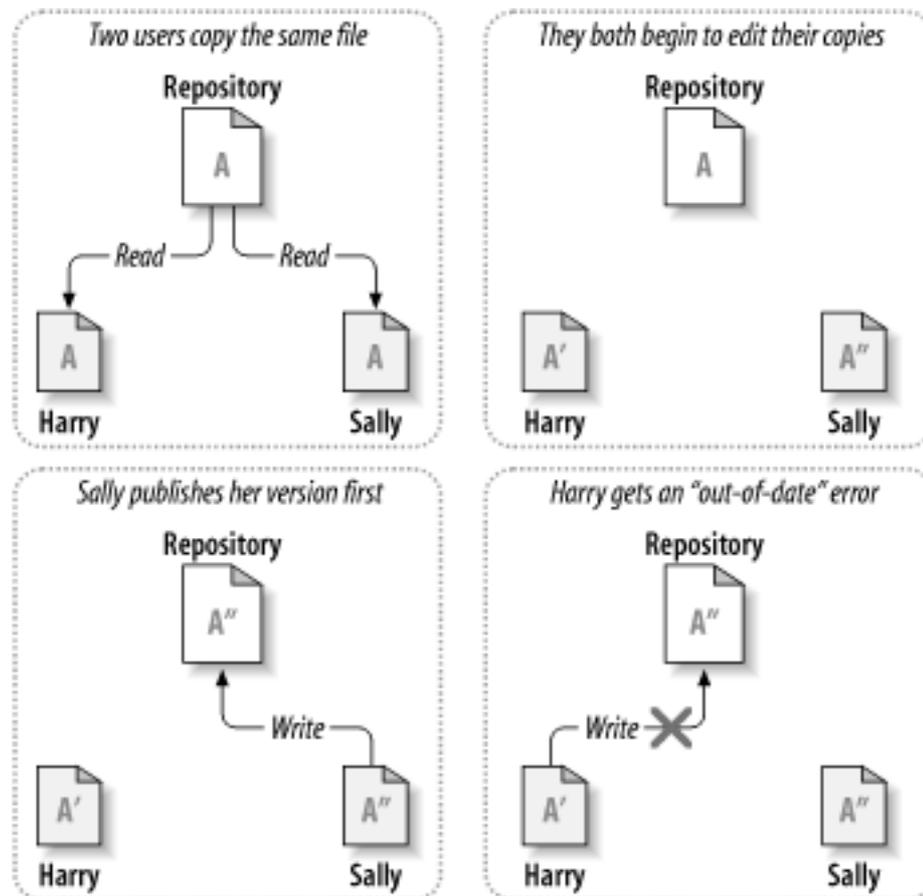
Figure 2.4. The Copy-Modify-Merge Solution
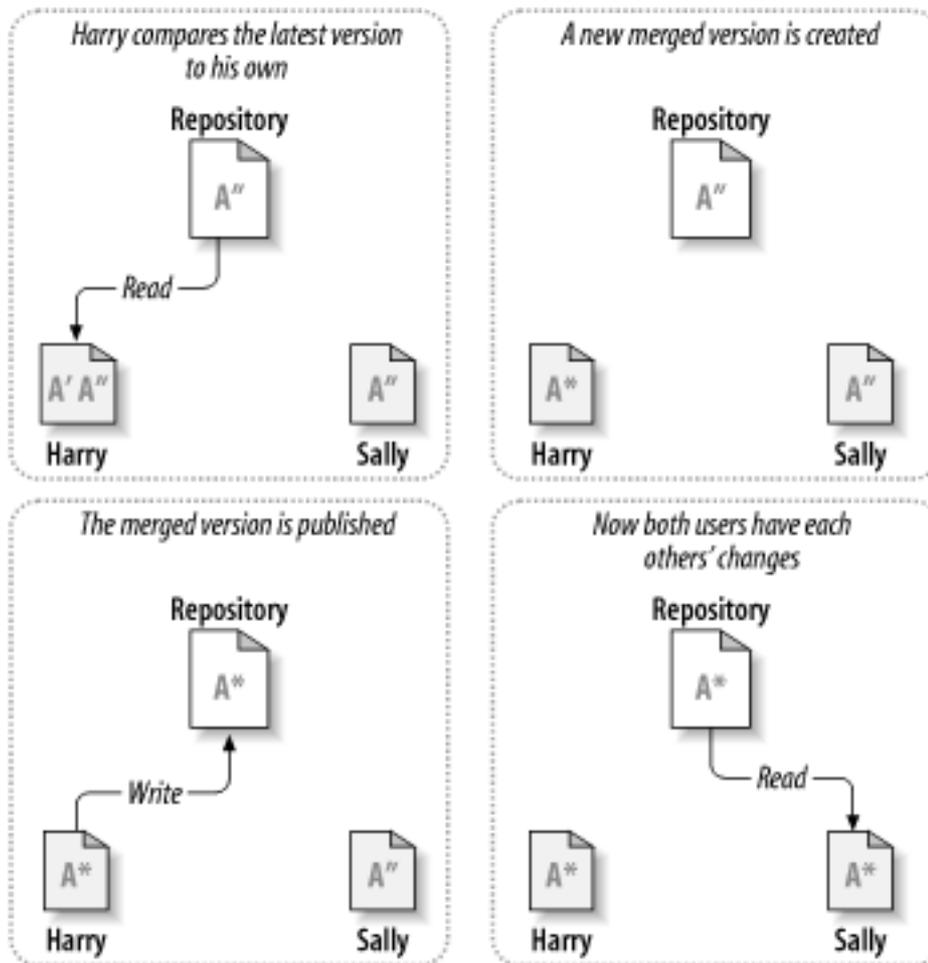
Figure 2.5. ...Copy-Modify-Merge Continued

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

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

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

There is one common situation where the lock-modify-unlock model comes out better, and that is where you have unmergeable files. For example if your repository contains some graphic images, and two people change the image at the same time, there is no way for those changes to be merged together. Either Harry or Sally will lose their changes.

2.2.4. What does Subversion Do?

Subversion uses the copy-modify-merge solution by default, and in many cases this is all you will ever need. However, as of Version 1.2, Subversion also supports file locking, so if you have unmergeable files, or if you are simply forced into a locking policy by management, Subversion will still provide the features you need.

2.3. Subversion in Action

2.3.1. Working Copies

You've already read about working copies; now we'll demonstrate how the Subversion client creates and uses them.

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

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

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

A typical Subversion repository often holds the files (or source code) for several projects; usually, each project is a subdirectory in the repository's filesystem tree. In this arrangement, a user's working copy will usually correspond to a particular subtree of the repository.

For example, suppose you have a repository that contains two software projects.
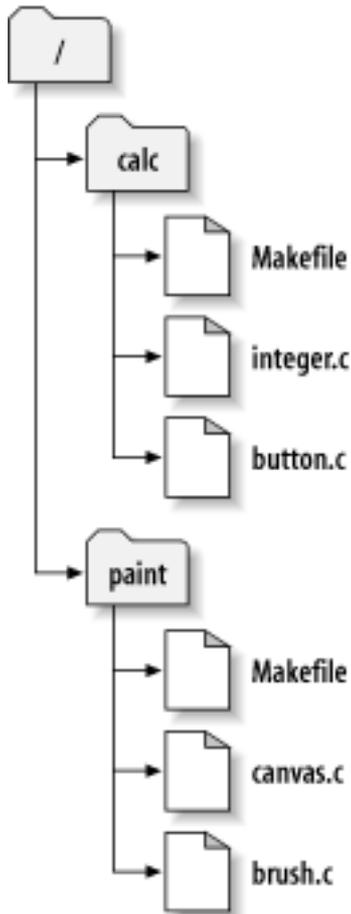
Figure 2.6. The Repository's Filesystem

In other words, the repository's root directory has two subdirectories: `paint` and `calc`.

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

Suppose you make changes to `button.c`. Since the `.svn` directory remembers the file's modification date and original contents, Subversion can tell that you've changed the file. However, Subversion does not make your changes public until you explicitly tell it to. The act of publishing your changes is more commonly known as *committing* (or *checking in*) changes to the repository.

To publish your changes to others, you can use Subversion's **commit** command.

Now your changes to `button.c` have been committed to the repository; if another user checks out a working copy of `/calc`, they will see your changes in the latest version of the file.

Suppose you have a collaborator, Sally, who checked out a working copy of `/calc` at the same time you did. When you commit your change to `button.c`, Sally's working copy is left unchanged; Subversion only modifies working copies at the user's request.

To bring her project up to date, Sally can ask Subversion to *update* her working copy, by using the Subversion **update** command. This will incorporate your changes into her working copy, as well as any others that have been committed since she checked it out.

Note that Sally didn't need to specify which files to update; Subversion uses the information in the `.svn` directory, and further information in the repository, to decide which files need to be brought up to date.

2.3.2. Repository URLs

Subversion repositories can be accessed through many different methods - on local disk, or through various network protocols. A repository location, however, is always a URL. The URL schema indicates the access method:

Schema	Access Method
`file://`	Direct repository access on local or network drive.
`http://`	Access via WebDAV protocol to Subversion-aware Apache server.
`https://`	Same as `http://`, but with SSL encryption.
`svn://`	Unauthenticated TCP/IP access via custom protocol to a `svnserve` server.
`svn+ssh://`	authenticated, encrypted TCP/IP access via custom protocol to a `svnserve` server.

Table 2.1. Repository Access URLs

For the most part, Subversion's URLs use the standard syntax, allowing for server names and port numbers to be specified as part of the URL. The `file://` access method is normally used for local access, although it can be used with UNC paths to a networked host. The URL therefore takes the form `file://hostname/path/to/repos`. For the local machine, the `hostname` portion of the URL is required to be either absent or `localhost`. For this reason, local paths normally appear with three slashes, `file:///path/to/repos`.

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

```
file:///X:/path/to/repos
...
file:///X|/path/to/repos
...
```

Note that a URL uses ordinary slashes even though the native (non-URL) form of a path on Windows uses backslashes.

You can safely access a FSFS repository via a network share, but you *cannot* access a BDB repository in this way.

> ## Warning
>
> Do not create or access a Berkeley DB repository on a network share. It *cannot* exist on a remote filesystem. Not even if you have the network drive mapped to a drive letter. If you attempt to use Berkeley DB on a network share, the results are unpredictable - you may see mysterious errors right away, or it may be months before you discover that your repository database is subtly corrupted.

2.3.3. Revisions

A **svn commit** operation can publish changes to any number of files and directories as a single atomic transaction. In your working copy, you can change files' contents, create, delete, rename and copy files and directories, and then commit the complete set of changes as a unit.

In the repository, each commit is treated as an atomic transaction: either all the commits changes take place, or none of them take place. Subversion retains this atomicity in the face of program crashes, system crashes, network problems, and other users' actions.

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

A nice way to visualize the repository is as a series of trees. Imagine an array of revision numbers, starting at 0, stretching from left to right. Each revision number has a filesystem tree hanging below it, and each tree is a "snapshot" of the way the repository looked after each commit.
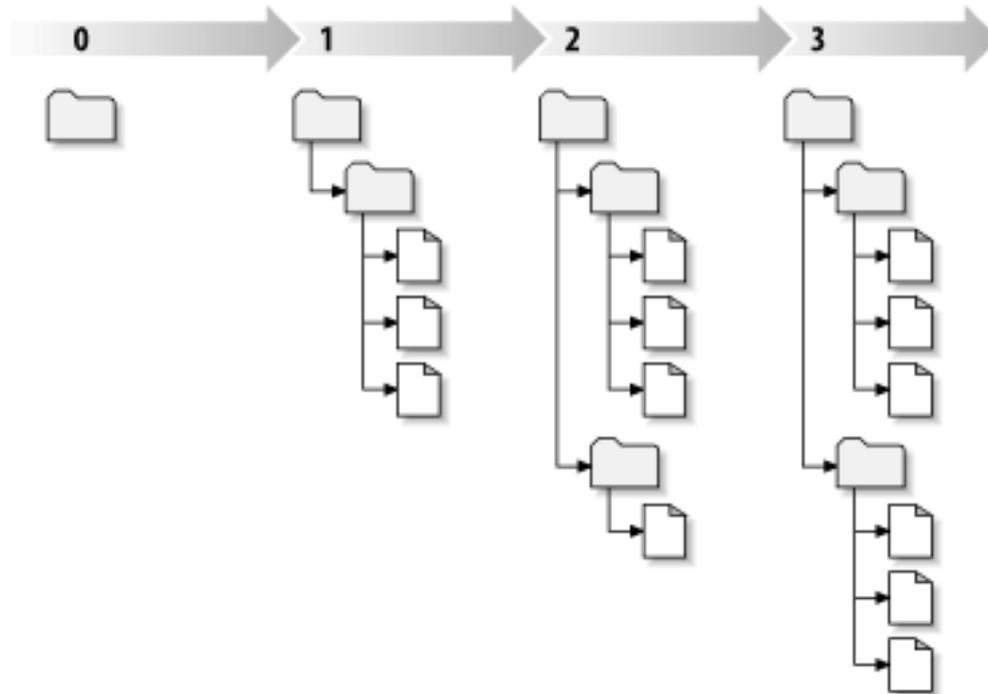
Figure 2.7. The Repository

Global Revision Numbers

Unlike those of many other version control systems, Subversion's revision numbers apply to *entire trees*, not individual files. Each revision number selects an entire tree, a particular state of the repository after some committed change. Another way to think about it is that revision N represents the state of the repository filesystem after the Nth commit. When a Subversion user talks about ``revision 5 of foo.c'', they really mean ``foo.c as it appears in revision 5.'' Notice that in general, revisions N and M of a file do *not* necessarily differ!

It's important to note that working copies do not always correspond to any single revision in the repository; they may contain files from several different revisions. For example, suppose you check out a working copy from a repository whose most recent revision is 4:

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

At the moment, this working directory corresponds exactly to revision 4 in the repository. However, suppose you make a change to button.c, and commit that change. Assuming no other commits have

taken place, your commit will create revision 5 of the repository, and your working copy will now look like this:

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

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

```
calc/Makefile:6
     integer.c:6
     button.c:6
```

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

2.3.4. How Working Copies Track the Repository

For each file in a working directory, Subversion records two essential pieces of information in the `.svn/` administrative area:

- what revision your working file is based on (this is called the file's *working revision*), and

- a timestamp recording when the local copy was last updated by the repository.

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

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

Locally changed, and current
> The file has been changed in the working directory, and no changes to that file have been committed to the repository since its base revision. There are local changes that have not been committed to the repository, thus a **commit** of the file will succeed in publishing your changes, and an **update** of the file will do nothing.

Unchanged, and out-of-date
> The file has not been changed in the working directory, but it has been changed in the repository. The file should eventually be updated, to make it current with the public revision. A **commit** of the file will do nothing, and an **update** of the file will fold the latest changes into your working copy.

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

2.4. Summary

We've covered a number of fundamental Subversion concepts in this chapter:

- We've introduced the notions of the central repository, the client working copy, and the array of repository revision trees.

- We've seen some simple examples of how two collaborators can use Subversion to publish and receive changes from one another, using the 'copy-modify-merge' model.

- We've talked a bit about the way Subversion tracks and manages information in a working copy.

Chapter 3. Setting Up A Server

To use TortoiseSVN (or any other Subversion client), you need a place where your repositories are located. You can either store your repositories locally and access them using the `file://` protocol or you can place them on a server and access them with the `http://` or `svn://` protocols. The two server protocols can also be encrypted. You use `https://` or `svn+ssh://`. This chapter shows you step by step on how you can set up such a server on a Windows machine.

More detailed information on the Subversion server options, and how to choose the best architecture for your situation, can be found in the Subversion book under *Server Configuration* [http://svnbook.red-bean.com/en/1.4/svn.serverconfig.html].

If you don't have a server and you work alone then local repositories are probably your best choice. You can skip this chapter and go directly to Chapter 4, *The Repository*.

If you were thinking about setting up a multi-user repository on a network share, think again. Read Section 4.1.4, "Accessing a Repository on a Network Share" to find out why we think this is a bad idea.

3.1. Apache Based Server

3.1.1. Introduction

The most flexible of all possible server setups for Subversion is the Apache based one. Although a bit more complicated to set up, it offers benefits that other servers cannot:

WebDAV
> The Apache based Subversion server uses the WebDAV protocol which is supported by many other programs as well. You could e.g. mount such a repository as a "Web folder" in the Windows explorer and then access it like any other folder in the file system.

Browsing The Repository
> You can point your browser to the URL of your repository and browse the contents of it without having a Subversion client installed. This gives access to your data to a much wider circle of users.

Authentication
> You can use any authentication mechanism Apache supports, including SSPI and LDAP.

Security
> Since Apache is very stable and secure, you automatically get the same security for your repository. This includes SSL encryption.

3.1.2. Installing Apache

The first thing you need before installing Apache is a computer with Windows 2000, Windows XP+SP1, Windows 2003, Vista or Server 2008.

> ### Warning
>
> Please note that Windows XP without the service pack 1 will lead to bogus network data and could therefore corrupt your repository!

1. Download the latest version of the Apache web server from *http://httpd.apache.org/download.cgi* [http://httpd.apache.org/download.cgi]. Make sure that you download the version 2.2.x - the version 1.3.xx won't work!

The msi installer for Apache can be found by clicking on `other   files`, then browse to `binaries/win32`. You may want to choose the msi file `apache-2.2.x-win32-x86-openssl-0.9.x.msi` (the one that includes OpenSSL).

2. Once you have the Apache2 installer you can double click on it and it will guide you through the installation process. Make sure that you enter the server-URL correctly (if you don't have a DNS name for your server just enter the IP-address). I recommend to install Apache *for All Users, on Port 80, as a Service*. Note: if you already have IIS or any other program running which listens on port 80 the installation might fail. If that happens, go to the programs directory, `\Apache Group \Apache2\conf` and locate the file `httpd.conf`. Edit that file so that `Listen 80` is changed to a free port, e.g. `Listen 81`. Then restart the installation - this time it should finish without problems.

3. Now test if the Apache web server is running correctly by pointing your web browser to `http://localhost/` - a preconfigured Website should show up.

> ### Caution
>
> If you decide to install Apache as a service, be warned that by default it will run as the local system account. It would be a more secure practice for you to create a separate account for Apache to run as.
>
> Make sure that the account on the server that Apache is running as has an explicit entry in the repository directory's access control list (right-click directory | properties | security), with full control. Otherwise, users will not be able to commit their changes.
>
> Even if Apache runs as local system, you still need such an entry (which will be the SYSTEM account in this case).
>
> If Apache does not have this permission set up, your users will get "Access denied" error messages, which show up in the Apache error log as error 500.

3.1.3. Installing Subversion

1. Download the latest version of the Subversion Win32 binaries for Apache. Be sure to get the right version to integrate with your version of Apache, otherwise you will get an obscure error message when you try to restart. If you have Apache 2.2.x go to *http://subversion.tigris.org/servlets/ ProjectDocumentList?folderID=8100*   [http://subversion.tigris.org/servlets/ProjectDocumentList? folderID=8100].

2. Run the Subversion installer and follow the instructions. If the Subversion installer recognized that you've installed Apache, then you're almost done. If it couldn't find an Apache server then you have to do some additional steps.

3.

   Using the windows explorer, go to the installation directory of Subversion (usually `c: \program  files\Subversion`) and find the files `/httpd/mod_dav_svn.so` and `mod_authz_svn.so`. Copy these files to the Apache modules directory (usually `c:\program files\apache group\apache2\modules`).

4. Copy the file `/bin/libdb*.dll` and `/bin/intl3_svn.dll` from the Subversion installation directory to the Apache bin directory.

5. Edit Apache's configuration file (usually   `C:\Program  Files\Apache  Group \Apache2\conf\httpd.conf`) with a text editor such as Notepad and make the following changes:

   Uncomment (remove the '#' mark) the following lines:

```
#LoadModule dav_fs_module modules/mod_dav_fs.so
#LoadModule dav_module modules/mod_dav.so
```

Add the following two lines to the end of the `LoadModule` section.

```
LoadModule dav_svn_module modules/mod_dav_svn.so
LoadModule authz_svn_module modules/mod_authz_svn.so
```

3.1.4. Configuration

Now you have set up Apache and Subversion, but Apache doesn't know how to handle Subversion clients like TortoiseSVN yet. To get Apache to know which URL will be used for Subversion repositories you have to edit the Apache configuration file (usually located in `c:\program files\apache group \apache2\conf\httpd.conf`) with any text editor you like (e.g. Notepad):

1. At the end of the config file add the following lines:

```
<Location /svn>
  DAV svn
  SVNListParentPath on
  SVNParentPath D:\SVN
  #SVNIndexXSLT "/svnindex.xsl"
  AuthType Basic
  AuthName "Subversion repositories"
  AuthUserFile passwd
  #AuthzSVNAccessFile svnaccessfile
  Require valid-user
</Location>
```

This configures Apache so that all your Subversion repositories are physically located below `D:\SVN`. The repositories are served to the outside world from the URL: `http://MyServer/svn/` . Access is restricted to known users/passwords listed in the `passwd` file.

2. To create the `passwd` file, open the command prompt (DOS-Box) again, change to the `apache2` folder (usually `c:\program files\apache group\apache2`) and create the file by entering

```
bin\htpasswd -c passwd <username>
```

This will create a file with the name `passwd` which is used for authentication. Additional users can be added with

```
bin\htpasswd passwd <username>
```

3. Restart the Apache service again.

4. Point your browser to `http://MyServer/svn/MyNewRepository` (where `MyNewRepository` is the name of the Subversion repository you created before). If all went well you should be prompted for a username and password, then you can see the contents of your repository.

A short explanation of what you just entered:

Setting	Explanation
<Location /svn>	means that the Subversion repositories are available from the URL `http://MyServer/svn/`
DAV svn	tells Apache which module will be responsible to serve that URL - in this case the Subversion module.
SVNListParentPath on	For Subversion version 1.3 and higher, this directive enables listing all the available repositories under `SVNParentPath`.
SVNParentPath D:\SVN	tells Subversion to look for repositories below `D:\SVN`
SVNIndexXSLT "/svnindex.xsl"	Used to make the browsing with a web browser prettier.
AuthType Basic	is to activate basic authentication, i.e. Username/password
AuthName "Subversion repositories"	is used as an information whenever an authentication dialog pops up to tell the user what the authentication is for
AuthUserFile passwd	specifies which password file to use for authentication
AuthzSVNAccessFile	Location of the Access file for paths inside a Subversion repository
Require valid-user	specifies that only users who entered a correct username/password are allowed to access the URL

Table 3.1. Apache `httpd.conf` Settings

But that's just an example. There are many, many more possibilities of what you can do with the Apache web server.

- If you want your repository to have read access for everyone but write access only for specific users you can change the line

```
Require valid-user
```

to

```
<LimitExcept GET PROPFIND OPTIONS REPORT>
Require valid-user
</LimitExcept>
```

- Using a `passwd` file limits and grants access to all of your repositories as a unit. If you want more control over which users have access to each folder inside a repository you can uncomment the line

```
#AuthzSVNAccessFile svnaccessfile
```

and create a Subversion access file. Apache will make sure that only valid users are able to access your `/svn` location, and will then pass the username to Subversion's `AuthzSVNAccessFile` module so that it can enforce more granular access based upon rules listed in the Subversion access file. Note that paths are specified either as `repos:path` or simply `path`. If you don't specify a particular repository, that access rule will apply to all repositories under `SVNParentPath`. The format of the authorization-policy file used by `mod_authz_svn` is described in Section 3.1.6, "Path-Based Authorization"

- To make browsing the repository with a web browser 'prettier', uncomment the line

```
#SVNIndexXSLT "/svnindex.xsl"
```

and put the files `svnindex.xsl`, `svnindex.css` and `menucheckout.ico` in your document root directory (usually `C:/Program Files/Apache Group/Apache2/htdocs`). The directory is set with the `DocumentRoot` directive in your Apache config file.

You can get those three files directly from our source repository at *http://tortoisesvn.tigris.org/svn/ tortoisesvn/trunk/contrib/other/svnindex* [http://tortoisesvn.tigris.org/svn/tortoisesvn/trunk/contrib/ other/svnindex]. If you're asked for authentication for this link, enter `guest` as username and leave the password empty.

The XSL file from the TortoiseSVN repository has a nice gimmick: if you browse the repository with your web browser, then every folder in your repository has an icon on the right shown. If you click on that icon, the TortoiseSVN checkout dialog is started for this URL.

3.1.5. Multiple Repositories

If you used the `SVNParentPath` directive then you don't have to change the Apache config file every time you add a new Subversion repository. Simply create the new repository under the same location as the first repository and you're done! In my company I have direct access to that specific folder on the server via SMB (normal windows file access). So I just create a new folder there, run the TortoiseSVN command TortoiseSVN → Create repository here... and a new project has a home...

If you are using Subversion 1.3 or later, you can use the `SVNListParentPath on` directive to allow Apache to produce a listing of all available projects if you point your browser at the parent path rather than at a specific repository.

3.1.6. Path-Based Authorization

The `mod_authz_svn` module permits fine-grained control of access permissions based on user names and repository paths. This is available with the Apache server, and as of Subversion 1.3 it is available with svnserve as well.

An example file would look like this:

```
[groups]
admin = john, kate
devteam1 = john, rachel, sally
devteam2 = kate, peter, mark
docs = bob, jane, mike
training = zak
# Default access rule for ALL repositories
# Everyone can read, admins can write, Dan German is excluded.
[/]
* = r
@admin = rw
dangerman =
# Allow developers complete access to their project repos
[proj1:/]
@devteam1 = rw
[proj2:/]
@devteam2 = rw
[bigproj:/]
@devteam1 = rw
@devteam2 = rw
trevor = rw
# Give the doc people write access to all the docs folders
```

```
[/trunk/doc]
@docs = rw
# Give trainees write access in the training repository only
[TrainingRepos:/]
@training = rw
```

Note that checking every path can be an expensive operation, particularly in the case of the revision log. The server checks every changed path in each revision and checks it for readability, which can be time-consuming on revisions which affect large numbers of files.

Authentication and authorization are separate processes. If a user wants to gain access to a repository path, she has to meet *both*, the usual authentication requirements and the authorization requirements of the access file.

3.1.7. Authentication With a Windows Domain

As you might have noticed you need to make a username/password entry in the `passwd` file for each user separately. And if (for security reasons) you want your users to periodically change their passwords you have to make the change manually.

But there's a solution for that problem - at least if you're accessing the repository from inside a LAN with a windows domain controller: `mod_auth_sspi`!

The original SSPI module was offered by Syneapps including source code. But the development for it has been stopped. But don't despair, the community has picked it up and improved it. It has a new home on *SourceForge* [http://sourceforge.net/projects/mod-auth-sspi/].

- Download the module which matches your apache version, then copy the file `mod_auth_sspi.so` into the Apache modules folder.

- Edit the Apache config file: add the line

  ```
  LoadModule sspi_auth_module modules/mod_auth_sspi.so
  ```

  to the `LoadModule` section. Make sure you insert this line *before* the line

  ```
  LoadModule auth_module modules/mod_auth.so
  ```

- To make the Subversion location use this type of authentication you have to change the line

  ```
  AuthType Basic
  ```

  to

  ```
  AuthType SSPI
  ```

  also you need to add

  ```
  SSPIAuth On
  SSPIAuthoritative On
  SSPIDomain <domaincontroller>
  SSPIOmitDomain on
  SSPIUsernameCase lower
  ```

```
SSPIPerRequestAuth on
SSPIOfferBasic On
```

within the `<Location /svn>` block. If you don't have a domain controller, leave the name of the domain control as `<domaincontroller>`.

Note that if you are authenticating using SSPI, then you don't need the `AuthUserFile` line to define a password file any more. Apache authenticates your username and password against your windows domain instead. You will need to update the users list in your `svnaccessfile` to reference DOMAIN `\username` as well.

> **Important**
>
> The SSPI authentication is only enabled for SSL secured connections (https). If you're only using normal http connections to your server, it won't work.
>
> To enable SSL on your server, see the chapter: Section 3.1.9, "Securing the server with SSL"

> **Tip**
>
> Subversion `AuthzSVNAccessFile` files are case sensitive in regard to user names (`JUser` is different from `juser`).
>
> In Microsoft's world, Windows domains and user names are not case sensitive. Even so, some network administrators like to create user accounts in CamelCase (e.g. `JUser`).
>
> This difference can bite you when using SSPI authentication as the windows domain and user names are passed to Subversion in the same case as the user types them in at the prompt. Internet Explorer often passes the username to Apache automatically using whatever case the account was created with.
>
> The end result is that you may need at least two entries in your `AuthzSVNAccessFile` for each user -- a lowercase entry and an entry in the same case that Internet Explorer passes to Apache. You will also need to train your users to also type in their credentials using lower case when accessing repositories via TortoiseSVN.
>
> Apache's Error and Access logs are your best friend in deciphering problems such as these as they will help you determine the username string passed onto Subversion's `AuthzSVNAccessFile` module. You may need to experiment with the exact format of the user string in the `svnaccessfile` (e.g. `DOMAIN\user` vs. `DOMAIN//user`) in order to get everything working.

3.1.8. Multiple Authentication Sources

It is also possible to have more than one authentication source for your Subversion repository. To do this, you need to make each authentication type non-authoritative, so that Apache will check multiple sources for a matching username/password.

A common scenario is to use both Windows domain authentication and a `passwd` file, so that you can provide SVN access to users who don't have a Windows domain login.

• To enable both Windows domain and `passwd` file authentication, add the following entries within the `<Location>` block of your Apache config file:

```
AuthBasicAuthoritative Off
SSPIAuthoritative Off
```

Here is an example of the full Apache configuration for combined Windows domain and `passwd` file authentication:

```
<Location /svn>
  DAV svn
  SVNListParentPath on
  SVNParentPath D:\SVN

  AuthName "Subversion repositories"
  AuthzSVNAccessFile svnaccessfile.txt

# NT Domain Logins.
  AuthType SSPI
  SSPIAuth On
  SSPIAuthoritative Off
  SSPIDomain <domaincontroller>
  SSPIOfferBasic On

# Htpasswd Logins.
  AuthType Basic
  AuthBasicAuthoritative Off
  AuthUserFile passwd

  Require valid-user
</Location>
```

3.1.9. Securing the server with SSL

Even though Apache 2.2.x has OpenSSL support, it is not activated by default. You need to activate this manually.

1. In the apache config file, uncomment the lines:

   ```
   #LoadModule ssl_module modules/mod_ssl.so
   ```

   and at the bottom

   ```
   #Include conf/extra/httpd-ssl.conf
   ```

   then change the line (on one line)

   ```
   SSLMutex "file:C:/Program Files/Apache Software Foundation/\
   Apache2.2/logs/ssl_mutex"
   ```

   to

   ```
   SSLMutex default
   ```

2. Next you need to create an SSL certificate. To do that open a command prompt (DOS-Box) and change to the Apache folder (e.g. `C:\program files\apache group\apache2`) and type the following command:

   ```
   bin\openssl req -config bin\openssl.cnf -new -out my-server.csr
   ```

You will be asked for a passphrase. Please don't use simple words but whole sentences, e.g. a part of a poem. The longer the phrase the better. Also you have to enter the URL of your server. All other questions are optional but we recommend you fill those in too.

Normally the `privkey.pem` file is created automatically, but if it isn't you need to type this command to generate it:

```
bin\openssl genrsa -out conf\privkey.pem 2048
```

Next type the commands

```
bin\openssl rsa -in conf\privkey.pem -out conf\server.key
```

and (on one line)

```
bin\openssl req -new -key conf\server.key -out conf\server.csr \
-config conf\openssl.cnf
```

and then (on one line)

```
bin\openssl x509 -in conf\server.csr -out conf\server.crt
            -req -signkey conf\server.key -days 4000
```

This will create a certificate which will expire in 4000 days. And finally enter (on one line):

```
bin\openssl x509 -in conf\server.cert -out conf\server.der.crt
            -outform DER
```

These commands created some files in the Apache `conf` folder (`server.der.crt`, `server.csr`, `server.key`, `.rnd`, `privkey.pem`, `server.cert`).

3. Restart the Apache service.

4. Point your browser to `https://servername/svn/project` ...

SSL and Internet Explorer

If you're securing your server with SSL and use authentication against a windows domain you will encounter that browsing the repository with the Internet Explorer doesn't work anymore. Don't worry - this is only the Internet Explorer not able to authenticate. Other browsers don't have that problem and TortoiseSVN and any other Subversion client are still able to authenticate.

If you still want to use IE to browse the repository you can either:

- define a separate `<Location /path>` directive in the Apache config file, and add the `SSPIBasicPreferred On`. This will allow IE to authenticate again, but other browsers and Subversion won't be able to authenticate against that location.

- Offer browsing with unencrypted authentication (without SSL) too. Strangely IE doesn't have any problems with authenticating if the connection is not secured with SSL.

- In the SSL "standard" setup there's often the following statement in Apache's virtual SSL host:

```
SetEnvIf User-Agent ".*MSIE.*" \
            nokeepalive ssl-unclean-shutdown \
            downgrade-1.0 force-response-1.0
```

There are (were?) good reasons for this configuration, see *http://www.modssl.org/ docs/2.8/ssl_faq.html#ToC49* [http://www.modssl.org/docs/2.8/ssl_faq.html#ToC49] But if you want NTLM authentication you have to use keepalive. If You uncomment the whole SetEnvIf you should be able to authenticate IE with windows authentication over SSL against the Apache on Win32 with included mod_auth_sspi.

Forcing SSL access

When you've set up SSL to make your repository more secure, you might want to disable the normal access via non-SSL (http) and only allow https access. To do this, you have to add another directive to the Subversion <Location> block: SSLRequireSSL.

An example <Location> block would look like this:

```
<Location /svn>
  DAV svn
  SVNParentPath D:\SVN
  SSLRequireSSL
  AuthType Basic
  AuthName "Subversion repositories"
  AuthUserFile passwd
  #AuthzSVNAccessFile svnaccessfile
  Require valid-user
</Location>
```

3.1.10. Using client certificates with virtual SSL hosts

Sent to the TortoiseSVN mailing list by Nigel Green. Thanks!

In some server configurations you may need to setup a single server containing 2 virtual SSL hosts: The first one for public web access, with no requirement for a client certificate. The second one to be secure with a required client certificate, running a Subversion server.

Adding an SSLVerifyClient Optional directive to the *per-server* section of the Apache configuration (i.e. outside of any VirtualHost and Directory blocks) forces Apache to request a client Certificate in the initial SSL handshake. Due to a bug in mod_ssl it is essential that the certificate is requested at this point as it does not work if the SSL connection is re-negotiated.

The solution is to add the following directive to the virtual host directory that you want to lock down for Subversion:

```
SSLRequire %{SSL_CLIENT_VERIFY} eq "SUCCESS"
```

This directive grants access to the directory only if a client certificate was received and verified successfully.

To summarise, the relevant lines of the Apache configuration are:

```
SSLVerifyClient Optional
```

```
### Virtual host configuration for the PUBLIC host
### (not requiring a certificate)

<VirtualHost 127.0.0.1:443>
  <Directory "pathtopublicfileroot">
  </Directory>
</VirtualHost>

### Virtual host configuration for SUBVERSION
### (requiring a client certificate)
<VirtualHost 127.0.0.1:443>
  <Directory "subversion host root path">
    SSLRequire %{SSL_CLIENT_VERIFY} eq "SUCCESS"
  </Directory>

  <Location /svn>
    DAV svn
    SVNParentPath /pathtorepository
  </Location>
</VirtualHost>
```

3.2. Svnserve Based Server

3.2.1. Introduction

There may be situations where it's not possible to use Apache as your server. Fortunately, Subversion includes Svnserve - a lightweight stand-alone server which uses a custom protocol over an ordinary TCP/IP connection.

In most cases svnserve is easier to setup and runs faster than the Apache based server. And now that SASL support is included it is easy to secure as well.

3.2.2. Installing svnserve

1. Get the latest version of Subversion from *http://subversion.tigris.org/servlets/ProjectDocumentList?folderID=91* [http://subversion.tigris.org/servlets/ProjectDocumentList?folderID=91]. Alternatively get a pre-packaged installer from CollabNet at *http://www.collab.net/downloads/subversion* [http://www.collab.net/downloads/subversion]. This installer will setup svnserve as a Windows service, and also includes some of the tools you need if you are going to use SASL for security.

2. If you already have a version of Subversion installed, and svnserve is running, you will need to stop it before continuing.

3. Run the Subversion installer. If you run the installer on your server (recommended) you can skip step 4.

4. Open the windows-explorer, go to the installation directory of Subversion (usually `C:\Program Files\Subversion`) and in the `bin` directory, find the files `svnserve.exe`, `intl3_svn.dll`, `libapr.dll`, `libapriconv.dll`, `libapriutil.dll`, `libdb*.dll`, `libeay32.dll` and `ssleay32.dll` - copy these files, or just copy all of the `bin` directory, into a directory on your server e.g. `c:\svnserve`

3.2.3. Running svnserve

Now that svnserve is installed, you need it running on your server. The simplest approach is to run the following from a DOS shell or create a windows shortcut:

```
svnserve.exe --daemon
```

svnserve will now start waiting for incoming requests on port 3690. The --daemon switch tells svnserve to run as a daemon process, so it will always exist until it is manually terminated.

If you have not yet created a repository, follow the instructions given with the Apache server setup Section 3.1.4, "Configuration".

To test that svnserve is working, use TortoiseSVN → Repo-Browser to view a repository.

Assuming your repository is located in `c:\repos\TestRepo`, and your server is called `localhost`, enter:

```
svn://localhost/repos/TestRepo
```

when prompted by the repo browser.

You can also increase security and save time entering URLs with svnserve by using the --root switch to set the root location and restrict access to a specified directory on the server:

```
svnserve.exe --daemon --root drive:\path\to\repository\root
```

Using the previous test as a guide, svnserve would now run as:

```
svnserve.exe --daemon --root c:\repos
```

And in TortoiseSVN our repo-browser URL is now shortened to:

```
svn://localhost/TestRepo
```

Note that the --root switch is also needed if your repository is located on a different partition or drive than the location of svnserve on your server.

Svnserve will service any number of repositories. Just locate them somewhere below the root folder you just defined, and access them using a URL relative to that root.

> # Warning
>
> Do not create or access a Berkeley DB repository on a network share. It *cannot* exist on a remote filesystem. Not even if you have the network drive mapped to a drive letter. If you attempt to use Berkeley DB on a network share, the results are unpredictable - you may see mysterious errors right away, or it may be months before you discover that your repository database is subtly corrupted.

3.2.3.1. Run svnserve as a Service

Running svnserve as a user is usually not the best way. It means always having a user logged in on your server, and remembering to restart it after a reboot. A better way is to run svnserve as a windows service. Starting with Subversion 1.4, svnserve can be installed as a native windows service.

To install svnserve as a native windows service, execute the following command all on one line to create a service which is automatically started when windows starts.

```
sc create svnserve binpath= "c:\svnserve\svnserve.exe --service
```

```
--root c:\repos" displayname= "Subversion" depend= tcpip
start= auto
```

If any of the paths include spaces, you have to use (escaped) quotes around the path, like this:

```
sc create svnserve binpath= "
    \"C:\Program Files\Subversion\bin\svnserve.exe\"
    --service --root c:\repos" displayname= "Subversion"
    depend= tcpip start= auto
```

You can also add a description after creating the service. This will show up in the Windows Services Manager.

```
sc description svnserve "Subversion server (svnserve)"
```

Note the rather unusual command line format used by sc. In the key= value pairs there must be no space between the key and the = but there must be a space before the value.

> **Tip**
>
> Microsoft now recommend services to be run as under either the Local Service or Network Service account. Refer to *The Services and Service Accounts Security Planning Guide* [http://www.microsoft.com/technet/security/topics/serversecurity/serviceaccount/default.mspx]. To create the service under the Local Service account, append the following to the example above.
>
> ```
> obj= "NT AUTHORITY\LocalService"
> ```
>
> Note that you would have to give the Local Service account appropriate rights to both Subversion and your repositories, as well as any applications which are used by hook scripts. The built-in group for this is called "LOCAL SERVICE".

Once you have installed the service, you need to go to the services manager to start it (this time only; it will start automatically when the server reboots).

For more detailed information, refer to *Windows Service Support for Svnserve* [http://svn.collab.net/repos/svn/trunk/notes/windows-service.txt].

If you installed an earlier version of svnserve using the SVNService wrapper, and you now want to use the native support instead, you will need to unregister the wrapper as a service (remember to stop the service first!). Simply use the command

```
svnservice -remove
```

to remove the service registry entry.

3.2.4. Basic Authentication with svnserve

The default svnserve setup provides anonymous read-only access. This means that you can use an svn:// URL to checkout and update, or use the repo-browser in TortoiseSVN to view the repository, but you won't be able to commit any changes.

To enable write access to a repository, you need to edit the conf/svnserve.conf file in your repository directory. This file controls the configuration of the svnserve daemon, and also contains useful documentation.

You can enable anonymous write access by simply setting:

```
[general]
anon-access = write
```

However, you will not know who has made changes to a repository, as the `svn:author` property will be empty. You will also be unable to control who makes changes to a repository. This is a somewhat risky setup!

One way to overcome this is to create a password database:

```
[general]
anon-access = none
auth-access = write
password-db = userfile
```

Where `userfile` is a file which exists in the same directory as `svnserve.conf`. This file can live elsewhere in your file system (useful for when you have multiple repositories which require the same access rights) and may be referenced using an absolute path, or a path relative to the `conf` directory. If you include a path, it must be written `/the/unix/way`. Using \ or drive letters will not work. The `userfile` should have a structure of:

```
[users]
username = password
...
```

This example would deny all access for unauthenticated (anonymous) users, and give read-write access to users listed in `userfile`.

Tip

If you maintain multiple repositories using the same password database, the use of an authentication realm will make life easier for users, as TortoiseSVN can cache your credentials so that you only have to enter them once. More information can be found in the Subversion book, specifically in the sections *Create a 'users' file and realm* [http://svnbook.red-bean.com/ en/1.4/svn.serverconfig.svnserve.html#svn.serverconfig.svnserve.auth.users] and *Client Credentials Caching* [http://svnbook.red-bean.com/en/1.4/ svn.serverconfig.netmodel.html#svn.serverconfig.netmodel.credcache]

3.2.5. Better Security with SASL

3.2.5.1. What is SASL?

The Cyrus Simple Authentication and Security Layer is open source software written by Carnegie Mellon University. It adds generic authentication and encryption capabilities to any network protocol, and as of Subversion 1.5 and later, both the svnserve server and TortoiseSVN client know how to make use of this library.

For a more complete discussion of the options available, you should look at the Subversion book in the section *Using svnserve with SASL* [http://svnbook.red-bean.com/nightly/en/ svn.serverconfig.svnserve.html#svn.serverconfig.svnserve.sasl]. If you are just looking for a simple way to set up secure authentication and encryption on a Windows server, so that your repository can be accessed safely over the big bad Internet, read on.

3.2.5.2. SASL Authentication

To activate specific SASL mechanisms on the server, you'll need to do three things. First, create a [sasl] section in your repository's svnserve.conf file, with this key-value pair:

```
use-sasl = true
```

Second, create a file called svn.conf in a convenient location - typically in the directory where subversion is installed.

Thirdly, create two new registry entries to tell SASL where to find things. Create a registry key named [HKEY_LOCAL_MACHINE\SOFTWARE\Carnegie Mellon\Project Cyrus\SASL Library] and place two new string values inside it: SearchPath set to the directory path containing the sasl*.dll plug-ins (normally in the Subversion install directory), and ConfFile set to the directory containing the svn.conf file. If you used the CollabNet installer, these registry keys will already have been created for you.

Edit the svn.conf file to contain the following:

```
pwcheck_method: auxprop
auxprop_plugin: sasldb
mech_list: DIGEST-MD5
sasldb_path: C:\TortoiseSVN\sasldb
```

The last line shows the location of the authentication database, which is a file called sasldb. This could go anywhere, but a convenient choice is the repository parent path. Make sure that the svnserve service has read access to this file.

If svnserve was already running, you will need to restart it to ensure it reads the updated configuration.

Now that everything is set up, all you need to do is create some users and passwords. To do this you need the saslpasswd2 program. If you used the CollabNet installer, that program will be in the install directory. Use a command something like this:

```
saslpasswd2 -c -f C:\TortoiseSVN\sasldb -u realm username
```

The -f switch gives the database location, realm must be the same as the value you defined in your repository's svnserve.conf file, and username is exactly what you expect it to be. Note that the realm is not allowed to contain space characters.

You can list the usernames stored in the database using the sasldblistusers2 program.

3.2.5.3. SASL Encryption

To enable or disable different levels of encryption, you can set two values in your repository's svnserve.conf file:

```
[sasl]
use-sasl = true
min-encryption = 128
max-encryption = 256
```

The min-encryption and max-encryption variables control the level of encryption demanded by the server. To disable encryption completely, set both values to 0. To enable simple checksumming of data (i.e., prevent tampering and guarantee data integrity without encryption), set both values to 1. If you wish to allow (but not require) encryption, set the minimum value to 0, and the maximum value to some bit-length. To require encryption unconditionally, set both values to numbers greater than 1.

In our previous example, we require clients to do at least 128-bit encryption, but no more than 256-bit encryption.

3.2.6. Authentication with svn+ssh

Another way to authenticate users with a svnserve based server is to use a secure shell (SSH) to tunnel requests through. It is not as simple to set up as SASL, but it may be useful is some cases.

With this approach, svnserve is not run as a daemon process, rather, the secure shell starts svnserve for you, running it as the SSH authenticated user. To enable this, you need a secure shell daemon on your server.

A basic method for setting up your server is given in Appendix G, *Securing Svnserve using SSH*. You can find other SSH topics within the FAQ by searching for "SSH".

Further information about svnserve can be found in the *Version Control with Subversion* [http:// svnbook.red-bean.com].

3.2.7. Path-based Authorization with svnserve

Starting with Subversion 1.3, svnserve supports the same `mod_authz_svn` path-based authorization scheme that is available with the Apache server. You need to edit the `conf/svnserve.conf` file in your repository directory and add a line referring to your authorization file.

```
[general]
authz-db = authz
```

Here, `authz` is a file you create to define the access permissions. You can use a separate file for each repository, or you can use the same file for several repositories. Read Section 3.1.6, "Path-Based Authorization" for a description of the file format.

Chapter 4. The Repository

No matter which protocol you use to access your repositories, you always need to create at least one repository. This can either be done with the Subversion command line client or with TortoiseSVN.

If you haven't created a Subversion repository yet, it's time to do that now.

4.1. Repository Creation

You can create a repository with the FSFS backend or with the older Berkeley Database (BDB) format. The FSFS format is generally faster and easier to administer, and it works on network shares and Windows 98 without problems. The BDB format was once considered more stable simply because it has been in use for longer, but since FSFS has now been in use in the field for several years, that argument is now rather weak. Read *Choosing a Data Store* [http://svnbook.red-bean.com/en/1.4/svn.reposadmin.planning.html#svn.reposadmin.basics.backends] in the Subversion book for more information.

4.1.1. Creating a Repository with the Command Line Client

1. Create an empty folder with the name SVN (e.g. D:\SVN\), which is used as root for all your repositories.

2. Create another folder MyNewRepository inside D:\SVN\.

3. Open the command prompt (or DOS-Box), change into D:\SVN\ and type

   ```
   svnadmin create --fs-type bdb MyNewRepository
   ```

   or

   ```
   svnadmin create --fs-type fsfs MyNewRepository
   ```

Now you've got a new repository located at D:\SVN\MyNewRepository.

4.1.2. Creating The Repository With TortoiseSVN
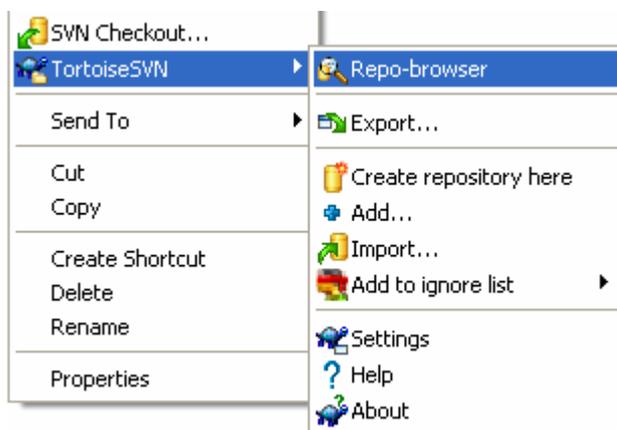
Figure 4.1. The TortoiseSVN menu for unversioned folders

1. Open the windows explorer

2. Create a new folder and name it e.g. `SVNRepository`

3. Right-click on the newly created folder and select TortoiseSVN → Create Repository here....

A repository is then created inside the new folder. *Don't edit those files yourself!!!*. If you get any errors make sure that the folder is empty and not write protected.

> **Tip**
>
> TortoiseSVN no longer offers the option to create BDB repositories, although you can still use the command line client to create them. FSFS repositories are generally easier for you to maintain, and also makes it easier for us to maintain TortoiseSVN due to compatibility issues between the different BDB versions.
>
> Future versions of TortoiseSVN will not support `file://` access to BDB repositories due to these compatibility issues, although it will of course always support this repository format when accessed via a server through the `svn://`, `http://` or `https://` protocols. For this reason, we strongly recommend that any new repository which must be accessed using `file://` protocol is created as FSFS.
>
> Of course we also recommend that you don't use `file://` access at all, apart from local testing purposes. Using a server is more secure and more reliable for all but single-developer use.

4.1.3. Local Access to the Repository

To access your local repository you need the path to that folder. Just remember that Subversion expects all repository paths in the form `file:///C:/SVNRepository/`. Note the use of forward slashes throughout.

To access a repository located on a network share you can either use drive mapping, or you can use the UNC path. For UNC paths, the form is `file://ServerName/path/to/repos/`. Note that there are only 2 leading slashes here.

Prior to SVN 1.2, UNC paths had to be given in the more obscure form `file:///\ServerName/path/to/repos`. This form is still supported, but not recommended.

> **Warning**
>
> Do not create or access a Berkeley DB repository on a network share. It *cannot* exist on a remote file system. Not even if you have the network drive mapped to a drive letter. If you attempt to use Berkeley DB on a network share, the results are unpredictable - you may see mysterious errors right away, or it may be months before you discover that your repository database is subtly corrupted.

4.1.4. Accessing a Repository on a Network Share

Although in theory it is possible to put a FSFS repository on a network share and have multiple users access it using `file://` protocol, this is most definitely *not* recommended. In fact we would *strongly* discourage it, and do not support such use.

Firstly you are giving every user direct write access to the repository, so any user could accidentally delete the entire repository or make it unusable in some other way.

Secondly not all network file sharing protocols support the locking that Subversion requires, so you may find your repository gets corrupted. It may not happen straight away, but one day two users will try to access the repository at the same time.

Thirdly the file permissions have to be set just so. You may just about get away with it on a native Windows share, but SAMBA is particularly difficult.

`file://` access is intended for local, single-user access only, particularly testing and debugging. When you want to share the repository you *really* need to set up a proper server, and it is not nearly as difficult as you might think. Read Chapter 3, *Setting Up A Server* for guidelines on choosing and setting up a server.

4.1.5. Repository Layout

Before you import your data into the repository you should first think about how you want to organize your data. If you use one of the recommended layouts you will later have it much easier.

There are some standard, recommended ways to organize a repository. Most people create a `trunk` directory to hold the "main line" of development, a `branches` directory to contain branch copies, and a `tags` directory to contain tag copies. If a repository holds only one project, then often people create these top-level directories:

```
/trunk
/branches
/tags
```

If a repository contains multiple projects, people often index their layout by branch:

```
/trunk/paint
/trunk/calc
/branches/paint
/branches/calc
/tags/paint
/tags/calc
```

...or by project:

```
/paint/trunk
/paint/branches
/paint/tags
/calc/trunk
/calc/branches
/calc/tags
```

Indexing by project makes sense if the projects are not closely related and each one is checked out individually. For related projects where you may want to check out all projects in one go, or where the projects are all tied together in a single distribution package, it is often better to index by branch. This way you have only one trunk to checkout, and the relationships between the sub-projects is more easily visible.

If you adopt a top level `/trunk /tags /branches` approach, there is nothing to say that you have to copy the entire trunk for every branch and tag, and in some ways this structure offers the most flexibility.

