SmartCVS Tutorial

syntevo GmbH, www.syntevo.com

January 2010

Contents

Chapter 1

Introduction

CVS (Concurrent Versions System; http://www.nongnu.org/cvs/) is one of the most widespread version control systems. Because its client-server-protocol and the source code of the command line tool are open, a couple of CVS clients are available for almost every platform. SmartCVS is such a CVS client, but in contrast with other CVS clients which mostly are graphical front-ends to the command line CVS, SmartCVS tries to go a step further. As its name claims, it builds a couple of smart features on top of the core CVS functionality to make the daily work with CVS as easy and comfortable as possible, even for CVS novices.

SmartCVS does not limit you to one platform, it runs on any platform where a Java Runtime Environment (JRE) 1.4.1 or higher is available, including all newer Windows versions, Linux, Solaris, Mac OS X, OS/2.

This tutorial shows you how to use SmartCVS for your daily work with CVS by example of a simple Java project. It should be sufficient, if you have a little experience with other version control systems. Nevertheless, even if you already have used CVS in the past, this tutorial might contain some useful tips for you.

Chapter 2

Before You Start

2.1 Download and Installation

You can download SmartCVS from http://www.syntevo.com/smartcvs/download.html. Depending on your operating system and whether you already have a JRE 1.4.1 or higher installed, choose the appropriate bundle for your system configuration.

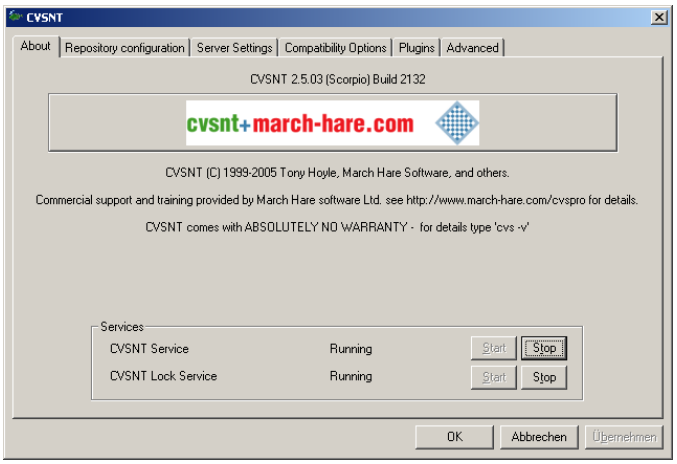
The installation is straight-forward. On Windows unpack the installer bundle, run the containing `setup.exe` and follow the instructions of the installer. On Mac OS X just unpack the bundle by double clicking on it and move the `SmartCVS.app` to the `/Application` directory. On the other systems, unpack the bundle into a directory of your choice. If you also have older JRE versions installed, it might be necessary to set the path to the JRE 1.4.1 or higher (recommended version: 1.6) in the corresponding launcher-script (`smartcvs.sh` or `smartcvs.cmd`) which are located in the `bin` directory. Alternatively, for the Unix/Linux-systems you can set the environment variable SMARTCVS_JAVA_HOME pointing to the JRE.

2.2 CVS Server

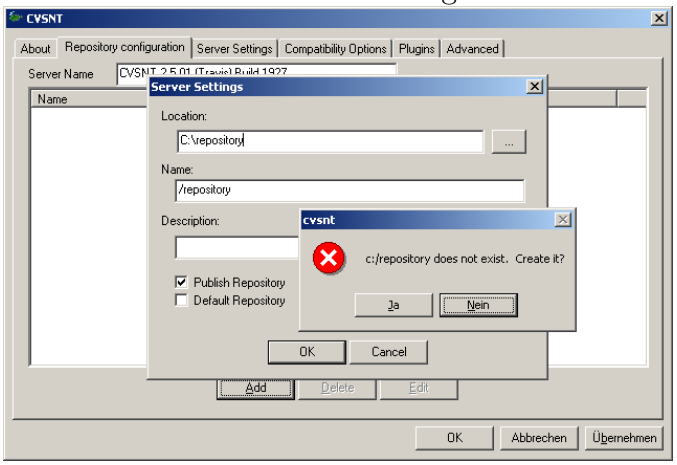
For this tutorial you need a working CVS server. It is highly recommended to use a separate installation for this tutorial as the tutorial requires some changes to the user accounts of your CVS repository.

2.2.1 CVSNT

If you are on Windows, you should download CVSNT from http://www.cvsnt.org. Install it with the default values (**Typical installation** and installing all services). Then open the **Service control panel** from the **CVSNT** start menu group.



Switch to the **Repository configuration** tab and click the **Add** button. In the **Location** input field enter the path to the new repository, e.g. `C:\repository`. The **Name** input field will be filled automatically while typing. It finally should contain the `/repository` path. Leave the checkboxes unchanged and click **OK**.



In the occuring message box, choose **Yes** to create the repository. Click **Apply** and switch back to the **About** tab.

Click **Stop** right beside the **CVSNT Service**, wait until `Stopped` occurs and click the **Start** button. After a few moments the CVSNT Service should be `Running` again. This stop-start-sequence is necessary, because otherwise the CVSNT service does not know about the changed repository configuration.

2.3 Setup

When starting SmartCVS the first time, you need to agree to its license agreement and select what program edition you plan to use. This tutorial assumes, that you are using the Professional edition of SmartCVS.

After starting SmartCVS, close the **Tip Of The Day** and **Welcome to SmartCVS** dialogs.

Chapter 3

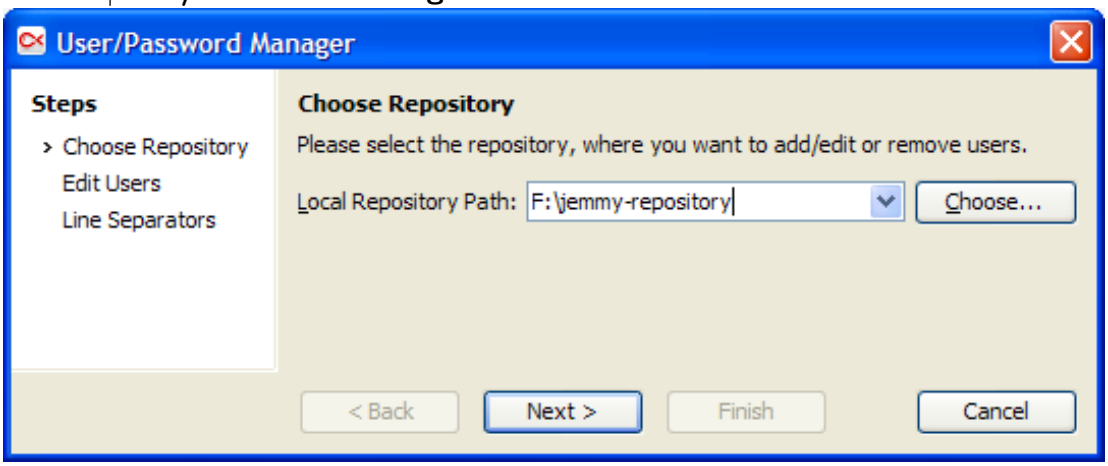
Susan and Mike

Susan and Mike are the two virtual test users who will work concurrently on the tutorial project. Don't worry, you can perform the tutorial as a single person, but with the help of two different user accounts.

CVS is a client-server-system. Each user has its own working copy (*sandbox*). The CVS server maintains the central database for the project(s) and keeps track of the file modifications. This central database is called *repository* for a CVS system. Beside the projects, it contains some administration files which are located in the `CVSROOT` directory. For the pserver access method which we will use in this tutorial, one important administration file is `CVSROOT/passwd` where all user accounts, who are allowed to access the repository, are listed as well as their encrypted passwords. To create new user accounts, this file must be edited. You can do this either by using command line tools or you can use SmartCVS' built-in User/Password Manager, as described below.

