TortoiseSVN

A Subversion client for Windows

Version 1.4.1

Stefan Küng
Lübbe Onken
Simon Large

TortoiseSVN: A Subversion client for Windows: Version 1.4.1

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

In *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 *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 *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.

SubWCRev 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 *How Do I...* section answers some common questions about performing tasks which are not explicitly covered elsewhere.

The section on *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 *Command Line 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, bughunting and taking care of the documentation

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

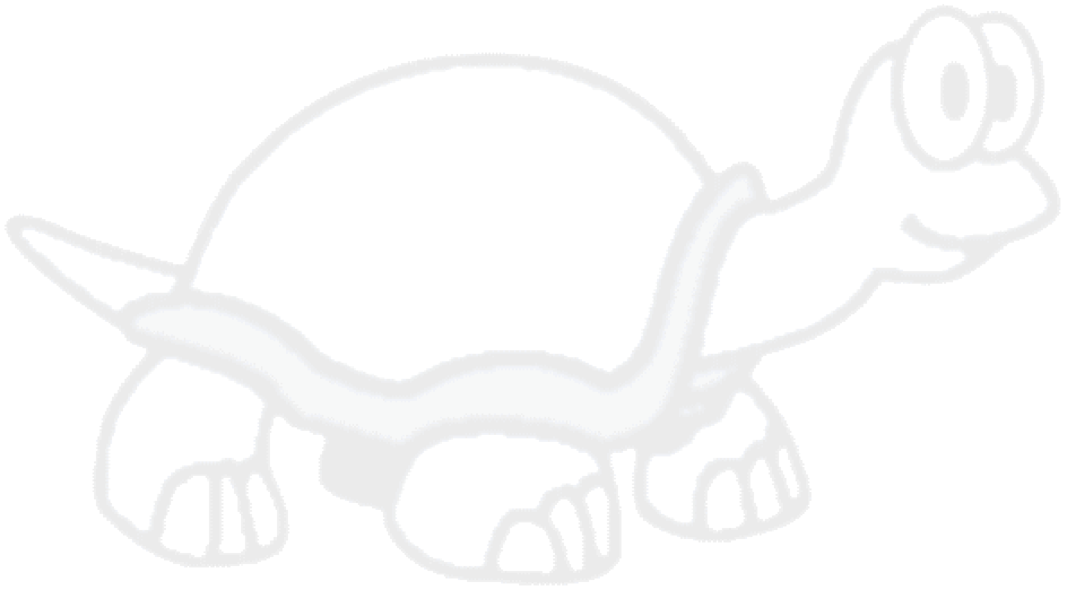
Tips that make your life easier.

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 sourcecode of TortoiseCVS and used it as a base for TortoiseSVN. He then started the project, registered the domain tortoisesvn.org and put the sourcecode 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 userbase 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 Win2k SP2, WinXP 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. Doubleclick on the installer file and follow the instructions. The installer will take care of the rest. If you want to install TortoiseSVN for `all users` then you must have administrator rights on your system. If you don't have those rights, TortoiseSVN will automatically install for the current user only.

Important

If you don't have the latest C-runtime and MFC libraries installed, you still must have Administrator privileges to install TortoiseSVN. But once those libraries are installed, you can update or install TortoiseSVN without those privileges.

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://lingucomponent.openoffice.org/spell_dic.html* [http://lingucomponent.openoffice.org/spell_dic.html]

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 5.15, “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.

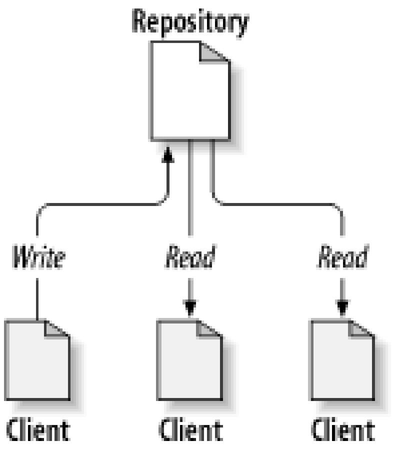


Figure 2.1. A Typical Client/Server System

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!

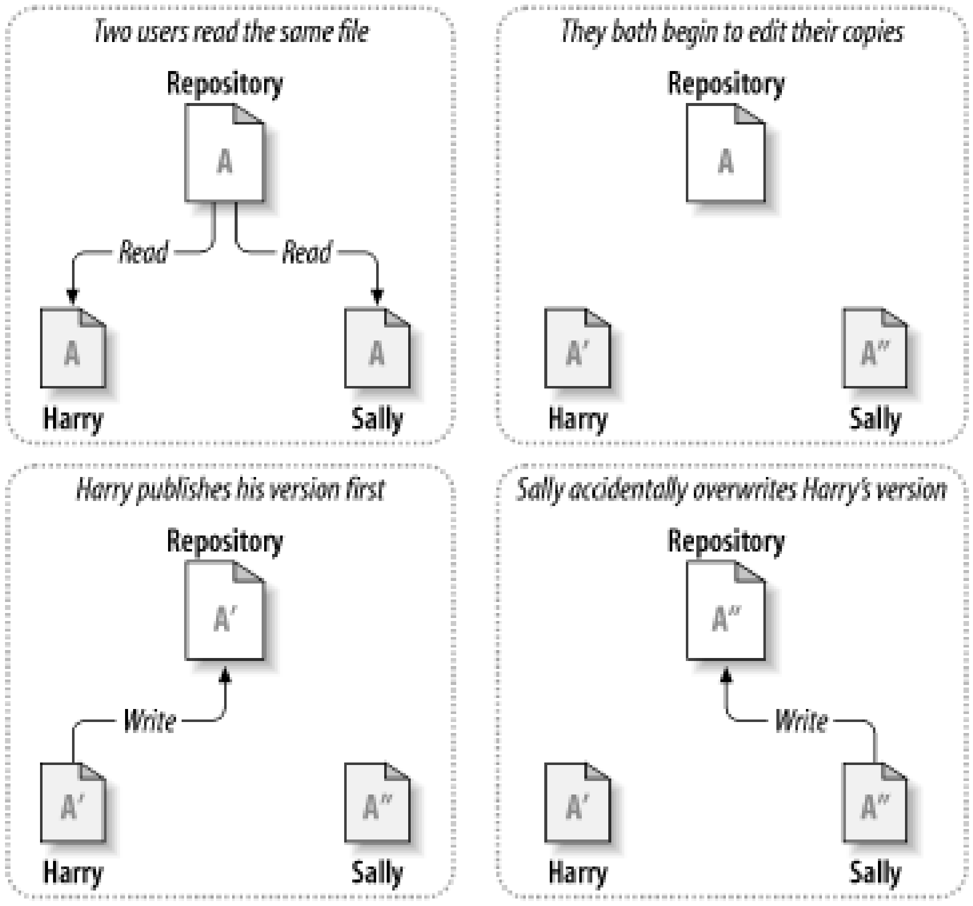

Two users read the same file
Repository
A
Read
Read
A
A
Harry
Sally
They both begin to edit their copies
Repository
A
A′
A″
Harry
Sally
Harry publishes his version first
Repository
A′
Write
A′
A″
Harry
Sally
Sally accidentally overwrites Harry's version
Repository
A″
Write
A′
A″
Harry
Sally

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.

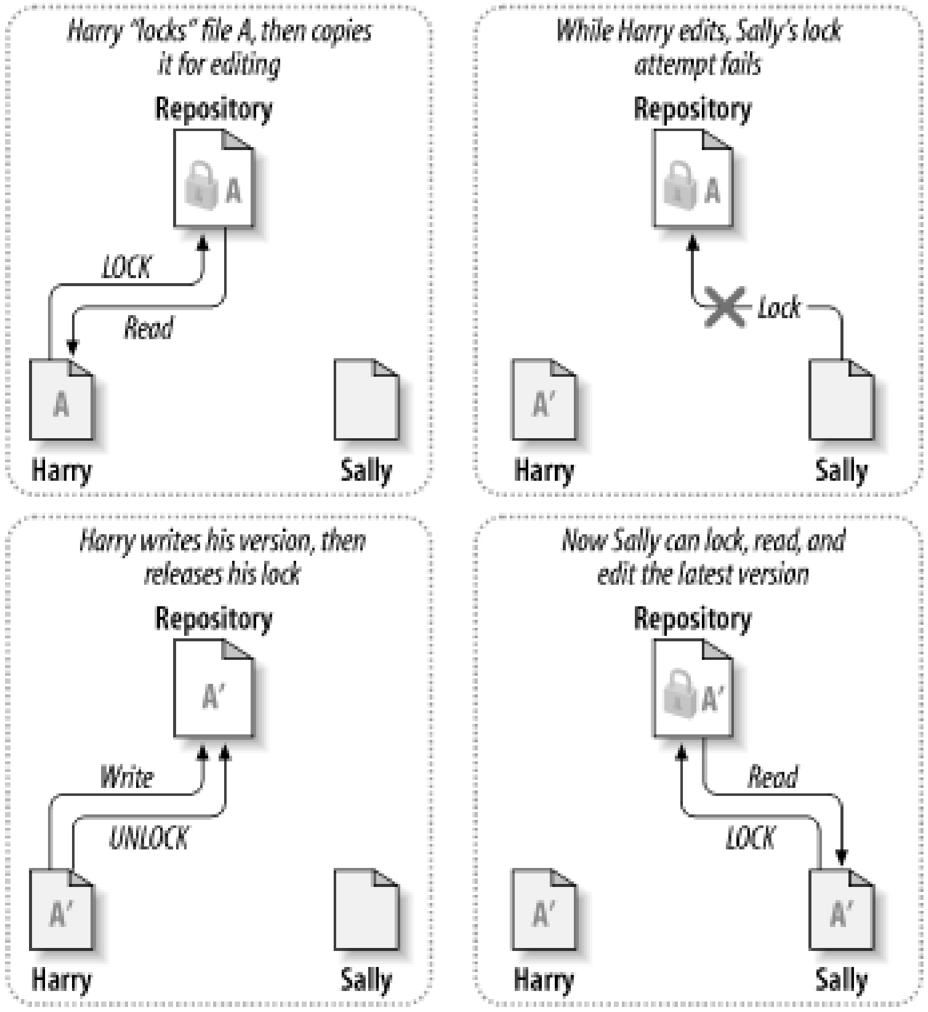

Harry "locks" file A, then copies it for editing
Repository
A
LOCK
Read
A
Harry
Sally
While Harry edits, Sally's lock attempt fails
Repository
A
Lock
A′
Harry
Sally
Harry writes his version, then releases his lock
Repository
A′
Write
UNLOCK
A′
Harry
Sally
Now Sally can lock, read, and edit the latest version
Repository
A′
Read
LOCK
A′
A′
Harry
Sally

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.

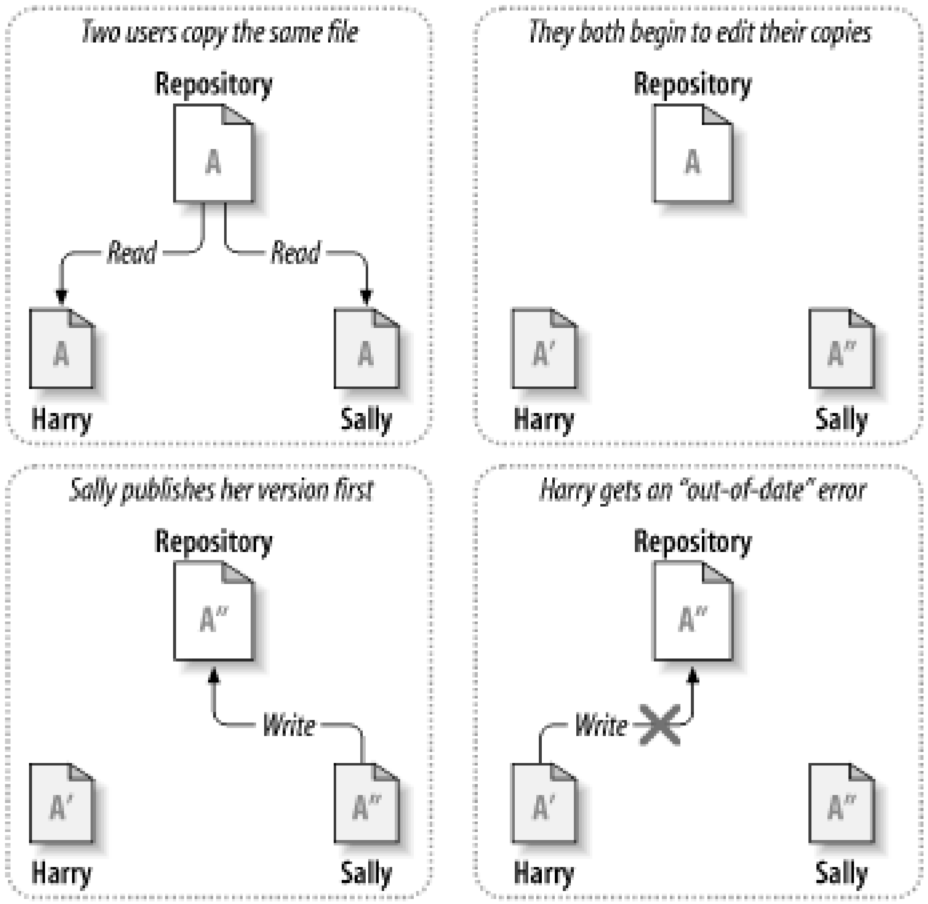

Two users copy the same file
Repository
A
Read
Read
A
A
Harry
Sally
They both begin to edit their copies
Repository
A
A′
A″
Harry
Sally
Sally publishes her version first
Repository
A″
Write
A′
A″
Harry
Sally
Harry gets an "out-of-date" error
Repository
A″
Write
A′
A″
Harry
Sally

Figure 2.4. The Copy-Modify-Merge Solution

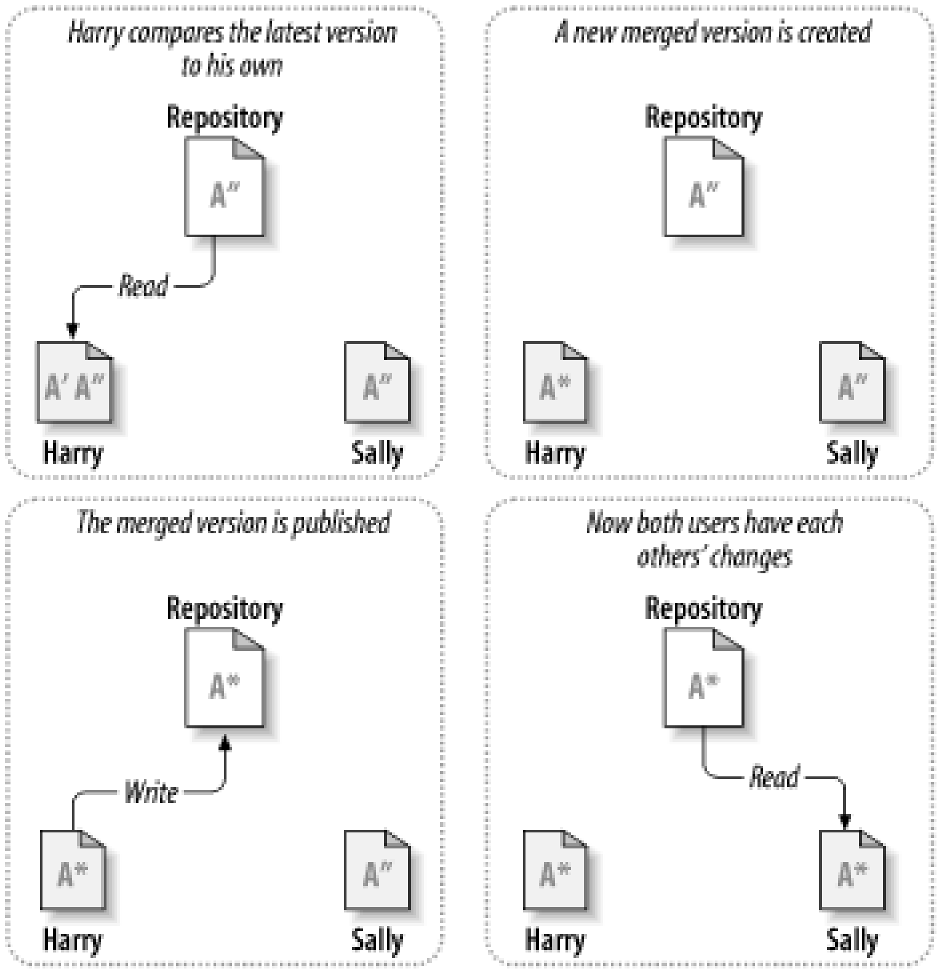

Harry compares the latest version to his own
Repository
A″
Read
A′ A″
A″
Harry
Sally
A new merged version is created
Repository
A″
A*
A″
Harry
Sally
The merged version is published
Repository
A*
Write
A*
A″
Harry
Sally
Now both users have each others' changes
Repository
A*
Read
A*
A*
Harry
Sally

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 un-mergeable 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.

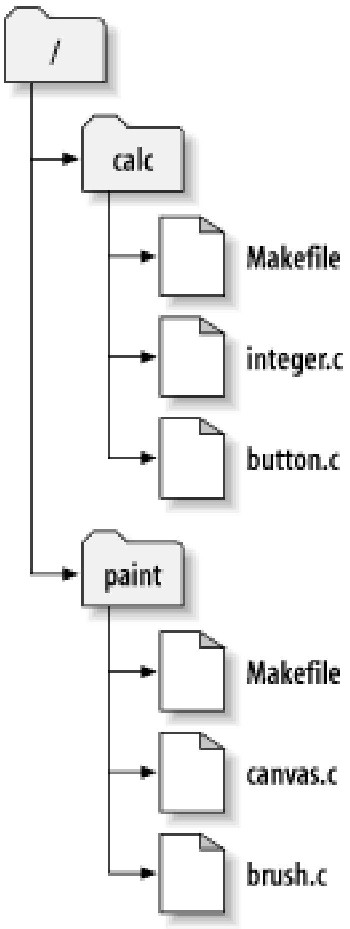


Figure 2.6. The Repository's Filesystem

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).

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.
`ht-tps://`	Same as `http://`, but with SSL encryption.
`svn://`	Unauthenticated TCP/IP access via custom protocol to an `svnserve` server.
`svn+ssh://`	authenticated, encrypted TCP/IP access via custom protocol to an `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 an 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.

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. Revisions

An **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 commit's changes take place, or none of them take place. Subversion tries to retain this atomicity in the face of program crashes, system crashes, network problems, and other users' actions.

Each time the repository accepts a commit, this creates a new state of the 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.

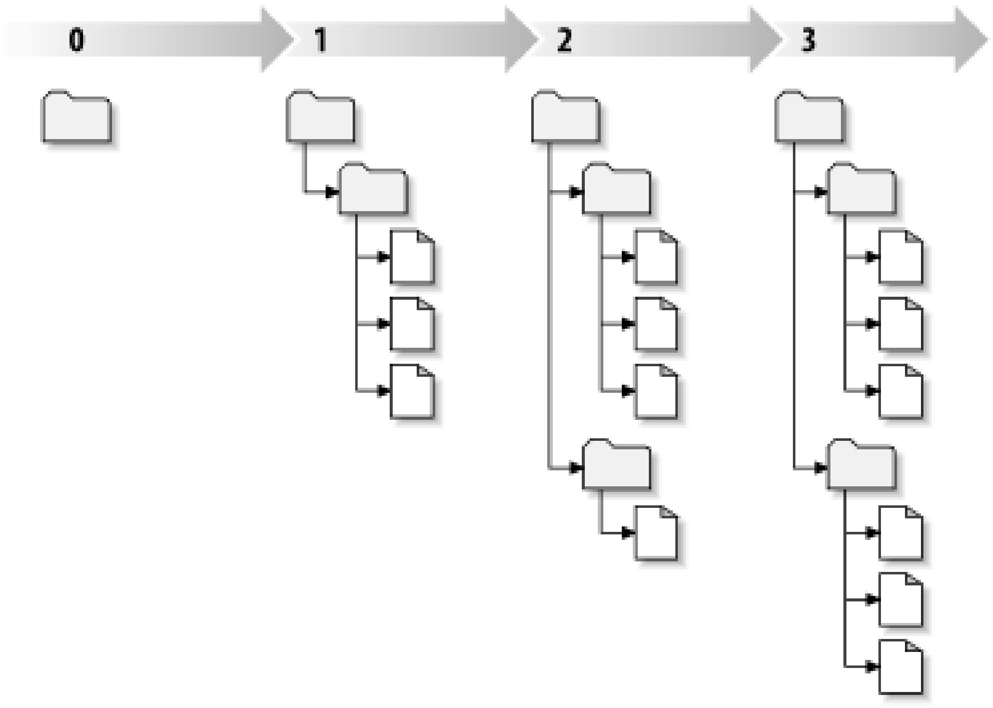


Figure 2.7. The Repository

> **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.

2.3.3. 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 an **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. An **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. An **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.

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

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 "Webfolder" in the Windows explorer and then access it like any other folder in the filesystem

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 either Windows2000 / WinXP+SP1 or Windows2003.

Warning

Please note that Windows XP without the servicepack 1 will lead to bogus network data and could therefore corrupt your repository!

1. Download the latest version of the Apache webserver from *http://httpd.apache.org/download.cgi* [http://httpd.apache.org/download.cgi]. Make sure that you download the version > 2.0.54 - the version 1.3.xx won't work! Also, versions lower than 2.0.54 won't work with Subversion 1.2 because of a bug in how Apache < 2.0.54 was built for Windows.

2. Once you have the Apache2 installer you can doubleclick 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 pro-

grams 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-webserver is running correctly by pointing your webbrowser 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 Subversion from *http://subversion.tigris.org/servlets/ProjectDocumentList?folderID=91* [http://subversion.tigris.org/servlets/ProjectDocumentList?folderID=91].

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/libdb43.dll` from the Subversion installation directory to the Apache modules 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 shall be used for Subversion repositories you have to edit the Apache config 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
   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`
AuthType Basic	is to activate basic authentication, i.e. Username/password

Setting	Explanation
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 webserver.

- 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”

3.1.5. Multiple Repositories

If you used the SVNParentPath directive then you don't have to change the Apache config file everytime 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.

If your Subversion server is earlier than 1.3 you will just get a nasty error page showing. To get a nice looking listing of all available projects instead, you can use the following PHP script which generates the index for you automatically. (You will need to install PHP on your server in order to use the script shown below).

```
<html>
<head>
<title>Subversion Repositories</title>
</head>
<body>

<h2>Subversion Repositories</h2>
<p>
<?php
    $svnparentpath = "C:/svn";
    $svnparenturl = "/svn";

    $dh = opendir( $svnparentpath );
    if( $dh ) {
        while( $dir = readdir( $dh ) ) {
            $svndir = $svnparentpath . "/" . $dir;
            $svndbdir = $svndir . "/db";
            $svnfstypefile = $svndbdir . "/fs-type";
            if( is_dir( $svndir ) && is_dir( $svndbdir ) ) {
                echo "<a href=\"" . $svnparenturl . "/" .
                        $dir . "\">" . $dir . "</a>\n";
                if( file_exists( $svnfstypefile ) ) {
                    $handle = fopen ("$svnfstypefile", "r");
                    $buffer = fgets($handle, 4096);
                    fclose( $handle );
                    $buffer = chop( $buffer );
                    if( strcmp( $buffer, "fsfs" )==0 ) {
                        echo " (FSFS) <br />\n";
                    } else {
                        echo " (BDB) <br />\n";
                    }
                } else {
                    echo " (BDB) <br />\n";
                }
            }
        }
        closedir( $dh );
    }
?>
</p>

</body>
</html>
```

Save the lines above to a file `svn_index.php` and store that file in your web root folder. Next you have to tell Apache to show that page instead of the error:

- Uncomment (remove the '#' char) from the following line in your Apache config file:

```
#LoadModule rewrite_module modules/mod_rewrite.so
```

- Add the following lines just below your <Location> block where you define your Subversion stuff:

```
RewriteEngine on
RewriteRule ^/svn$ /svn_index.php [PT]
RewriteRule ^/svn/$ /svn_index.php [PT]
RewriteRule ^/svn/index.html$ /svn_index.php [PT]
```

3.1.6. Path-Based Authorization

The mod_authz_svn module permits fine-grained control of access permissions based on usernames 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 authorizarion 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 sourcecode. 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, 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's 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.

Tip

Subversion AuthzSVNAccessFile files are case sensitive in regard to user names ("JUser" is different from "juser").

In Microsoft's world, Windows domains and usernames 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.

SSL and InternetExplorer

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: *http://www.microsoft.com/resources/documentation/WindowsServ/2003/standard/proddocs/en-us/qos_enablekeepalives.asp* [http://www.microsoft.com/resources/documentation/WindowsServ/2003/standard/proddocs/en-us/qos_enablekeepalives.asp] 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.

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:

  ```
  AuthAthoritative Off
  SSPIAuthoritative Off
  ```

Here is an example of the full Apache configuration for combined Windows domain & 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 On
SSPIDomain <domaincontroller>
SSPIOfferBasic On

# Htpasswd Logins.
AuthType Basic
AuthAuthoritative Off
AuthUserFile passwd

Require valid-user
</Location>
```

3.1.9. Securing the server with SSL

The apache server doesn't have SSL support installed by default due to US-export restrictions. But you can easily download the required module from somewhere else and install it yourself.

1. First you need the required files to enable SSL. You can find those in the package available at *http://hunter.campbus.com/* [http://hunter.campbus.com/]. Just unzip the package and then copy `mod_ssl.so` to the `modules` folder of Apache and the file `openssl.exe` to the `bin` folder. Also copy the file `conf/ssl.conf` to the `conf` folder of Apache.

2. Open the file `ssl.conf` in the Apache conf folder with a text editor.

3. Place a comment char (#) in front of the following lines:

   ```
   DocumentRoot "c:/apache/htdocs"
   ServerName www.example.com:443
   ServerAdmin you@example.com
   ErrorLog logs/error_log
   TransferLog logs/access_log
   ```

4. change the line

   ```
   SSLCertificateFile conf/ssl.crt/server.crt
   ```

   to

   ```
   SSLCertificateFile conf/ssl/my-server.cert
   ```

   the line

   ```
   SSLCertificateKeyFile conf/ssl.key/server.key
   ```

to

```
SSLCertificateKeyFile conf/ssl/my-server.key
```

and the line

```
SSLMutex  file:logs/ssl_mutex
```

to

```
SSLMutex  default
```

5. Delete the lines

   ```
   <IfDefine SSL>
   ```

   and

   ```
   </IfDefine>
   ```

6. Open the Apache config file (`httpd.conf`) and uncomment the line

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

7. Openssl needs a config file. You can download a working one from *http://tud.at/programm/openssl.cnf* [http://tud.at/programm/openssl.cnf]. Save the file to `bin/openssl.cnf`. Please note: the file has the type `*.cnf`. Windows treats such files in a special way but it really is just a text file!

8. 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 privkey.pem 2048
   ```

   Next type the commands

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

   and (on one line)

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

This will create a certificate which will expire in 4000 days. And finally enter:

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

These commands created some files in the Apache folder (`my-server.der.crt`, `my-server.csr`, `my-server.key`, `.rnd`, `privkey.pem`, `my-server.cert`). Copy the files to the folder `conf/ssl` (e.g. `C:\program files\apache group\apache2\conf\ssl`) - if this folder does not exist you have to create it first.

9. Restart the apache service.

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

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>
```

3.2. 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.

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].

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 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`, `libdb44.dll`, `libeay32.dll` and `ssleay32.dll` - copy these files 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 Url's 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
```

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.

