Table of Contents

[1. CHECKING OUT A WORKING COPY 4](#_Toc462735031)

[2. COMMITTING YOUR CHANGES TO THE REPOSITORY 5](#_Toc462735033)

[2.1. The Commit Dialog 5](#_Toc462735034)

[2.2. Local and Remote Status 6](#_Toc462735035)

[*Commit files or folders?* 7](#_Toc462735036)

[*Many unversioned files in the commit dialog* 7](#_Toc462735037)

[*Repairing External Renames* 7](#_Toc462735038)

[*Repairing External Copies* 7](#_Toc462735039)

[2.3. Excluding Items from the Commit List 7](#_Toc462735040)

[2.4. Commit Log Messages 8](#_Toc462735041)

[2.5. Commit Progress 9](#_Toc462735042)

[3. UPDATE YOUR WORKING COPY WITH CHANGES FROM OTHERS 10](#_Toc462735043)

[3.1. Update to Revision 11](#_Toc462735045)

[*Multiple Files/Folders* 11](#_Toc462735046)

[4. RESOLVING CONFLICTS 12](#_Toc462735047)

[4.1. File Conflicts 12](#_Toc462735048)

[4.2. Property Conflicts 13](#_Toc462735049)

[4.3. Tree Conflicts 13](#_Toc462735050)

[*4.3.1* *Local delete, incoming edit upon update* 13](#_Toc462735051)

[*4.3.2* *Local edit, incoming delete upon update* 14](#_Toc462735052)

[*4.3.3* *Local delete, incoming delete upon update* 14](#_Toc462735053)

[*4.3.4* *Local missing, incoming edit upon merge* 15](#_Toc462735054)

[*4.3.5* *Local edit, incoming delete upon merge* 15](#_Toc462735055)

[*4.3.6* *Local delete, incoming delete upon merge* 15](#_Toc462735056)

[*4.3.7* *Other tree conflicts* 16](#_Toc462735057)

[5. GETTING STATUS INFORMATION 17](#_Toc462735058)

[5.1. Icon Overlays 17](#_Toc462735059)

[*Repairing External Renames* 19](#_Toc462735061)

[*Repairing External Copies* 19](#_Toc462735062)

[5.2. Viewing Diffs 20](#_Toc462735063)

[6. VIEWING DIFFERENCES 21](#_Toc462735065)

[6.1. File Differences 21](#_Toc462735066)

[6.2. Line-end and Whitespace Options 22](#_Toc462735067)

[6.3. Comparing Folders 23](#_Toc462735068)

[7. ADDING NEW FILES AND DIRECTORIES 25](#_Toc462735069)

[*Many Adds* 25](#_Toc462735070)

[8. COPYING/MOVING/RENAMING FILES AND FOLDERS 26](#_Toc462735071)

[*Cannot copy between repositories* 26](#_Toc462735072)

[9. DELETING, MOVING AND RENAMING 27](#_Toc462735073)

[9.1. Deleting files and folders 27](#_Toc462735074)

[*Getting a deleted file or folder back* 27](#_Toc462735075)

[9.2. Moving files and folders 28](#_Toc462735076)

[*Commit the parent folder* 28](#_Toc462735077)

[*Do Not SVN Move Externals* 28](#_Toc462735078)

[9.3. Dealing with filename case conflicts 28](#_Toc462735079)

[*Preventing two files with the same name* 29](#_Toc462735080)

[9.4. Repairing File Renames 29](#_Toc462735081)

[9.5. Deleting Unversioned Files 29](#_Toc462735082)

[10. UNDO CHANGES 30](#_Toc462735083)

[*Undoing Changes which have been Committed* 30](#_Toc462735084)

[*Revert is Slow* 31](#_Toc462735085)

[11. EXTERNAL ITEMS 32](#_Toc462735086)

[11.1. External Folders 32](#_Toc462735087)

[*Use explicit revision numbers* 32](#_Toc462735089)

[11.2. External Files 34](#_Toc462735090)

[11.3. Creating externals via drag and drop 34](#_Toc462735091)

[12. MERGING 35](#_Toc462735092)

[12.1. Merging a Range of Revisions 36](#_Toc462735093)

[12.2. Merging Two Different Trees 38](#_Toc462735095)

[12.3. Merge Options 38](#_Toc462735096)

[12.4. Reviewing the Merge Results 39](#_Toc462735097)

[12.5. Merge Tracking 40](#_Toc462735101)

[12.6. Handling Conflicts during Merge 40](#_Toc462735102)

[12.7. Feature Branch Maintenance 42](#_Toc462735103)

[13. WHO CHANGED WHICH LINE? 43](#_Toc462735104)

[13.1. Blame for Files 43](#_Toc462735105)

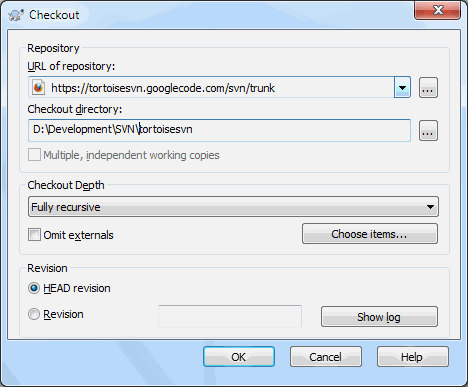
[13.2. Blame Differences 45](#_Toc462735106)

# CHECKING OUT A WORKING COPY

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

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

**Figure 4.7. The Checkout dialog**



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

**[](https://www.bing.com/images/search?q=important&view=detailv2&&id=FE726C290560E017C7D7CD5523EDDAF9E947DAEB&selectedIndex=1&ccid=1TZo0V2n&simid=608053965959662141&thid=OIP.Md53668d15da7c7ebd79f70e687986f8co0)** **Important!**

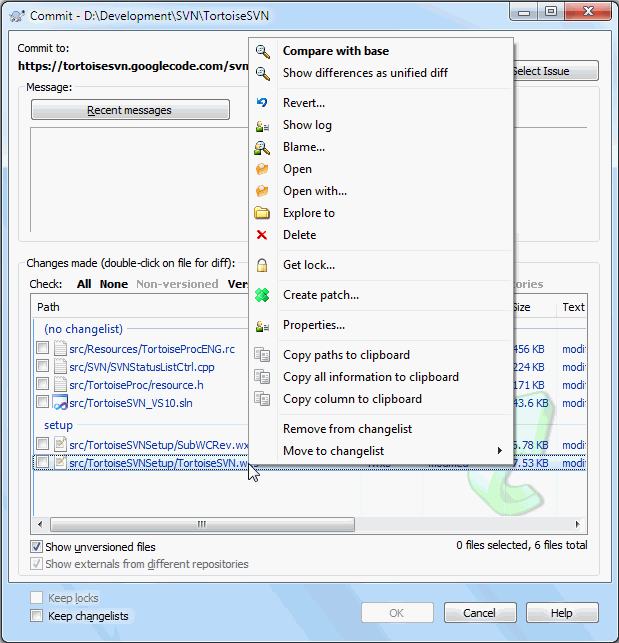
In the default setting, the checkout menu item is not located in the TortoiseSVN submenu but is shown at the top explorer menu. TortoiseSVN commands that are not in the submenu have SVN prepended:  **SVN Checkout...**

# COMMITTING YOUR CHANGES TO THE REPOSITORY

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

## The Commit Dialog

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

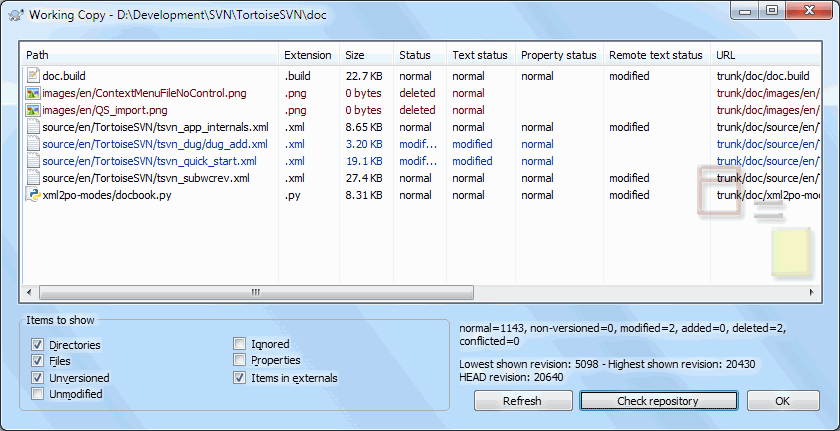
**Figure 4.8. The Commit dialog**

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.

To quickly check or uncheck types of files like all versioned files or all modified files, click the link items just above the list of shown items.

## Local and Remote Status

**Figure 4.14. 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. By default, the **Check Repository** button only fetches the remote status with the checkout depth of the working copy. If you want to see all files and folders in the repository, even those you have not checked out, then you have to hold down the **Shift** key when you click on the **Check Repository** button.