3.1 Creating User Accounts

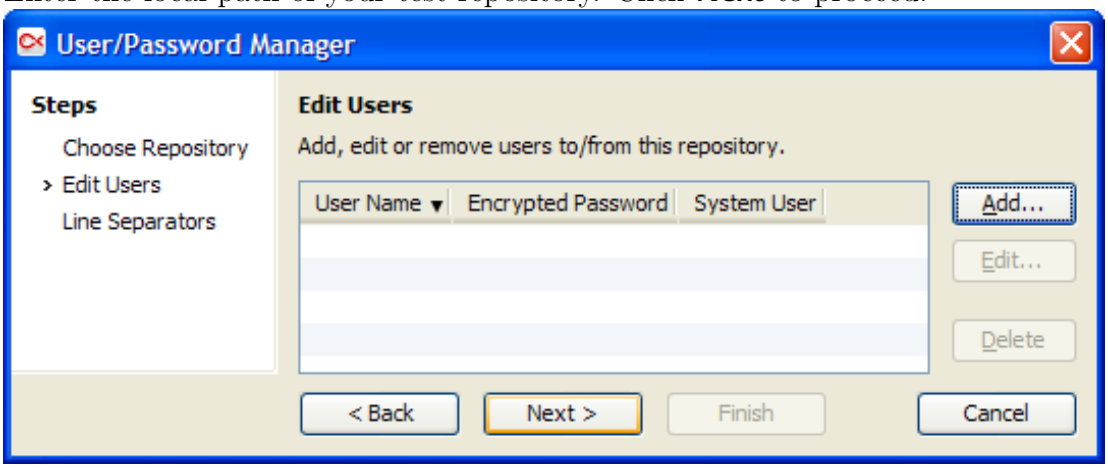
To create the two user accounts 'susan' and 'mike' (we use lowercase user accounts to distinguish from names of the virtual users), open the User/Password Manager by clicking the **Admin|User/Password Manager** menu item.



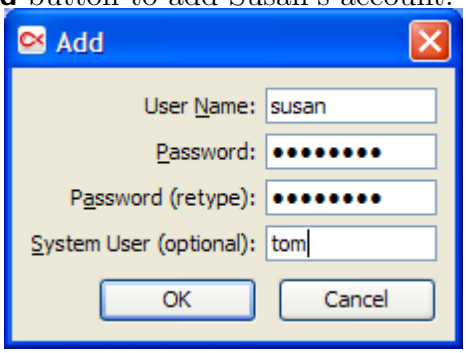
First, SmartCVS needs to know in which repository the user accounts should be created. Because a user account is necessary to log into the CVS server, the modifications

must be done directly in the repository without a client-server connection. Therefore the repository must be available in the local file system. If your CVS server is on a different machine, mount the repository directory, so you are able to access it as a directory in your local file system.

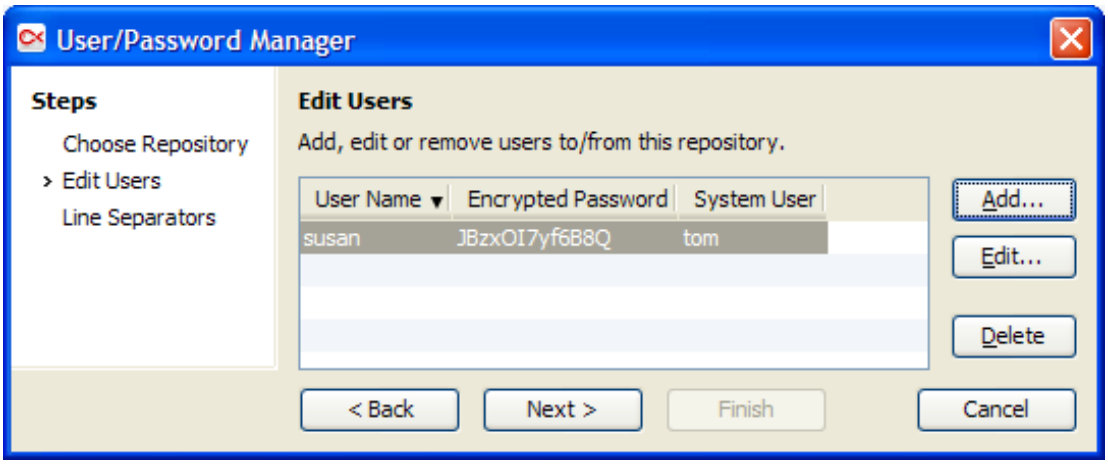
Enter the local path of your test repository. Click **Next** to proceed.



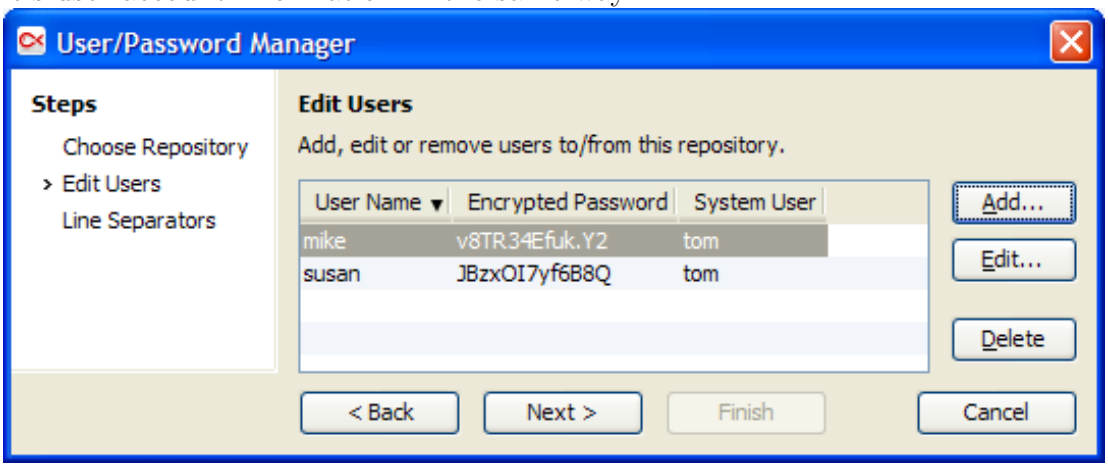
On this page you define the user accounts who can access this repository. Click the **Add** button to add Susan's account.



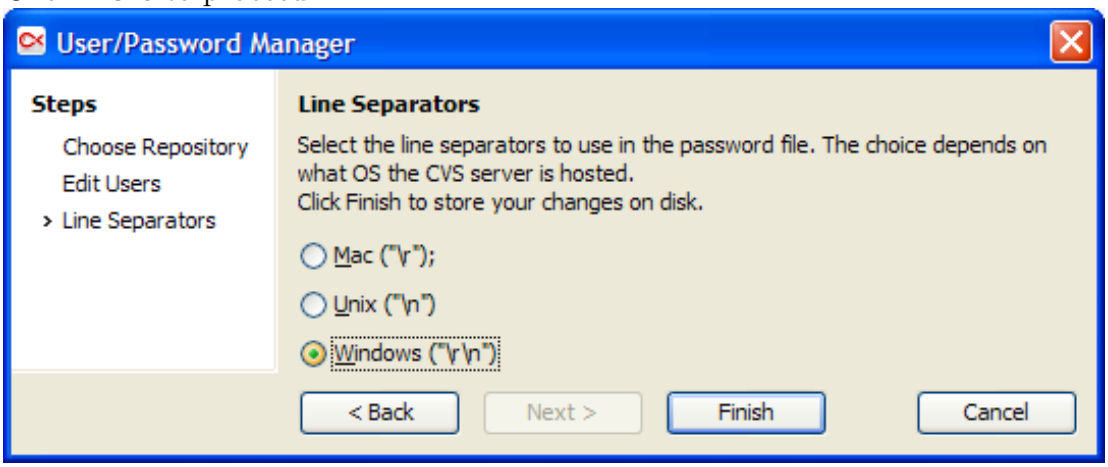
Enter `susan` as the user name and a password for her account. Retype the password to prevent possible typing-errors. For some CVS servers, e.g. newer versions of CVSNT, it is necessary to specify a system user which needs write access in the repository file system of the CVS server. If you are unsure, contact the documentation of your CVS server. The system user must be a user account from the CVS server's operating system. Click **OK** to create the new user account.



The passwords will be stored encrypted on the CVS server. Don't wonder, if the same password produces different encrypted passwords – this is intended behaviour. Now enter Mike's user account information in the same way.



Click **Next** to proceed.



Specify the line separators for writing the `CVSROOT/passwd`-file. This depends on what operating system the CVS server is running. For the tutorial, a CVSNT server on Windows was used.

Finally, to write the new user accounts to disk, click **Finish**.

Chapter 4

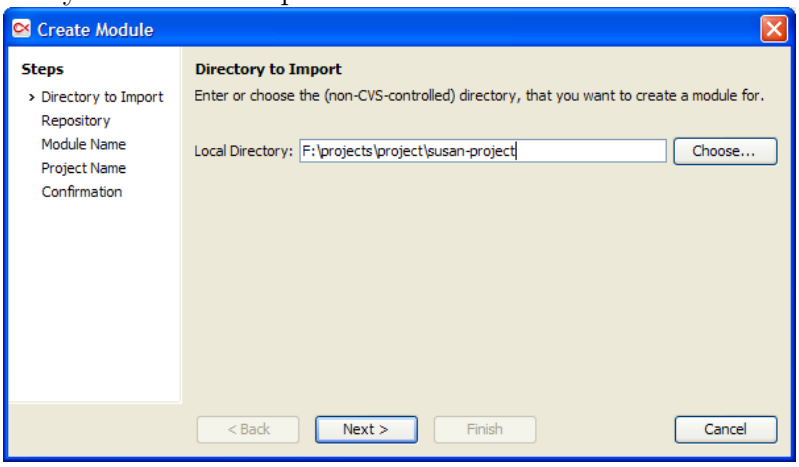
Filling the Repository

For the tutorial, Susan has started coding a 'Text-Viewer'. The files she has created in her project are listed in Section 9.1. You may create them with your text-editor, but you might unpack Susan's whole project from the zip-file `susans-first-version.zip` into Susan's working directory.