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

If you are concerned about always having a user logged in on your server, or worried about someone shutting down svnserve or forgetting to restart it after a reboot, it is possible to run svnserve as a windows service. Starting with Subversion 1.4, svnserve can be installed as a native windows service, in previous versions it can be installed using a wrapper.

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
```

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.

To install svnserve using a wrapper, one written specifically for svnserve is `SvnService`. Magnus Norddahl adapted some skeleton code from Microsoft, and further improvements have been made by Daniel Thompson. Daniel's version is available for download from *tigris.org* [http://tortoisesvn.tigris.org/files/documents/406/29202/SVNServiceDT.zip].

More generic tools like *firedaemon* [http://www.firedaemon.com/] will also work. Note that you will still need to run svnserve with the --daemon switch.

Finally, if you have access to the Windows 2000/XP/2003 resource kit you can use *SrvAny* [http://support.microsoft.com/kb/q137890/] from Microsoft. This is the official Microsoft way of running programs as services, but it is a bit messy (requires registry editing) and if you stop the service it will kill svnserve immediately without letting it clean up. If you don't want to install the entire reskit, you can download *just* the SrvAny components from *Daniel Petri* [http://www.petri.co.il/software/srvany.zip].

3.2.4. 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 filesytem (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/1.2/svn.serverconfig.svnserve.html] and *Client Credentials Caching* [http://svnbook.red-bean.com/en/1.2/svn.serverconfig.netmodel.html]

3.2.5. 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.

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.

It is beyond the scope of this documentation to detail the installation and setup of a secure shell, however you can find further information in the *TortoiseSVN FAQ* [http://tortoisesvn.net/faq]. Search for “SSH”.

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

3.2.6. 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 old but stable Berkeley Database (BDB) format. The FSFS format is faster and it now works on network shares and Windows 98 without problems. The BDB format is more stable because it has been tested longer. Read *Repository Data Stores* [http://svnbook.red-bean.com/en/1.2/svn.reposadmin.html] 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

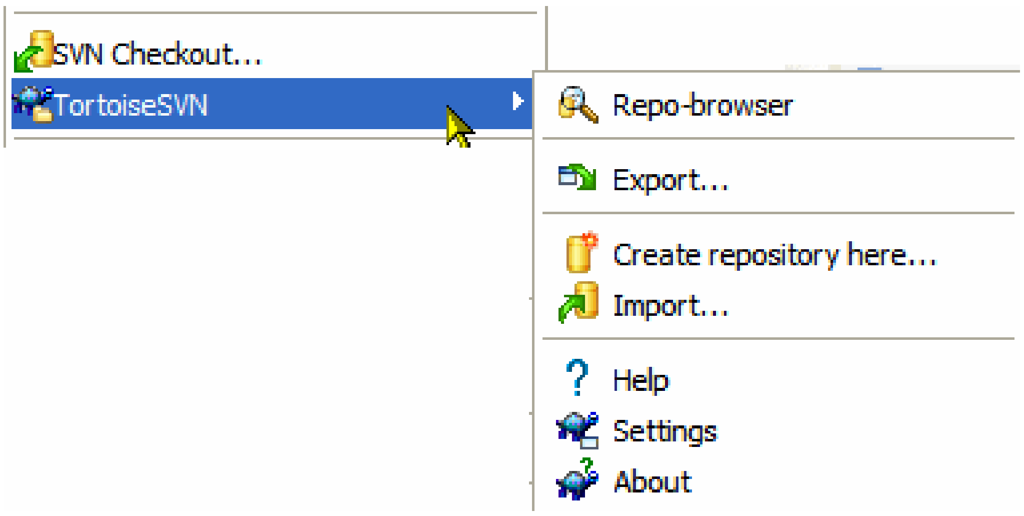


Figure 4.1. The TortoiseSVN menu for unversioned folders

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.

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 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.

Tip

If you really need to access a repository through a network share, create the repository with fsfs format. If you need to provide server access as well, you will need Subversion Server 1.1 or higher.

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/1.2/svn.reposadmin.create.html] in the Subversion Book for full details about the hooks which are implemented.

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 TSVN users.

When you install TortoiseSVN, it registers a new `tsvn:` protocol. When a TSVN 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.

- 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

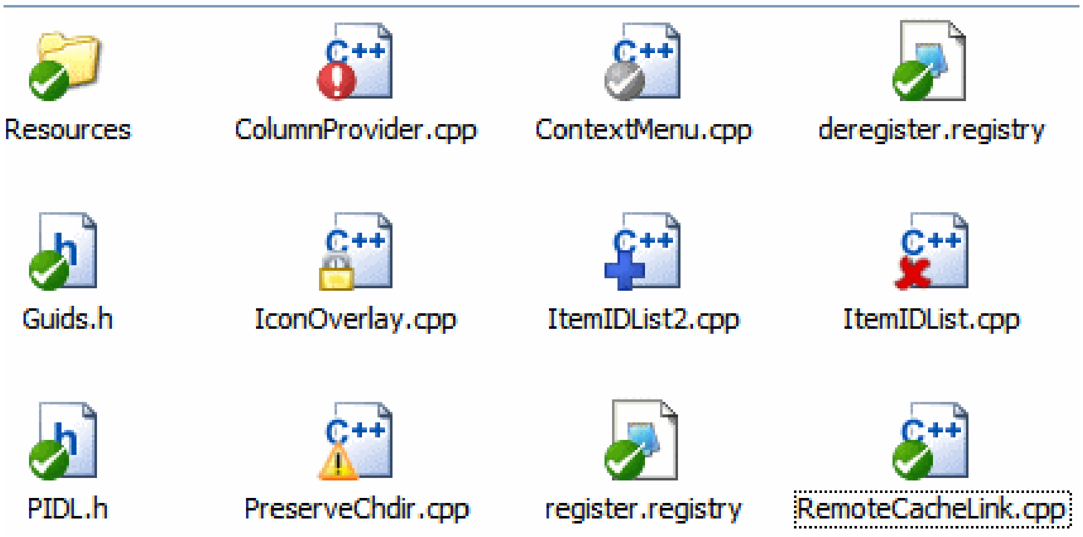


Figure 5.1. Explorer showing icon overlays

One of the most visible features of TortoiseSVN are 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

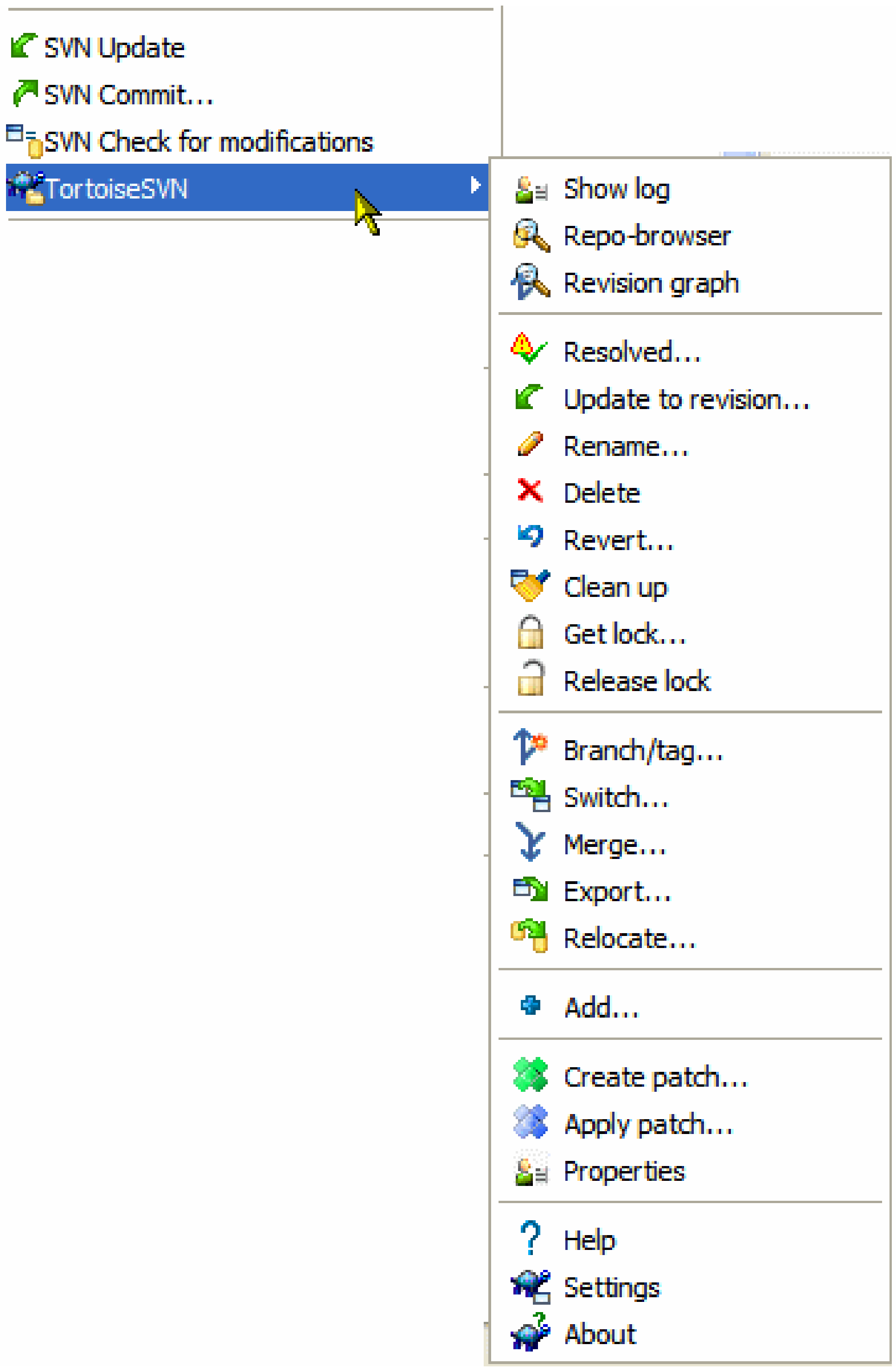


Figure 5.2. Context menu for a directory under version control

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.

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

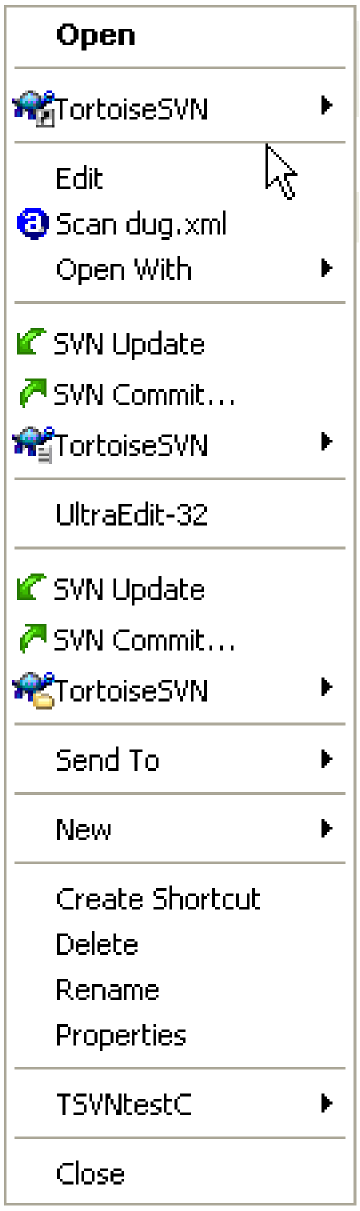


Figure 5.3. Explorer file menu for a shortcut in a versioned folder

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.

5.1.3. Drag and Drop

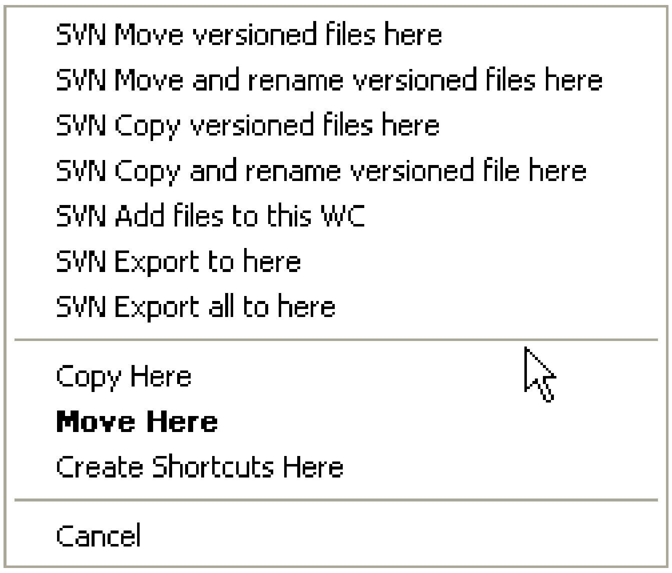

SVN Move versioned files here
SVN Move and rename versioned files here
SVN Copy versioned files here
SVN Copy and rename versioned file here
SVN Add files to this WC
SVN Export to here
SVN Export all to here
Copy Here
Move Here
Create Shortcuts Here
Cancel

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-know 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.

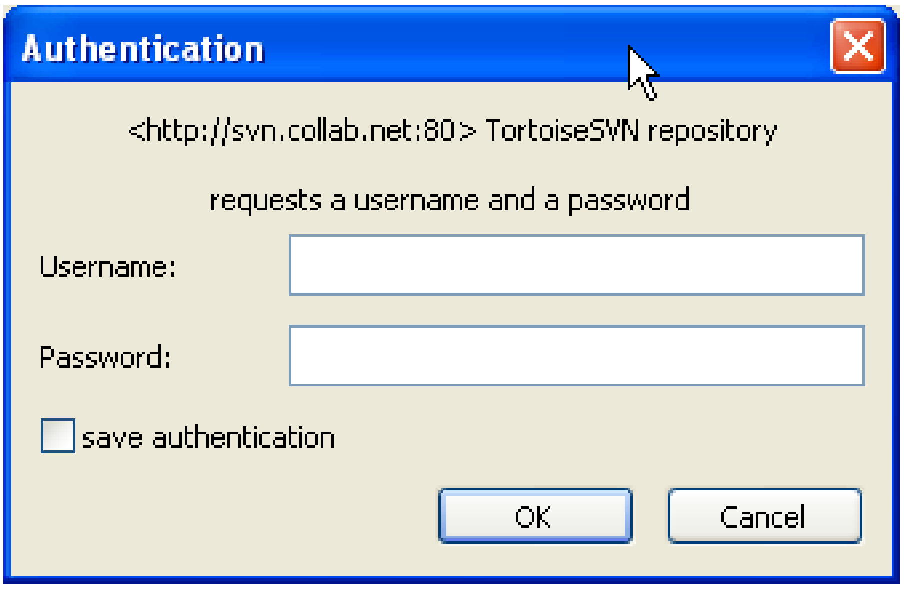
F5
: Refresh the current view. This is perhaps the single most useful one-key command. For example ... In Explorer this will refresh the icon overlays on your working copy. In the commit dialog it will re-scan the working copy to see what may need to be committed. In the Revision Log dialog it will contact 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.


Authentication

<http://svn.collab.net:80> TortoiseSVN repository

requests a username and a password

Username:

Password:

save authentication

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).

There is one file for each server that you access, formatted as plain text, so you can use a text editor to check which server it applies to. If you want to make Subversion and TortoiseSVN (and any other Subversion client) forget your credentials for a particular server, you have to delete the corresponding file.

If you want to clear the authentication cache for *all* servers, you can do so from the General page of TortoiseSVN's settings dialog. That button will clear all cached auth data from the Subversion auth directories, as well as any auth data stored in the registry by earlier versions of TortoiseSVN. Refer to Section 5.27.1, “General Settings”.

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

Tip

If you have to authenticate against a Windows NT domain, enter your username including the domain name, like: `MYDOMAIN/johnd`.

5.2. Importing Data Into A Repository

5.2.1. 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:

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
```

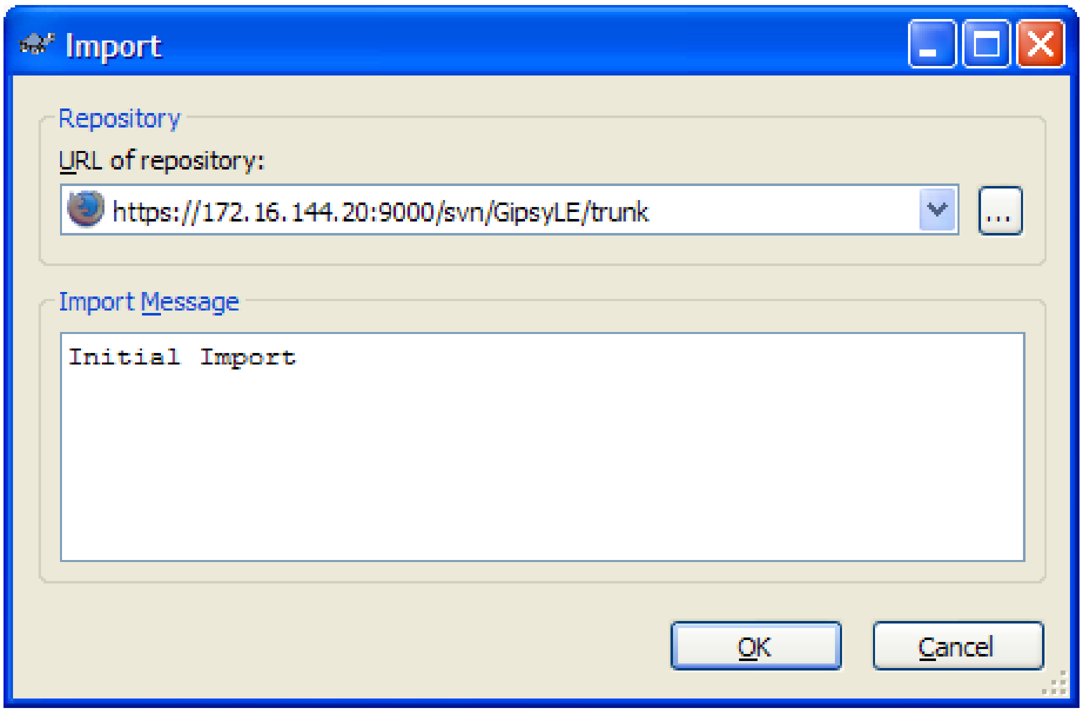
You can also use the repository browser to create new folders directly in the repository.

5.2.2. Import

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 subfolders. 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:



URL of repository: https://172.16.144.20:9000/svn/GipsyLE/trunk

Import Message: Initial Import

Figure 5.6. The Import dialog

In this dialog you have to enter the URL of the repository into which you want to import your project.

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 5.27.1, “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. As before, the name of the folder you import does not appear in the repository, only the folder contents. 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.

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. 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 properly. For example you must replace each space with `%20`. Note that it is not possible to use foldernames 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 subfolder `D:\dev\project1\sounds` and checkout the sounds project, another subfolder `D:\dev\project1\quick graphs` containing the graphics project, and finally a nested subfolder `D:\dev\project1\skins\toolkit` containing revision 21 of the skin-maker project.

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.

Note that if you change the URL in an `svn:externals` property, then next time you update your working copy Subversion will delete the old external folder and make a fresh checkout, so you will see files being `Added`, rather than being `Updated` as you might have expected. This situation might occur if you reference a tag from another project. When a new version of that project is released, you change your external reference to point to the new tag.

Tip

You should strongly consider using explicit revision numbers in all of your externals definitions. 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 codebase.

URLs in `svn:externals` definitions are absolute. You cannot use relative URLs. If you relocate your working copy, or if the external repository is relocated, these URLs will not be updated automatically. Also, if you branch a project which has external references within the same repository, the URLs in the branched copy will be unchanged; you may then want to change trunk references to branch references instead.

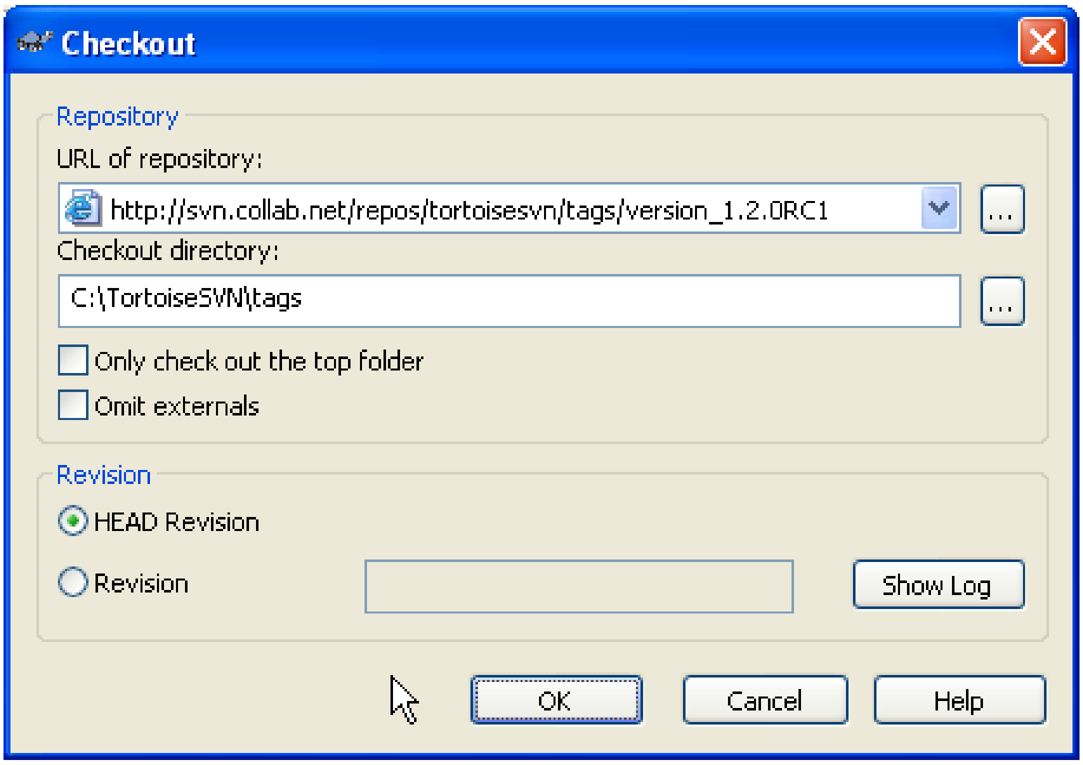
If you need more information how TortoiseSVN handles Properties read Section 5.15, “Project Settings”.

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:



URL of repository: http://svn.collab.net/repos/tortoisesvn/tags/version_1.2.0RC1

Checkout directory: C:\TortoiseSVN\tags

[ ] Only check out the top folder

[ ] Omit externals

Revision

[✓ HEAD Revision]

Revision

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.

Important

You should only check out into an empty directory. If you want to check out your source tree into the same directory that you imported from, Subversion will throw an error message because it will not overwrite the existing unversioned files with versioned ones. You will have to check out into a different directory or delete the existing sourcetree first.

If you want to check out the top level folder only and omit all sub-folders, use the Only check out the top folder checkbox.

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 either of these options are checked, 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 sub-folders and all externals.

It is recommended that you check out only the `trunk` part of the directory tree. 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!

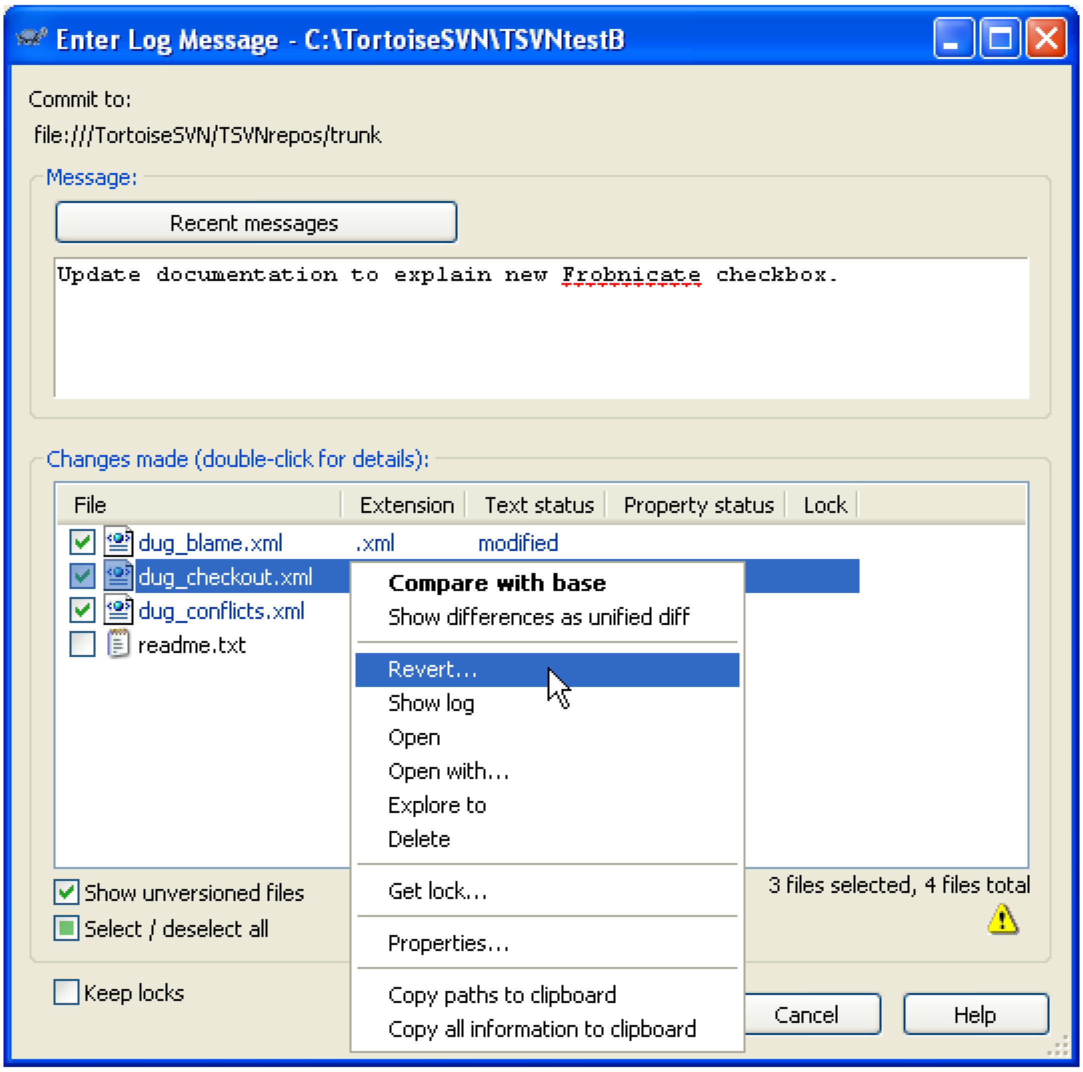
Exporting

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 5.23, “Exporting a Subversion Working Copy” to find out how to do that.

5.4. Sending 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.

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....


Enter Log Message - C:\TortoiseSVN\TSVNtestB

Commit to:
file:///TortoiseSVN/TSVNrepos/trunk

Message:

Update documentation to explain new Frobnicate checkbox.

Changes made (double-click for details):

File	Extension	Text status	Property status	Lock
✓ dug_blame.xml	.xml	modified		
✓ dug_checkout.xml				
✓ dug_conflicts.xml				
readme.txt				

3 files selected, 4 files total

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 TSVN 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 5.11, "Ignoring Files And Directories" for more information.

Doubleclicking 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.

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.

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.