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.

### *[https://tse1.mm.bing.net/th?&id=OIP.M0347418a11c15c8213c3fb8473f9ed9fo0&w=295&h=300&c=0&pid=1.9&rs=0&p=0&r=0](https://www.bing.com/images/search?q=important&view=detailv2&&id=77B981E696D9E23B52FDAB9B7E2090F2669FF916&selectedIndex=71&ccid=A0dBihHB&simid=608022213276732253&thid=OIP.M0347418a11c15c8213c3fb8473f9ed9fo0)**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.

### *[https://tse1.mm.bing.net/th?&id=OIP.M0347418a11c15c8213c3fb8473f9ed9fo0&w=295&h=300&c=0&pid=1.9&rs=0&p=0&r=0](https://www.bing.com/images/search?q=important&view=detailv2&&id=77B981E696D9E23B52FDAB9B7E2090F2669FF916&selectedIndex=71&ccid=A0dBihHB&simid=608022213276732253&thid=OIP.M0347418a11c15c8213c3fb8473f9ed9fo0)**Many unversioned files in the commit dialog*

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

* Add the file (or a wildcard extension) to the list of files to exclude on the settings page. This will affect every working copy you have.
* Add the file to the svn:ignore list using **TortoiseSVN** → **Add to ignore list** This will only affect the directory on which you set the svn:ignore property. Using the SVN Property Dialog, you can alter the svn:ignore property for a directory.
* Add the file to the svn:global-ignores list using **TortoiseSVN** → **Add to ignore list (recursively)** This will affect the directory on which you set the svn:global-ignores property and all subfolders as well.

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

### *[https://tse1.mm.bing.net/th?&id=OIP.M0347418a11c15c8213c3fb8473f9ed9fo0&w=295&h=300&c=0&pid=1.9&rs=0&p=0&r=0](https://www.bing.com/images/search?q=important&view=detailv2&&id=77B981E696D9E23B52FDAB9B7E2090F2669FF916&selectedIndex=71&ccid=A0dBihHB&simid=608022213276732253&thid=OIP.M0347418a11c15c8213c3fb8473f9ed9fo0)**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.

### *[https://tse1.mm.bing.net/th?&id=OIP.M0347418a11c15c8213c3fb8473f9ed9fo0&w=295&h=300&c=0&pid=1.9&rs=0&p=0&r=0](https://www.bing.com/images/search?q=important&view=detailv2&&id=77B981E696D9E23B52FDAB9B7E2090F2669FF916&selectedIndex=71&ccid=A0dBihHB&simid=608022213276732253&thid=OIP.M0347418a11c15c8213c3fb8473f9ed9fo0)**Repairing External Copies*

If you made a copy of a file but forgot to use the Subversion command to do so, you can repair that copy so the new file doesn't lose its history. Simply select both the old name (normal or modified) and the new name (unversioned) and use **Context Menu** → **Repair Copy** to pair the two files as a copy.

## Excluding Items from the Commit List

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

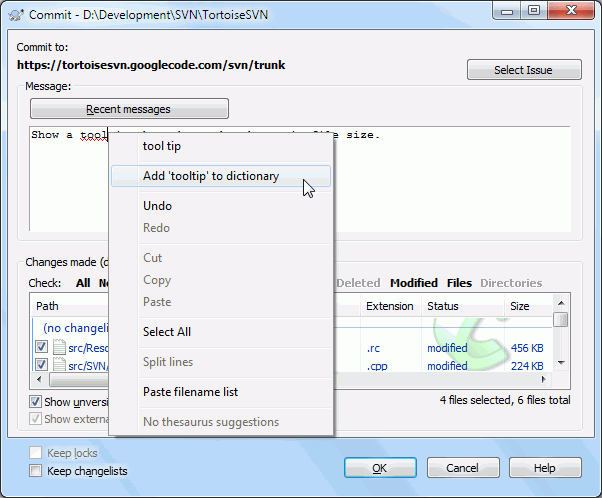
To help out in awkward cases like this, we have reserved a changelist called ignore-on-commit. Any file added to this changelist will automatically be unchecked in the commit dialog. You can still commit changes, but you have to select it manually in the commit dialog.

## Commit Log Messages

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

You can apply simple formatting to your log messages using a convention similar to that used within emails. To apply styling to text, use\*text\* for bold, \_text\_ for underlining, and ^text^ for italics.

**Figure 4.9. The Commit Dialog Spellchecker**

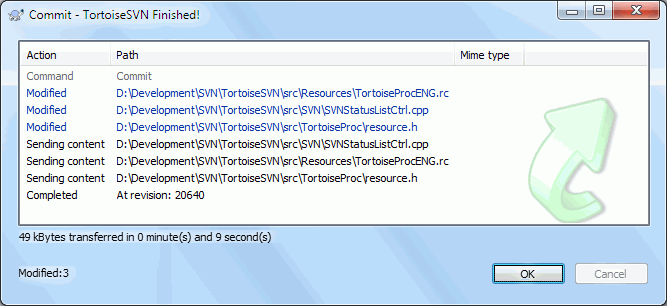


TortoiseSVN includes a spellchecker to help you get your log messages right. This will highlight any mis-spelled words. Use the context menu to access the suggested corrections. Of course, it doesn't know *every* technical term that you do, so correctly spelt words will sometimes show up as errors. But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

## Commit Progress

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

**Figure 4.10. The Progress dialog showing a commit in progress**



The progress dialog uses colour coding to highlight different commit actions

Blue: Committing a modification.

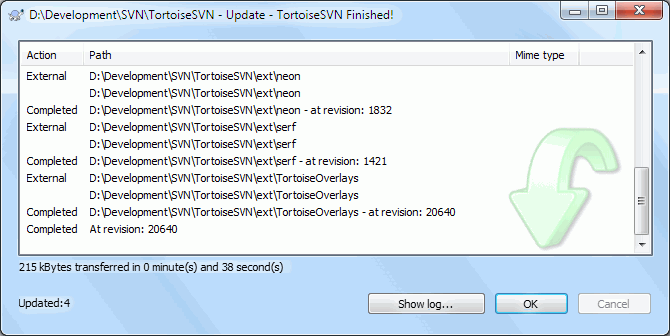
Purple: Committing a new addition.

Dark red: Committing a deletion or a replacement.

Black: All other items.

# UPDATE YOUR WORKING COPY WITH CHANGES FROM OTHERS

**Figure 4.11. 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.

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.

To make it easier to include or exclude specific items from the checkout click the **Choose items...** button. This opens a new dialog where you can check all items you want in your working copy and uncheck all the items you don't want.

You can also choose whether to ignore any external projects in the update (i.e. projects referenced using svn:externals).

### *Image result for caution icon**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 [the section called “Roll back (Undo) revisions in the repository”](https://tortoisesvn.net/docs/nightly/TortoiseSVN_en/tsvn-howto-rollback.html) for further instructions, and alternative methods.

## Update to Revision

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. You should also note that the item will show a normal green overlay, so it is indistinguishable from files which are up-to-date.

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.

### *[https://tse1.mm.bing.net/th?&id=OIP.M0347418a11c15c8213c3fb8473f9ed9fo0&w=295&h=300&c=0&pid=1.9&rs=0&p=0&r=0](https://www.bing.com/images/search?q=important&view=detailv2&&id=77B981E696D9E23B52FDAB9B7E2090F2669FF916&selectedIndex=71&ccid=A0dBihHB&simid=608022213276732253&thid=OIP.M0347418a11c15c8213c3fb8473f9ed9fo0)**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.

# RESOLVING CONFLICTS

Once in a while, you will get a *conflict* when you update/merge your files from the repository or when you switch your working copy to a different URL. There are two kinds of conflicts:

1. **File conflicts**:

A file conflict occurs if two (or more) developers have changed the same few lines of a file.

1. **Tree conflicts**:

A tree conflict occurs when a developer moved/renamed/deleted a file or folder, which another developer either also has moved/renamed/deleted or just modified.

## File Conflicts

A file 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. The conflicting area in a text file 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:

1. **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.

1. **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.

1. **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 text editor to resolve the conflict manually. You should decide what the code should look like, do the necessary changes and save the file. Using a merge tool such as TortoiseMerge or one of the other popular tools is generally the easier option as they generally present the files involved in a 3-pane view and you don't have to worry about the conflict markers. If you do use a text editor then you should search for lines starting with the string <<<<<<<.

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.

## Property Conflicts

A property conflict occurs when two or more developers have changed the same property. As with file content, resolving the conflict can only be done by the developers.

If one of the changes must override the other then choose the option to **Resolve using local property** or **Resolve using remote property**. If the changes must be merged then select **Manually edit property**, sort out what the property value should be and mark as resolved.