For unrelated projects you may prefer to use separate repositories. When you commit changes, it is the revision number of the whole repository which changes, not the revision number of the project. Having 2 unrelated projects share a repository can mean large gaps in the revision numbers. The Subversion and TortoiseSVN projects appear at the same host address, but are completely separate repositories allowing independent development, and no confusion over build numbers.

Of course, you're free to ignore these common layouts. You can create any sort of variation, whatever works best for you or your team. Remember that whatever you choose, it's not a permanent commitment. You can reorganize your repository at any time. Because branches and tags are ordinary directories, TortoiseSVN can move or rename them however you wish.

Switching from one layout to another is just a matter of issuing a series of server-side moves; If you don't like the way things are organized in the repository, just juggle the directories around.

So if you haven't already created a basic folder structure inside your repository you should do that now. There are two ways to achieve this. If you simply want to create a `/trunk /tags /branches` structure, you can use the repository browser to create the three folders (in three separate commits). If you want to create a deeper hierarchy then it is simpler to create a folder structure on disk first and import it in a single commit, like this:

1. create a new empty folder on your hard drive

2. create your desired top-level folder structure inside that folder - don't put any files in it yet!

3. import this structure into the repository via a right click on the folder and selecting TortoiseSVN → Import... This will import your temp folder into the repository root to create the basic repository layout.

Note that the name of the folder you are importing does not appear in the repository, only its contents. For example, create the following folder structure:

```
C:\Temp\New\trunk
C:\Temp\New\branches
C:\Temp\New\tags
```

Import `C:\Temp\New` into the repository root, which will then look like this:

```
/trunk
/branches
/tags
```

4.2. Repository Backup

Whichever type of repository you use, it is vitally important that you maintain regular backups, and that you verify the backup. If the server fails, you may be able to access a recent version of your files, but without the repository all your history is lost forever.

The simplest (but not recommended) way is just to copy the repository folder onto the backup medium. However, you have to be absolutely sure that no process is accessing the data. In this context, access means *any* access at all. A BDB repository is written to even when the operation only appears to require reading, such as getting status. If your repository is accessed at all during the copy, (web browser left open, WebSVN, etc.) the backup will be worthless.

The recommended method is to run

```
svnadmin hotcopy path/to/repository path/to/backup --clean-logs
```

to create a copy of your repository in a safe manner. Then backup the copy. The `--clean-logs` option is not required, but removes any redundant log files when you backup a BDB repository, which may save some space.

The `svnadmin` tool is installed automatically when you install the Subversion command line client. If you are installing the command line tools on a Windows PC, the best way is to download the

Windows installer version. It is compressed more efficiently than the `.zip` version, so the download is smaller, and it takes care of setting the paths for you. You can download the latest version of the Subversion command line client from *http://subversion.tigris.org/servlets/ProjectDocumentList? folderID=91* [http://subversion.tigris.org/servlets/ProjectDocumentList?folderID=91].

4.3. Hook Scripts

A hook script is a program triggered by some repository event, such as the creation of a new revision or the modification of an unversioned property. Each hook is handed enough information to tell what that event is, what target(s) it's operating on, and the username of the person who triggered the event. Depending on the hook's output or return status, the hook program may continue the action, stop it, or suspend it in some way. Please refer to the chapter on *Hook Scripts* [http://svnbook.red-bean.com/en/1.4/ svn.reposadmin.create.html#svn.reposadmin.create.hooks] in the Subversion Book for full details about the hooks which are implemented.

These hook scripts are executed by the server that hosts the repository. TortoiseSVN also allows you to configure client side hook scripts that are executed locally upon certain events. See Section 5.29.7, "Client Side Hook Scripts" for more information.

Sample hook scripts can be found in the `hooks` directory of the repository. These sample scripts are suitable for Unix/Linux servers but need to be modified if your server is Windows based. The hook can be a batch file or an executable. The sample below shows a batch file which might be used to implement a pre-revprop-change hook.

```
rem Only allow log messages to be changed.
if "%4" == "svn:log" exit 0
echo Property '%4' cannot be changed >&2
exit 1
```

Note that anything sent to stdout is discarded. if you want a message to appear in the Commit Reject dialog you must send it to stderr. In a batch file this is achieved using `>&2`

4.4. Checkout Links

If you want to make your Subversion repository available to others you may want to include a link to it from your website. One way to make this more accessible is to include a *checkout link* for other TortoiseSVN users.

When you install TortoiseSVN, it registers a new `tsvn:` protocol. When a TortoiseSVN user clicks on such a link, the checkout dialog will open automatically with the repository URL already filled in.

To include such a link in your own html page, you need to add code which looks something like this:

```
<a href="tsvn:https://tortoisesvn.tigris.org/svn/tortoisesvn/trunk">
</a>
```

Of course it would look even better if you included a suitable picture. You can use the *TortoiseSVN logo* [http://tortoisesvn.tigris.org/images/TortoiseCheckout.png] or you can provide your own image.

```
<a href="tsvn:https://tortoisesvn.tigris.org/svn/tortoisesvn/trunk">
<img src=TortoiseCheckout.png></a>
```

Chapter 5. Daily Use Guide

This document describes day to day usage of the TortoiseSVN client. It is *not* an introduction to version control systems, and *not* an introduction to Subversion (SVN). It is more like a place you may turn to when you know approximately what you want to do, but don't quite remember how to do it.

If you need an introduction to version control with Subversion, then we recommend you read the fantastic book: *Version Control with Subversion* [http://svnbook.red-bean.com/].

This document is also a work in progress, just as TortoiseSVN and Subversion are. If you find any mistakes, please report them to the mailing list so we can update the documentation. Some of the screenshots in the Daily Use Guide (DUG) might not reflect the current state of the software. Please forgive us. We're working on TortoiseSVN in our free time.

In order to get the most out of the Daily Use Guide:

• You should have installed TortoiseSVN already.

• You should be familiar with version control systems.

• You should know the basics of Subversion.

• You should have set up a server and/or have access to a Subversion repository.

5.1. Getting Started

5.1.1. Icon Overlays
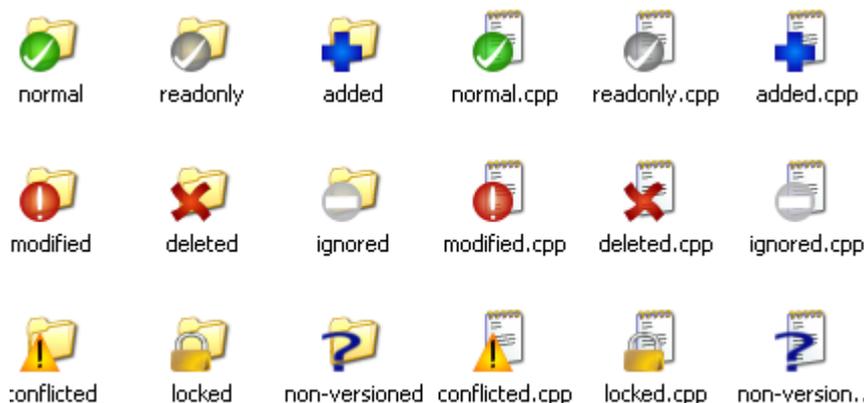
Figure 5.1. Explorer showing icon overlays

One of the most visible features of TortoiseSVN is the icon overlays which appear on files in your working copy. These show you at a glance which of your files have been modified. Refer to Section 5.7.1, "Icon Overlays" to find out what the different overlays represent.

5.1.2. Context Menus
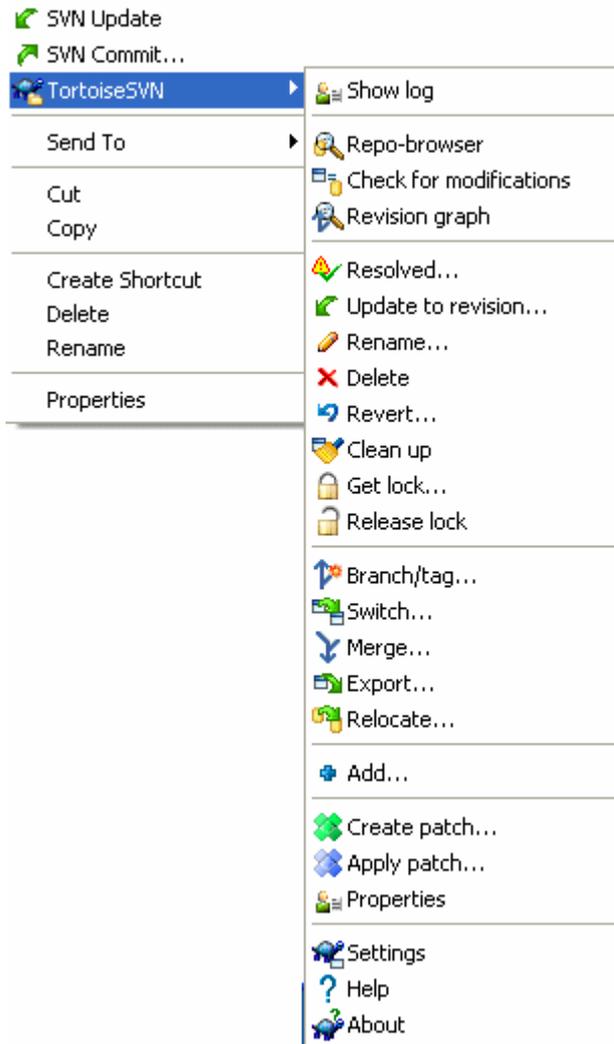
Figure 5.2. Context menu for a directory under version control

All TortoiseSVN commands are invoked from the context menu of the windows explorer. Most are directly visible, when you right click on a file or folder. The commands that are available depend on whether the file or folder or its parent folder is under version control or not. You can also see the TortoiseSVN menu as part of the Explorer file menu.

> # Tip
>
> Some commands which are very rarely used are only available in the extended context menu. To bring up the extended context menu, hold down the **Shift** key when you right-click.

In some cases you may see several TortoiseSVN entries. This is not a bug!

Figure 5.3. Explorer file menu for a shortcut in a versioned folder

This example is for an unversioned shortcut within a versioned folder, and in the Explorer file menu there are *three* entries for TortoiseSVN. One is for the folder, one for the shortcut itself, and the third for the object the shortcut is pointing to. To help you distinguish between them, the icons have an indicator in the lower right corner to show whether the menu entry is for a file, a folder, a shortcut or for multiple selected items.

If you are using Windows 2000 you will find that the context menus are shown as plain text, without the menu icons shown above. We are aware that this was working in previous versions, but Microsoft has changed the way its icon handlers work for Vista, requiring us to use a different display method which unfortunately does not work on Windows 2000.

5.1.3. Drag and Drop

Figure 5.4. Right drag menu for a directory under version control

Other commands are available as drag handlers, when you right drag files or folders to a new location inside working copies or when you right drag a non-versioned file or folder into a directory which is under version control.

5.1.4. Common Shortcuts

Some common operations have well-known Windows shortcuts, but do not appear on buttons or in menus. If you can't work out how to do something obvious, like refreshing a view, check here.

F1

Help, of course.

F5

Refresh the current view. This is perhaps the single most useful one-key command. For example ... In Explorer this will refresh the icon overlays on your working copy. In the commit dialog it will re-scan the working copy to see what may need to be committed. In the Revision Log dialog it will contact the repository again to check for more recent changes.

Ctrl-A

Select all. This can be used if you get an error message and want to copy and paste into an email. Use Ctrl-A to select the error message and then ...

Ctrl-C

... Copy the selected text.

5.1.5. Authentication

If the repository that you are trying to access is password protected, an authentication Dialog will show up.
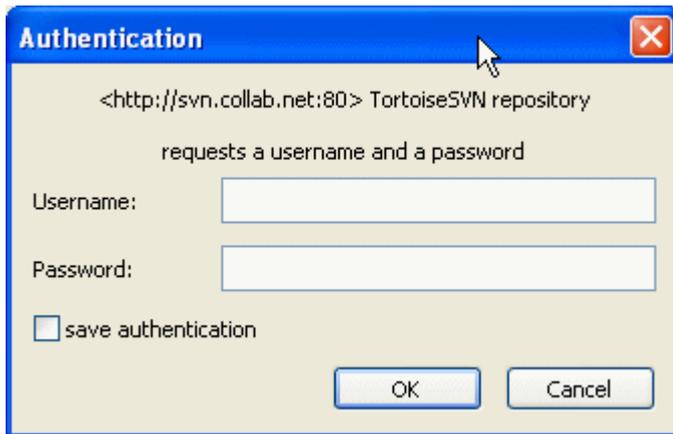
Figure 5.5. Authentication Dialog

Enter your username and password. The checkbox will make TortoiseSVN store the credentials in Subversion's default directory: `%APPDATA%\Subversion\auth` in three subdirectories:

- `svn.simple` contains credentials for basic authentication (username/password).

- `svn.ssl.server` contains SSL server certificates.

- `svn.username` contains credentials for username-only authentication (no password needed).

If you want to clear the authentication cache for all servers, you can do so from the Saved Data page of TortoiseSVN's settings dialog. That button will clear all cached authentication data from the Subversion `auth` directories, as well as any authentication data stored in the registry by earlier versions of TortoiseSVN. Refer to Section 5.29.5, "Saved Data Settings".

For more information on how to set up your server for authentication and access control, refer to Chapter 3, *Setting Up A Server*

5.1.6. Maximizing Windows

Many of TortoiseSVN's dialogs have a lot of information to display, but it is often useful to maximize only the height, or only the width, rather than maximizing to fill the screen. As a convenience, there are shortcuts for this on the Maximize button. Use the middle mouse button to maximize vertically, and right mouse to maximize horizontally.

5.2. Importing Data Into A Repository

5.2.1. Import

If you are importing into an existing repository which already contains some projects, then the repository structure will already have been decided. If are importing data into a new repository then it is worth taking the time to think about how it will be organised. Read Section 4.1.5, "Repository Layout" for further advice.

This section describes the Subversion import command, which was designed for importing a directory hierarchy into the repository in one shot. Although it does the job, it has several shortcomings:

- There is no way to select files and folders to include, aside from using the global ignore settings.

- The folder imported does not become a working copy. You have to do a checkout to copy the files back from the server.

- It is easy to import to the wrong folder level in the repository.

For these reasons we recommend that you do not use the import command at all but rather follow the two-step method described in Section 5.2.2, "Import in Place". But since you are here, this is how the basic import works ...

Before you import your project into a repository you should:

1. Remove all files which are not needed to build the project (temporary files, files which are generated by a compiler e.g. *.obj, compiled binaries, ...)

2. Organize the files in folders and sub-folders. Although it is possible to rename/move files later it is highly recommended to get your project's structure straight before importing!

Now select the top-level folder of your project directory structure in the windows explorer and right click to open the context menu. Select the command TortoiseSVN → Import... which brings up a dialog box:
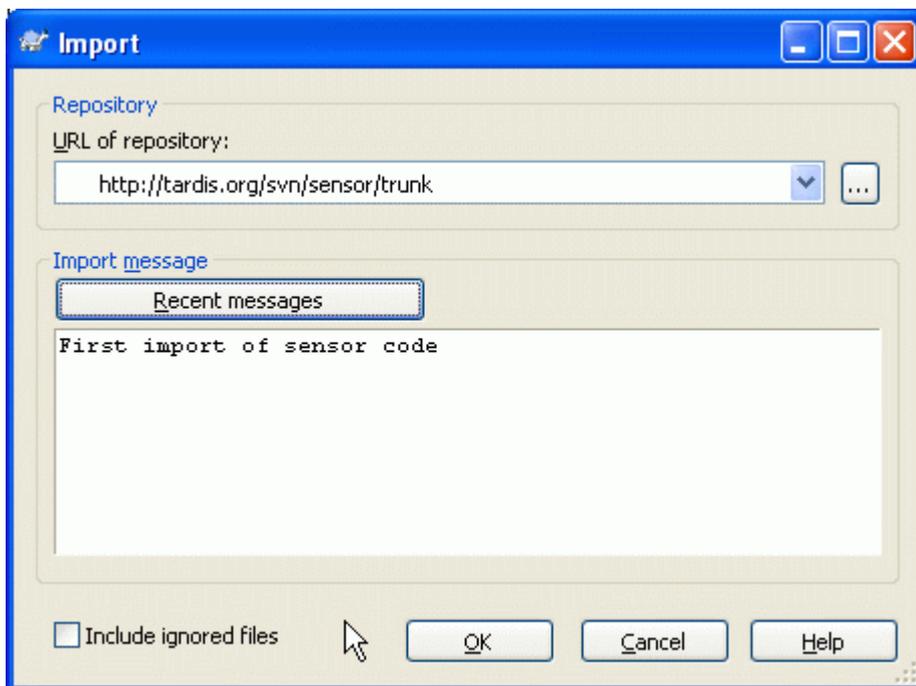
Figure 5.6. The Import dialog

In this dialog you have to enter the URL of the repository location where you want to import your project. It is very important to realise that the local folder you are importing does not itself appear in the repository, only its content. For example if you have a structure:

```
C:\Projects\Widget\source
C:\Projects\Widget\doc
C:\Projects\Widget\images
```

and you import `C:\Projects\Widget` into `http://mydomain.com/svn/trunk` then you may be surprised to find that your subdirectories go straight into `trunk` rather than being in a `Widget` subdirectory. You need to specify the subdirectory as part of the URL, `http://mydomain.com/`

`svn/trunk/Widget-X`. Note that the import command will automatically create subdirectories within the repository if they do not exist.

The import message is used as a log message.

By default, files and folders which match the global-ignore patterns are *not* imported. To override this behaviour you can use the Include ignored files checkbox. Refer to Section 5.29.1, "General Settings" for more information on setting a global ignore pattern.

As soon as you press OK TortoiseSVN imports the complete directory tree including all files into the repository. The project is now stored in the repository under version control. Please note that the folder you imported is *NOT* under version control! To get a version-controlled *working copy* you need to do a Checkout of the version you just imported. Or read on to find out how to import a folder in place.

5.2.2. Import in Place

Assuming you already have a repository, and you want to add a new folder structure to it, just follow these steps:

1. Use the repository browser to create a new project folder directly in the repository.

2. Checkout the new folder over the top of the folder you want to import. You will get a warning that the local folder is not empty. Now you have a versioned top level folder with unversioned content.

3. Use TortoiseSVN → Add... on this versioned folder to add some or all of the content. You can add and remove files, set `svn:ignore` properties on folders and make any other changes you need to.

4. Commit the top level folder, and you have a new versioned tree, and a local working copy, created from your existing folder.

5.2.3. Special Files

Sometimes you need to have a file under version control which contains user specific data. That means you have a file which every developer/user needs to modify to suit his/her local setup. But versioning such a file is difficult because every user would commit his/her changes every time to the repository.

In such cases we suggest to use *template* files. You create a file which contains all the data your developers will need, add that file to version control and let the developers check this file out. Then, each developer has to *make a copy* of that file and rename that copy. After that, modifying the copy is not a problem anymore.

As an example, you can have a look at TortoiseSVN's build script. It calls a file named `TortoiseVars.bat` which doesn't exist in the repository. Only the file `TortoiseVars.tmpl`. `TortoiseVars.tmpl` is the template file which every developer has to create a copy from and rename that file to `TortoiseVars.bat`. Inside that file, we added comments so that the users will see which lines they have to edit and change according to their local setup to get it working.

So as not to disturb the users, we also added the file `TortoiseVars.bat` to the ignore list of its parent folder, i.e. we've set the Subversion property `svn:ignore` to include that filename. That way it won't show up as unversioned on every commit.

5.2.4. Referenced Projects

Sometimes it is useful to construct a working copy that is made out of a number of different checkouts. For example, you may want different subdirectories to come from different locations in a repository, or

perhaps from different repositories altogether. If you want every user to have the same layout, you can define the `svn:externals` properties.

Let's say you check out a working copy of `/project1` to `D:\dev\project1`. Select the folder `D:\dev\project1`, right click and choose Windows Menu → Properties from the context menu. The Properties Dialog comes up. Then go to the Subversion tab. There, you can set properties. Click Add.... Select the `svn:externals` property from the combobox and write in the edit box the repository URL in the format `name url` or if you want to specify a particular revision, `name -rREV url` You can add multiple external projects, 1 per line. Note that URLs must be properly escaped or they will not work. For example you must replace each space with `%20`. Note that it is not possible to use folder names with spaces in them. Suppose that you have set these properties on `D:\dev\project1`:

```
sounds   http://sounds.red-bean.com/repos
quick_graphs  http://graphics.red-bean.com/repos/fast%20graphics
skins/toolkit -r21 http://svn.red-bean.com/repos/skin-maker
```

Now click Set and commit your changes. When you (or any other user) update your working copy, Subversion will create a sub-folder `D:\dev\project1\sounds` and checkout the sounds project, another sub-folder `D:\dev\project1\quick graphs` containing the graphics project, and finally a nested sub-folder `D:\dev\project1\skins\toolkit` containing revision 21 of the skin-maker project.

Tip

You should strongly consider using explicit revision numbers in all of your externals definitions, as described above. Doing so means that you get to decide when to pull down a different snapshot of external information, and exactly which snapshot to pull. Besides the common sense aspect of not being surprised by changes to third-party repositories that you might not have any control over, using explicit revision numbers also means that as you backdate your working copy to a previous revision, your externals definitions will also revert to the way they looked in that previous revision, which in turn means that the external working copies will be updated to match they way *they* looked back when your repository was at that previous revision. For software projects, this could be the difference between a successful and a failed build of an older snapshot of your complex code base.

If the external project is in the same repository, any changes you make there there will be included in the commit list when you commit your main project.

If the external project is in a different repository, any changes you make to the external project will be notified when you commit the main project, but you have to commit those external changes separately.

If you use absolute URLs in `svn:externals` definitions and you have to relocate your working copy (i.e., if the URL of your repository changes), then your externals won't change and might not work anymore.

To avoid such problems, Subversion clients version 1.5 and higher support relative external URLs. Four different methods of specifying a relative URL are supported. In the following examples, assume we have two repositories: one at `http://example.com/svn/repos-1` and another at `http://example.com/svn/repos-2`. We have a checkout of `http://example.com/svn/repos-1/project/trunk` into `C:\Working` and the `svn:externals` property is set on trunk.

Relative to parent directory
    These URLs always begin with the string `../` for example:

```
../../widgets/foo  common/foo-widget
```

This will extract `http://example.com/svn/repos-1/widgets/foo` into `C:\Working\common\foo-widget`.

Note that the URL is relative to the URL of the directory with the `svn:externals` property, not to the directory where the external is written to disk.

Relative to repository root
These URLs always begin with the string `^/` for example:

```
^/widgets/foo  common/foo-widget
```

This will extract `http://example.com/svn/repos-1/widgets/foo` into `C:\Working\common\foo-widget`.

You can easily refer to other repositories with the same `SVNParentPath` (a common directory holding several repositories). For example:

```
^/../repos-2/hammers/claw  common/claw-hammer
```

This will extract `http://example.com/svn/repos-2/hammers/claw` into `C:\Working\common\claw-hammer`.

Relative to scheme
URLs beginning with the string `//` copy only the scheme part of the URL. This is useful when the same hostname must the accessed with different schemes depending upon network location; e.g. clients in the intranet use `http://` while external clients use `svn+ssh://`. For example:

```
//example.com/svn/repos-1/widgets/foo  common/foo-widget
```

This will extract `http://example.com/svn/repos-1/widgets/foo` or `svn+ssh://example.com/svn/repos-1/widgets/foo` depending on which method was used to checkout `C:\Working`.

Relative to the server's hostname
URLs beginning with the string `/` copy the scheme and the hostname part of the URL, for example:

```
/svn/repos-1/widgets/foo  common/foo-widget
```

This will extract `http://example.com/svn/repos-1/widgets/foo` into `C:\Working\common\foo-widget`. But if you checkout your working copy from another server at `svn+ssh://another.mirror.net/svn/repos-1/project1/trunk` then the external reference will extract `svn+ssh://another.mirror.net/svn/repos-1/widgets/foo`.

If you need more information how TortoiseSVN handles Properties read Section 5.17, "Project Settings".

To find out about different methods of accessing common sub-projects read Section B.6, "Include a common sub-project".

5.3. Checking Out A Working Copy

To obtain a working copy you need to do a *checkout* from a repository.

Select a directory in windows explorer where you want to place your working copy. Right click to pop up the context menu and select the command TortoiseSVN → Checkout..., which brings up the following dialog box:
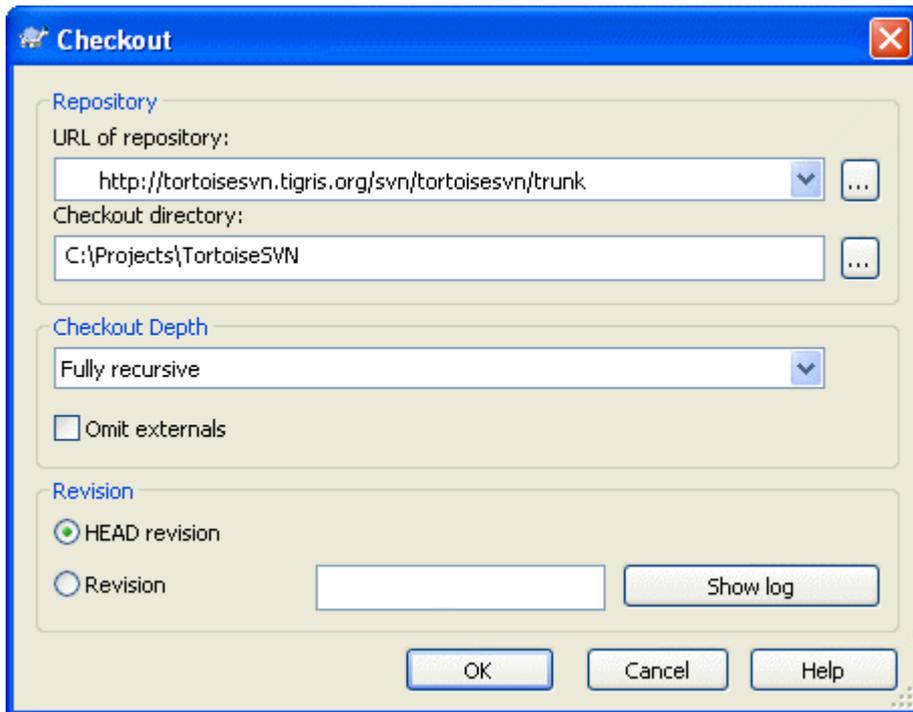
Figure 5.7. The Checkout dialog

If you enter a folder name that does not yet exist, then a directory with that name is created.

5.3.1. Checkout Depth

You can choose the *depth* you want to checkout, which allows you to specify the depth of recursion into child folders. If you want just a few sections of a large tree, You can checkout the top level folder only, then update selected folders recursively.

Fully recursive
>   Checkout the entire tree, including all child folders and sub-folders.

Immediate children, including folders
>   Checkout the specified directory, including all files and child folders, but do not populate the child folders.

Only file children
>   Checkout the specified directory, including all files but do not checkout any child folders.

Only this item
>   Checkout the directory only. Do not populate it with files or child folders.

Working copy
>   Retain the depth specified in the working copy. This option is not used in the checkout dialog, but it is the default in all other dialogs which have a depth setting.

If you check out a sparse working copy (i.e., by choosing something other than `fully recursive` for the checkout depth), you can fetch additional sub-folders by using the repository browser (Section 5.23, "The Repository Browser") or the check for modifications dialog (Section 5.7.3, "Local and Remote Status").

In the repository browser, Right click on the checked out folder, then use TortoiseSVN → Repo-Browser to bring up the repository browser. Find the sub-folder you would like to add to your working copy, then use Context menu → Update item to revision... That menu will only be visible if the selected item does not exist yet in your working copy, but the parent item does exist.

In the check for modifications dialog, first click on the button Check repository. The dialog will show all the files and folders which are in the repository but which you have not checked out as `remotely added`. Right click on the folder(s) you would like to add to your working copy, then use Context menu → Update.

This feature is very useful when you only want to checkout parts of a large tree, but you want the convenience of updating a single working copy. Suppose you have a large tree which has subfolders `Project01` to `Project99`, and you only want to checkout `Project03`, `Project25` and `Project76/SubProj`. Use these steps:

1. Checkout the parent folder with depth "Only this item" You now have an empty top level folder.

2. Select the new folder and use TortoiseSVN → Repo browser to display the repository content.

3. Right click on `Project03` and Context menu → Update item to revision.... Keep the default settings and click on OK. You now have that folder fully populated.

   Repeat the same process for `Project25`.

4. Navigate to `Project76/SubProj` and do the same. This time note that the `Project76` folder has no content except for `SubProj`, which itself is fully populated. Subversion has created the intermediate folders for you without populating them.

Working copy depth cannot be reduced

Once you have checked out a working copy to a particular depth you can increase that depth later to get more content. However you cannot reduce the depth again to remove content. That feature may be added in a later version of Subversion.

Using an older server

Pre-1.5 servers do not understand the working copy depth request, so they cannot always deal with requests efficiently. The command will still work, but an older server may send all the data, leaving the client to filter out what is not required, which may mean a lot of network traffic. If possible you should upgrade your server to 1.5.

If the project contains references to external projects which you do *not* want checked out at the same time, use the Omit externals checkbox.

Important

If Omit externals is checked, or if you wish to increase the depth value, you will have to perform updates to your working copy using TortoiseSVN → Update to Revision... instead of TortoiseSVN → Update. The standard update will include all externals and keep the existing depth.

It is recommended that you check out only the `trunk` part of the directory tree, or lower. If you specify the parent path of the directory tree in the URL then you might end up with a full hard disk since you will get a copy of the entire repository tree including every branch and tag of your project!

Exporting

Sometimes you may want to create a local copy without any of those `.svn` directories, e.g. to create a zipped tarball of your source. Read Section 5.25, "Exporting a Subversion Working Copy" to find out how to do that.

5.4. Committing Your Changes To The Repository

Sending the changes you made to your working copy is known as *committing* the changes. But before you commit you have to make sure that your working copy is up to date. You can either use TortoiseSVN → Update directly. Or you can use TortoiseSVN → Check for Modifications first, to see which files have changed locally or on the server.

5.4.1. The Commit Dialog

If your working copy is up to date and there are no conflicts, you are ready to commit your changes. Select any file and/or folders you want to commit, then TortoiseSVN → Commit....
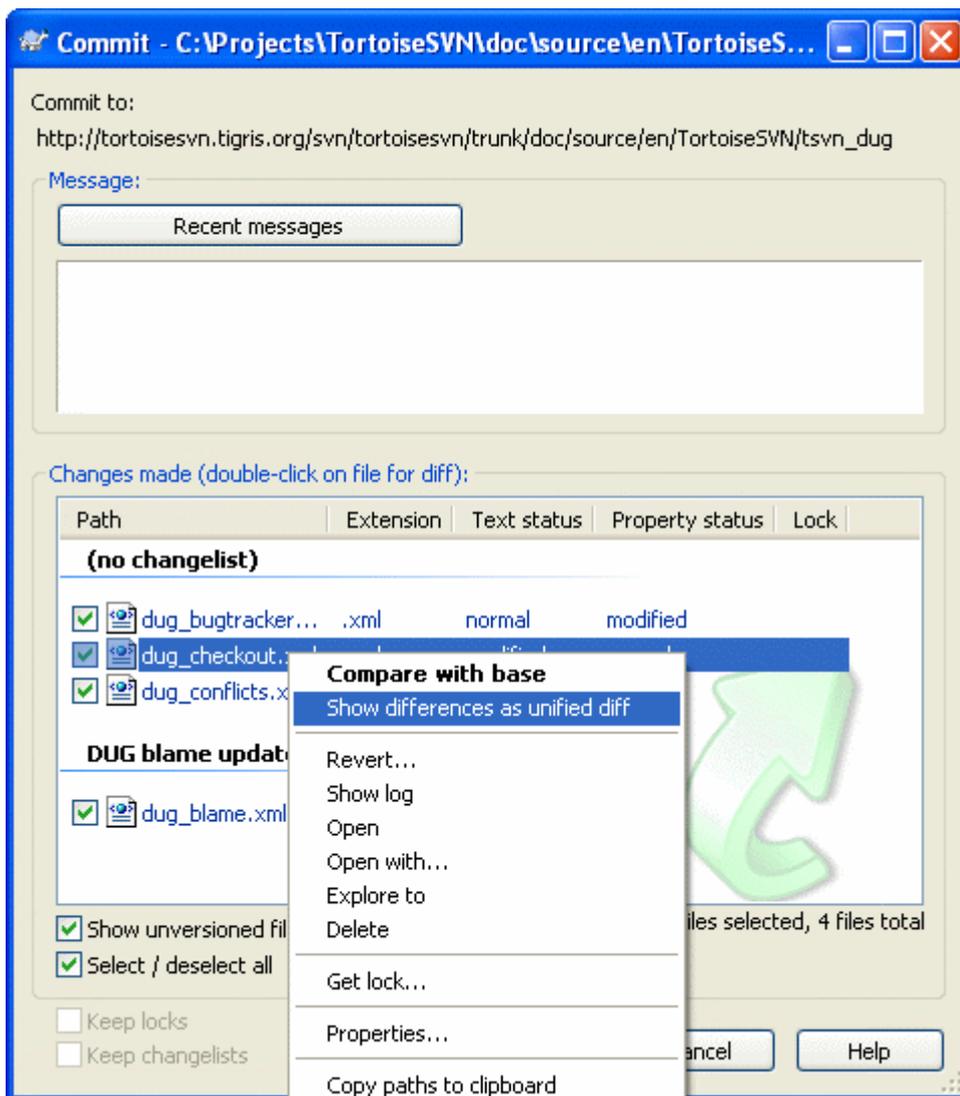
Figure 5.8. The Commit dialog

The commit dialog will show you every changed file, including added, deleted and unversioned files. If you don't want a changed file to be committed, just uncheck that file. If you want to include an unversioned file, just check that file to add it to the commit.

Items which have been switched to a different repository path are also indicated using an (s) marker. You may have switched something while working on a branch and forgotten to switch back to trunk. This is your warning sign!

Commit files or folders?

When you commit files, the commit dialog shows only the files you have selected. When you commit a folder the commit dialog will select the changed files automatically. If you forget about a new file you created, committing the folder will find it anyway. Committing a folder does *not* mean that every file gets marked as changed; It just makes your life easier by doing more work for you.

If you have modified files which have been included from a different repository using `svn:externals`, those changes cannot be included in the same atomic commit. A warning symbol below the file list tells you if this has happened, and the tooltip explains that those external files have to be committed separately.

Many unversioned files in the commit dialog

If you think that the commit dialog shows you too many unversioned (e.g. compiler generated or editor backup) files, there are several ways to handle this. You can:

- add the file (or a wildcard extension) to the list of files to exclude on the settings page. This will affect every working copy you have.

- add the file to the `svn:ignore` list using TortoiseSVN → Add to ignore list This will only affect the directory on which you set the `svn:ignore` property. Using the SVN Property Dialog, you can alter the `svn:ignore` property for a directory.

Read Section 5.13, "Ignoring Files And Directories" for more information.

Double clicking on any modified file in the commit dialog will launch the external diff tool to show your changes. The context menu will give you more options, as shown in the screenshot. You can also drag files from here into another application such as a text editor or an IDE.

You can select or deselect items by clicking on the checkbox to the left of the item. For directories you can use **Shift**-select to make the action recursive.

The columns displayed in the bottom pane are customizable. If you right click on any column header you will see a context menu allowing you to select which columns are displayed. You can also change column width by using the drag handle which appears when you move the mouse over a column boundary. These customizations are preserved, so you will see the same headings next time.

By default when you commit changes, any locks that you hold on files are released automatically after the commit succeeds. If you want to keep those locks, make sure the Keep locks checkbox is checked. The default state of this checkbox is taken from the `no_unlock` option in the Subversion configuration file. Read Section 5.29.1, "General Settings" for information on how to edit the Subversion configuration file.

Drag and Drop

You can drag files into the commit dialog from elsewhere, so long as the working copies are checked out from the same repository. For example, you may have a huge working copy with several explorer windows open to look at distant folders of the hierarchy. If you want to avoid committing from the top level folder (with a lengthy folder crawl to check for changes) you can open the commit dialog for one folder and drag in items from the other windows to include within the same atomic commit.

You can drag unversioned files which reside within a working copy into the commit dialog, and they will be SVN added automatically.

> ### Repairing External Renames
>
> Sometimes files get renamed outside of Subversion, and they show up in the file list as a missing file and an unversioned file. To avoid losing the history you need to notify Subversion about the connection. Simply select both the old name (missing) and the new name (unversioned) and use Context Menu → Repair Move to pair the two files as a rename.

5.4.2. Change Lists

The commit dialog supports Subversion's changelist feature to help with grouping related files together. Find out about this feature in Section 5.8, "Change Lists".

5.4.3. Excluding Items from the Commit List

Sometimes you have versioned files that change frequently but that you really don't want to commit. Sometimes this indicates a flaw in your build process - why are those files versioned? should you be using template files? But occasionally it is inevitable. A classic reason is that your IDE changes a timestamp in the project file every time you build. The project file has to be versioned as it includes all the build settings, but it doesn't need to be committed just because the timestamp changed.

To help out in awkward cases like this, we have reserved a changelist called `ignore-on-commit`. Any file added to this changelist will automatically be unchecked in the commit dialog. You can still commit changes, but you have to select it manually in the commit dialog.

5.4.4. Commit Log Messages

Be sure to enter a log message which describes the changes you are committing. This will help you to see what happened and when, as you browse through the project log messages at a later date. The message can be as long or as brief as you like; many projects have guidelines for what should be included, the language to use, and sometimes even a strict format.

You can apply simple formatting to your log messages using a convention similar to that used within emails. To apply styling to `text`, use `*text*` for bold, `_text_` for underlining, and `^text^` for italics.
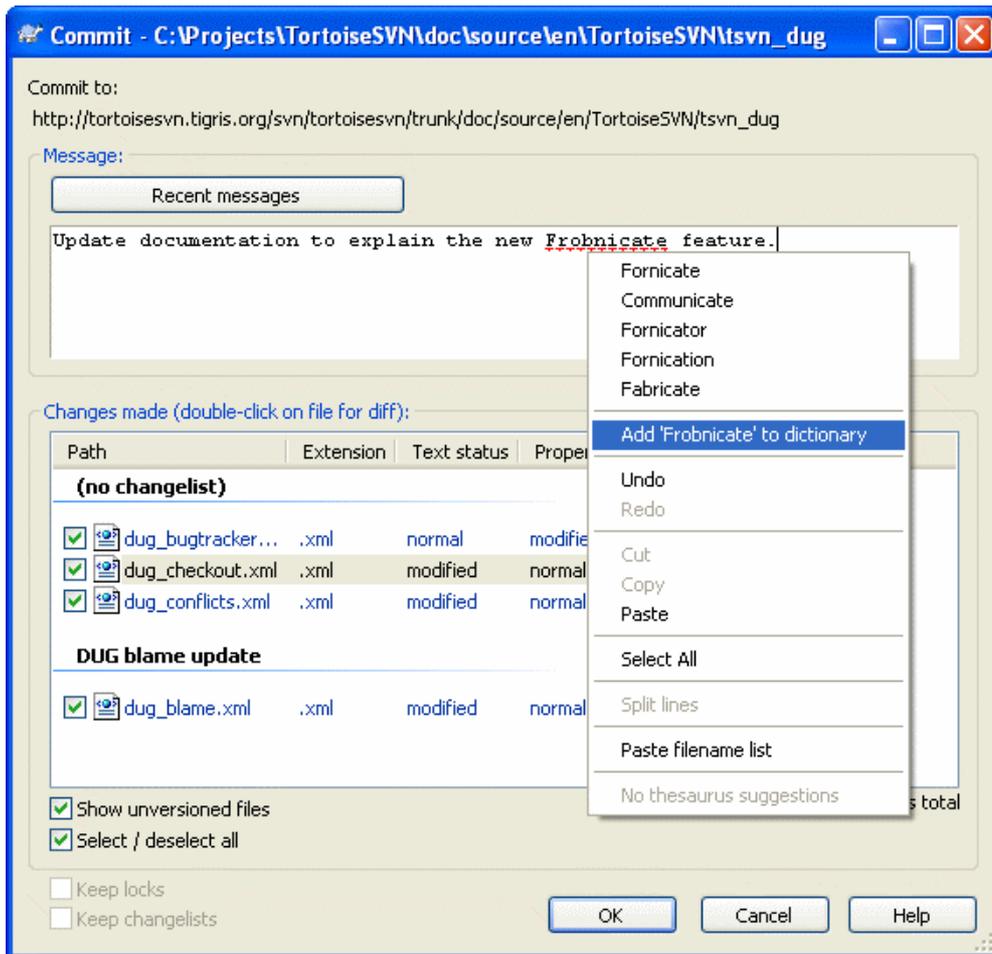
Figure 5.9. The Commit Dialog Spellchecker

TortoiseSVN includes a spellchecker to help you get your log messages right. This will highlight any mis-spelled words. Use the context menu to access the suggested corrections. Of course, it doesn't know *every* technical term that you do, so correctly spelt words will sometimes show up as errors. But don't worry. You can just add them to your personal dictionary using the context menu.

The log message window also includes a filename and function auto-completion facility. This uses regular expressions to extract class and function names from the (text) files you are committing, as well as the filenames themselves. If a word you are typing matches anything in the list (after you have typed at least 3 characters, or pressed **Ctrl+Space**), a drop-down appears allowing you to select the full name. The regular expressions supplied with TortoiseSVN are held in the TortoiseSVN installation `bin` folder. You can also define your own regexes and store them in `%APPDATA%\TortoiseSVN\autolist.txt`. Of course your private autolist will not be overwritten when you update your installation of TortoiseSVN. If you are unfamiliar with regular expressions, take a look at the introduction at *http://en.wikipedia.org/wiki/Regular_expression* [http://en.wikipedia.org/wiki/Regular_expression], and the online documentation and tutorial at *http://www.regular-expressions.info/* [http://www.regular-expressions.info/].

You can re-use previously entered log messages. Just click on Recent messages to view a list of the last few messages you entered for this working copy. The number of stored messages can be customized in the TortoiseSVN settings dialog.

You can clear all stored commit messages from the Saved data page of TortoiseSVN's settings, or you can clear individual messages from within the Recent messages dialog using the **Delete** key.

> ### Special Folder Properties
>
> There are several special folder properties which can be used to help give more control over the formatting of commit log messages and the language used by the spellchecker module. Read Section 5.17, "Project Settings" for further information.

> ### Integration with Bug Tracking Tools
>
> If you have activated the bug tracking system, you can set one or more Issues in the Bug-ID / Issue-Nr: text box. Multiple issues should be comma separated. Alternatively, if you are using regex-based bug tracking support, just add your issue references as part of the log message. Learn more in Section 5.27, "Integration with Bug Tracking Systems / Issue Trackers".

5.4.5. Commit Progress

After pressing OK, a dialog appears displaying the progress of the commit.
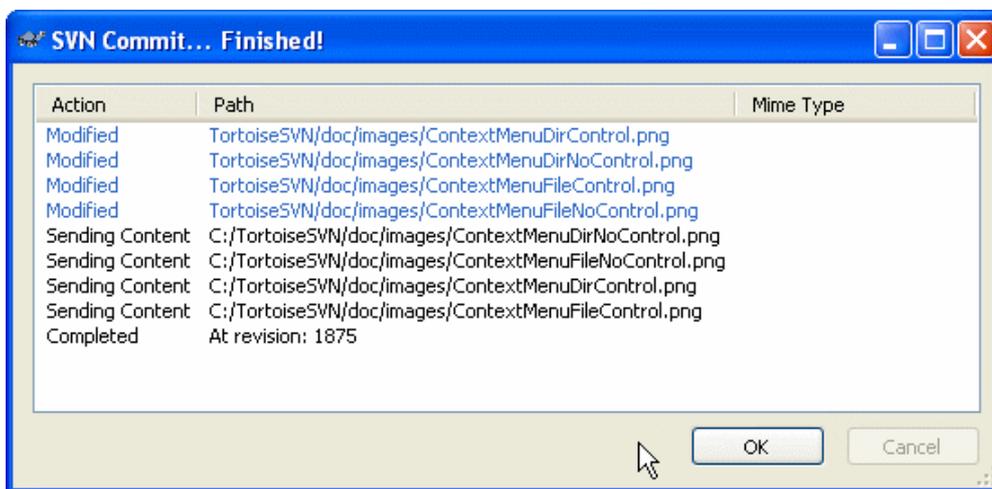
Figure 5.10. The Progress dialog showing a commit in progress

The progress dialog uses colour coding to highlight different commit actions

Blue
: Committing a modification.

Purple
: Committing a new addition.

Dark red
: Committing a deletion or a replacement.

Black
: All other items.

This is the default colour scheme, but you can customise those colours using the settings dialog. Read Section 5.29.1.4, "TortoiseSVN Colour Settings" for more information.

5.5. Update Your Working Copy With Changes From Others
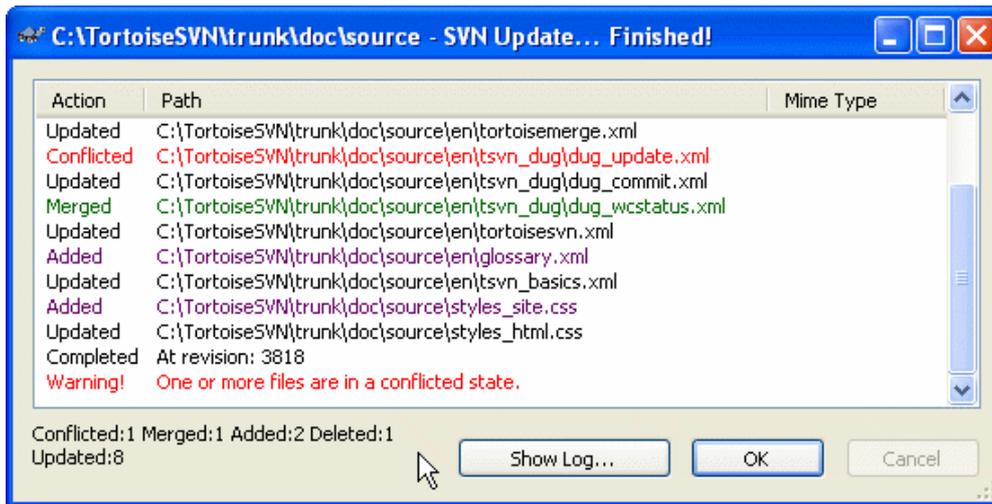
Figure 5.11. Progress dialog showing finished update

Periodically, you should ensure that changes done by others get incorporated in your local working copy. The process of getting changes from the server to your local copy is known as *updating*. Updating may be done on single files, a set of selected files, or recursively on entire directory hierarchies. To update, select the files and/or directories you want, right click and select TortoiseSVN → Update in the explorer context menu. A window will pop up displaying the progress of the update as it runs. Changes done by others will be merged into your files, keeping any changes you may have done to the same files. The repository is *not* affected by an update.

The progress dialog uses colour coding to highlight different update actions

Purple
:   New item added to your WC.

Dark red
:   Redundant item deleted from your WC, or missing item replaced in your WC.

Green
:   Changes from repository successfully merged with your local changes.

Bright red
:   Changes from repository merged with local changes, resulting in conflicts which you need to resolve.

Black
:   Unchanged item in your WC updated with newer version from the repository.

This is the default colour scheme, but you can customise those colours using the settings dialog. Read Section 5.29.1.4, "TortoiseSVN Colour Settings" for more information.

If you get any *conflicts* during an update (this can happen if others changed the same lines in the same file as you did and those changes don't match) then the dialog shows those conflicts in red. You can double click on these lines to start the external merge tool to resolve the conflicts.

When the update is complete, the progress dialog shows a summary of the number of items updated, added, removed, conflicted, etc. below the file list. This summary information can be copied to the clipboard using **Ctrl+C**.

The standard Update command has no options and just updates your working copy to the HEAD revision of the repository, which is the most common use case. If you want more control over the update process, you should use TortoiseSVN → Update to Revision... instead. This allows you to update your working copy to a specific revision, not only to the most recent one. Suppose your working copy is at revision 100, but you want it to reflect the state which it had in revision 50 - then simply update to revision 50.

In the same dialog you can also choose the *depth* at which to update the current folder. The terms used are described in Section 5.3.1, "Checkout Depth". The default depth is Working copy, which preserves the existing depth setting. You can also choose whether to ignore any external projects in the update (i.e. projects referenced using svn:externals).

> ### Caution
>
> If you update a file or folder to a specific revision, you should not make changes to those files. You will get "out of date" error messages when you try to commit them! If you want to undo changes to a file and start afresh from an earlier revision, you can rollback to a previous revision from the revision log dialog. Take a look at Section B.4, "Roll back (Undo) revisions in the repository" for further instructions, and alternative methods.

Update to Revision can occasionally be useful to see what your project looked like at some earlier point in its history. But in general, updating individual files to an earlier revision is not a good idea as it leaves your working copy in an inconsistent state. If the file you are updating has changed name, you may even find that the file just disappears from your working copy because no file of that name existed in the earlier revision. If you simply want a local copy of an old version of a file it is better to use the Context Menu → Save revision to... command from the log dialog for that file.

> ### Multiple Files/Folders
>
> If you select multiple files and folders in the explorer and then select Update, all of those files/folders are updated one by one. TortoiseSVN makes sure that all files/folders which are from the same repository are updated to the exact same revision! Even if between those updates another commit occurred.

> ### Local File Already Exists
>
> Sometimes when you try to update, the update fails with a message to say that there is already a local file of the same name. This typically happens when Subversion tries to checkout a newly versioned file, and finds that an unversioned file of the same name already exists in your working folder. Subversion will never overwrite an unversioned file - it might contain something you are working on, which coincidentally has the same filename as another developer has used for his newly committed file.
>
> If you get this error message, the solution is simply to rename the local unversioned file. After completing the update, you can check whether the renamed file is still needed.
>
> If you keep getting error messages, use TortoiseSVN → Check for Modifications instead to list all the problem files. That way you can deal with them all at once.

5.6. Resolving Conflicts

Once in a while, you will get a *conflict* when you update your files from the repository. A conflict occurs when two or more developers have changed the same few lines of a file. As Subversion knows nothing of your project, it leaves resolving the conflicts to the developers. Whenever a conflict is reported, you should open the file in question, and search for lines starting with the string <<<<<<. The conflicting area is marked like this:

```
<<<<<< filename
    your changes
=======
```

```
     code merged from repository
>>>>>>> revision
```

Also, for every conflicted file Subversion places three additional files in your directory:

filename.ext.mine
> This is your file as it existed in your working copy before you updated your working copy - that is, without conflict markers. This file has your latest changes in it and nothing else.

filename.ext.rOLDREV
> This is the file that was the BASE revision before you updated your working copy. That is, it the file that you checked out before you made your latest edits.

filename.ext.rNEWREV
> This is the file that your Subversion client just received from the server when you updated your working copy. This file corresponds to the HEAD revision of the repository.

You can either launch an external merge tool / conflict editor with TortoiseSVN → Edit Conflicts or you can use any other editor to manually resolve the conflict. You should decide what the code should look like, do the necessary changes and save the file.

Afterwards execute the command TortoiseSVN → Resolved and commit your modifications to the repository. Please note that the Resolve command does not really resolve the conflict. It just removes the `filename.ext.mine` and `filename.ext.r*` files, to allow you to commit your changes.

If you have conflicts with binary files, Subversion does not attempt to merge the files itself. The local file remains unchanged (exactly as you last changed it) and you have `filename.ext.r*` files. If you want to discard your changes and keep the repository version, just use the Revert command. If you want to keep your version and overwrite the repository version, use the Resolved command, then commit your version.

You can use the Resolved command for multiple files if you right click on the parent folder and select TortoiseSVN → Resolved... This will bring up a dialog listing all conflicted files in that folder, and you can select which ones to mark as resolved.

5.7. Getting Status Information

While you are working on your working copy you often need to know which files you have changed/added/removed or renamed, or even which files got changed and committed by others.