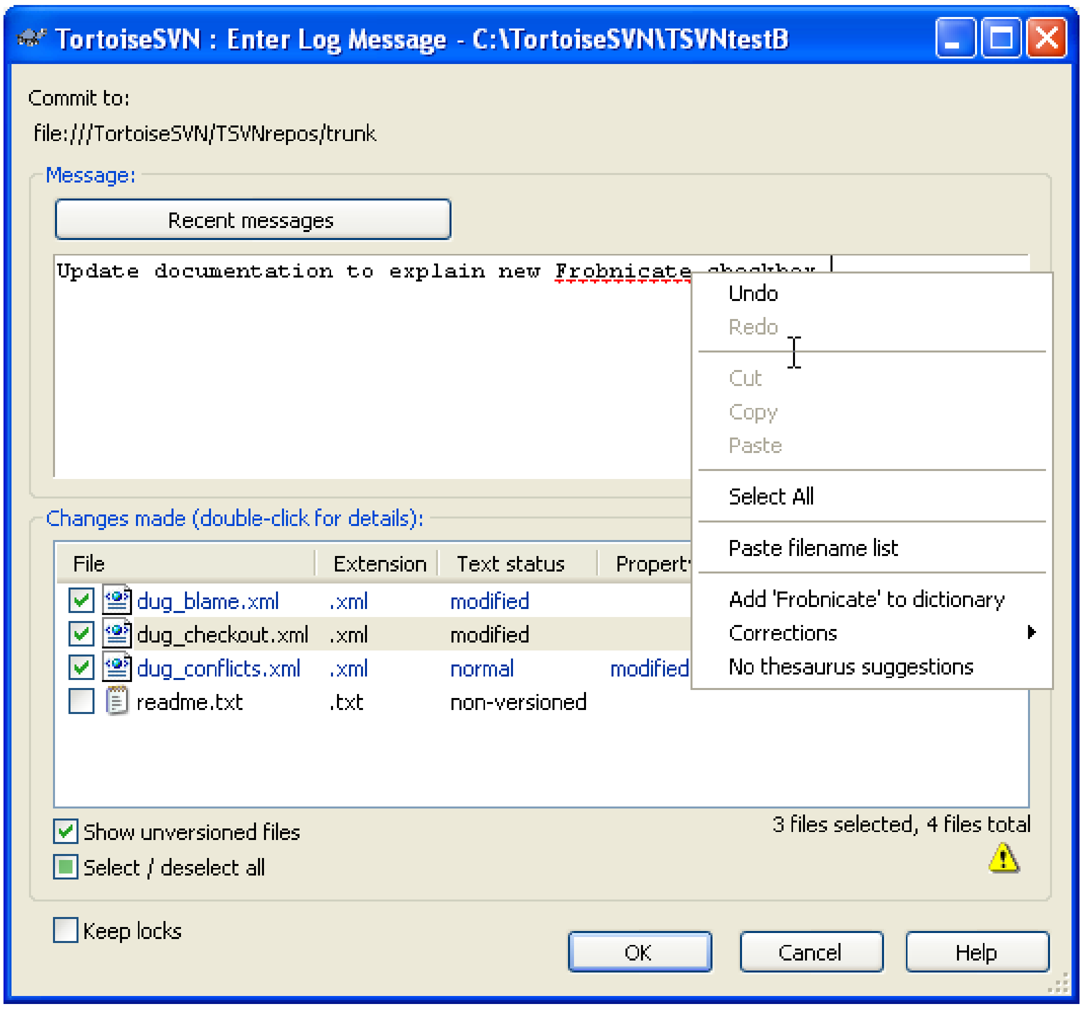
You can drag unversioned files which reside within a working copy into the commit dialog, and they will be SVN added automatically.

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.

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.



TortoiseSVN : Enter Log Message - C:\TortoiseSVN\TSVNtestB

Commit to:
file:///TortoiseSVN/TSVNrepos/trunk

Message:

Update documentation to explain new Frobnicate

Changes made (double-click for details):

File	Extension	Text status	Propert
✓ dug_blame.xml	.xml	modified	
✓ dug_checkout.xml	.xml	modified	
✓ dug_conflicts.xml	.xml	normal	modified
readme.txt	.txt	non-versioned	

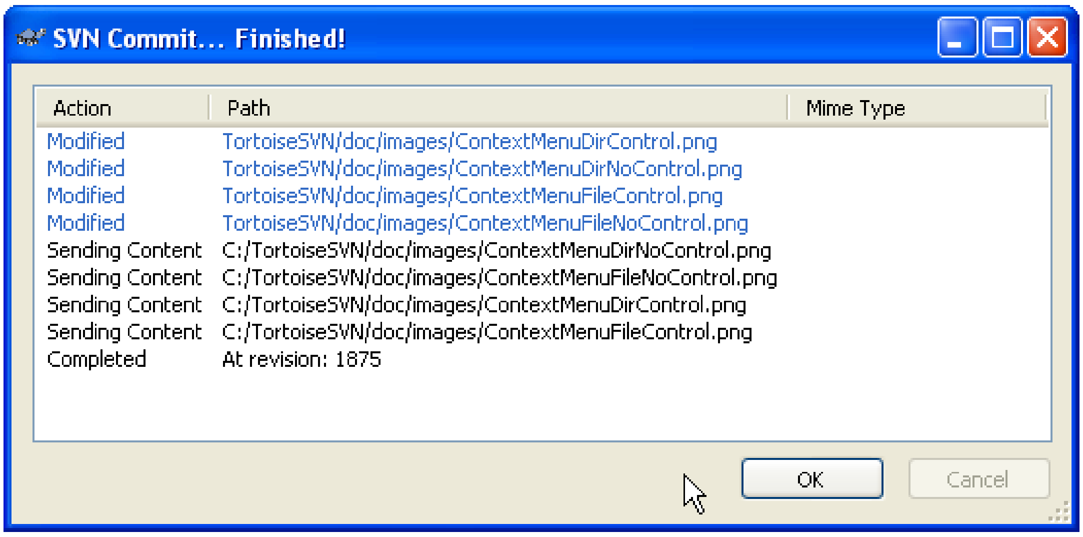
✓ Show unversioned files
Select / deselect all
3 files selected, 4 files total
Keep locks

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), 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 online documentation and tutorial at *http://www.regular-expressions.info/* [http://www.regular-expressions.info/].

After pressing OK, a dialog appears displaying the progress of the commit.



Action	Path	Mime Type
Modified	TortoiseSVN/doc/images/ContextMenuDirControl.png	
Modified	TortoiseSVN/doc/images/ContextMenuDirNoControl.png	
Modified	TortoiseSVN/doc/images/ContextMenuFileControl.png	
Modified	TortoiseSVN/doc/images/ContextMenuFileNoControl.png	
Sending Content	C:/TortoiseSVN/doc/images/ContextMenuDirNoControl.png	
Sending Content	C:/TortoiseSVN/doc/images/ContextMenuFileNoControl.png	
Sending Content	C:/TortoiseSVN/doc/images/ContextMenuDirControl.png	
Sending Content	C:/TortoiseSVN/doc/images/ContextMenuFileControl.png	
Completed	At revision: 1875	

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 5.27.2.5, “TortoiseSVN Colour Settings” for more information.

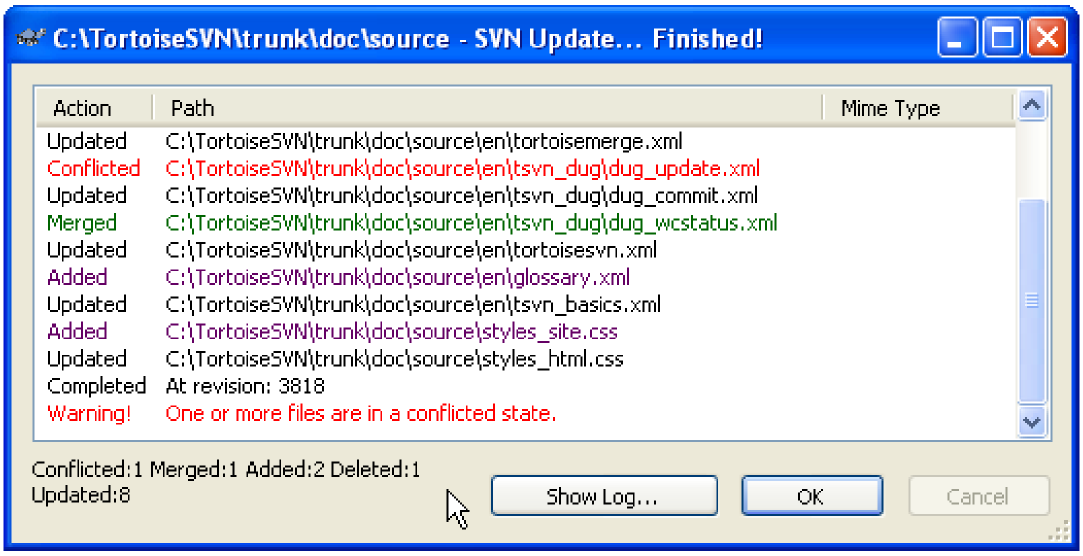
Special Folder Properties

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 5.15, “Project Settings” for further information.

Integration with Bugtracking Tools

If you have activated the bugtracking 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 bugtracking support, just add your issue references as part of the log message. Learn more Section 5.25, “Integration with Bugtracking Systems / Issue trackers”.

5.5. Update Your Working Copy With Changes From Others



```
Action     Path                                                        Mime Type
Updated    C:\TortoiseSVN\trunk\doc\source\en\tortoisemerge.xml
Conflicted C:\TortoiseSVN\trunk\doc\source\en\tsvn_dug\dug_update.xml
Updated    C:\TortoiseSVN\trunk\doc\source\en\tsvn_dug\dug_commit.xml
Merged     C:\TortoiseSVN\trunk\doc\source\en\tsvn_dug\dug_wcstatus.xml
Updated    C:\TortoiseSVN\trunk\doc\source\en\tortoisesvn.xml
Added      C:\TortoiseSVN\trunk\doc\source\en\glossary.xml
Updated    C:\TortoiseSVN\trunk\doc\source\en\tsvn_basics.xml
Added      C:\TortoiseSVN\trunk\doc\source\styles_site.css
Updated    C:\TortoiseSVN\trunk\doc\source\styles_html.css
Completed  At revision: 3818
Warning!   One or more files are in a conflicted state.

Conflicted:1 Merged:1 Added:2 Deleted:1
Updated:8
```

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 5.27.2.5, “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 to update the current folder non-recursively (without all the subfolders) and you can 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 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

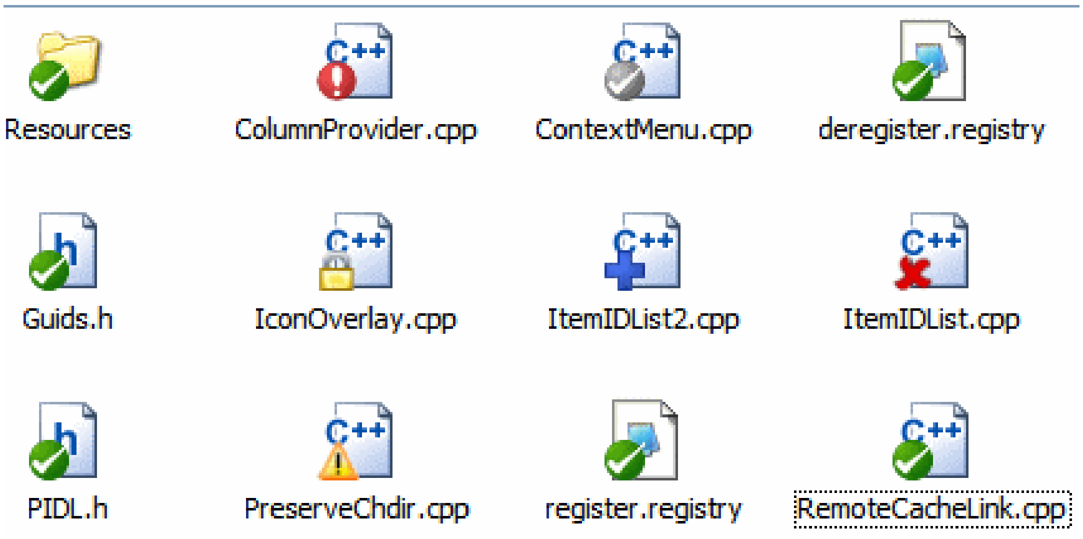


Figure 5.12. Explorer showing icon overlays

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. Read-only 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.

Unlike TortoiseCVS (the CVS shell integration) no overlay icon for unversioned files is shown. We do this because the number of icon overlays are limited system wide and should be used economically.

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 you are also using TortoiseCVS, then there are not enough overlay slots available, so 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.
- `ReadOnly` 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.

5.7.2. TortoiseSVN Columns In Windows Explorer

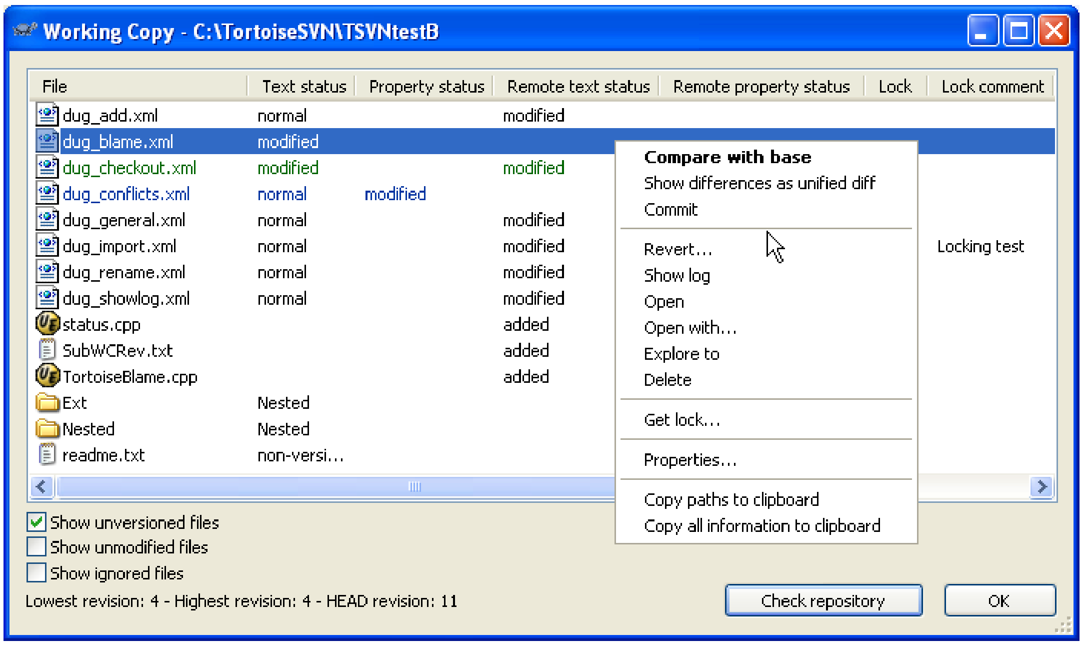
The same information which is available from the icon overlays (and much more) can be displayed as additional columns in Windows Explorer's Details View.

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.

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


Working Copy - C:\TortoiseSVN\TSVNtestB

File	Text status	Property status	Remote text status	Remote property status	Lock	Lock comment
dug_add.xml	normal		modified			
dug_blame.xml	modified					
dug_checkout.xml	modified		modified			
dug_conflicts.xml	normal	modified				
dug_general.xml	normal		modified			
dug_import.xml	normal		modified			Locking test
dug_rename.xml	normal		modified			
dug_showlog.xml	normal		modified			
status.cpp			added			
SubWCRev.txt			added			
TortoiseBlame.cpp			added			
Ext	Nested					
Nested	Nested					
readme.txt	non-versi...					

✓ Show unversioned files
☐ Show unmodified files
☐ Show ignored files

Lowest revision: 4 - Highest revision: 4 - HEAD revision: 11

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 5.27.2.5, “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.

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. 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.

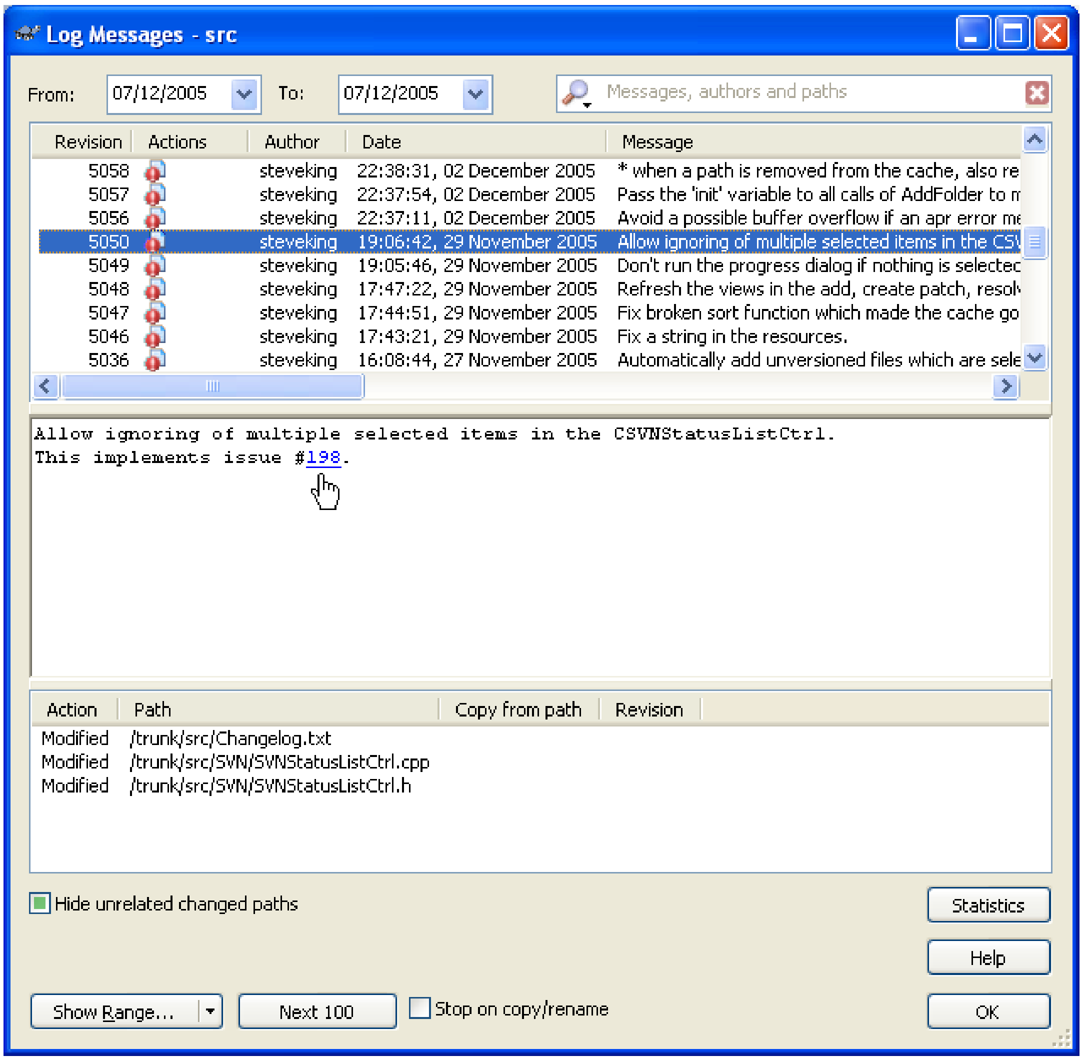
  Lines shown in blue indicate that something has been copied to this development line (perhaps from a branch).

- 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.

5.8.1. Invoking the Revision Log Dialog



Revision	Actions	Author	Date	Message
5058		steveking	22:38:31, 02 December 2005	* when a path is removed from the cache, also re
5057		steveking	22:37:54, 02 December 2005	Pass the 'init' variable to all calls of AddFolder to r
5056		steveking	22:37:11, 02 December 2005	Avoid a possible buffer overflow if an apr error me
5050		steveking	19:06:42, 29 November 2005	Allow ignoring of multiple selected items in the CSV
5049		steveking	19:05:46, 29 November 2005	Don't run the progress dialog if nothing is selectec
5048		steveking	17:47:22, 29 November 2005	Refresh the views in the add, create patch, resolv
5047		steveking	17:44:51, 29 November 2005	Fix broken sort function which made the cache go
5046		steveking	17:43:21, 29 November 2005	Fix a string in the resources.
5036		steveking	16:08:44, 27 November 2005	Automatically add unversioned files which are sele

```
Allow ignoring of multiple selected items in the CSVNStatusListCtrl.
This implements issue #198.
```

Action	Path	Copy from path	Revision
Modified	/trunk/src/Changelog.txt		
Modified	/trunk/src/SVN/SVNStatusListCtrl.cpp		
Modified	/trunk/src/SVN/SVNStatusListCtrl.h		

Figure 5.14. 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

5.8.2. Getting Additional Information

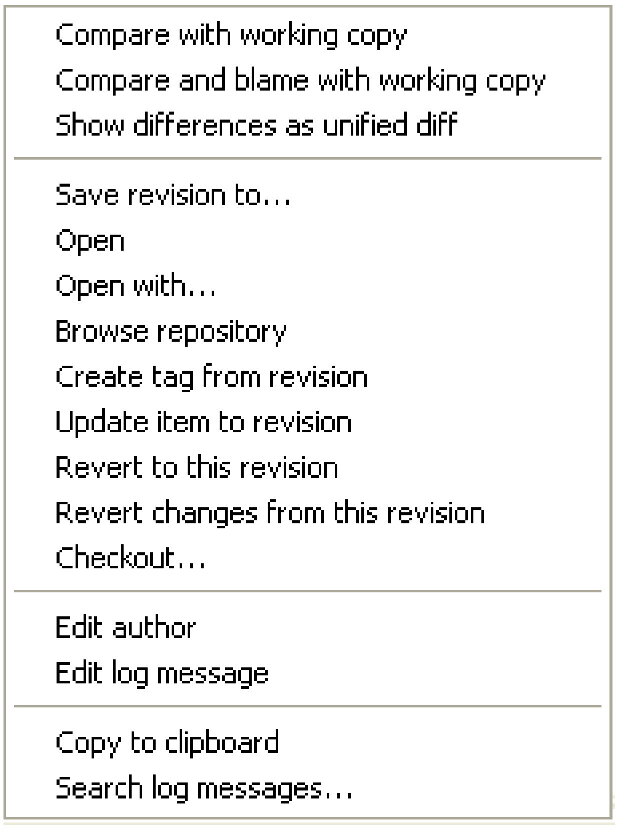


Figure 5.15. The Revision Log Dialog Top Pane with Context Menu

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

- 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 results using a visual diff tool. Read Section 5.20.2, “Blame Differences” for more detail.
- Blame the selected revision, and the previous revision, and compare the results using a visual diff tool.
- Save the selected revision to a file so you have an older version of that file. This option is only available when you access the log for a file, and it saves a version of that one file only.
- Open the selected file, either with the default viewer for that file type, or with a program you choose. This option is only available when you access the log for a file.
- Open the repository browser to examine the selected folder. This option is only available when you access the log for a directory.
- 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.
- 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.
- 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 5.8.4, “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.

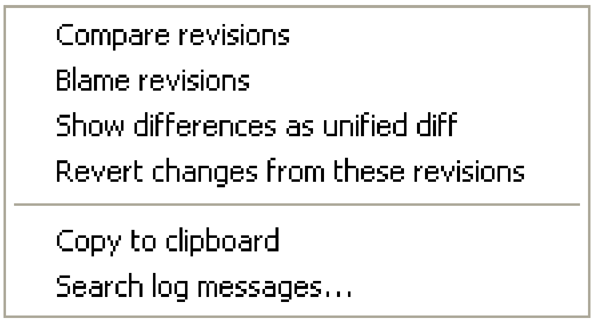


Figure 5.16. Top Pane Context Menu for 2 Selected Revisions

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 5.9.2, “Comparing Folders”.

- Blame the two revisions and compare the results using a visual difference tool. Read Section 5.20.2, “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 multiple consecutive revisions (using the usual **Ctrl** or **Shift** modifiers), the context menu will include an entry to Revert all changes which were made in that range of revisions. This is the easiest way to rollback a group of revisions in one go.

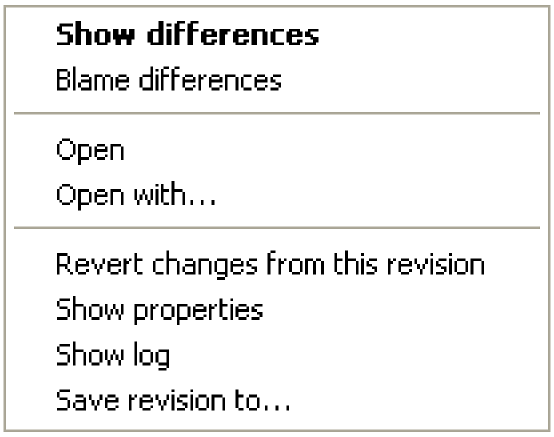


Figure 5.17. The Log Dialog Bottom Pane with Context Menu

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

- Show differences 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 show the differences using a visual diff tool. Read Section 5.20.2, “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.

5.8.3. 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 5.27, “TortoiseSVN's Settings”),
- 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.

5.8.4. 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/1.2/svn.reposadmin.create.html] 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*.

5.8.5. 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.

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 tri-state: you can show all paths, grey out the unrelated ones, or hide the unrelated paths completely.

5.8.6. 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 ;-)

5.8.6.1. 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.

5.8.6.2. Commits by Author Page

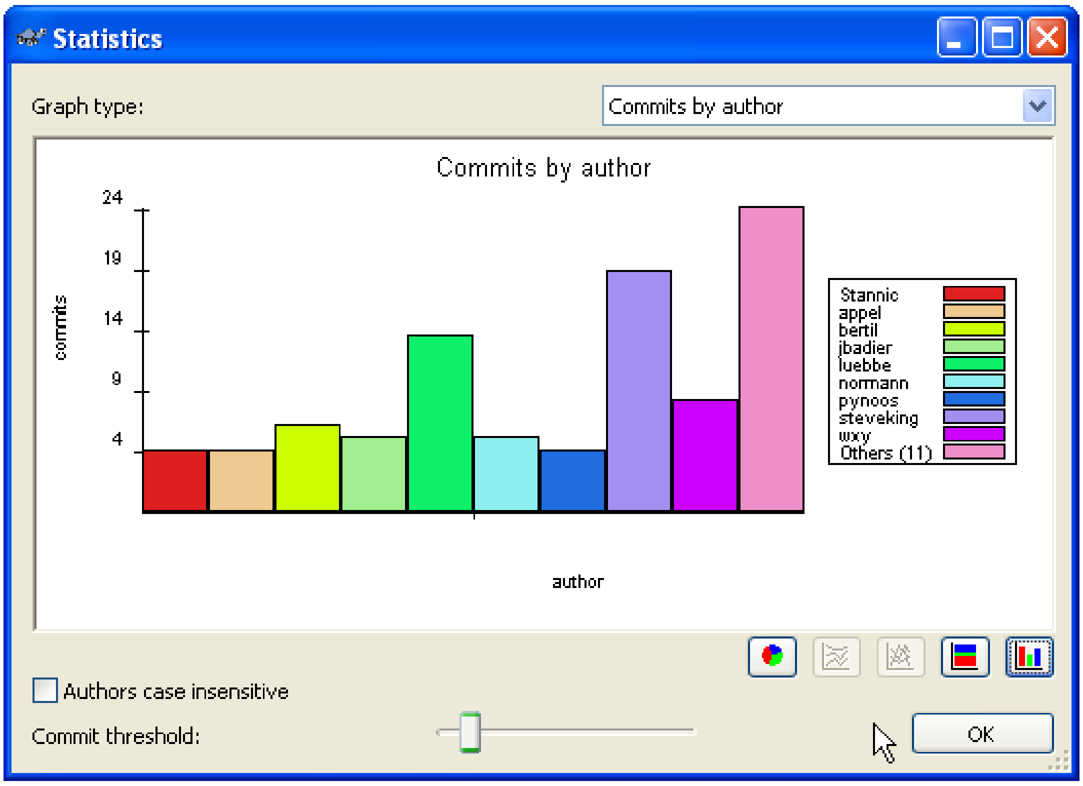


Figure 5.18. Commits-by-Author Histogram

This graph shows you which authors have been active on the project as a simple histogram, stacked histogram or pie chart.