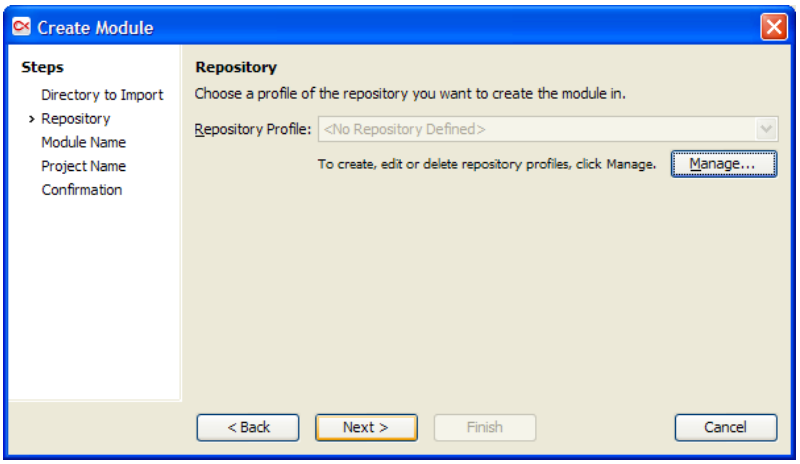
Now Susan wants to put her source files under CVS control. Therefore she will create a new *module* in the repository.

4.1 Creating a CVS Module

Click the **Project|Create Module** menu item. This wizard lets you select the local root directory which will be put under CVS control in order to create a new module.

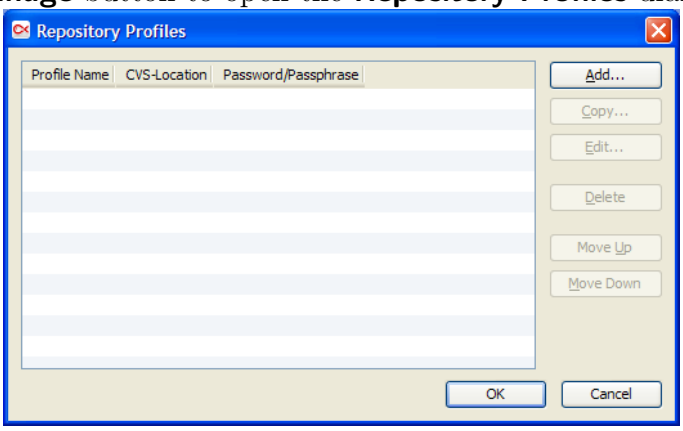


Enter or select the path of the directory which contains Susan's project and shall be put under CVS control or choose it by clicking the **Choose** button right to the input field. Click **Next** to proceed.

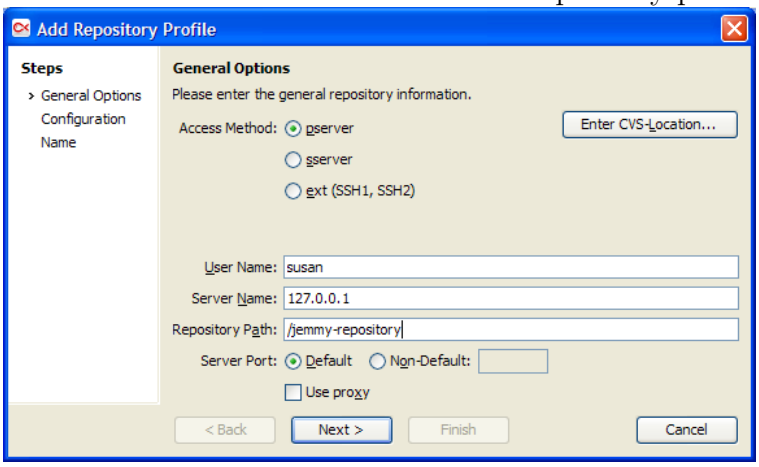


SmartCVS needs to know in which repository the new module shall be created. It maintains a list of *Repository Profiles* which specify the location of the repository and additional parameters about how to connect to it. Among others, this includes the user account.

For the first start of SmartCVS the list of repository profiles is empty, so click on the **Manage** button to open the **Repository Profiles** dialog.



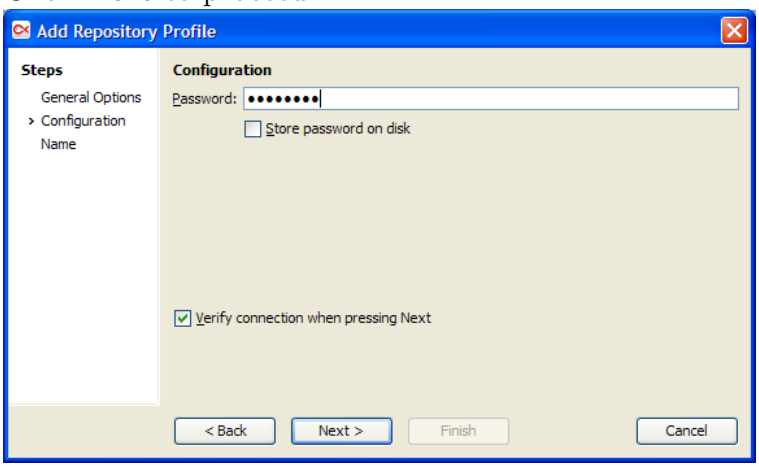
Click the **Add** button to create a new repository profile.



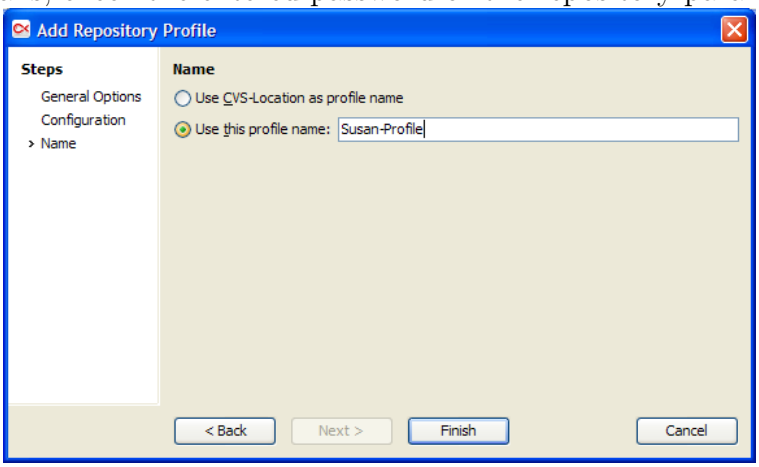
Keep the access method **pserver** selected. Enter `susan` in the **User Name** input field, the name or IP-address of your CVS server in the **Server Name** input field and the path of the repository in the **Repository Path** input field.

Note	If you are using CVSNT on Windows as described in Section 2.2.1, use the 'Name' of the repository, not the 'Location' with the drive letter.

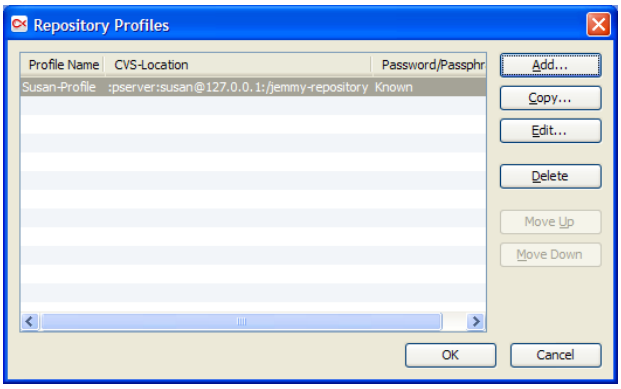
Click **Next** to proceed.