## Tree Conflicts

A tree conflict occurs when a developer moved/renamed/deleted a file or folder, which another developer either also has moved/renamed/deleted or just modified. There are many different situations that can result in a tree conflict, and all of them require different steps to resolve the conflict.

When a file is deleted locally in Subversion, the file is also deleted from the local file system, so even if it is part of a tree conflict it cannot show a conflicted overlay and you cannot right click on it to resolve the conflict. Use the Check for Modifications dialog instead to access the Edit conflicts option.

TortoiseSVN can help find the right place to merge changes, but there may be additional work required to sort out the conflicts. Remember that after an update the working BASE will always contain the revision of each item as it was in the repository at the time of update. If you revert a change after updating it goes back to the repository state, not to the way it was when you started making your own local changes.

### *Local delete, incoming edit upon update*

1. Developer A modifies Foo.c and commits it to the repository.
2. Developer B has simultaneously moved Foo.c to Bar.c in his working copy, or simply deleted Foo.c or its parent folder.

An update of developer B's working copy results in a tree conflict:

* Foo.c has been deleted from working copy, but is marked with a tree conflict.
* If the conflict results from a rename rather than a delete then Bar.c is marked as added, but does not contain developer A's modifications.

Developer B now has to choose whether to keep Developer A's changes. In the case of a file rename, he can merge the changes toFoo.c into the renamed file Bar.c. For simple file or directory deletions he can choose to keep the item with Developer A's changes and discard the deletion. Or, by marking the conflict as resolved without doing anything he effectively discards Developer A's changes.

The conflict edit dialog offers to merge changes if it can find the original file of the renamed Bar.c. Depending on where the update was invoked, it may not be possible to find the source file.

### *Local edit, incoming delete upon update*

1. Developer A moves Foo.c to Bar.c and commits it to the repository.
2. Developer B modifies Foo.c in his working copy.

Or in the case of a folder move ...

1. Developer A moves parent folder FooFolder to BarFolder and commits it to the repository.
2. Developer B modifies Foo.c in his working copy.

An update of developer B's working copy results in a tree conflict. For a simple file conflict:

* Bar.c is added to the working copy as a normal file.
* Foo.c is marked as added (with history) and has a tree conflict.

For a folder conflict:

* BarFolder is added to the working copy as a normal folder.
* FooFolder is marked as added (with history) and has a tree conflict.

Foo.c is marked as modified.

Developer B now has to decide whether to go with developer A's reorganisation and merge her changes into the corresponding file in the new structure, or simply revert A's changes and keep the local file.

To merge her local changes with the reshuffle, Developer B must first find out to what filename the conflicted file Foo.c was renamed/moved in the repository. This can be done by using the log dialog. The changes must then be merged by hand as there is currently no way to automate or even simplify this process. Once the changes have been ported across, the conflicted path is redundant and can be deleted. In this case use the Remove button in the conflict editor dialog to clean up and mark the conflict as resolved.

If Developer B decides that A's changes were wrong then she must choose the Keep button in the conflict editor dialog. This marks the conflicted file/folder as resolved, but Developer A's changes need to be removed by hand. Again the log dialog helps to track down what was moved.

### *Local delete, incoming delete upon update*

1. Developer A moves Foo.c to Bar.c and commits it to the repository.
2. Developer B moves Foo.c to Bix.c.

An update of developer B's working copy results in a tree conflict:

* Bix.c is marked as added with history.
* Bar.c is added to the working copy with status 'normal'.
* Foo.c is marked as deleted and has a tree conflict.

To resolve this conflict, Developer B has to find out to what filename the conflicted file Foo.c was renamed/moved in the repository. This can be done by using the log dialog.

Then developer B has to decide which new filename of Foo.c to keep - the one done by developer A or the rename done by himself.

After developer B has manually resolved the conflict, the tree conflict has to be marked as resolved with the button in the conflict editor dialog.

### *Local missing, incoming edit upon merge*

1. Developer A working on trunk modifies Foo.c and commits it to the repository
2. Developer B working on a branch moves Foo.c to Bar.c and commits it to the repository

A merge of developer A's trunk changes to developer B's branch working copy results in a tree conflict:

* Bar.c is already in the working copy with status 'normal'.
* Foo.c is marked as missing with a tree conflict.

To resolve this conflict, Developer B has to mark the file as resolved in the conflict editor dialog, which will remove it from the conflict list. She then has to decide whether to copy the missing file Foo.c from the repository to the working copy, whether to merge Developer A's changes to Foo.c into the renamed Bar.c or whether to ignore the changes by marking the conflict as resolved and doing nothing else.

Note that if you copy the missing file from the repository and then mark as resolved, your copy will be removed again. You have to resolve the conflict first.

### *Local edit, incoming delete upon merge*

1. Developer A working on trunk moves Foo.c to Bar.c and commits it to the repository.
2. Developer B working on a branch modifies Foo.c and commits it to the repository.
3. Developer A working on trunk moves parent folder FooFolder to BarFolder and commits it to the repository.
4. Developer B working on a branch modifies Foo.c in her working copy.

A merge of developer A's trunk changes to developer B's branch working copy results in a tree conflict:

* Bar.c is marked as added.
* Foo.c is marked as modified with a tree conflict.

Developer B now has to decide whether to go with developer A's reorganisation and merge her changes into the corresponding file in the new structure, or simply revert A's changes and keep the local file.

To merge her local changes with the reshuffle, Developer B must first find out to what filename the conflicted file Foo.c was renamed/moved in the repository. This can be done by using the log dialog for the merge source. The conflict editor only shows the log for the working copy as it does not know which path was used in the merge, so you will have to find that yourself. The changes must then be merged by hand as there is currently no way to automate or even simplify this process. Once the changes have been ported across, the conflicted path is redundant and can be deleted. In this case use the Remove button in the conflict editor dialog to clean up and mark the conflict as resolved.

If Developer B decides that A's changes were wrong then she must choose the Keep button in the conflict editor dialog. This marks the conflicted file/folder as resolved, but Developer A's changes need to be removed by hand. Again the log dialog for the merge source helps to track down what was moved.

### *Local delete, incoming delete upon merge*

1. Developer A working on trunk moves Foo.c to Bar.c and commits it to the repository.
2. Developer B working on a branch moves Foo.c to Bix.c and commits it to the repository.

A merge of developer A's trunk changes to developer B's branch working copy results in a tree conflict:

* Bix.c is marked with normal (unmodified) status.
* Bar.c is marked as added with history.
* Foo.c is marked as missing and has a tree conflict.

To resolve this conflict, Developer B has to find out to what filename the conflicted file Foo.c was renamed/moved in the repository. This can be done by using the log dialog for the merge source. The conflict editor only shows the log for the working copy as it does not know which path was used in the merge, so you will have to find that yourself.

Then developer B has to decide which new filename of Foo.c to keep - the one done by developer A or the rename done by himself.

After developer B has manually resolved the conflict, the tree conflict has to be marked as resolved with the button in the conflict editor dialog.

### *Other tree conflicts*

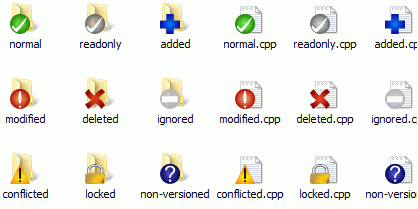
There are other cases which are labeled as tree conflicts simply because the conflict involves a folder rather than a file. For example if you add a folder with the same name to both trunk and branch and then try to merge you will get a tree conflict. If you want to keep the folder from the merge target, just mark the conflict as resolved. If you want to use the one in the merge source then you need to SVN delete the one in the target first and run the merge again. If you need anything more complicated then you have to resolve manually.

# 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.

## Icon Overlays

**Figure 4.12. 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.

https://tortoisesvn.net/docs/nightly/TortoiseSVN_en/images/InSubVersionIcon.png

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

https://tortoisesvn.net/docs/nightly/TortoiseSVN_en/images/ModifiedIcon.png

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.

https://tortoisesvn.net/docs/nightly/TortoiseSVN_en/images/ConflictIcon.png

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

https://tortoisesvn.net/docs/nightly/TortoiseSVN_en/images/ReadOnlyIcon.png

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

https://tortoisesvn.net/docs/nightly/TortoiseSVN_en/images/LockedIcon.png

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.

https://tortoisesvn.net/docs/nightly/TortoiseSVN_en/images/DeletedIcon.png

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.

https://tortoisesvn.net/docs/nightly/TortoiseSVN_en/images/AddedIcon.png

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

https://tortoisesvn.net/docs/nightly/TortoiseSVN_en/images/IgnoredIcon.png

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

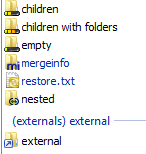
https://tortoisesvn.net/docs/nightly/TortoiseSVN_en/images/UnversionedIcon.png

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

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

Now that there are more Tortoise clients around (TortoiseCVS, TortoiseHg, ...) the icon limit becomes a real problem. To work around this, the TortoiseSVN project introduced a common shared icon set, loaded as a DLL, which can be used by all Tortoise clients. Check with your client provider to see if this has been integrated yet :-)

Overlay icons are used to indicate other states as well. The screenshot below shows all the possible overlays that are shown if necessary.