5.7.1. Icon Overlays
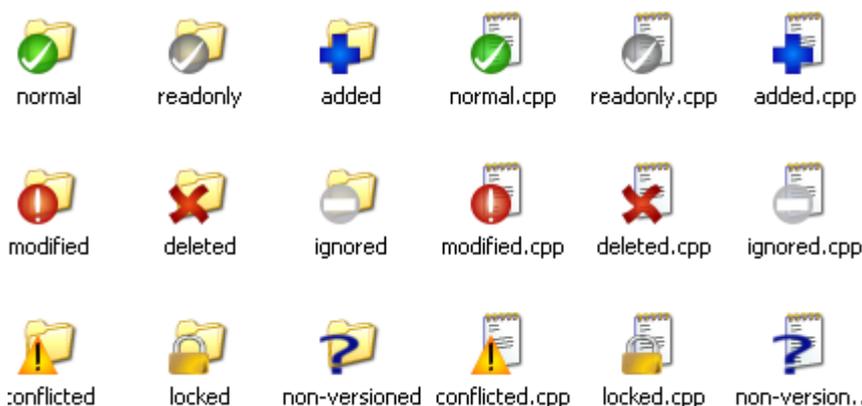
Figure 5.12. Explorer showing icon overlays

Now that you have checked out a working copy from a Subversion repository you can see your files in the windows explorer with changed icons. This is one of the reasons why TortoiseSVN is so popular. TortoiseSVN adds a so called overlay icon to each file icon which overlaps the original file icon. Depending on the Subversion status of the file the overlay icon is different.

A fresh checked out working copy has a green checkmark as overlay. That means the Subversion status is *normal*.

As soon as you start editing a file, the status changes to *modified* and the icon overlay then changes to a red exclamation mark. That way you can easily see which files were changed since you last updated your working copy and need to be committed.

If during an update a *conflict* occurs then the icon changes to a yellow exclamation mark.

If you have set the `svn:needs-lock` property on a file, Subversion makes that file read-only until you get a lock on that file. Such files have this overlay to indicate that you have to get a lock first before you can edit that file.

If you hold a lock on a file, and the Subversion status is *normal*, this icon overlay reminds you that you should release the lock if you are not using it to allow others to commit their changes to the file.

This icon shows you that some files or folders inside the current folder have been scheduled to be *deleted* from version control or a file under version control is missing in a folder.

The plus sign tells you that a file or folder has been scheduled to be *added* to version control.

The bar sign tells you that a file or folder is *ignored* for version control purposes. This overlay is optional.

This icon shows files and folders which are not under version control, but have not been ignored. This overlay is optional.

In fact, you may find that not all of these icons are used on your system. This is because the number of overlays allowed by Windows is very limited and if you are also using an old version of TortoiseCVS, then there are not enough overlay slots available. TortoiseSVN tries to be a "Good Citizen (TM)" and limits its use of overlays to give other apps a chance too.

Now that there are more Tortoise clients around (TortoiseCVS, TortoiseHG, ...) the icon limit becomes a real problem. To work around this, the TortoiseSVN project introduced a common shared icon set,

loaded as a DLL, which can be used by all Tortoise clients. Check with your client provider to see if this has been integrated yet :-)

For a description of how icon overlays correspond to Subversion status and other technical details, read Section F.1, "Icon Overlays".

5.7.2. TortoiseSVN Columns In Windows Explorer

The same information which is available from the icon overlays (and much more) can be displayed as additional columns in Windows Explorer's Details View.

Simply right click on one of the headings of a column, choose More... from the context menu displayed. A dialog will appear where you can specify the columns and their order, which is displayed in the "Detailed View". Scroll down until the entries starting with SVN come into view. Check the ones you would like to have displayed and close the dialog by pressing OK. The columns will be appended to the right of those currently displayed. You can reorder them by drag and drop, or resize them, so that they fit your needs.

> **Important**
>
> The additional columns in the Windows Explorer are not available on Vista, since Microsoft decided to not allow such columns for *all* files anymore but only for specific file types.

> **Tip**
>
> If you want the current layout to be displayed in all your working copies, you may want to make this the default view.

5.7.3. Local and Remote Status
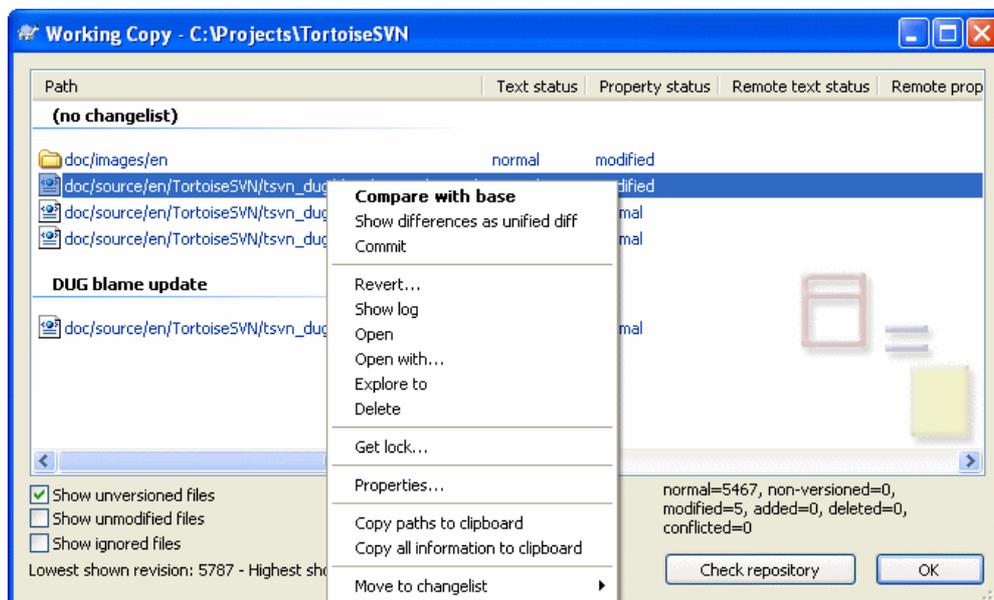
Figure 5.13. Check for Modifications

It's often very useful to know which files you have changed and also which files got changed and committed by others. That's where the command TortoiseSVN → Check For Modifications... comes in handy. This dialog will show you every file that has changed in any way in your working copy, as well as any unversioned files you may have.

If you click on the Check Repository then you can also look for changes in the repository. That way you can check before an update if there's a possible conflict. You can also update selected files from the repository without updating the whole folder.

The dialog uses colour coding to highlight the status.

Blue
> Locally modified items.

Purple
> Added items. Items which have been added with history have a + sign in the Text status column, and a tooltip shows where the item was copied from.

Dark red
> Deleted or missing items.

Green
> Items modified locally and in the repository. The changes will be merged on update. These *may* produce conflicts on update.

Bright red
> Items modified locally and deleted in repository, or modified in repository and deleted locally. These *will* produce conflicts on update.

Black
> Unchanged and unversioned items.

This is the default colour scheme, but you can customise those colours using the settings dialog. Read Section 5.29.1.4, "TortoiseSVN Colour Settings" for more information.

Items which have been switched to a different repository path are also indicated using an (s) marker. You may have switched something while working on a branch and forgotten to switch back to trunk. This is your warning sign!

From the context menu of the dialog you can show a diff of the changes. Check the local changes *you* made using Context Menu → Compare with Base. Check the changes in the repository made by others using Context Menu → Show Differences as Unified Diff.

You can also revert changes in individual files. If you have deleted a file accidentally, it will show up as *Missing* and you can use *Revert* to recover it.

Unversioned and ignored files can be sent to the recycle bin from here using Context Menu → Delete. If you want to delete files permanently (bypassing the recycle bin) hold the **Shift** key while clicking on Delete.

If you want to examine a file in detail, you can drag it from here into another application such as a text editor or IDE.

The columns are customizable. If you right click on any column header you will see a context menu allowing you to select which columns are displayed. You can also change column width by using the drag handle which appears when you move the mouse over a column boundary. These customizations are preserved, so you will see the same headings next time.

If you are working on several unrelated tasks at once, you can also group files together into changelists. Read Section 5.4.2, "Change Lists" for more information.

At the bottom of the dialog you can see a summary of the range of repository revisions in use in your working copy. These are the *commit* revisions, not the *update* revisions; they represent the range of revisions where these files were last committed, not the revisions to which they have been updated. Note that the revision range shown applies only to the items displayed, not to the entire working copy. If you want to see that information for the whole working copy you must check the Show unmodified files checkbox.

> **Tip**
>
> If you want a flat view of your working copy, i.e. showing all files and folders at every level of the folder hierarchy, then the Check for Modifications dialog is the easiest way to achieve that. Just check the Show unmodified files checkbox to show all files in your working copy.

> **Repairing External Renames**
>
> Sometimes files get renamed outside of Subversion, and they show up in the file list as a missing file and an unversioned file. To avoid losing the history you need to notify Subversion about the connection. Simply select both the old name (missing) and the new name (unversioned) and use Context Menu → Repair Move to pair the two files as a rename.

5.7.4. Viewing Diffs

Often you want to look inside your files, to have a look at what you've changed. You can accomplish this by selecting a file which has changed, and selecting Diff from TortoiseSVN's context menu. This starts the external diff-viewer, which will then compare the current file with the pristine copy (BASE revision), which was stored after the last checkout or update.

> **Tip**
>
> Even when not inside a working copy or when you have multiple versions of the file lying around, you can still display diffs:
>
> Select the two files you want to compare in explorer (e.g. using **Ctrl** and the mouse) and choose Diff from TortoiseSVN's context menu. The file clicked last (the one with the focus, i.e. the dotted rectangle) will be regarded as the later one.

5.8. Change Lists

In an ideal world, you only ever work on one thing at a time, and your working copy contains only one set of logical changes. OK, back to reality. It often happens that you have to work on several unrelated tasks at once, and when you look in the commit dialog, all the changes are mixed in together. The *changelist* feature helps you group files together, making it easier to see what you are doing. Of course this can only work if the changes do not overlap. If two different tasks affect the same file, there is no way to separate the changes.

> **Important**
>
> The changelist feature in TortoiseSVN is only available in Windows XP and later, as it depends on a shell capability which is not present in Windows 2000. Sorry, but Win2K is really quite old now, so please don't complain.

You can see changelists in several places, but the most important ones are the commit dialog and the check-for-modifications dialog. Let's start in the check-for-modifications dialog after you have worked on several features and many files. When you first open the dialog, all the changed files are listed together. Suppose you now want to organise things and group those files according to feature.

Select one or more files and use Context Menu → Move to changelist to add an item to a changelist. Initially there will be no changelists, so the first time you do this you will create a new changelist. Give

it name which describes what you are using it for, and click OK. The dialog will now change to show groups of items.

Once you have created a changelist you can drag and drop items into it, either from another changelist, or from Windows Explorer. Dragging from Explorer can be useful as it allows you to add items to a changelist before the file is modified. You could do that from the check-for-modifications dialog, but only by displaying all unmodified files.
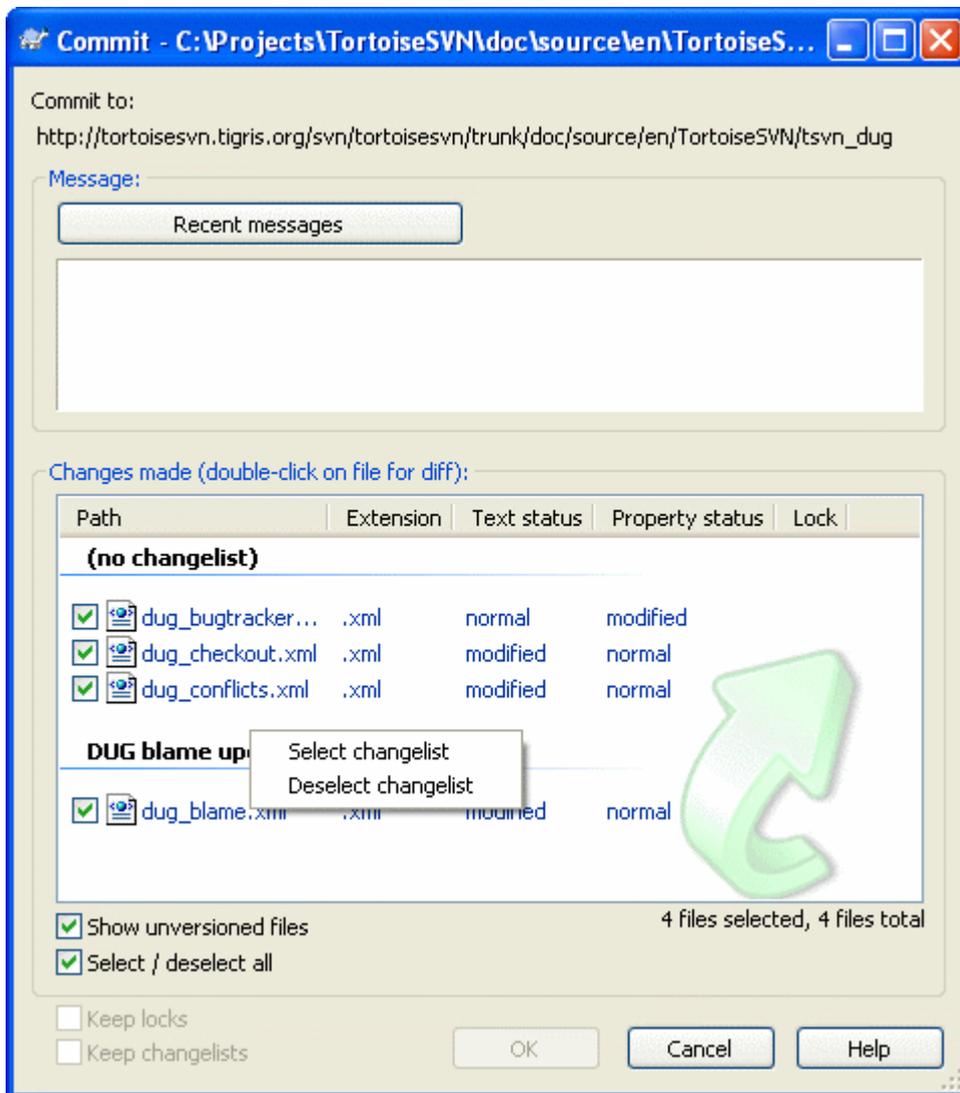
Figure 5.14. Commit dialog with Changelists

In the commit dialog you can see those same files, grouped by changelist. Apart from giving an immediate visual indication of groupings, you can also use the group headings to select which files to commit.

On XP, there is a context menu when you right click on a group heading which gives you the choice to check or uncheck all group entries. On Vista however the context menu is not necessary. Click on the group header to select all entries, then check one of the selected entries to check all.

TortoiseSVN reserves one changelist name for its own use, namely `ignore-on-commit`. This is used to mark versioned files which you almost never want to commit even though they have local changes. This feature is described in Section 5.4.3, "Excluding Items from the Commit List".

When you commit files belonging to a changelist then normally you would expect that the changelist membership is no longer needed. So by default, files are removed from changelists automatically on

commit. If you wish to retain the file in its changelist, use the Keep changelists checkbox at the bottom of the commit dialog.

> **Tip**
>
> Changelists are purely a local client feature. Creating and removing changelists will not affect the repository, nor anyone else's working copy. They are simply a convenient way for you to organise your files.

5.9. Revision Log Dialog

For every change you make and commit, you should provide a log message for that change. That way you can later find out what changes you made and why, and you have a detailed log for your development process.

The Revision Log Dialog retrieves all those log messages and shows them to you. The display is divided into 3 panes.

- The top pane shows a list of revisions where changes to the file/folder have been committed. This summary includes the date and time, the person who committed the revision and the start of the log message.

  Lines shown in blue indicate that something has been copied to this development line (perhaps from a branch).

- The middle pane shows the full log message for the selected revision.

- The bottom pane shows a list of all files and folders that were changed as part of the selected revision.

But it does much more than that - it provides context menu commands which you can use to get even more information about the project history.

5.9.1. Invoking the Revision Log Dialog
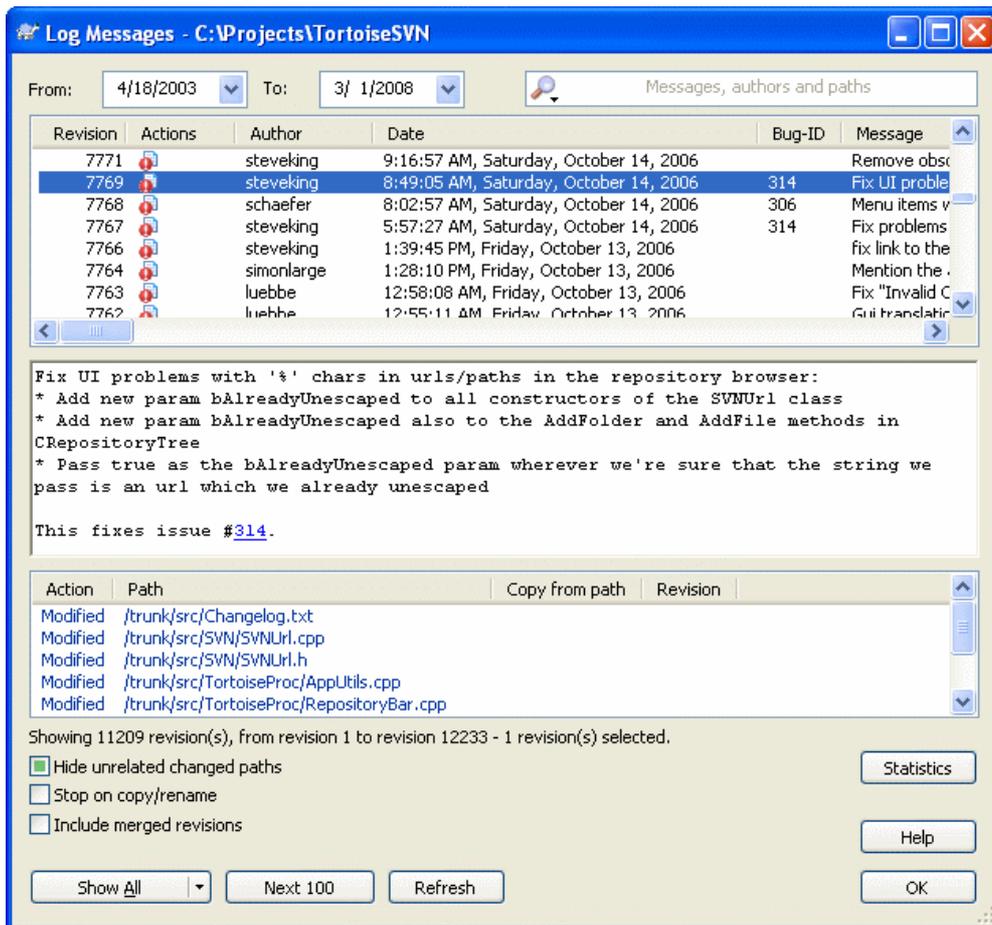
Figure 5.15. The Revision Log Dialog

There are several places from where you can show the Log dialog:

- From the TortoiseSVN context submenu

- From the property page

- From the Progress dialog after an update has finished. Then the Log dialog only shows those revisions which were changed since your last update

5.9.2. Revision Log Actions

The top pane has an Actions column containing icons that summarize what has been done in that revision. There are four different icons, each shown in its own column.

If a revision modified a file or directory, the *modified* icon is shown in the first column.

If a revision added a file or directory, the *added* icon is shown in the second column.

If a revision deleted a file or directory, the *deleted* icon is shown in the third column.

If a revision replaced a file or directory, the *replaced* icon is shown in the fourth column.

5.9.3. Getting Additional Information

Compare with working copy
Show changes as unified diff
Compare with previous revision
Compare and blame with previous revision

Browse repository
Create branch/tag from revision
Update item to revision
Revert to this revision
Revert changes from this revision
Merge revision to...
Checkout...
Export...

Edit author
Edit log message
Show revision properties

Copy to clipboard
Search log messages...

Figure 5.16. The Revision Log Dialog Top Pane with Context Menu

The top pane of the Log dialog has a context menu that allows you to access much more information. Some of these menu entries appear only when the log is shown for a file, and some only when the log is shown for a folder.

- Compare the selected revision with your working copy. The default Diff-Tool is TortoiseMerge which is supplied with TortoiseSVN. If the log dialog is for a folder, this will show you a list of changed files, and allow you to review the changes made to each file individually.

- View the changes made in the selected revision as a Unified-Diff file (GNU patch format). This shows only the differences with a few lines of context. It is harder to read than a visual file compare, but will show all file changes together in a compact format.

- Compare the selected revision with the previous revision. This works in a similar manner to comparing with your working copy.

- Blame the selected revision, and the file in your working BASE and compare the blame reports using a visual diff tool. Read Section 5.22.2, "Blame Differences" for more detail. (files only).

- Blame the selected revision, and the previous revision, and compare the results using a visual diff tool. (folders only).

- Save the selected revision to a file so you have an older version of that file. (files only).

- Open the selected file, either with the default viewer for that file type, or with a program you choose. (files only).

- Blame the file up to the selected revision. (files only).

- Open the repository browser to examine the selected file or folder in the repository as it was at the selected revision.

- Create a branch/tag from a selected revision. This is useful e.g. if you forgot to create a tag and already committed some changes which weren't supposed to get into that release.

- Update your working copy to the selected revision. Useful if you want to have your working copy reflect a time in the past. It is best to update a whole directory in your working copy, not just one file, otherwise your working copy will be inconsistent and you will be unable to commit any changes.

- Revert changes from which were made in the selected revision. The changes are reverted in your working copy so this operation does *not* affect the repository at all! Note that this will undo the changes made in that revision only. It does not replace your working copy with the entire file at the earlier revision. This is very useful for undoing an earlier change when other unrelated changes have been made since. If you have made local changes, this command will merge these changes into your working copy.

- Revert to an earlier revision. If you have made several changes, and then decide that you really want to go back to how things were in revision N, this is the command you need. Again, the changes are reverted in your working copy so this operation does *not* affect the repository until you commit the changes. Note that this will undo *all* changes made after the selected revision, replacing the file/folder with the earlier version. If you have made local changes, this command will merge these changes into your working copy.

- Merge the selected revision(s) into a different working copy. A folder selection dialog allows you to choose the working copy to merge into, but after that there is no confirmation dialog, nor any opportunity to try a dry run. It is a good idea to merge into an unmodified working copy so that you can revert the changes if it doesn't work out! This is a useful feature if you want to merge selected revisions from one branch to another.

- Make a fresh checkout of the selected folder at the selected revision. This brings up a dialog for you to confirm the URL and revision, and select a location for the checkout.

- Export the selected file/folder at the selected revision. This brings up a dialog for you to confirm the URL and revision, and select a location for the export.

- Edit the log message or author attached to a previous commit. Read Section 5.9.6, "Changing the Log Message and Author" to find out how this works.

- Copy the log details of the selected revisions to the clipboard. This will copy the revision number, author, date, log message and the list of changed items for each revision.

- Search log messages for the text you enter. This searches the log messages that you entered and also the action summaries created by Subversion (shown in the bottom pane). The search is not case sensitive.
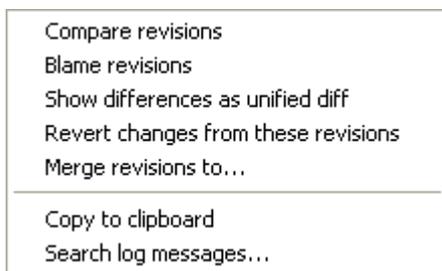
Figure 5.17. Top Pane Context Menu for 2 Selected Revisions

If you select two revisions at once (using the usual **Ctrl**-modifier), the context menu changes and gives you fewer options:

- Compare the two selected revisions using a visual difference tool. The default Diff-Tool is TortoiseMerge which is supplied with TortoiseSVN.

If you select this option for a folder, a further dialog pops up listing the changed files and offering you further diff options. Read more about the Compare Revisions dialog in Section 5.10.3, "Comparing Folders".

- Blame the two revisions and compare the blame reports using a visual difference tool. Read Section 5.22.2, "Blame Differences" for more detail.

- View the differences between the two selected revisions as a Unified-Diff file. This works for files and folders.

- Copy log messages to clipboard as described above.

- Search log messages as described above.

If you select two or more revisions (using the usual **Ctrl** or **Shift** modifiers), the context menu will include an entry to Revert all changes which were made in the selected revisions. This is the easiest way to rollback a group of revisions in one go.

You can also choose to merge the selected revisions to another working copy, as described above.
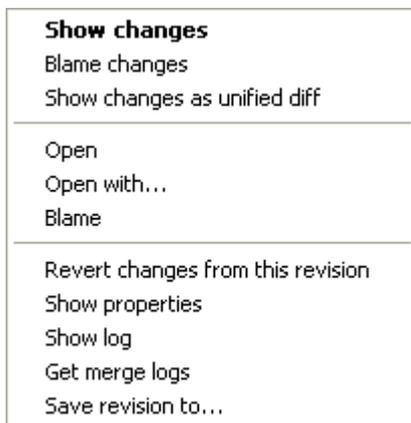
Figure 5.18. The Log Dialog Bottom Pane with Context Menu

The bottom pane of the Log dialog also has a context menu that allows you to

- Show changes made in the selected revision for the selected file. This context menu is only available for files shown as *modified*.

- Blame the selected revision and the previous revision for the selected file, and compare the blame reports using a visual diff tool. Read Section 5.22.2, "Blame Differences" for more detail.

- Open the selected file, either with the default viewer for that file type, or with a program you choose.

- Revert the changes made to the selected file in that revision.

- View the Subversion properties for the selected item.

- Show the revision log for the selected single file.

- Save the selected revision to a file so you have an older version of that file.

Tip

You may notice that sometimes we refer to changes and other times to differences. What's the difference?

Subversion uses revision numbers to mean 2 different things. A revision generally represents the state of the repository at a point in time, but it can also be used to represent the changeset which created that revision, eg. "Done in r1234" means that the changes committed in r1234 implement feature X. To make it clearer which sense is being used, we use two different terms.

If you select two revisions N and M, the context menu will offer to show the *difference* between those two revisions. In Subversion terms this is `diff -r M:N`.

If you select a single revision N, the context menu will offer to show the *changes* made in that revision. In Subversion terms this is `diff -r N-1:N` or `diff -c N`.

The bottom pane shows the files changed in all selected revisions, so the context menu always offers to show *changes*.

5.9.4. Getting more log messages

The Log dialog does not always show all changes ever made for a number of reasons:

- For a large repository there may be hundreds or even thousands of changes and fetching them all could take a long time. Normally you are only interested in the more recent changes. By default, the number of log messages fetched is limited to 100, but you can change this value in TortoiseSVN → Settings (Section 5.29.1.2, "TortoiseSVN Dialog Settings 1"),

- When the Stop on copy/rename box is checked, Show Log will stop at the point that the selected file or folder was copied from somewhere else within the repository. This can be useful when looking at branches (or tags) as it stops at the root of that branch, and gives a quick indication of changes made in that branch only.

  Normally you will want to leave this option unchecked. TortoiseSVN remembers the state of the checkbox, so it will respect your preference.

  When the Show Log dialog is invoked from within the Merge dialog, the box is always checked by default. This is because merging is most often looking at changes on branches, and going back beyond the root of the branch does not make sense in that instance.

  Note that Subversion currently implements renaming as a copy/delete pair, so renaming a file or folder will also cause the log display to stop if this option is checked.

If you want to see more log messages, click the Next 100 to retrieve the next 100 log messages. You can repeat this as many times as needed.

Next to this button there is a multi-function button which remembers the last option you used it for. Click on the arrow to see the other options offered.

Use Show Range ... if you want to view a specific range of revisions. A dialog will then prompt you to enter the start and end revision.

Use Show All if you want to see *all* log messages from HEAD right back to revision 1.

5.9.5. Merge Tracking Features

Subversion 1.5 and later keeps a record of merges using properties. This allows us to get a more detailed history of merged changes. For example, if you develop a new feature on a branch and then merge that branch back to trunk, the feature development will show up on the trunk log as a single commit for the merge, even though there may have been 1000 commits during branch development.
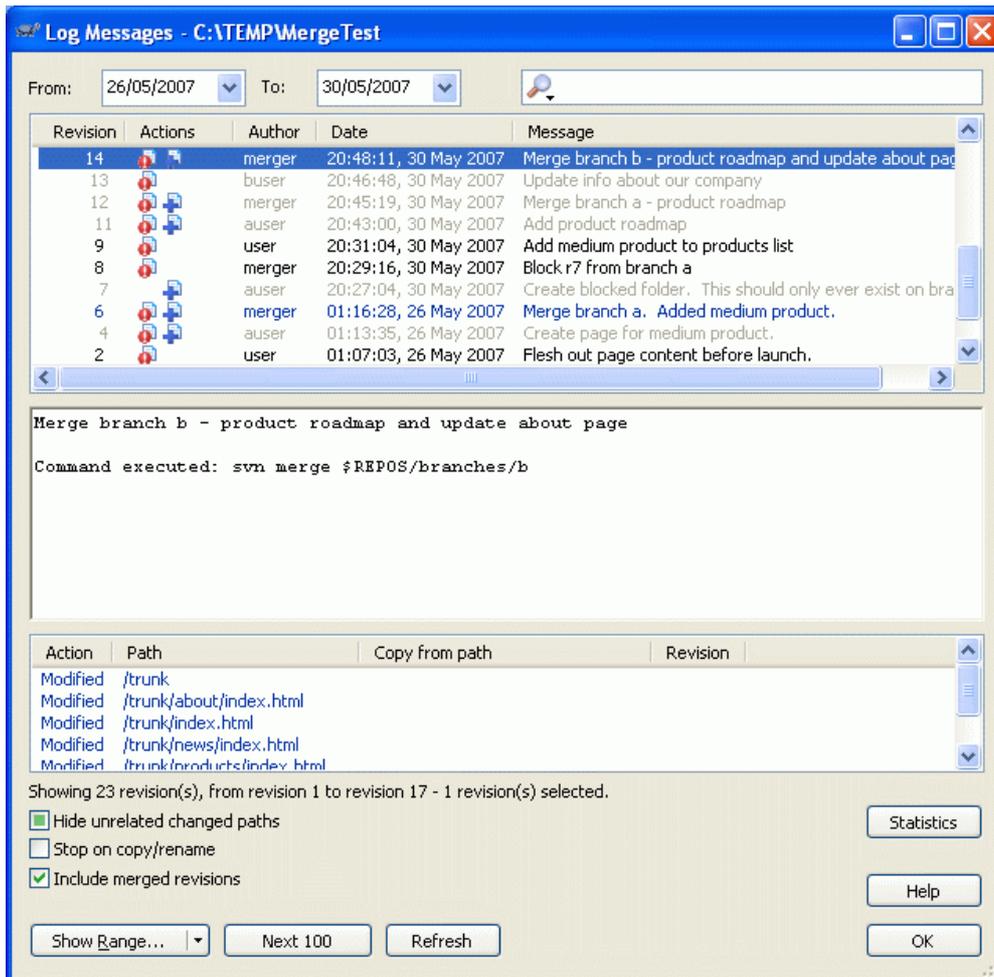
Figure 5.19. The Log Dialog Showing Merge Tracking Revisions

If you want to see the detail of which revisions were merged as part of that commit, use the Include merged revisions checkbox. This will fetch the log messages again, but will also interleave the log messages from revisions which were merged. Merged revisions are shown in grey because they represent changes made on a different part of the tree.

Of course, merging is never simple! During feature development on the branch there will probably be occasional merges back from trunk to keep the branch in sync with the main line code. So the merge history of the branch will also include another layer of merge history. These different layers are shown in the log dialog using indentation levels.

5.9.6. Changing the Log Message and Author

Sometimes you might want to change a log message you once entered, maybe because there's a spelling error in it or you want to improve the message or change it for other reasons. Or you want to change the author of the commit because you forgot to set up authentication or...

Subversion lets you change both the log message and the author of revisions any time you want. But since such changes can't be undone (those changes are not versioned) this feature is disabled by default. To make this work, you must set up a pre-revprop-change hook. Please refer to the chapter on *Hook Scripts* [http://svnbook.red-bean.com/en/1.4/svn.reposadmin.create.html#svn.reposadmin.create.hooks] in the Subversion Book for details about how to do that. Read Section 4.3, "Hook Scripts" to find some further notes on implementing hooks on a Windows machine.

Once you've set up your server with the required hooks, you can change both author and log message of any revision, using the context menu from the top pane of the Log dialog.

> ## Warning
>
> Because Subversion's revision properties are not versioned, making modifications to such a property (for example, the `svn:log` commit message property) will overwrite the previous value of that property *forever*.

5.9.7. Filtering Log Messages

If you want to restrict the log messages to show only those you are interested in rather than scrolling through a list of hundreds, you can use the filter controls at the top of the Log Dialog. The start and end date controls allow you to restrict the output to a known date range. The search box allows you to show only messages which contain a particular phrase.

Click on the search icon to select which information you want to search in, and to choose *regex* mode. Normally you will only need a simple text search, but if you need to more flexible search terms, you can use regular expressions. If you hover the mouse over the box, a tooltip will give hints on how to use the regex functions. You can also find online documentation and a tutorial at *http://www.regular-expressions.info/* [http://www.regular-expressions.info/].

Note that these filters act on the messages already retrieved. They do not control downloading of messages from the repository.

You can also filter the path names in the bottom pane using the Hide unrelated changed paths checkbox. Related paths are those which contain the path used to display the log. If you fetch the log for a folder, that means anything in that folder or below it. For a file it means just that one file. The checkbox is tristate: you can show all paths, grey out the unrelated ones, or hide the unrelated paths completely.

Sometimes your working practices will require log messages to follow a particular format, which means that the text describing the changes is not visible in the abbreviated summary shown in the top pane. The property `tsvn:logsummary` can be used to extract a portion of the log message to be shown in the top pane. Read Section 5.17.2, "TortoiseSVN Project Properties" to find out how to use this property.

5.9.8. Statistical Information

The Statistics button brings up a box showing some interesting information about the revisions shown in the Log dialog. This shows how many authors have been at work, how many commits they have made, progress by week, and much more. Now you can see at a glance who has been working hardest and who is slacking ;-)

5.9.8.1. Statistics Page

This page gives you all the numbers you can think of, in particular the period and number of revisions covered, and some min/max/average values.

5.9.8.2. Commits by Author Page
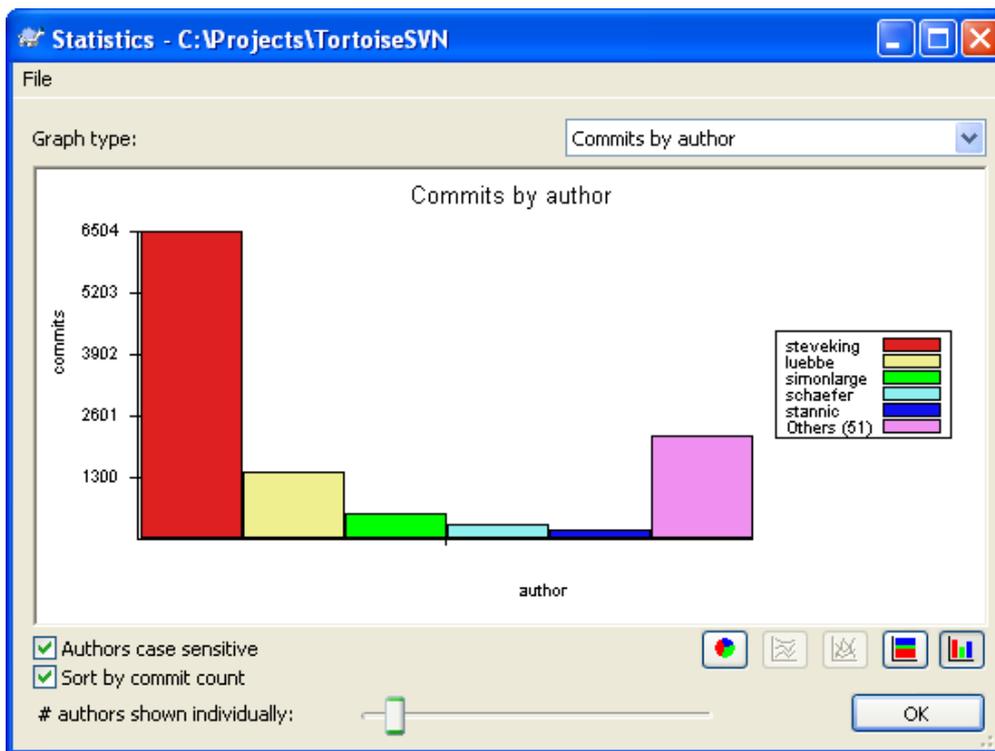
Figure 5.20. Commits-by-Author Histogram

This graph shows you which authors have been active on the project as a simple histogram, stacked histogram or pie chart.
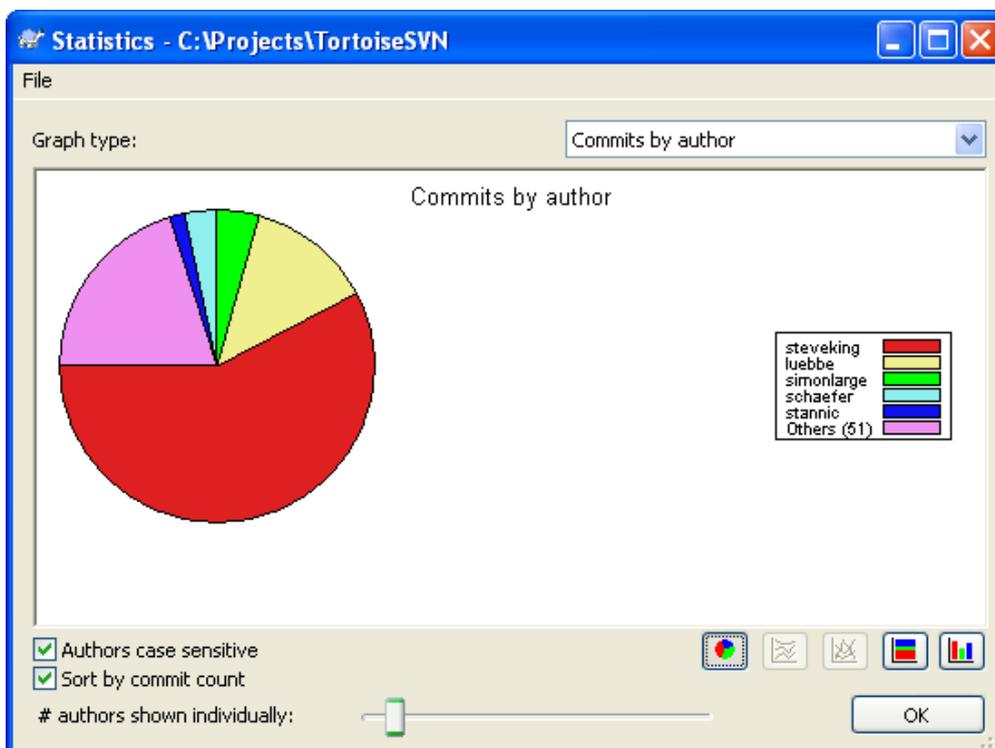
Figure 5.21. Commits-by-Author Pie Chart

Where there are a few major authors and many minor contributors, the number of tiny segments can make the graph more difficult to read. The slider at the bottom allows you to set a threshold (as a percentage of total commits) below which any activity is grouped into an *Others* category.

5.9.8.3. Commits by date Page
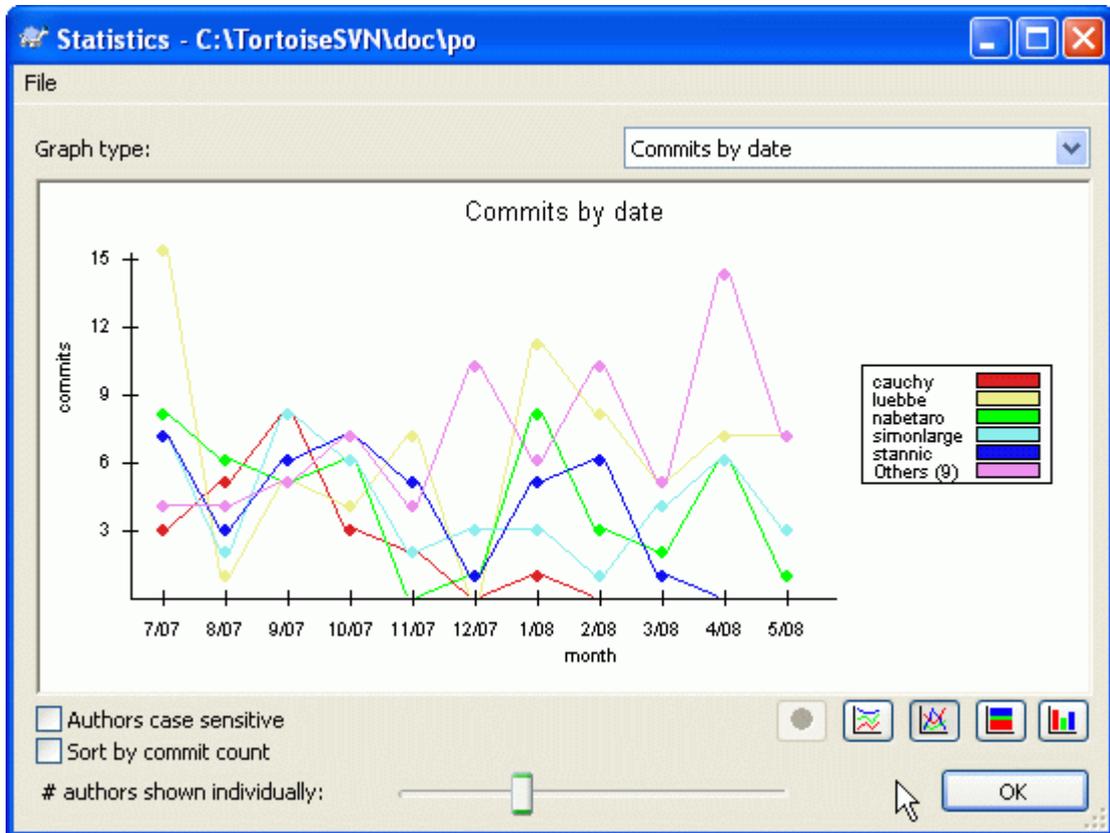
Figure 5.22. Commits-by-date Graph

This page gives you a graphical representation of project activity in terms of number of commits *and* author. This gives some idea of when a project is being worked on, and who was working at which time.

When there are several authors, you will get many lines on the graph. There are two views available here: *normal*, where each author's activity is relative to the base line, and *stacked*, where each author's activity is relative to the line underneath. The latter option avoids the lines crossing over, which can make the graph easier to read, but less easy to see one author's output.

By default the analysis is case-sensitive, so users `PeterEgan` and `PeteRegan` are treated as different authors. However, in many cases user names are not case-sensitive, and are sometimes entered inconsistently, so you may want `DavidMorgan` and `davidmorgan` to be treated as the same person. Use the Authors case insensitive checkbox to control how this is handled.

Note that the statistics cover the same period as the Log dialog. If that is only displaying one revision then the statistics will not tell you very much.

5.9.9. Offline Mode

If the server is not reachable, and you have log caching enabled you can use the log dialog and revision graph in offline mode. This uses data from the cache, which allows you to continue working although the information may not be up-to-date or even complete.

5.9.10. Refreshing the View

If you want to check the server again for newer log messages, you can simply refresh the view using **F5**. If you are using the log cache (enabled by default), this will check the repository for newer messages and fetch only the new ones. If the log cache was in offline mode, this will also attempt to go back online.

If you are using the log cache and you think the message content or author may have changed, you can use **Shift-F5** or **Ctrl-F5** to re-fetch the displayed messages from the server and update the log cache. Note that this only affects messages currently shown and does not invalidate the entire cache for that repository.

5.10. Viewing Differences

One of the commonest requirements in project development is to see what has changed. You might want to look at the differences between two revisions of the same file, or the differences between two separate files. TortoiseSVN provides a built-in tool named TortoiseMerge for viewing differences of text files. For viewing differences of image files, TortoiseSVN also has a tool named TortoiseIDiff. Of course, you can use your own favourite diff program if you like.

5.10.1. File Differences

Local changes

If you want to see what changes *you* have made in your working copy, just use the explorer context menu and select TortoiseSVN → Diff.

Difference to another branch/tag

If you want to see what has changed on trunk (if you are working on a branch) or on a specific branch (if you are working on trunk), you can use the explorer context menu. Just hold down the **Shift** key while you right click on the file. Then select TortoiseSVN → Diff with URL. In the following dialog, specify the URL in the repository with which you want to compare your local file to.

You can also use the repository browser and select two trees to diff, perhaps two tags, or a branch/tag and trunk. The context menu there allows you to compare them using Compare revisions. Read more in Section 5.10.3, "Comparing Folders".

Difference from a previous revision

If you want to see the difference between a particular revision and your working copy, use the Revision Log dialog, select the revision of interest, then select Compare with working copy from the context menu.

If you want to see the difference between the last committed revision and your working copy, assuming that the working copy hasn't been modified, just right click on the file. Then select TortoiseSVN → Diff with previous version. This will perform a diff between the revision before the last-commit-date (as recorded in your working copy) and the working BASE. This shows you the last change made to that file to bring it to the state you now see in your working copy. It will not show changes newer than your working copy.

Difference between two previous revisions

If you want to see the difference between two revisions which are already committed, use the Revision Log dialog and select the two revisions you want to compare (using the usual **Ctrl**-modifier). Then select Compare revisions from the context menu.

If you did this from the revision log for a folder, a Compare Revisions dialog appears, showing a list of changed files in that folder. Read more in Section 5.10.3, "Comparing Folders".

All changes made in a commit

If you want to see the changes made to all files in a particular revision in one view, you can use Unified-Diff output (GNU patch format). This shows only the differences with a few lines of context.

It is harder to read than a visual file compare, but will show all the changes together. From the Revision Log dialog select the revision of interest, then select Show Differences as Unified-Diff from the context menu.

Difference between files

If you want to see the differences between two different files, you can do that directly in explorer by selecting both files (using the usual **Ctrl**-modifier). Then from the explorer context menu select TortoiseSVN → Diff.

Difference between WC file/folder and a URL

If you want to see the differences between a file in your working copy, and a file in any Subversion repository, you can do that directly in explorer by selecting the file then holding down the **Shift** key whilst right clicking to obtain the context menu. Select TortoiseSVN → Diff with URL. You can do the same thing for a working copy folder. TortoiseMerge shows these differences in the same way as it shows a patch file - a list of changed files which you can view one at a time.

Difference with blame information

If you want to see not only the differences but also the author, revision and date that changes were made, you can combine the diff and blame reports from within the revision log dialog. Read Section 5.22.2, "Blame Differences" for more detail.

Difference between folders

The built-in tools supplied with TortoiseSVN do not support viewing differences between directory hierarchies. But if you have an external tool which does support that feature, you can use that instead. In Section 5.10.5, "External Diff/Merge Tools" we tell you about some tools which we have used.

If you have configured a third party diff tool, you can use **Shift** when selecting the Diff command to use the alternate tool. Read Section 5.29.4, "External Program Settings" to find out about configuring other diff tools.

5.10.2. Line-end and Whitespace Options

Sometimes in the life of a project you might change the line endings from CRLF to LF, or you may change the indentation of a section. Unfortunately this will mark a large number of lines as changed, even though there is no change to the meaning of the code. The options here will help to manage these changes when it comes to comparing and applying differences. You will see these settings in the Merge and Blame dialogs, as well as in the settings for TortoiseMerge.

Ignore line endings excludes changes which are due solely to difference in line-end style.

Compare whitespaces includes all changes in indentation and inline whitespace as added/removed lines.

Ignore whitespace changes excludes changes which are due solely to a change in the amount or type of whitespace, eg. changing the indentation or changing tabs to spaces. Adding whitespace where there was none before, or removing a whitespace completely is still shown as a change.

Ignore all whitespaces excludes all whitespace-only changes.

Naturally, any line with changed content is always included in the diff.

5.10.3. Comparing Folders
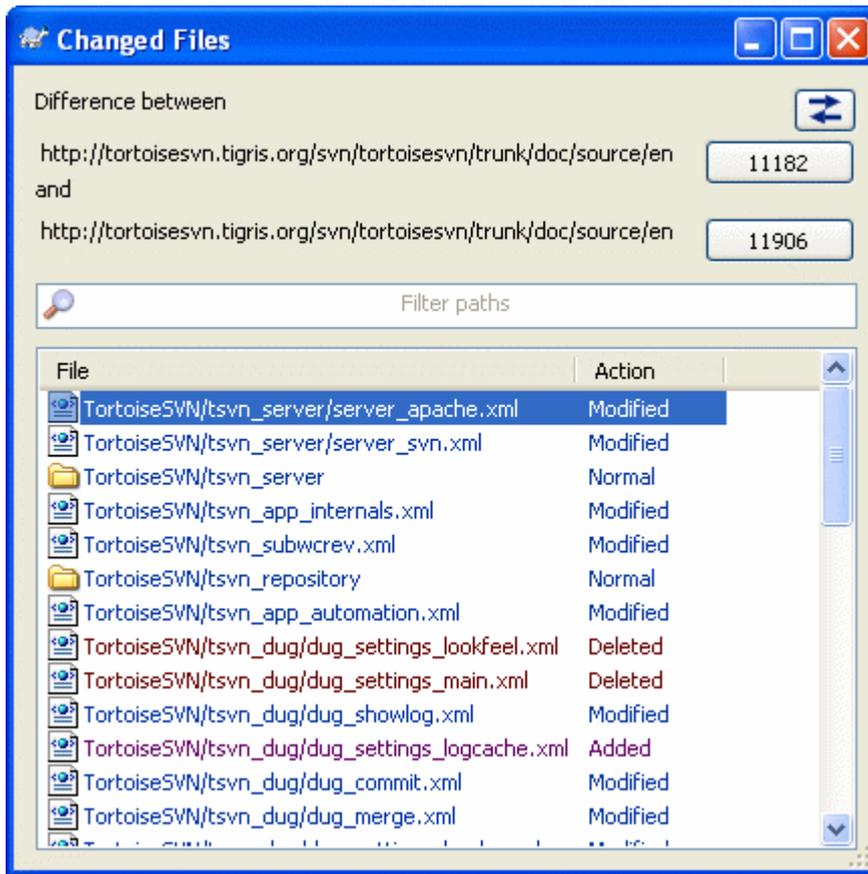
Figure 5.23. The Compare Revisions Dialog

When you select two trees within the repository browser, or when you select two revisions of a folder in the log dialog, you can Context menu → Compare revisions.

This dialog shows a list of all files which have changed and allows you to compare or blame them individually using context menu.

You can also export the list of changed files to a text file, or you can export the changed files themselves to a folder. This operation works on the selected files only, so you need to select the files of interest - usually that means all of them.

If you want to export the list of files *and* the actions (modified, added, deleted) as well, you can do that using the keyboard shortcuts **Ctrl-A** to select all entries and **Ctrl-C** to copy the detailed list to the clipboard.

The button at the top allows you to change the direction of comparison. You can show the changes need to get from A to B, or if you prefer, from B to A.

The buttons with the revision numbers on can be used to change to a different revision range. When you change the range, the list of items which differ between the two revisions will be updated automatically.

If the list of filenames is very long, you can use the search box to reduce the list to filenames containing specific text. Note that a simple text search is used, so if you want to restrict the list to C source files you should enter `.c` rather than `*.c`.