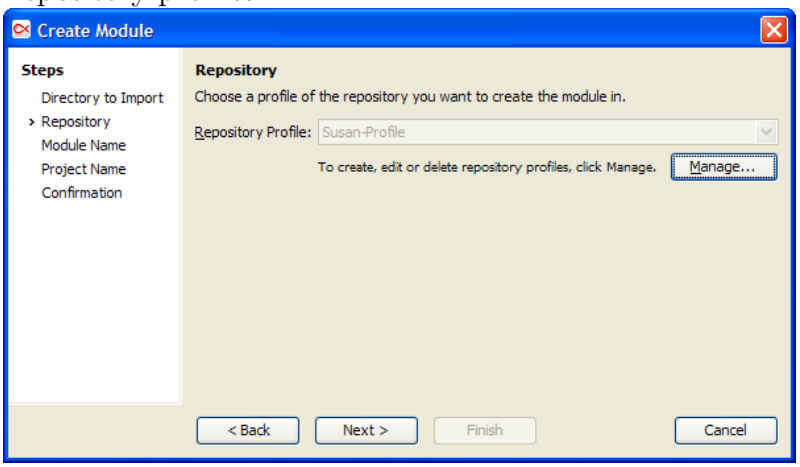
Enter Susan's password in the **Password** inputfield. If you don't want to be asked for the password the next time, select the option **Store password on disk**. Keep the option **Verify connection when pressing Next** selected and click **Next**. This will try to connect to the repository using the entered password. If it succeeds, the next page will occur. If it fails, check the entered password or the repository parameters on the previous page.



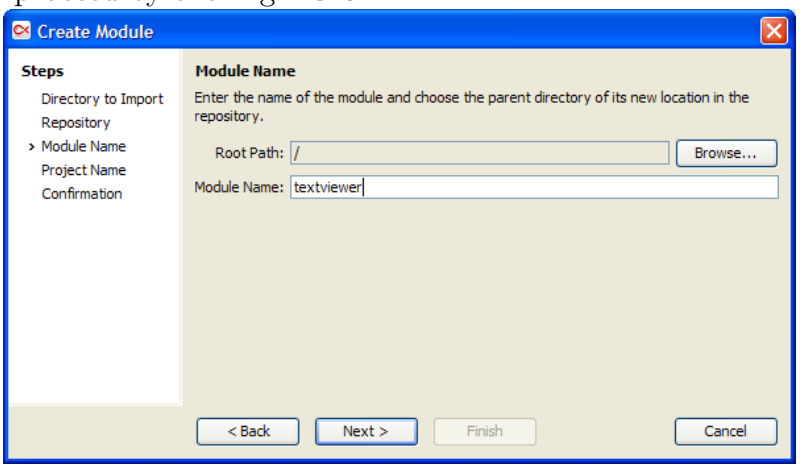
Because the default names of the repository profiles look a little bit cryptic for new CVS users, select **Use this profile name** and enter `Susan-Profile` in the corresponding input field. Click **Finish** to finally create the new repository profile.



The repository profile was created and verified. Click **OK** to confirm your changes of the repository profiles.

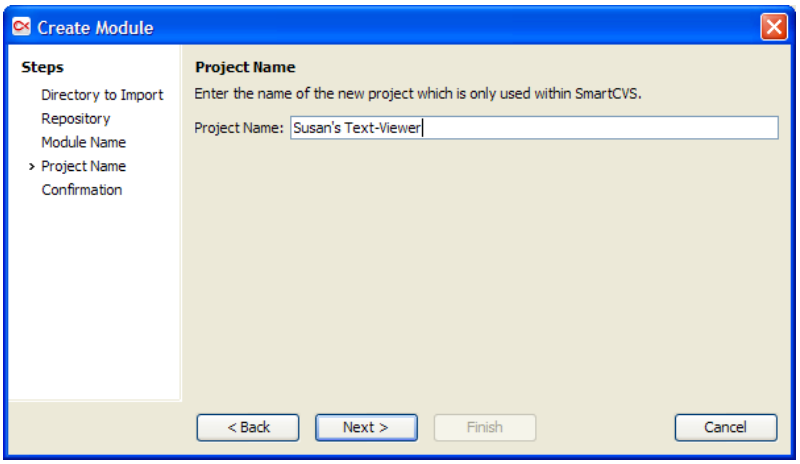


SmartCVS automatically chooses the newly created repository profile, so you simply can proceed by clicking **Next**.



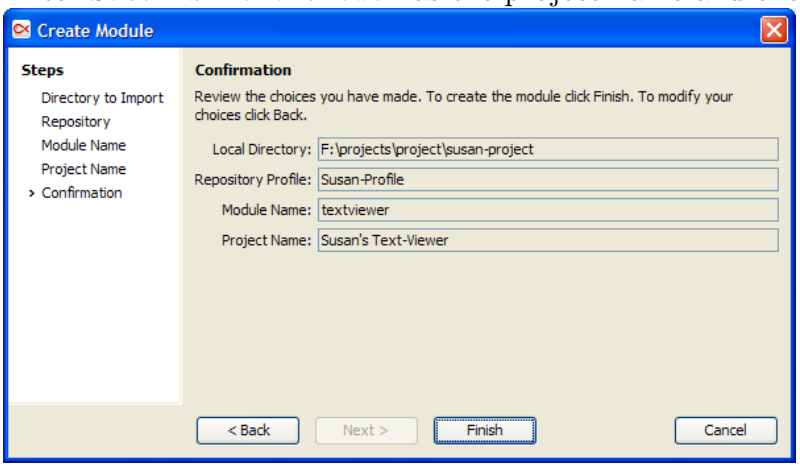
Next you need to specify where the new module should be located in the repository and how it should be named. SmartCVS suggests the name of the selected directory.

Because in this case the suggested module name is inappropriate, change it to `textviewer` and click **Next**.



When creating a new module, SmartCVS automatically creates a corresponding SmartCVS-project with the selected directory as project root.

Enter `Susan's Text-Viewer` as the project name and click **Next** to proceed.

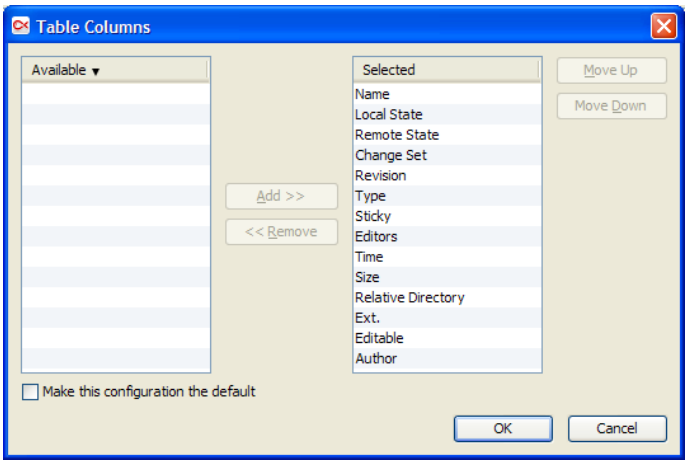


Before the module actually is created, you can review your choices. If everything is fine, click **Finish** to confirm to create the module. SmartCVS now should open the newly created SmartCVS-project.

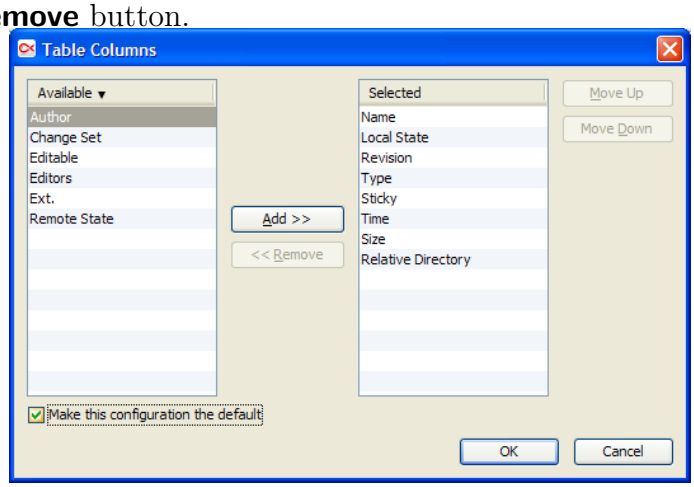
After the module has been created, the corresponding local directory has been put under CVS control. Files and subdirectories have not been regarded by this operation. You'll see soon, how to put the files under CVS control.

4.2 Define the Visible Table Columns

Because by default too much table columns are displayed, we want to hide the ones, which are not needed for this tutorial. Click the **View|Table Columns** menu item.