Overlays are shown for the following states:

* Checkout depth empty, meaning only the item itself.
* Checkout depth files, meaning only the item itself and all file children without child folders.
* Checkout depth immediates, meaning only the item itself and all file and folder children, but without children of the child folders.
* Nested items, i.e., working copies inside the working copy.
* External items, i.e., all items that are added via an svn:externals property.
* Items that are restored after a commit. See [the section called “Commit only parts of files”](https://tortoisesvn.net/docs/nightly/TortoiseSVN_en/tsvn-dug-commit.html#tsvn-dug-commit-restore) for details.
* Items that have property modifications, but only to the svn:mergeinfo property. If any other property is modified, the overlay is not used.

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! The context menu allows you to switch them back to the normal path again.

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, or you can save a copy simply by dragging it into a folder in explorer.

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.

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

### *[https://tse1.mm.bing.net/th?&id=OIP.M0347418a11c15c8213c3fb8473f9ed9fo0&w=295&h=300&c=0&pid=1.9&rs=0&p=0&r=0](https://www.bing.com/images/search?q=important&view=detailv2&&id=77B981E696D9E23B52FDAB9B7E2090F2669FF916&selectedIndex=71&ccid=A0dBihHB&simid=608022213276732253&thid=OIP.M0347418a11c15c8213c3fb8473f9ed9fo0)**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.

### *[https://tse1.mm.bing.net/th?&id=OIP.M0347418a11c15c8213c3fb8473f9ed9fo0&w=295&h=300&c=0&pid=1.9&rs=0&p=0&r=0](https://www.bing.com/images/search?q=important&view=detailv2&&id=77B981E696D9E23B52FDAB9B7E2090F2669FF916&selectedIndex=71&ccid=A0dBihHB&simid=608022213276732253&thid=OIP.M0347418a11c15c8213c3fb8473f9ed9fo0)**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.

### *[https://tse1.mm.bing.net/th?&id=OIP.M0347418a11c15c8213c3fb8473f9ed9fo0&w=295&h=300&c=0&pid=1.9&rs=0&p=0&r=0](https://www.bing.com/images/search?q=important&view=detailv2&&id=77B981E696D9E23B52FDAB9B7E2090F2669FF916&selectedIndex=71&ccid=A0dBihHB&simid=608022213276732253&thid=OIP.M0347418a11c15c8213c3fb8473f9ed9fo0)**Repairing External Copies*

If you made a copy of a file but forgot to use the Subversion command to do so, you can repair that copy so the new file doesn't lose its history. Simply select both the old name (normal or modified) and the new name (unversioned) and use **Context Menu** → **Repair Copy** to pair the two files as a copy.

## 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.

### *[https://tse1.mm.bing.net/th?&id=OIP.M0347418a11c15c8213c3fb8473f9ed9fo0&w=295&h=300&c=0&pid=1.9&rs=0&p=0&r=0](https://www.bing.com/images/search?q=important&view=detailv2&&id=77B981E696D9E23B52FDAB9B7E2090F2669FF916&selectedIndex=71&ccid=A0dBihHB&simid=608022213276732253&thid=OIP.M0347418a11c15c8213c3fb8473f9ed9fo0)**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.

# VIEWING DIFFERENCES

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

## 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**.

Difference from a previous revision

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

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

Difference between two previous revisions

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

If you did this from the revision log for a folder, a Compare Revisions dialog appears, showing a list of changed files in that folder.

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**.

If the files to compare are not located in the same folder, use the command **TortoiseSVN** → **Diff later** to mark the first file for diffing, then browse to the second file and use **TortoiseSVN** → **Diff with "path/of/marked/file"**. To remove the marked file, use the command **TortoiseSVN** → **Diff later** again, but hold down the **Ctrl**-modifier while clicking on it.

Difference between WC file/folder and a URL

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

Difference with blame information

If you want to see not only the differences but also the author, revision and date that changes were made, you can combine the diff and blame reports from within the revision log dialog.

Difference between folders

The built-in tools supplied with TortoiseSVN do not support viewing differences between directory hierarchies. But if you have an external tool which does support that feature, you can use that instead.

If you have configured a third party diff tool, you can use **Shift** when selecting the Diff command to use the alternate tool.

## Line-end and Whitespace Options

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

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

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

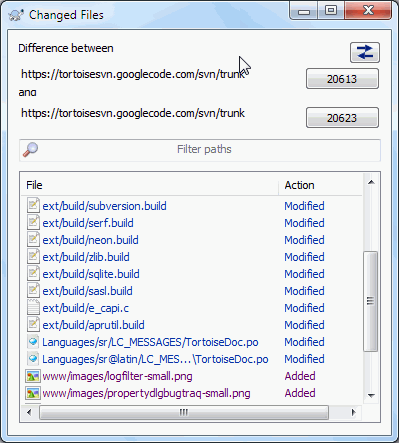
Ignore whitespace changes excludes changes which are due solely to a change in the amount or type of whitespace, e.g. changing the indentation or changing tabs to spaces. Adding whitespace where there was none before, or removing a whitespace completely is still shown as a change.

Ignore all whitespaces excludes all whitespace-only changes.

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

## Comparing Folders

**Figure 4.27. 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 export a change tree, which is useful if you need to send someone else your project tree structure, but containing only the files which have changed. This operation works on the selected files only, so you need to select the files of interest - usually that means all of them - and then **Context menu** → **Export selection to...**. You will be prompted for a location to save the change tree.

You can also export the list of changed files to a text file using **Context menu** → **Save list of selected files to...**.

If you want to export the list of files and the actions (modified, added, deleted) as well, you can do that using **Context menu** → **Copy selection to clipboard**.

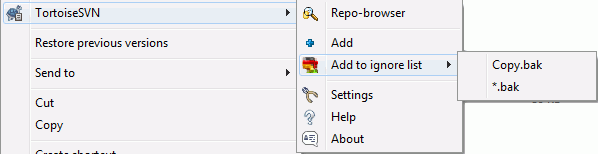
The button at the top allows you to change the direction of comparison. You can show the changes need to get from A to B, or if you prefer, from B to A.

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

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

# ADDING NEW FILES AND DIRECTORIES

**Figure 4.29. 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!

### *[https://tse1.mm.bing.net/th?&id=OIP.M0347418a11c15c8213c3fb8473f9ed9fo0&w=295&h=300&c=0&pid=1.9&rs=0&p=0&r=0](https://www.bing.com/images/search?q=important&view=detailv2&&id=77B981E696D9E23B52FDAB9B7E2090F2669FF916&selectedIndex=71&ccid=A0dBihHB&simid=608022213276732253&thid=OIP.M0347418a11c15c8213c3fb8473f9ed9fo0)**Many Adds*

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

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

1. select the files you want to add
2. **right drag** them to the new location inside the working copy
3. release the right mouse button
4. select **Context Menu** → **SVN Add files to this WC**. The files will then be copied to the working copy and added to version control.

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

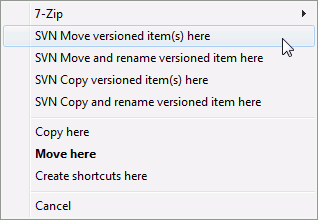
If you add a file or folder by mistake, you can undo the addition before you commit using **TortoiseSVN** → **Undo add...**

# COPYING/MOVING/RENAMING FILES AND FOLDERS

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

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

**Figure 4.30. Right drag menu for a directory under version control**



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

You can also copy or move versioned files within a working copy, or between two working copies, using the familiar cut-and-paste method. Use the standard Windows **Copy** or **Cut** to copy one or more versioned items to the clipboard. If the clipboard contains such versioned items, you can then use **TortoiseSVN** → **Paste** (note: not the standard Windows **Paste**) to copy or move those items to the new working copy location.

You can copy files and folders from your working copy to another location in the repository using **TortoiseSVN** → **Branch/Tag**.

You can locate an older version of a file or folder in the log dialog and copy it to a new location in the repository directly from the log dialog using **Context menu** → **Create branch/tag from revision**.

You can also use the repository browser to locate content you want, and copy it into your working copy directly from the repository, or copy between two locations within the repository.