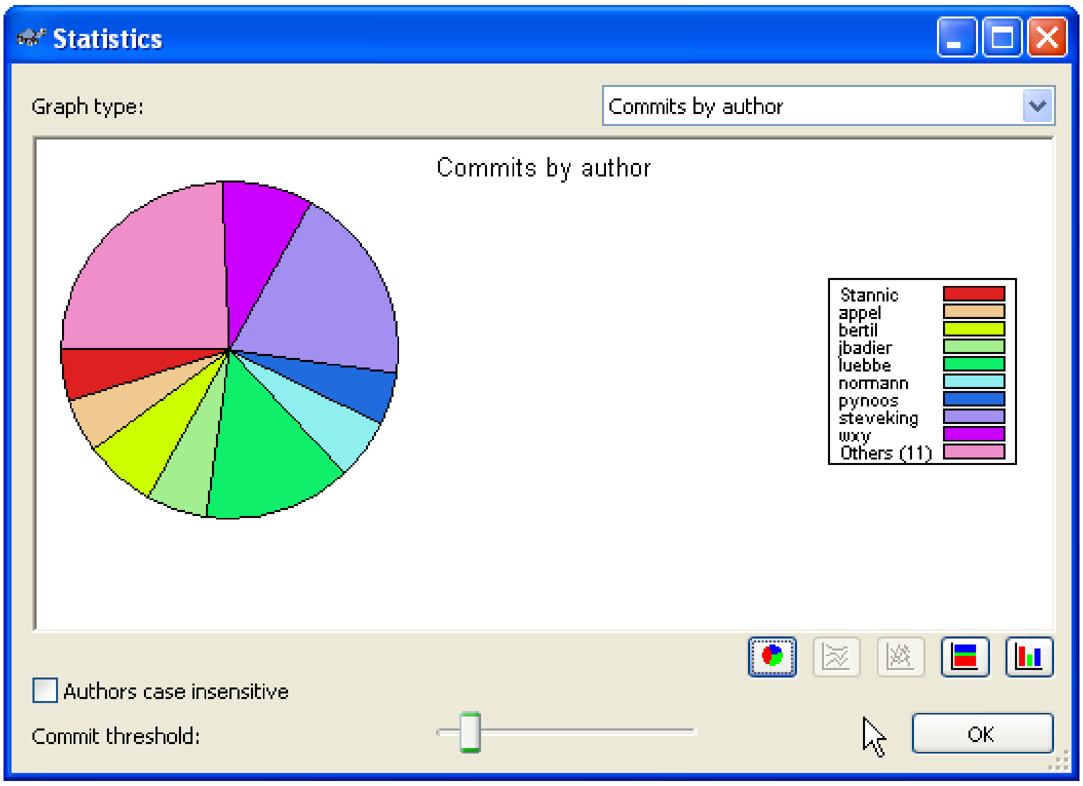


Figure 5.19. Commits-by-Author Pie Chart

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 per centage of total commits) below which any activity is grouped into an *Others* category.

5.8.6.3. Commits by date Page

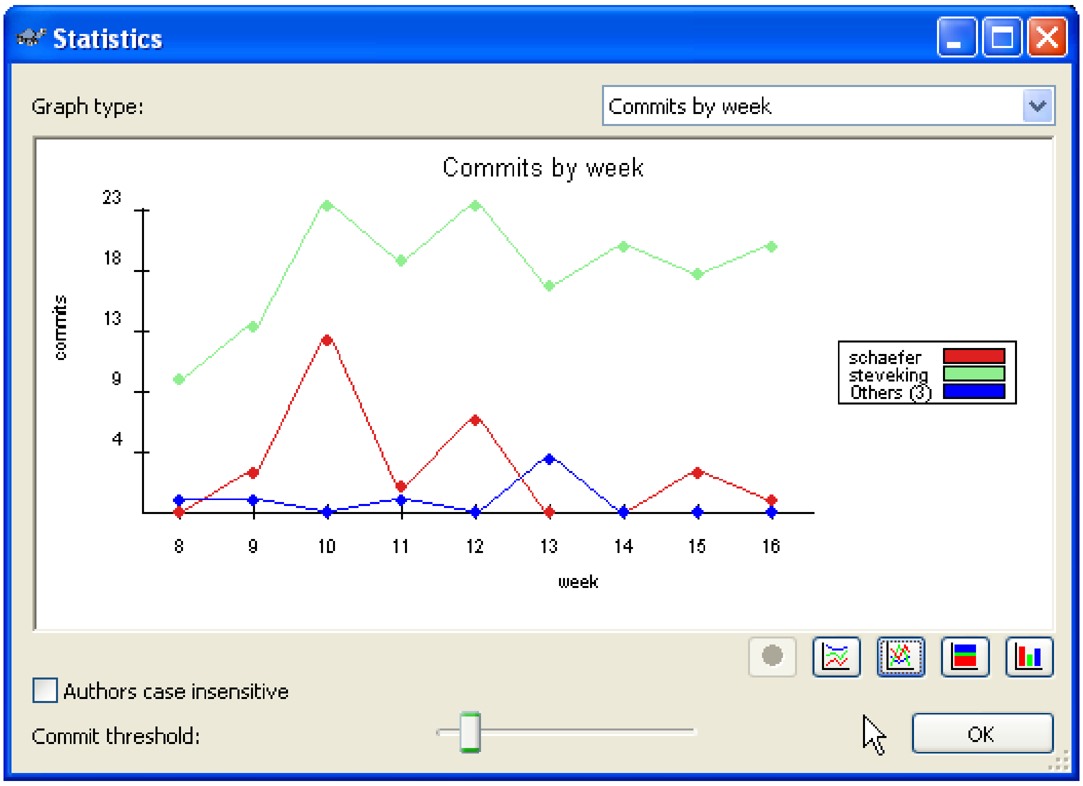


Figure 5.20. Commits-by-date Graph

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 usernames 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. 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 builtin 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.

5.9.1. 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 5.9.2, “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.

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 5.9.2, “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.

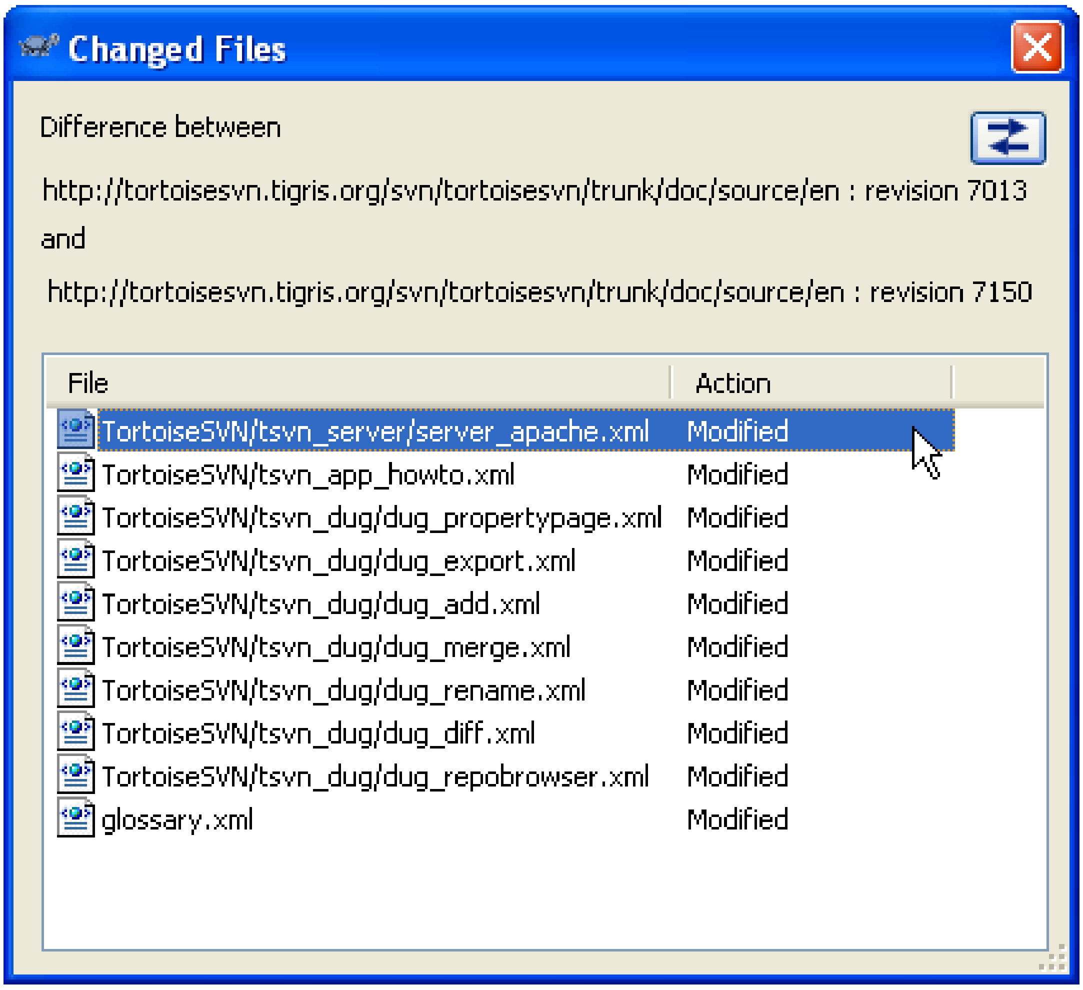
Difference between files
: If you want to see the differences between two different files, you can do that directly in explorer by selecting both files (using the usual **Ctrl**-modifier). Then from the explorer context menu select TortoiseSVN → Diff.

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 patchfile - 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 5.20.2, “Blame Differences” for more detail.

Difference between folders
: The builtin 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 5.9.4, “External Diff/Merge Tools” we tell you about some tools which we have used.

5.9.2. Comparing Folders



Difference between

http://tortoisesvn.tigris.org/svn/tortoisesvn/trunk/doc/source/en : revision 7013

and

http://tortoisesvn.tigris.org/svn/tortoisesvn/trunk/doc/source/en : revision 7150

File	Action
TortoiseSVN/tsvn_server/server_apache.xml	Modified
TortoiseSVN/tsvn_app_howto.xml	Modified
TortoiseSVN/tsvn_dug/dug_propertypage.xml	Modified
TortoiseSVN/tsvn_dug/dug_export.xml	Modified
TortoiseSVN/tsvn_dug/dug_add.xml	Modified
TortoiseSVN/tsvn_dug/dug_merge.xml	Modified
TortoiseSVN/tsvn_dug/dug_rename.xml	Modified
TortoiseSVN/tsvn_dug/dug_diff.xml	Modified
TortoiseSVN/tsvn_dug/dug_repobrowser.xml	Modified
glossary.xml	Modified

Figure 5.21. 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.

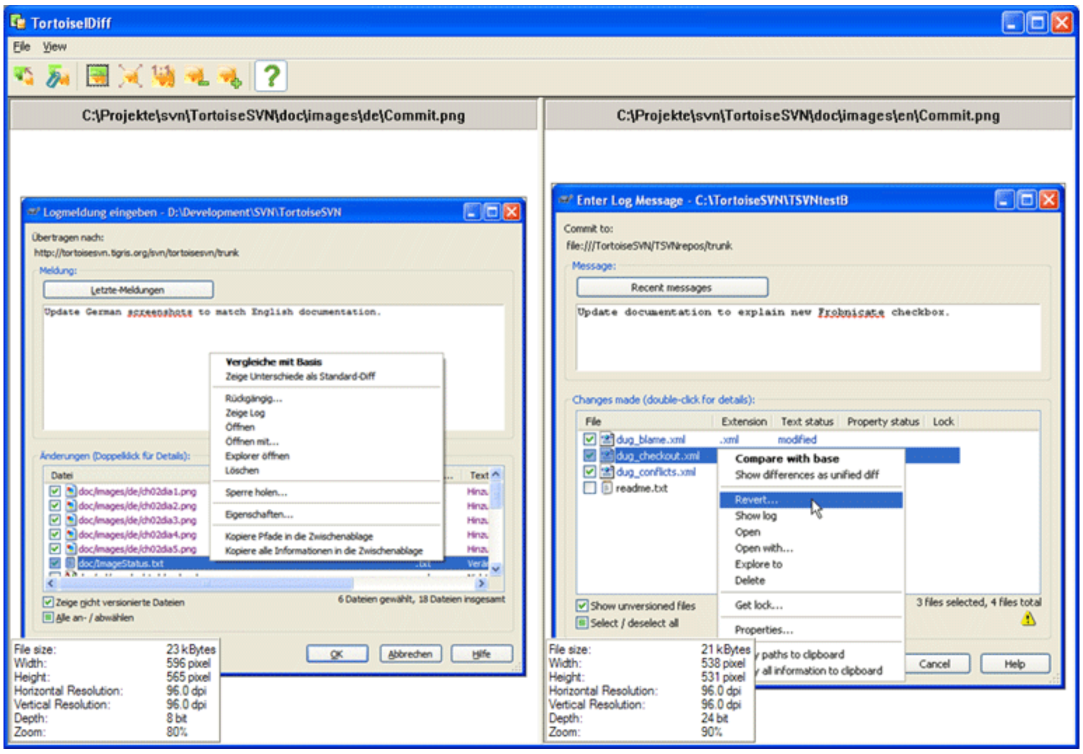
If you want to export the list of files *and* the actions (modified, added, deleted) as well, you can do that using the keyboard shortcuts Ctrl-A to select all entries and Ctrl-C to copy the detailed list to the clipboard.

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.

5.9.3. 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.


TortoiseIDiff
File View
C:\Projekte\svn\TortoiseSVN\doc\images\de\Commit.png
C:\Projekte\svn\TortoiseSVN\doc\images\en\Commit.png
Logmeldung eingeben - D:\Development\SVN\TortoiseSVN
Enter Log Message - C:\TortoiseSVN\TSVNtestB
Letzte Meldungen
Recent messages
Vergleiche mit Basis
Compare with base
Revert...
Show log
OK
Abbrechen
Hilfe
Cancel
Help

Update German screenshots to match English documentation.

Update documentation to explain new Frobnicate checkbox.

File size:	23 kBytes
Width:	596 pixel
Height:	565 pixel
Horizontal Resolution:	96.0 dpi
Vertical Resolution:	96.0 dpi
Depth:	8 bit
Zoom:	80%

File size:	21 kBytes
Width:	538 pixel
Height:	531 pixel
Horizontal Resolution:	96.0 dpi
Vertical Resolution:	96.0 dpi
Depth:	24 bit
Zoom:	90%

Figure 5.22. 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. A slider at the top controls the relative intensity of the images (alpha blend). You can also use Ctrl-Shift-Wheel to change the blend.

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, wheelmouse) on both images are linked.

In 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.

5.9.4. 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 comparision 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 5.27.4, “External Program Settings” for information on how to set up TortoiseSVN to use these tools.

5.10. Adding New Files And Directories

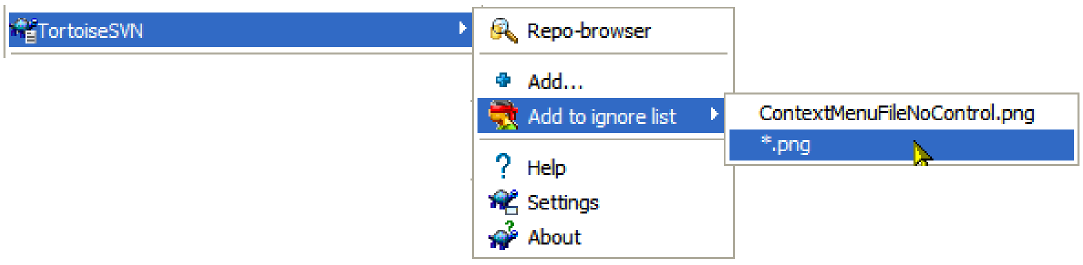


Figure 5.23. Explorer context menu for unversioned files

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 them to add 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.

5.11. Ignoring Files And Directories

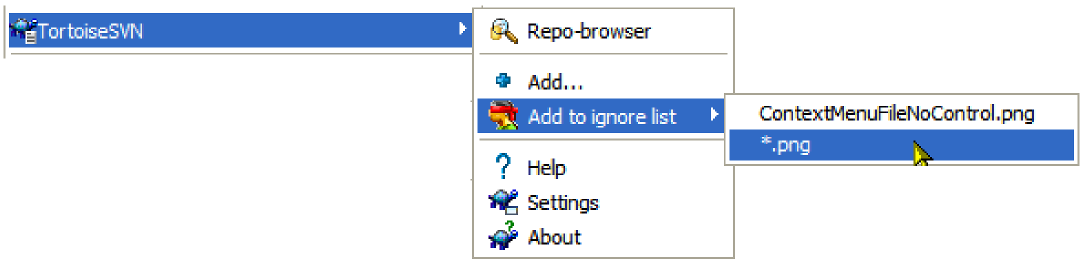


Figure 5.24. Explorer context menu for unversioned files

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, TSVN 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 5.15, "Project Settings" for more information on setting properties directly.

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 5.27.1, "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.

5.11.1. Filename Globbing 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`.

The globbing is done by Subversion, so the path delimiter is always `/`, 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]`.

If directory names are present in a path, they are included in the matching, so pattern `Fred.*` will match `Fred.c` but not `subdir/Fred.c`. This is significant if you add a folder containing some files that you want to be ignored, because those filenames will be preceded with the folder name.

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].

5.12. Deleting, Renaming And Moving

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

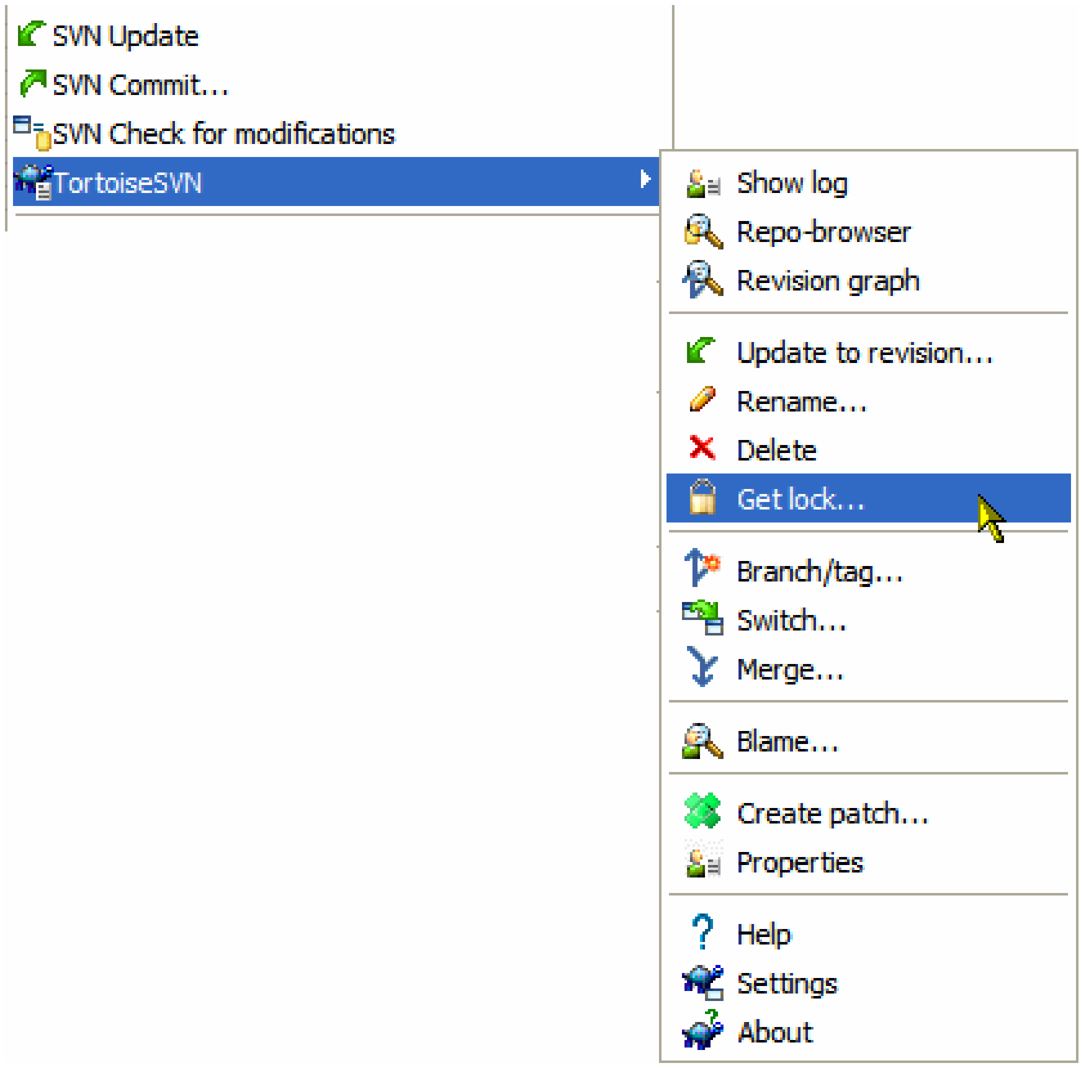


Figure 5.25. Explorer context menu for versioned files

If you delete a file/directory using TSVN, the file is removed from your working copy and marked for deletion. The file's parent folder shows a "deleted" icon overlay. You can always get the file back, if you call TortoiseSVN → Revert on the parent folder.

If you want to move files inside a working copy, use the drag-and-drop handler again:

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

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.

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.

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 an update of your coworkers won't remove the old file. 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.

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.

5.12.1. Renaming a file only in case

In case you have two files in the repository with the same name but differing only in case (e.g. TEST.TXT and test.txt) you can't update or checkout the directory where those files are in anymore.

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.

There are (at least) two possible solutions 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.

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.

5.12.2. 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.

5.12.3. Deleting Unversioned Files

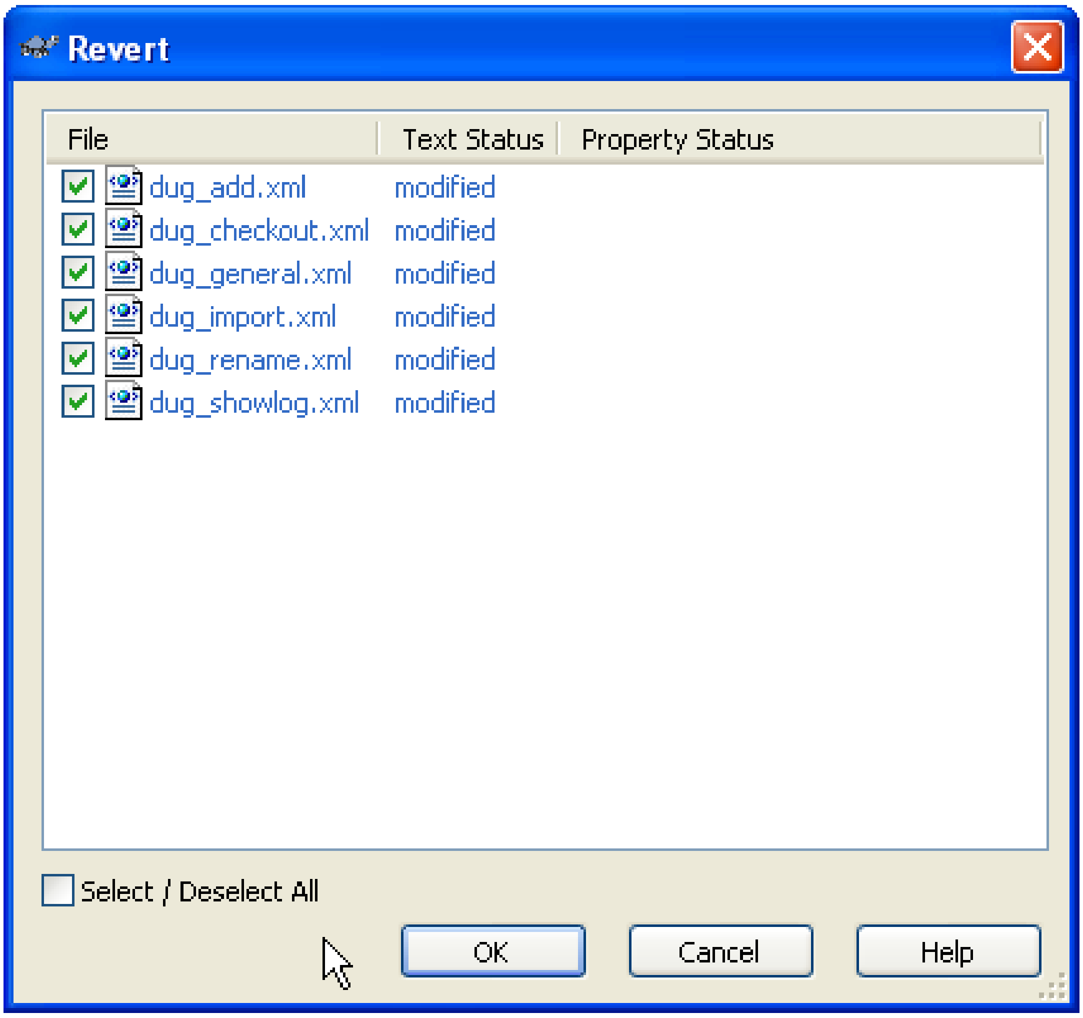
Usually you set your ignore list such that all generated files are ignored in Subversion. But what if you want to clear all those ignored items to produce a clean build? Usually you would set that in your makefile, but if you are debugging the makefile, or changing the build system it is useful to have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

5.13. 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.



File	Text Status	Property Status
✓ dug_add.xml	modified	
✓ dug_checkout.xml	modified	
✓ dug_general.xml	modified	
✓ dug_import.xml	modified	
✓ dug_rename.xml	modified	
✓ dug_showlog.xml	modified	

Figure 5.26. Revert dialog

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 5.8, “Revision Log Dialog” for further information.