5.10.4. Diffing Images Using TortoiseIDiff

There are many tools available for diffing text files, including our own TortoiseMerge, but we often find ourselves wanting to see how an image file has changed too. That's why we created TortoiseIDiff.
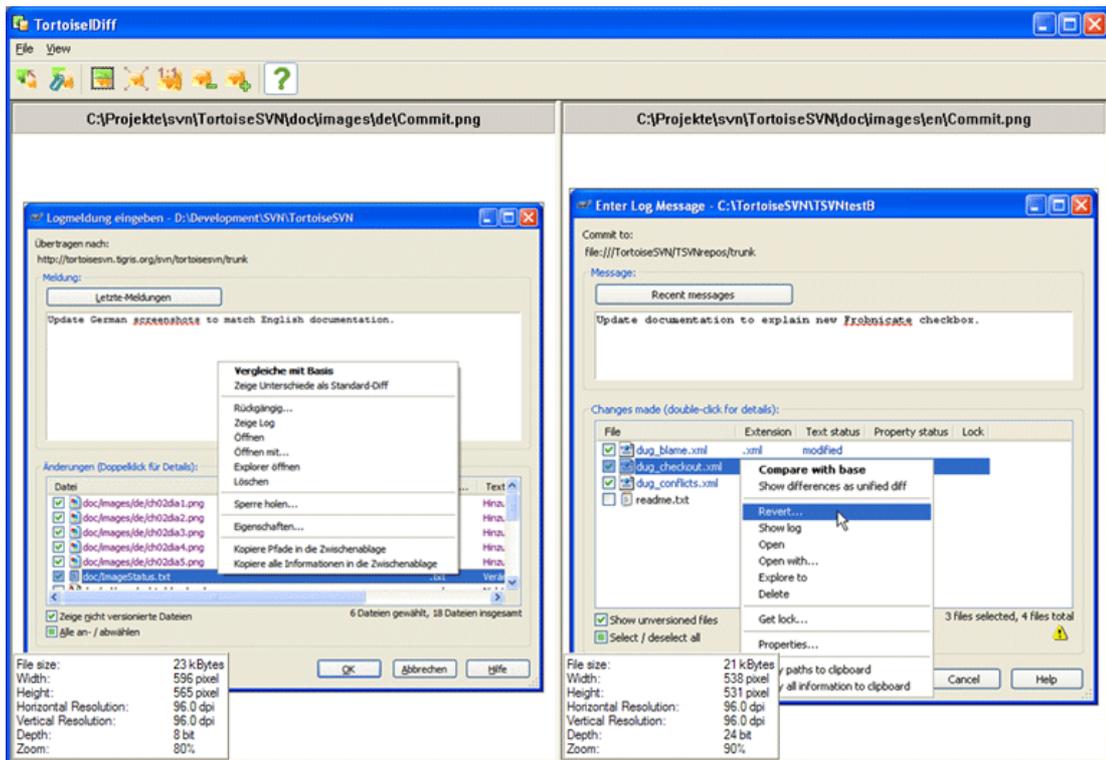
Figure 5.24. The image difference viewer

TortoiseSVN → Diff for any of the common image file formats will start TortoiseIDiff to show image differences. By default the images are displayed side-by-side but you can use the View menu or toolbar to switch to a top-bottom view instead, or if you prefer, you can overlay the images and pretend you are using a lightbox.

Naturally you can also zoom in and out and pan around the image. You can also pan the image simply by left-dragging it. If you select the Link images together option, then the pan controls (scrollbars, mousewheel) on both images are linked.

An image info box shows details about the image file, such as the size in pixels, resolution and colour depth. If this box gets in the way, use View → Image Info to hide it. You can get the same information in a tooltip if you hover the mouse over the image title bar.

When the images are overlaid, the relative intensity of the images (alpha blend) is controlled by a slider control at the left side. You can click anywhere in the slider to set the blend directly, or you can drag the slider to change the blend interactively. **Ctrl+Shift**-Wheel to change the blend.

The button above the slider toggles between two preset blends, indicated by the markers on either side of the blend slider. By default one is at the top and the other at the bottom, so the toggle button just switches between one image and the other. You can move the markers to choose the two blend values that the toggle button will use.

Sometimes you want to see a difference rather than a blend. You might have the image files for two revisions of a printed circuit board and want to see which tracks have changed. If you disable alpha blend mode, the difference will be shown as an *XOR* of the pixel colour values. Unchanged areas will be plain white and changes will be coloured.

5.10.5. External Diff/Merge Tools

If the tools we provide don't do what you need, try one of the many open-source or commercial programs available. Everyone has their own favourites, and this list is by no means complete, but here are a few that you might consider:

WinMerge

WinMerge [http://winmerge.sourceforge.net/] is a great open-source diff tool which can also handle directories.

Perforce Merge

Perforce is a commercial RCS, but you can download the diff/merge tool for free. Get more information from *Perforce* [http://www.perforce.com/perforce/products/merge.html].

KDiff3

KDiff3 is a free diff tool which can also handle directories. You can download it from *here* [http://kdiff3.sf.net/].

ExamDiff

ExamDiff Standard is freeware. It can handle files but not directories. ExamDiff Pro is shareware and adds a number of goodies including directory diff and editing capability. In both flavours, version 3.2 and above can handle unicode. You can download them from *PrestoSoft* [http://www.prestosoft.com/].

Beyond Compare

Similar to ExamDiff Pro, this is an excellent shareware diff tool which can handle directory diffs and unicode. Download it from *Scooter Software* [http://www.scootersoftware.com/].

Araxis Merge

Araxis Merge is a useful commercial tool for diff and merging both files and folders. It does three-way comparison in merges and has synchronization links to use if you've changed the order of functions. Download it from *Araxis* [http://www.araxis.com/merge/index.html].

SciTE

This text editor includes syntax colouring for unified diffs, making them much easier to read. Download it from *Scintilla* [http://www.scintilla.org/SciTEDownload.html].

Notepad2

Notepad2 is designed as a replacement for the standard Windows Notepad program, and is based on the Scintilla open-source edit control. As well as being good for viewing unified diffs, it is much better than the Windows notepad for most jobs. Download it for free *here* [http://www.flos-freeware.ch/notepad2.html].

Read Section 5.29.4, "External Program Settings" for information on how to set up TortoiseSVN to use these tools.

5.11. Adding New Files And Directories
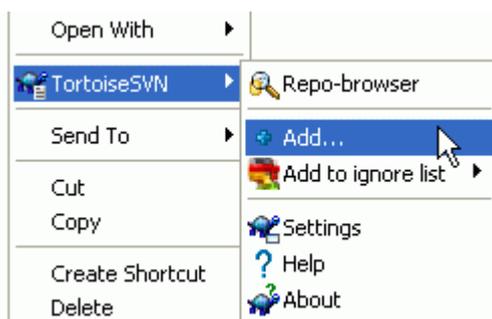
Figure 5.25. Explorer context menu for unversioned files

If you created new files and/or directories during your development process then you need to add them to source control too. Select the file(s) and/or directory and use TortoiseSVN → Add.

After you added the files/directories to source control the file appears with a `added` icon overlay which means you first have to commit your working copy to make those files/directories available to other developers. Adding a file/directory does *not* affect the repository!

Many Adds

You can also use the Add command on already versioned folders. In that case, the add dialog will show you all unversioned files inside that versioned folder. This helps if you have many new files and need to add them all at once.

To add files from outside your working copy you can use the drag-and-drop handler:

1. select the files you want to add

2. right-drag them to the new location inside the working copy

3. release the right mouse button

4. select Context Menu → SVN Add files to this WC. The files will then be copied to the working copy and added to version control.

You can also add files within a working copy simply by left-dragging and dropping them onto the commit dialog.

If you add a file or folder by mistake, you can undo the addition before you commit using TortoiseSVN → Undo add....

5.12. Copying/Moving/Renaming Files and Folders

It often happens that you already have the files you need in another project in your repository, and you simply want to copy them across. You could simply copy the files and add them as described above, but that would not give you any history. And if you subsequently fix a bug in the original files, you can only merge the fix automatically if the new copy is related to the original in Subversion.

The easiest way to copy files and folders from within a working copy is to use the right-drag menu. When you right-drag a file or folder from one working copy to another, or even within the same folder, a context menu appears when you release the mouse.

Figure 5.26. Right drag menu for a directory under version control

Now you can copy existing versioned content to a new location, possibly renaming it at the same time.

You can copy files and folders from your working copy to another location in the repository using TortoiseSVN → Branch/Tag. Refer to Section 5.18.1, "Creating a Branch or Tag" to find out more.

You can locate an older version of a file or folder in the log dialog and copy it to a new location in the repository directly from the log dialog using Context menu → Create branch/tag from revision. Refer to Section 5.9.3, "Getting Additional Information" to find out more.

You can also use the repository browser to locate content you want, and copy it into your working copy directly from the repository, or copy between two locations within the repository. Refer to Section 5.23, "The Repository Browser" to find out more.

Cannot copy between repositories

Whilst you can copy and files and folders *within* a repository, you *cannot* copy or move from one repository to another while preserving history using TortoiseSVN. Not even if the repositories live on the same server. All you can do is copy the content in its current state and add it as new content to the second repository.

If you are uncertain whether two URLs on the same server refer to the same or different repositories, use the repo browser to open one URL and find out where the repository root is. If you can see both locations in one repo browser window then they are in the same repository.

5.13. Ignoring Files And Directories
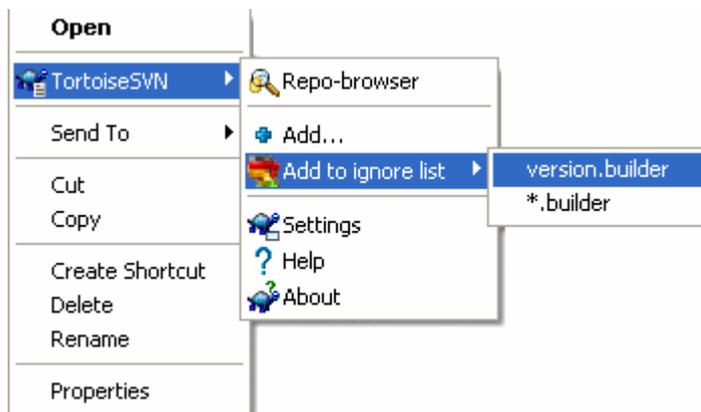
Figure 5.27. Explorer context menu for unversioned files

In most projects you will have files and folders that should not be subject to version control. These might include files created by the compiler, `*.obj`, `*.lst`, maybe an output folder used to store the executable. Whenever you commit changes, TortoiseSVN shows your unversioned files, which fills up the file list in the commit dialog. Of course you can turn off this display, but then you might forget to add a new source file.

The best way to avoid these problems is to add the derived files to the project's ignore list. That way they will never show up in the commit dialog, but genuine unversioned source files will still be flagged up.

If you right click on a single unversioned file, and select the command TortoiseSVN → Add to Ignore List from the context menu, a submenu appears allowing you to select just that file, or all files with the same extension. If you select multiple files, there is no submenu and you can only add those specific files/folders.

If you want to remove one or more items from the ignore list, right click on those items and select TortoiseSVN → Remove from Ignore List You can also access a folder's `svn:ignore` property directly. That allows you to specify more general patterns using filename globbing, described in the section below. Read Section 5.17, "Project Settings" for more information on setting properties directly. Please be aware that each ignore pattern has to be placed on a separate line. Separating them by spaces does not work.

The Global Ignore List

Another way to ignore files is to add them to the *global ignore list*. The big difference here is that the global ignore list is a client property. It applies to *all* Subversion projects, but on the client PC only. In general it is better to use the `svn:ignore` property where possible, because it can be applied to specific project areas, and it works for everyone who checks out the project. Read Section 5.29.1, "General Settings" for more information.

Ignoring Versioned Items

Versioned files and folders can never be ignored - that's a feature of Subversion. If you versioned a file by mistake, read Section B.8, "Ignore files which are already versioned" for instructions on how to "unversion" it.

5.13.1. Pattern Matching in Ignore Lists

Subversion's ignore patterns make use of filename globbing, a technique originally used in Unix to specify files using meta-characters as wildcards. The following characters have special meaning:

*

Matches any string of characters, including the empty string (no characters).

?

Matches any single character.

[...]

Matches any one of the characters enclosed in the square brackets. Within the brackets, a pair of characters separated by "-" matches any character lexically between the two. For example `[AGm-p]` matches any one of A, G, m, n, o or p.

Subversion uses the / as the path delimiter in all internal pathnames, and all pattern matching is done against this style of path names. If you want to use a path delimiter in your ignore pattern, be sure to use /, and not the Windows backslash.

Pattern matching is case sensitive, which can cause problems on Windows. You can force case insensitivity the hard way by pairing characters, eg. to ignore `*.tmp` regardless of case, you could use a pattern like `*.[Tt][Mm][Pp]`.

Subversion uses this pattern matching against every path presented to it for action. These paths are generally relative to the directory being acted upon for import, add, commit, etc. The matching pattern therefore has to take account of the fact that there may or may not be path components before the filename.

If directory names are present in a path, the matching algorithm will not trim them off, so pattern `Fred.*` will match `Fred.c` but not `subdir/Fred.c`. This is significant if you add a folder which contains some files that you want to be ignored, because those filenames will be preceded with the folder name when Subversion compares them with the ignore pattern.

The / character is not treated in any special way for pattern matching purposes, so the pattern `abc*xyz` would match `abcdxyz` but also `abcdir/subdir/anything/morexyz`.

To ignore all `CVS` folders you should either specify a pattern of `*CVS` or better, the pair `CVS */CVS`. The first option works, but would also exclude something called `ThisIsNotCVS`. Using `*/CVS` alone will not work on an immediate child `CVS` folder, and `CVS` alone will not work on sub-folders.

If you want an official definition for globbing, you can find it in the IEEE specifications for the shell command language *Pattern Matching Notation* [http://www.opengroup.org/onlinepubs/009695399/utilities/xcu_chap02.html#tag_02_13].

5.14. Deleting, Moving and Renaming

Unlike CVS, Subversion allows renaming and moving of files and folders. So there are menu entries for delete and rename in the TortoiseSVN submenu.
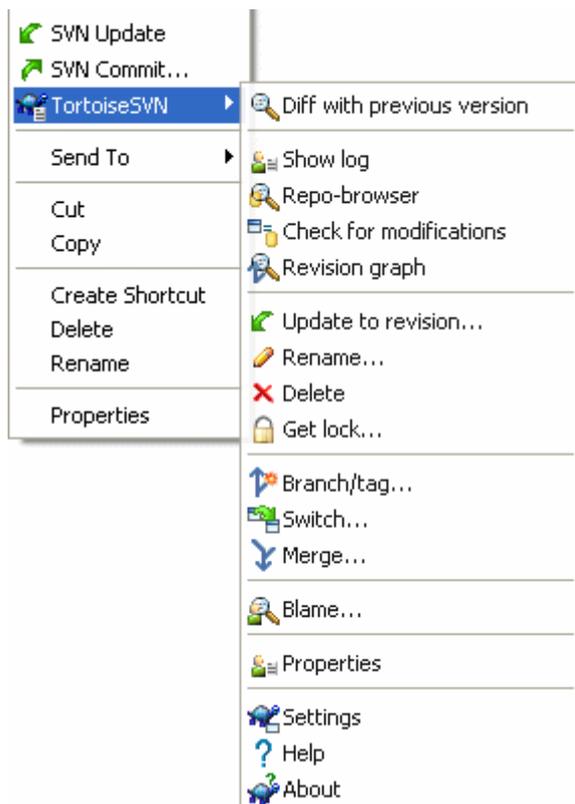
Figure 5.28. Explorer context menu for versioned files

5.14.1. Deleting files and folders

Use TortoiseSVN → Delete to remove files or folders from subversion.

When you TortoiseSVN → Delete a file, it is removed from your working copy immediately as well as being marked for deletion in the repository on next commit. The file's parent folder shows a "deleted" icon overlay. Up until you commit the change, you can get the file back using TortoiseSVN → Revert on the parent folder.

When you TortoiseSVN → Delete a folder, it remains in your working copy, but the overlay changes to indicate that it is marked for deletion. Up until you commit the change, you can get the folder back using TortoiseSVN → Revert on the folder itself. This difference in behaviour between files and folders is a part of Subversion, not TortoiseSVN.

If you want to delete an item from the repository, but keep it locally as an unversioned file/folder, use Extended Context Menu → Delete (keep local). You have to hold the **Shift** key while right clicking on the item in the explorer list pane (right pane) in order to see this in the extended context menu.

If a *file* is deleted via the explorer instead of using the TortoiseSVN context menu, the commit dialog shows those files and lets you remove them from version control too before the commit. However, if you update your working copy, Subversion will spot the missing file and replace it with the latest version from the repository. If you need to delete a version-controlled file, always use TortoiseSVN → Delete so that Subversion doesn't have to guess what you really want to do.

If a *folder* is deleted via the explorer instead of using the TortoiseSVN context menu, your working copy will be broken and you will be unable to commit. If you update your working copy, Subversion will replace the missing folder with the latest version from the repository and you can then delete it the correct way using TortoiseSVN → Delete.

Getting a deleted file or folder back

If you have deleted a file or a folder and already committed that delete operation to the repository, then a normal TortoiseSVN → Revert can't bring it back anymore. But the file or folder is not lost at all. If you know the revision the file or folder got deleted (if you don't, use the log dialog to find out) open the repository browser and switch to that revision. Then select the file or folder you deleted, right-click and select Context Menu → Copy to... as the target for that copy operation select the path to your working copy.

5.14.2. Moving files and folders

If you want to do a simple in-place rename of a file or folder, use Context Menu → Rename... Enter the new name for the item and you're done.

If you want to move files around inside your working copy, perhaps to a different sub-folder, use the right-mouse drag-and-drop handler:

1. select the files or directories you want to move

2. right-drag them to the new location inside the working copy

3. release the right mouse button

4. in the popup menu select Context Menu → SVN Move versioned files here

Commit the parent folder

Since renames and moves are done as a delete followed by an add you must commit the parent folder of the renamed/moved file so that the deleted part of the rename/move will show up in the commit dialog. If you don't commit the removed part of the rename/move, it will stay behind in the repository and when your co-workers update, the old file will not be removed. i.e. they will have *both* the old and the new copies.

You *must* commit a folder rename before changing any of the files inside the folder, otherwise your working copy can get really messed up.

You can also use the repository browser to move items around. Read Section 5.23, "The Repository Browser" to find out more.

Do Not SVN Move Externals

You should *not* use the TortoiseSVN Move or Rename commands on a folder which has been created using `svn:externals`. This action would cause the external item to be deleted from its parent repository, probably upsetting many other people. If you need to move an externals folder you should use an ordinary shell move, then adjust the `svn:externals` properties of the source and destination parent folders.

5.14.3. Changing case in a filename

Making case-only changes to a filename is tricky with Subversion on Windows, because for a short time during a rename, both filenames have to exist. As Windows has a case-insensitive file system, this does not work using the usual Rename command.

Fortunately there are (at least) two possible methods to rename a file without losing its log history. It is important to rename it within subversion. Just renaming in the explorer will corrupt your working copy!

Solution A) (recommended)

1. Commit the changes in your working copy.

2. Rename the file from UPPERcase to upperCASE directly in the repository using the repository browser.

3. Update your working copy.

Solution B)

1. Rename from UPPERcase to UPPERcase_ with the rename command in the TortoiseSVN submenu.

2. Commit the changes.

3. Rename from UPPERcase_ to upperCASE.

4. Commit the changes.

5.14.4. Dealing with filename case conflicts

If the repository already contains two files with the same name but differing only in case (e.g. `TEST.TXT` and `test.txt`), you will not be able to update or checkout the parent directory on a Windows client. Whilst Subversion supports case-sensitive filenames, Windows does not.

This sometimes happens when two people commit, from separate working copies, files which happen to have the same name, but with a case difference. It can also happen when files are committed from a system with a case-sensitive file system, like Linux.

In that case, you have to decide which one of them you want to keep and delete (or rename) the other one from the repository.

Preventing two files with the same name

There is a server hook script available at: *http://svn.collab.net/repos/svn/trunk/contrib/hook-scripts/* [http://svn.collab.net/repos/svn/trunk/contrib/hook-scripts/] that will prevent checkins which result in case conflicts.

5.14.5. Repairing File Renames

Sometimes your friendly IDE will rename files for you as part of a refactoring exercise, and of course it doesn't tell Subversion. If you try to commit your changes, Subversion will see the old filename as

missing and the new one as an unversioned file. You could just check the new filename to get it added in, but you would then lose the history tracing, as Subversion does not know the files are related.

A better way is to notify Subversion that this change is actually a rename, and you can do this within the Commit and Check for Modifications dialogs. Simply select both the old name (missing) and the new name (unversioned) and use Context Menu → Repair Move to pair the two files as a rename.

5.14.6. Deleting Unversioned Files

Usually you set your ignore list such that all generated files are ignored in Subversion. But what if you want to clear all those ignored items to produce a clean build? Usually you would set that in your makefile, but if you are debugging the makefile, or changing the build system it is useful to have a way of clearing the decks.

TortoiseSVN provides just such an option using Extended Context Menu → Delete unversioned items.... You have to hold the **Shift** while right clicking on a folder in the explorer list pane (right pane) in order to see this in the extended context menu. This will produce a dialog which lists all unversioned files anywhere in your working copy. You can then select or deselect items to be removed.

When such items are deleted, the recycle bin is used, so if you make a mistake here and delete a file that should have been versioned, you can still recover it.

5.15. Undo Changes

If you want to undo all changes you made in a file since the last update you need to select the file, right click to pop up the context menu and then select the command TortoiseSVN → Revert A dialog will pop up showing you the files that you've changed and can revert. Select those you want to revert and click on OK.
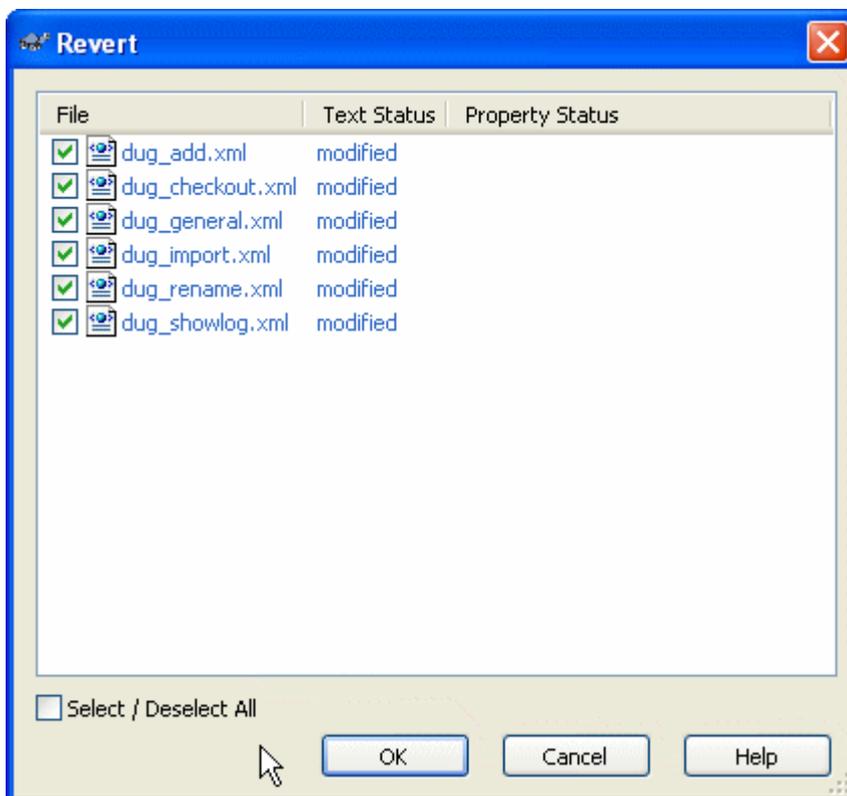
Figure 5.29. Revert dialog

If you want to undo a deletion or a rename, you need to use Revert on the parent folder as the deleted item does not exist for you to right-click on.

If you want to undo the addition of an item, this appears in the context menu as TortoiseSVN → Undo add.... This is really a revert as well, but the name has been changed to make it more obvious.

The columns in this dialog can be customized in the same way as the columns in the Check for modifications dialog. Read Section 5.7.3, "Local and Remote Status" for further details.

Undoing Changes which have been Committed

Revert will only undo your local changes. It does *not* undo any changes which have already been committed. If you want to undo all the changes which were committed in a particular revision, read Section 5.9, "Revision Log Dialog" for further information.

Revert is Slow

When you revert changes you may find that the operation takes a lot longer than you expect. This is because the modified version of the file is sent to the recycle bin, so you can retrieve your changes if you reverted by mistake. However, if your recycle bin is full, Windows takes a long time to find a place to put the file. The solution is simple: either empty the recycle bin or deactivate the Use recycle bin when reverting box in TortoiseSVN's settings.

5.16. Cleanup

If a Subversion command cannot complete successfully, perhaps due to server problems, your working copy can be left in an inconsistent state. In that case you need to use TortoiseSVN → Cleanup on the folder. It is a good idea to do this at the top level of the working copy.

Cleanup has another useful side effect. If a file date changes but its content doesn't, Subversion cannot tell whether it has really changed except by doing a byte-by-byte comparison with the pristine copy. If you have a lot of files in this state it makes acquiring status very slow, which will make many dialogs slow to respond. Executing a Cleanup on your working copy will repair these "broken" timestamps and restore status checks to full speed.

Use Commit Timestamps

Some earlier releases of Subversion were affected by a bug which caused timestamp mismatch when you check out with the Use commit timestamps option checked. Use the Cleanup command to speed up these working copies.

5.17. Project Settings
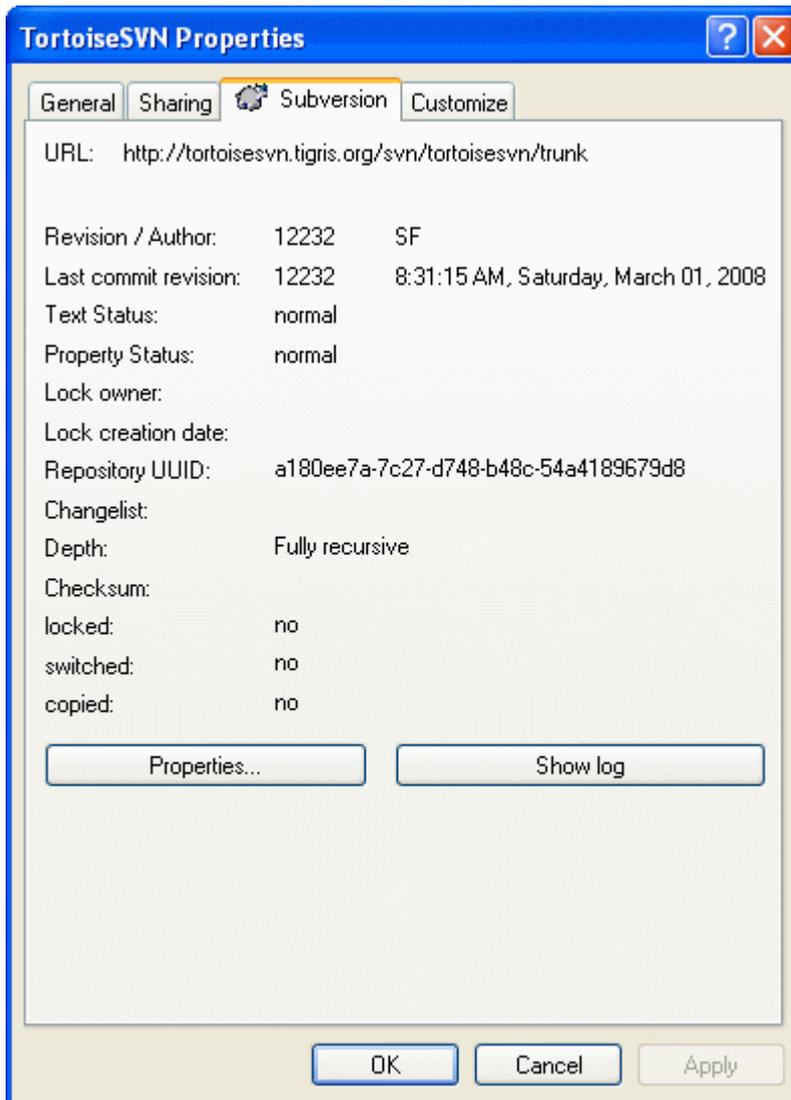
Figure 5.30. Explorer property page, Subversion tab

Sometimes you want to have more detailed information about a file/directory than just the icon overlay. You can get all the information Subversion provides in the explorer properties dialog. Just select the file or directory and select Windows Menu → properties in the context menu (note: this is the normal properties menu entry the explorer provides, not the one in the TortoiseSVN submenu!). In the properties dialog box TortoiseSVN has added a new property page for files/folders under Subversion control, where you can see all relevant information about the selected file/directory.

5.17.1. Subversion Properties
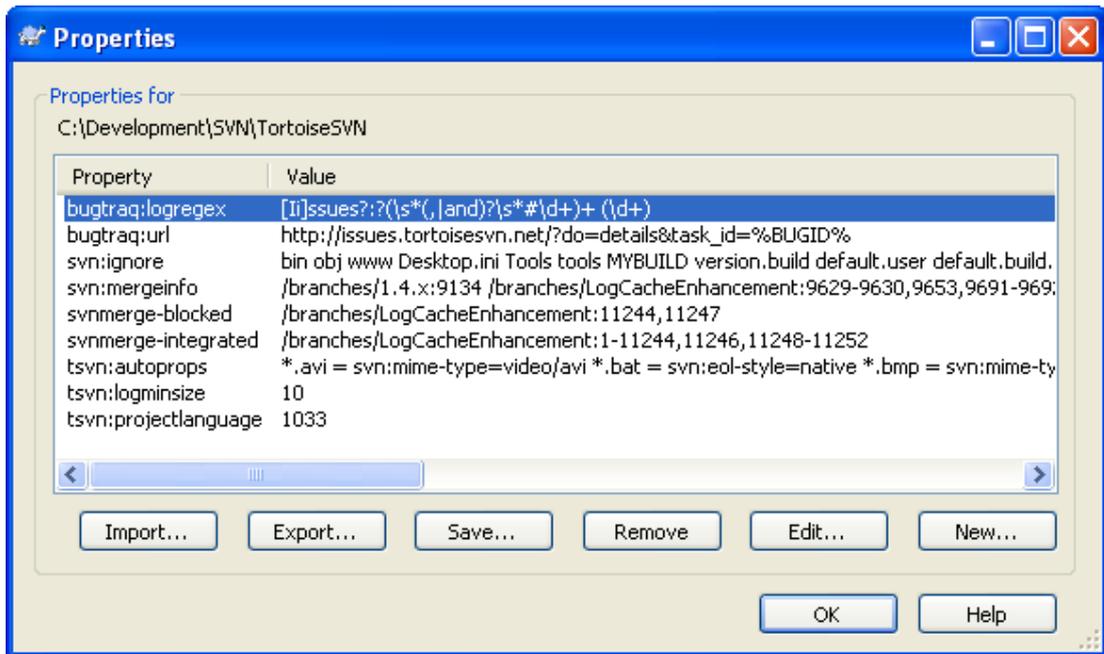
Figure 5.31. Subversion property page

You can read and set the Subversion properties from the Windows properties dialog, but also from TortoiseSVN → properties and within TortoiseSVN's status lists, from Context menu → properties.

You can add your own properties, or some properties with a special meaning in Subversion. These begin with `svn:`. `svn:externals` is such a property; see how to handle externals in Section 5.2.4, "Referenced Projects".

5.17.1.1. svn:keywords

Subversion supports CVS-like keyword expansion which can be used to embed filename and revision information within the file itself. Keywords currently supported are:

$Date$
> Date of last known commit. This is based on information obtained when you update your working copy. It does *not* check the repository to find more recent changes.

$Revision$
> Revision of last known commit.

$Author$
> Author who made the last known commit.

$HeadURL$
> The full URL of this file in the repository.

Id
> A compressed combination of the previous four keywords.

To find out how to use these keywords, look at the *svn:keywords section* [http://svnbook.red-bean.com/en/1.4/svn.advanced.props.special.keywords.html] in the Subversion book, which gives a full description of these keywords and how to enable and use them.

For more information about properties in Subversion see the *Special Properties* [http://svnbook.red-bean.com/en/1.4/svn.advanced.props.html].

5.17.1.2. Adding and Editing Properties
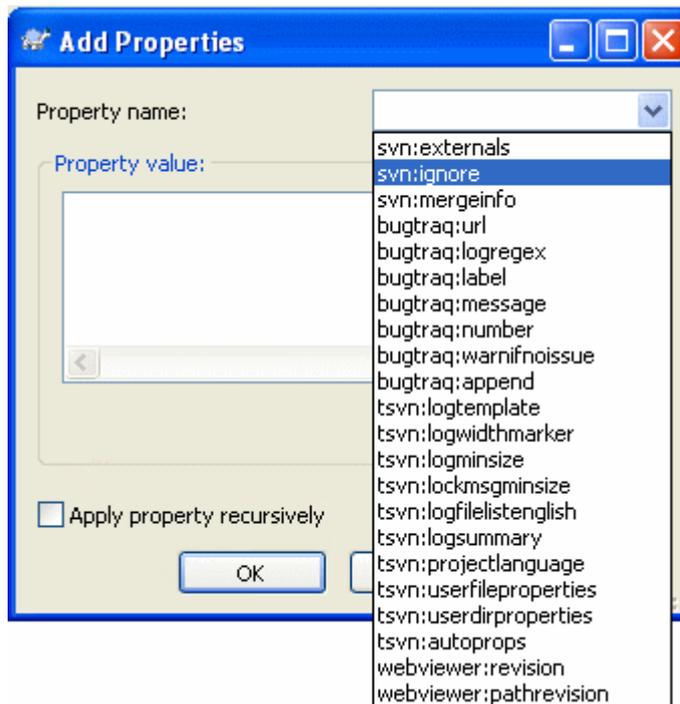
Figure 5.32. Adding properties

To add a new property, first click on Add.... Select the required property name from the combo box, or type in a name of your own choice, then enter a value in the box below. Properties which take multiple values, such as an ignore list, can be entered on multiple lines. Click on OK to add that property to the list.

If you want to apply a property to many items at once, select the files/folders in explorer, then select Context menu → properties

If you want to apply the property to *every* file and folder in the hierarchy below the current folder, check the Recursive checkbox.

Some properties, for example `svn:needs-lock`, can only be applied to files, so the property name doesn't appear in the drop down list for folders. You can still apply such a property recursively to all files in a hierarchy, but you have to type in the property name yourself.

If you wish to edit an existing property, select that property from the list of existing properties, then click on Edit....

If you wish to remove an existing property, select that property from the list of existing properties, then click on Remove.

The `svn:externals` property can be used to pull in other projects from the same repository or a completely different repository. For more information, read Section 5.2.4, "Referenced Projects".

5.17.1.3. Exporting and Importing Properties

Often you will find yourself applying the same set of properties many times, for example `bugtraq:logregex`. To simplify the process of copying properties from one project to another, you can use the Export/Import feature.

From the file or folder where the properties are already set, use TortoiseSVN → properties, select the properties you wish to export and click on Export.... You will be prompted for a filename where the property names and values will be saved.

From the folder(s) where you wish to apply these properties, use TortoiseSVN → properties and click on Import.... You will be prompted for a filename to import from, so navigate to the place you saved the export file previously and select it. The properties will be added to the folders non-recursively.

If you want to add properties to a tree recursively, follow the steps above, then in the property dialog select each property in turn, click on Edit..., check the Apply property recursively box and click on OK.

The Import file format is binary and proprietary to TortoiseSVN. Its only purpose is to transfer properties using Import and Export, so there is no need to edit these files.

5.17.1.4. Binary Properties

TortoiseSVN can handle binary property values using files. To read a binary property value, Save... to a file. To set a binary value, use a hex editor or other appropriate tool to create a file with the content you require, then Load... from that file.

Although binary properties are not often used, they can be useful in some applications. For example if you are storing huge graphics files, or if the application used to load the file is huge, you might want to store a thumbnail as a property so you can obtain a preview quickly.

5.17.1.5. Automatic property setting

You can configure Subversion and TortoiseSVN to set properties automatically on files and folders when they are added to the repository. There are two ways of doing this.

You can edit the subversion configuration file to enable this feature on your client. The General page of TortoiseSVN's settings dialog has an edit button to take you there directly. The config file is a simple text file which controls some of subversion's workings. You need to change two things: firstly in the section headed `miscellany` uncomment the line `enable-auto-props = yes`. Secondly you need to edit the section below to define which properties you want added to which file types. This method is a standard subversion feature and works with any subversion client. However it has to be defined on each client individually - there is no way to propagate these settings from the repository.

An alternative method is to set the `tsvn:autoprops` property on folders, as described in the next section. This method only works for TortoiseSVN clients, but it does get propagated to all working copies on update.

Whichever method you choose, you should note that auto-props are only applied to files at the time they are added to the repository. Auto-props will never change the properties of files which are already versioned.

If you want to be absolutely sure that new files have the correct properties applied, you should set up a repository pre-commit hook to reject commits where the required properties are not set.

Commit properties

Subversion properties are versioned. After you change or add a property you have to commit your changes.

Conflicts on properties

If there's a conflict on committing the changes, because another user has changed the same property, Subversion generates a `.prej` file. Delete this file after you have resolved the conflict.

5.17.2. TortoiseSVN Project Properties

TortoiseSVN has a few special properties of its own, and these begin with `tsvn:`.

- `tsvn:logminsize` sets the minimum length of a log message for a commit. If you enter a shorter message than specified here, the commit is disabled. This feature is very useful for reminding you to supply a proper descriptive message for every commit. If this property is not set, or the value is zero, empty log messages are allowed.

  `tsvn:lockmsgminsize` sets the minimum length of a lock message. If you enter a shorter message than specified here, the lock is disabled. This feature is very useful for reminding you to supply a proper descriptive message for every lock you get. If this property is not set, or the value is zero, empty lock messages are allowed.

- `tsvn:logwidthmarker` is used with projects which require log messages to be formatted with some maximum width (typically 80 characters) before a line break. Setting this property to a non-zero will do 2 things in the log message entry dialog: it places a marker to indicate the maximum width, and it disables word wrap in the display, so that you can see whether the text you entered is too long. Note: this feature will only work correctly if you have a fixed-width font selected for log messages.

- `tsvn:logtemplate` is used with projects which have rules about log message formatting. The property holds a multi-line text string which will be inserted in the commit message box when you start a commit. You can then edit it to include the required information. Note: if you are also using `tsvn:logminsize`, be sure to set the length longer than the template or you will lose the protection mechanism.

- Subversion allows you to set "autoprops" which will be applied to newly added or imported files, based on the file extension. This depends on every client having set appropriate autoprops in their subversion configuration file. `tsvn:autoprops` can be set on folders and these will be merged with the user's local autoprops when importing or adding files. The format is the same as for subversion autoprops, e.g. `*.sh = svn:eol-style=native;svn:executable` sets two properties on files with the `.sh` extension.

  If there is a conflict between the local autoprops and `tsvn:autoprops`, the project settings take precedence because they are specific to that project.

- In the Commit dialog you have the option to paste in the list of changed files, including the status of each file (added, modified, etc). `tsvn:logfilelistenglish` defines whether the file status is inserted in English or in the localized language. If the property is not set, the default is `true`.

- TortoiseSVN can use spell checker modules which are also used by OpenOffice and Mozilla. If you have those installed this property will determine which spell checker to use, i.e. in which language the log messages for your project should be written. `tsvn:projectlanguage` sets the language module the spell checking engine should use when you enter a log message. You can find the values for your language on this page: *MSDN: Language Identifiers* [http://msdn2.microsoft.com/en-us/library/ms776260.aspx].

  You can enter this value in decimal, or in hexadecimal if prefixed with `0x`. For example English (US) can be entered as `0x0409` or `1033`.

- The property `tsvn:logsummary` is used to extract a portion of the log message which is then shown in the log dialog as the log message summary.

  The value of the `tsvn:logsummary` property must be set to a one line regex string which contains one regex group. Whatever matches that group is used as the summary.

  An example: `\[SUMMARY\]:\s+(.*)` Will catch everything after "[SUMMARY]" in the log message and use that as the summary.

- When you want to add a new property, you can either pick one from the list in the combo box, or you can enter any property name you like. If your project uses some custom properties, and you want

those properties to appear in the list in the combo box (to avoid typos when you enter a property name), you can create a list of your custom properties using `tsvn:userfileproperties` and `tsvn:userdirproperties`. Apply these properties to a folder. When you go to edit the properties of any child item, your custom properties will appear in the list of pre-defined property names.

Some `tsvn:` properties require a `true/false` value. TortoiseSVN also understands `yes` as a synonym for `true` and `no` as a synonym for `false`.

TortoiseSVN can integrate with some bug tracking tools. This uses project properties that start with `bugtraq:`. Read Section 5.27, "Integration with Bug Tracking Systems / Issue Trackers" for further information.

It can also integrate with some web-based repository browsers, using project properties that start with `webviewer:`. Read Section 5.28, "Integration with Web-based Repository Viewers" for further information.

> ### Set the project properties on folders
>
> These special project properties must be set on *folders* for the system to work. When you commit a file or folder the properties are read from that folder. If the properties are not found there, TortoiseSVN will search upwards through the folder tree to find them until it comes to an unversioned folder, or the tree root (eg. `C:\`) is found. If you can be sure that each user checks out only from e.g `trunk/` and not some sub-folder, then it is sufficient to set the properties on `trunk/`. If you can't be sure, you should set the properties recursively on each sub-folder. A property setting deeper in the project hierarchy overrides settings on higher levels (closer to `trunk/`).
>
> For project properties *only* you can use the Recursive checkbox to set the property to all sub-folders in the hierarchy, without also setting it on all files.

When you add new sub-folders using TortoiseSVN, any project properties present in the parent folder will automatically be added to the new child folder too.

> ### Caution
>
> Although TortoiseSVN's project properties are extremely useful, they only work with TortoiseSVN, and some will only work in newer versions of TortoiseSVN. If people working on your project use a variety of Subversion clients, or possibly have old versions of TortoiseSVN, you may want to use repository hooks to enforce project policies. project properties can only help to implement a policy, they cannot enforce it.

5.18. Branching / Tagging

One of the features of version control systems is the ability to isolate changes onto a separate line of development. This line is known as a *branch*. Branches are often used to try out new features without disturbing the main line of development with compiler errors and bugs. As soon as the new feature is stable enough then the development branch is *merged* back into the main branch (trunk).

Another feature of version control systems is the ability to mark particular revisions (e.g. a release version), so you can at any time recreate a certain build or environment. This process is known as *tagging*.

Subversion does not have special commands for branching or tagging, but uses so-called "cheap copies" instead. Cheap copies are similar to hard links in Unix, which means that instead of making a complete copy in the repository, an internal link is created, pointing to a specific tree/revision. As a result branches and tags are very quick to create, and take up almost no extra space in the repository.

5.18.1. Creating a Branch or Tag

If you have imported your project with the recommended directory structure, creating a branch or tag version is very simple:
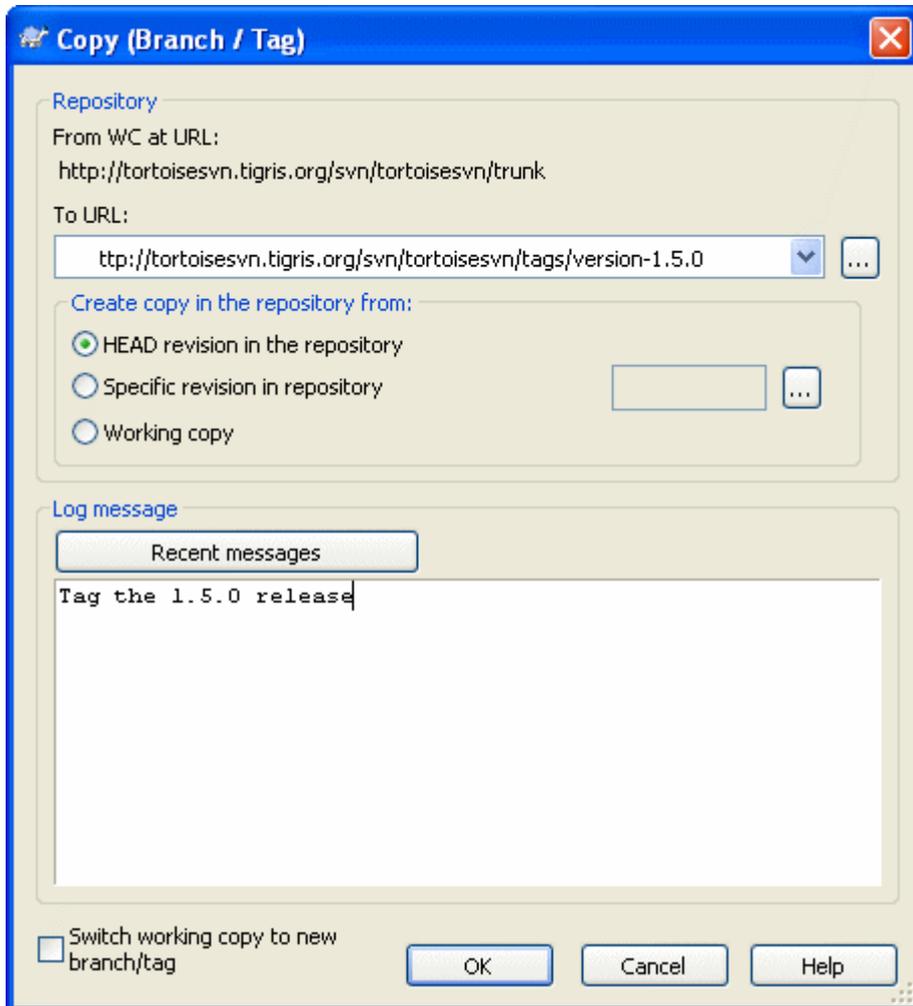
Figure 5.33. The Branch/Tag Dialog

Select the folder in your working copy which you want to copy to a branch or tag, then select the command TortoiseSVN → Branch/Tag....

The default destination URL for the new branch will be the source URL on which your working copy is based. You will need to edit that URL to the new path for your branch/tag. So instead of

```
http://svn.collab.net/repos/ProjectName/trunk
```

you might now use something like

```
http://svn.collab.net/repos/ProjectName/tags/Release_1.10
```

If you can't remember the naming convention you used last time, click the button on the right to open the repository browser so you can view the existing repository structure.

Now you have to select the source of the copy. Here you have three options:

HEAD revision in the repository
> The new branch is copied directly in the repository from the HEAD revision. No data needs to be transferred from your working copy, and the branch is created very quickly.

Specific revision in the repository
> The new branch is copied directly in the repository but you can choose an older revision. This is useful if you forgot to make a tag when you released your project last week. If you can't remember the revision number, click the button on the right to show the revision log, and select the revision number from there. Again no data is transferred from your working copy, and the branch is created very quickly.

Working copy
> The new branch is an identical copy of your local working copy. If you have updated some files to an older revision in your WC, or if you have made local changes, that is exactly what goes into the copy. Naturally this sort of complex tag may involve transferring data from your WC back to the repository if it does not exist there already.

If you want your working copy to be switched to the newly created branch automatically, use the Switch working copy to new branch/tag checkbox. But if you do that, first make sure that your working copy does not contain modifications. If it does, those changes will be merged into the branch WC when you switch.

Press OK to commit the new copy to the repository. Don't forget to supply a log message. Note that the copy is created *inside the repository*.

Note that unless you opted to switch your working copy to the newly created branch, creating a Branch or Tag does *not* affect your working copy. Even if you create the branch from your WC, those changes are committed to the new branch, not to the trunk, so your WC may still be marked as modified with respect to the trunk.

5.18.2. To Checkout or to Switch...

...that is (not really) the question. While a checkout downloads everything from the desired branch in the repository to your working directory, TortoiseSVN → Switch... only transfers the changed data to your working copy. Good for the network load, good for your patience. :-)

To be able to work with your freshly generated branch or tag you have several ways to handle it. You can:

- TortoiseSVN → Checkout to make a fresh checkout in an empty folder. You can check out to any location on your local disk and you can create as many working copies from your repository as you like.

- Switch your current working copy to the newly created copy in the repository. Again select the top level folder of your project and use TortoiseSVN → Switch... from the context menu.

  In the next dialog enter the URL of the branch you just created. Select the Head Revision radio button and click on OK. Your working copy is switched to the new branch/tag.

  Switch works just like Update in that it never discards your local changes. Any changes you have made to your working copy which have not yet been committed will be merged when you do the Switch. If you do not want this to happen then you must either commit the changes before switching, or revert your working copy to an already-committed revision (typically HEAD).

- If you want to work on trunk and branch, but don't want the expense of a fresh checkout, you can use Windows Explorer to make a copy of your trunk checkout in another folder, then TortoiseSVN → Switch... that copy to your new branch.
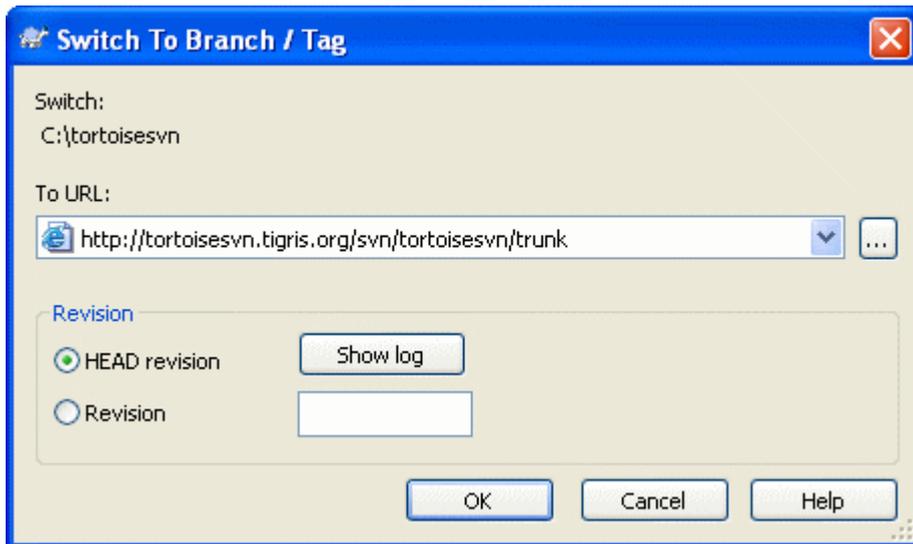
Figure 5.34. The Switch Dialog

Although Subversion itself makes no distinction between tags and branches, the way they are typically used differs a bit.

- Tags are typically used to create a static snapshot of the project at a particular stage. As such they not normally used for development - that's what branches are for, which is the reason we recommended the `/trunk /branches /tags` repository structure in the first place. Working on a tag revision is *not a good idea*, but because your local files are not write protected there is nothing to stop you doing this by mistake. However, if you try to commit to a path in the repository which contains `/tags/`, TortoiseSVN will warn you.

- It may be that you need to make further changes to a release which you have already tagged. The correct way to handle this is to create a new branch from the tag first and commit the branch. Do your Changes on this branch and then create a new tag from this new branch, e.g. `Version_1.0.1`.

- If you modify a working copy created from a branch and commit, then all changes go to the new branch and *not* the trunk. Only the modifications are stored. The rest remains a cheap copy.

5.19. Merging

Where branches are used to maintain separate lines of development, at some stage you will want to merge the changes made on one branch back into the trunk, or vice versa.

It is important to understand how branching and merging works in Subversion before you start using it, as it can become quite complex. It is highly recommended that you read the chapter *Branching and Merging* [http://svnbook.red-bean.com/en/1.4/svn.branchmerge.html] in the Subversion book, which gives a full description and many examples of how it is used.

The next point to note is that merging *always* takes place within a working copy. If you want to merge changes *into* a branch, you have to have a working copy for that branch checked out, and invoke the merge wizard from that working copy using TortoiseSVN → Merge....

In general it is a good idea to perform a merge into an unmodified working copy. If you have made other changes in your WC, commit those first. If the merge does not go as you expect, you may want to revert the changes, and the Revert command will discard *all* changes including any you made before the merge.

There are three common use cases for merging which are handled in slightly different ways, as described below. The first page of the merge wizard asks you to select the method you need.

Merge a range of revisions
> This method covers the case when you have made one or more revisions to a branch (or to the trunk) and you want to port those changes across to a different branch.
>
> What you are asking Subversion to do is this: "Calculate the changes necessary to get [FROM] revision 1 of branch A [TO] revision 7 of branch A, and apply those changes to my working copy (of trunk or branch B)."

Reintegrate a branch
> This method covers the case when you have made a feature branch as discussed in the Subversion book. All trunk changes have been ported to the feature branch, week by week, and now the feature is complete you want to merge it back into the trunk. Because you have kept the feature branch synchronized with the trunk, the latest versions of branch and trunk will be absolutely identical except for your branch changes.
>
> This is a special case of the tree merge described below, and it requires only the URL to merge from (normally) your development branch. It uses the merge-tracking features of Subversion to calculate the correct revision ranges to use, and perform additional checks which ensure that the branch has been fully updated with trunk changes. This ensures that you don't accidentally undo work that others have committed to trunk since you last synchronized changes.
>
> After the merge, all branch development has been completely merged back into the main development line. The branch is now redundant and can be deleted.