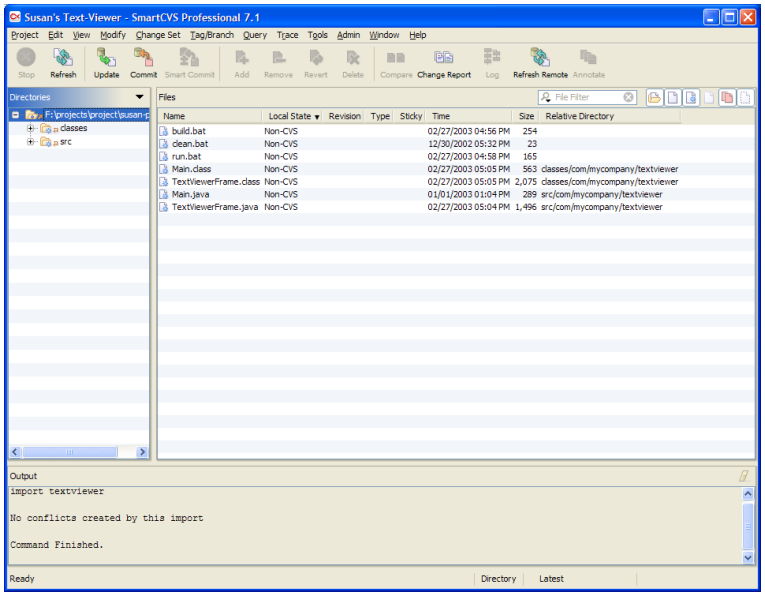


Select the table columns **Remote State**, **Change Set**, **Editors**, **Ext.**, **Editable** and **Author** in the right **Selected** list and drag them to the left **Available** list or click the **Remove** button.



Select the option **Make this configuration the default** and click **OK**.

4.3 Adding and Committing Files



SmartCVS' main window consists of the following major parts:

- The directory tree on the left shows the directory structure of the project.
- The file table on the right shows the files in the selected directory. By default, the files from subdirectories are shown as well. The values in the table column **Relative Directory** indicate the subdirectories of the files.
- The Output view at the bottom shows the output of the CVS commands.

The small Java project Susan created contains two main directories, `src` and `classes`. The first one contains subdirectories (which are called *packages* in Java) with the source files and the latter one the generated `class`-files. In the project's root directory you'll find the files required for building and running the application. For simplicity, batch files are used instead of a makefile or an ANT build file (http://www.apache.org/ant/).

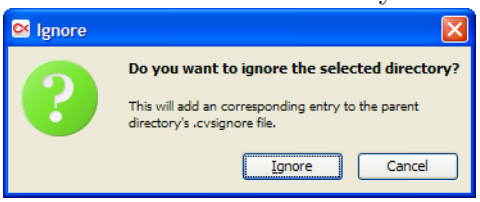
Tip	To run the application yourself, you can use SmartCVS' JRE. Change the environment variable `JAVA_HOME` in the file `run.bat` to point to the JRE and launch this batch file. If you want to compile the source files yourself, you need to have a JDK (version 1.3 or higher; see http://java.sun.com for downloads) installed on your system. Edit the batch files, so their `JAVA_HOME` environment variable points to the JDK.

Except for the root directory, Susan's project is not yet under CVS control. Hence the files and directories are treated as new indicated by an icon with a blue star. Because by default SmartCVS shows you the files from the selected directory and its subdirectories, it is hard to accidently forget some files.

4.3.1 Ignoring Files

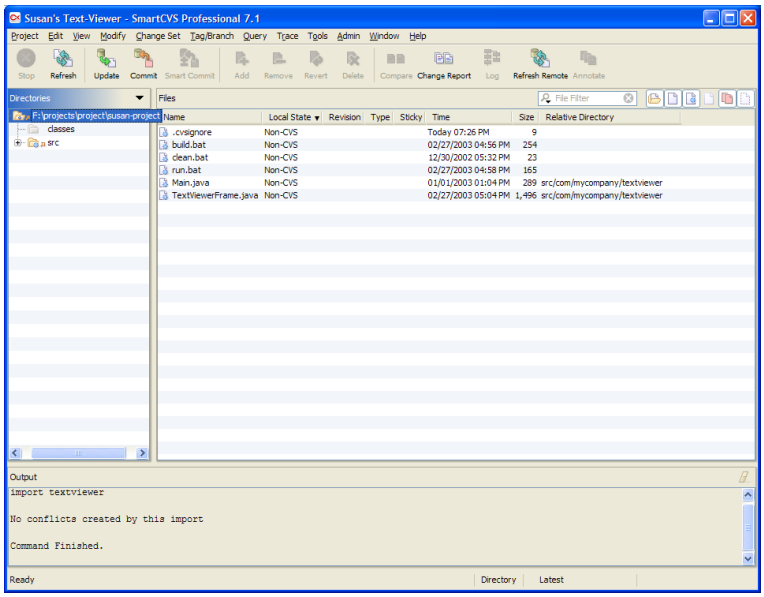
The `classes` directory is created by the `build.bat` batch file and the containing `class`-files can be generated from the source files. Therefore it does not make sense to put them under CVS control. You can achieve this by marking the `classes` directory to be ignored.

Select the `classes` directory and click the **Modify|Ignore** menu item.



Click **Ignore** to ignore the directory, then select the project's root directory.

Note	Because you invoked this action on a directory, you don't have any changeable options. If you would have selected one or more files, you would have the option to ignore either by file patterns or by file names. Furthermore you would have the choice, whether all directories of all projects should be affected or just the directory of the corresponding file. Directories can only be ignored locally (in the parent directory where they are located).

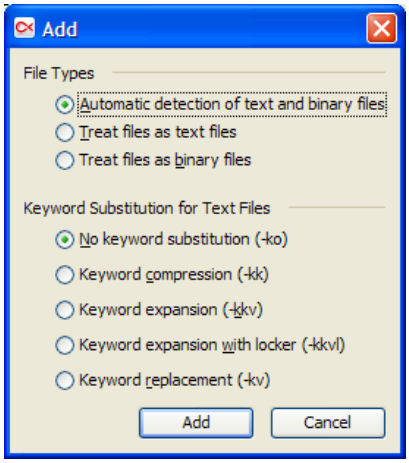


The `classes` directory is now displayed grayed to indicate, that it is ignored. The `class` files aren't listed any more, but a new file `.cvsignore` was created in the root directory. This is the file, where CVS stores the information, what files or directories should be ignored in the local directory. It's common practice to put this file also under CVS control, because other users, like Mike, need to have the same files and directories ignored.

4.3.2 Adding Files

Select all files in the file table and click the **Add** tool bar button.

Note	For nearly all SmartCVS operations it is important, which directory and files are selected and whether the focus is in the directory tree or the file table. There is a general rule: if the focus is in the directory tree, the operation is performed on the selected directory (the selected files don't matter). If the focus is in the file table, the operation is performed on the selected files. SmartCVS highlights the currently focussed component with a blue background in its title bar.

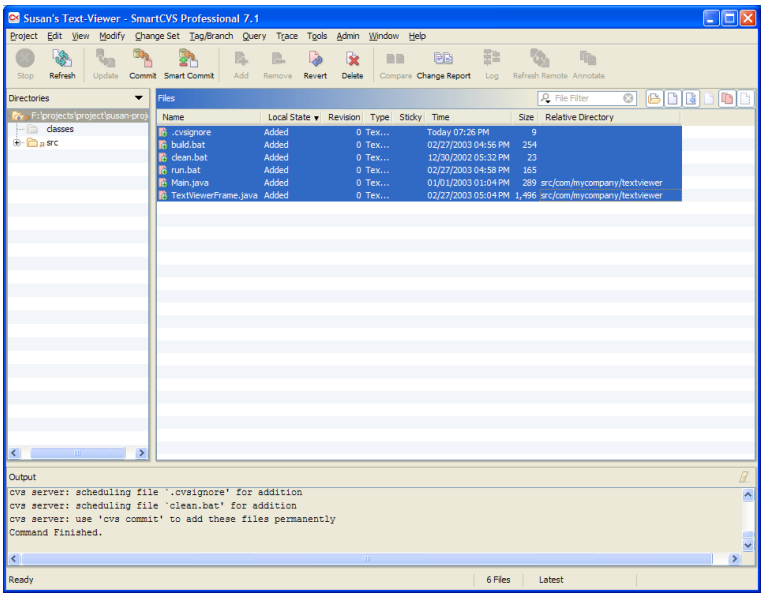


With the **File Types** options you can decide, whether the file types of the selected files should be detected automatically or whether SmartCVS should be forced to treat them as text or binary files.

Note	The distinction between text and binary files is essential for CVS, because line separators are converted in text files, which would corrupt binary data. On the other hand, binary files cannot be merged, because the CVS server does not know about how to merge the binary content and hence treats them as a black box.

With the **Keyword Substitution for Text Files** options you can decide, whether or how special keywords in text files, like *$Author$*, *$Date$* or *$Revision$*, should be expanded (e.g., to `susan`, `2003-01-10` or `1.6`) when the files are *committed*.

Keep the options **Automatic detection of text and binary files** and **No keyword substitution (-ko)** selected and click **Add** to add the files.