### *[https://tse1.mm.bing.net/th?&id=OIP.M0347418a11c15c8213c3fb8473f9ed9fo0&w=295&h=300&c=0&pid=1.9&rs=0&p=0&r=0](https://www.bing.com/images/search?q=important&view=detailv2&&id=77B981E696D9E23B52FDAB9B7E2090F2669FF916&selectedIndex=71&ccid=A0dBihHB&simid=608022213276732253&thid=OIP.M0347418a11c15c8213c3fb8473f9ed9fo0)**Cannot copy between repositories*

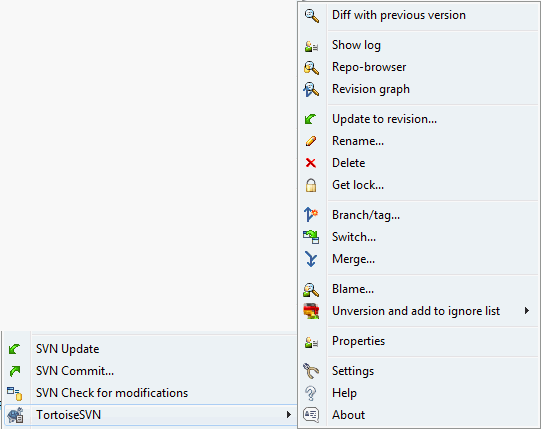
Whilst you can copy or move files and folders within a repository, you cannot copy or move from one repository to another while preserving history using TortoiseSVN. Not even if the repositories live on the same server. All you can do is copy the content in its current state and add it as new content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

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

# DELETING, MOVING AND RENAMING

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

**Figure 4.32. Explorer context menu for versioned files**



## Deleting files and folders

Use **TortoiseSVN** → **Delete** to remove files or folders from Subversion.

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

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

If an item is deleted via the explorer instead of using the TortoiseSVN context menu, the commit dialog shows those items as missing and lets you remove them from version control too before the commit. However, if you update your working copy, Subversion will spot the missing item 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.

### *[https://tse1.mm.bing.net/th?&id=OIP.M0347418a11c15c8213c3fb8473f9ed9fo0&w=295&h=300&c=0&pid=1.9&rs=0&p=0&r=0](https://www.bing.com/images/search?q=important&view=detailv2&&id=77B981E696D9E23B52FDAB9B7E2090F2669FF916&selectedIndex=71&ccid=A0dBihHB&simid=608022213276732253&thid=OIP.M0347418a11c15c8213c3fb8473f9ed9fo0)**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.

## Moving files and folders

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

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

1. select the files or directories you want to move
2. **right drag** them to the new location inside the working copy
3. release the right mouse button
4. in the popup menu select **Context Menu** → **SVN Move versioned files here**

### *Image result for caution icon**Commit the parent folder*

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

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

Another way of moving or copying files is to use the Windows copy/cut commands. Select the files you want to copy, right click and choose **Context Menu** → **Copy** from the explorer context menu. Then browse to the target folder, right click and choose**TortoiseSVN** → **Paste**. For moving files, choose **Context Menu** → **Cut** instead of **Context Menu** → **Copy**.

You can also use the repository browser to move items around.

### *Image result for caution icon**Do Not SVN Move Externals*

You should not use the TortoiseSVN Move or Rename commands on a folder which has been created usingsvn: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.

## Dealing with filename case conflicts

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

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

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

### *[https://tse1.mm.bing.net/th?&id=OIP.M0347418a11c15c8213c3fb8473f9ed9fo0&w=295&h=300&c=0&pid=1.9&rs=0&p=0&r=0](https://www.bing.com/images/search?q=important&view=detailv2&&id=77B981E696D9E23B52FDAB9B7E2090F2669FF916&selectedIndex=71&ccid=A0dBihHB&simid=608022213276732253&thid=OIP.M0347418a11c15c8213c3fb8473f9ed9fo0)**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/>that will prevent checkins which result in case conflicts.

## 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.

## 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.

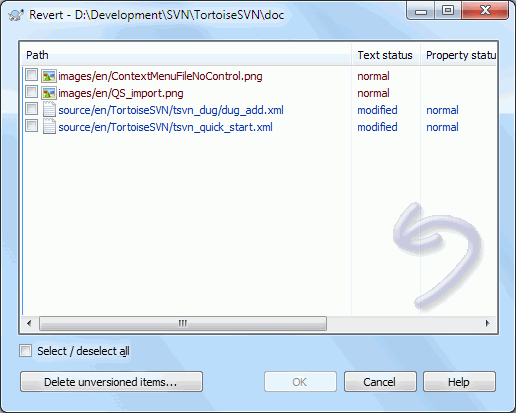
TortoiseSVN provides just such an option using **Extended Context Menu** → **Delete unversioned items...**. You have to hold the**Shift** while right clicking on a folder in the explorer list pane (right pane) in order to see this in the extended context menu. This will produce a dialog which lists all 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.

# UNDO CHANGES

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

**Figure 4.33. Revert dialog**



If you also want to clear all the changelists that are set, check the box at the bottom of the dialog.If you want to undo a deletion or a rename, you need to use Revert on the parent folder as the deleted item does not exist for you to right click on.

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

The columns in this dialog can be customized in the same way as the columns in the Check for modifications dialog. [](https://www.bing.com/images/search?q=important&view=detailv2&&id=77B981E696D9E23B52FDAB9B7E2090F2669FF916&selectedIndex=71&ccid=A0dBihHB&simid=608022213276732253&thid=OIP.M0347418a11c15c8213c3fb8473f9ed9fo0)Since revert is sometimes used to clean up a working copy, there is an extra button which allows you to delete unversioned items as well. When you click this button another dialog comes up listing all the unversioned items, which you can then select for deletion.

### *Undoing Changes which have been Committed*

**Revert** will only undo your local changes. It does not undo any changes which have already been committed.

### *[https://tse1.mm.bing.net/th?&id=OIP.M0347418a11c15c8213c3fb8473f9ed9fo0&w=295&h=300&c=0&pid=1.9&rs=0&p=0&r=0](https://www.bing.com/images/search?q=important&view=detailv2&&id=77B981E696D9E23B52FDAB9B7E2090F2669FF916&selectedIndex=71&ccid=A0dBihHB&simid=608022213276732253&thid=OIP.M0347418a11c15c8213c3fb8473f9ed9fo0)**Revert is Slow*

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

# EXTERNAL ITEMS

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 files or 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 to pull in the specified resource at the locations where they are needed.

## External Folders

Let's say you check out a working copy of /project1 to D:\dev\project1. Select the folder D:\dev\project1, **right click** and choose **Windows Menu** → **Properties** from the context menu. The Properties Dialog comes up. Then go to the Subversion tab. There, you can set properties. Click **Properties...**. In the properties dialog, either double click on the svn:externals if it already exists, or click on the **New...**button and select externals from the menu. To add a new external, click the **New...** and then fill in the required information in the shown dialog.

### *Image result for caution icon**Caution*

URLs must be properly escaped or they will not work, e.g. you must replace each space with %20.

If you want the local path to include spaces or other special characters, you can enclose it in double quotes, or you can use the \(backslash) character as a Unix shell style escape character preceding any special character. Of course this also means that you must use/ (forward slash) as a path delimiter. Note that this behaviour is new in Subversion 1.6 and will not work with older clients.

### *[https://tse1.mm.bing.net/th?&id=OIP.M0347418a11c15c8213c3fb8473f9ed9fo0&w=295&h=300&c=0&pid=1.9&rs=0&p=0&r=0](https://www.bing.com/images/search?q=important&view=detailv2&&id=77B981E696D9E23B52FDAB9B7E2090F2669FF916&selectedIndex=71&ccid=A0dBihHB&simid=608022213276732253&thid=OIP.M0347418a11c15c8213c3fb8473f9ed9fo0)**Use explicit revision numbers*

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

The edit dialog for svn:externals properties allows you to select the externals and automatically set them explicitly to the HEAD revision.

If the external project is in the same repository, any changes you make 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 shown or indicated when you commit the main project, but you have to commit those external changes separately.

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

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

Relative to parent directory

These URLs always begin with the string ../ for example:

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

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

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

Relative to repository root

These URLs always begin with the string ^/ for example:

^/widgets/foo common/foo-widget

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

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

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

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

Relative to scheme

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

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

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

Relative to the server's hostname

URLs beginning with the string / copy the scheme and the hostname part of the URL, for example:

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

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

[](https://www.bing.com/images/search?q=important&view=detailv2&&id=FE726C290560E017C7D7CD5523EDDAF9E947DAEB&selectedIndex=1&ccid=1TZo0V2n&simid=608053965959662141&thid=OIP.Md53668d15da7c7ebd79f70e687986f8co0)You can also specify a peg and operative revision for the URL if required. To learn more about peg and operative revisions, please read the [corresponding chapter](http://svnbook.red-bean.com/en/1.8/svn.advanced.pegrevs.html)in the Subversion book.

**Important!**

If you specify the target folder for the external as a subfolder like in the examples above, make sure that allfolders in between are versioned as well. So for the examples above, the folder common should be versioned!

While the external will work in most situations properly if folders in between are not versioned, there are some operations that won't work as you expect. And the status overlay icons in explorer will also not show the correct status.