Merge two different trees
> This is a more general case of the reintegrate method. What you are asking Subversion to do is: "Calculate the changes necessary to get [FROM] the head revision of the trunk [TO] the head revision of the branch, and apply those changes to my working copy (of the trunk)." The net result is that trunk now looks exactly like the branch.
>
> If your server/repository does not support merge-tracking then this is the only way to merge a branch back to trunk. Another use case occurs when you are using vendor branches and you need to merge the changes following a new vendor drop into your trunk code. For more information read the chapter on *vendor branches* [http://svnbook.red-bean.com/en/1.4/svn.advanced.vendorbr.html] in the Subversion Book.

5.19.1. Merging a Range of Revisions
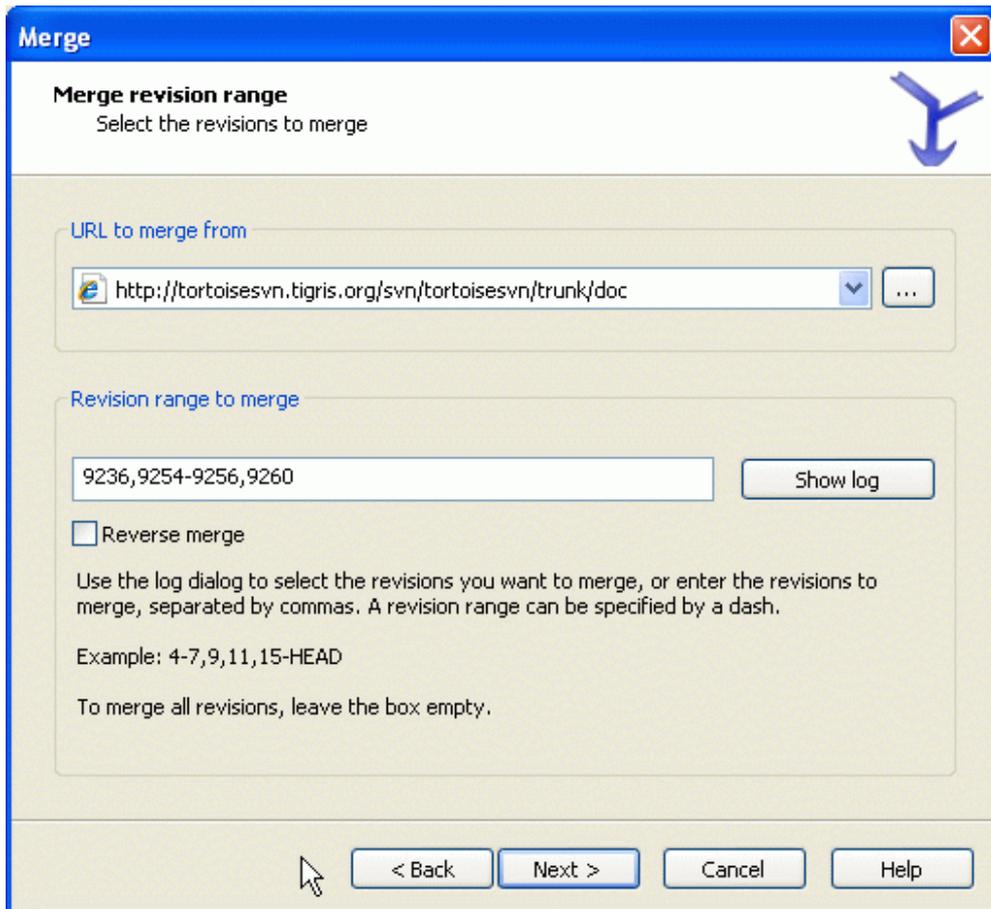
Figure 5.35. The Merge Wizard - Select Revision Range

In the From: field enter the full folder URL of the branch or tag containing the changes you want to port into your working copy. You may also click ... to browse the repository and find the desired branch. If you have merged from this branch before, then just use the drop down list which shows a history of previously used URLs.

In the Revision range to merge field enter the list of revisions you want to merge. This can be a single revision, a list of specific revisions separated by commas, or a range of revisions separated by a dash, or any combination of these.

The easiest way to select the range of revisions you need is to click on Show Log, as this will list recent changes with their log comments. If you want to merge the changes from a single revision, just select that revision. If you want to merge changes from several revisions, then select that range (using the usual **Shift**-modifier). Click on OK and the list of revision numbers to merge will be filled in for you.

If you want to merge changes back *out* of your working copy, to revert a change which has already been committed, select the revisions to revert and make sure the Reverse merge box is checked.

If you have already merged some changes from this branch, hopefully you will have made a note of the last revision merged in the log message when you committed the change. In that case, you can use Show Log for the Working Copy to trace that log message. Use the end point of the last merge as the start point for this merge. For example, if you have merged revisions 37 to 39 last time, then the start point for this merge should be revision 39.

If you are using the merge tracking features of Subversion, you do not need to remember which revisions have already been merged - Subversion will record that for you. If you leave the revision range blank,

all revisions which have not yet been merged will be included. Read Section 5.19.6, "Merge Tracking" to find out more.

If other people may be committing changes then be careful about using the HEAD revision. It may not refer to the revision you think it does if someone else made a commit after your last update.

Click Next and go to Section 5.19.4, "Merge Options"

5.19.2. Reintegrate a branch
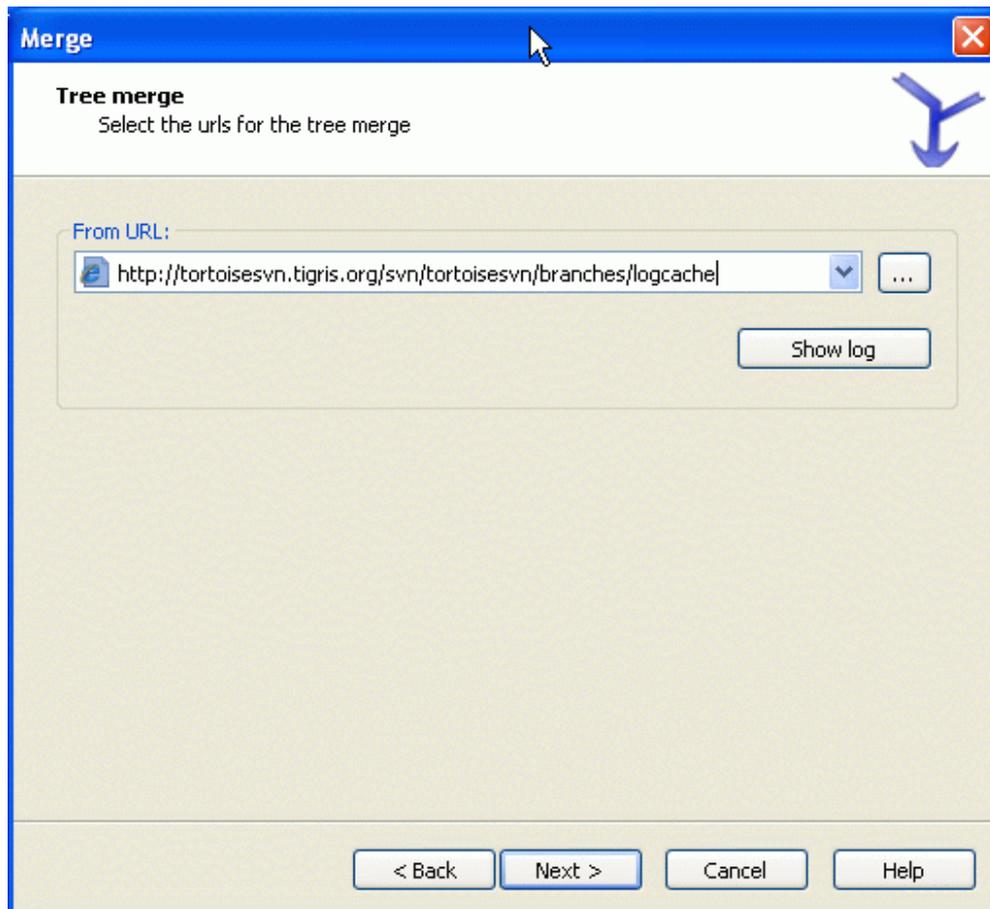
Figure 5.36. The Merge Wizard - Reintegrate Merge

To merge a feature branch back into the trunk you must start the merge wizard from within a working copy of the trunk.

In the From URL: field enter the full folder URL of the branch that you want to merge back. You may also click ... to browse the repository.

There are some conditions which apply to a reintegrate merge. Firstly, the server must support merge tracking. The working copy must be of depth infinite (no sparse checkouts), and it must not have any local modifications, switched items or items that have been updated to revisions other than HEAD. All changes to trunk made during branch development must have been merged across to the branch (or marked as having been merged). The range of revisions to merge will be calculated automatically.

5.19.3. Merging Two Different Trees
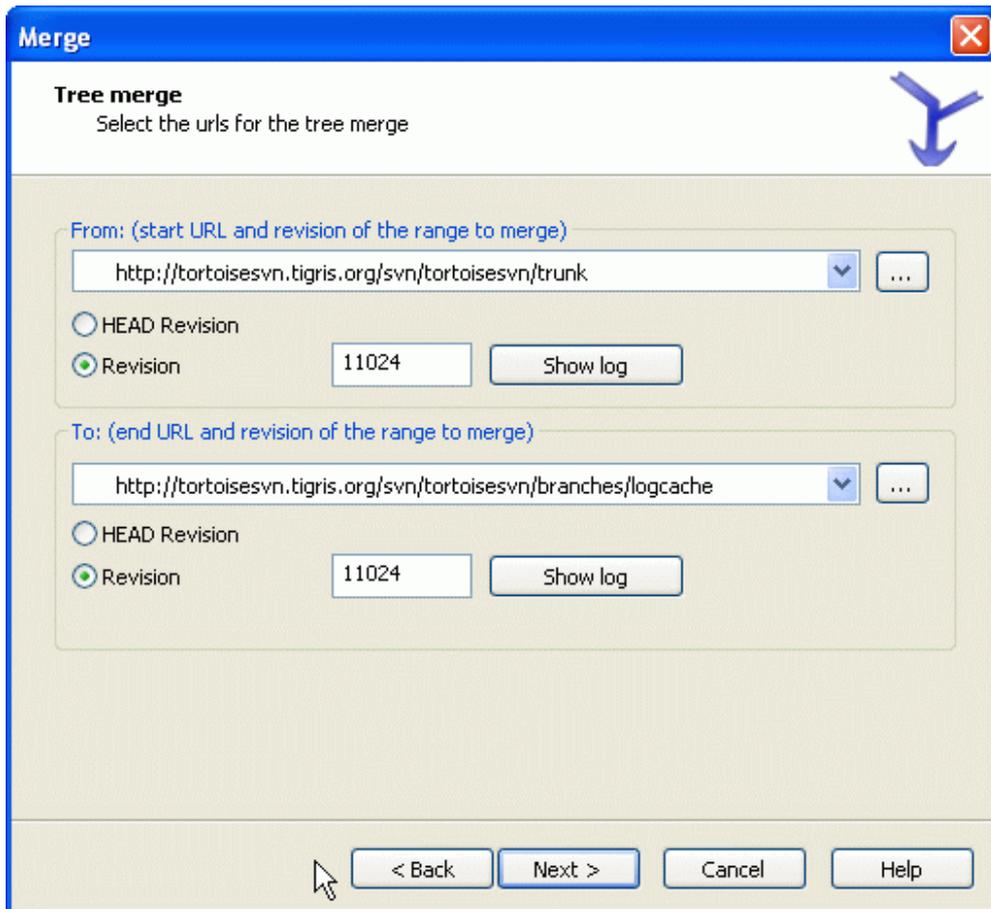
Figure 5.37. The Merge Wizard - Tree Merge

If you are using this method to merge a feature branch back to trunk, you need to start the merge wizard from within a working copy of trunk.

In the From: field enter the full folder URL of the *trunk*. This may sound wrong, but remember that the trunk is the start point to which you want to add the branch changes. You may also click ... to browse the repository.

In the To: field enter the full folder URL of the feature branch.

In both the From Revision field and the To Revision field, enter the last revision number at which the two trees were synchronized. If you are sure no-one else is making commits you can use the HEAD revision in both cases. If there is a chance that someone else may have made a commit since that synchronization, use the specific revision number to avoid losing more recent commits.

You can also use Show Log to select the revision.

5.19.4. Merge Options

This page of the wizard lets you specify advanced options, before starting the merge process. Most of the time you can just use the default settings.

You can specify the depth to use for the merge, i.e. how far down into your working copy the merge should go. The depth terms used are described in Section 5.3.1, "Checkout Depth". The default depth is Working copy, which uses the existing depth setting, and is almost always what you want.

Most of the time you want merge to take account of the file's history, so that changes relative to a common ancestor are merged. Sometimes you may need to merge files which are perhaps related, but not in

your repository. For example you may have imported versions 1 and 2 of a third party library into two separate directories. Although they are logically related, Subversion has no knowledge of this because it only sees the tarballs you imported. If you attempt to merge the difference between these two trees you would see a complete removal followed by a complete add. To make Subversion use only path-based differences rather than history-based differences, check the Ignore ancestry box. Read more about this topic in the Subversion book, *Noticing or Ignoring Ancestry* [http://svnbook.red-bean.com/en/1.4/svn.branchmerge.copychanges.html#svn.branchmerge.copychanges.bestprac.ancestry]

You can specify the way that line ending and whitespace changes are handled. These options are described in Section 5.10.2, "Line-end and Whitespace Options". The default behaviour is to treat all whitespace and line-end differences as real changes to be merged.

If you are using merge tracking and you want to mark a revision as having been merged, without actually doing the merge here, check the Only record the merge checkbox. There are two possible reasons you might want to do this. It may be that the merge is too complicated for the merge algorithms, so you code the changes by hand, then mark the change as merged so that the merge tracking algorithm is aware of it. Or you might want to prevent a particular revision from being merged. Marking it as already merged will prevent the merge occurring with merge-tracking-aware clients.

Now everything is set up, all you have to do is click on the Merge button. If you want to preview the results Test Merge performs the merge operation, but does *not* modify the working copy at all. It shows you a list of the files that will be changed by a real merge, and notes those areas where conflicts will occur.

The merge progress dialog shows each stage of the merge, with the revision ranges involved. This may indicate one more revision than you were expecting. For example if you asked to merge revision 123 the progress dialog will report "Merging revisions 122 through 123". To understand this you need to remember that Merge is closely related to Diff. The merge process works by generating a list of differences between two points in the repository, and applying those differences to your working copy. The progress dialog is simply showing the start and end points for the diff.

5.19.5. Reviewing the Merge Results

The merge is now complete. It's a good idea to have a look at the merge and see if it's as expected. Merging is usually quite complicated. Conflicts often arise if the branch has drifted far from the trunk.

For Subversion clients and servers prior to 1.5, no merge information is stored and merged revisions have to be tracked manually. When you have tested the changes and come to commit this revision, your commit log message should *always* include the revision numbers which have been ported in the merge. If you want to apply another merge at a later time you will need to know what you have already merged, as you do not want to port a change more than once. For more information about this, refer to *Best Practices for Merging* [http://svnbook.red-bean.com/en/1.4/svn.branchmerge.copychanges.html#svn.branchmerge.copychanges.bestprac] in the Subversion book.

If your server and all clients are running Subversion 1.5 or higher, the merge tracking facility will record the revisions merged and avoid a revision being merged more than once. This makes your life much simpler as you can simply merge the entire revision range each time and know that only new revisions will actually be merged.

Branch management is important. If you want to keep this branch up to date with the trunk, you should be sure to merge often so that the branch and trunk do not drift too far apart. Of course, you should still avoid repeated merging of changes, as explained above.

Tip

If you have just merged a feature branch back into the trunk, the trunk now contains all the new feature code, and the branch is obsolete. You can now delete it from the repository if required.

> **Important**
>
> Subversion can't merge a file with a folder and vice versa - only folders to folders and files to files. If you click on a file and open up the merge dialog, then you have to give a path to a file in that dialog. If you select a folder and bring up the dialog, then you must specify a folder URL for the merge.

5.19.6. Merge Tracking

Subversion 1.5 introduced facilities for merge tracking. When you merge changes from one tree into another, the revision numbers merged are stored and this information can be used for several different purposes.

- You can avoid the danger of merging the same revision twice (repeated merge problem). Once a revision is marked as having been merged, future merges which include that revision in the range will skip over it.

  When you merge a branch back into trunk, the log dialog can show you the branch commits as part of the trunk log, giving better traceability of changes.

  When showing blame information for a file, you can choose to show the original author of merged revisions, rather than the person who did the merge.

  You can mark revisions as *do not merge* by including them in the list of merged revisions without actually doing the merge.

Merge tracking information is stored in the `svn:mergeinfo` property by the client when it performs a merge. When the merge is committed the server stores that information in a database, and when you request merge, log or blame information, the server can respond appropriately. For the system to work properly you must ensure that the server, the repository and all clients are upgraded. Earlier clients will not store the `svn:mergeinfo` property and earlier servers will not provide the information requested by new clients.

Find out more about merge tracking from Subversion's *Merge tracking documentation* [http://subversion.tigris.org/merge-tracking/index.html].

5.19.7. Handling Conflicts during Merge

Merging does not always go smoothly. Sometimes there is a conflict, and if you are merging multiple ranges, you generally want to resolve the conflict before merging of the next range starts. TortoiseSVN helps you through this process by showing the *merge conflict callback* dialog.
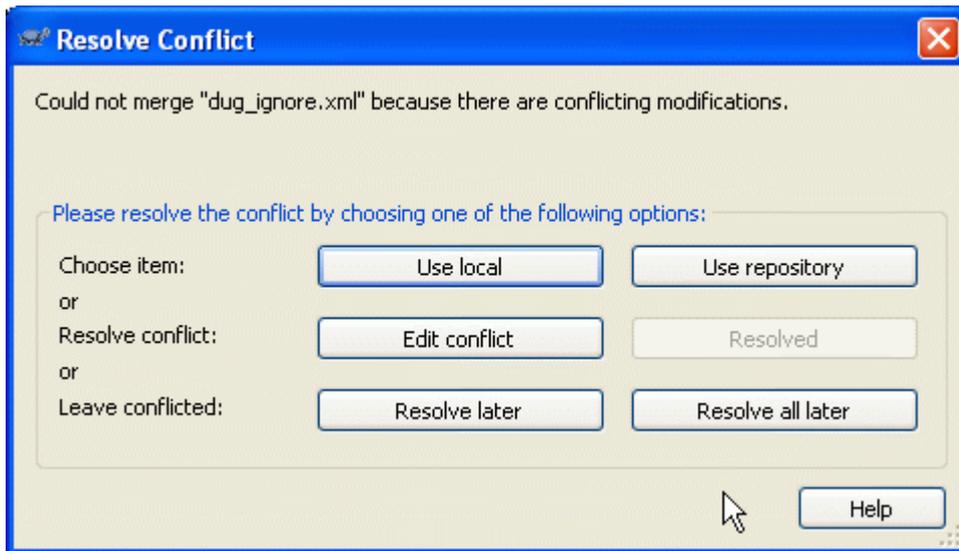
Figure 5.38. The Merge Conflict Callback Dialog

When a conflict occurs during the merge, you have three ways to handle it.

1. You may decide that your local changes are much more important, so you want to discard the version from the repository and keep your local version. Or you might discard your local changes in favour of the repository version. Either way, no attempt is made to merge the changes - you choose one or the other.

2. Normally you will want to look at the conflicts and resolve them. In that case, choose the Edit Conflict which will start up your merge tool. When you are satisfied with the result, click Resolved.

3. The last option is to postpone resolution and continue with merging. You can choose to do that for the current conflicted file, or for all files in the rest of the merge. However, if there are further changes in that file, it will not be possible to complete the merge.

5.19.8. Merge a Completed Branch

If you want to merge all changes from a feature branch back to trunk, then you can use the TortoiseSVN → Merge reintegrate... from the extended context menu (hold down the **Shift** key while you right click on the file).
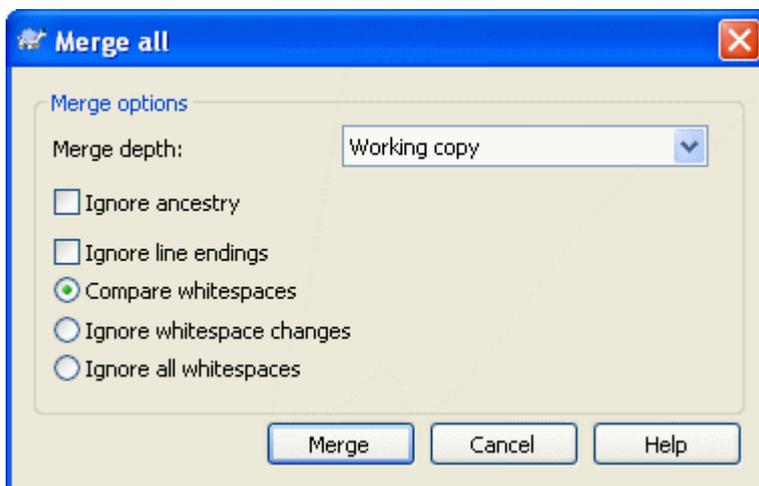
Figure 5.39. The Merge reintegrate Dialog

This dialog is very easy. All you have to do is set the options for the merge, as described in Section 5.19.4, "Merge Options". The rest is done by TortoiseSVN automatically using merge tracking.

5.20. Locking

Subversion generally works best without locking, using the "Copy-Modify-Merge" methods described earlier in Section 2.2.3, "The Copy-Modify-Merge Solution". However there are a few instances when you may need to implement some form of locking policy.

- You are using "unmergeable" files, for example, graphics files. If two people change the same file, merging is not possible, so one of you will lose their changes.

- Your company has always used a locking revision control system in the past and there has been a management decision that "locking is best".

Firstly you need to ensure that your Subversion server is upgraded to at least version 1.2. Earlier versions do not support locking at all. If you are using `file://` access, then of course only your client needs to be updated.

5.20.1. How Locking Works in Subversion

By default, nothing is locked and anyone who has commit access can commit changes to any file at any time. Others will update their working copies periodically and changes in the repository will be merged with local changes.

If you *Get a Lock* on a file, then only you can commit that file. Commits by all other users will be blocked until you release the lock. A locked file cannot be modified in any way in the repository, so it cannot be deleted or renamed either, except by the lock owner.

However, other users will not necessarily know that you have taken out a lock. Unless they check the lock status regularly, the first they will know about it is when their commit fails, which in most cases is not very useful. To make it easier to manage locks, there is a new Subversion property `svn:needs-lock`. When this property is set (to any value) on a file, whenever the file is checked out or updated, the local copy is made read-only *unless* that working copy holds a lock for the file. This acts as a warning that you should not edit that file unless you have first acquired a lock. Files which are versioned and read-only are marked with a special overlay in TortoiseSVN to indicate that you need to acquire a lock before editing.

Locks are recorded by working copy location as well as by owner. If you have several working copies (at home, at work) then you can only hold a lock in *one* of those working copies.

If one of your co-workers acquires a lock and then goes on holiday without releasing it, what do you do? Subversion provides a means to force locks. Releasing a lock held by someone else is referred to as *Breaking* the lock, and forcibly acquiring a lock which someone else already holds is referred to as *Stealing* the lock. Naturally these are not things you should do lightly if you want to remain friends with your co-workers.

Locks are recorded in the repository, and a lock token is created in your local working copy. If there is a discrepancy, for example if someone else has broken the lock, the local lock token becomes invalid. The repository is always the definitive reference.

5.20.2. Getting a Lock

Select the file(s) in your working copy for which you want to acquire a lock, then select the command TortoiseSVN → Get Lock....
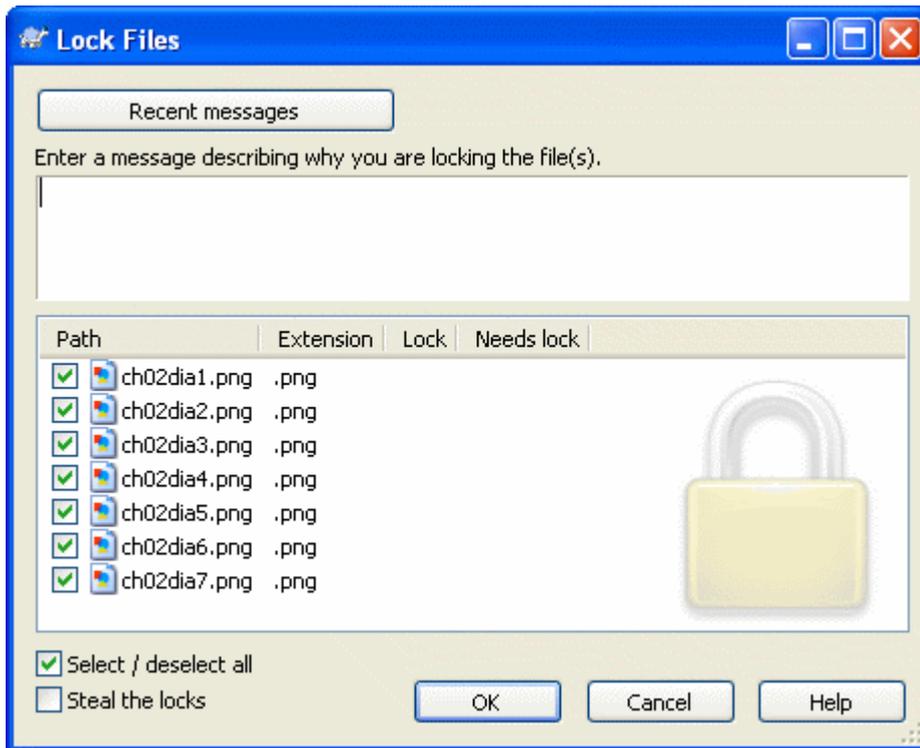
Figure 5.40. The Locking Dialog

A dialog appears, allowing you to enter a comment, so others can see why you have locked the file. The comment is optional and currently only used with Svnserve based repositories. If (and *only* if) you need to steal the lock from someone else, check the Steal lock box, then click on OK.

If you select a folder and then use TortoiseSVN → Get Lock... the lock dialog will open with *every* file in *every* sub-folder selected for locking. If you really want to lock an entire hierarchy, that is the way to do it, but you could become very unpopular with your co-workers if you lock them out of the whole project. Use with care ...

5.20.3. Releasing a Lock

To make sure you don't forget to release a lock you don't need any more, locked files are shown in the commit dialog and selected by default. If you continue with the commit, locks you hold on the selected files are removed, even if the files haven't been modified. If you don't want to release a lock on certain files, you can uncheck them (if they're not modified). If you want to keep a lock on a file you've modified, you have to enable the Keep locks checkbox before you commit your changes.

To release a lock manually, select the file(s) in your working copy for which you want to release the lock, then select the command TortoiseSVN → Release Lock There is nothing further to enter so TortoiseSVN will contact the repository and release the locks. You can also use this command on a folder to release all locks recursively.

5.20.4. Checking Lock Status
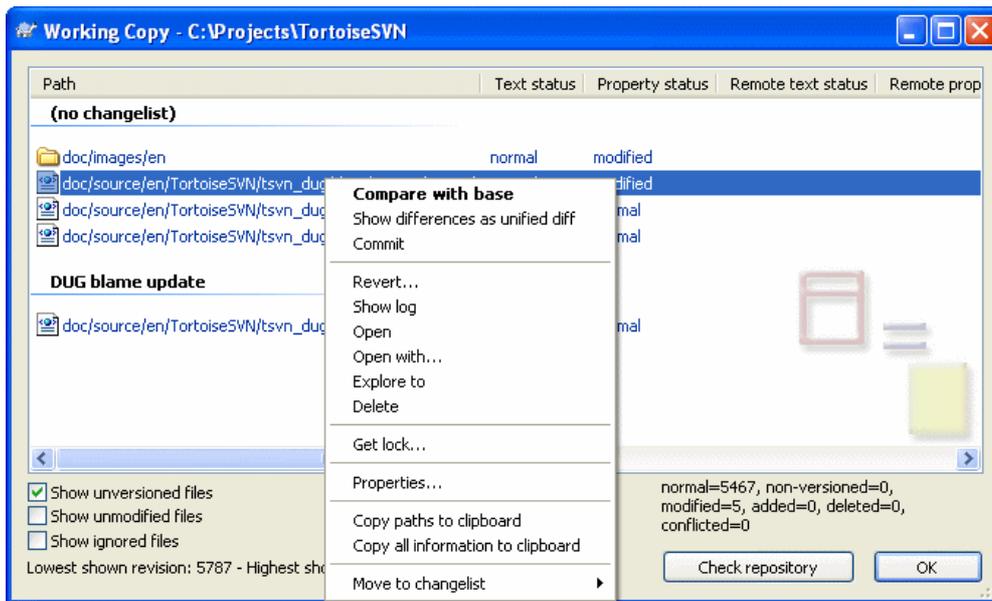
Figure 5.41. The Check for Modifications Dialog

To see what locks you and others hold, you can use TortoiseSVN → Check for Modifications.... Locally held lock tokens show up immediately. To check for locks held by others (and to see if any of your locks are broken or stolen) you need to click on Check Repository.

From the context menu here, you can also get and release locks, as well as breaking and stealing locks held by others.

Avoid Breaking and Stealing Locks

If you break or steal someone else's lock without telling them, you could potentially cause loss of work. If you are working with unmergeable file types and you steal someone else's lock, once you release the lock they are free to check in their changes and overwrite yours. Subversion doesn't lose data, but you have lost the team-working protection that locking gave you.

5.20.5. Making Non-locked Files Read-Only

As mentioned above, the most effective way to use locking is to set the `svn:needs-lock` property on files. Refer to Section 5.17, "Project Settings" for instructions on how to set properties. Files with this property set will always be checked out and updated with the read-only flag set unless your working copy holds a lock.

As a reminder, TortoiseSVN uses a special overlay to indicate this.

If you operate a policy where every file has to be locked then you may find it easier to use Subversion's auto-props feature to set the property automatically every time you add new files. Read Section 5.17.1.5, "Automatic property setting" for further information.

5.20.6. The Locking Hook Scripts

When you create a new repository with Subversion 1.2 or higher, four hook templates are created in the repository `hooks` directory. These are called before and after getting a lock, and before and after releasing a lock.

It is a good idea to install a `post-lock` and `post-unlock` hook script on the server which sends out an email indicating the file which has been locked. With such a script in place, all your users can be notified if someone locks/unlocks a file. You can find an example hook script `hooks/post-lock.tmpl` in your repository folder.

You might also use hooks to disallow breaking or stealing of locks, or perhaps limit it to a named administrator. Or maybe you want to email the owner when one of their locks is broken or stolen.

Read Section 4.3, "Hook Scripts" to find out more.

5.21. Creating and Applying Patches

For open source projects (like this one) everyone has read access to the repository, and anyone can make a contribution to the project. So how are those contributions controlled? If just anyone could commit changes, the project would be permanently unstable and probably permanently broken. In this situation the change is managed by submitting a *patch* file to the development team, who do have write access. They can review the patch first, and then either submit it to the repository or reject it back to the author.

Patch files are simply Unified-Diff files showing the differences between your working copy and the base revision.

5.21.1. Creating a Patch File

First you need to make *and test* your changes. Then instead of using TortoiseSVN → Commit... on the parent folder, you select TortoiseSVN → Create Patch...
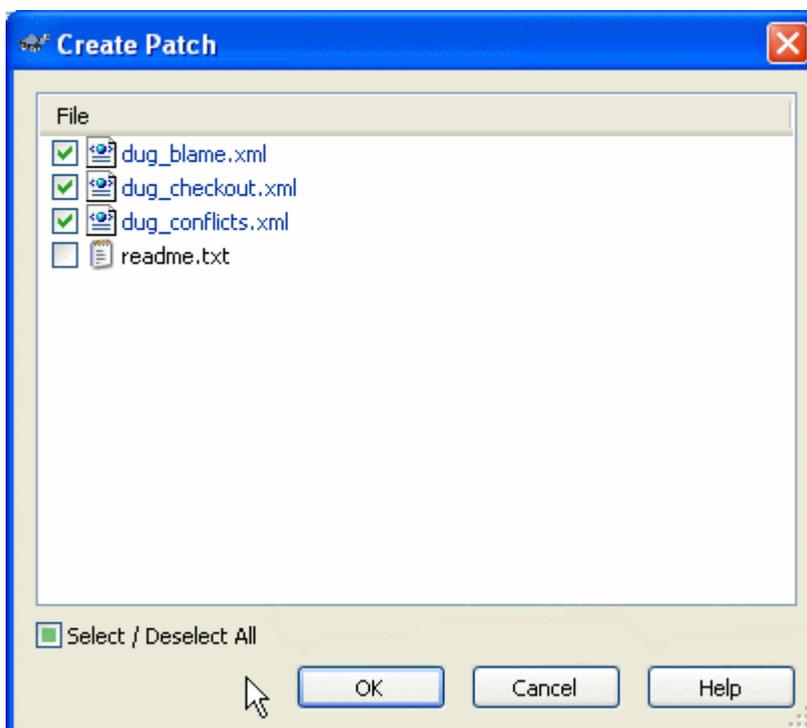
Figure 5.42. The Create Patch dialog

you can now select the files you want included in the patch, just as you would with a full commit. This will produce a single file containing a summary of all the changes you have made to the selected files since the last update from the repository.

The columns in this dialog can be customized in the same way as the columns in the Check for modifications dialog. Read Section 5.7.3, "Local and Remote Status" for further details.

You can produce separate patches containing changes to different sets of files. Of course, if you create a patch file, make some more changes to the *same* files and then create another patch, the second patch file will include *both* sets of changes.

Just save the file using a filename of your choice. Patch files can have any extension you like, but by convention they should use the .patch or .diff extension. You are now ready to submit your patch file.

You can also save the patch to the clipboard instead of to a file. You might want to do this so that you can paste it into an email for review by others. Or if you have two working copies on one machine and you want to transfer changes from one to the other, a patch on the clipboard is a convenient way of doing this.

5.21.2. Applying a Patch File

Patch files are applied to your working copy. This should be done from the same folder level as was used to create the patch. If you are not sure what this is, just look at the first line of the patch file. For example, if the first file being worked on was doc/source/english/chapter1.xml and the first line in the patch file is Index: english/chapter1.xml then you need to apply the patch to the doc/source/ folder. However, provided you are in the correct working copy, if you pick the wrong folder level, TortoiseSVN will notice and suggest the correct level.

In order to apply a patch file to your working copy, you need to have at least read access to the repository. The reason for this is that the merge program must reference the changes back to the revision against which they were made by the remote developer.

From the context menu for that folder, click on TortoiseSVN → Apply Patch... This will bring up a file open dialog allowing you to select the patch file to apply. By default only .patch or .diff files are shown, but you can opt for "All files". If you previously saved a patch to the clipboard, you can use Open from clipboard... in the file open dialog.

Alternatively, if the patch file has a .patch or .diff extension, you can right click on it directly and select TortoiseSVN → Apply Patch.... In this case you will be prompted to enter a working copy location.

These two methods just offer different ways of doing the same thing. With the first method you select the WC and browse to the patch file. With the second you select the patch file and browse to the WC.

Once you have selected the patch file and working copy location, TortoiseMerge runs to merge the changes from the patch file with your working copy. A small window lists the files which have been changed. Double click on each one in turn, review the changes and save the merged files.

The remote developer's patch has now been applied to your working copy, so you need to commit to allow everyone else to access the changes from the repository.

5.22. Who Changed Which Line?

Sometimes you need to know not only what lines have changed, but also who exactly changed specific lines in a file. That's when the TortoiseSVN → Blame... command, sometimes also referred to as *annotate* command comes in handy.

This command lists, for every line in a file, the author and the revision the line was changed.

5.22.1. Blame for Files
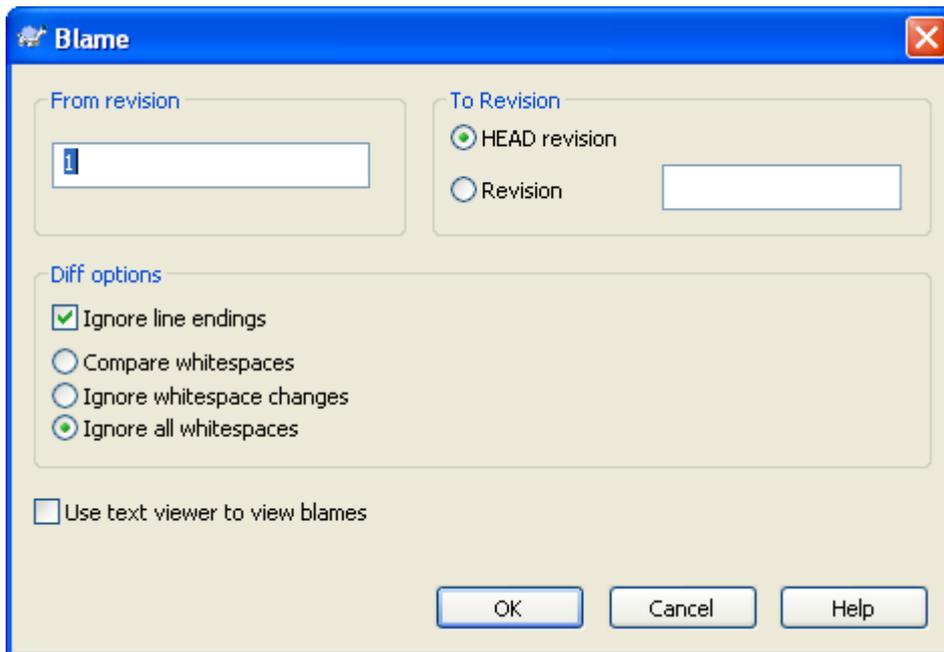
Figure 5.43. The Annotate / Blame Dialog

If you're not interested in changes from earlier revisions you can set the revision from which the blame should start. Set this to `1`, if you want the blame for *every* revision.

By default the blame file is viewed using *TortoiseBlame*, which highlights the different revisions to make it easier to read. If you wish to print or edit the blame file, select Use Text viewer to view blames

You can specify the way that line ending and whitespace changes are handled. These options are described in Section 5.10.2, "Line-end and Whitespace Options". The default behaviour is to treat all whitespace and line-end differences as real changes, but if you want to ignore an indentation change and find the original author, you can choose an appropriate option here.

Once you press OK TortoiseSVN starts retrieving the data to create the blame file. Please note: This can take several minutes to finish, depending on how much the file has changed and of course your network connection to the repository. Once the blame process has finished the result is written into a temporary file and you can view the results.
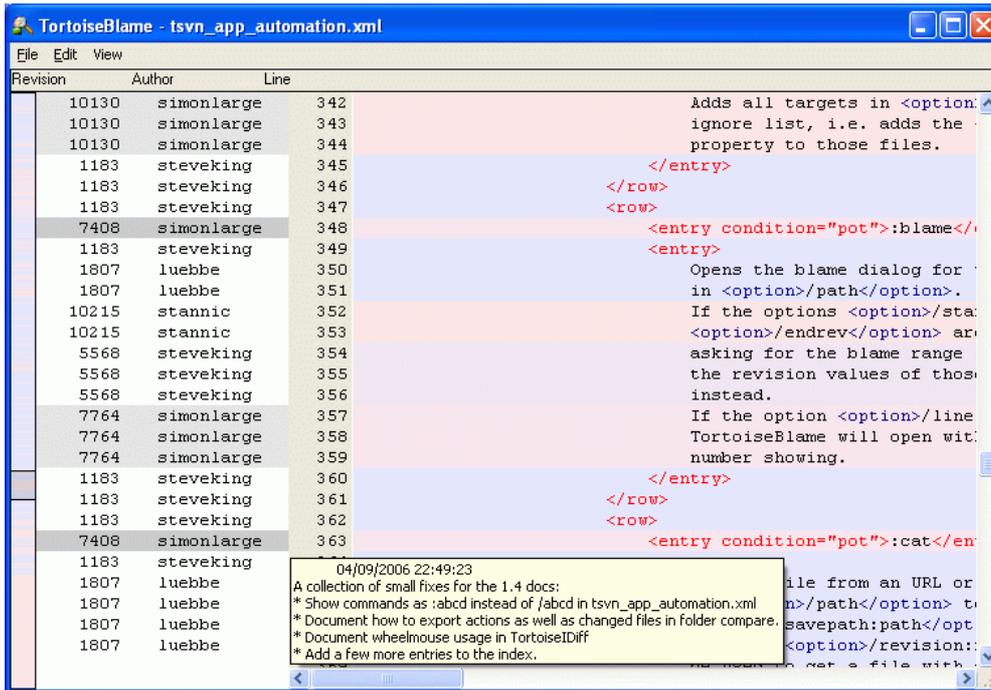
Figure 5.44. TortoiseBlame

TortoiseBlame, which is included with TortoiseSVN, makes the blame file easier to read. When you hover the mouse over a line in the blame info column, all lines with the same revision are shown with a darker background. Lines from other revisions which were changed by the same author are shown with a light background. The colouring may not work as clearly if you have your display set to 256 colour mode.

If you left click on a line, all lines with the same revision are highlighted, and lines from other revisions by the same author are highlighted in a lighter colour. This highlighting is sticky, allowing you to move the mouse without losing the highlights. Click on that revision again to turn off highlighting.

The revision comments (log message) are shown in a hint box whenever the mouse hovers over the blame info column. If you want to copy the log message for that revision, use the context menu which appears when you right click on the blame info column.

You can search within the Blame report using Edit → Find.... This allows you to search for revision numbers, authors and the content of the file itself. Log messages are not included in the search - you should use the Log Dialog to search those.

You can also jump to a specific line number using Edit → Go To Line....

When the mouse is over the blame info columns, a context menu is available which helps with comparing revisions and examining history, using the revision number of the line under the mouse as a reference. Context menu → Blame previous revision generates a blame report for the same file, but using the previous revision as the upper limit. This gives you the blame report for the state of the file just before the line you are looking at was last changed. Context menu → Show changes starts your diff viewer, showing you what changed in the referenced revision. Context menu → Show log displays the revision log dialog starting with the referenced revision.

If you need a better visual indicator of where the oldest and newest changes are, select View → Color age of lines. This will use a colour gradient to show newer lines in red and older lines in blue. The default colouring is quite light, but you can change it using the TortoiseBlame settings.

If you are using Merge Tracking, where lines have changed as a result of merging from another path, TortoiseBlame will show the revision and author of the last change in the original file rather than the revision where the merge took place. These lines are indicated by showing the revision and author in italics.

If you want to see the paths involved in the merge, select View → Merge paths.

The settings for TortoiseBlame can be accessed using TortoiseSVN → Settings... on the TortoiseBlame tab. Refer to Section 5.29.8, "TortoiseBlame Settings".

5.22.2. Blame Differences

One of the limitations of the Blame report is that it only shows the file as it was in a particular revision, and shows the last person to change each line. Sometimes you want to know what change was made, as well as who made it. What you need here is a combination of the diff and blame reports.

The revision log dialog includes several options which allow you to do this.

Blame Revisions

> In the top pane, select 2 revisions, then select Context menu → Blame revisions. This will fetch the blame data for the 2 revisions, then use the diff viewer to compare the two blame files.

Blame Changes

> Select one revision in the top pane, then pick one file in the bottom pane and select Context menu → Blame changes. This will fetch the blame data for the selected revision and the previous revision, then use the diff viewer to compare the two blame files.

Compare and Blame with Working BASE

> Show the log for a single file, and in the top pane, select a single revision, then select Context menu → Compare and Blame with Working BASE. This will fetch the blame data for the selected revision, and for the file in the working BASE, then use the diff viewer to compare the two blame files.

5.23. The Repository Browser

Sometimes you need to work directly on the repository, without having a working copy. That's what the *Repository Browser* is for. Just as the explorer and the icon overlays allow you to view your working copy, so the Repository Browser allows you to view the structure and status of the repository.
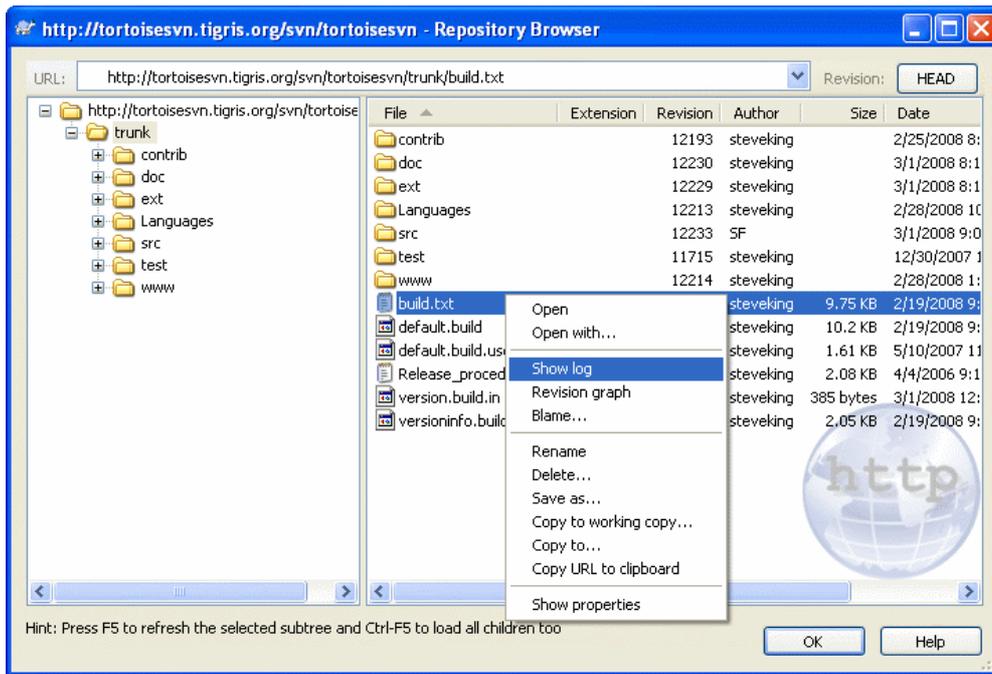
Figure 5.45. The Repository Browser

With the Repository Browser you can execute commands like copy, move, rename, ... directly on the repository.

The repository browser looks very similar to the Windows explorer, except that it is showing the content of the repository at a particular revision rather than files on your computer. In the left pane you can see a directory tree, and in the right pane are the contents of the selected directory. At the top of the Repository Browser Window you can enter the URL of the repository and the revision you want to browse.

Just like Windows explorer, you can click on the column headings in the right pane if you want to set the sort order. And as in explorer there are context menus available in both panes.

The context menu for a file allows you to:

- Open the selected file, either with the default viewer for that file type, or with a program you choose.

- Save an unversioned copy of the file to your hard drive.

- Show the revision log for that file, or show a graph of all revisions so you can see where the file came from.

- Blame the file, to see who changed which line and when.

- Delete or rename the file.

- Make a copy of the file, either to a different part of the repository, or to a working copy rooted in the same repository.

- View/Edit the file's properties.

The context menu for a folder allows you to:

- Show the revision log for that folder, or show a graph of all revisions so you can see where the folder came from.

- Export the folder to a local unversioned copy on your hard drive.

- Checkout the folder to produce a local working copy on your hard drive.

- Create a new folder in the repository.

- Add files or folders directly to the repository.

- Delete or rename the folder.

- Make a copy of the folder, either to a different part of the repository, or to a working copy rooted in the same repository.

- View/Edit the folder's properties.

- Mark the folder for comparison. A marked folder is shown in bold.

- Compare the folder with a previously marked folder, either as a unified diff, or as a list of changed files which can then be visually diffed using the default diff tool. This can be particularly useful for comparing two tags, or trunk and branch to see what changed.

If you select two folders in the right pane, you can view the differences either as a unified-diff, or as a list of files which can be visually diffed using the default diff tool.

If you select multiple folders in the right pane, you can checkout all of them at once into a common parent folder.

If you select 2 tags which are copied from the same root (typically /trunk/), you can use Context Menu → Show Log... to view the list of revisions between the two tag points.

You can use **F5** to refresh the view as usual. This will refresh everything which is currently displayed. If you want to pre-fetch or refresh the information for nodes which have not been opened yet, use **Ctrl-F5**. After that, expanding any node will happen instantly without a network delay while the information is fetched.

You can also use the repository browser for drag-and-drop operations. If you drag a folder from explorer into the repo-browser, it will be imported into the repository. Note that if you drag multiple items, they will be imported in separate commits.

If you want to move an item within the repository, just left drag it to the new location. If you want to create a copy rather than moving the item, **Ctrl**-left drag instead. When copying, the cursor has a "plus" symbol on it, just as it does in Explorer.

If you want to copy/move a file or folder to another location and also give it a new name at the same time, you can right drag or **Ctrl**-right drag the item instead of using left drag. In that case, a rename dialog is shown where you can enter a new name for the file or folder.

Whenever you make changes in the repository using one of these methods, you will be presented with a log message entry dialog. If you dragged something by mistake, this is also your chance to cancel the action.

Sometimes when you try to open a path you will get an error message in place of the item details. This might happen if you specified an invalid URL, or if you don't have access permission, or if there is some other server problem. If you need to copy this message to include it in an email, just right click on it and use Context Menu → Copy error message to clipboard, or simply use **Ctrl+C**.

5.24. Revision Graphs
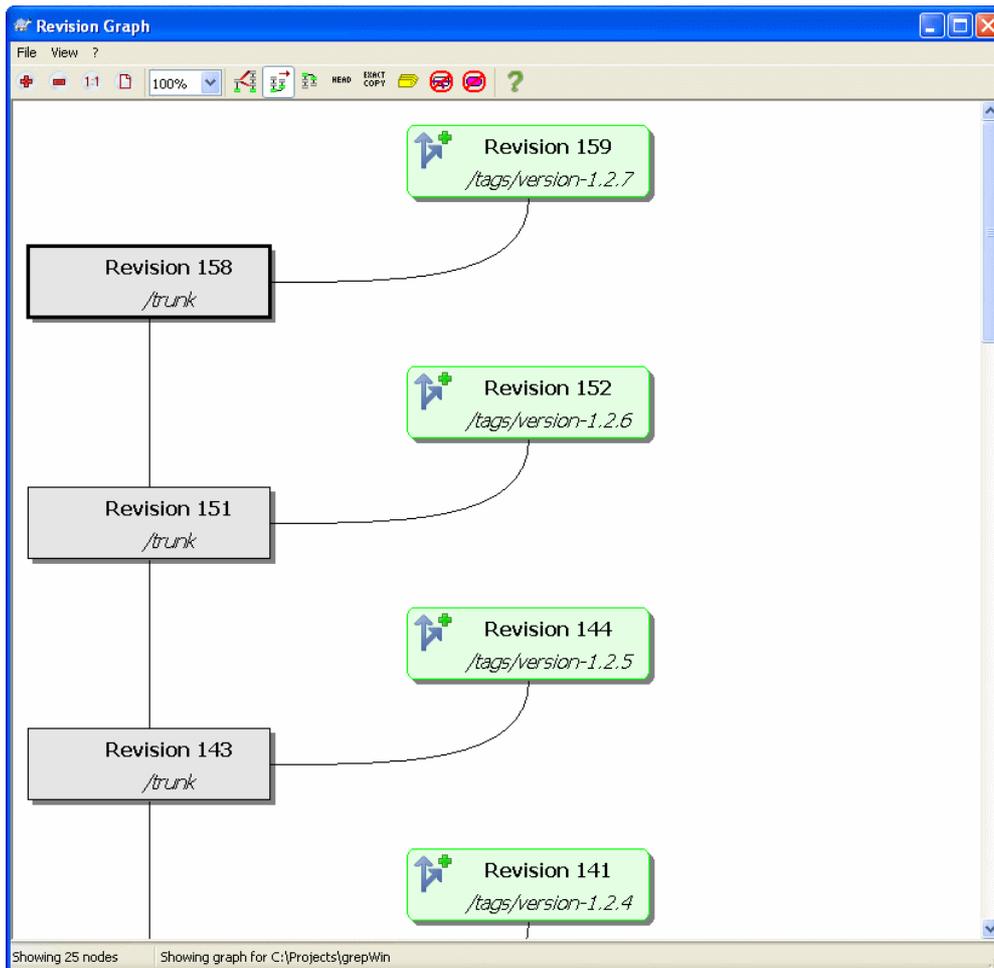
Figure 5.46. A Revision Graph

Sometimes you need to know where branches and tags were taken from the trunk, and the ideal way to view this sort of information is as a graph or tree structure. That's when you need to use TortoiseSVN → Revision Graph...

This command analyses the revision history and attempts to create a tree showing the points at which copies were taken, and when branches/tags were deleted.