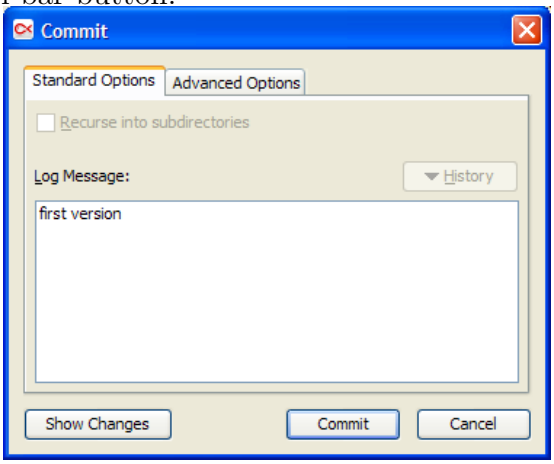
The file icons change to red files with a green plus-sign and the values in the **Local States** table column change to `Added` to indicate files, that are just marked for adding, but need to be *committed* to be actually stored in the repository.

Note	You don't need to add all the directories, where the files are located in, explicitly like with other CVS clients. SmartCVS puts them automatically under CVS control when files from these directories are added.

4.3.3 Committing Changes

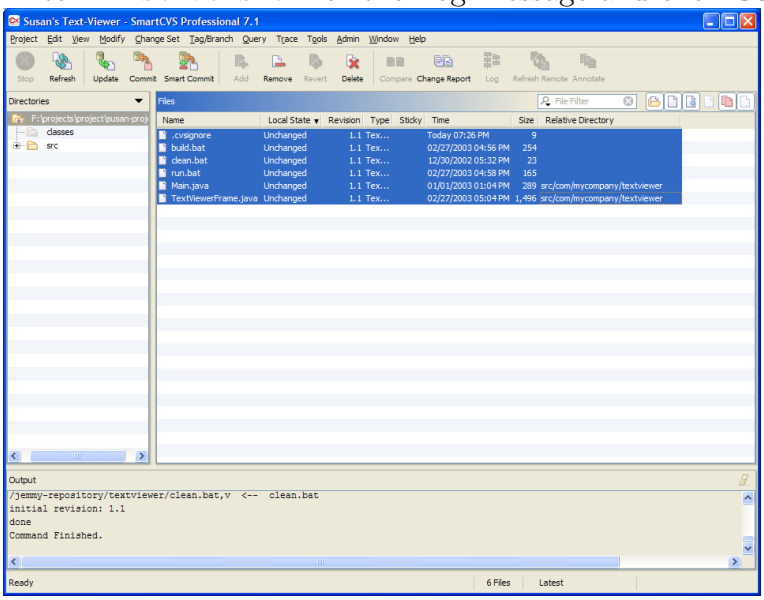
Currently the files are only *marked* for adding, but aren't stored in the repository yet. To store added files or modifications of already CVS controlled files in the repository, you need to *commit* them.

Keep the added files selected and click the **Modify**|**Commit** menu item or the **Commit** tool bar button.



When committing, you have the option to enter a log message. This helps you to better keep track, why a commit was made, especially why a revision was created.

Enter `first version` for the Log Message and click **Commit**.



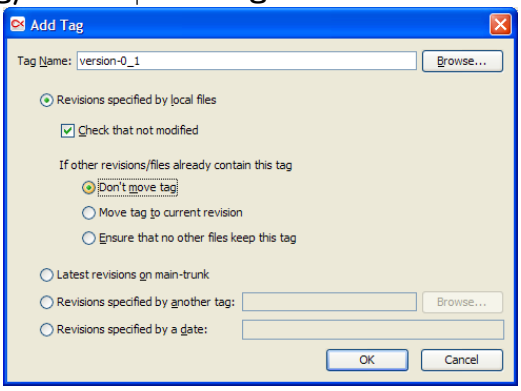
The file icons change to white to indicate, that these local files are under CVS control and don't differ from their revisions in the repository.

Note	The previous statement is only valid immediately after running a commit, update or checkout command or after refreshing the Local State. A file icon corresponds directly to the Local State of that file. Because the Local State is detected by the file modification time, the icon does not indicate, whether another user changed the file in the repository. To detect changes in the repository use the **Query**|**Refresh Remote State** menu item.

4.4 Marking Versions

Now Susan is satisfied with her work and wants to make a beta-release. She decides to mark all files of the project with a specific label, to *add a tag* to the files. This helps in the future to reconstruct an old project state from the repository.

To add a tag to the whole project, select the project root directory and click the **Tag/Branch**|**Add Tag** menu item or the **Add Tag** tool bar button.



Enter `version-0_1` as the Tag Name and click **OK** to start adding the tag to the selected files.

Note	You cannot enter an arbitrary name as a tag name, because some characters like spaces and periods aren't allowed by CVS. It is common, to use the minus or underscore instead.

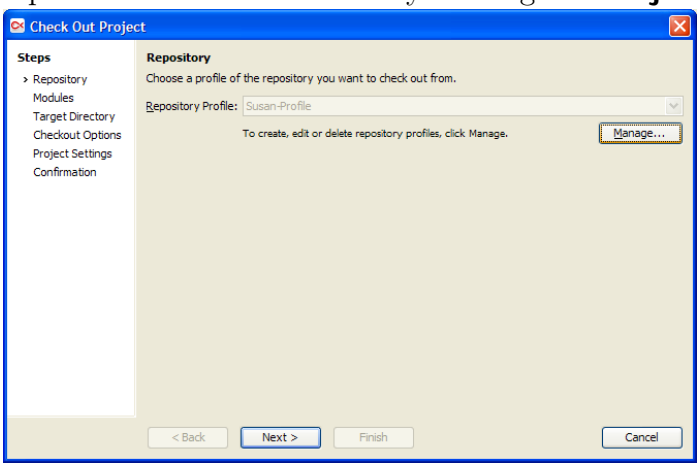
Chapter 5

Mike Improves the Project

5.1 Checking Out Mike's Working Copy

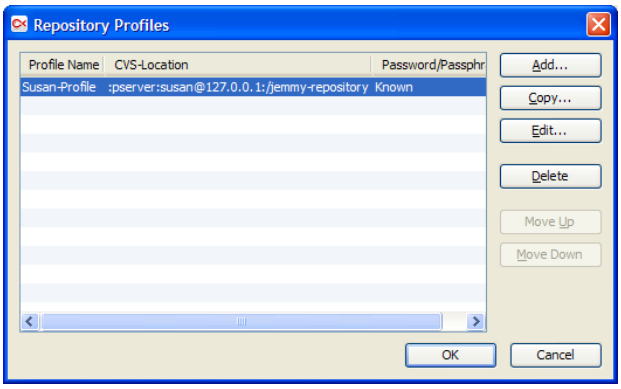
Because Mike wants to change some things on the Text-Viewer project, he needs to *check out* his own working copy. For easier handling, you can do it in the same SmartCVS instance and not on a different machine as usually.

Open the Checkout Wizard by clicking the **Project**|**Check Out** menu item.

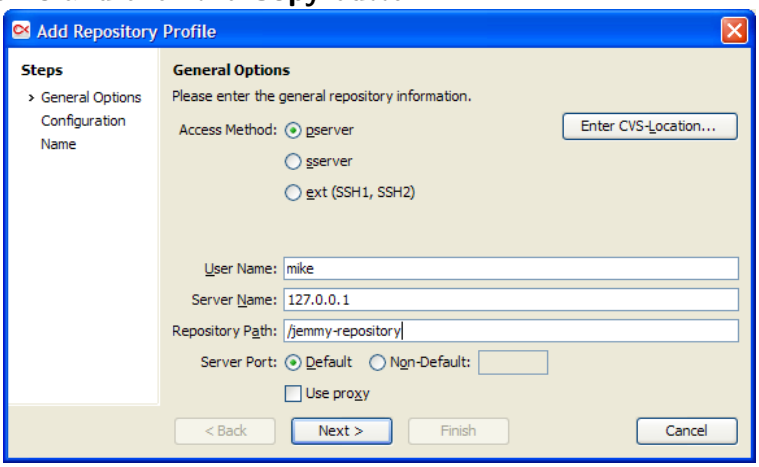


First, SmartCVS needs to know, where to check out from and what user account to use. Again, this is determined by repository profiles. Currently, there is only Susan's repository profile defined. To check out with the user account `mike`, you need to create a new repository profile.