5.14. Cleanup

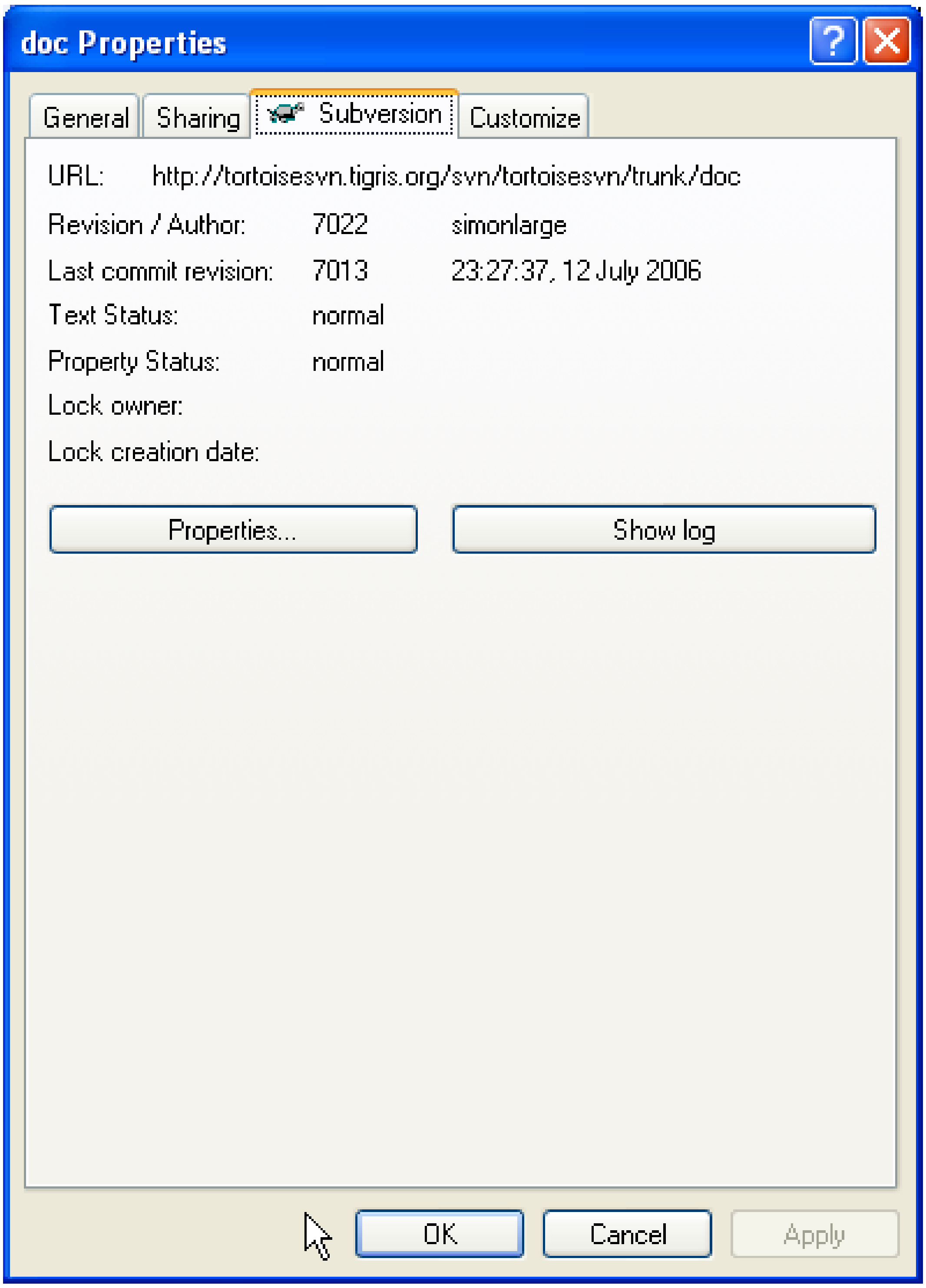
If a Subversion command cannot complete successfully, perhaps due to server problems, your working copy can be left in an inconsistent state. In that case you need to use TortoiseSVN → Cleanup on the folder. It is a good idea to do this at the top level of the working copy.

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 repond. 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.

5.15. Project Settings



URL: http://tortoisesvn.tigris.org/svn/tortoisesvn/trunk/doc

Revision / Author: 7022 simonlarge

Last commit revision: 7013 23:27:37, 12 July 2006

Text Status: normal

Property Status: normal

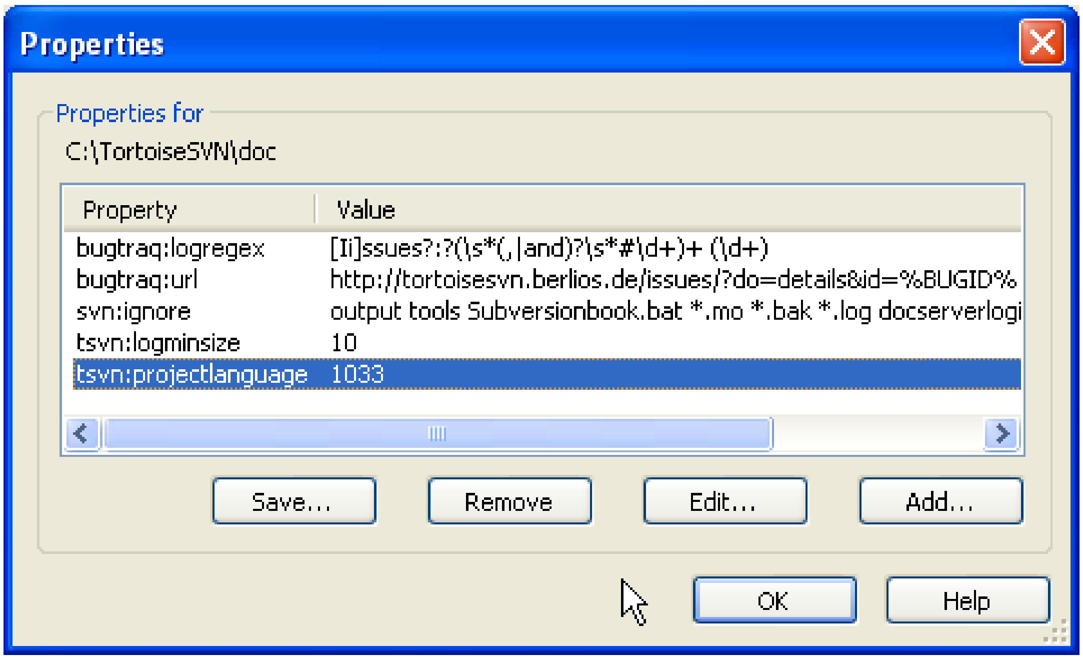
Lock owner:

Lock creation date:

Figure 5.27. 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.

5.15.1. Subversion Properties



C:\TortoiseSVN\doc

Property	Value
bugtraq:logregex	[Ii]ssues?:?(\s*(,|and)?\s*#\d+)+ (\d+)
bugtraq:url	http://tortoisesvn.berlios.de/issues/?do=details&id=%BUGID%
svn:ignore	output tools Subversionbook.bat *.mo *.bak *.log docserverlogi
tsvn:logminsize	10
tsvn:projectlanguage	1033

Figure 5.28. Subversion property page

You can read and set the Subversion properties from the Windows properties dialog, but also from TortoiseSVN → properties and within TSVN'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”. For more information about properties in Subversion see the *Special Properties* [http://svnbook.red-bean.com/en/1.2/svn.advanced.props.html].

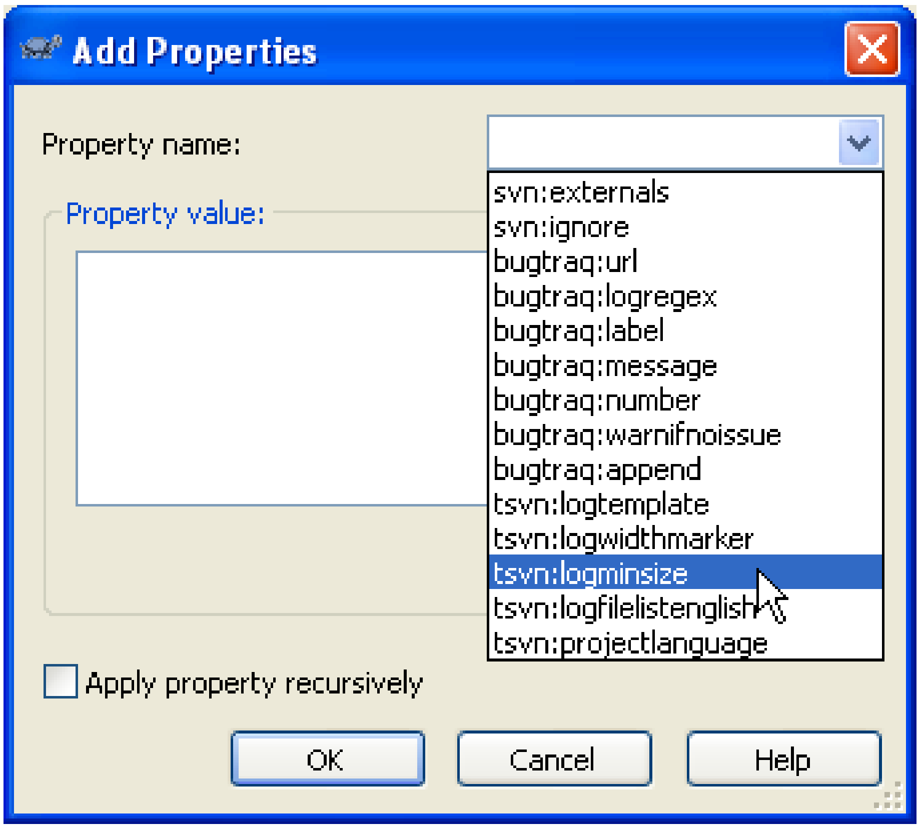


Figure 5.29. Adding properties

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”.

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.

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.

Automatic property setting

You can configure Subversion to set properties automatically on files and folders when they are added to the repository. Read Section 5.27, “TortoiseSVN's Settings” for further information.

5.15.2. TortoiseSVN 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.

- 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://msdn.microsoft.com/library/default.asp?url=/library/en-us/intl/nls_238z.asp].

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`.

- 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. TSVN also understands `yes` as a synonym for `true` and `no` as a synonym for `false`.

Set the tsvn: properties on folders

These `tsvn:` 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 subfolder, then it is sufficient to set the properties on `trunk/`. If you can't be sure, you should set the properties recursively on each subfolder. 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 subfolders in the hierarchy, without also setting it on all files.

TortoiseSVN can integrate with some bugtracking tools. This uses properties, which start with `bugtraq:`. Read Section 5.25, “Integration with Bugtracking Systems / Issue trackers” for further information.

It can also integrate with some web-based repository browsers. Read Section 5.26, “Integration with Web-based Repository Viewers” for further information.

5.16. Branching / Tagging

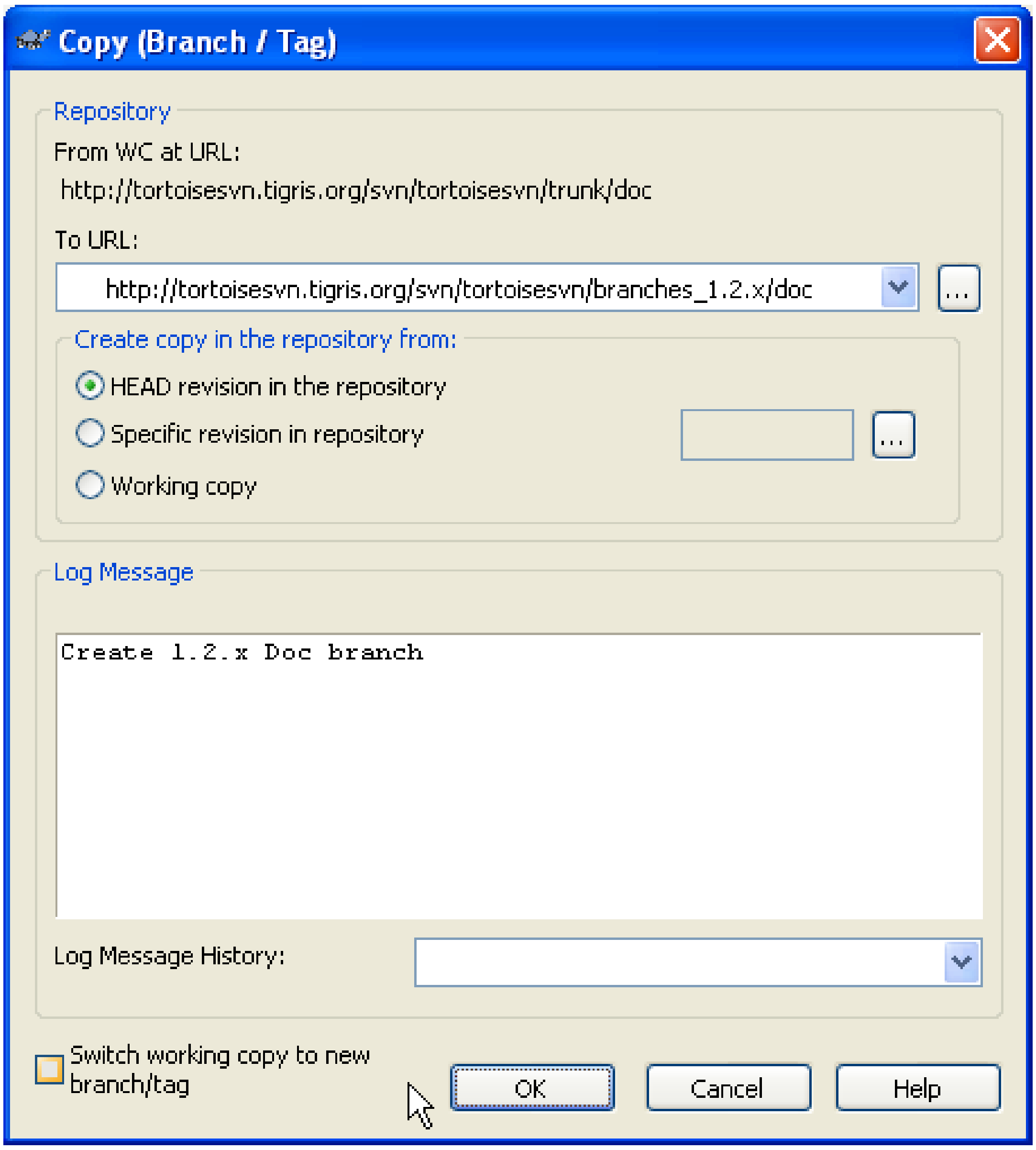
One of the features of version control systems is the ability to isolate changes onto a separate line of development. This line is known as a *branch*. Branches are often used to try out new features without disturbing the main line of development with compiler errors and bugs. As soon as the new feature 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.

5.16.1. 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:


Copy (Branch / Tag)
Repository
From WC at URL:
http://tortoisesvn.tigris.org/svn/tortoisesvn/trunk/doc
To URL:
http://tortoisesvn.tigris.org/svn/tortoisesvn/branches_1.2.x/doc
...
Create copy in the repository from:
HEAD revision in the repository
Specific revision in repository
...
Working copy
Log Message
Log Message History:
Switch working copy to new branch/tag
OK
Cancel
Help

Create 1.2.x Doc branch

Figure 5.30. 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 creating a Branch or Tag does *not* affect your working copy. Even if you copy your WC, those changes are committed to the new branch, not to the trunk, so your WC may still be marked as modified.

5.16.2. To Checkout or to Switch...

...that is (not really) the question. While a checkout checks out everything from the desired branch into 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 copy 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).

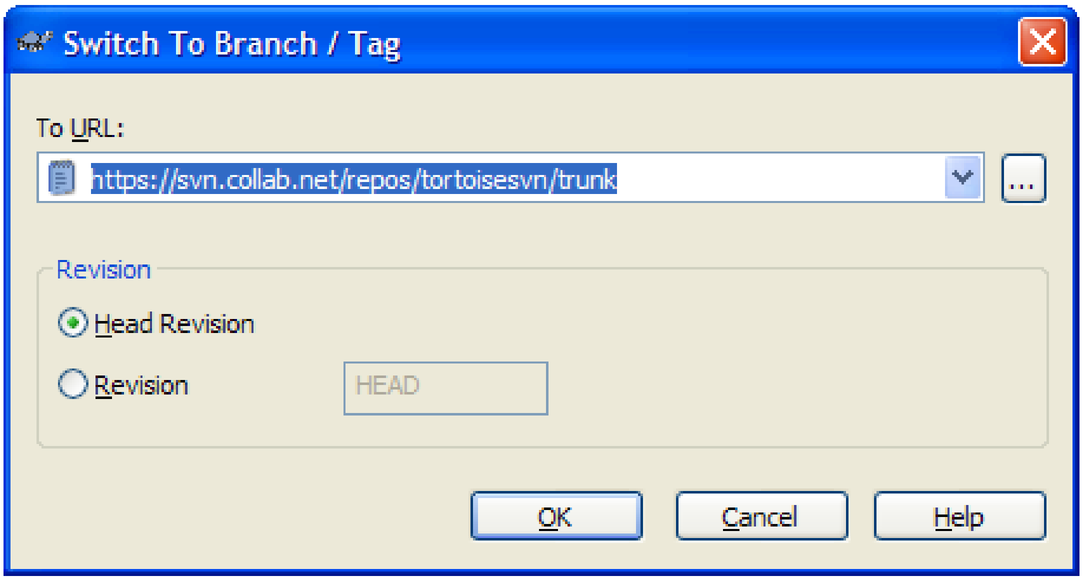


Figure 5.31. The Switch Dialog

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.

5.17. 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/1.2/svn.branchmerge.html] in the Subversion book, which gives a full description and many examples of how it is used.

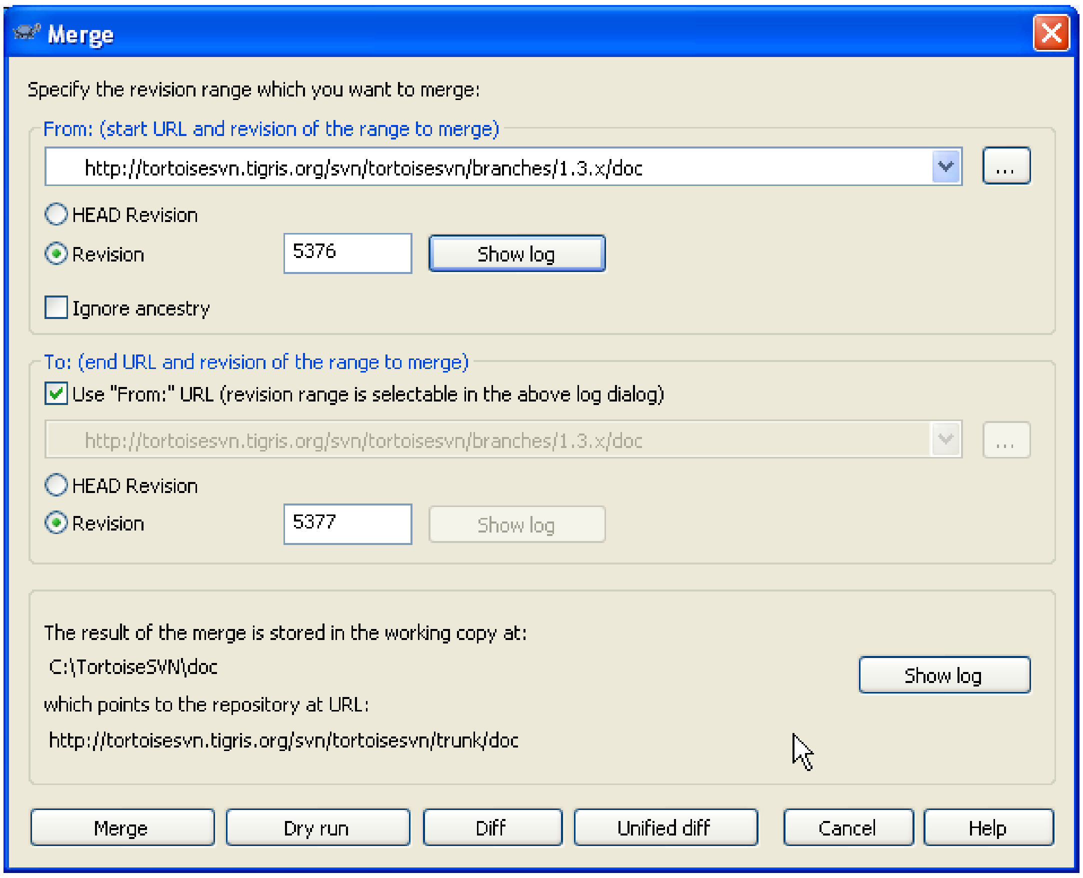
An important point to remember is that Merge is closely related to Diff. The merge process works by generating a list of differences between two points in the repository, and applying those differences to your working copy. For example if you want to merge the changes made in revision N then you have to compare revision N with revision (N-1). Novices often ask “Why do I have to subtract 1 from the start revision.” Think of the underlying Diff process and it will become clearer. TO make this easier, when you use Show Log to select a range of revisions to merge, TortoiseSVN makes this adjustment for you automatically.

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 two common use cases for merging which are handled in slightly different ways, as described below.

5.17.1. Merging 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.


Merge

Specify the revision range which you want to merge:

From: (start URL and revision of the range to merge)

http://tortoisesvn.tigris.org/svn/tortoisesvn/branches/1.3.x/doc

HEAD Revision [✓ Revision] 5376

Ignore ancestry: ☐

To: (end URL and revision of the range to merge)

☑ Use "From:" URL (revision range is selectable in the above log dialog)

http://tortoisesvn.tigris.org/svn/tortoisesvn/branches/1.3.x/doc

HEAD Revision [✓ Revision] 5377

The result of the merge is stored in the working copy at:

C:\TortoiseSVN\doc

which points to the repository at URL:

http://tortoisesvn.tigris.org/svn/tortoisesvn/trunk/doc

Figure 5.32. The Merge Dialog

To merge revisions you need to go to a working copy of the branch in which you want to receive the changes, often the trunk. Select TortoiseSVN → Merge... from the context menu.

1. 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.

2. Because you are porting a range of revisions from the same branch into your working copy, make sure the Use "From:" URL checkbox is checked.

3. In the From Revision field enter the start revision number. This is the revision *before* the changes you want to merge. Remember that Subversion will create a diff file in order to perform the merge, so the start point has to be just before the first change you are interested in. For

example, your log messages may look something like this:

```
Rev Comments
39. Working on MyBranch
38. Working on trunk
37. Working on MyBranch
36. Create branch MyBranch
35. Working on trunk
34. Working on trunk
         ...
```

If you now want to merge all the changes from MyBranch into the trunk you have to choose 36 as the From Revision, not 37 as you might think. If you select revision 37 as the start point, then the difference engine compares the end point with revision 37, and will miss the changes made in revision 37 itself. If that sounds complicated, don't worry, there is an easier way in TortoiseSVN ...

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 revision numbers of the From revision and To revision in the Merge dialog will *both* be filled in for you.

When the Use "From:" URL checkbox is checked, only one Show Log button is enabled. This is because the Show Log dialog sets both From: and To: revisions, so you need to use the multiple selection method outlined above.

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.

4. If you have not used Show Log to select the revision range, then you will need to set the To Revision manually. Enter the last revision number in the range you want to merge. Often this will be the HEAD revision, although it doesn't need to be - you may just want to merge a single revision.

   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.

5. Click OK to complete the merge.

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.

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. Unfortunately merge information is not stored by Subversion. For more information about this, refer to *Best Practices for Merging* [http://svnbook.red-bean.com/en/1.2/svn.branchmerge.copychanges.html] in the Subversion book.

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.

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.

5.17.2. Merging Two Different Trees

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. So in this special case, you would merge by comparing the branch with the trunk.

To merge the feature branch back into the trunk you need to go to a working copy of the trunk. Select TortoiseSVN → Merge... from the context menu.

1. 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.
2. Because you are comparing two different trees, make sure the Use "From:" URL checkbox is *not* checked.
3. In the To: field enter the full folder url of the feature branch.
4. 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. Note that in this case you are not selecting a range of revisions, so the revision you select there is what will actually appear in the Revision field.
5. Click OK to complete the merge.

In this case you will not need the feature branch again because the new feature is now integrated into the trunk. The feature branch is redundant and can be deleted from the repository if required.

5.17.3. Previewing Merge Results

If you are uncertain about the merge operation, you may want to preview what will happen do before you allow it to change your working copy. There are three additional buttons to help you.

Unified Diff creates the diff file (remember that merge is based on diff) and shows you which lines will be changed in your working copy files. As this is a unified diff (patch) file it is not always easy to read out of context, but for small scale changes it can be helpful as it shows all the changes in one hit.

Diff shows you a list of changed files. Double click on any of the listed files to start the diff viewer. Unlike the unified diff, this shows you the changes in their full contextual detail. As with the unified diff, the changes you see here are the changes between the From: and To: revisions. It does not show how your working copy will be affected by applying that change.

Dry Run 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.

5.17.4. Ignoring Ancestry

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/1.2/svn.branchmerge.copychanges.html]

5.18. 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 VCS 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.

5.18.1. 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.

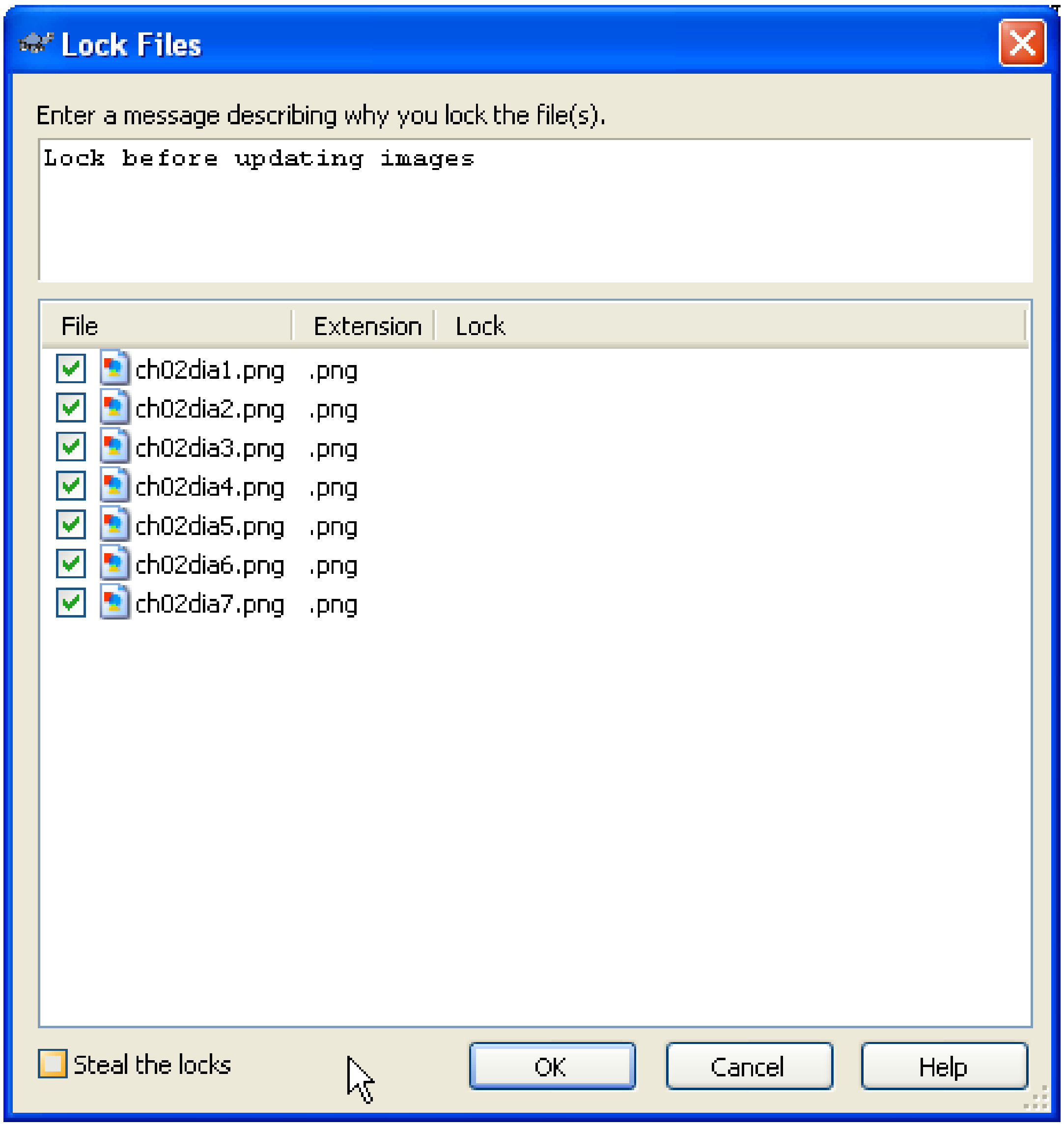
Locks are recorded by working copy location as well as by owner. If you have several working copies (at home, at work) then you can only hold a lock in *one* of those working copies.