> **Important**
>
> In order to generate the graph, TortoiseSVN must fetch all log messages from the repository root. Needless to say this can take several minutes even with a repository of a few thousand revisions, depending on server speed, network bandwidth, etc. If you try this with something like the *Apache* project which currently has over 500,000 revisions you could be waiting for some time.
>
> The good news is that if you are using Log Caching, you only have to suffer this delay once. After that, log data is held locally. Log caching is enabled in TortoiseSVN's settings.

5.24.1. Revision Graph Nodes

The revision graph shows several types of node:

Added file/folder
  Items which have been added, or created by copying another file/folder are shown using a rounded rectangle.

Deleted file/folder
> Deleted items eg. a branch which is no longer required, are shown using an octagon (rectangle with corners cut off).

Branch tip revision
> Where a branch (or trunk or tag) has been modified since the last branch node, this is shown using an ellipse. Shown when the Show HEAD revisions option is selected.

Normal file/folder
> All other items are shown using a plain rectangle.

Note that by default the graph only shows the points at which items were added or deleted. Showing every revision of a project will generate a very large graph for non-trivial cases. If you really want to see *all* revisions where changes were made, there is an option to do this in the View menu and on the toolbar.

5.24.2. Changing the View

Because a revision graph is often quite complex, there are a number of features which can be used to tailor the view the way you want it. These are available in the View menu and from the toolbar.

Group branches
> The default behavior (grouping off) will use one row per revision and all rows are sorted strictly by revision. As a result, long-living branches occupy a whole column for only a few changes and the graph becomes very broad.
>
> This mode groups changes by branch, so that there is no global revision ordering: Consecutive revisions on a branch will be shown in (often) consecutive lines. Sub-branches, however, are arranged in such a way that later branches will be shown in the same column above older branches to keep the graph slim. As a result, a given row may contain changes from different revisions.

Oldest on top
> Normally the graph shows the oldest revision at the bottom, and the tree grows upwards. Use this option to grow down from the top instead.

Show HEAD revisions
> This ensures that the latest revision on every branch is always shown on the graph.

Exact copy sources
> When a branch/tag is made, the default behaviour is to show the branch as taken from the last node where a change was made. Strictly speaking this is inaccurate since the branches are often made from the current HEAD rather than a specific revision. So it is possible to show the more correct (but less useful) revision that was used to create the copy.

Fold tags
> If you want to see a graph of software development, tagged releases may be of little interest to you. This option hides the nodes for tags and shows them instead in the tooltip for the node that they were copied from. A tag icon on the right side of the source node indicates that tags were made.

Reduce cross lines
> If the layout of the graph has produced a lot of crossing lines, use this option to clean it up. This may make the layout columns appear in less logical places, for example in a diagonal line rather than a column, and it may take a little time to optimise.

Filter
> Sometimes the revision graph contains more revisions than you want to see. This option opens a dialog which allows you to restrict the range of revisions displayed, and to hide particular paths by name.

5.24.3. Using the Graph

To make it easier to navigate a large graph, use the overview window. This shows the entire graph in a small window, with the currently displayed portion highlighted. You can drag the highlighted area to change the displayed region.

The revision date, author and comments are shown in a hint box whenever the mouse hovers over a revision box.

If you select two revisions (Use **Ctrl**-left click), you can use the context menu to show the differences between these revisions. You can choose to show differences as at the branch creation points, but usually you will want to show the differences at the branch end points, i.e. at the HEAD revision.

You can view the differences as a Unified-Diff file, which shows all differences in a single file with minimal context. If you opt to Context Menu → Compare Revisions you will be presented with a list of changed files. Double click on a file name to fetch both revisions of the file and compare them using the visual difference tool.

If you right click on a revision you can use Context Menu → Show Log to view the history.

You can also merge changes in the selected revision(s) into a different working copy. A folder selection dialog allows you to choose the working copy to merge into, but after that there is no confirmation dialog, nor any opportunity to try a dry run. It is a good idea to merge into an unmodified working copy so that you can revert the changes if it doesn't work out! This is a useful feature if you want to merge selected revisions from one branch to another.

> ### Caution
>
> Because Subversion cannot provide all the information required, a certain amount of interpretation is required, which can sometimes give strange results. Nevertheless, the output for the trunk will generally give useful results.

5.24.4. Refreshing the View

If you want to check the server again for newer information, you can simply refresh the view using **F5**. If you are using the log cache (enabled by default), this will check the repository for newer commits and fetch only the new ones. If the log cache was in offline mode, this will also attempt to go back online.

If you are using the log cache and you think the message content or author may have changed, you should use the log dialog to refresh the messages you need. Since the revision graph works from the repository root, we would have to invalidate the entire log cache, and refilling it could take a *very* long time.

5.25. Exporting a Subversion Working Copy

Sometimes you may want a copy of your working tree without any of those .svn directories, e.g. to create a zipped tarball of your source, or to export to a web server. Instead of making a copy and then deleting all those .svn directories manually, TortoiseSVN offers the command TortoiseSVN → Export.... Exporting from a URL and exporting from a working copy are treated slightly differently.
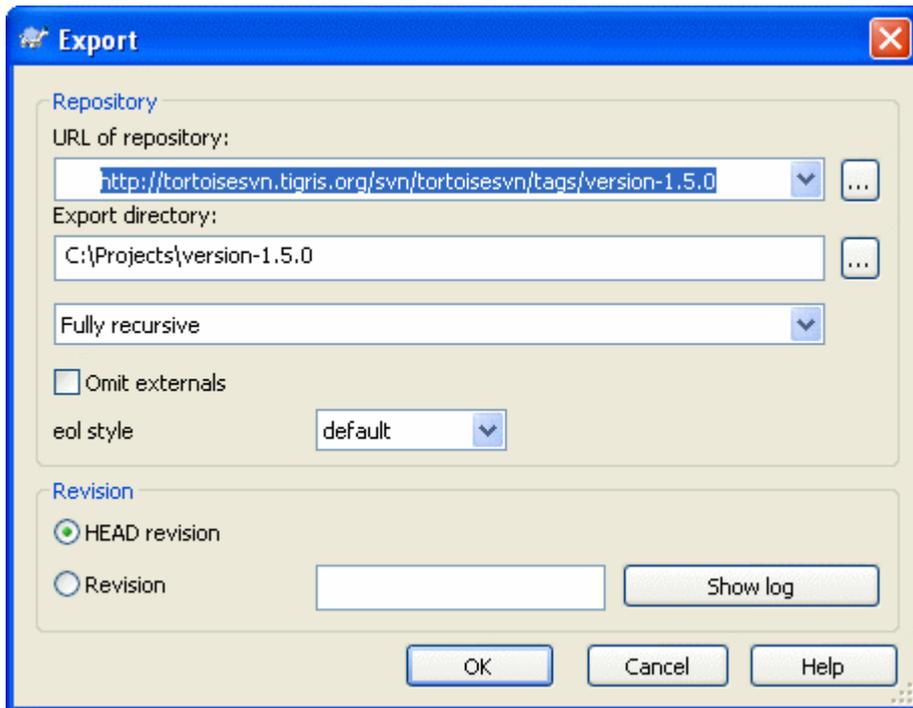
Figure 5.47. The Export-from-URL Dialog

If you execute this command on an unversioned folder, TortoiseSVN will assume that the selected folder is the target, and open a dialog for you to enter the URL and revision to export from. This dialog has options to export only the top level folder, to omit external references, and to override the line end style for files which have the `svn:eol-style` property set.

Of course you can export directly from the repository too. Use the Repository Browser to navigate to the relevant subtree in your repository, then use Context Menu → Export. You will get the Export from URL dialog described above.

If you execute this command on your working copy you'll be asked for a place to save the *clean* working copy without the `.svn` folders. By default, only the versioned files are exported, but you can use the Export unversioned files too checkbox to include any other unversioned files which exist in your WC and not in the repository. External references using `svn:externals` can be omitted if required.

Another way to export from a working copy is to right drag the working copy folder to another location and choose Context Menu → SVN Export here or Context Menu → SVN Export all here. The second option includes the unversioned files as well.

When exporting from a working copy, if the target folder already contains a folder of the same name as the one you are exporting, you will be given the option to overwrite the existing content, or to create a new folder with an automatically generated name, eg. `Target (1)`.

5.25.1. Removing a working copy from version control

Sometimes you have a working copy which you want to convert back to a normal folder without the `.svn` directories. What you really need is an export-in-place command, that just removes the control directories rather than generating a new clean directory tree.

The answer is surprisingly simple - export the folder to itself! TortoiseSVN detects this special case and asks if you want to make the working copy unversioned. If you answer *yes* the control directories will be removed and you will have a plain, unversioned directory tree.

5.26. Relocating a working copy
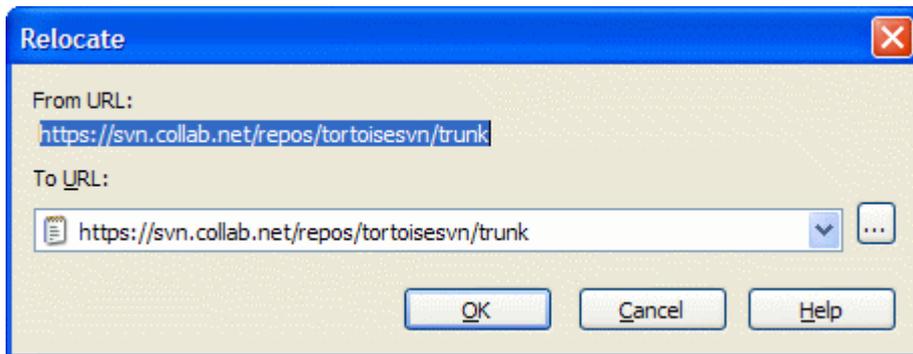
Figure 5.48. The Relocate Dialog

If your repository has for some reason changed it's location (IP/URL). Maybe you're even stuck and can't commit and you don't want to checkout your working copy again from the new location and to move all your changed data back into the new working copy, TortoiseSVN → Relocate is the command you are looking for. It basically does very little: it scans all `entries` files in the `.svn` folder and changes the URL of the entries to the new value.

> # Warning
>
> *This is a very infrequently used operation*. The relocate command is *only* used if the URL of the repository root has changed. Possible reasons are:
>
> • The IP address of the server has changed.
>
> • The protocol has changed (e.g. http:// to https://).
>
> • The repository root path in the server setup has changed.
>
> Put another way, you need to relocate when your working copy is referring to the same location in the same repository, but the repository itself has moved.
>
> It does not apply if:
>
> • You want to move to a different Subversion repository. In that case you should perform a clean checkout from the new repository location.
>
> • You want to switch to a different branch or directory within the same repository. To do that you should use TortoiseSVN → Switch.... Read Section 5.18.2, "To Checkout or to Switch..." for more information.
>
> If you use relocate in either of the cases above, it *will corrupt your working copy* and you will get many unexplainable error messages while updating, committing, etc. Once that has happened, the only fix is a fresh checkout.

5.27. Integration with Bug Tracking Systems / Issue Trackers

It is very common in Software Development for changes to be related to a specific bug or issue ID. Users of bug tracking systems (issue trackers) would like to associate the changes they make in Subversion with a specific ID in their issue tracker. Most issue trackers therefore provide a pre-commit hook script which parses the log message to find the bug ID with which the commit is associated. This is somewhat

error prone since it relies on the user to write the log message properly so that the pre-commit hook script can parse it correctly.

TortoiseSVN can help the user in two ways:

1. When the user enters a log message, a well defined line including the issue number associated with the commit can be added automatically. This reduces the risk that the user enters the issue number in a way the bug tracking tools can't parse correctly.

   Or TortoiseSVN can highlight the part of the entered log message which is recognized by the issue tracker. That way the user knows that the log message can be parsed correctly.

2. When the user browses the log messages, TortoiseSVN creates a link out of each bug ID in the log message which fires up the browser to the issue mentioned.

5.27.1. Adding Issue Numbers to Log Messages

You can integrate a bug tracking tool of your choice in TortoiseSVN. To do this, you have to define some properties, which start with `bugtraq:`. They must be set on Folders: (Section 5.17, "Project Settings")

There are two ways to integrate TortoiseSVN with issue trackers. One is based on simple strings, the other is based on *regular expressions*. The properties used by both approaches are:

bugtraq:url
> Set this property to the URL of your bug tracking tool. It must be properly URI encoded and it has to contain `%BUGID%`. `%BUGID%` is replaced with the Issue number you entered. This allows TortoiseSVN to display a link in the log dialog, so when you are looking at the revision log you can jump directly to your bug tracking tool. You do not have to provide this property, but then TortoiseSVN shows only the issue number and not the link to it. e.g the TortoiseSVN project is using `http://issues.tortoisesvn.net/?do=details&id=%BUGID%`
>
> You can also use relative URLs instead of absolute ones. This is useful when your issue tracker is on the same domain/server as your source repository. In case the domain name ever changes, you don't have to adjust the `bugtraq:url` property. There are two ways to specify a relative URL:
>
> If it begins with the string `^/` it is assumed to be relative to the repository root. For example, `^/../?do=details&id=%BUGID%` will resolve to `http://tortoisesvn.net/?do=details&id=%BUGID%` if your repository is located on `http://tortoisesvn.net/svn/trunk/`.
>
> A URL beginning with the string `/` is assumed to be relative to the server's hostname. For example `/?do=details&id=%BUGID%` will resolve to `http://tortoisesvn.net/?do=details&id=%BUGID%` if your repository is located anywhere on `http://tortoisesvn.net`.

bugtraq:warnifnoissue
> Set this to `true`, if you want TortoiseSVN to warn you because of an empty issue-number text field. Valid values are `true/false`. *If not defined, `false` is assumed.*

5.27.1.1. Issue Number in Text Box

In the simple approach, TortoiseSVN shows the user a separate input field where a bug ID can be entered. Then a separate line is appended/prepended to the log message the user entered.

bugtraq:message
> This property activates the bug tracking system in *Input field* mode. If this property is set, then TortoiseSVN will prompt you to enter an issue number when you commit your changes. It's used to add a line at the end of the log message. It must contain `%BUGID%`, which is replaced with the issue number on commit. This ensures that your commit log contains a reference to the issue number which is always in a consistent format and can be parsed by your bug tracking tool to associate the

issue number with a particular commit. As an example you might use `Issue : %BUGID%`, but this depends on your Tool.

bugtraq:append

> This property defines if the bug-ID is appended (true) to the end of the log message or inserted (false) at the start of the log message. Valid values are `true/false`. *If not defined, `true` is assumed, so that existing projects don't break.*

bugtraq:label

> This text is shown by TortoiseSVN on the commit dialog to label the edit box where you enter the issue number. If it's not set, `Bug-ID / Issue-Nr:` will be displayed. Keep in mind though that the window will not be resized to fit this label, so keep the size of the label below 20-25 characters.

bugtraq:number

> If set to `true` only numbers are allowed in the issue-number text field. An exception is the comma, so you can comma separate several numbers. Valid values are `true/false`. *If not defined, `true` is assumed.*

5.27.1.2. Issue Numbers Using Regular Expressions

In the approach with *regular expressions*, TortoiseSVN doesn't show a separate input field but marks the part of the log message the user enters which is recognized by the issue tracker. This is done while the user writes the log message. This also means that the bug ID can be anywhere inside a log message! This method is much more flexible, and is the one used by the TortoiseSVN project itself.

bugtraq:logregex

> This property activates the bug tracking system in *Regex* mode. It contains either a single regular expressions, or two regular expressions separated by a newline.
>
> If two expressions are set, then the first expression is used as a pre-filter to find expressions which contain bug IDs. The second expression then extracts the bare bug IDs from the result of the first regex. This allows you to use a list of bug IDs and natural language expressions if you wish. e.g. you might fix several bugs and include a string something like this: "This change resolves issues #23, #24 and #25"
>
> If you want to catch bug IDs as used in the expression above inside a log message, you could use the following regex strings, which are the ones used by the TortoiseSVN project: `[Ii]ssues?:? (\s*(,|and)?\s*#\d+)+` and `(\d+)`
>
> The first expression picks out "issues #23, #24 and #25" from the surrounding log message. The second regex extracts plain decimal numbers from the output of the first regex, so it will return "23", "24" and "25" to use as bug IDs.
>
> Breaking the first regex down a little, it must start with the word "issue", possibly capitalised. This is optionally followed by an "s" (more than one issue) and optionally a colon. This is followed by one or more groups each having zero or more leading whitespace, an optional comma or "and" and more optional space. Finally there is a mandatory "#" and a mandatory decimal number.
>
> If only one expression is set, then the bare bug IDs must be matched in the groups of the regex string. Example: `[Ii]ssue(?:s)? #?(\d+)` This method is required by a few issue trackers, e.g. trac, but it is harder to construct the regex. We recommend that you only use this method if your issue tracker documentation tells you to.
>
> If you are unfamiliar with regular expressions, take a look at the introduction at *http://en.wikipedia.org/wiki/Regular_expression* [http://en.wikipedia.org/wiki/Regular_expression], and the online documentation and tutorial at *http://www.regular-expressions.info/* [http://www.regular-expressions.info/].

If both the `bugtraq:message` and `bugtraq:logregex` properties are set, `logregex` takes precedence.

> **Tip**
>
> Even if you don't have an issue tracker with a pre-commit hook parsing your log messages, you still can use this to turn the issues mentioned in your log messages into links!
>
> And even if you don't need the links, the issue numbers show up as a separate column in the log dialog, making it easier to find the changes which relate to a particular issue.

Some `tsvn:` properties require a `true/false` value. TortoiseSVN also understands `yes` as a synonym for `true` and `no` as a synonym for `false`.

> **Set the Properties on Folders**
>
> These properties must be set on folders for the system to work. When you commit a file or folder the properties are read from that folder. If the properties are not found there, TortoiseSVN will search upwards through the folder tree to find them until it comes to an unversioned folder, or the tree root (eg. `C:\`) is found. If you can be sure that each user checks out only from e.g `trunk/` and not some sub-folder, then it's enough if you set the properties on `trunk/`. If you can't be sure, you should set the properties recursively on each sub-folder. A property setting deeper in the project hierarchy overrides settings on higher levels (closer to `trunk/`).
>
> For `tsvn:` properties *only* you can use the Recursive checkbox to set the property to all sub-folders in the hierarchy, without also setting it on all files.

This issue tracker integration is not restricted to TortoiseSVN; it can be used with any Subversion client. For more information, read the full *Issue Tracker Integration Specification* [http://tortoisesvn.tigris.org/svn/tortoisesvn/trunk/doc/issuetrackers.txt].

5.27.2. Getting Information from the Issue Tracker

The previous section deals with adding issue information to the log messages. But what if you need to get information from the issue tracker? The commit dialog has a Windows COM interface which allows integration an external program that can talk to your tracker. Typically you might want to query the tracker to get a list of open issues assigned to you, so that you can pick the issues that are being addressed in this commit.

Any such interface is of course highly specific to your system, so we cannot provide this part, and describing how to create such a program is beyond the scope of this manual. The interface definition and sample programs can be obtained from the `contrib` folder in the in the *TortoiseSVN repository* [http://tortoisesvn.tigris.org/svn/tortoisesvn/trunk/contrib/issue-tracker-plugins].

For illustration purposes, let's suppose that your system administrator has provided you with an issue tracker plugin which you have installed, and that you have set up some of your working copies to use the plugin in TortoiseSVN's settings dialog. When you open the commit dialog from a working copy to which the plugin has been assigned, you will see a new button at the top of the dialog.
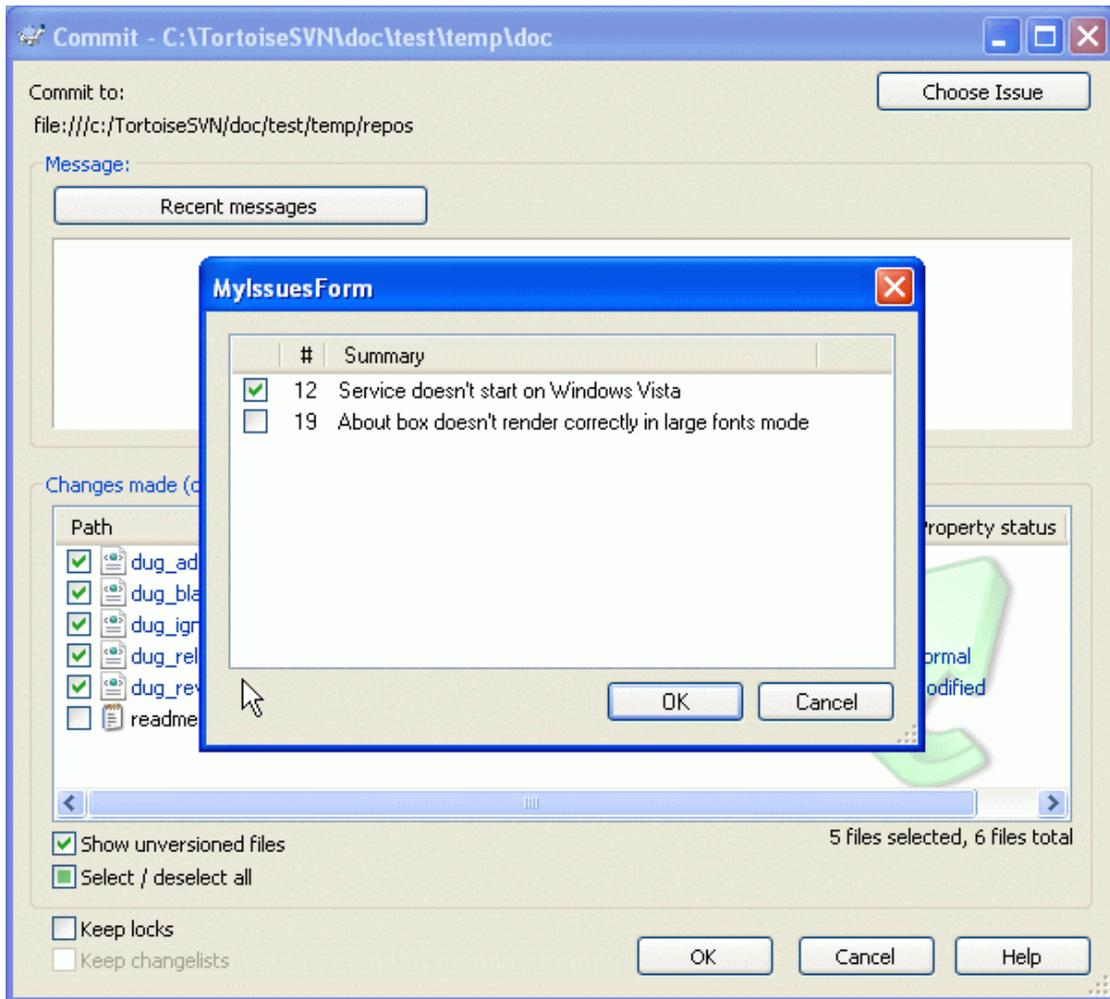
Figure 5.49. Example issue tracker query dialog

In this example you can select one or more open issues. The plugin can then generate specially formatted text which it adds to your log message.

5.28. Integration with Web-based Repository Viewers

There are several web-based repository viewers available for use with Subversion such as *ViewVC* [http://www.viewvc.org/] and *WebSVN* [http://websvn.tigris.org/]. TortoiseSVN provides a means to link with these viewers.

You can integrate a repo viewer of your choice in TortoiseSVN. To do this, you have to define some properties which define the linkage. They must be set on Folders: (Section 5.17, "Project Settings")

webviewer:revision

> Set this property to the URL of your repo viewer to view all changes in a specific revision. It must be properly URI encoded and it has to contain %REVISION%. %REVISION% is replaced with the revision number in question. This allows TortoiseSVN to display a context menu entry in the log
>
> dialog Context Menu → View revision in webviewer

webviewer:pathrevision

> Set this property to the URL of your repo viewer to view changes to a specific file in a specific revision. It must be properly URI encoded and it has to contain %REVISION% and %PATH%. %PATH% is replaced with the path relative to the repository root. This allows TortoiseSVN to display a
>
> context menu entry in the log dialog Context Menu → View revision and path in webviewer For

example, if you right-click in the log dialog bottom pane on a file entry /trunk/src/file then the %PATH% in the URL will be replaced with /trunk/src/file.

You can also use relative URLs instead of absolute ones. This is useful in case your web viewer is on the same domain/server as your source repository. In case the domain name ever changes, you don't have to adjust the webviewer:revision and webviewer:pathrevision property. The format is the same as for the bugtraq:url property. See Section 5.27, "Integration with Bug Tracking Systems / Issue Trackers".

> ### Set the Properties on Folders
>
> These properties must be set on folders for the system to work. When you commit a file or folder the properties are read from that folder. If the properties are not found there, TortoiseSVN will search upwards through the folder tree to find them until it comes to an unversioned folder, or the tree root (eg. C:\) is found. If you can be sure that each user checks out only from e.g trunk/ and not some sub-folder, then it's enough if you set the properties on trunk/. If you can't be sure, you should set the properties recursively on each sub-folder. A property setting deeper in the project hierarchy overrides settings on higher levels (closer to trunk/).
>
> For tsvn: properties *only* you can use the Recursive checkbox to set the property to all sub-folders in the hierarchy, without also setting it on all files.

## 5.29. TortoiseSVN's Settings

To find out what the different settings are for, just leave your mouse pointer a second on the editbox/checkbox... and a helpful tooltip will popup.

5.29.1. General Settings
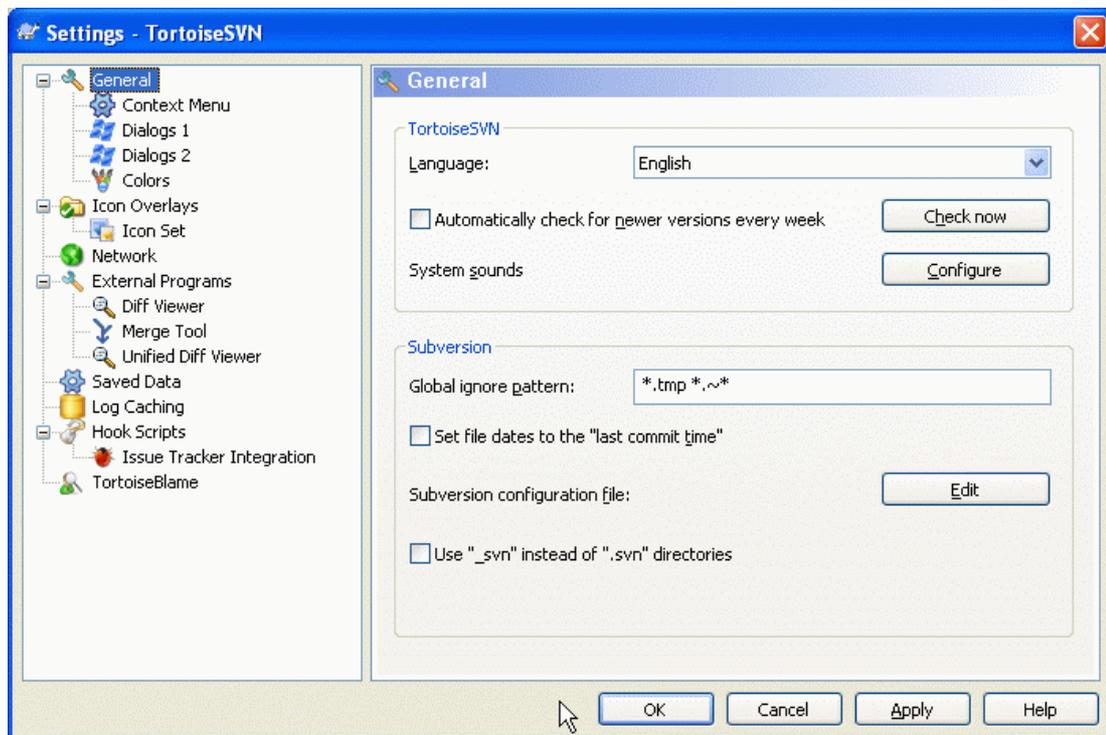
Figure 5.50. The Settings Dialog, General Page

This dialog allows you to specify your preferred language, and the Subversion-specific settings.

Language
:   Selects your user interface language. What else did you expect?

Automatically check for newer versions every week
:   If checked, TortoiseSVN will contact its download site once a week to see if there is a newer version of the program available. Use Check now if you want an answer right away. The new version will not be downloaded; you simply receive an information dialog telling you that the new version is available.

System sounds
:   TortoiseSVN has three custom sounds which are installed by default.

    • Error

    • Notice

    • Warning

    You can select different sounds (or turn these sounds off completely) using the Windows Control Panel. Configure is a shortcut to the Control Panel.

Global ignore pattern

Global ignore patterns are used to prevent unversioned files from showing up e.g. in the commit dialog. Files matching the patterns are also ignored by an import. Ignore files or directories by typing in the names or extensions. Patterns are separated by spaces e.g. `*/bin */obj *.bak *.~??` `*.jar *.[Tt]mp`. Remember that these patterns may be used against paths which include N levels of parent directory. Setting ignore patterns for anything other than simple file extensions is not as trivial as it first appears, so be sure to read Section 5.13.1, "Pattern Matching in Ignore Lists" for more information on the pattern-matching syntax, and how paths are checked.

Note that the ignore patterns you specify here will also affect other Subversion clients running on your PC, including the command line client.

> ### Caution
>
> If you use the Subversion configuration file to set a `global-ignores` pattern, it will override the settings you make here. The Subversion configuration file is accessed using the Edit as described below.

This ignore pattern will affect all your projects. It is not versioned, so it will not affect other users. By contrast you can also use the versioned `svn:ignore` property to exclude files or directories from version control. Read Section 5.13, "Ignoring Files And Directories" for more information.

Set file dates to the "last commit time"
:   This option tells TortoiseSVN to set the file dates to the last commit time when doing a checkout or an update. Otherwise TortoiseSVN will use the current date. If you are developing software it is generally best to use the current date because build systems normally look at the date stamps to decide which files need compiling. If you use "last commit time" and revert to an older file revision, your project may not compile as you expect it to.

Subversion configuration file
:   Use Edit to edit the Subversion configuration file directly. Some settings cannot be modified directly by TortoiseSVN, and need to be set here instead. For more information about the Subversion `config` file see the *Runtime Configuration Area* [http://

svnbook.red-bean.com/en/1.4/svn.advanced.confarea.html]. The section on *Automatic Property Setting* [http://svnbook.red-bean.com/en/1.4/svn.advanced.props.html#svn.advanced.props.auto] is of particular interest, and that is configured here. Note that Subversion can read configuration information from several places, and you need to know which one takes priority. Refer to *Configuration and the Windows Registry* [http://svnbook.red-bean.com/en/1.4/svn.advanced.confarea.html#svn.advanced.confarea.windows-registry] to find out more.

Use _svn instead of .svn directories

VS.NET when used with web projects can't handle the .svn folders that Subversion uses to store its internal information. This is not a bug in Subversion. The bug is in VS.NET and the frontpage extensions it uses. Read Section 5.29.10, "Subversion Working Folders" to find out more about this issue.

If you want to change the behaviour of Subversion and TortoiseSVN, you can use this checkbox to set the environment variable which controls this.

You should note that changing this option will not automatically convert existing working copies to use the new admin directory. You will have to do that yourself using a script (See our FAQ) or simply check out a fresh working copy.

5.29.1.1. Context Menu Settings
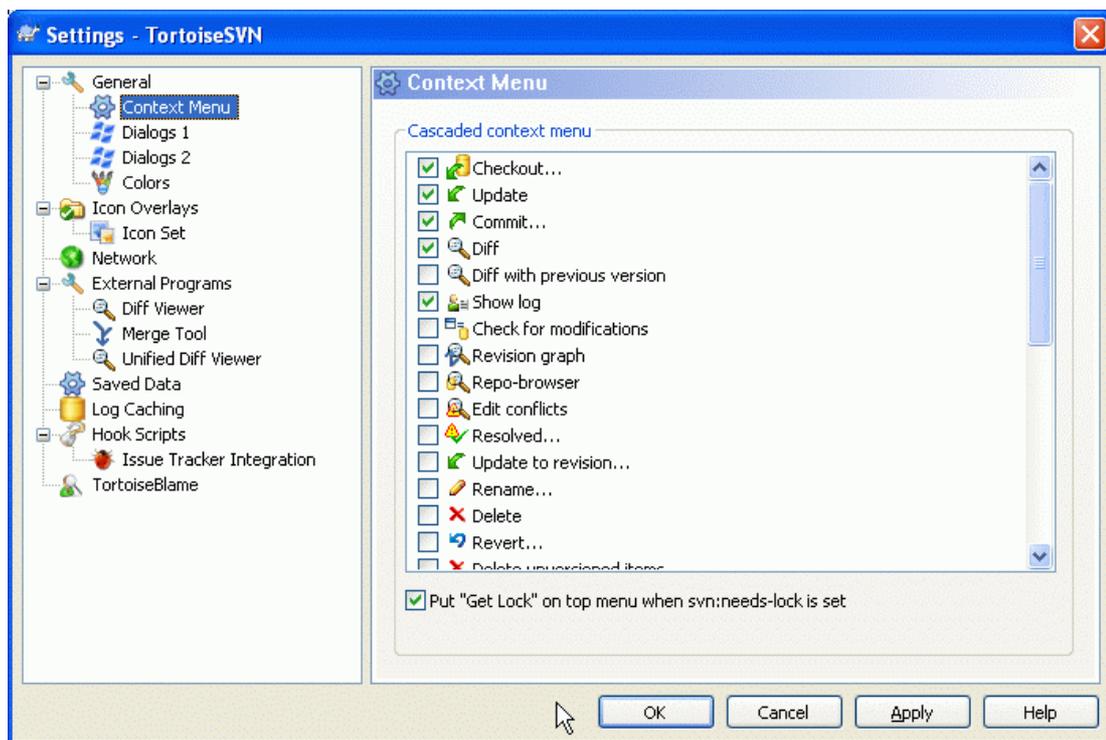
Figure 5.51. The Settings Dialog, Context Menu Page

This page allows you to specify which of the TortoiseSVN context menu entries will show up in the main context menu, and which will appear in the TortoiseSVN submenu. By default most items are unchecked and appear in the submenu.

There is a special case for Get Lock. You can of course promote it to the top level using the list above, but as most files don't need locking this just adds clutter. However, a file with the svn:needs-lock property needs this action every time it is edited, so in that case it is very useful to have at the top level. Checking the box here means that when a file is selected which has the svn:needs-lock property set, Get Lock will always appear at the top level.

5.29.1.2. TortoiseSVN Dialog Settings 1
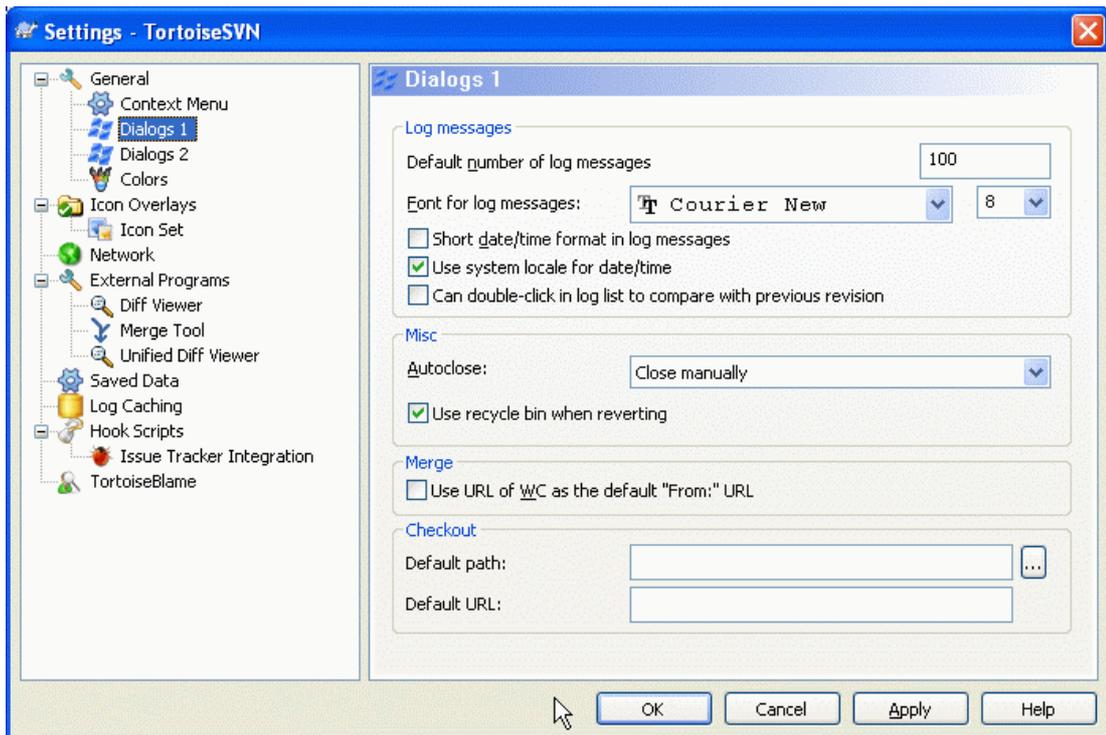
Figure 5.52. The Settings Dialog, Dialogs 1 Page

This dialog allows you to configure some of TortoiseSVN's dialogs the way you like them.

Default number of log messages
>   Limits the number of log messages that TortoiseSVN fetches when you first select TortoiseSVN
>   → Show Log Useful for slow server connections. You can always use Show All or Next 100 to
>   get more messages.

Font for log messages
>   Selects the font face and size used to display the log message itself in the middle pane of the Revision
>   Log dialog, and when composing log messages in the Commit dialog.

Short date / time format in log messages
>   If the standard long messages use up too much space on your screen use the short format.

Progress Dialog
>   TortoiseSVN can automatically close all progress dialogs when the action is finished without error.
>   This setting allows you to select the conditions for closing the dialogs. The default (recommended)
>   setting is Close manually which allows you to review all messages and check what has happened.
>   However, you may decide that you want to ignore some types of message and have the dialog close
>   automatically if there are no critical changes.
>
>   Auto-close if no merges, adds or deletes means that the progress dialog will close if there were
>   simple updates, but if changes from the repository were merged with yours, or if any files were added
>   or deleted, the dialog will remain open. It will also stay open if there were any conflicts or errors
>   during the operation.
>
>   Auto-close if no merges, adds or deletes for local operations means that the progress dialog
>   will close as for Auto-close if no merges, adds or deletes but only for local operations like adding
>   files or reverting changes. For remote operations the dialog will stay open.
>
>   Auto-close if no conflicts relaxes the criteria further and will close the dialog even if there were
>   merges, adds or deletes. However, if there were any conflicts or errors, the dialog remains open.

Auto-close if no errors always closes the dialog even if there were conflicts. The only condition that keeps the dialog open is an error condition, which occurs when Subversion is unable to complete the task. For example, an update fails because the server is inaccessible, or a commit fails because the working copy is out-of-date.

Use URL of WC as the default "From:" URL

In the merge dialog, the default behaviour is for the From: URL to be remembered between merges. However, some people like to perform merges from many different points in their hierarchy, and find it easier to start out with the URL of the current working copy. This can then be edited to refer to a parallel path on another branch.

Default checkout path

You can specify the default path for checkouts. If you keep all your checkouts in one place, it is useful to have the drive and folder pre-filled so you only have to add the new folder name to the end.

Default checkout URL

You can also specify the default URL for checkouts. If you often checkout sub-projects of some very large project, it can be useful to have the URL pre-filled so you only have to add the sub-project name to the end.

5.29.1.3. TortoiseSVN Dialog Settings 2
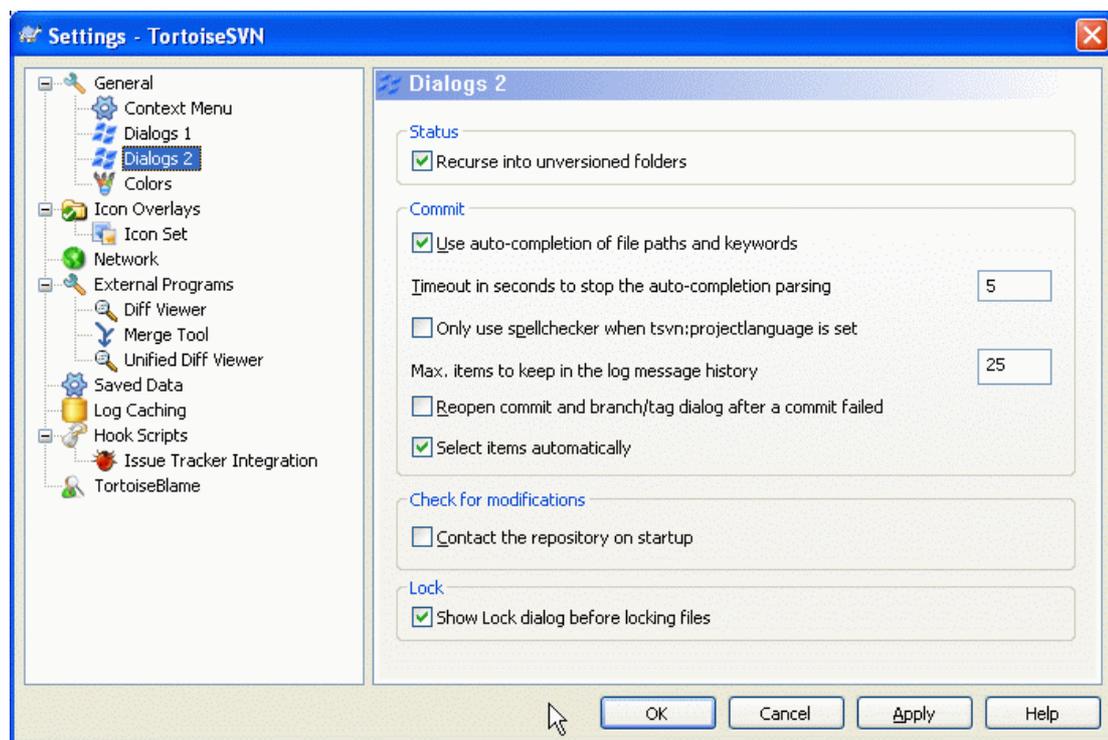
Figure 5.53. The Settings Dialog, Dialogs 2 Page

Recurse into unversioned folders

If this box is checked (default state), then whenever the status of an unversioned folder is shown in the Add, Commit or Check for Modifications dialog, every child file and folder is also shown. If you uncheck this box, only the unversioned parent is shown. Unchecking reduces clutter in these dialogs. In that case if you select an unversioned folder for Add, it is added recursively.

Use auto-completion of file paths and keywords

The commit dialog includes a facility to parse the list of filenames being committed. When you type the first 3 letters of an item in the list, the auto-completion box pops up, and you can press Enter to complete the filename. Check the box to enable this feature.

Timeout in seconds to stop the auto-completion parsing
: The auto-completion parser can be quite slow if there are a lot of large files to check. This timeout stops the commit dialog being held up for too long. If you are missing important auto-completion information, you can extend the timeout.

Only use spellchecker when `tsvn:projectlanguage` is set
: If you don't wish to use the spellchecker for all commits, check this box. The spellchecker will still be enabled where the project properties require it.

Max. items to keep in the log message history
: TortoiseSVN stores the last 25 log messages you entered for each repository. You can customize the number stored here. If you have many different repositories, you may wish to reduce this to avoid filling your registry.

Re-open commit and branch/tag dialog after a commit failed
: When a commit fails for some reason (working copy needs updating, pre-commit hook rejects commit, network error, etc), you can select this option to keep the commit dialog open ready to try again. However, you should be aware that this can lead to problems. If the failure means you need to update your working copy, and that update leads to conflicts you must resolve those first.

Select items automatically
: The normal behaviour in the commit dialog is for all modified (versioned) items to be selected for commit automatically. If you prefer to start with nothing selected and pick the items for commit manually, uncheck this box.

Contact the repository on startup
: The Check for Modifications dialog checks the working copy by default, and only contacts the repository when you click Check repository. If you always want to check the repository, you can use this setting to make that action happen automatically.

Show Lock dialog before locking files
: When you select one or more files and then use TortoiseSVN → Lock to take out a lock on those files, on some projects it is customary to write a lock message explaining why you have locked the files. If you do not use lock messages, you can uncheck this box to skip that dialog and lock the files immediately.

: If you use the lock command on a folder, you are always presented with the lock dialog as that also gives you the option to select files for locking.

: If your project is using the `tsvn:lockmsgminsize` property, you will see the lock dialog regardless of this setting because the project *requires* lock messages.

5.29.1.4. TortoiseSVN Colour Settings
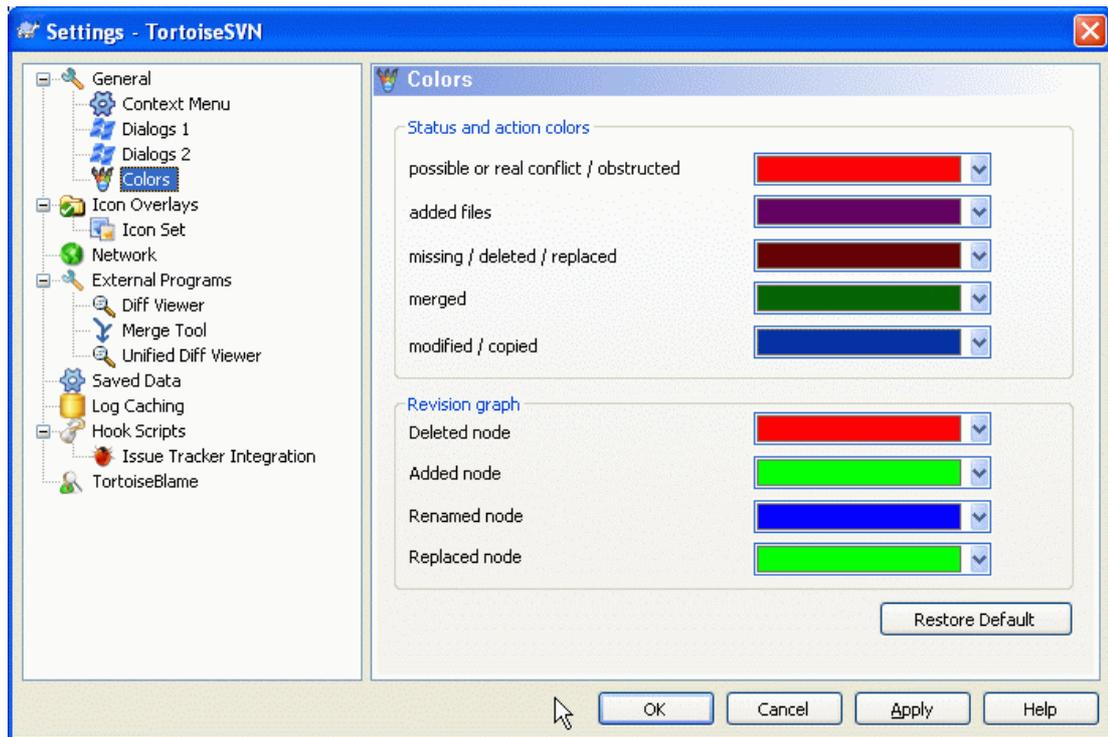
Figure 5.54. The Settings Dialog, Colours Page

This dialog allows you to configure the text colours used in TortoiseSVN's dialogs the way you like them.

Possible or real conflict / obstructed
> A conflict has occurred during update, or may occur during merge. Update is obstructed by an existing unversioned file/folder of the same name as a versioned one.
>
> This colour is also used for error messages in the progress dialogs.

Added files
> Items added to the repository.

Missing / deleted / replaced
> Items deleted from the repository, missing from the working copy, or deleted from the working copy and replaced with another file of the same name.

Merged
> Changes from the repository successfully merged into the WC without creating any conflicts.

Modified / copied
> Add with history, or paths copied in the repository. Also used in the log dialog for entries which include copied items.

Deleted node
> An item which has been deleted from the repository.

Added node
> An item which has been added to the repository, by an add, copy or move operation.

Renamed node
> An item which has been renamed within the repository.

Replaced node
> The original item has been deleted and a new item with the same name replaces it.

5.29.2. Icon Overlay Settings
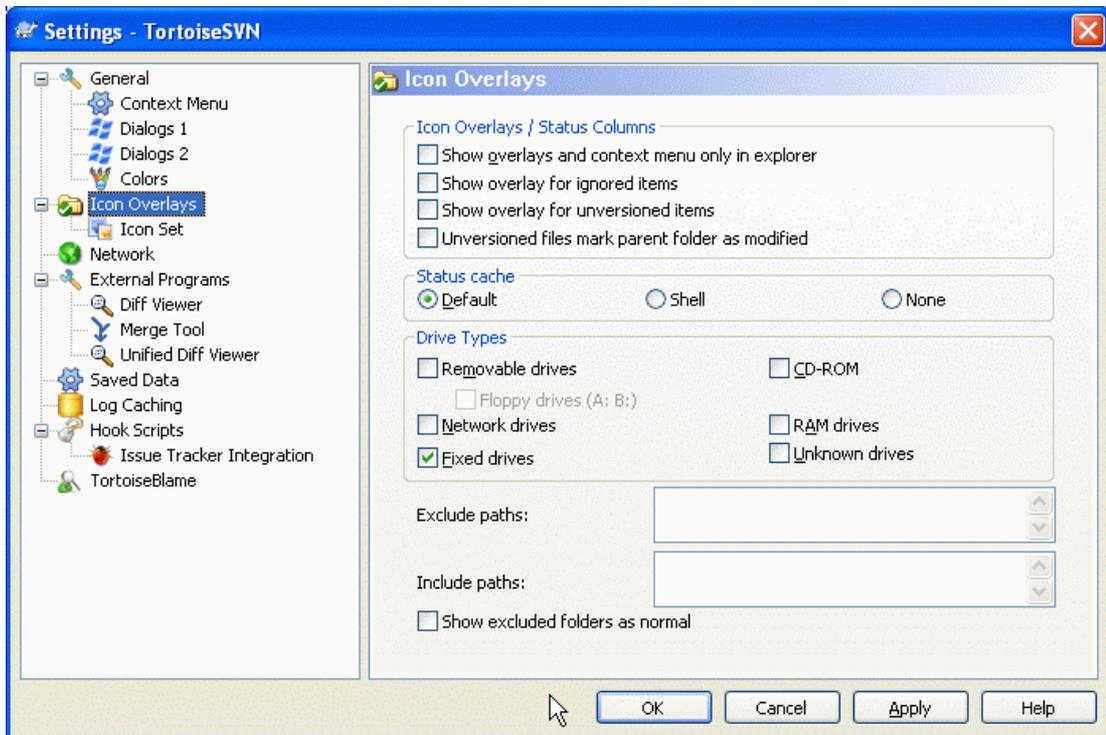
Figure 5.55. The Settings Dialog, Icon Overlays Page

This page allows you to choose the items for which TortoiseSVN will display icon overlays.

By default, overlay icons and context menus will appear in all open/save dialogs as well as in Windows Explorer. If you want them to appear *only* in Windows Explorer, check the Show overlays and context menu only in explorer box.

Ignored items and Unversioned items are not usually given an overlay. If you want to show an overlay in these cases, just check the boxes.

You can also choose to mark folders as modified if they contain unversioned items. This could be useful for reminding you that you have created new files which are not yet versioned. This option is only available when you use the *default* status cache option (see below).

Since it takes quite a while to fetch the status of a working copy, TortoiseSVN uses a cache to store the status so the explorer doesn't get hogged too much when showing the overlays. You can choose which type of cache TortoiseSVN should use according to your system and working copy size here:

Default
> Caches all status information in a separate process (`TSVNCache.exe`). That process watches all drives for changes and fetches the status again if files inside a working copy get modified. The process runs with the least possible priority so other programs don't get hogged because of it. That also means that the status information is not *real time* but it can take a few seconds for the overlays to change.
>
> Advantage: the overlays show the status recursively, i.e. if a file deep inside a working copy is modified, all folders up to the working copy root will also show the modified overlay. And since the process can send notifications to the shell, the overlays on the left tree view usually change too.

Disadvantage: the process runs constantly, even if you're not working on your projects. It also uses around 10-50 MB of RAM depending on number and size of your working copies.

Shell

Caching is done directly inside the shell extension dll, but only for the currently visible folder. Each time you navigate to another folder, the status information is fetched again.

Advantage: needs only very little memory (around 1 MB of RAM) and can show the status in *real time*.