## External Files

As of Subversion 1.6 you can add single file externals to your working copy using the same syntax as for folders. However, there are some restrictions.

* The path to the file external must be a direct child of the folder where you set the svn:externals property.
* The URL for a file external must be in the same repository as the URL that the file external will be inserted into; inter-repository file externals are not supported.

A file external behaves just like any other versioned file in many respects, but they cannot be moved or deleted using the normal commands; the svn:externals property must be modified instead.

## Creating externals via drag and drop

If you already have a working copy of the files or folders you want to include as externals in another working copy, you can simply add those via drag and drop from the windows explorer.

Simply **right drag** the file or folder from one working copy to where you want those to be included as externals. A context menu appears when you release the mouse button: **SVN Add as externals here** if you click on that context menu entry, the svn:externals property is automatically added. All you have to do after that is commit the property changes and update to get those externals properly included in your working copy.

# 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/1.8/svn.branchmerge.html)in the Subversion book, which gives a full description and many examples of how it is used.

The next point to note is that merging always takes place within a working copy. If you want to merge changes into a branch, you have to have a working copy for that branch checked out, and invoke the merge wizard from that working copy using **TortoiseSVN** → **Merge...**.

In general it is a good idea to perform a merge into an unmodified working copy. If you have made other changes in your WC, commit those first. If the merge does not go as you expect, you may want to revert the changes, and the Revert command will discard all changes including any you made before the merge.

There are three common use cases for merging which are handled in slightly different ways, as described below. The first page of the merge wizard asks you to select the method you need.

Merge a range of revisions

This method covers the case when you have made one or more revisions to a branch (or to the trunk) and you want to port those changes across to a different branch.

What you are asking Subversion to do is this: “ Calculate the changes necessary to get [FROM] revision 1 of branch A [TO] revision 7 of branch A, and apply those changes to my working copy (of trunk or branch B). ”

If you leave the revision range empty, Subversion uses the merge-tracking features to calculate the correct revision range to use. This is known as a reintegrate or automatic merge.

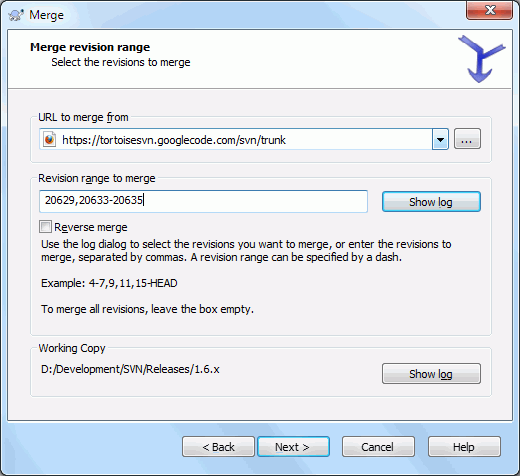
Merge two different trees

This is a more general case of the reintegrate method. What you are asking Subversion to do is: “ Calculate the changes necessary to get [FROM] the head revision of the trunk [TO] the head revision of the branch, and apply those changes to my working copy (of the trunk). ” The net result is that trunk now looks exactly like the branch.

If your server/repository does not support merge-tracking then this is the only way to merge a branch back to trunk. Another use case occurs when you are using vendor branches and you need to merge the changes following a new vendor drop into your trunk code. For more information read the chapter on [vendor branches](http://svnbook.red-bean.com/en/1.8/svn.advanced.vendorbr.html)in the Subversion Book.

## Merging a Range of Revisions

**Figure 4.54. The Merge Wizard - Select Revision Range**



In the From: field enter the full folder URL of the branch or tag containing the changes you want to port into your working copy. You may also click **...** to browse the repository and find the desired branch. If you have merged from this branch before, then just use the drop down list which shows a history of previously used URLs.

If you are merging from a renamed or deleted branch then you will have to go back to a revision where that branch still existed. In this case you will also need to specify that revision as a peg revision in the range of revisions being merged (see below), otherwise the merge will fail when it can't find that path at HEAD.

In the Revision range to merge field enter the list of revisions you want to merge. This can be a single revision, a list of specific revisions separated by commas, or a range of revisions separated by a dash, or any combination of these.

If you need to specify a peg revision for the merge, add the peg revision at the end of the revisions, e.g. 5-7,10@3. In the above example, the revisions 5,6,7 and 10 would be merged, with 3 being the peg revision.

### *[https://tse1.mm.bing.net/th?&id=OIP.Md53668d15da7c7ebd79f70e687986f8co0&w=300&h=250&c=0&pid=1.9&rs=0&p=0&r=0](https://www.bing.com/images/search?q=important&view=detailv2&&id=FE726C290560E017C7D7CD5523EDDAF9E947DAEB&selectedIndex=1&ccid=1TZo0V2n&simid=608053965959662141&thid=OIP.Md53668d15da7c7ebd79f70e687986f8co0)**Important*

There is an important difference in the way a revision range is specified with TortoiseSVN compared to the command line client. The easiest way to visualise it is to think of a fence with posts and fence panels.

With the command line client you specify the changes to merge using two “fence post” revisions which specify the before and after points.

With TortoiseSVN you specify the changeset to merge using “fence panels”. The reason for this becomes clear when you use the log dialog to specify revisions to merge, where each revision appears as a changeset.

If you are merging revisions in chunks, the method shown in the Subversion book will have you merge 100-200 this time and 200-300 next time. With TortoiseSVN you would merge 100-200 this time and 201-300 next time.

This difference has generated a lot of heat on the mailing lists. We acknowledge that there is a difference from the command line client, but we believe that for the majority of GUI users it is easier to understand the method we have implemented.

The easiest way to select the range of revisions you need is to click on **Show Log**, as this will list recent changes with their log comments. If you want to merge the changes from a single revision, just select that revision. If you want to merge changes from several revisions, then select that range (using the usual **Shift**-modifier). Click on **OK** and the list of revision numbers to merge will be filled in for you.

If you want to merge changes back out of your working copy, to revert a change which has already been committed, select the revisions to revert and make sure the Reverse merge box is checked.

If you have already merged some changes from this branch, hopefully you will have made a note of the last revision merged in the log message when you committed the change. In that case, you can use **Show Log** for the Working Copy to trace that log message. Remembering that we are thinking of revisions as changesets, you should Use the revision after 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 40.

If you are using the merge tracking features of Subversion, you do not need to remember which revisions have already been merged - Subversion will record that for you. If you leave the revision range blank, all revisions which have not yet been merged will be included.

When merge tracking is used, the log dialog will show previously merged revisions, and revisions pre-dating the common ancestor point, i.e. before the branch was copied, as greyed out. The Hide non-mergeable revisions checkbox allows you to filter out these revisions completely so you see only the revisions which can be merged.

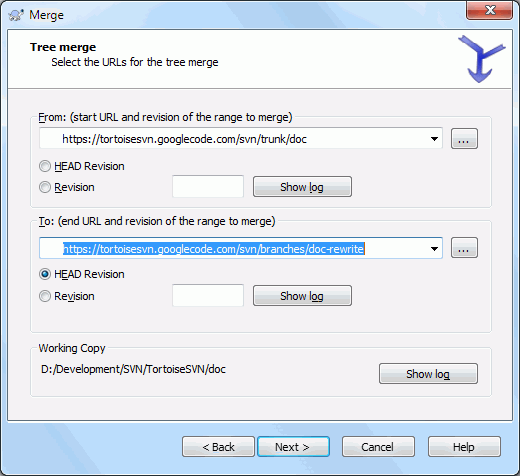
If other people may be committing changes then be careful about using the HEAD revision. It may not refer to the revision you think it does if someone else made a commit after your last update.

If you leave the range of revisions empty or have the radio button all revisions checked, then Subversion merges all not-yet merged revisions. This is known as a reintegrate or automatic merge.

There are some conditions which apply to a reintegrate merge. Firstly, the server must support merge tracking. The working copy must be of depth infinite (no sparse checkouts), and it must not have any local modifications, switched items or items that have been updated to revisions other than HEAD. All changes to trunk made during branch development must have been merged across to the branch (or marked as having been merged). The range of revisions to merge will be calculated automatically.

## Merging Two Different Trees

**Figure 4.55. The Merge Wizard - Tree Merge**



If you are using this method to merge a feature branch back to trunk, you need to start the merge wizard from within a working copy of trunk.

In the From: field enter the full folder URL of the trunk. This may sound wrong, but remember that the trunk is the start point to which you want to add the branch changes. You may also click **...** to browse the repository.

In the To: field enter the full folder URL of the feature branch.

In both the From Revision field and the To Revision field, enter the last revision number at which the two trees were synchronized. If you are sure no-one else is making commits you can use the HEAD revision in both cases. If there is a chance that someone else may have made a commit since that synchronization, use the specific revision number to avoid losing more recent commits.