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.

5.18.2. 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....



```
Lock before updating images
```

File	Extension	Lock
✓ ch02dia1.png	.png	
✓ ch02dia2.png	.png	
✓ ch02dia3.png	.png	
✓ ch02dia4.png	.png	
✓ ch02dia5.png	.png	
✓ ch02dia6.png	.png	
✓ ch02dia7.png	.png	

Figure 5.33. 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.

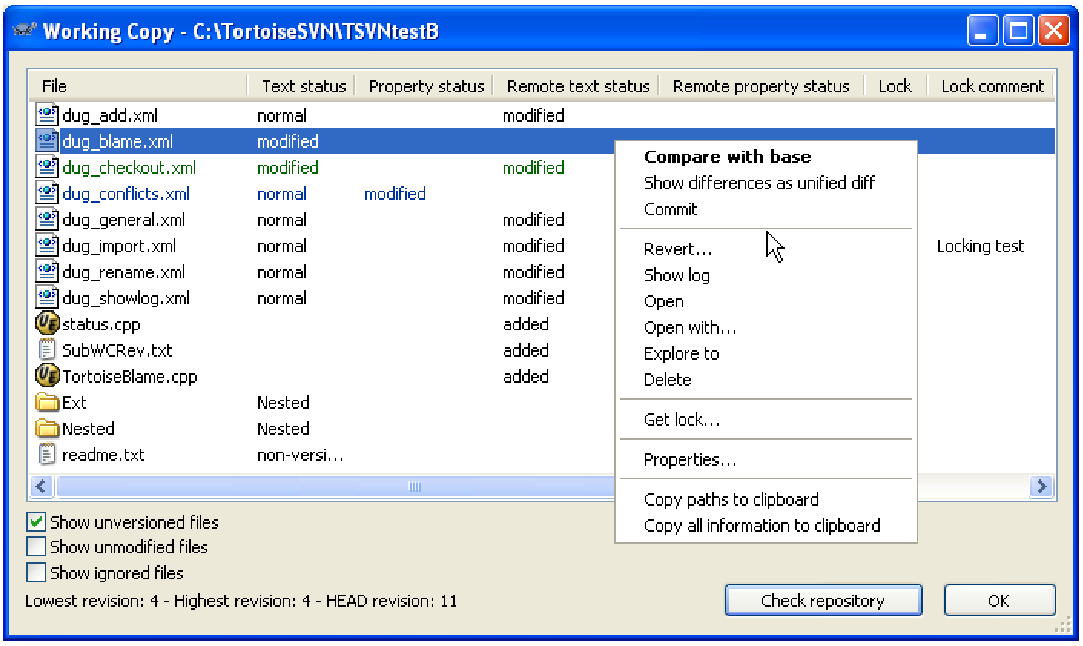
If you select a folder and then use TortoiseSVN → Get Lock... the lock dialog will open with *every* file in *every* subfolder selected for locking. If you really want to lock an entire hierarchy, that is the way to do it, but you could become very unpopular with your co-workers if you lock them out of the whole project. Use with care ...

5.18.3. 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.

5.18.4. Checking Lock Status



File	Text status	Property status	Remote text status	Remote property status	Lock	Lock comment
dug_add.xml	normal		modified			
dug_blame.xml	modified					
dug_checkout.xml	modified		modified			
dug_conflicts.xml	normal	modified				
dug_general.xml	normal		modified			
dug_import.xml	normal		modified			Locking test
dug_rename.xml	normal		modified			
dug_showlog.xml	normal		modified			
status.cpp			added			
SubWCRev.txt			added			
TortoiseBlame.cpp			added			
Ext	Nested					
Nested	Nested					
readme.txt	non-versi...					

[✓] Show unversioned files
[ ] Show unmodified files
[ ] Show ignored files

Lowest revision: 4 - Highest revision: 4 - HEAD revision: 11

Figure 5.34. 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.

5.18.5. 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 5.15, “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 5.27, “TortoiseSVN's Settings” for further information.

5.18.6. 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.

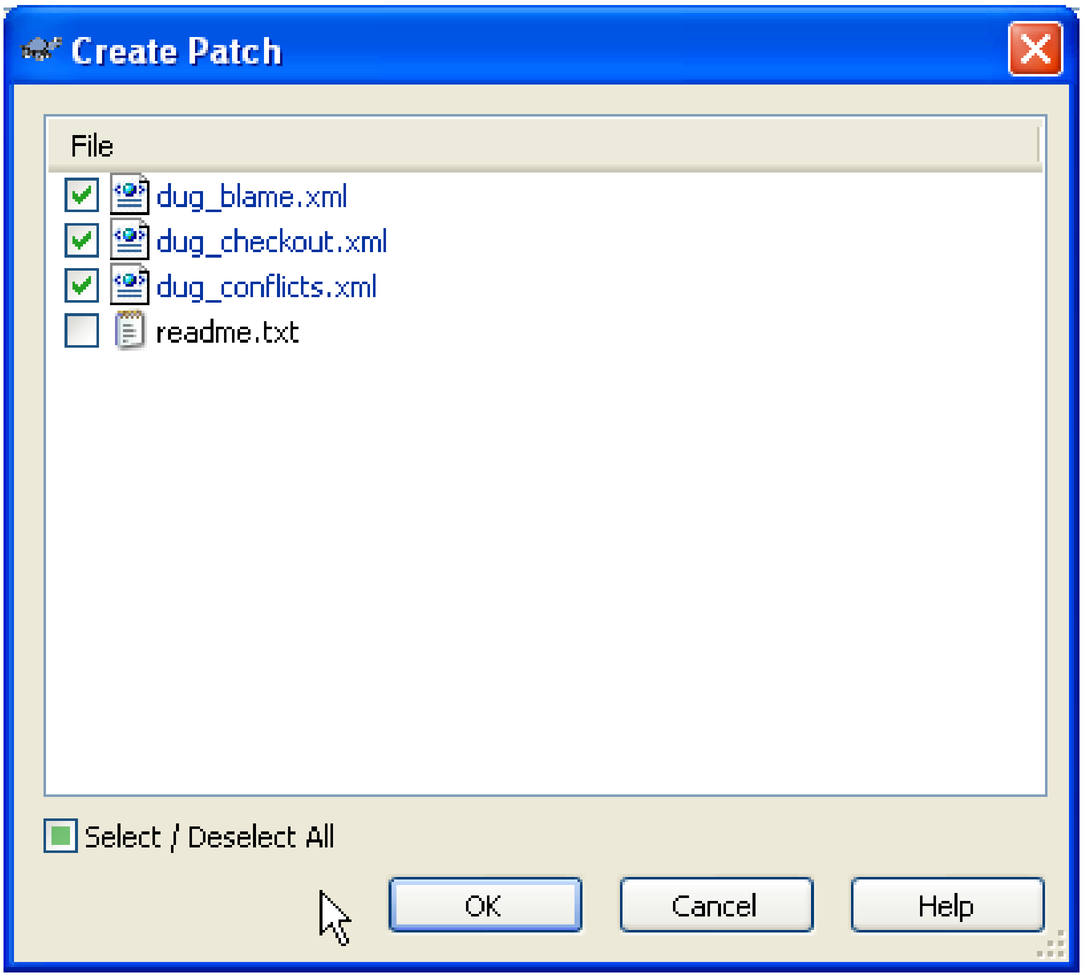
5.19. Creating and Applying Patches

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.

5.19.1. 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...



- [✓] dug_blame.xml
- [✓] dug_checkout.xml
- [✓] dug_conflicts.xml
- [ ] readme.txt

Figure 5.35. 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.

5.19.2. 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 patchfile is `Index: english/chapter1.xml` then you need to apply the patch to the `english` folder. However, provided you are in the correct working copy, if you pick the wrong folder level, TSVN 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.

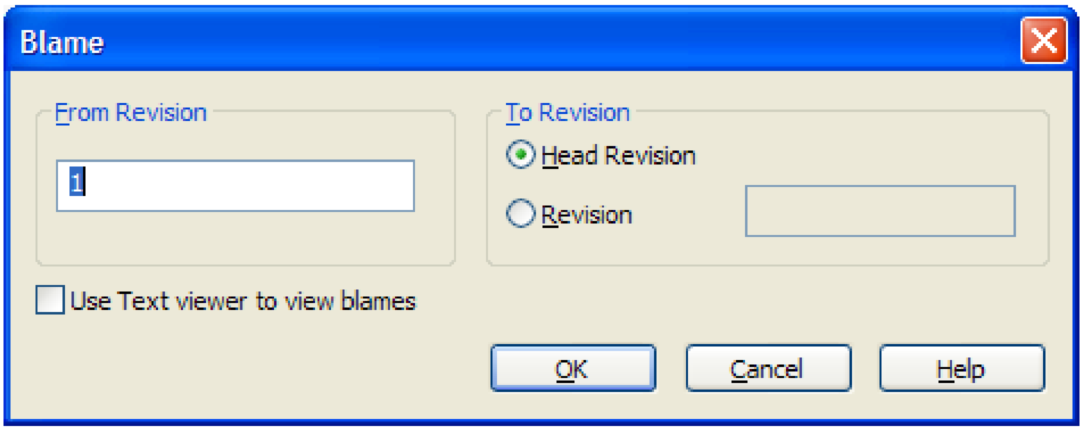
The remote developer's patch has now been applied to your working copy, so you need to commit to allow everyone else to access the changes from the repository.

5.20. 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.

5.20.1. Blame for Files



From Revision: 1

To Revision: [✓ Head Revision] Revision

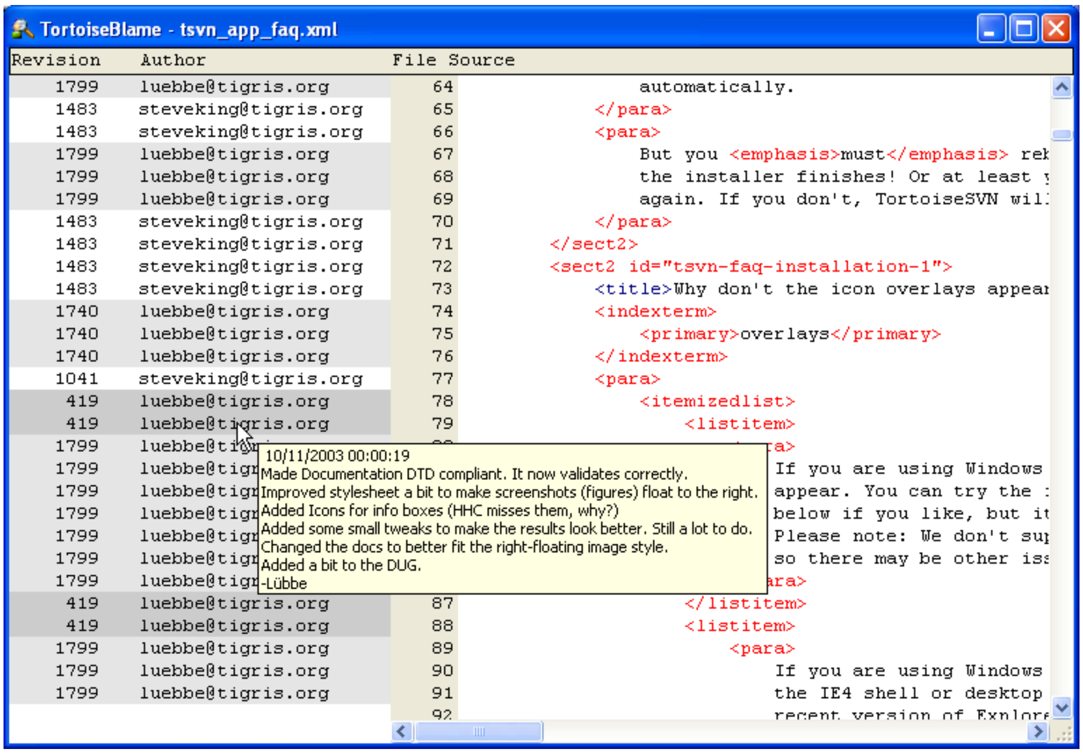
☐ Use Text viewer to view blames

Figure 5.36. 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

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.



```
Revision    Author                File Source
    1799    luebbe@tigris.org      64           automatically.
    1483    steveking@tigris.org   65       </para>
    1483    steveking@tigris.org   66       <para>
    1799    luebbe@tigris.org      67           But you <emphasis>must</emphasis> rek
    1799    luebbe@tigris.org      68           the installer finishes! Or at least y
    1799    luebbe@tigris.org      69           again. If you don't, TortoiseSVN wil
    1483    steveking@tigris.org   70       </para>
    1483    steveking@tigris.org   71   </sect2>
    1483    steveking@tigris.org   72   <sect2 id="tsvn-faq-installation-1">
    1483    steveking@tigris.org   73       <title>Why don't the icon overlays appear
    1740    luebbe@tigris.org      74       <indexterm>
    1740    luebbe@tigris.org      75           <primary>overlays</primary>
    1740    luebbe@tigris.org      76       </indexterm>
    1041    steveking@tigris.org   77       <para>
     419    luebbe@tigris.org      78           <itemizedlist>
     419    luebbe@tigris.org      79               <listitem>
    1799    luebbe@ti                                       a>
    1799    luebbe@tigr                                 If you are using Windows
    1799    luebbe@tigr                                 appear. You can try the
    1799    luebbe@tigr                                 below if you like, but it
    1799    luebbe@tigr                                 Please note: We don't su
    1799    luebbe@tigr                                 so there may be other is
    1799    luebbe@tigr                                 ra>
     419    luebbe@tigris.org      87               </listitem>
     419    luebbe@tigris.org      88               <listitem>
    1799    luebbe@tigris.org      89                   <para>
    1799    luebbe@tigris.org      90                   If you are using Windows
    1799    luebbe@tigris.org      91                   the IE4 shell or desktop
                                   92                   recent version of Explore
```

```
10/11/2003 00:00:19
Made Documentation DTD compliant. It now validates correctly.
Improved stylesheet a bit to make screenshots (figures) float to the right.
Added Icons for info boxes (HHC misses them, why?)
Added some small tweaks to make the results look better. Still a lot to do.
Changed the docs to better fit the right-floating image style.
Added a bit to the DUG.
-Lübbe
```

Figure 5.37. 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....

5.20.2. 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.

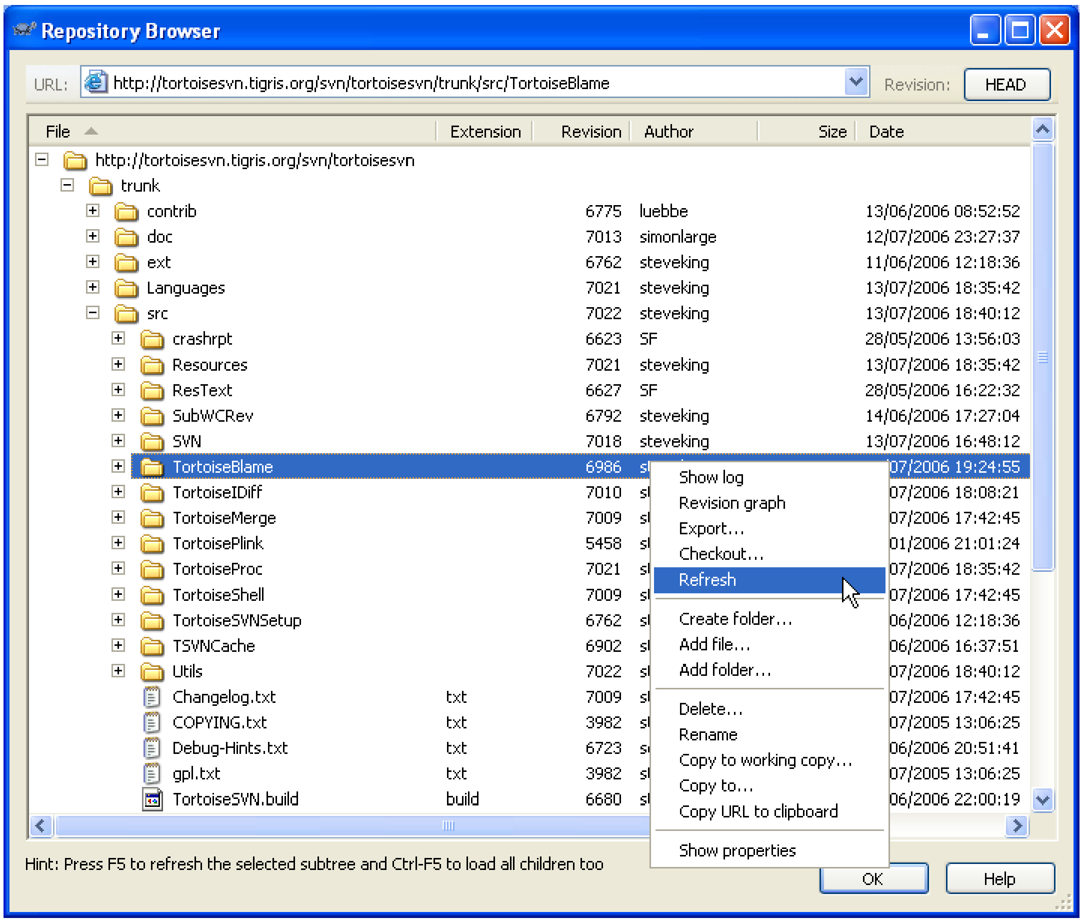
Blame Revisions
: In the top pane, select 2 revisions, then select Context menu → Blame revisions. This will fetch the blame data for the 2 revisions, then use the diff viewer to compare the two blame files.

Blame Differences
: Select one revision in the top pane, then pick one file in the bottom pane and select Context menu → Blame differences. 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.

5.21. 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.



URL: http://tortoisesvn.tigris.org/svn/tortoisesvn/trunk/src/TortoiseBlame

File	Extension	Revision	Author	Size	Date
http://tortoisesvn.tigris.org/svn/tortoisesvn					
trunk					
contrib		6775	luebbe		13/06/2006 08:52:52
doc		7013	simonlarge		12/07/2006 23:27:37
ext		6762	steveking		11/06/2006 12:18:36
Languages		7021	steveking		13/07/2006 18:35:42
src		7022	steveking		13/07/2006 18:40:12
crashrpt		6623	SF		28/05/2006 13:56:03
Resources		7021	steveking		13/07/2006 18:35:42
ResText		6627	SF		28/05/2006 16:22:32
SubWCRev		6792	steveking		14/06/2006 17:27:04
SVN		7018	steveking		13/07/2006 16:48:12
TortoiseBlame		6986	st		07/2006 19:24:55
TortoiseIDiff		7010	st		07/2006 18:08:21
TortoiseMerge		7009	st		07/2006 17:42:45
TortoisePlink		5458	st		01/2006 21:01:24
TortoiseProc		7021	st		07/2006 18:35:42
TortoiseShell		7009	st		07/2006 17:42:45
TortoiseSVNSetup		6762	st		06/2006 12:18:36
TSVNCache		6902	st		06/2006 16:37:51
Utils		7022	st		07/2006 18:40:12
Changelog.txt	txt	7009	st		07/2006 17:42:45
COPYING.txt	txt	3982	st		07/2005 13:06:25
Debug-Hints.txt	txt	6723	s		06/2006 20:51:41
gpl.txt	txt	3982	st		07/2005 13:06:25
TortoiseSVN.build	build	6680	st		06/2006 22:00:19

Hint: Press F5 to refresh the selected subtree and Ctrl-F5 to load all children too

Figure 5.38. The Repository Browser

With the Repository Browser you can execute commands like copy, move, rename, ... directly on the repository.

At the top of the Repository Browser Window you can enter the URL of the repository and the revision you want to browse. Browsing an older revision is useful if you want to e.g. recover a previously deleted file. Use the Context Menu → Copy To... command to do that and enter the full working copy path to where you want to recover your deleted file.

The main browser window looks much like any other tree browser. You can click on the column headings at the top to change the sort order. If you right click on the `File` heading you can turn `Numerical Sort` on or off. This provides a more intelligent handling of numbers than a plain ascii sort does, and is useful for getting version number tags in the correct order. The default state of this setting can be changed in the settings dialog.

If you select two items, 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. This can be particularly useful for comparing two tags to see what changed.

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 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.

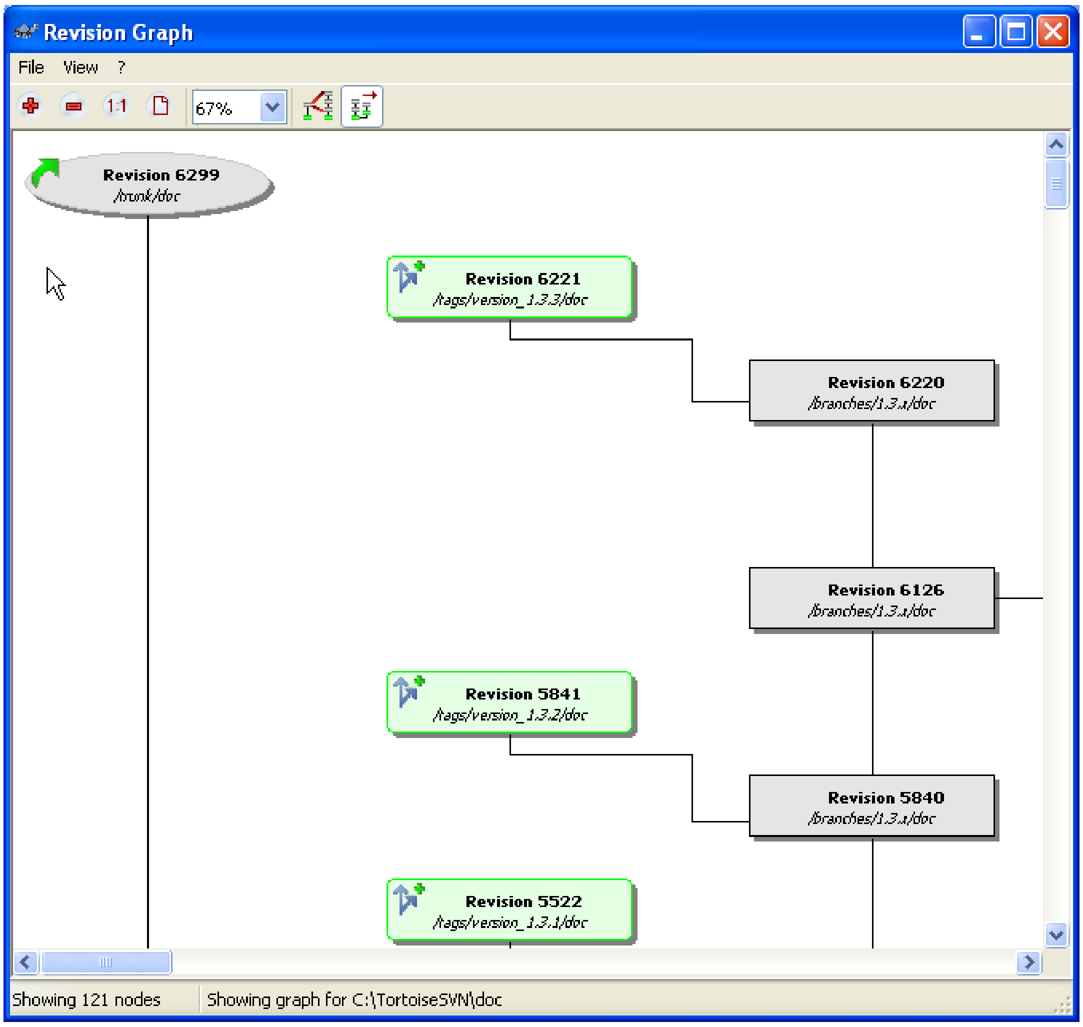
If you want to move an item within the repository, just left drag it to the new location. If you want to create a copy rather than moving the item, Ctrl left drag instead. When copying, the cursor has a “plus” symbol on it, just as it does in Explorer.

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.

5.22. Revision Graphs



Showing 121 nodes

Showing graph for C:\TortoiseSVN\doc

Figure 5.39. 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 300,000 revisions you could be waiting for some time.

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. This means that the latest revision on every branch is always shown on the graph.

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.

There is also an option to arrange the graph *by path*. This attempts to sort the branches from the tags. Paths which contain no modifications after copying are assumed to be tags and are stacked in a single column. Branches (which contain modifications after creation) each have their own column, so you can see how the branch develops.

Sometimes the revision graph contains more detail than you want to see. View → Filter opens a dialog which allows you to restrict the range of revisions displayed, and to hide particular paths by name.

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.

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.

5.23. 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.

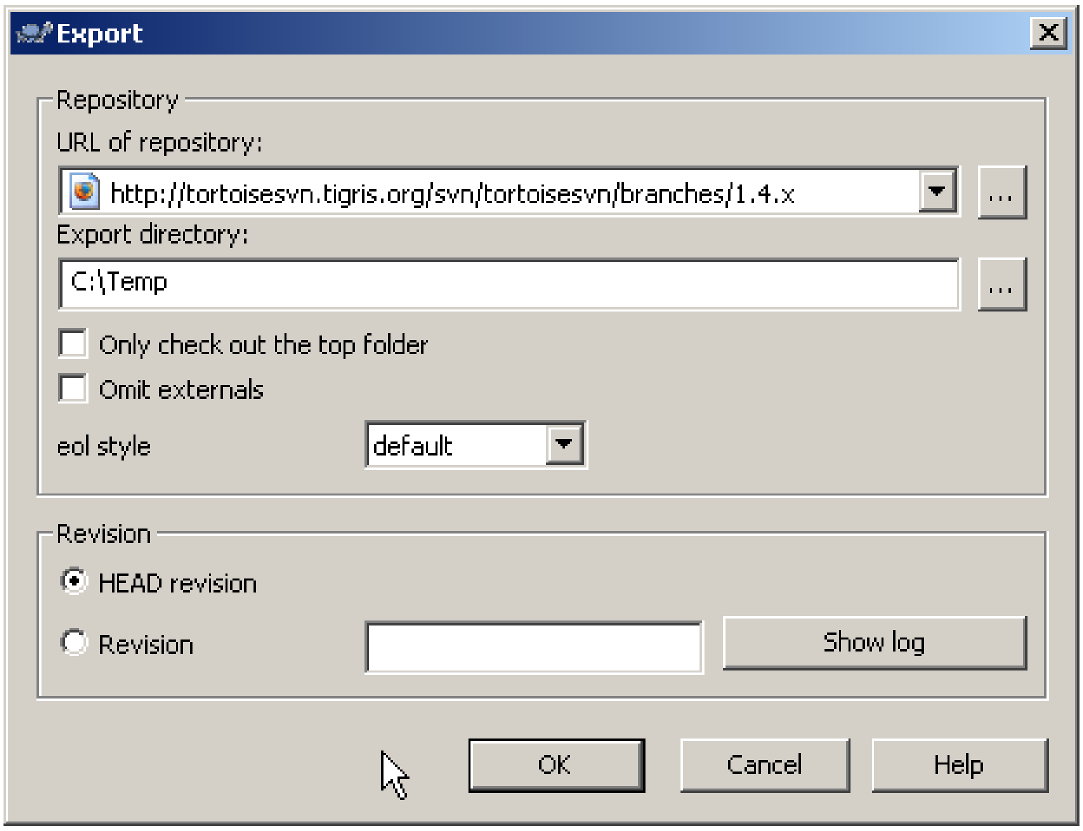


Figure 5.40. The Export-from-URL Dialog

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)`.