Disadvantage: Since only one folder is cached, the overlays don't show the status recursively. For big working copies, it can take more time to show a folder in explorer than with the default cache. Also the mime-type column is not available.

None

With this setting, the TortoiseSVN does not fetch the status at all in Explorer. Because of that, files don't get an overlay and folders only get a 'normal' overlay if they're versioned. No other overlays are shown, and no extra columns are available either.

Advantage: uses absolutely no additional memory and does not slow down the Explorer at all while browsing.

Disadvantage: Status information of files and folders is not shown in Explorer. To see if your working copies are modified, you have to use the "Check for modifications" dialog.

The next group allows you to select which classes of storage should show overlays. By default, only hard drives are selected. You can even disable all icon overlays, but where's the fun in that?

Network drives can be very slow, so by default icons are not shown for working copies located on network shares.

USB Flash drives appear to be a special case in that the drive type is identified by the device itself. Some appear as fixed drives, and some as removable drives.

The Exclude Paths are used to tell TortoiseSVN those paths for which it should *not* show icon overlays and status columns. This is useful if you have some very big working copies containing only libraries which you won't change at all and therefore don't need the overlays. For example:

`f:\development\SVN\Subversion` will disable the overlays *only* on that specific folder. You still can see the overlays on all files and folder inside that folder.

`f:\development\SVN\Subversion*` will disable the overlays on *all* files and folders whose path starts with `f:\development\SVN\Subversion`. That means you won't see overlays for any files and folders below that path.

The same applies to the Include Paths. Except that for those paths the overlays are shown even if the overlays are disabled for that specific drive type, or by an exclude path specified above.

Users sometimes ask how these three settings interact, and the definitive answer is:

```
if (path is in include list)
  show overlays
if (path is allowed drive type) AND (path is not in exclude list)
  show overlays
```

The include list *always* makes the overlays show. Otherwise, overlays are shown for all marked drive types *unless* the path is excluded.

TSVNCache.exe also uses these paths to restrict its scanning. If you want it to look only in particular folders, disable all drive types and include only the folders you specifically want to be scanned.

> ### Exclude **SUBST** Drives
>
> It is often convenient to use a `SUBST` drive to access your working copies, e.g. using the command
>
> ```
> subst T: C:\TortoiseSVN\trunk\doc
> ```
>
> However this can cause the overlays not to update, as `TSVNCache` will only receive one notification when a file changes, and that is normally for the original path. This means that your overlays on the `subst` path may never be updated.
>
> An easy way to work around this is to exclude the original path from showing overlays, so that the overlays show up on the `subst` path instead.

Sometimes you will exclude areas that contain working copies, which saves TSVNCache from scanning and monitoring for changes, but you still want a visual indication that such folders are versioned. The Show excluded folders as 'normal' checkbox allows you to do this. With this option, versioned folders in any excluded area (drive type not checked, or specifically excluded) will show up as normal and up-to-date, with a green check mark. This reminds you that you are looking at a working copy, even though the folder overlays may not be correct. Files do not get an overlay at all. Note that the context menus still work, even though the overlays are not shown.

As a special exception to this, drives `A:` and `B:` are never considered for the Show excluded folders as 'normal' option. This is because Windows is forced to look on the drive, which can result in a delay of several seconds when starting Explorer, even if your PC does have a floppy drive.

5.29.2.1. Icon Set Selection

Figure 5.56. The Settings Dialog, Icon Set Page

You can change the overlay icon set to the one you like best. Note that if you change overlay set, you may have to restart your computer for the changes to take effect.

5.29.3. Network Settings

Figure 5.57. The Settings Dialog, Network Page

Here you can configure your proxy server, if you need one to get through your company's firewall.

If you need to set up per-repository proxy settings, you will need to use the Subversion servers file to configure this. Use Edit to get there directly. Consult the *Runtime Configuration Area* [http:// svnbook.red-bean.com/en/1.4/svn.advanced.confarea.html] for details on how to use this file.

You can also specify which program TortoiseSVN should use to establish a secure connection to a svn +ssh repository. We recommend that you use TortoisePlink.exe. This is a version of the popular Plink program, and is included with TortoiseSVN, but it is compiled as a Windowless app, so you don't get a DOS box popping up every time you authenticate.

You must specify the full path to the executable. For TortoisePlink.exe this is the standard TortoiseSVN bin directory. Use the Browse button to help locate it.

One side-effect of not having a window is that there is nowhere for any error messages to go, so if authentication fails you will simply get a message saying something like "Unable to write to standard output". For this reason we recommend that you first set up using standard Plink. When everything is working, you can use TortoisePlink with exactly the same parameters.

TortoisePlink does not have any documentation of its own because it is just a minor variant of Plink. Find out about command line parameters from the *PuTTY website* [http://www.chiark.greenend.org.uk/ ~sgtatham/putty/]

To avoid being prompted for a password repeatedly, you might also consider using a password caching tool such as Pageant. This is also available for download from the PuTTY website.

Finally, setting up SSH on server and clients is a non-trivial process which is beyond the scope of this help file. However, you can find a guide in the TortoiseSVN FAQ listed under *Subversion/TortoiseSVN SSH How-To* [http://tortoisesvn.net/ssh_howto].

5.29.4. External Program Settings
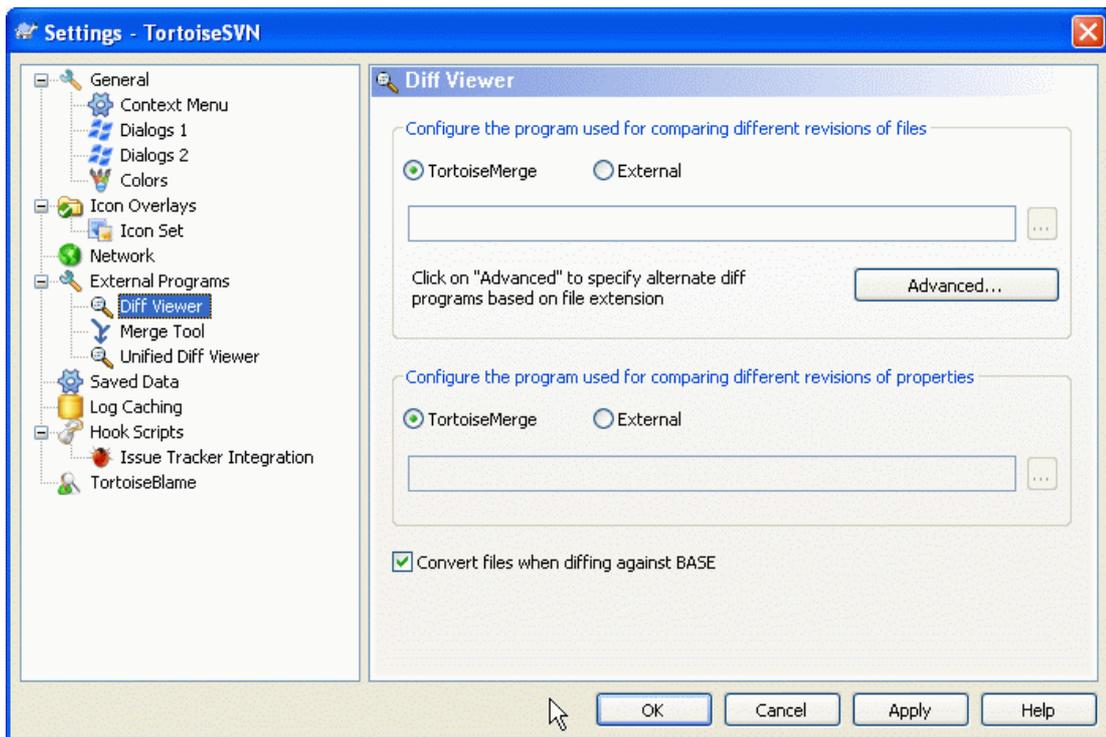
Figure 5.58. The Settings Dialog, Diff Viewer Page

Here you can define your own diff/merge programs that TortoiseSVN should use. The default setting is to use TortoiseMerge which is installed alongside TortoiseSVN.

Read Section 5.10.5, "External Diff/Merge Tools" for a list of some of the external diff/merge programs that people are using with TortoiseSVN.

5.29.4.1. Diff Viewer

An external diff program may be used for comparing different revisions of files. The external program will need to obtain the filenames from the command line, along with any other command line options. TortoiseSVN uses substitution parameters prefixed with %. When it encounters one of these it will substitute the appropriate value. The order of the parameters will depend on the Diff program you use.

%base
    The original file without your changes

%bname
    The window title for the base file

%mine
    Your own file, with your changes

%yname
    The window title for your file

The window titles are not pure filenames. TortoiseSVN treats that as a name to display and creates the names accordingly. So e.g. if you're doing a diff from a file in revision 123 with a file in your working copy, the names will be `filename : revision 123` and `filename : working copy`

For example, with ExamDiff Pro:

```
C:\Path-To\ExamDiff.exe %base %mine
```

or with KDiff3:

```
C:\Path-To\kdiff3.exe %base %mine --L1 %bname --L2 %yname
```

or with WinMerge:

```
C:\Path-To\WinMerge.exe -e -ub -dl %bname -dr %yname %base %mine
```

or with Araxis:

```
C:\Path-To\compare.exe /max /wait /title1:%bname /title2:%yname
    %base %mine
```

If you use the svn:keywords property to expand keywords, and in particular the *revision* of a file, then there may be a difference between files which is purely due to the current value of the keyword. Also if you use svn:eol-style = native the BASE file will have pure LF line endings whereas your file will have CR-LF line endings. TortoiseSVN will normally hide these differences automatically by first parsing the BASE file to expand keywords and line endings before doing the diff operation. However, this can take a long time with large files. If Convert files when diffing against BASE is unchecked then TortoiseSVN will skip pre-processing the files.

You can also specify a different diff tool to use on Subversion properties. Since these tend to be short simple text strings, you may want to use a simpler more compact viewer.

If you have configured an alternate diff tool, you can access TortoiseMerge *and* the third party tool from the context menus. Context menu → Diff uses the primary diff tool, and **Shift**+ Context menu → Diff uses the secondary diff tool.

5.29.4.2. Merge Tool

An external merge program used to resolve conflicted files. Parameter substitution is used in the same way as with the Diff Program.

%base
  the original file without your or the others changes

%bname
  The window title for the base file

%mine
  your own file, with your changes

%yname
  The window title for your file

%theirs
  the file as it is in the repository

%tname
  The window title for the file in the repository

%merged
  the conflicted file, the result of the merge operation

%mname
  The window title for the merged file

For example, with Perforce Merge:

```
C:\Path-To\P4Merge.exe %base %theirs %mine %merged
```

or with KDiff3:

```
C:\Path-To\kdiff3.exe %base %mine %theirs -o %merged
    --L1 %bname --L2 %yname --L3 %tname
```

or with Araxis:

```
C:\Path-To\compare.exe /max /wait /3 /title1:%tname /title2:%bname
    /title3:%yname %theirs %base %mine %merged /a2
```

or with WinMerge (2.8 or later):

```
C:\Path-To\WinMerge.exe %merged
```

5.29.4.3. Diff/Merge Advanced Settings
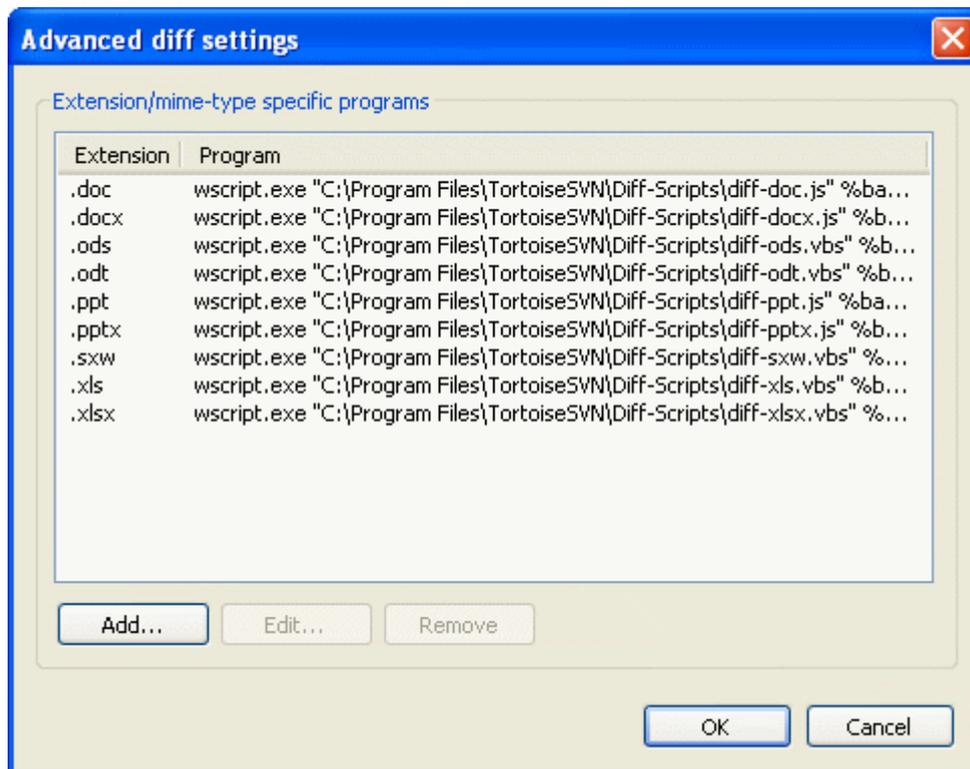
Figure 5.59. The Settings Dialog, Diff/Merge Advanced Dialog

In the advanced settings, you can define a different diff and merge program for every file extension. For instance you could associate Photoshop as the "Diff" Program for `.jpg` files :-) You can also associate the `svn:mime-type` property with a diff or merge program.

To associate using a file extension, you need to specify the extension. Use `.bmp` to describe Windows bitmap files. To associate using the `svn:mime-type` property, specify the mime type, including a slash, for example `text/xml`.

5.29.4.4. Unified Diff Viewer

A viewer program for unified-diff files (patch files). No parameters are required. The Default option is to check for a file association for `.diff` files, and then for `.txt` files. If you don't have a viewer for `.diff` files, you will most likely get NotePad.

The original Windows NotePad program does not behave well on files which do not have standard CR-LF line-endings. Since most unified diff files have pure LF line-endings, they do not view well in NotePad. However, you can download a free NotePad replacement *Notepad2* [http://www.flos-freeware.ch/notepad2.html] which not only displays the line-endings correctly, but also colour codes the added and removed lines.

5.29.5. Saved Data Settings
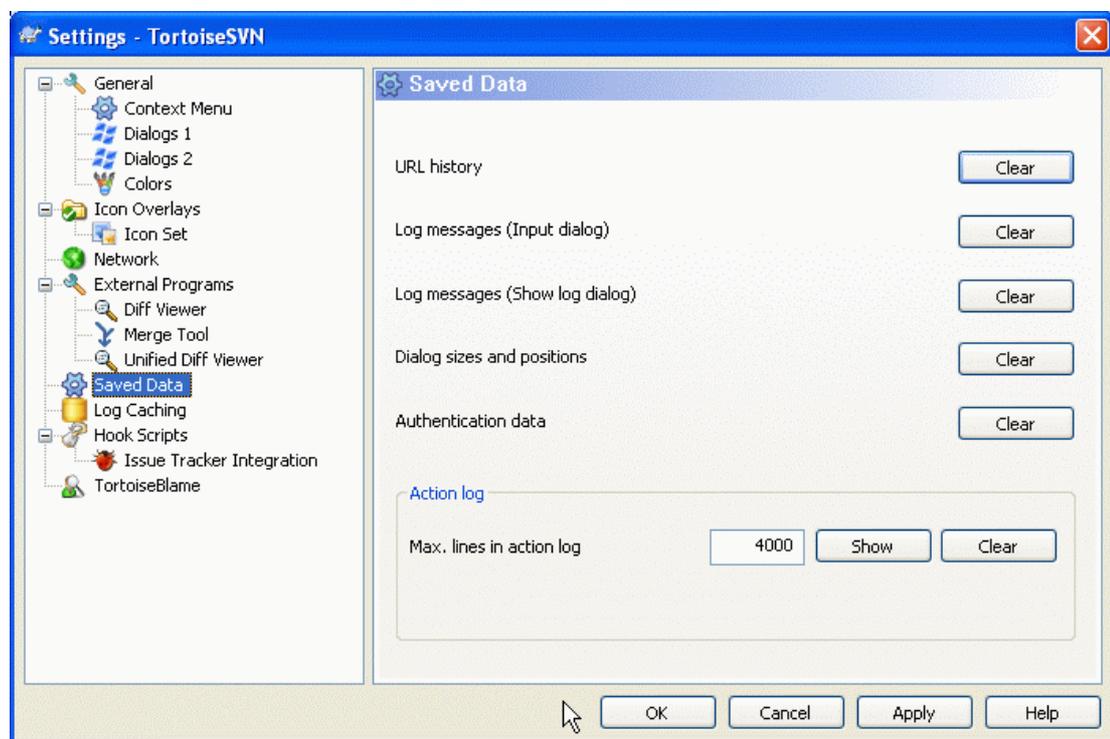
Figure 5.60. The Settings Dialog, Saved Data Page

For your convenience, TortoiseSVN saves many of the settings you use, and remembers where you have been lately. If you want to clear out that cache of data, you can do it here.

URL history
    Whenever you checkout a working copy, merge changes or use the repository browser, TortoiseSVN keeps a record of recently used URLs and offers them in a combo box. Sometimes that list gets cluttered with outdated URLs so it is useful to flush it out periodically.

    If you want to remove a single item from one of the combo boxes you can do that in-place. Just click on the arrow to drop the combo box down, move the mouse over the item you want to remove and type **Shift+Del**.

Log messages (Input dialog)
    TortoiseSVN stores recent commit log messages that you enter. These are stored per repository, so if you access many repositories this list can grow quite large.

Log messages (Show log dialog)

TortoiseSVN caches log messages fetched by the Show Log dialog to save time when you next show the log. If someone else edits a log message and you already have that message cached, you will not see the change until you clear the cache. Log message caching is enabled on the Dialogs 1 tab.

Dialog sizes and positions

Many dialogs remember the size and screen position that you last used.

Authentication data

When you authenticate with a Subversion server, the username and password are cached locally so you don't have to keep entering them. You may want to clear this for security reasons, or because you want to access the repository under a different username ... does John know you are using his PC?

If you want to clear authentication data for one particular server only, read Section 5.1.5, "Authentication" for instructions on how to find the cached data.

Action log

TortoiseSVN keeps a log of everything written to its progress dialogs. This can be useful when, for example, you want to check what happened in a recent update command.

The log file is limited in length and when it grows too big the oldest content is discarded. By default 4000 lines are kept, but you can customize that number.

From here you can view the log file content, and also clear it.

5.29.6. Log Caching
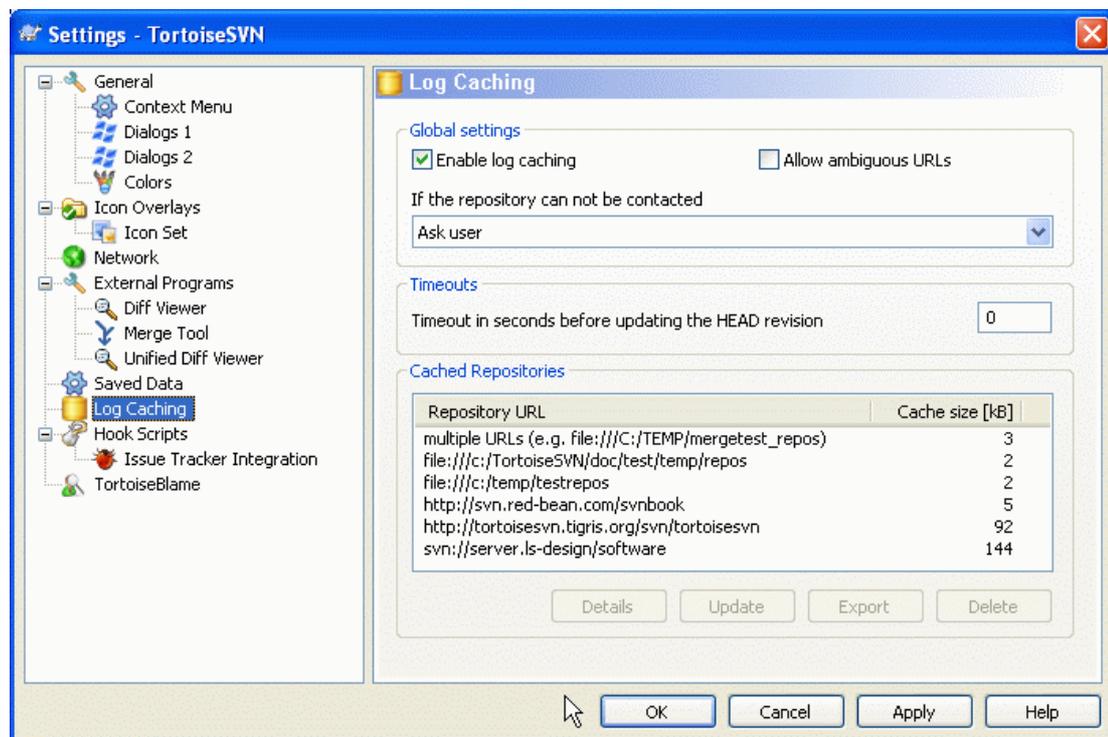
Figure 5.61. The Settings Dialog, Log Cache Page

This dialog allows you to configure the log caching feature of TortoiseSVN, which retains a local copy of log messages and changed paths to avoid time-consuming downloads from the server. Using the log cache can dramatically speed up the log dialog and the revision graph. Another useful feature is that the log messages can still be accessed when offline.

Enable log caching
  Enables log caching whenever log data is requested. If checked, data will be retrieved from the cache when available, and any messages not in the cache will be retrieved from the server and added to the cache.

  If caching is disabled, data will always be retrieved directly from the server and not stored locally.

If the repository cannot be contacted
  If you are working offline, or if the repository server is down, the log cache can still be used to supply log messages already held in the cache. Of course the cache may not be up-to-date, so there are options to allow you to select whether this feature should be used.

  When log data is being taken from the cache without contacting the server, the dialog using those message will show the offline state in its title bar.

Timeout before updating the HEAD revision
  When you invoke the log dialog you will normally want to contact the server to check for any newer log messages. If the timeout set here is non-zero then the server will only be contacted when the timeout has elapsed since the last time contact. This can reduce server round-trips if you open the log dialog frequently and the server is slow, but the data shown may not be completely up-to-date. If you want to use this feature we suggest using a value of 300 (5 minutes) as a compromise.

Below the settings you can see a list of the repositories that are cached locally, and the space used for the cache. If you select one of the repositories you can then use the buttons underneath.

5.29.6.1. Log Cache Statistics
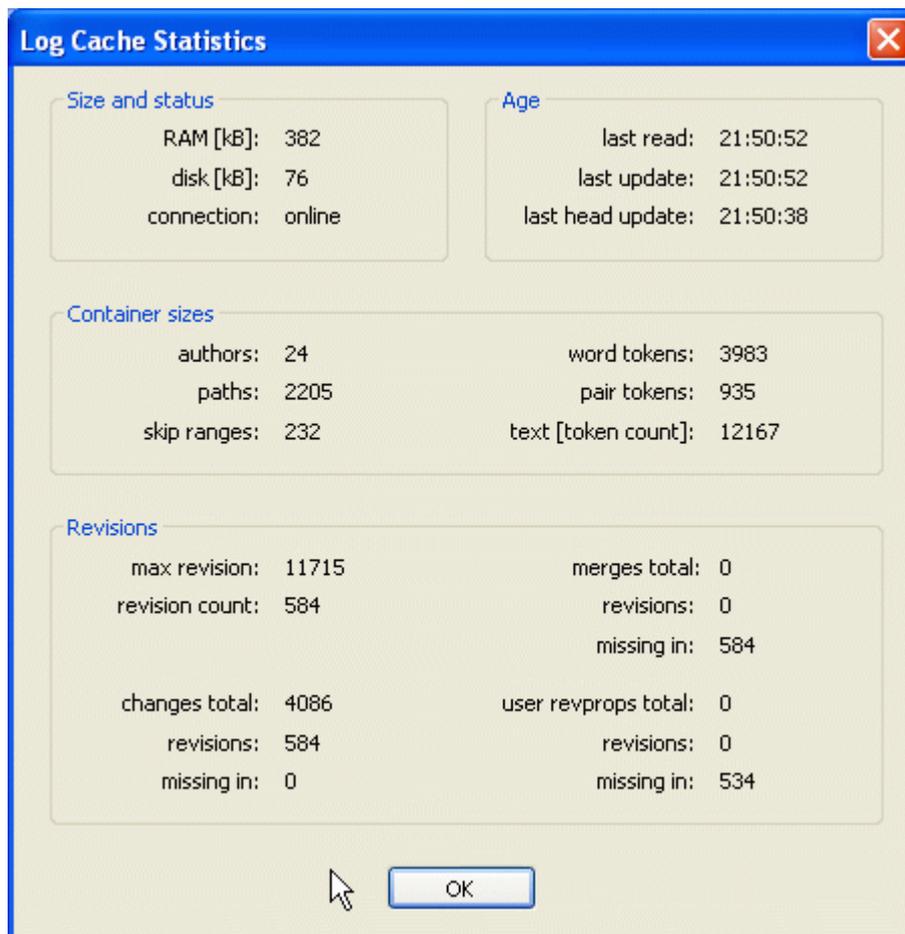
Figure 5.62. The Settings Dialog, Log Cache Statistics

Click on the Details button to see detailed statistics for a particular cache. Many of the fields shown here are mainly of interest to the developers of TortoiseSVN, so they are not all described in detail.

RAM
> The amount of memory required to service this cache.

Disk
> The amount of disk space used for the cache. Data is compressed, so disk usage is generally fairly modest.

Connection
> Shows whether the repository was available last time the cache was used.

Last update
> The last time the cache content was changed.

Last head update
> The last time we requested the HEAD revision from the server.

Authors
> The number of different authors with messages recorded in the cache.

Paths
> The number of paths listed, as you would see using `svn log -v`.

Skip ranges
> The number of revision ranges which we have not fetched, simply because they haven't been requested. This is a measure of the number of holes in the cache.

Max revision
> The highest revision number stored in the cache.

Revision count
> The number of revisions stored in the cache. This is another measure of cache completeness.

Click on the Update to completely refresh the cache and fill in any holes. For a large repository this could be very time consuming, but useful if you are about to go offline and want the best available cache.

Click on the Export button to export the entire cache as a set of CSV files. This could be useful if you want to process the log data using an external program, although it is mainly useful to the developers.

Click on Delete to remove all cached data for the selected repositories. This does not disable caching for the repository so the next time you request log data, a new cache will be created.

5.29.7. Client Side Hook Scripts
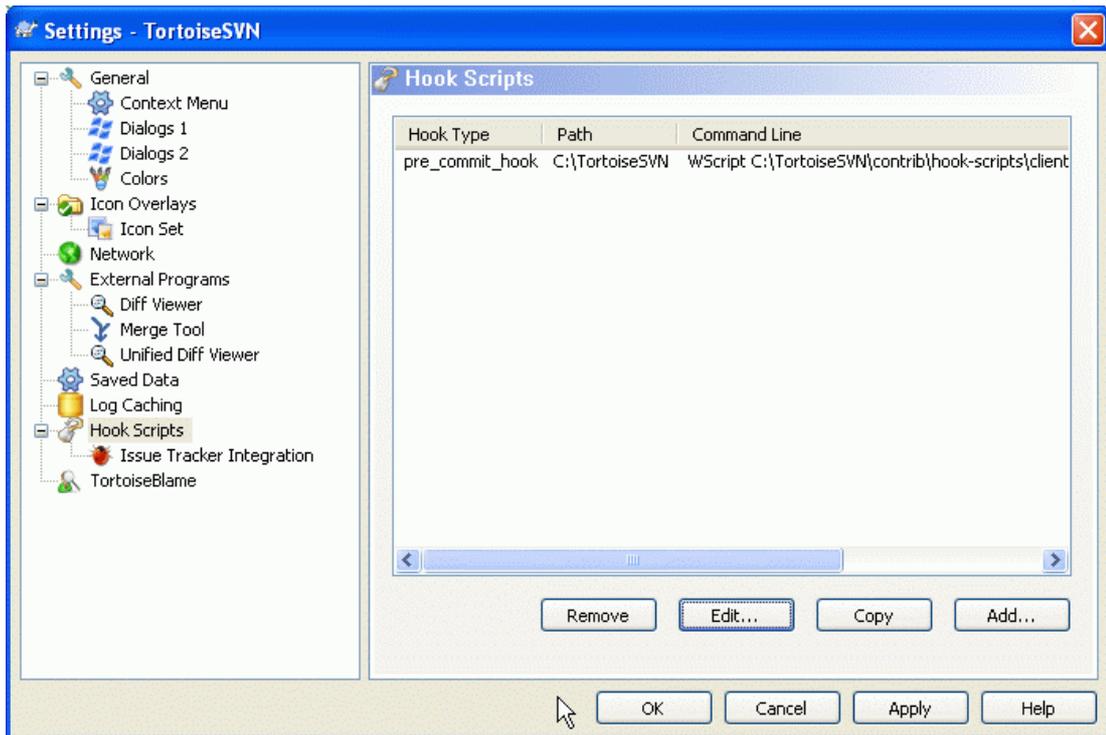
Figure 5.63. The Settings Dialog, Hook Scripts Page

This dialog allows you to set up hook scripts which will be executed automatically when certain Subversion actions are performed. As opposed to the hook scripts explained in Section 4.3, "Hook Scripts", these scripts are executed locally on the client.

One application for such hooks might be to call a program like `SubWCRev.exe` to update version numbers after a commit, and perhaps to trigger a rebuild.

For various security and implementation reasons, hook scripts are defined locally on a machine, rather than as project properties. You define what happens, no matter what someone else commits to the repository. Of course you can always choose to call a script which is itself under version control.
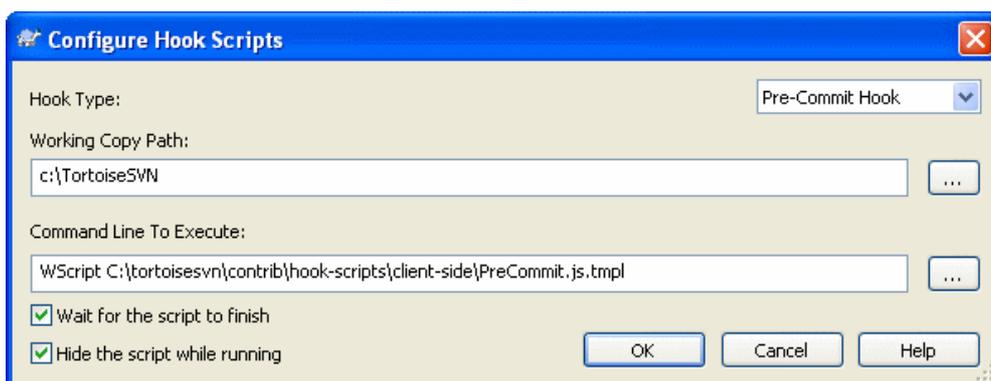
Figure 5.64. The Settings Dialog, Configure Hook Scripts

To add a new hook script, simply click Add and fill in the details.

There are currently six types of hook script available

Start-commit
> Called before the commit dialog is shown. You might want to use this if the hook modifies a versioned file and affects the list of files that need to be committed and/or commit message.

Pre-commit
    Called after the user clicks OK in the commit dialog, and before the actual commit begins.

Post-commit
    Called after the commit finishes (whether successful or not).

Start-update
    Called before the update-to-revision dialog is shown.

Pre-update
    Called before the actual Subversion update begins.

Post-update
    Called after the update finishes (whether successful or not).

A hook is defined for a particular working copy path. You only need to specify the top level path; if you perform an operation in a sub-folder, TortoiseSVN will automatically search upwards for a matching path.

Next you must specify the command line to execute, starting with the path to the hook script or executable. This could be a batch file, an executable file or any other file which has a valid windows file association, eg. a perl script.

The command line includes several parameters which get filled in by TortoiseSVN. The parameters available depend upon which hook is called. Each hook has its own parameters which are passed in the following order:

Start-commit
    `PATH MESSAGEFILE CWD`

Pre-commit
    `PATH DEPTH MESSAGEFILE CWD`

Post-commit
    `PATH DEPTH MESSAGEFILE REVISION ERROR CWD`

Start-update
    `PATH CWD`

Pre-update
    `PATH DEPTH REVISION CWD`

Post-update
    `PATH DEPTH REVISION ERROR CWD`

The meaning of each of these parameters is described here:

PATH
    A path to a temporary file which contains all the paths for which the operation was started. Each path is on a separate line in the temp file.

DEPTH
    The depth with which the commit/update is done.

    Possible values are:

    -2
        `svn_depth_unknown`

    -1
        `svn_depth_exclude`

0

```
svn_depth_empty
```

1

```
svn_depth_files
```

2

```
svn_depth_immediates
```

3

```
svn_depth_infinity
```

MESSAGEFILE
>   Path to a file containing the log message for the commit. The file contains the text in UTF-8 encoding. After successful execution of the start-commit hook, the log message is read back, giving the hook a chance to modify it.

REVISION
>   The repository revision to which the update should be done or after a commit completes.

ERROR
>   Path to a file containing the error message. If there was no error, the file will be empty.

CWD
>   The current working directory with which the script is run. This is set to the common root directory of all affected paths.

If you want the Subversion operation to hold off until the hook has completed, check Wait for the script to finish.

Normally you will want to hide ugly DOS boxes when the script runs, so Hide the script while running is checked by default.

5.29.7.1. Issue Tracker Integration

TortoiseSVN can use a COM plugin to query issue trackers when in the commit dialog. The use of such plugins is described in Section 5.27.2, "Getting Information from the Issue Tracker". If your system administrator has provided you with a plugin, which you have already installed and registered, this is the place to specify how it integrates with your working copy.
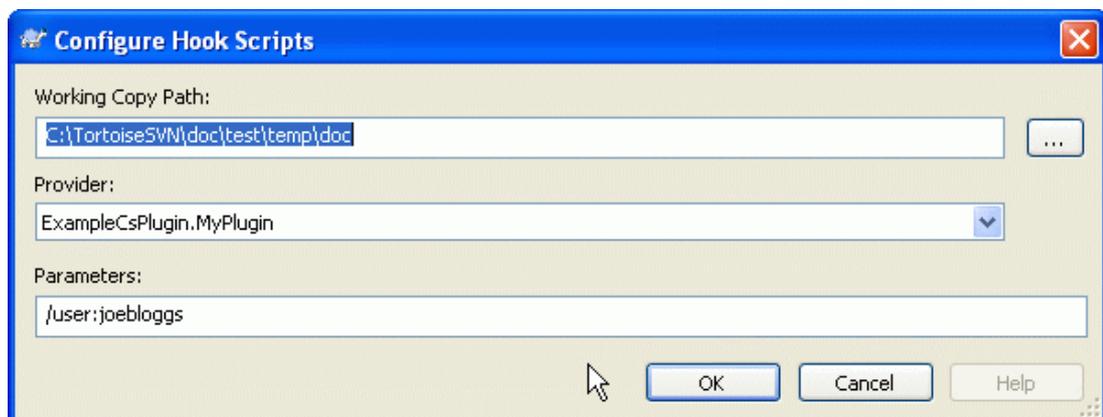
Figure 5.65. The Settings Dialog, Issue Tracker Integration Page

Click on Add... to use the plugin with a particular working copy. Here you can specify the working copy path, choose which plugin to use from a drop down list of all registered issue tracker plugins, and any

parameters to pass. The parameters will be specific to the plugin, but might include your user name on the issue tracker so that the plugin can query for issues which are assigned to you.

5.29.8. TortoiseBlame Settings
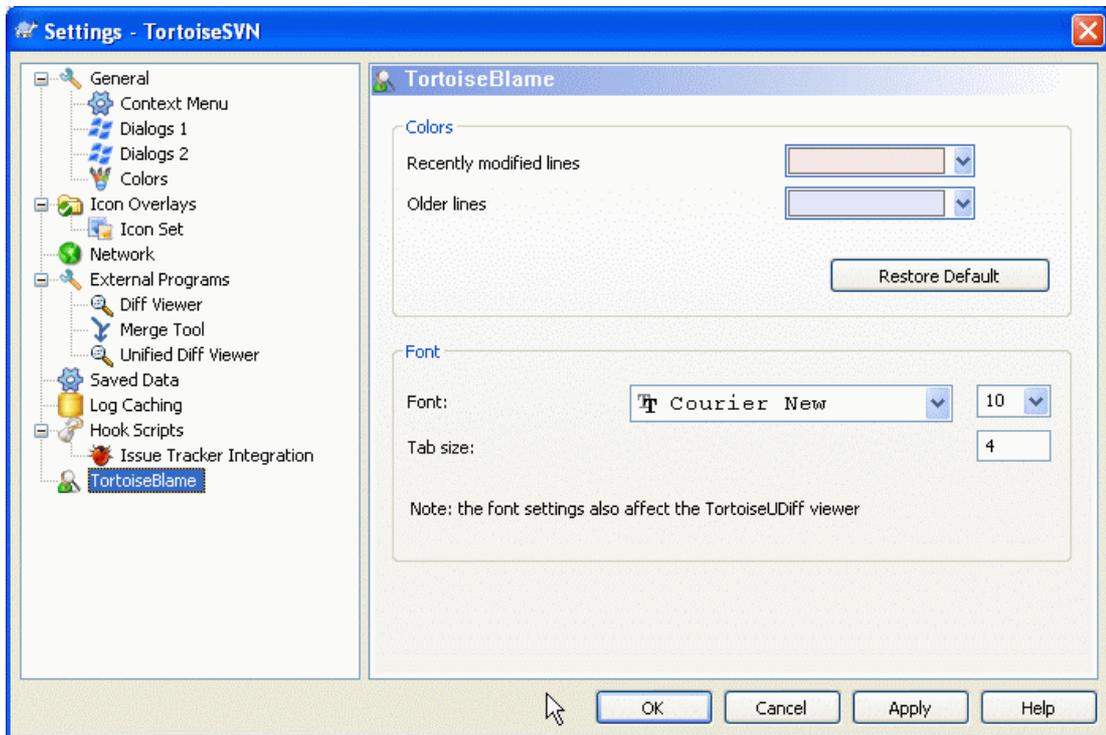
Figure 5.66. The Settings Dialog, TortoiseBlame Page

The settings used by TortoiseBlame are controlled from the main context menu, not directly with TortoiseBlame itself.

Colors
> TortoiseBlame can use the background colour to indicate the age of lines in a file. You set the endpoints by specifying the colours for the newest and oldest revisions, and TortoiseBlame uses a linear interpolation between these colours according to the repository revision indicated for each line.

Font
> You can select the font used to display the text, and the point size to use. This applies both to the file content, and to the author and revision information shown in the left pane.

Tabs
> Defines how many spaces to use for expansion when a tab character is found in the file content.

5.29.9. Registry Settings

A few infrequently used settings are available only by editing the registry directly.

Configuration
> You can specify a different location for the Subversion configuration file using registry location `HKCU\Software\TortoiseSVN\ConfigDir`. This will affect all TortoiseSVN operations.

Cache tray icon
> To add a cache tray icon for the TSVNCache program, create a `DWORD` key with a value of 1 at `HKCU\Software\TortoiseSVN\CacheTrayIcon`. This is really only useful for developers as it allows you to terminate the program gracefully.

Filenames without extensions in auto-completion list
    The auto-completion list shown in the commit message editor displays the names of files listed for commit. To also include these names with extensions removed, create a `DWORD` key with a value of 1 at `HKCU\Software\TortoiseSVN\AutocompleteRemovesExtensions`.

Explorer columns everywhere
    The extra columns the TortoiseSVN adds to the details view in Windows Explorer are normally only active in a working copy. If you want those to be accessible everywhere, not just in working copies, create a `DWORD` key with a value of 1 at `HKCU\Software\TortoiseSVN\ColumnsEveryWhere`.

Merge log separator
    When you merge revisions from another branch, and merge tracking information is available, the log messages from the revisions you merge will be collected to make up a commit log message. A pre-defined string is used to separate the individual log messages of the merged revisions. If you prefer, you can create a `SZ` key at `HKCU\Software\TortoiseSVN\MergeLogSeparator` containing a separator string of your choice.

Always blame changes with TortoiseMerge
    TortoiseSVN allows you to assign external diff viewer. Most such viewers, however, are not suited for change blaming (Section 5.22.2, "Blame Differences"), so you might wish to fall back to TortoiseMerge in this case. To do so, create a `DWORD` key with a value of 1 at `HKCU\Software\TortoiseSVN\DiffBlamesWithTortoiseMerge`.

5.29.10. Subversion Working Folders

VS.NET 2003 when used with web projects can't handle the `.svn` folders that Subversion uses to store its internal information. This is not a bug in Subversion. The bug is in VS.NET 2003 and the frontpage extensions it uses.

Note that the bug is fixed in VS2005 and later versions.

As of Version 1.3.0 of Subversion and TortoiseSVN, you can set the environment variable `SVN_ASP_DOT_NET_HACK`. If that variable is set, then Subversion will use `_svn` folders instead of `.svn` folders. You must restart your shell for that environment variable to take effect. Normally that means rebooting your PC. To make this easier, you can now do this from the general settings page using a simple checkbox - refer to Section 5.29.1, "General Settings".

For more information, and other ways to avoid this problem in the first place, check out the article about this in our *FAQ* [http://tortoisesvn.net/aspdotnethack].

5.30. Final Step

Donate!

Even though TortoiseSVN and TortoiseMerge are free, you can support the developers by sending in patches and play an active role in the development. You can also help to cheer us up during the endless hours we spend in front of our computers.

While working on TortoiseSVN we love to listen to music. And since we spend many hours on the project we need a *lot* of music. Therefore we have set up some wish-lists with our favourite music CDs and DVDs: *http://tortoisesvn.tigris.org/donate.html* [http://tortoisesvn.tigris.org/donate.html] Please also have a look at the list of people who contributed to the project by sending in patches or translations.

Chapter 6. The SubWCRev Program

SubWCRev is Windows console program which can be used to read the status of a Subversion working copy and optionally perform keyword substitution in a template file. This is often used as part of the build process as a means of incorporating working copy information into the object you are building. Typically it might be used to include the revision number in an "About" box.

6.1. The SubWCRev Command Line

SubWCRev reads the Subversion status of all files in a working copy, excluding externals by default. It records the highest commit revision number found, and the commit timestamp of that revision, It also records whether there are local modifications in the working copy, or mixed update revisions. The revision number, update revision range and modification status are displayed on stdout.

SubWCRev.exe is called from the command line or a script, and is controlled using the command line parameters.

```
SubWCRev WorkingCopyPath [SrcVersionFile DstVersionFile] [-nmdfe]
```

`WorkingCopyPath` is the path to the working copy being checked. You can only use SubWCRev on working copies, not directly on the repository. The path may be absolute or relative to the current working directory.

If you want SubWCRev to perform keyword substitution, so that fields like repository revision and URL are saved to a text file, you need to supply a template file `SrcVersionFile` and an output file `DstVersionFile` which contains the substituted version of the template.

There are a number of optional switches which affect the way SubWCRev works. If you use more than one, they must be specified as a single group, eg. `-nm`, not `-n  -m`.

Switch	Description
-n	If this switch is given, SubWCRev will exit with `ERRORLEVEL  7` if the working copy contains local modifications. This may be used to prevent building with uncommitted changes present.
-m	If this switch is given, SubWCRev will exit with `ERRORLEVEL  8` if the working copy contains mixed revisions. This may be used to prevent building with a partially updated working copy.
-d	If this switch is given, SubWCRev will exit with `ERRORLEVEL  9` if the destination file already exists.
-f	If this switch is given, SubWCRev will include the last-changed revision of folders. The default behaviour is to use only files when getting the revision numbers.
-e	If this switch is given, SubWCRev will examine directories which are included with `svn:externals`, but only if they are from the same repository. The default behaviour is to ignore externals.
-x	If this switch is given, SubWCRev will output the revision numbers in HEX.
-X	If this switch is given, SubWCRev will output the revision numbers in HEX, with '0X' prepended.

Table 6.1. List of available command line switches

6.2. Keyword Substitution

If a source and destination files are supplied, SubWCRev copies source to destination, performing keyword substitution as follows:

Keyword	Description
$WCREV$	Replaced with the highest commit revision in the working copy.
$WCDATE$	Replaced with the commit date/time of the highest commit revision. By default, international format is used: `yyyy-mm-dd hh:mm:ss`. Alternatively, you can specify a custom format which will be used with `strftime()`, for example: `$WCDATE=%a %b %d %I:%M:%S %p$`. For a list of available formatting characters, look at the *online reference* [http://www.cppreference.com/stddate/strftime.html].
$WCNOW$	Replaced with the current system date/time. This can be used to indicate the build time. Time formatting is as described above.
$WCRANGE$	Replaced with the update revision range in the working copy. If the working copy is in a consistent state, this will be a single revision. If the working copy contains mixed revisions, either due to being out of date, or due to a deliberate update-to-revision, then the range will be shown in the form 100:200
$WCMIXED$	`$WCMIXED?TText:FText$` is replaced with `TText` if there are mixed update revisions, or `FText` if not.
$WCMODS$	`$WCMODS?TText:FText$` is replaced with `TText` if there are local modifications, or `FText` if not.
$WCURL$	Replaced with the repository URL of the working copy path passed to SubWCRev.
$WCNOW$	Replaced with the current time and date
$WCNOW=	Replaced with the current time and date in standard format
$WCINSVN$	`$WCINSVN?TText:FText$` is replaced with `TText` if the entry is versioned, or `FText` if not.
$WCNEEDSLOCK$	`$WCNEEDSLOCK?TText:FText$` is replaced with `TText` if the entry has the `svn:needs-lock` property set, or `FText` if not.
$WCISLOCKED$	`$WCISLOCKED?TText:FText$` is replaced with `TText` if the entry is locked, or `FText` if not.
$WCLOCKDATE$	Replaced with the lock date
$WCLOCKOWNER$	Replaced with the name of the lock owner
$WCLOCKCOMMENT$	Replaced with the comment of the lock

Table 6.2. List of available command line switches

6.3. Keyword Example

The example below shows how keywords in a template file are substituted in the output file.

```
// Test file for SubWCRev: testfile.tmpl

char *Revision = "$WCREV$";
char *Modified = "$WCMODS?Modified:Not modified$";
char *Date     = "$WCDATE$";
char *Range    = "$WCRANGE$";
char *Mixed    = "$WCMIXED?Mixed revision WC:Not mixed$";
char *URL      = "$WCURL$";

#if $WCMODS?1:0$
#error Source is modified
```

```
#endif

// End of file
```

After running SubWCRev.exe path\to\workingcopy testfile.tmpl testfile.txt, the output file testfile.txt would looks like this:

```
// Test file for SubWCRev: testfile.txt

char *Revision = "3701";
char *Modified = "Modified";
char *Date     = "2005/06/15 11:15:12";
char *Range    = "3699:3701";
char *Mixed    = "Mixed revision WC";
char *URL      = "http://tortoisesvn.tigris.org/svn/tortoisesvn/...
trunk/src/SubWCRev";

#if 1
#error Source is modified
#endif

// End of file
```

> **Tip**
>
> A file like this will be included in the build so you would expect it to be versioned. Be sure to version the template file, not the generated file, otherwise each time you regenerate the version file you need to commit the change, which in turn means the version file needs to be updated.

6.4. COM interface

If you need to access Subversion revision information from other programs, you can use the COM interface of SubWCRev. The object to create is SubWCRev.object, and the following methods are supported:

Method	Description
.GetWCInfo	This method traverses the working copy gathering the revision information. Naturally you must call this before you can access the information using the remaining methods. The first parameter is the path. The second parameter should be true if you want to include folder revisions. Equivalent to the -f command line switch. The third parameter should be true if you want to include svn:externals. Equivalent to the -e command line switch.
.Revision	The highest commit revision in the working copy. Equivalent to $WCREV$
.Date	The commit date/time of the highest commit revision. Equivalent to $WCDATE$
.Author	The author of the highest commit revision, that is, the last person to commit changes to the working copy.
.MinRev	The minimum update revision, as shown in $WCRANGE$
.MaxRev	The maximum update revision, as shown in $WCRANGE$

Method	Description
.HasModifications	True if there are local modifications
.Url	Replaced with the repository URL of the working copy path used in `GetWCInfo`. Equivalent to `$WCURL$`

Table 6.3. COM/automation methods supported

The following example shows how the interface might be used.

```
// testCOM.js - javascript file
// test script for the SubWCRev COM/Automation-object

filesystem = new ActiveXObject("Scripting.FileSystemObject");

SubWCRev1 = new ActiveXObject("SubWCRev.object");
SubWCRev2 = new ActiveXObject("SubWCRev.object");
SubWCRev3 = new ActiveXObject("SubWCRev.object");

SubWCRev1.GetWCInfo(filesystem.GetAbsolutePathName("."), 0, 0);
SubWCRev2.GetWCInfo(filesystem.GetAbsolutePathName(".."), 1, 1);
SubWCRev3.GetWCInfo(filesystem.GetAbsolutePathName("SubWCRev.cpp"),
                    0, 0);

sInfo1 = "Revision = " + SubWCRev1.Revision +
         "\nMin Revision = " + SubWCRev1.MinRev +
         "\nMax Revision = " + SubWCRev1.MaxRev +
         "\nDate = " + SubWCRev1.Date +
         "\nURL = " + SubWCRev1.Url +
         "\nAuthor = " + SubWCRev1.Author +
         "\nHasMods = " + SubWCRev1.HasModifications;
sInfo2 = "Revision = " + SubWCRev2.Revision +
         "\nMin Revision = " + SubWCRev2.MinRev +
         "\nMax Revision = " + SubWCRev2.MaxRev +
         "\nDate = " + SubWCRev2.Date +
         "\nURL = " + SubWCRev2.Url +
         "\nAuthor = " + SubWCRev2.Author +
         "\nHasMods = " + SubWCRev2.HasModifications;
sInfo3 = "Revision = " + SubWCRev3.Revision +
         "\nMin Revision = " + SubWCRev3.MinRev +
         "\nMax Revision = " + SubWCRev3.MaxRev +
         "\nDate = " + SubWCRev3.Date +
         "\nURL = " + SubWCRev3.Url +
         "\nAuthor = " + SubWCRev3.Author +
         "\nHasMods = " + SubWCRev3.HasModifications;

WScript.Echo(sInfo1);
WScript.Echo(sInfo2);
WScript.Echo(sInfo3);
```

Appendix A. Frequently Asked Questions (FAQ)

Because TortoiseSVN is being developed all the time it is sometimes hard to keep the documentation completely up to date. We maintain an *online FAQ* [http://tortoisesvn.tigris.org/faq.html] which contains a selection of the questions we are asked the most on the TortoiseSVN mailing lists `<dev@tortoisesvn.tigris.org>` and `<users@tortoisesvn.tigris.org>`.

We also maintain a project *Issue Tracker* [http://issues.tortoisesvn.net] which tells you about some of the things we have on our To-Do list, and bugs which have already been fixed. If you think you have found a bug, or want to request a new feature, check here first to see if someone else got there before you.

If you have a question which is not answered anywhere else, the best place to ask it is on one of the mailing lists. `<users@tortoisesvn.tigris.org>` is the one to use if you have questions about using TortoiseSVN. If you want to help out with the development of TortoiseSVN, then you should take part in discussions on `<dev@tortoisesvn.tigris.org>`.

Appendix B. How Do I...

This appendix contains solutions to problems/questions you might have when using TortoiseSVN.

B.1. Move/copy a lot of files at once

Moving/Copying single files can be done by using TortoiseSVN → Rename.... But if you want to move/copy a lot of files, this way is just too slow and too much work.

The recommended way is by right-dragging the files to the new location. Simply right-click on the files you want to move/copy without releasing the mouse button. Then drag the files to the new location and release the mouse button. A context menu will appear where you can either choose Context Menu → SVN Copy versioned files here. or Context Menu → SVN Move versioned files here.

B.2. Force users to enter a log message

There are two ways to prevent users from committing with an empty log message. One is specific to TortoiseSVN, the other works for all Subversion clients, but requires access to the server directly.