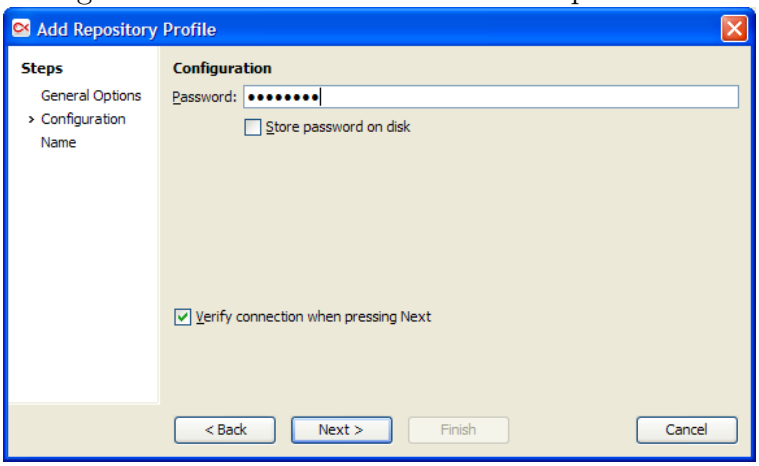
Click **Manage** to open the list of repository profiles.



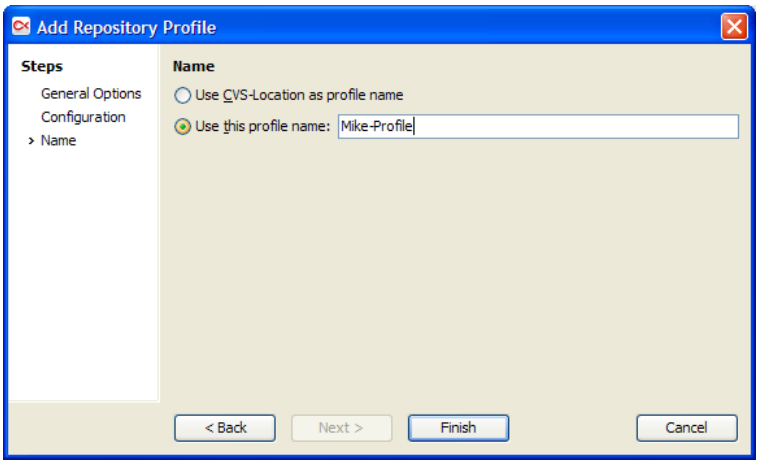
You could add Mike's profile as done with Susan's, but there is an easier solution, where you don't need to enter all the repository parameters again. Select the **Susan Profile** and click the **Copy** button.



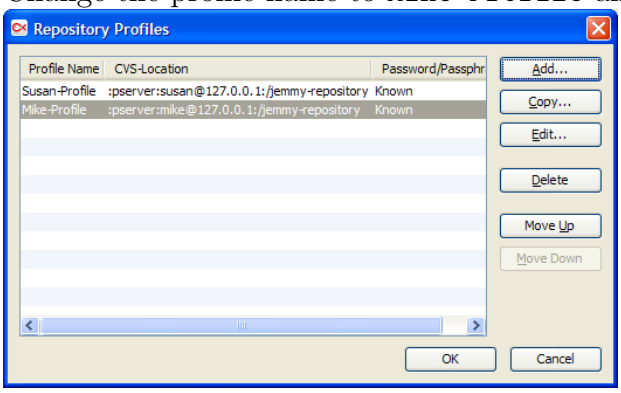
Change the content of the **User Name** input field to `mike` and click **Next**.



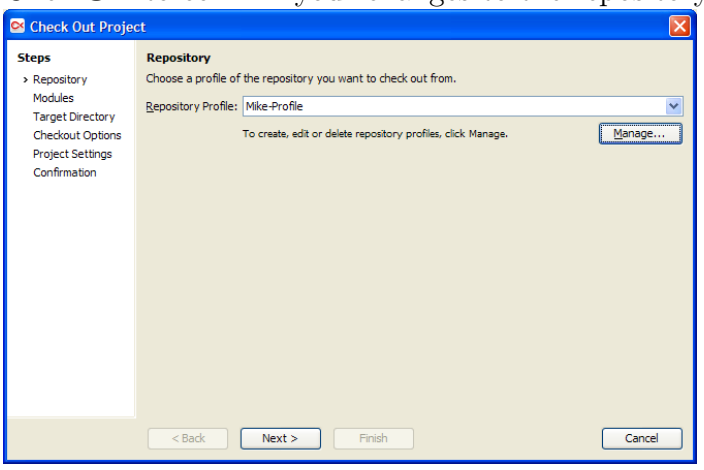
Enter Mike's password and click **Next**.



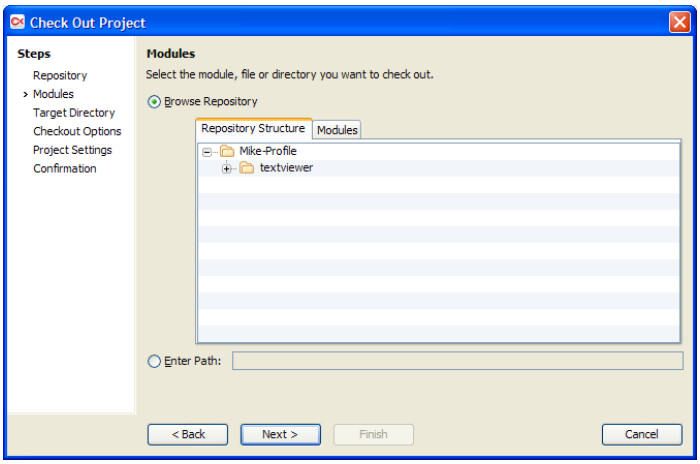
Change the profile name to `Mike-Profile` and click **Finish** to create the profile.



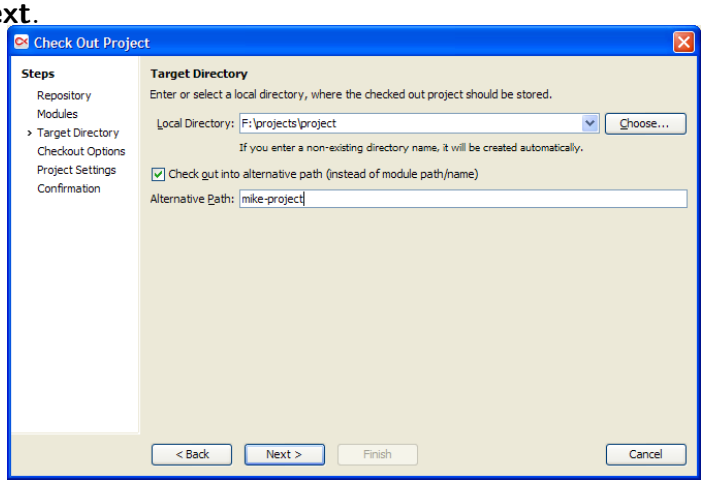
Click **OK** to confirm your changes to the repository profiles.



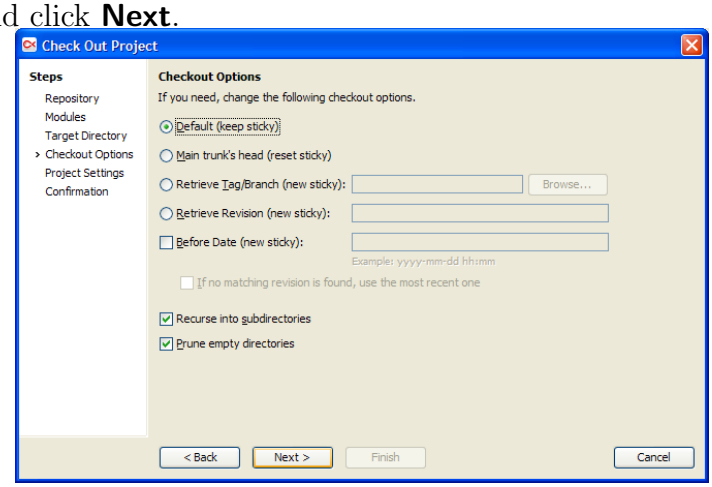
The newly created repository profile will be preselected automatically. Click **Next** to start scanning the repository defined by the selected repository profile.



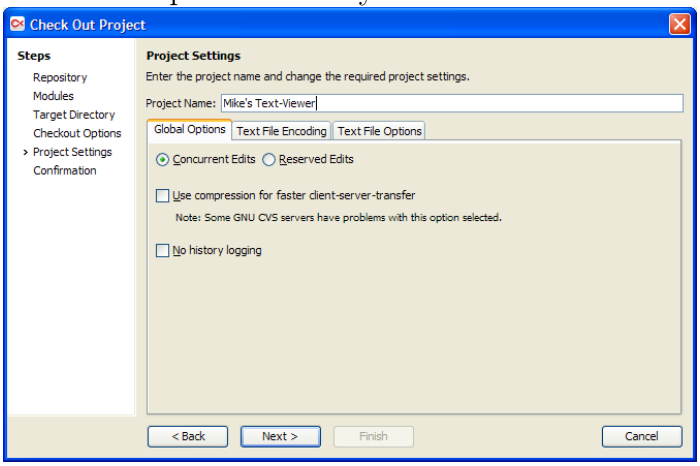
After a few moments, SmartCVS should have scanned the repository and displayed its content. Select the **textviewer** directory in the **Repository Structure** tab and click **Next**.