You can also use **Show Log** to select the revision.

## Merge Options

This page of the wizard lets you specify advanced options, before starting the merge process. Most of the time you can just use the default settings.

You can specify the depth to use for the merge, i.e. how far down into your working copy the merge should go. The depth terms used are described in [the section called “Checkout Depth”](https://tortoisesvn.net/docs/nightly/TortoiseSVN_en/tsvn-dug-checkout.html#tsvn-dug-checkout-depth). The default depth is Working copy, which uses the existing depth setting, and is almost always what you want.

Most of the time you want merge to take account of the file's history, so that changes relative to a common ancestor are merged. Sometimes you may need to merge files which are perhaps related, but not in your repository. For example you may have imported versions 1 and 2 of a third party library into two separate directories. Although they are logically related, Subversion has no knowledge of this because it only sees the tarballs you imported. If you attempt to merge the difference between these two trees you would see a complete removal followed by a complete add. To make Subversion use only path-based differences rather than history-based differences, check the Ignore ancestry box. Read more about this topic in the Subversion book, [Noticing or Ignoring Ancestry](http://svnbook.red-bean.com/en/1.8/svn.branchmerge.advanced.html#svn.branchmerge.advanced.ancestry).

You can specify the way that line ending and whitespace changes are handled. These options are described in [the section called “Line-end and Whitespace Options”](https://tortoisesvn.net/docs/nightly/TortoiseSVN_en/tsvn-dug-diff.html#tsvn-dug-diff-eol-whitespace). The default behaviour is to treat all whitespace and line-end differences as real changes to be merged.

The checkbox marked Force the merge is used to avoid a tree conflict where an incoming delete affects a file that is either modified locally or not versioned at all. If the file is deleted then there is no way to recover it, which is why that option is not checked by default.

If you are using merge tracking and you want to mark a revision as having been merged, without actually doing the merge here, check the Only record the merge checkbox. There are two possible reasons you might want to do this. It may be that the merge is too complicated for the merge algorithms, so you code the changes by hand, then mark the change as merged so that the merge tracking algorithm is aware of it. Or you might want to prevent a particular revision from being merged. Marking it as already merged will prevent the merge occurring with merge-tracking-aware clients.

Now everything is set up, all you have to do is click on the **Merge** button. If you want to preview the results **Test Merge** simulates 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 files where conflicts may occur. Because merge tracking makes the merge process a lot more complicated, there is no guaranteed way to find out in advance whether the merge will complete without conflicts, so files marked as conflicted in a test merge may in fact merge without any problem.

The merge progress dialog shows each stage of the merge, with the revision ranges involved. This may indicate one more revision than you were expecting. For example if you asked to merge revision 123 the progress dialog will report “ Merging revisions 122 through 123”. To understand this you need to remember that Merge is closely related to Diff. The merge process works by generating a list of differences between two points in the repository, and applying those differences to your working copy. The progress dialog is simply showing the start and end points for the diff.

## Reviewing the Merge Results

The merge is now complete. It's a good idea to have a look at the merge and see if it's as expected. Merging is usually quite complicated. Conflicts often arise if the branch has drifted far from the trunk.

### *[https://tse1.mm.bing.net/th?&id=OIP.M0347418a11c15c8213c3fb8473f9ed9fo0&w=295&h=300&c=0&pid=1.9&rs=0&p=0&r=0](https://www.bing.com/images/search?q=important&view=detailv2&&id=77B981E696D9E23B52FDAB9B7E2090F2669FF916&selectedIndex=71&ccid=A0dBihHB&simid=608022213276732253&thid=OIP.M0347418a11c15c8213c3fb8473f9ed9fo0)**Tip*

Whenever revisions are merged into a working copy, TortoiseSVN generates a log message from all the merged revisions. Those are then available from the **Recent Messages** button in the commit dialog.

To customize that generated message, set the corresponding project properties on your working copy.

For Subversion clients and servers prior to 1.5, no merge information is stored and merged revisions have to be tracked manually. When you have tested the changes and come to commit this revision, your commit log message should always include the revision numbers which have been ported in the merge. If you want to apply another merge at a later time you will need to know what you have already merged, as you do not want to port a change more than once. For more information about this, refer to [Best Practices for Merging](http://svnbook.red-bean.com/en/1.4/svn.branchmerge.copychanges.html#svn.branchmerge.copychanges.bestprac)in the Subversion book.

If your server and all clients are running Subversion 1.5 or higher, the merge tracking facility will record the revisions merged and avoid a revision being merged more than once. This makes your life much simpler as you can simply merge the entire revision range each time and know that only new revisions will actually be merged.

Branch management is important. If you want to keep this branch up to date with the trunk, you should be sure to merge often so that the branch and trunk do not drift too far apart. Of course, you should still avoid repeated merging of changes, as explained above.

### *[https://tse1.mm.bing.net/th?&id=OIP.M0347418a11c15c8213c3fb8473f9ed9fo0&w=295&h=300&c=0&pid=1.9&rs=0&p=0&r=0](https://www.bing.com/images/search?q=important&view=detailv2&&id=77B981E696D9E23B52FDAB9B7E2090F2669FF916&selectedIndex=71&ccid=A0dBihHB&simid=608022213276732253&thid=OIP.M0347418a11c15c8213c3fb8473f9ed9fo0)**Tip*

[](https://www.bing.com/images/search?q=important&view=detailv2&&id=FE726C290560E017C7D7CD5523EDDAF9E947DAEB&selectedIndex=1&ccid=1TZo0V2n&simid=608053965959662141&thid=OIP.Md53668d15da7c7ebd79f70e687986f8co0)If you have just merged a feature branch back into the trunk, the trunk now contains all the new feature code, and the branch is obsolete. You can now delete it from the repository if required.

### *Important*

Subversion can't merge a file with a folder and vice versa - only folders to folders and files to files. If you click on a file and open up the merge dialog, then you have to give a path to a file in that dialog. If you select a folder and bring up the dialog, then you must specify a folder URL for the merge.

## Merge Tracking

Subversion 1.5 introduced facilities for merge tracking. When you merge changes from one tree into another, the revision numbers merged are stored and this information can be used for several different purposes.

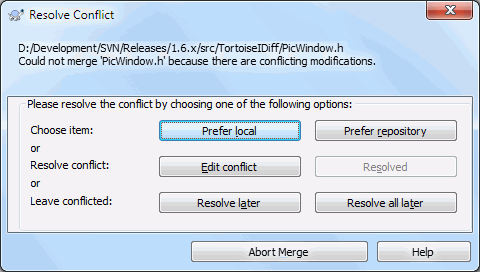
* You can avoid the danger of merging the same revision twice (repeated merge problem). Once a revision is marked as having been merged, future merges which include that revision in the range will skip over it.
* When you merge a branch back into trunk, the log dialog can show you the branch commits as part of the trunk log, giving better traceability of changes.
* When you show the log dialog from within the merge dialog, revisions already merged are shown in grey.
* When showing blame information for a file, you can choose to show the original author of merged revisions, rather than the person who did the merge.
* You can mark revisions as do not merge by including them in the list of merged revisions without actually doing the merge.

Merge tracking information is stored in the svn:mergeinfo property by the client when it performs a merge. When the merge is committed the server stores that information in a database, and when you request merge, log or blame information, the server can respond appropriately. For the system to work properly you must ensure that the server, the repository and all clients are upgraded. Earlier clients will not store the svn:mergeinfo property and earlier servers will not provide the information requested by new clients.

## Handling Conflicts during Merge

Merging does not always go smoothly. Sometimes there is a conflict, and if you are merging multiple ranges, you generally want to resolve the conflict before merging of the next range starts. TortoiseSVN helps you through this process by showing the merge conflict callback dialog.

**Figure 4.56. The Merge Conflict Callback Dialog**



It is likely that some of the changes will have merged smoothly, while other local changes conflict with changes already committed to the repository. All changes which can be merged are merged. The Merge Conflict Callback dialog gives you three different ways of handling the lines which are in conflict.

1. If you are merging text files then these first two buttons allow you to merge non-conflicting lines as normal and always prefer one version where there are conflicts. Choosing Prefer local will select your local version in every conflict, i.e. it will prefer what was already there before the merge over the incoming change from the merge source. Likewise, Prefer repository will select the repository changes in every conflict, i.e. it will prefer the incoming changes from the merge source over what was already in your working copy. This sounds easy, but the conflicts often cover more lines than you think they will and you may get unexpected results.

If your merge includes binary files, merging of conflicts in those is not possible in a line-by-line mode. A conflict in a binary file always refers to the complete file. Use Prefer local to select the local version as it was in your working copy prior to the merge, or Prefer repository to select the incoming file from the merge source in the repository.