5.24. Relocating a working copy

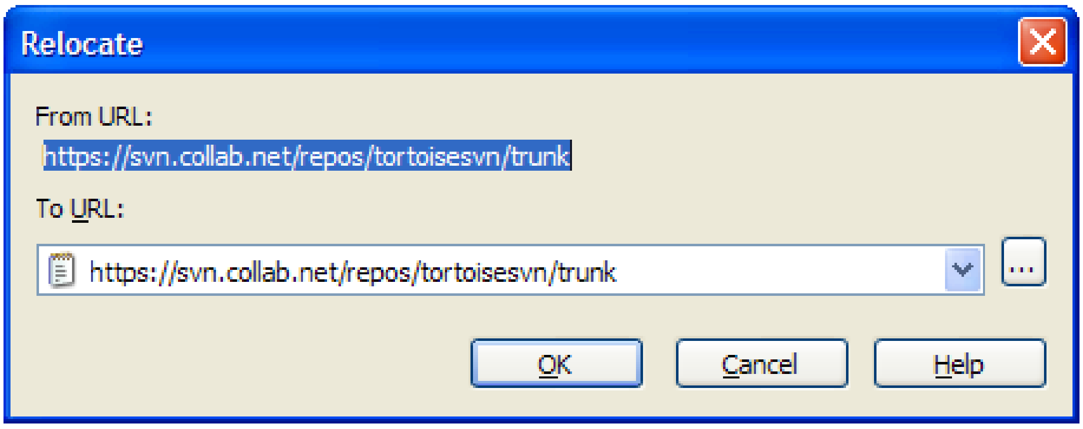

Relocate
From URL:
https://svn.collab.net/repos/tortoisesvn/trunk
To URL:
https://svn.collab.net/repos/tortoisesvn/trunk
...
OK
Cancel
Help

Figure 5.41. 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 5.16.2, “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.

5.25. Integration with Bugtracking 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.

You can integrate a Bugtracking 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 5.15, “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 bugtracking tool. It must be properly URI encoded and it has to contain `%BUGID%`. `%BUGID%` is replaced with the Issuenumber 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 bugtracking tool. You do not have to provide this property, but then TortoiseSVN shows only the issuenumber and not the link to it. e.g the TortoiseSVN project is using `http://issues.tortoisesvn.net/?do=details&id=%BUGID%`

bugtraq:warnifnoissue
: Set this to `true`, if you want TortoiseSVN to warn you because of an empty issuenumber text-field. Valid values are `true/false`. *If not defined, `false` is assumed.*

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 Bugtracking 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 logmessage. It must contain `%BUGID%`, which is replaced with the issuenumber on commit. This ensures that your commit log contains a reference to the issuenumber which is always in a consistent format and can be parsed by your Bugtracking tool to associate the issuenumber with a particular commit. e.g the TortoiseSVN project is using `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 issuenumber. 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 issuenumber textfield. An exception is the

comma, so you can comma separate several numbers. Valid values are `true/false`. *If not defined, `true` is assumed.*

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 Bugtracking System in `Regex` mode. It contains one or two regular expressions, separated by a newline.

If only one expression is set, then the bare bug ID's must be matched in the groups of the regex string. Example: `[Ii]ssue(?:s)? #?(\d+)`

If two expressions are set, then the first expression is used to find a string which relates to the bug ID but may contain more than just the bug ID (e.g. "Issue #123" or "resolves issue 123"). The second expression is then used to extract the bare bug ID from the string extracted with the first expression. An example:

If you want to catch every pattern "issue #XXX" and "issue #890, #789" inside a log message you could use the following regex strings: `[Ii]ssue #?(\d+)(,? ?#(\d+))*` and the second expression as `(\d+)`

If you are unfamiliar with regular expressions, take a look at 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!

Some `tsvn:` properties require a `true/false` value. TSVN 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 subfolder, 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 subfolder. 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 subfolders 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 *Issuetracker Integration Specification* [http://tortoisesvn.tigris.org/svn/tortoisesvn/trunk/doc/issuetrackers.txt].

5.26. 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 repoviewer 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 5.15, “Project Settings”)

webviewer:revision
: Set this property to the url of your repoviewer 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 repoviewer 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`.

Set the Properties on Folders

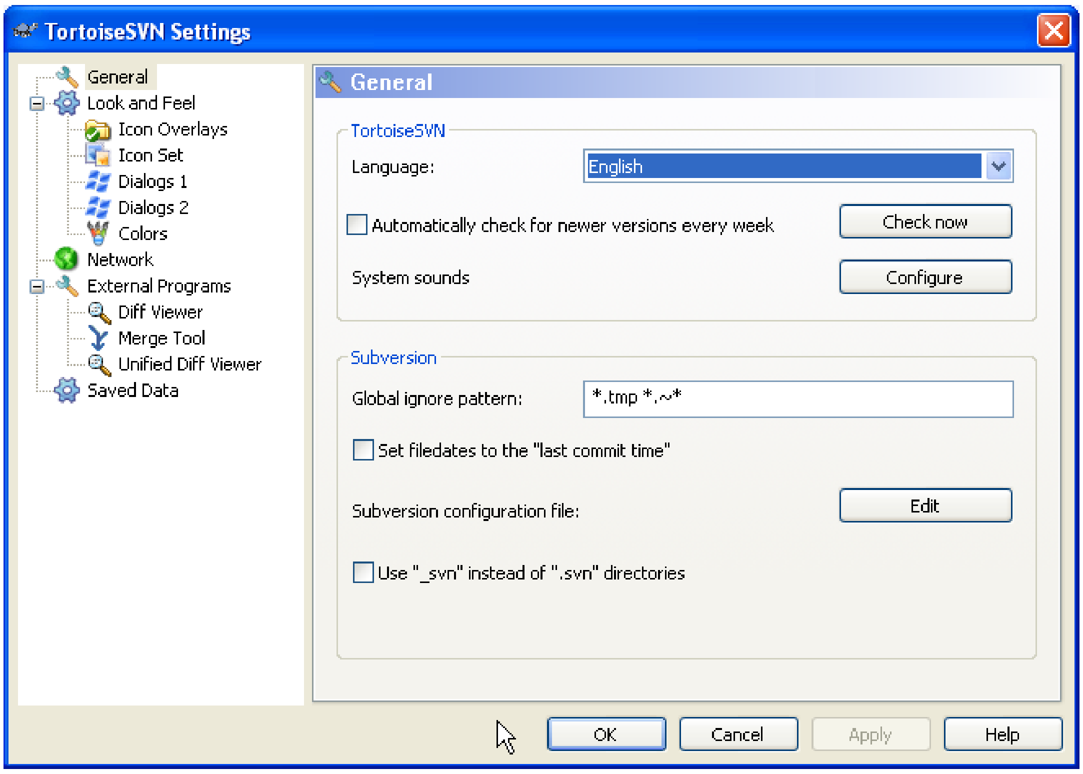
These properties must be set on folders for the system to work. When you commit a file or folder the properties are read from that folder. If the properties are not found there, TortoiseSVN will search upwards through the folder tree to find them until it comes to an unversioned folder, or the tree root (eg. `C:\`) is found. If you can be sure that each user checks out only from e.g `trunk/` and not some subfolder, 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 subfolder. 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 subfolders in the hierarchy, without also setting it on all files.

5.27. 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.

5.27.1. General Settings



General

TortoiseSVN

Language: English

☐ Automatically check for newer versions every week

System sounds

Subversion

Global ignore pattern: *.tmp *.~*

☐ Set filedates to the "last commit time"

Subversion configuration file:

☐ Use "_svn" instead of ".svn" directories

Figure 5.42. 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`. The first two entries refer to directories, the other four to files. These patterns use filename globbing. Read Section 5.11.1, “Filename Globbing in Ignore Lists” for more information.

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 5.11, “Ignoring Files And Directories” for more information.

Set filedates to the "last commit time"

This option tells TortoiseSVN to set the filedates 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 datestamps 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/1.2/svn.advanced.html]. The section on *Automatic Property Setting* [http://svnbook.red-bean.com/en/1.2/svn.advanced.props.html] 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/1.2/svn.advanced.html] to find out more.

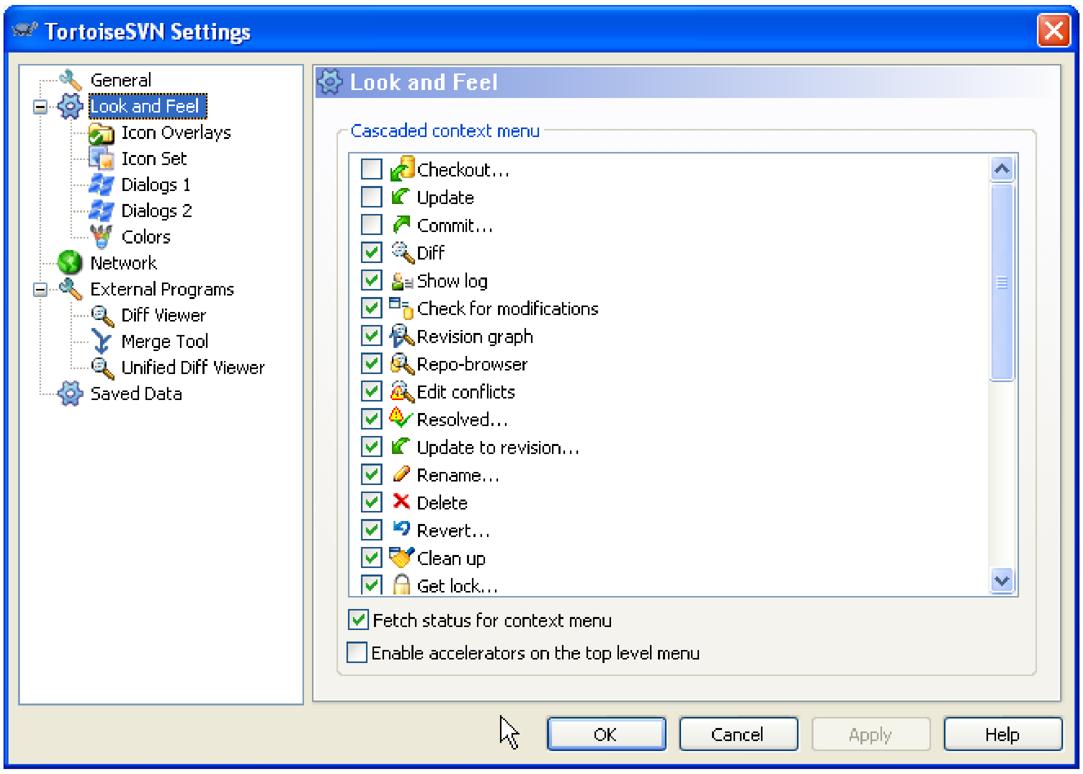
Use "_svn" instead of ".svn" directories

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 5.27.7, “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.

5.27.2. Look and Feel Settings



Look and Feel

Cascaded context menu

- [ ] Checkout...
- [ ] Update
- [ ] Commit...
- [x] Diff
- [x] Show log
- [x] Check for modifications
- [x] Revision graph
- [x] Repo-browser
- [x] Edit conflicts
- [x] Resolved...
- [x] Update to revision...
- [x] Rename...
- [x] Delete
- [x] Revert...
- [x] Clean up
- [x] Get lock...

- [x] Fetch status for context menu
- [ ] Enable accelerators on the top level menu

Figure 5.43. The Settings Dialog, Look and Feel 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.

If you have a very large number of files in your working copy folders, it can take a long time before the context menu appears when you right click on a folder. This is because Subversion fetches the status for all files when you ask for folder status. To avoid this delay you can uncheck the Fetch status for context menu box. Be warned that the context menu for folders will not always be correct, and may include items which should not really be there. For example, you will see TortoiseSVN → Show Log for an Added folder, which will not work because the folder is not yet in the repository.

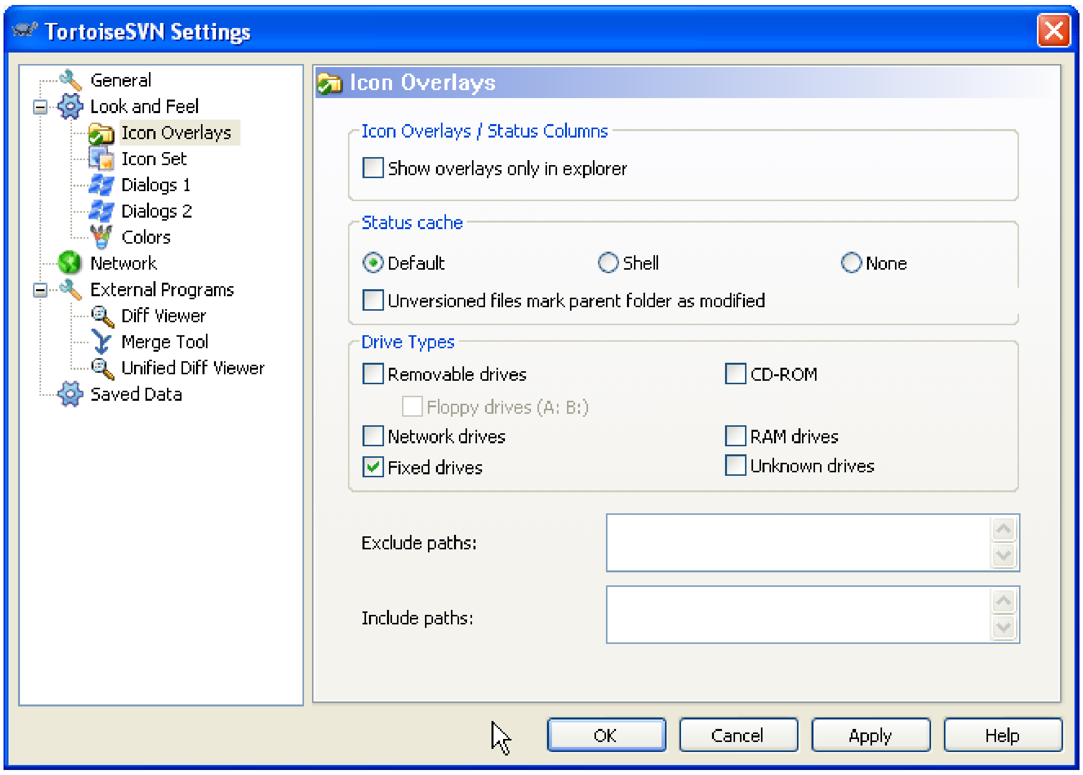
The option Enable accelerators on the top level menu has three states:

Unchecked (default)
: In this state the menu items are all drawn by TortoiseSVN. No accelerator keys are shown.

Checked
: This activates the accelerators for TortoiseSVN commands, but of course these may conflict with the accelerators for anything else in the explorer context menu. Pressing the shortcut key multiple times will cycle through the matching context menu items. In this state, the menu items are drawn by Windows which makes the icons look ugly.

Indeterminate
: In this mode the accelerator keys are active and the menu items are drawn in text only mode without icons.

5.27.2.1. Icon Overlay Settings


TortoiseSVN Settings

- General
- Look and Feel
  - Icon Overlays
  - Icon Set
  - Dialogs 1
  - Dialogs 2
  - Colors
- Network
- External Programs
  - Diff Viewer
  - Merge Tool
  - Unified Diff Viewer
- Saved Data

Icon Overlays

Icon Overlays / Status Columns
☐ Show overlays only in explorer

Status cache
[✓ Default] Shell None
☐ Unversioned files mark parent folder as modified

Drive Types
☐ Removable drives ☐ CD-ROM
☐ Floppy drives (A: B:)
☐ Network drives ☐ RAM drives
☑ Fixed drives ☐ Unknown drives

Exclude paths:

Include paths:

Figure 5.44. The Settings Dialog, Look and Feel Page

This page allows you to choose the items for which TortoiseSVN will display icon overlays. Network drives can be very slow, so by default icons are not shown for working copies located on network shares. You can even disable all icon overlays, but where's the fun in that?

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.

By default, overlay icons 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 only in explorer box.

Since it takes quite a while to fetch the status of a working copy, TortoiseSVN uses a cache to store the status in 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 *realtime* 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 treeview 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 *realtime*.

  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.

If you select the default option, 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.

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:

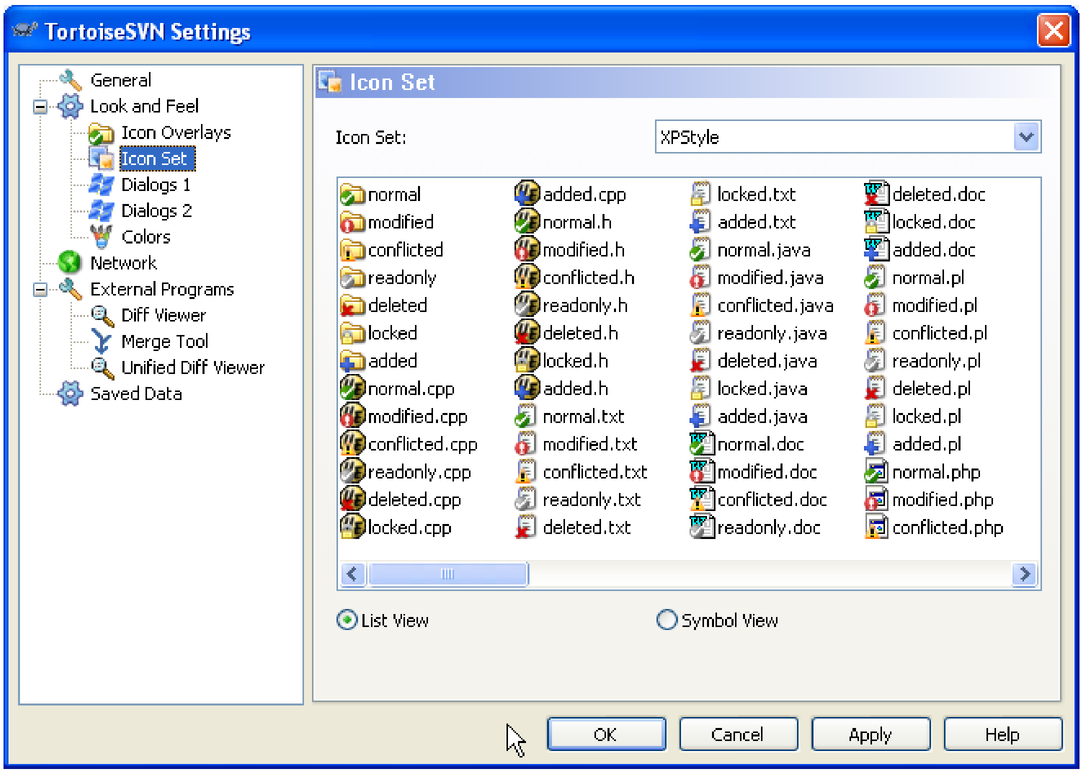
`f:\development\SVN\Subversion` will disable the overlays *only* on that specific folder. You still can see the overlays on all files and folder inside that folder.

`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.

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.

5.27.2.2. Icon Set Selection



Icon Set: XPStyle

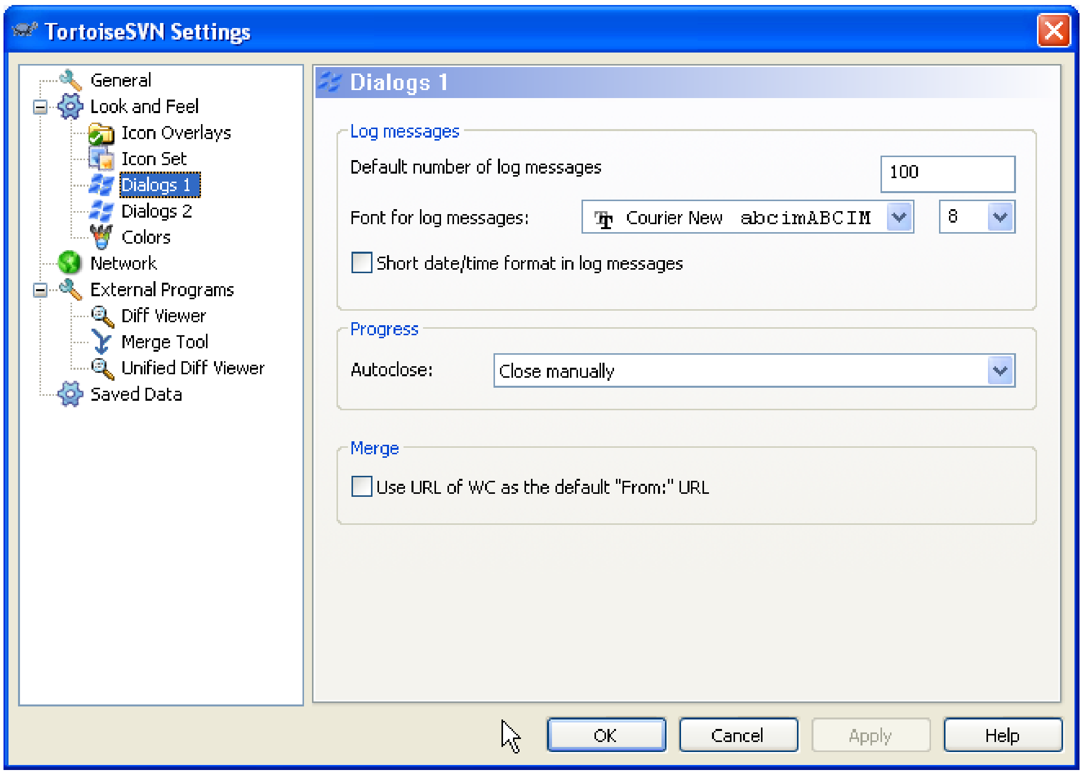
normal	added.cpp	locked.txt	deleted.doc
modified	normal.h	added.txt	locked.doc
conflicted	modified.h	normal.java	added.doc
readonly	conflicted.h	modified.java	normal.pl
deleted	readonly.h	conflicted.java	modified.pl
locked	deleted.h	readonly.java	conflicted.pl
added	locked.h	deleted.java	readonly.pl
normal.cpp	added.h	locked.java	deleted.pl
modified.cpp	normal.txt	added.java	locked.pl
conflicted.cpp	modified.txt	normal.doc	added.pl
readonly.cpp	conflicted.txt	modified.doc	normal.php
deleted.cpp	readonly.txt	conflicted.doc	modified.php
locked.cpp	deleted.txt	readonly.doc	conflicted.php

[✓ List View] Symbol View

Figure 5.45. 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.

5.27.2.3. TortoiseSVN Dialog Settings 1



Log messages

Default number of log messages: 100

Font for log messages: Courier New abcimABCIM 8

☐ Short date/time format in log messages

Progress

Autoclose: Close manually

Merge

☐ Use URL of WC as the default "From:" URL

Figure 5.46. 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 Get 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.

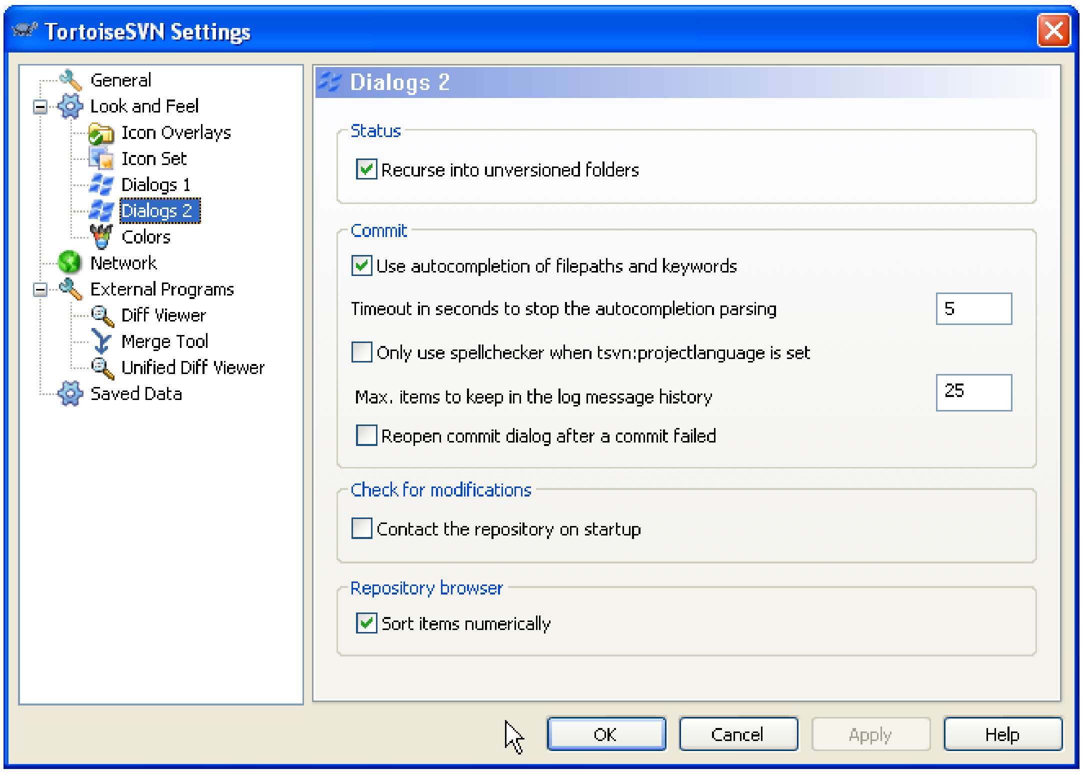
  Auto-close if no errors always closes the dialog even if there were conflicts. The only condition that keeps the dialog open is an error condition, which occurs when Subversion is unable to complete the task. For example, an update fails because the server is inaccessible, or a commit fails because the 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.

5.27.2.4. TortoiseSVN Dialog Settings 2



Status

✓ Recurse into unversioned folders

Commit

✓ Use autocompletion of filepaths and keywords

Timeout in seconds to stop the autocompletion parsing: 5

☐ Only use spellchecker when tsvn:projectlanguage is set

Max. items to keep in the log message history: 25

☐ Reopen commit dialog after a commit failed

Check for modifications

☐ Contact the repository on startup

Repository browser

✓ Sort items numerically

Figure 5.47. 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 autocompletion of filepaths 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 autocompletion 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 autocompletion parsing
: The autocompletion 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 autocompletion 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.