B.2.1. Hook-script on the server

If you have direct access to the repository server, you can install a pre-commit hook script which rejects all commits with an empty or too short log message.

In the repository folder on the server, there's a sub-folder `hooks` which contains some example hook scripts you can use. The file `pre-commit.tmpl` contains a sample script which will reject commits if no log message is supplied, or the message is too short. The file also contains comments on how to install/use this script. Just follow the instructions in that file.

This method is the recommended way if your users also use other Subversion clients than TortoiseSVN. The drawback is that the commit is rejected by the server and therefore users will get an error message. The client can't know before the commit that it will be rejected. If you want to make TortoiseSVN have the OK button disabled until the log message is long enough then please use the method described below.

B.2.2. Project properties

TortoiseSVN uses properties to control some of its features. One of those properties is the `tsvn:logminsize` property.

If you set that property on a folder, then TortoiseSVN will disable the OK button in all commit dialogs until the user has entered a log message with at least the length specified in the property.

For detailed information on those project properties, please refer to Section 5.17, "Project Settings"

B.3. Update selected files from the repository

Normally you update your working copy using TortoiseSVN → Update. But if you only want to pick up some new files that a colleague has added without merging in any changes to other files at the same time, you need a different approach.

Use TortoiseSVN → Check for Modifications. and click on Check repository to see what has changed in the repository. Select the files you want to update locally, then use the context menu to update just those files.

B.4. Roll back (Undo) revisions in the repository

B.4.1. Use the revision log dialog

The easiest way to revert the changes from a single revision, or from a range of revisions, is to use the revision log dialog. This is also the method to use of you want to discard recent changes and make an earlier revision the new HEAD.

1. Select the file or folder in which you need to revert the changes. If you want to revert all changes, this should be the top level folder.

2. Select TortoiseSVN → Show Log to display a list of revisions. You may need to use Show All or Next 100 to show the revision(s) you are interested in.

3. Select the revision you wish to revert. If you want to undo a range of revisions, select the first one and hold the **Shift** key while selecting the last one. Note that for multiple revisions, the range must be unbroken with no gaps. Right click on the selected revision(s), then select Context Menu → Revert changes from this revision.

4. Or if you want to make an earlier revision the new HEAD revision, right click on the selected revision, then select Context Menu → Revert to this revision. This will discard *all* changes after the selected revision.

You have reverted the changes within your working copy. Check the results, then commit the changes.

B.4.2. Use the merge dialog

To undo a larger range of revisions, you can use the Merge dialog. The previous method uses merging behind the scenes; this method uses it explicitly.

1. In your working copy select TortoiseSVN → Merge.

2. In the From: field enter the full folder URL of the branch or tag containing the changes you want to revert in your working copy. This should come up as the default URL.

3. In the From Revision field enter the revision number that you are currently at. If you are sure there is no-one else making changes, you can use the HEAD revision.

4. make sure the Use "From:" URL checkbox is checked.

5. In the To Revision field enter the revision number that you want to revert to, namely the one *before* the first revision to be reverted.

6. Click OK to complete the merge.

You have reverted the changes within your working copy. Check the results, then commit the changes.

B.4.3. Use `svndumpfilter`

Since TortoiseSVN never loses data, your "rolled back" revisions still exist as intermediate revisions in the repository. Only the HEAD revision was changed to a previous state. If you want to make revisions disappear completely from your repository, erasing all trace that they ever existed, you have to use more extreme measures. Unless there is a really good reason to do this, it is *not recommended*. One possible reason would be that someone committed a confidential document to a public repository.

The only way to remove data from the repository is to use the Subversion command line tool `svnadmin`. You can find a description of how this works in the *Repository Maintenance* [http://svnbook.red-bean.com/en/1.4/svn.reposadmin.maint.html].

B.5. Compare two revisions of a file

If you want to compare two revisions in a file's history, for example revisions 100 and 200 of the same file, just use TortoiseSVN → Show Log to list the revision history for that file. Pick the two revisions you want to compare then use Context Menu → Compare Revisions.

If you want to compare the same file in two different trees, for example the trunk and a branch, you can use the repository browser to open up both trees, select the file in both places, then use Context Menu → Compare Revisions.

If you want to compare two trees to see what has changed, for example the trunk and a tagged release, you can use TortoiseSVN → Revision Graph Select the two nodes to compare, then use Context Menu → Compare HEAD Revisions. This will show a list of changed files, and you can then select individual files to view the changes in detail. Alternatively use Context Menu → Unified Diff of HEAD Revisions to see a summary of all differences, with minimal context.

B.6. Include a common sub-project

Sometimes you will want to include another project within your working copy, perhaps some library code. You don't want to make a duplicate of this code in your repository because then you would lose connection with the original (and maintained) code. Or maybe you have several projects which share core code. There are at least 3 ways of dealing with this.

B.6.1. Use svn:externals

Set the `svn:externals` property for a folder in your project. This property consists of one or more lines; each line has the name of a sub-folder which you want to use as the checkout folder for common code, and the repository URL that you want to be checked out there. For full details refer to Section 5.2.4, "Referenced Projects".

Commit the new folder. Now when you update, Subversion will pull a copy of that project from its repository into your working copy. The sub-folders will be created automatically if required. Each time you update your main working copy, you will also receive the latest version of all external projects.

If the external project is in the same repository, any changes you make there there will be included in the commit list when you commit your main project.

If the external project is in a different repository, any changes you make to the external project will be notified when you commit the main project, but you have to commit those external changes separately.

Of the three methods described, this is the only one which needs no setup on the client side. Once externals are specified in the folder properties, all clients will get populated folders when they update.

B.6.2. Use a nested working copy

Create a new folder within your project to contain the common code, but do not add it to Subversion

Select TortoiseSVN → Checkout for the new folder and checkout a copy of the common code into it. You now have a separate working copy nested within your main working copy.

The two working copies are independent. When you commit changes to the parent, changes to the nested WC are ignored. Likewise when you update the parent, the nested WC is not updated.

B.6.3. Use a relative location

If you use the same common core code in several projects, and you do not want to keep multiple working copies of it for every project that uses it, you can just check it out to a separate location which is related to all the other projects which use it. For example:

```
C:\Projects\Proj1
C:\Projects\Proj2
C:\Projects\Proj3
C:\Projects\Common
```

and refer to the common code using a relative path, eg. `..\..\Common\DSPcore`.

If your projects are scattered in unrelated locations you can use a variant of this, which is to put the common code in one location and use drive letter substitution to map that location to something you can hard code in your projects, eg. Checkout the common code to `D:\Documents\Framework` or `C:\Documents and Settings\{login}\My Documents\framework` then use

```
SUBST X: "D:\Documents\framework"
```

to create the drive mapping used in your source code. Your code can then use absolute locations.

```
#include "X:\superio\superio.h"
```

This method will only work in an all-PC environment, and you will need to document the required drive mappings so your team know where these mysterious files are. This method is strictly for use in closed development environments, and not recommended for general use.

B.7. Create a shortcut to a repository

If you frequently need to open the repository browser at a particular location, you can create a desktop shortcut using the automation interface to TortoiseProc. Just create a new shortcut and set the target to:

```
TortoiseProc.exe /command:repobrowser /path:"url/to/repository"
```

Of course you need to include the real repository URL.

B.8. Ignore files which are already versioned

If you accidentally added some files which should have been ignored, how do you get them out of version control without losing them? Maybe you have your own IDE configuration file which is not part of the project, but which took you a long time to set up just the way you like it.

If you have not yet committed the add, then all you have to do is use TortoiseSVN → Revert... to undo the add. You should then add the file(s) to the ignore list so they don't get added again later by mistake.

If the files are already in the repository, you have to do a little more work.

1. Hold the **Shift** key to get the extended context menu and use TortoiseSVN → Delete (keep local) to mark the file/folder for deletion from the repository without losing the local copy.

2. TortoiseSVN → Commit the parent folder.

3. Add the file/folder to the ignore list so you don't get into the same trouble again.

B.9. Unversion a working copy

If you have a working copy which you want to convert back to a plain folder tree without the `.svn` directories, you can simply export it to itself. Read Section 5.25.1, "Removing a working copy from version control" to find out how.

B.10. Remove a working copy

If you have a working copy which you no longer need, how do you get rid of it cleanly? Easy - just delete it in Windows Explorer! Working copies are private local entities, and they are self-contained.

Appendix C. Useful Tips For Administrators

This appendix contains solutions to problems/questions you might have when you are responsible for deploying TortoiseSVN to multiple client computers.

## C.1. Deploy TortoiseSVN via group policies

The TortoiseSVN installer comes as an MSI file, which means you should have no problems adding that MSI file to the group policies of your domain controller.

A good walk-through on how to do that can be found in the knowledge base article 314934 from Microsoft: *http://support.microsoft.com/?kbid=314934* [http://support.microsoft.com/?kbid=314934].

Versions 1.3.0 and later of TortoiseSVN must be installed under *Computer Configuration* and not under *User Configuration*. This is because those versions need the new CRT and MFC DLLs, which can only be deployed *per computer* and not *per user*. If you really must install TortoiseSVN on a per user basis, then you must first install the MFC and CRT package version 8 from Microsoft on each computer you want to install TortoiseSVN as per user.

C.2. Redirect the upgrade check

TortoiseSVN checks if there's a new version available every few days. If there is a newer version available, a dialog shows up informing the user about that.
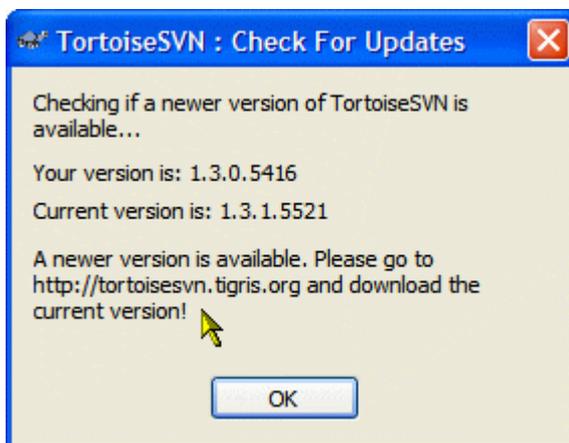
Figure C.1. The upgrade dialog

If you're responsible for a lot of users in your domain, you might want your users to use only versions you have approved and not have them install always the latest version. You probably don't want that upgrade dialog to show up so your users don't go and upgrade immediately.

Versions 1.4.0 and later of TortoiseSVN allow you to redirect that upgrade check to your intranet server. You can set the registry key `HKCU\Software\TortoiseSVN\UpdateCheckURL` (string value) to an URL pointing to a text file in your intranet. That text file must have the following format:

```
1.4.1.6000
A new version of TortoiseSVN is available for you to download!
http://192.168.2.1/downloads/TortoiseSVN-1.4.1.6000-svn-1.4.0.msi
```

The first line in that file is the version string. You must make sure that it matches the exact version string of the TortoiseSVN installation package. The second line is a custom text, shown in the upgrade dialog. You can write there whatever you want. Just note that the space in the upgrade dialog is limited. Too long messages will get truncated! The third line is the URL to the new installation package. This URL is opened when the user clicks on the custom message label in the upgrade dialog. You can also just point the user to a web page instead of the MSI file directly. The URL is opened with the default web browser, so if you specify a web page, that page is opened and shown to the user. If you specify the MSI package, the browser will ask the user to save the MSI file locally.

C.3. Setting the `SVN_ASP_DOT_NET_HACK` environment variable

As of version 1.4.0 and later, the TortoiseSVN installer doesn't provide the user with the option to set the `SVN_ASP_DOT_NET_HACK` environment variable anymore, since that caused many problems and confusions with users which always install *everything* no matter if they know what it is for.

But that option is only hidden for the user. You still can force the TortoiseSVN installer to set that environment variable by setting the `ASPDOTNETHACK` property to TRUE. For example, you can start the installer like this:

```
msiexec /i TortoiseSVN-1.4.0.msi ASPDOTNETHACK=TRUE
```

C.4. Disable context menu entries

As of version 1.5.0 and later, TortoiseSVN allows you to disable (actually, hide) context menu entries. Since this is a feature which should not be used lightly but only if there is a compelling reason, there is no GUI for this and it has to be done directly in the registry. This can be used to disable certain commands for users who should not use them. But please note that only the context menu entries in the *explorer* are hidden, and the commands are still available through other means, e.g. the command line or even other dialogs in TortoiseSVN itself!

The registry keys which hold the information on which context menus to show are `HKEY_CURRENT_USER\Software\TortoiseSVN\ContextMenuEntriesMaskLow` and `HKEY_CURRENT_USER\Software\TortoiseSVN\ContextMenuEntriesMaskHigh`.

Each of these registry entries is a `DWORD` value, with each bit corresponding to a specific menu entry. A set bit means the corresponding menu entry is deactivated.

Value	Menu entry
0x0000000000000001	Checkout
0x0000000000000002	Update
0x0000000000000004	Commit
0x0000000000000008	Add
0x0000000000000010	Revert
0x0000000000000020	Cleanup
0x0000000000000040	Resolve
0x0000000000000080	Switch
0x0000000000000100	Import
0x0000000000000200	Export
0x0000000000000400	Create Repository here
0x0000000000000800	Branch/Tag

Value	Menu entry
0x0000000000001000	Merge
0x0000000000002000	Delete
0x0000000000004000	Rename
0x0000000000008000	Update to revision
0x0000000000010000	Diff
0x0000000000020000	Show Log
0x0000000000040000	Edit Conflicts
0x0000000000080000	Relocate
0x0000000000100000	Check for modifications
0x0000000000200000	Ignore
0x0000000000400000	Repository Browser
0x0000000000800000	Blame
0x0000000001000000	Create Patch
0x0000000002000000	Apply Patch
0x0000000004000000	Revision graph
0x0000000008000000	Lock
0x0000000010000000	Remove Lock
0x0000000020000000	Properties
0x0000000040000000	Diff with URL
0x0000000080000000	Delete unversioned items
0x2000000000000000	Settings
0x4000000000000000	Help
0x8000000000000000	About

Table C.1. Menu entries and their values

Example: to disable the "Relocate" the "Delete unversioned items" and the "Settings" menu entries, add the values assigned to the entries like this:

```
  0x0000000000080000
+ 0x0000000080000000
+ 0x2000000000000000
= 0x2000000080080000
```

The lower DWORD value (0x80080000) must then be stored in HKEY_CURRENT_USER\Software \TortoiseSVN\ContextMenuEntriesMaskLow, the higher DWORD value (0x20000000) in HKEY_CURRENT_USER\Software\TortoiseSVN\ContextMenuEntriesMaskHigh.

To enable the menu entries again, simply delete the two registry keys.

Appendix D. Automating TortoiseSVN

Since all commands for TortoiseSVN are controlled through command line parameters, you can automate it with batch scripts or start specific commands and dialogs from other programs (e.g. your favourite text editor).

> **Important**
>
> Remember that TortoiseSVN is a GUI client, and this automation guide shows you how to make the TortoiseSVN dialogs appear to collect user input. If you want to write a script which requires no input, you should use the official Subversion command line client instead.

D.1. TortoiseSVN Commands

The TortoiseSVN GUI program is called `TortoiseProc.exe`. All commands are specified with the parameter `/command:abcd` where `abcd` is the required command name. Most of these commands need at least one path argument, which is given with `/path:"some\path"`. In the following table the command refers to the `/command:abcd` parameter and the path refers to the `/path:"some\path"` parameter.

Since some of the commands can take a list of target paths (e.g. committing several specific files) the `/path` parameter can take several paths, separated by a `*` character.

TortoiseSVN uses temporary files to pass multiple arguments between the shell extension and the main program. From TortoiseSVN 1.5.0 on and later, `/notempfile` parameter is obsolete and there is no need to add it anymore.

The progress dialog which is used for commits, updates and many more commands usually stays open after the command has finished until the user presses the OK button. This can be changed by checking the corresponding option in the settings dialog. But using that setting will close the progress dialog, no matter if you start the command from your batch file or from the TortoiseSVN context menu.

To specify a different location of the configuration file, use the parameter `/configdir:"path\to\config\directory"`. This will override the default path, including any registry setting.

To close the progress dialog at the end of a command automatically without using the permanent setting you can pass the `/closeonend` parameter.

- `/closeonend:0` don't close the dialog automatically

- `/closeonend:1` auto close if no errors

- `/closeonend:2` auto close if no errors and conflicts

- `/closeonend:3` auto close if no errors, conflicts and merges

- `/closeonend:4` auto close if no errors, conflicts and merges for local operations

The table below lists all the commands which can be accessed using the TortoiseProc.exe command line. As described above, these should be used in the form `/command:abcd`. In the table, the `/command` prefix is omitted to save space.

Command	Description
:about	Shows the about dialog. This is also shown if no command is given.
:log	Opens the log dialog. The `/path` specifies the file or folder for which the log should be shown. Three additional options can be set: `/startrev:xxx`, `/endrev:xxx` and `/strict`

Command	Description
:checkout	Opens the checkout dialog. The `/path` specifies the target directory and the `/url` specifies the URL to checkout from.
:import	Opens the import dialog. The `/path` specifies the directory with the data to import.
:update	Updates the working copy in `/path` to HEAD. If the option `/rev` is given then a dialog is shown to ask the user to which revision the update should go. To avoid the dialog specify a revision number `/rev:1234`. Other options are `/nonrecursive` and `/ignoreexternals`.
:commit	Opens the commit dialog. The `/path` specifies the target directory or the list of files to commit. You can also specify the `/logmsg` switch to pass a predefined log message to the commit dialog. Or, if you don't want to pass the log message on the command line, use `/logmsgfile:path`, where `path` points to a file containing the log message. To pre-fill the bug ID box (in case you've set up integration with bug trackers properly), you can use the `/bugid:"the bug id here"` to do that.
:add	Adds the files in `/path` to version control.
:revert	Reverts local modifications of a working copy. The `/path` tells which items to revert.
:cleanup	Cleans up interrupted or aborted operations and unlocks the working copy in `/path`.
:resolve	Marks a conflicted file specified in `/path` as resolved. If `/noquestion` is given, then resolving is done without asking the user first if it really should be done.
:repocreate	Creates a repository in `/path`
:switch	Opens the switch dialog. The `/path` specifies the target directory.
:export	Exports the working copy in `/path` to another directory. If the `/path` points to an unversioned directory, a dialog will ask for an URL to export to the directory in `/path`.
:merge	Opens the merge dialog. The `/path` specifies the target directory.
:mergeall	Opens the merge all dialog. The `/path` specifies the target directory.
:copy	Brings up the branch/tag dialog. The `/path` is the working copy to branch/tag from. And the `/url` is the target URL. You can also specify the `/logmsg` switch to pass a predefined log message to the branch/tag dialog. Or, if you don't want to pass the log message on the command line, use `/logmsgfile:path`, where `path` points to a file containing the log message.
:settings	Opens the settings dialog.
:remove	Removes the file(s) in `/path` from version control.
:rename	Renames the file in `/path`. The new name for the file is asked with a dialog. To avoid the question about renaming similar files in one step, pass `/noquestion`.
:diff	Starts the external diff program specified in the TortoiseSVN settings. The `/path` specifies the first file. If the option `/path2` is set, then the diff program is started with those two files. If `/path2` is omitted, then the diff is done between the file in `/path` and its BASE. To explicitly set the revision numbers use `/startrev:xxx` and `/endrev:xxx`. If `/blame` is set and `/path2` is not set, then the diff is done by first blaming the files with the given revisions.

Command	Description
:showcompare	Depending on the URLs and revisions to compare, this either shows a unified diff (if the option `unified` is set), a dialog with a list of files that have changed or if the URLs point to files starts the diff viewer for those two files. The options `url1`, `url2`, `revision1` and `revision2` must be specified. The options `pegrevision`, `ignoreancestry`, `blame` and `unified` are optional.
:conflicteditor	Starts the conflict editor specified in the TortoiseSVN settings with the correct files for the conflicted file in `/path`.
:relocate	Opens the relocate dialog. The `/path` specifies the working copy path to relocate.
:help	Opens the help file.
:repostatus	Opens the check-for-modifications dialog. The `/path` specifies the working copy directory.
:repobrowser	Starts the repository browser dialog, pointing to the URL of the working copy given in `/path` or `/path` points directly to an URL. An additional option `/rev:xxx` can be used to specify the revision which the repository browser should show. If the `/rev:xxx` is omitted, it defaults to HEAD. If `/path` points to an URL, the `/projectpropertiespath:path/to/wc` specifies the path from where to read and use the project properties.
:ignore	Adds all targets in `/path` to the ignore list, i.e. adds the `svn:ignore` property to those files.
:blame	Opens the blame dialog for the file specified in `/path`. If the options `/startrev` and `/endrev` are set, then the dialog asking for the blame range is not shown but the revision values of those options are used instead. If the option `/line:nnn` is set, TortoiseBlame will open with the specified line number showing. The options `/ignoreeol`, `/ignorespaces` and `/ignoreallspaces` are also supported.
:cat	Saves a file from an URL or working copy path given in `/path` to the location given in `/savepath:path`. The revision is given in `/revision:xxx`. This can be used to get a file with a specific revision.
:createpatch	Creates a patch file for the path given in `/path`.
:revisiongraph	Shows the revision graph for the path given in `/path`.
:lock	Locks a file or all files in a directory given in `/path`. The 'lock' dialog is shown so the user can enter a comment for the lock.
:unlock	Unlocks a file or all files in a directory given in `/path`.
:rebuildiconcache	Rebuilds the windows icon cache. Only use this in case the windows icons are corrupted. A side effect of this (which can't be avoided) is that the icons on the desktop get rearranged. To suppress the message box, pass `/noquestion`.
:properties	Shows the properties dialog for the path given in `/path`.

Table D.1. List of available commands and options

Examples (which should be entered on one line):

```
TortoiseProc.exe /command:commit
               /path:"c:\svn_wc\file1.txt*c:\svn_wc\file2.txt"
               /logmsg:"test log message" /closeonend:0

TortoiseProc.exe /command:update /path:"c:\svn_wc\" /closeonend:0

TortoiseProc.exe /command:log /path:"c:\svn_wc\file1.txt"
               /startrev:50 /endrev:60 /closeonend:0
```

D.2. TortoiseIDiff Commands

The image diff tool has a few command line options which you can use to control how the tool is started. The program is called `TortoiseIDiff.exe`.

The table below lists all the options which can be passed to the image diff tool on the command line.

Option	Description
:left	Path to the file shown on the left.
:lefttitle	A title string. This string is used in the image view title instead of the full path to the image file.
:right	Path to the file shown on the right.
:righttitle	A title string. This string is used in the image view title instead of the full path to the image file.
:overlay	If specified, the image diff tool switches to the overlay mode (alpha blend).
:fit	If specified, the image diff tool fits both images together.
:showinfo	Shows the image info box.

Table D.2. List of available options

Example (which should be entered on one line):

```
TortoiseIDiff.exe /left:"c:\images\img1.jpg" /lefttitle:"image 1"
               /right:"c:\images\img2.jpg" /righttitle:"image 2"
               /fit /overlay
```

Appendix E. Command Line Interface Cross Reference

Sometimes this manual refers you to the main Subversion documentation, which describes Subversion in terms of the Command Line Interface (CLI). To help you understand what TortoiseSVN is doing behind the scenes, we have compiled a list showing the equivalent CLI commands for each of TortoiseSVN's GUI operations.

> **Note**
>
> Even though there are CLI equivalents to what TortoiseSVN does, remember that TortoiseSVN does *not* call the CLI but uses the Subversion library directly.

If you think you have found a bug in TortoiseSVN, we may ask you to try to reproduce it using the CLI, so that we can distinguish TortoiseSVN issues from Subversion issues. This reference tells you which command to try.

E.1. Conventions and Basic Rules

In the descriptions which follow, the URL for a repository location is shown simply as `URL`, and an example might be `http://tortoisesvn.tigris.org/svn/tortoisesvn/trunk`. The working copy path is shown simply as `PATH`, and an example might be `C:\TortoiseSVN\trunk`.

> **Important**
>
> Because TortoiseSVN is a Windows Shell Extension, it is not able to use the notion of a current working directory. All working copy paths must be given using the absolute path, not a relative path.

Certain items are optional, and these are often controlled by checkboxes or radio buttons in TortoiseSVN. These options are shown in [square brackets] in the command line definitions.

E.2. TortoiseSVN Commands

E.2.1. Checkout

```
svn checkout [-N] [--ignore-externals] [-r rev] URL PATH
```

If **Only checkout the top folder** is checked, use the `-N` switch.

If **Omit externals** is checked, use the `--ignore-externals` switch.

If you are checking out a specific revision, specify that after the URL using `-r` switch.

E.2.2. Update

```
svn info URL_of_WC
svn update [-r rev] PATH
```

Updating multiple items is currently not an atomic operation in Subversion. So TortoiseSVN first finds the HEAD revision of the repository, and then updates all items to that particular revision number to avoid creating a mixed revision working copy.

If only one item is selected for updating or the selected items are not all from the same repository, TortoiseSVN just updates to HEAD.

No command line options are used here. Update to revision also implements the update command, but offers more options.

E.2.3. Update to Revision

```
svn info URL_of_WC
svn update [-r rev] [-N] [--ignore-externals] PATH
```

If Only update the top folder is checked, use the `-N` switch.

If Omit externals is checked, use the `--ignore-externals` switch.

E.2.4. Commit

In TortoiseSVN, the commit dialog uses several Subversion commands. The first stage is a status check which determines the items in your working copy which can potentially be committed. You can review the list, diff files against BASE and select the items you want to be included in the commit.

```
svn status -v PATH
```

If Show unversioned files is checked, TortoiseSVN will also show all unversioned files and folders in the working copy hierarchy, taking account of the ignore rules. This particular feature has no direct equivalent in Subversion, as the `svn status` command does not descend into unversioned folders.

If you check any unversioned files and folders, those items will first be added to your working copy.

```
svn add PATH...
```

When you click on OK, the Subversion commit takes place. If you have left all the file selection checkboxes in their default state, TortoiseSVN uses a single recursive commit of the working copy. If you deselect some files, then a non-recursive commit (`-N`) must be used, and every path must be specified individually on the commit command line.

```
svn commit -m "LogMessage" [-N] [--no-unlock] PATH...
```

`LogMessage` here represents the contents of the log message edit box. This can be empty.

If Keep locks is checked, use the `--no-unlock` switch.

E.2.5. Diff

```
svn diff PATH
```

If you use Diff from the main context menu, you are diffing a modified file against its BASE revision. The output from the CLI command above also does this and produces output in unified-diff format. However, this is not what TortoiseSVN is using. TortoiseSVN uses TortoiseMerge (or a diff program of your choosing) to display differences visually between full-text files, so there is no direct CLI equivalent.

You can also diff any 2 files using TortoiseSVN, whether or not they are version controlled. TortoiseSVN just feeds the two files into the chosen diff program and lets it work out where the differences lie.

E.2.6. Show Log

```
svn log -v -r 0:N --limit 100 [--stop-on-copy] PATH
  or
svn log -v -r M:N [--stop-on-copy] PATH
```

By default, TortoiseSVN tries to fetch 100 log messages using the --limit method. If the settings instruct it to use old APIs, then the second form is used to fetch the log messages for 100 repository revisions.

If **Stop on copy/rename** is checked, use the `--stop-on-copy` switch.

E.2.7. Check for Modifications

```
svn status -v PATH
  or
svn status -u -v PATH
```

The initial status check looks only at your working copy. If you click on **Check repository** then the repository is also checked to see which files would be changed by an update, which requires the `-u` switch.

If **Show unversioned files** is checked, TortoiseSVN will also show all unversioned files and folders in the working copy hierarchy, taking account of the ignore rules. This particular feature has no direct equivalent in Subversion, as the `svn status` command does not descend into unversioned folders.

E.2.8. Revision Graph

The revision graph is a feature of TortoiseSVN only. There's no equivalent in the command line client.

What TortoiseSVN does is an

```
svn info URL_of_WC
svn log -v URL
```

where URL is the repository *root* and then analyzes the data returned.

E.2.9. Repo Browser

```
svn info URL_of_WC
svn list [-r rev] -v URL
```

You can use `svn info` to determine the repository root, which is the top level shown in the repository browser. You cannot navigate Up above this level. Also, this command returns all the locking information shown in the repository browser.

The `svn list` call will list the contents of a directory, given a URL and revision.

E.2.10. Edit Conflicts

This command has no CLI equivalent. It invokes TortoiseMerge or an external 3-way diff/merge tool to look at the files involved in the conflict and sort out which lines to use.

E.2.11. Resolved

```
svn resolved PATH
```

E.2.12. Rename

```
svn rename CURR_PATH NEW_PATH
```

E.2.13. Delete

```
svn delete PATH
```

E.2.14. Revert

```
svn status -v PATH
```

The first stage is a status check which determines the items in your working copy which can potentially be reverted. You can review the list, diff files against BASE and select the items you want to be included in the revert.

When you click on OK, the Subversion revert takes place. If you have left all the file selection checkboxes in their default state, TortoiseSVN uses a single recursive (-R) revert of the working copy. If you deselect some files, then every path must be specified individually on the revert command line.

```
svn revert [-R] PATH...
```

E.2.15. Cleanup

```
svn cleanup PATH
```

E.2.16. Get Lock

```
svn status -v PATH
```

The first stage is a status check which determines the files in your working copy which can potentially be locked. You can select the items you want to be locked.

```
svn lock -m "LockMessage" [--force] PATH...
```

`LockMessage` here represents the contents of the lock message edit box. This can be empty.

If Steal the locks is checked, use the `--force` switch.

E.2.17. Release Lock

```
svn unlock PATH
```

E.2.18. Branch/Tag

```
svn copy -m "LogMessage" URL URL
  or
svn copy -m "LogMessage" URL@rev URL@rev
  or
svn copy -m "LogMessage" PATH URL
```

The Branch/Tag dialog performs a copy to the repository. There are 3 radio button options:

- HEAD revision in the repository

- Specific revision in repository

- Working copy

which correspond to the 3 command line variants above.

`LogMessage` here represents the contents of the log message edit box. This can be empty.

E.2.19. Switch

```
svn info URL_of_WC
svn switch [-r rev] URL PATH
```

E.2.20. Merge

```
svn merge [--dry-run] --force From_URL@revN To_URL@revM PATH
```

The Dry run performs the same merge with the `--dry-run` switch.

```
svn diff From_URL@revN To_URL@revM
```

The Unified diff shows the diff operation which will be used to do the merge.

E.2.21. Export

```
svn export [-r rev] [--ignore-externals] URL Export_PATH
```

This form is used when accessed from an unversioned folder, and the folder is used as the destination.

Exporting a working copy to a different location is done without using the Subversion library, so there's no matching command line equivalent.

What TortoiseSVN does is to copy all files to the new location while showing you the progress of the operation. Unversioned files/folders can optionally be exported too.

In both cases, if Omit externals is checked, use the `--ignore-externals` switch.

E.2.22. Relocate

```
svn switch --relocate From_URL To_URL
```

E.2.23. Create Repository Here

```
svnadmin create --fs-type fsfs PATH
  or
svnadmin create --fs-type bdb PATH
```

E.2.24. Add

```
svn add PATH...
```

If you selected a folder, TortoiseSVN first scans it recursively for items which can be added.

E.2.25. Import

```
svn import -m LogMessage PATH URL
```

`LogMessage` here represents the contents of the log message edit box. This can be empty.

E.2.26. Blame

```
svn blame -r N:M -v PATH
svn log -r N:M PATH
```

If you use TortoiseBlame to view the blame info, the file log is also required to show log messages in a tooltip. If you view blame as a text file, this information is not required.

E.2.27. Add to Ignore List

```
svn propget svn:ignore PATH > tempfile
{edit new ignore item into tempfile}
svn propset svn:ignore -F tempfile PATH
```

Because the `svn:ignore` property is often a multi-line value, it is shown here as being changed via a text file rather than directly on the command line.

E.2.28. Create Patch

```
svn diff PATH > patch-file
```

TortoiseSVN creates a patch file in unified diff format by comparing the working copy with its BASE version.

E.2.29. Apply Patch

Applying patches is a tricky business unless the patch and working copy are at the same revision. Luckily for you, you can use TortoiseMerge, which has no direct equivalent in Subversion.

Appendix F. Implementation Details

This appendix contains a more detailed discussion of the implementation of some of TortoiseSVN's features.

F.1. Icon Overlays

Every file and folder has a Subversion status value as reported by the Subversion library. In the command line client, these are represented by single letter codes, but in TortoiseSVN they are shown graphically using the icon overlays. Because the number of overlays is very limited, each overlay may represent one of several status values.

The *Conflicted* overlay is used to represent the `conflicted` state, where an update or switch results in conflicts between local changes and changes downloaded from the repository. It is also used to indicate the `obstructed` state, which can occur when an operation is unable to complete.

The *Modified* overlay represents the `modified` state, where you have made local modifications, the `merged` state, where changes from the repository have been merged with local changes, and the `replaced` state, where a file has been deleted and replaced by another different file with the same name.

The *Deleted* overlay represents the `deleted` state, where an item is scheduled for deletion, or the `missing` state, where an item is not present. Naturally an item which is missing cannot have an overlay itself, but the parent folder can be marked if one of its child items is missing.

The *Added* overlay is simply used to represent the `added` status when an item has been added to version control.

The *In Subversion* overlay is used to represent an item which is in the `normal` state, or a versioned item whose state is not yet known. Because TortoiseSVN uses a background caching process to gather status, it may take a few seconds before the overlay updates.

The *Needs Lock* overlay is used to indicate when a file has the `svn:needs-lock` property set. For working copies which were created using Subversion 1.4.0 and later, the `svn:needs-lock` status is cached locally by Subversion and this is used to determine when to show this overlay. For working copies which are in pre-1.4.x format, TortoiseSVN shows this overlay when the file has read-only status. Note that Subversion automatically upgrades working copies when you update them, although the caching of the `svn:needs-lock` property may not happen until the file itself is updated.

The *Locked* overlay is used when the local working copy holds a lock for that file.

The *Ignored* overlay is used to represent an item which is in the `ignored` state, either due to a global ignore pattern, or the `svn:ignore` property of the parent folder. This overlay is optional.

The *Unversioned* overlay is used to represent an item which is in the `unversioned` state. This is an item in a versioned folder, but which is not under version control itself. This overlay is optional.

If an item has subversion status `none` (the item is not within a working copy) then no overlay is shown. If you have chosen to disable the *Ignored* and *Unversioned* overlays then no overlay will be shown for those files either.

An item can only have one Subversion status value. For example a file could be locally modified and it could be marked for deletion at the same time. Subversion returns a single status value - in this case `deleted`. Those priorities are defined within Subversion itself.

When TortoiseSVN displays the status recursively (the default setting), each folder displays an overlay reflecting its own status and the status of all its children. In order to display a single *summary* overlay, we use the priority order shown above to determine which overlay to use, with the *Conflicted* overlay taking highest priority.

In fact, you may find that not all of these icons are used on your system. This is because the number of overlays allowed by Windows is limited to 15. Windows uses 4 of those, and the remaining 11 can be used by other applications. If there are not enough overlay slots available, TortoiseSVN tries to be a "Good Citizen (TM)" and limits its use of overlays to give other apps a chance.

• *Normal*, *Modified* and *Conflicted* are always loaded and visible.

• *Deleted* is loaded if possible, but falls back to *Modified* if there are not enough slots.

• *Read-Only* is loaded if possible, but falls back to *Normal* if there are not enough slots.

• *Locked* is only loaded if there are fewer than 13 overlays already loaded. It falls back to *Normal* if there are not enough slots.

• *Added* is only loaded if there are fewer than 14 overlays already loaded. It falls back to *Modified* if there are not enough slots.

Appendix G. Securing Svnserve using SSH

This section provides a step-by-step guide to setting up Subversion and TortoiseSVN to use the `svn +ssh` protocol. If you already use authenticated SSH connections to login to your server, then you are already there and you can find more detail in the Subversion book. If you are not using SSH but would like to do so to protect your Subversion installation, this guide gives a simple method which does not involve creating a separate SSH user account on the server for every subversion user.

In this implementation we create a single SSH user account for all subversion users, and use different authentication keys to differentiate between the real Subversion users.

In this appendix we assume that you already have the subversion tools installed, and that you have created a repository as detailed elsewhere in this manual. Note that you should *not* start svnserve as a service or daemon when used with SSH.

Much of the information here comes from a tutorial provided by Marc Logemann, which can be found at *www.logemann.org* [http://www.logemann.org/2007/03/13/subversion-tortoisesvn-ssh-howto/] Additional information on setting up a Windows server was provided by Thorsten Müller. Thanks guys!

G.1. Setting Up a Linux Server

You need to have SSH enabled on the server, and here we assume that you will be using OpenSSH. On most distributions this will already be installed. To find out, type:

```
ps xa | grep sshd
```

and look for ssh jobs.

One point to note is that if you build Subversion from source and do not provide any argument to `./ configure`, Subversion creates a `bin` directory under `/usr/local` and places its binaries there. If you want to use tunneling mode with SSH, you have to be aware that the user logging in via SSH needs to execute the svnserve program and some other binaries. For this reason, either place `/usr/local/ bin` into the `PATH` variable or create symbolic links of your binaries to the `/usr/sbin` directory, or to any other directory which is commonly in the `PATH`.

To check that everything is OK, login in as the target user with SSH and type:

```
which svnserve
```

This command should tell you if svnserve is reachable.

Create a new user which we will use to access the svn repository:

```
useradd -m svnuser
```

Be sure to give this user full access rights to the repository.

G.2. Setting Up a Windows Server

Install Cygwin SSH daemon as described here: *http://pigtail.net/LRP/printsrv/cygwin-sshd.html* [http:// pigtail.net/LRP/printsrv/cygwin-sshd.html]

Create a new Windows user account `svnuser` which we will use to access the repository. Be sure to give this user full access rights to the repository.

If there is no password file yet then create one from the Cygwin console using:

```
mkpasswd -l > /etc/passwd
```

G.3. SSH Client Tools for use with TortoiseSVN

Grab the tools we need for using SSH on the windows client from this site: *http://www.chiark.greenend.org.uk/~sgtatham/putty/* [http://www.chiark.greenend.org.uk/~sgtatham/putty/] Just go to the download section and get `Putty`, `Plink`, `Pageant` and `Puttygen`.

G.4. Creating OpenSSH Certificates

The next step is to create a key pair for authentication. There are two possible ways to create keys. The first is to create the keys with PuTTYgen on the client, upload the public key to your server and use the private key with PuTTY. The other is to create the key pair with the OpenSSH tool ssh-keygen, download the private key to your client and convert the private key to a PuTTY-style private key.

G.4.1. Create Keys using ssh-keygen

Login to the server as `root` or `svnuser` and type:

```
ssh-keygen -b 1024 -t dsa -N passphrase -f keyfile
```

substituting a real pass-phrase (which only you know) and key file. We just created a SSH2 DSA key with 1024 bit key-phrase. If you type

```
ls -l keyfile*
```

you will see two files, `keyfile` and `keyfile.pub`. As you might guess, the `.pub` file is the public key file, the other is the private one.

Append the public key to those in the `.ssh` folder within the `svnuser` home directory:

```
cat keyfile.pub >> /home/svnuser/.ssh/authorized_keys
```

In order to use the private key we generated, we have to convert it to a putty format. This is because the private key file format is not specified by a standards body. After you download the private key file to your client PC, start PuTTYgen and use Conversions → Import key. Browse to your file `keyfile` which you got from the server the passphrase you used when creating the key. Finally click on Save private key and save the file as `keyfile.PPK`.

G.4.2. Create Keys using PuTTYgen

Use PuTTYgen to generate a public-key/private-key pair and save it. Copy the public key to the server and append it to those in the `.ssh` folder within the `svnuser` home directory:

```
cat keyfile.pub >> /home/svnuser/.ssh/authorized_keys
```

G.5. Test using PuTTY

To test the connection we will use PuTTY. Start the program and on the Session tab set the hostname to the name or IP address of your server, the protocol to SSH and save the session as `SvnConnection` or whatever name you prefer. On the SSH tab set the preferred SSH protocol version to 2 and from Auth set the full path to the `.PPK` private key file you converted earlier. Go back to the Sessions tab and hit the Save button. You will now see `SvnConnection` in the list of saved sessions.

Click on Open and you should see a telnet style login prompt. Use `svnuser` as the user name and if all is well you should connect directly without being prompted for a password.

You may need to edit `/etc/sshd_config` on the server. Edit lines as follows and restart the SSH service afterwards.

```
PubkeyAuthentication yes
PasswordAuthentication no
PermitEmptyPasswords no
ChallengeResponseAuthentication no
```

G.6. Testing SSH with TortoiseSVN

So far we have only tested that you can login using SSH. Now we need to make sure that the SSH connection can actually run svnserve. On the server modify `/home/svnuser/.ssh/authorized_keys` as follows to allow many subversion authors to use the same system account, `svnuser`. Note that every subversion author uses the same login but a different authentication key, thus you have to add one line for every author.

Note: This is all on one very long line.

```
command="svnserve -t -r <ReposRootPath> --tunnel-user=<author>",
            no-port-forwarding,no-agent-forwarding,no-X11-forwarding,
            no-pty ssh-rsa <PublicKey> <Comment>
```

There are several values that you need to set according to your setup.

`<ReposRootPath>` should be replaced with the path to the directory containing your repositories. This avoids the need to specify full server paths within URLs. Note that you must use forward slashes even on a Windows server, e.g. `c:/svn/reposroot`. In the examples below we assume that you have a repository folder within the repository root called `repos`.

`<author>` should be replaced with the svn author that you want to be stored on commit. This also allows svnserve to use its own access rights within `svnserve.conf`.

`<PublicKey>` should be replaced with the public key that you generated earlier.

`<Comment>` can be any comment you like, but it is useful for mapping an svn author name to the person's real name.

Right click on any folder in Windows Explorer and select TortoiseSVN → Repo-Browser. You will be prompted to enter a URL, so enter one in this form:

```
svn+ssh://svnuser@SvnConnection/repos
```

What does this URL mean? The Schema name is `svn+ssh` which tells TortoiseSVN how to handle the requests to the server. After the double slash, you specify the user to connect to the server, in our case `svnuser`. After the @ we supply our PuTTY session name. This session name contains all details like where to find the private key and the server's IP or DNS. Lastly we have to provide the path to the repository, relative to the repository root on the server, as specified in the `authorized_keys` file.

Click on OK and you should be able to browse the repository content. If so you now have a running SSH tunnel in conjunction with TortoiseSVN.

Note that by default TortoiseSVN uses its own version of Plink to connect. This avoids a console window popping up for every authentication attempt, but it also means that there is nowhere for error messages to appear. If you receive the error "Unable to write to standard output", you can try specifying Plink as

the client in TortoiseSVN's network settings. This will allow you to see the real error message generated by Plink.

G.7. SSH Configuration Variants

One way to simplify the URL in TortoiseSVN is to set the user inside the PuTTY session. For this you have to load your already defined session `SvnConnection` in PuTTY and in the Connection tab set Auto login user to the user name, e.g. `svnuser`. Save your PuTTY session as before and try the following URL inside TortoiseSVN:

```
svn+ssh://SvnConnection/repos
```

This time we only provide the PuTTY session `SvnConnection` to the SSH client TortoiseSVN uses (TortoisePlink.exe). This client will check the session for all necessary details.

At the time of writing PuTTY does not check all saved configurations, so if you have multiple configurations with the same server name, it will pick the first one which matches. Also, if you edit the default configuration and save it, the auto login user name is *not* saved.

Many people like to use Pageant for storing all their keys. Because a PuTTY session is capable of storing a key, you don't always need Pageant. But imagine you want to store keys for several different servers; in that case you would have to edit the PuTTY session over and over again, depending on the server you are trying to connect with. In this situation Pageant makes perfect sense, because when PuTTY, Plink, TortoisePlink or any other PuTTY-based tool is trying to connect to an SSH server, it checks all private keys that Pageant holds to initiate the connection.

For this task, simply run Pageant and add the private key. It should be the same private key you defined in the PuTTY session above. If you use Pageant for private key storage, you can delete the reference to the private key file in your saved PuTTY session. You can add more keys for other servers, or other users of course.

If you don't want to repeat this procedure after every reboot of your client, you should place Pageant in the auto-start group of your Windows installation. You can append the keys with complete paths as command line arguments to Pageant.exe

The last way to connect to an SSH server is simply by using this URL inside TortoiseSVN:

```
svn+ssh://svnuser@100.101.102.103/repos
svn+ssh://svnuser@mydomain.com/repos
```

As you can see, we don't use a saved PuTTY session but an IP address (or domain name) as the connection target. We also supply the user, but you might ask how the private key file will be found. Because TortoisePlink.exe is just a modified version of the standard Plink tool from the PuTTY suite, TortoiseSVN will also try all the keys stored in Pageant.

Glossary

Add	A Subversion command that is used to add a file or directory to your working copy. The new items are added to the repository when you commit.
BASE revision	The current base revision of a file or folder in your *working copy*. This is the revision the file or folder was in, when the last checkout, update or commit was run. The BASE revision is normally not equal to the HEAD revision.
BDB	Berkeley DB. A well tested database backend for repositories, that cannot be used on network shares. Default for pre 1.2 repositories.
Blame	This command is for text files only, and it annotates every line to show the repository revision in which it was last changed, and the author who made that change. Our GUI implementation is called TortoiseBlame and it also shows the commit date/time and the log message when you hover the mouse of the revision number.
Branch	A term frequently used in revision control systems to describe what happens when development forks at a particular point and follows 2 separate paths. You can create a branch off the main development line so as to develop a new feature without rendering the main line unstable. Or you can branch a stable release to which you make only bug fixes, while new developments take place on the unstable trunk. In Subversion a branch is implemented as a "cheap copy".
Checkout	A Subversion command which creates a local working copy in an empty directory by downloading versioned files from the repository.
Cleanup	To quote from the Subversion book: " Recursively clean up the working copy, removing locks and resuming unfinished operations. If you ever get a *working copy locked* error, run this command to remove stale locks and get your working copy into a usable state again. " Note that in this context *lock* refers to local filesystem locking, not repository locking.
Commit	This Subversion command is used to pass the changes in your local working copy back into the repository, creating a new repository revision.
Conflict	When changes from the repository are merged with local changes, sometimes those changes occur on the same lines. In this case Subversion cannot automatically decide which version to use and the file is said to be in conflict. You have to edit the file manually and resolve the conflict before you can commit any further changes.
Copy	In a Subversion repository you can create a copy of a single file or an entire tree. These are implemented as "cheap copies" which act a bit like a link to the original in that they take up almost no space. Making a copy preserves the history of the item in the copy, so you can trace changes made before the copy was made.
Delete	When you delete a versioned item (and commit the change) the item no longer exists in the repository after the committed revision. But of course it still exists in earlier repository revisions, so you can still access it. If necessary, you can copy a deleted item and "resurrect" it complete with history.

Diff	Shorthand for "Show Differences". Very useful when you want to see exactly what changes have been made.
Export	This command produces a copy of a versioned folder, just like a working copy, but without the local `.svn` folders.
FSFS	A proprietary Subversion filesystem backend for repositories. Can be used on network shares. Default for 1.2 and newer repositories.
GPO	Group policy object
HEAD revision	The latest revision of a file or folder in the *repository*.
History	Show the revision history of a file or folder. Also known as "Log".
Import	Subversion command to import an entire folder hierarchy into the repository in a single revision.
Lock	When you take out a lock on a versioned item, you mark it in the repository as non-committable, except from the working copy where the lock was taken out.
Log	Show the revision history of a file or folder. Also known as "History".
Merge	The process by which changes from the repository are added to your working copy without disrupting any changes you have already made locally. Sometimes these changes cannot be reconciled automatically and the working copy is said to be in conflict.
	Merging happens automatically when you update your working copy. You can also merge specific changes from another branch using TortoiseSVN's Merge command.
Patch	If a working copy has changes to text files only, it is possible to use Subversion's Diff command to generate a single file summary of those changes in Unified Diff format. A file of this type is often referred to as a "Patch", and it can be emailed to someone else (or to a mailing list) and applied to another working copy. Someone without commit access can make changes and submit a patch file for an authorized committer to apply. Or if you are unsure about a change you can submit a patch for others to review.
Property	In addition to versioning your directories and files, Subversion allows you to add versioned metadata - referred to as "properties" to each of your versioned directories and files. Each property has a name and a value, rather like a registry key. Subversion has some special properties which it uses internally, such as `svn:eol-style`. TortoiseSVN has some too, such as `tsvn:logminsize`. You can add your own properties with any name and value you choose.
Relocate	If your repository moves, perhaps because you have moved it to a different directory on your server, or the server domain name has changed, you need to "relocate" your working copy so that its repository URLs point to the new location.
	Note: you should only use this command if your working copy is referring to the same location in the same repository, but the repository itself has moved. In any other circumstance you probably need the "Switch" command instead.

Repository	A repository is a central place where data is stored and maintained. A repository can be a place where multiple databases or files are located for distribution over a network, or a repository can be a location that is directly accessible to the user without having to travel across a network.
Resolve	When files in a working copy are left in a conflicted state following a merge, those conflicts must be sorted out by a human using an editor (or perhaps TortoiseMerge). This process is referred to as "Resolving Conflicts". When this is complete you can mark the conflicted files as being resolved, which allows them to be committed.
Revert	Subversion keeps a local "pristine" copy of each file as it was when you last updated your working copy. If you have made changes and decide you want to undo them, you can use the "revert" command to go back to the pristine copy.
Revision	Every time you commit a set of changes, you create one new "revision" in the repository. Each revision represents the state of the repository tree at a certain point in its history. If you want to go back in time you can examine the repository as it was at revision N.
	In another sense, a revision can refer to the set of changes that were made when that revision was created.
Revision Property (revprop)	Just as files can have properties, so can each revision in the repository. Some special revprops are added automatically when the revision is created, namely: svn:date svn:author svn:log which represent the commit date/time, the committer and the log message respectively. These properties can be edited, but they are not versioned, so any change is permanent and cannot be undone.
SVN	A frequently-used abbreviation for Subversion.
	The name of the Subversion custom protocol used by the "svnserve" repository server.
Switch	Just as "Update-to-revision" changes the time window of a working copy to look at a different point in history, so "Switch" changes the space window of a working copy so that it points to a different part of the repository. It is particularly useful when working on trunk and branches where only a few files differ. You can switch your working copy between the two and only the changed files will be transferred.
Update	This Subversion command pulls down the latest changes from the repository into your working copy, merging any changes made by others with local changes in the working copy.
Working Copy	This is your local "sandbox", the area where you work on the versioned files, and it normally resides on your local hard disk. You create a working copy by doing a "Checkout" from a repository, and you feed your changes back into the repository using "Commit".

Index

Symbols

.svn folder, 141
_svn folder, 141

A

Access, 33
add, 75
add files to repository, 41
annotate, 104
Apache, 16
ASP projects, 153
authentication, 40
Authorization, 20
auto-props, 87
automation, 155, 158

B

backup, 35
blame, 104
branch, 76, 89
bug tracker, 114
bug tracking, 114
bugtracker, 114

C

case change, 81
changelist, 59
changes, 149
check in, 48
check new version, 152
checkout, 45
checkout link, 36
cleanup, 83
CLI,
client hooks, 136
COM,
COM SubWCRev interface, 144
command line, 155, 158
command line client,
commit, 48
commit message, 147
commit messages, 61
common projects, 149
compare, 71
compare files, 149
compare revisions, 72
conflict, 9, 54
context menu, 37
context menu entries, 153
copy, 89, 107
copy files, 76
Create
    Command Line Client, 32

TortoiseSVN, 32
create repository, 32
create working copy, 45

D

delete, 79
deploy, 152
detach from repository, 150
dictionary, 3
diff, 71, 103
diff tools, 74
diffing, 59
disable functions, 153
domain controller, 152
domaincontroller, 21
drag handler, 39
drag-n-drop, 39

E

edit log/author, 67
empty message, 147
exclude pattern, 120
expand keywords, 85
explorer, 1
Explorer Columns, 57
export, 112
export changes, 72
external repositories, 43
externals, 43, 149

F

FAQ,
fetch changes, 52
filter, 68

G

global ignore, 120
globbing, 78
GPO, 152
graph, 109
group policies, 152, 153

H

history, 61
hook scripts, 36, 136
hooks, 36

I

icons, 55
ignore, 77
image diff, 73
import, 41
import in place, 43
Index of projects, 20
install, 3
issue tracker, 114