1. Normally you will want to look at the conflicts and resolve them yourself. In that case, choose the **Edit Conflict** which will start up your merge tool. When you are satisfied with the result, click **Resolved**.
2. The last option is to postpone resolution and continue with merging. You can choose to do that for the current conflicted file, or for all files in the rest of the merge. However, if there are further changes in that file, it will not be possible to complete the merge.

If you do not want to use this interactive callback, there is a checkbox in the merge progress dialog Merge non-interactive. If this is set for a merge and the merge would result in a conflict, the file is marked as in conflict and the merge goes on. You will have to resolve the conflicts after the whole merge is finished. If it is not set, then before a file is marked as conflicted you get the chance to resolve the conflict during the merge. This has the advantage that if a file gets multiple merges (multiple revisions apply a change to that file), subsequent merges might succeed depending on which lines are affected. But of course you can't walk away to get a coffee while the merge is running ;)

## Feature Branch Maintenance

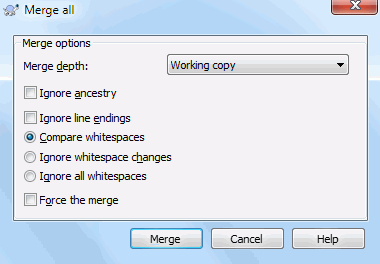
When you develop a new feature on a separate branch it is a good idea to work out a policy for re-integration when the feature is complete. If other work is going on in trunk at the same time you may find that the differences become significant over time, and merging back becomes a nightmare.

If the feature is relatively simple and development will not take long then you can adopt a simple approach, which is to keep the branch entirely separate until the feature is complete, then merge the branch changes back into trunk. In the merge wizard this would be a simple Merge a range of revisions, with the revision range being the revision span of the branch.

If the feature is going to take longer and you need to account for changes in trunk, then you need to keep the branch synchronised. This simply means that periodically you merge trunk changes into the branch, so that the branch contains all the trunk changes plus the new feature. The synchronisation process uses Merge a range of revisions. When the feature is complete then you can merge it back to trunkusing either Reintegrate a branch or Merge two different trees.

Another (fast) way to merge all changes from trunk to the feature branch is to use the **TortoiseSVN** → **Merge all...** from the extended context menu (hold down the **Shift** key while you right click on the file).

**Figure 4.57. The Merge-All Dialog**



This dialog is very easy. All you have to do is set the options for the merge, as described in [the section called “Merge Options”](https://tortoisesvn.net/docs/nightly/TortoiseSVN_en/tsvn-dug-merge.html#tsvn-dug-merge-options). The rest is done by TortoiseSVN automatically using merge tracking.

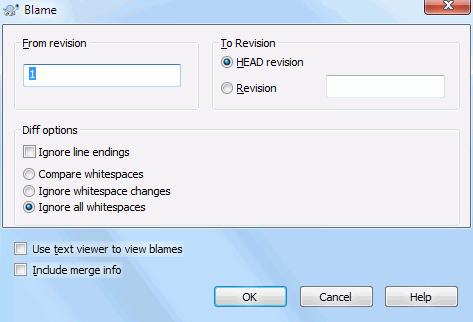
# WHO CHANGED WHICH LINE?

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.

## Blame for Files

**Figure 4.61. The Annotate / Blame Dialog**



If you're not interested in changes from earlier revisions you can set the revision from which the blame should start. Set this to 1, if you want the blame for every revision.

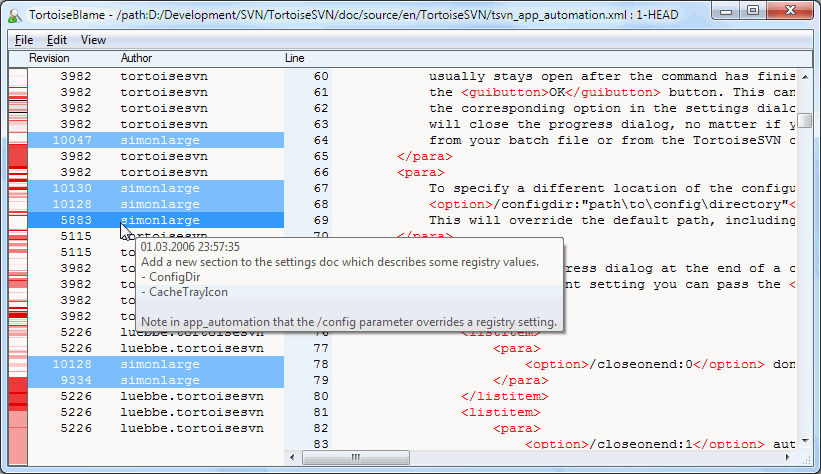
By default the blame file is viewed using TortoiseBlame, which highlights the different revisions to make it easier to read. If you wish to print or edit the blame file, select Use Text viewer to view blames.

You can specify the way that line ending and whitespace changes are handled. These options are described in [the section called “Line-end and Whitespace Options”](https://tortoisesvn.net/docs/nightly/TortoiseSVN_en/tsvn-dug-diff.html#tsvn-dug-diff-eol-whitespace). The default behaviour is to treat all whitespace and line-end differences as real changes, but if you want to ignore an indentation change and find the original author, you can choose an appropriate option here.

You can include merge information as well if you wish, although this option can take considerably longer to retrieve from the server. When lines are merged from another source, the blame information shows the revision the change was made in the original source as well as the revision when it was merged into this file.

Once you press **OK** TortoiseSVN starts retrieving the data to create the blame file. Once the blame process has finished the result is written into a temporary file and you can view the results.

**Figure 4.62. TortoiseBlame**



TortoiseBlame, which is included with TortoiseSVN, makes the blame file easier to read. When you hover the mouse over a line in the blame info column, all lines with the same revision are shown with a darker background. Lines from other revisions which were changed by the same author are shown with a light background. The colouring may not work as clearly if you have your display set to 256 colour mode.

If you **left click** on a line, all lines with the same revision are highlighted, and lines from other revisions by the same author are highlighted in a lighter colour. This highlighting is sticky, allowing you to move the mouse without losing the highlights. Click on that revision again to turn off highlighting.

The revision comments (log message) are shown in a hint box whenever the mouse hovers over the blame info column. If you want to copy the log message for that revision, use the context menu which appears when you right click on the blame info column.

You can search within the Blame report using **Edit** → **Find...**. This allows you to search for revision numbers, authors and the content of the file itself. Log messages are not included in the search - you should use the Log Dialog to search those.

You can also jump to a specific line number using **Edit** → **Go To Line...**.

When the mouse is over the blame info columns, a context menu is available which helps with comparing revisions and examining history, using the revision number of the line under the mouse as a reference. **Context menu** → **Blame previous revision** generates a blame report for the same file, but using the previous revision as the upper limit. This gives you the blame report for the state of the file just before the line you are looking at was last changed. **Context menu** → **Show changes** starts your diff viewer, showing you what changed in the referenced revision. **Context menu** → **Show log** displays the revision log dialog starting with the referenced revision.

If you need a better visual indicator of where the oldest and newest changes are, select **View** → **Color age of lines**. This will use a colour gradient to show newer lines in red and older lines in blue. The default colouring is quite light, but you can change it using the TortoiseBlame settings.

If you are using Merge Tracking and you requested merge info when starting the blame, merged lines are shown slightly differently. Where a line has changed as a result of merging from another path, TortoiseBlame will show the revision and author of the last change in the original file rather than the revision where the merge took place. These lines are indicated by showing the revision and author in italics. The revision where the merge took place is shown separately in the tooltip when you hover the mouse over the blame info columns. If you do not want merged lines shown in this way, uncheck the Include merge info checkbox when starting the blame.

If you want to see the paths involved in the merge, select **View** → **Merge paths**. This shows the path where the line was last changed, excluding changes resulting from a merge.

The revision shown in the blame information represents the last revision where the content of that line changed. If the file was created by copying another file, then until you change a line, its blame revision will show the last change in the original source file, not the revision where the copy was made. This also applies to the paths shown with merge info. The path shows the repository location where the last change was made to that line.

The settings for TortoiseBlame can be accessed using **TortoiseSVN** → **Settings...** on the TortoiseBlame tab.

## 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 the last person to change each line. Sometimes you want to know what change was made, as well as who made it. If you right click on a line in TortoiseBlame you have a context menu item to show the changes made in that revision. But if you want to see the changes and the blame information simultaneously then you need a combination of the diff and blame reports.

The revision log dialog includes several options which allow you to do this.

Blame Revisions

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

Blame Changes

Select one revision in the top pane, then pick one file in the bottom pane and select **Context menu** → **Blame changes**. This will fetch the blame data for the selected revision and the previous revision, then use the diff viewer to compare the two blame files.

Compare and Blame with Working BASE

Show the log for a single file, and in the top pane, select a single revision, then select **Context menu** → **Compare and Blame with Working BASE**. This will fetch the blame data for the selected revision, and for the file in the working BASE, then use the diff viewer to compare the two blame files.