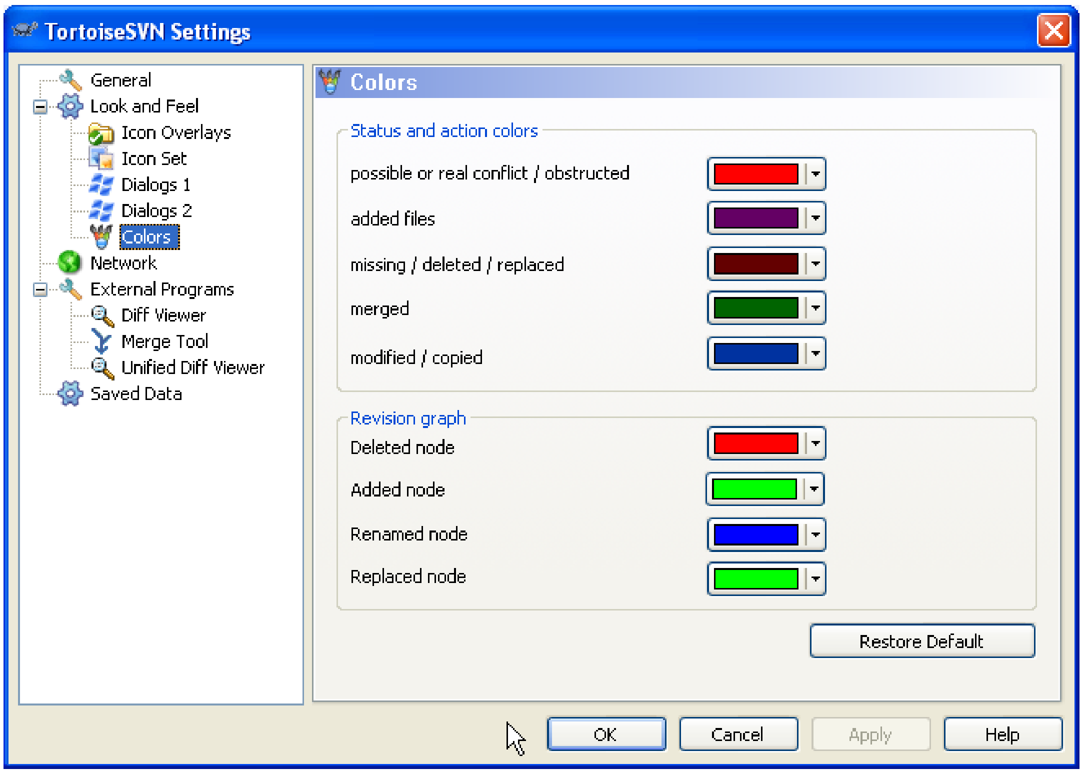
Re-open commit dialog after a commit failed
: When a commit fails for some reason (working copy needs updating, pre-commit hook rejects commit, network error, etc), you can select this option to keep the commit dialog open ready to try again. However, you should be aware that this can lead to problems. If the failure means you need to update your working copy, and that update leads to conflicts you must resolve those

first.

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.

Sort items numerically
: The repository browser can use a more intelligent sorting algorithm which handles paths containing numbers better than a plain ascii sort. This is sometimes useful for getting version number tags in the correct order. This option controls the default sort type used.

5.27.2.5. TortoiseSVN Colour Settings



Colors

Status and action colors

possible or real conflict / obstructed	🟥
added files	🟪
missing / deleted / replaced	🟫
merged	🟩
modified / copied	🟦

Revision graph

Deleted node	🟥
Added node	🟩
Renamed node	🟦
Replaced node	🟩

Figure 5.48. 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.

5.27.3. Network Settings

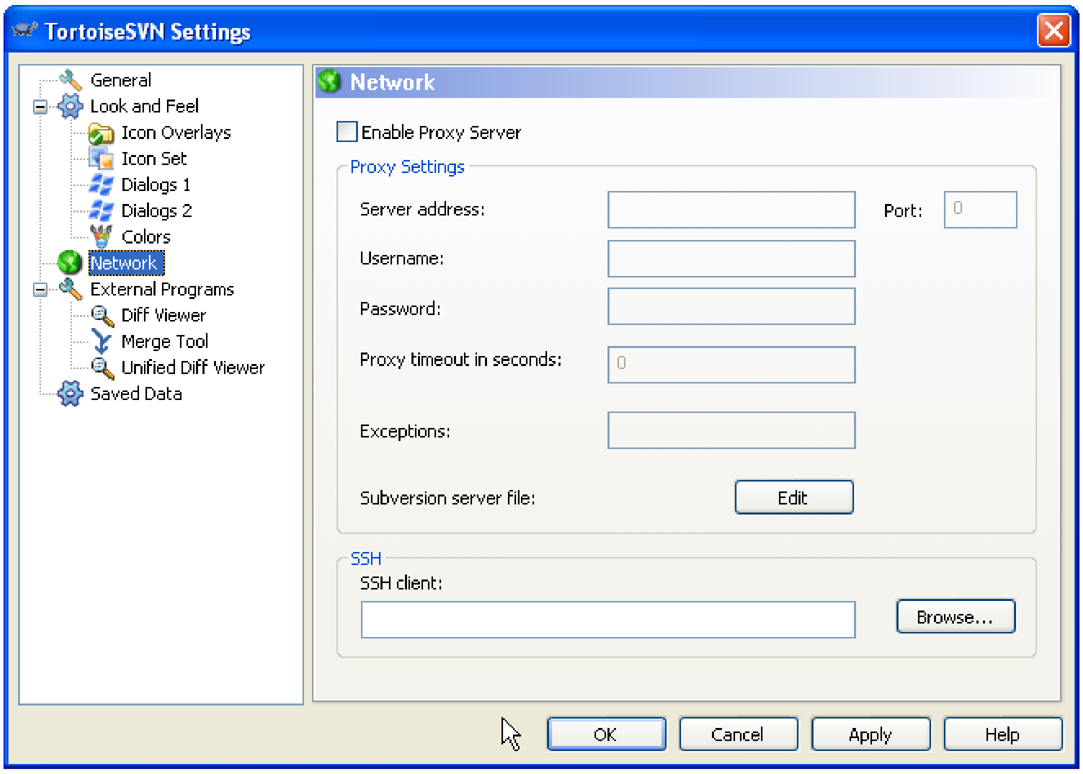


Figure 5.49. The Settings Dialog, Network Page

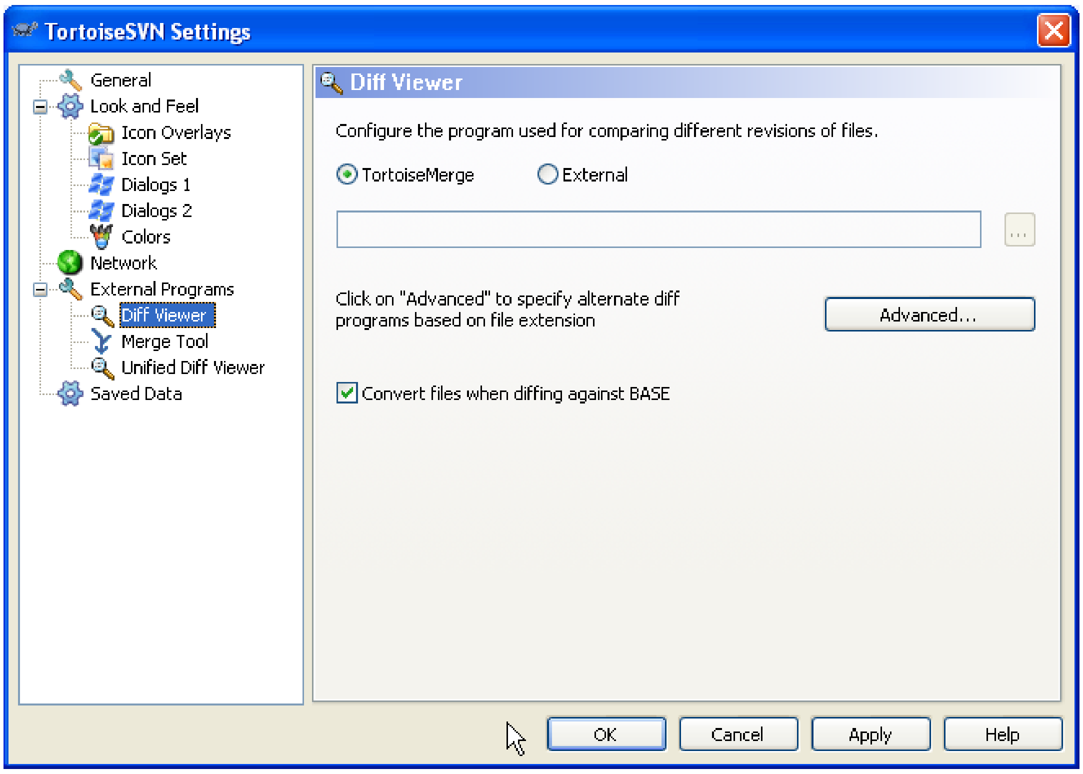
Here you can configure your proxy server, if you need one to get through your company's firewall.

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/1.2/svn.advanced.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.

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.

5.27.4. External Program Settings



Diff Viewer

Configure the program used for comparing different revisions of files.

[✓ TortoiseMerge] External

Click on "Advanced" to specify alternate diff programs based on file extension

Advanced...

[✓] Convert files when diffing against BASE

Figure 5.50. 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 5.9.4, "External Diff/Merge Tools" for a list of some of the external diff/merge programs that people are using with TortoiseSVN.

5.27.4.1. 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. TSVN 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 TSVN will skip pre-processing the files.

5.27.4.2. 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
```

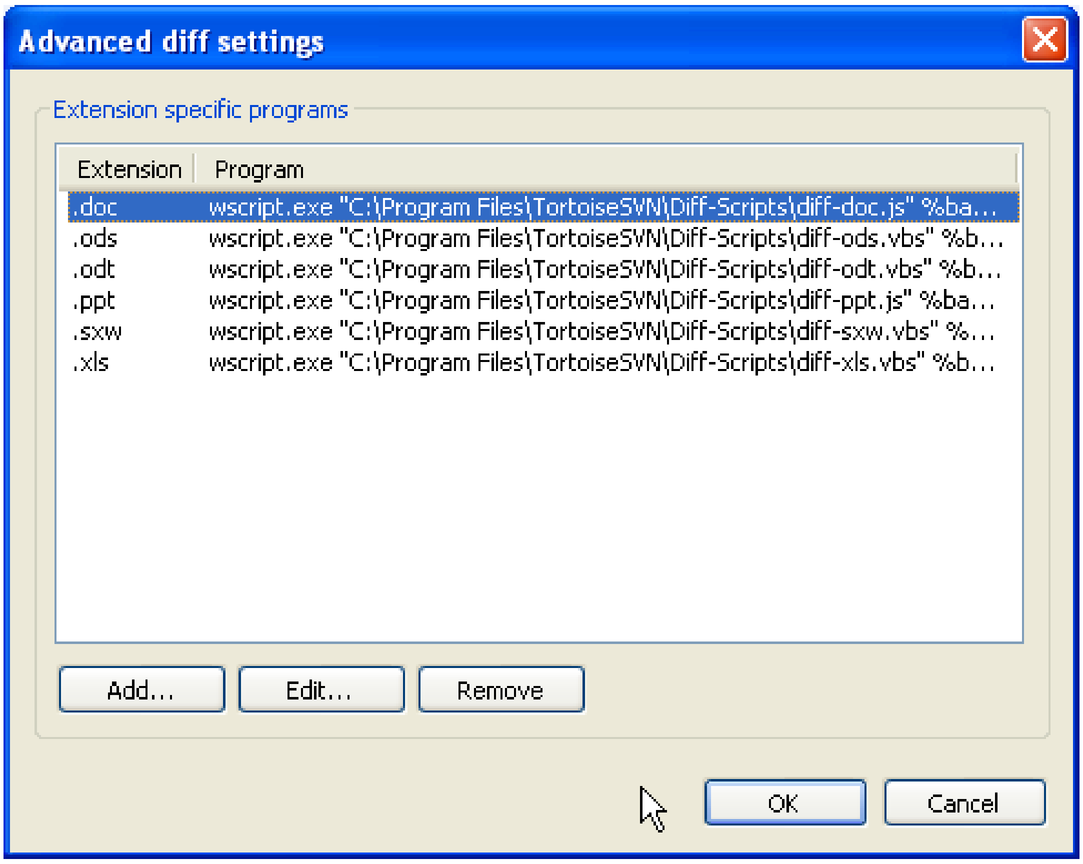
or with KDiff3:

```
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
```

5.27.4.3. Diff/Merge Advanced Settings



Extension	Program
.doc	wscript.exe "C:\Program Files\TortoiseSVN\Diff-Scripts\diff-doc.js" %ba...
.ods	wscript.exe "C:\Program Files\TortoiseSVN\Diff-Scripts\diff-ods.vbs" %b...
.odt	wscript.exe "C:\Program Files\TortoiseSVN\Diff-Scripts\diff-odt.vbs" %b...
.ppt	wscript.exe "C:\Program Files\TortoiseSVN\Diff-Scripts\diff-ppt.js" %ba...
.sxw	wscript.exe "C:\Program Files\TortoiseSVN\Diff-Scripts\diff-sxw.vbs" %...
.xls	wscript.exe "C:\Program Files\TortoiseSVN\Diff-Scripts\diff-xls.vbs" %b...

Figure 5.51. 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 just the extension, with the leading dot but with no wildcard spec. Use `.BMP` to describe Windows bitmap files, *not* `*.BMP`. To associate using the `svn:mime-type` property, specify the mime type, including a slash, for example `text/xml`.

5.27.4.4. 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.

5.27.5. Saved Data Settings

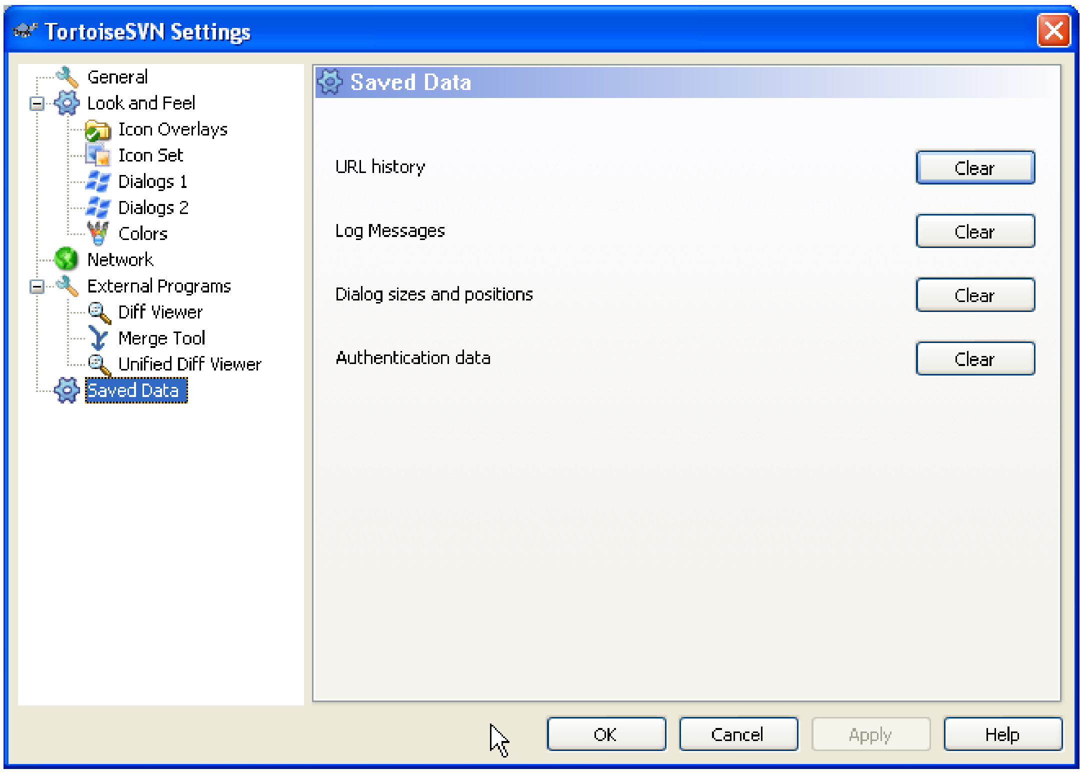


Figure 5.52. The Settings Dialog, Saved Data Page

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+DELETE.

Log messages
: 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.

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 auth data for one particular server only, read Section 5.1.5, “Authentication” for instructions on how to find the cached data.

5.27.6. 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.

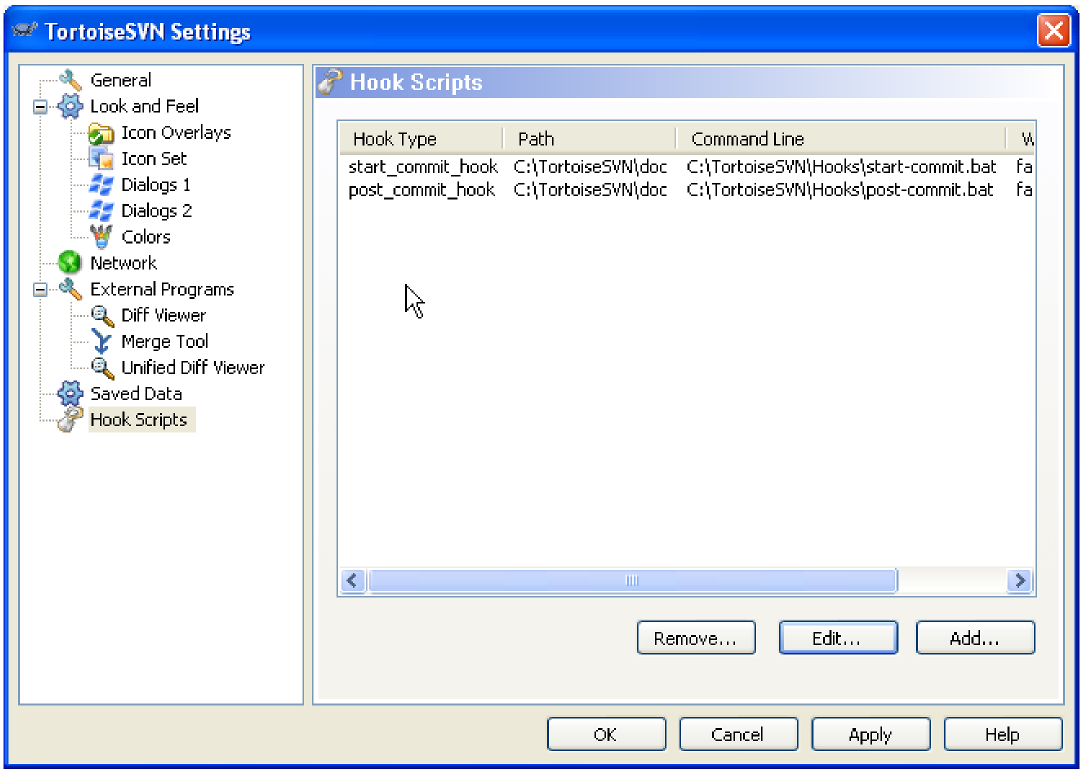
5.27.7. Subversion Working Folders

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.

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 env 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 5.27.1, “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/node/aspdotnethack].

5.27.8. Hook Scripts



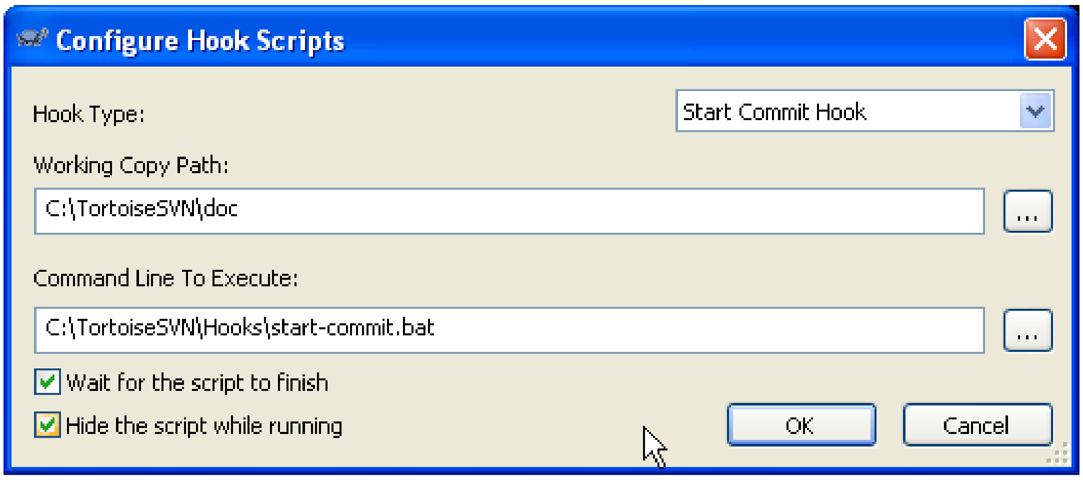
Hook Type	Path	Command Line	W
start_commit_hook	C:\TortoiseSVN\doc	C:\TortoiseSVN\Hooks\start-commit.bat	fa
post_commit_hook	C:\TortoiseSVN\doc	C:\TortoiseSVN\Hooks\post-commit.bat	fa

Figure 5.53. 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.

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.



Hook Type: Start Commit Hook

Working Copy Path: C:\TortoiseSVN\doc

Command Line To Execute: C:\TortoiseSVN\Hooks\start-commit.bat

✓ Wait for the script to finish

✓ Hide the script while running

Figure 5.54. 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.

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 can include several parameters which get filled in by TortoiseSVN. The parameters available depend upon which hook is called.

Start-commit
: `%PATHS%`

Pre-commit
: `%PATHS%` `%SELECTEDPATHS%`

Post-commit
: `%SELECTEDPATHS%` `%REVISION%` `%ERROR%`

Start-update
: `%PATHS%`

Pre-update
: `%PATHS%`

Post-update
: `%PATHS%` `%REVISION%` `%ERROR%`

The meaning of each of these variables is described here:

%PATHS%
: The paths which were selected when the operation was started, eg. the paths selected in Explorer when invoking the Commit dialog. If multiple paths were selected they are separated by a `*` character.

%SELECTEDPATHS%
: The paths which were selected within the Commit dialog. If multiple paths were selected they

are separated by a `*` character.

%REVISION%
: The repository revision after a commit completes.

%ERROR%
: Empty if the operation was successful, or the error message if the operation was unsuccessful.

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. For debugging, you may want to watch what happens in the DOS window.

5.28. Final Step

Wishlist

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 CD's and DVD's: *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.

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. To avoid confusion, international format is used, ie. `yyyy-mm-dd hh:mm:ss`
$WCNOW$	Replaced with the current system date/time. This can be used to indicate the build time. International format is used 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
$WCMIXED?TText:FText$	Replaced with TText if there are mixed update revisions, or FText if not.
$WCMODS?TText:FText$	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.

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

char *Revision = "$WCREV$";
char *Modified = "$WCMODS?Modified:Not modified$";
char *Date     = "$WCDATE$";
char *RevRange = "$WCRANGE$";
char *Mixed    = "$WCMIXED?Mixed revision WC:Not mixed$";
char *URL      = "$WCURL$";

#if $WCMODS?1:0$
#error Source is modified
#endif

// EndOfFile
```

After running SubWCRev.exe, the output file looks like this:

```
// Test file for SubWCRev

char *Revision = "3701";
char *Modified = "Modified";
char *Date     = "2005/06/15 11:15:12";
char *RevRange = "3699:3701";
char *Mixed    = "Mixed revision WC";
char *URL      = "http://tortoisesvn.tigris.org/svn/tortoisesvn/trunk/src/SubWC

#if 1
#error Source is modified
#endif

// EndOfFile
```

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 *interactive online FAQ* [http://tortoisesvn.net/faq] 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 the mailing list.

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 → Copy in Subversion to here. or Context Menu → Move in Subversion to 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 subfolder `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 5.15, “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 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 Get 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(s), 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, ie. 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/1.2/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 subfolder which you want the 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 subfolders 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.

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" /notempfile
```

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. Move the file to somewhere safe, not inside your working copy.
2. TortoiseSVN → Commit the parent folder. TortoiseSVN will see that the file is missing and you can mark it for deletion from the repository.
3. Move the file back to its original location.
4. Add the file to the ignore list so you don't get into the same trouble again.

If you need to remove a whole folder/hierarchy from version control, the procedure is different again.

1. TortoiseSVN → Export the folder to somewhere safe, not inside your working copy.
2. TortoiseSVN → Delete the folder from your working copy.
3. TortoiseSVN → Commit the deleted folder to remove it from the repository.
4. Move the exported folder back to its original location in your working copy.

5. Add the folder to the ignore list so you don't get into the same trouble again.

Appendix C. 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

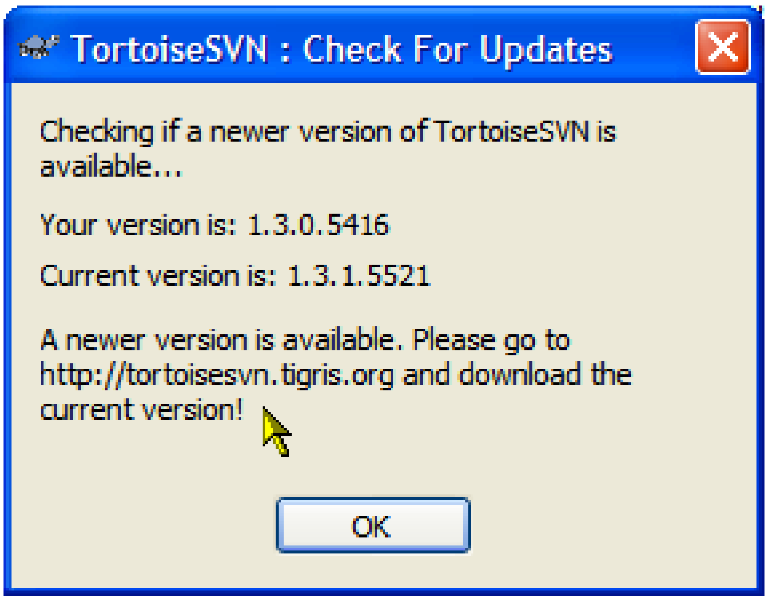
The TortoiseSVN installer comes as an msi file, which means you should have no problems adding that msi file to the group policies of your domain controller.

A good walkthrough 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.



Checking if a newer version of TortoiseSVN is available...

Your version is: 1.3.0.5416

Current version is: 1.3.1.5521

A newer version is available. Please go to http://tortoisesvn.tigris.org and download the current version!

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 textfile 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 webpage instead of the msi file directly. The URL is opened with the default web browser, so if you specify a webpage, 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
```

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.

Since TortoiseSVN uses temporary files to pass multiple arguments between the shell extension and the main program, you *must* add the `/notempfile` parameter! If you don't, the command won't work and the file you pass with the `/path` parameter will be deleted!

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 batchfile or from the TortoiseSVN context menu.

To specify a different location of the configuration file, use the param `/configdir:"path\to\config\dir"`. 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: `/revstart:xxx`, `/revend:xxx` and `/strict`
: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 prefill the bug ID box (in case you've set up integration with bugtrackers 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 dir in `/path`.
:merge	Opens the merge dialog. The path specifies the target directory. Three additional options can be set: `/revstart:xxx`, `/revend:xxx` and `/mergefrom:URL`. These pre-fill the relevant fields in the merge dialog.
:copy	Brings up the branch/tag dialog. The `/path` is the working copy to branch/tag from.
: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 ommitted, then the diff is done between the file in `/path` and its BASE.
:conflicteditor	Starts the conflicteditor 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 work-

Command	Description
	ing 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.
: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.
: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. The 'lock' dialog is shown so the user can enter a comment for the lock. `/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. `/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" /notempfile /closeonend

TortoiseProc.exe /command:update /path:"c:\svn_wc\" /notempfile /closeonend

TortoiseProc.exe /command:log /path:"c:\svn_wc\file1.txt"
                 /revstart:50 /revend:60 /notempfile /closeonend
```

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 TSVN 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 fulltext 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 textfile, 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 > patchfile
```

TortoiseSVN creates a patchfile 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.

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.
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.
BDB	Berkeley DB. A well tested database backend for repositories, that cannot be used on network shares. Default for pre 1.2 repositories.
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 bugfixes, 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 commited 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	FS Filesystem. 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*.
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 uncommittable, 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”.
History	Show the revision history of a file or folder. Also known as “Log”.
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