Now enter the local directory where Mike's Text-Viewer project should be created. By default SmartCVS checks out into a new directory with the name of the selected repository directory (here `textviewer`). To better distinguish Susan's and Mike's project, we name Mike's projects root directory differently. Keep the **Check out into alternative path (instead of module name)** option selected, enter `mike-project` as **Alternative Path** and click **Next**.

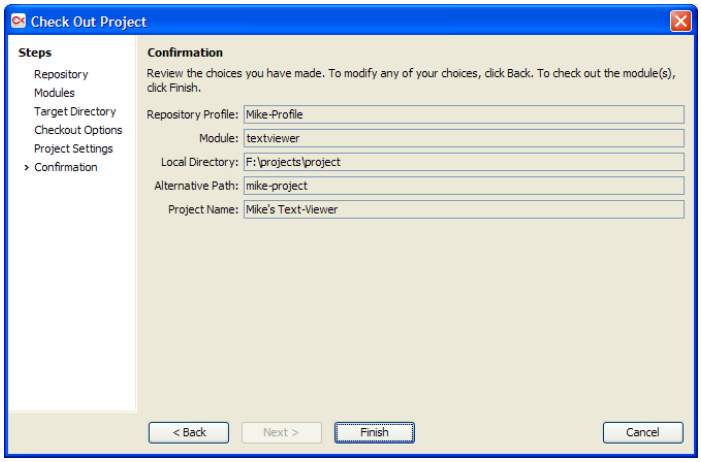


In some cases it is necessary to specify different options, but for this tutorial it is sufficient to keep them as they are. Click **Next** to continue.

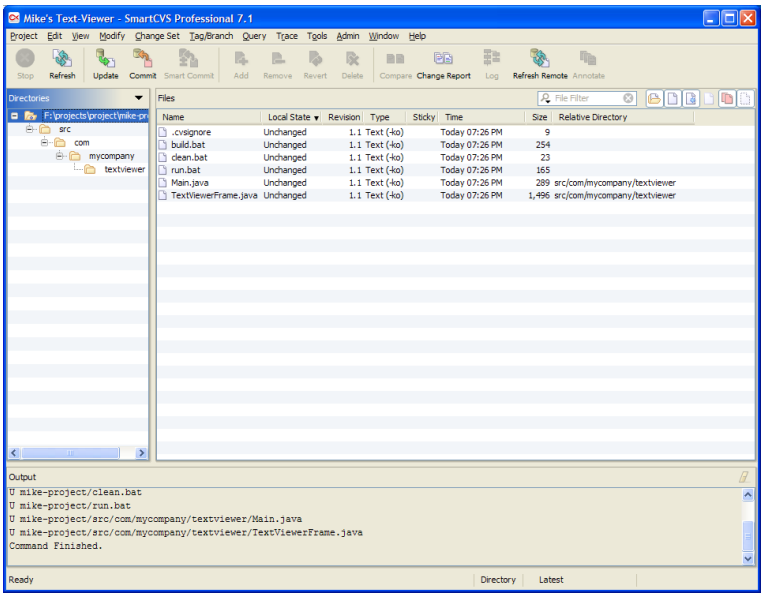


As the checkout will implicitly create a new SmartCVS-project, you can now adjust the project settings. Enter `Mike's Text-Viewer` as project name, leave all other options at their default values and click **Next** to proceed.

Note	If the default values do not match your needs and you always have the change them, you may change the default values in the preferences (**Edit**|**Preferences**).



Review your choices. If everything is fine, click **Finish** to start checking out Mike's project.

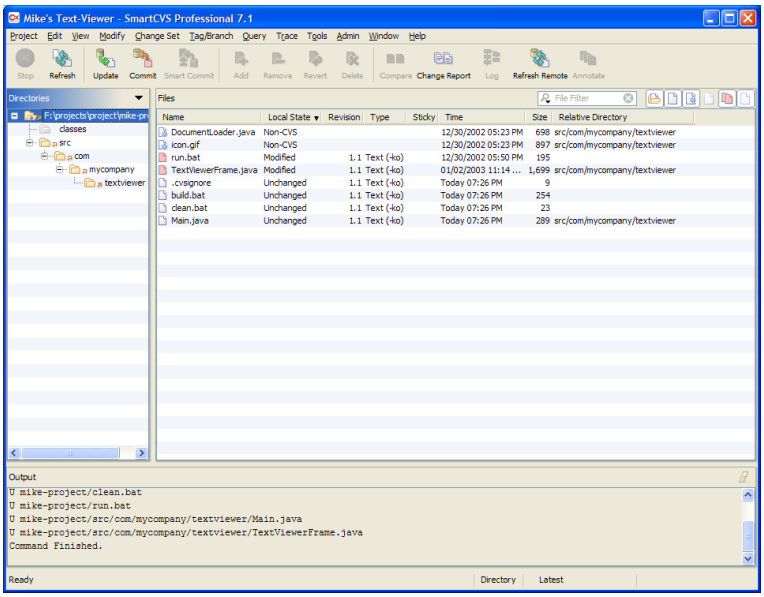


After a few moments all files have been checked out and Mike's project is created. He got all files, that are now present in the repository. Because Susan ignored the `classes` directory and therefore did not add any files from it, it has not been checked out into Mike's project. But after he ran `build.bat`, he will get the `classes` directory, too. Because Susan put the `.cvsignore` file also under CVS control, Mike got it with the check out, so the `classes` directory will be ignored in his project automatically.

5.2 Mike's Changes

Mike modifies the code a little bit and adds an icon to the application's window. Because the application needs to find the icon resource at runtime, he needs to change the `run.bat` file, too. All modified text files are available in the Section 9.2. You can modify the necessary files or unpack them from the zip-file `mikes-changes.zip`.

Select the root directory in the directory tree and click the **View**|**Refresh** menu item or the **Refresh** tool bar button to see the modifications. This forces SmartCVS to reload the file-system below the selected directory and update the Local States of the corresponding files.

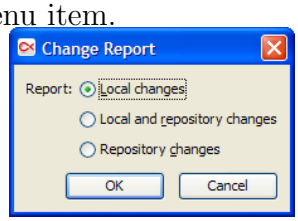


Note	By default, SmartCVS sorts the files by the Local State table column. This has the advantage, that the most important files will be shown on top.

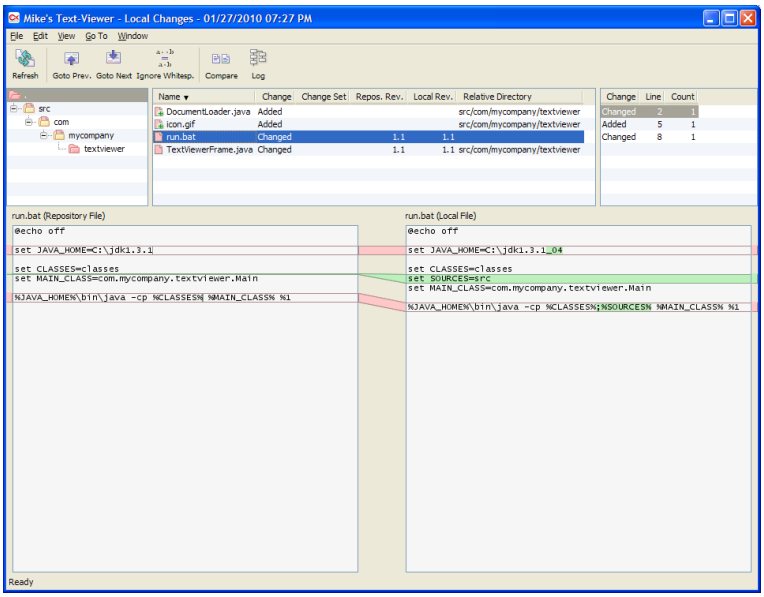
5.3 Change Report

After a while, Mike decides to commit his changes. As he is a cautious guy, he wants to review his local changes. This easily can be done using the Change Report.

Ensure, that the project root directory is selected and click the **Query|Change Report** menu item.



The change report offers three kinds of reports: Reporting only the local changes, comparing against the current revisions within the repository and and reporting only the changes made in the repository. Because Mike only wants to see his local changes, keep the option **Local changes** selected and click **OK**.



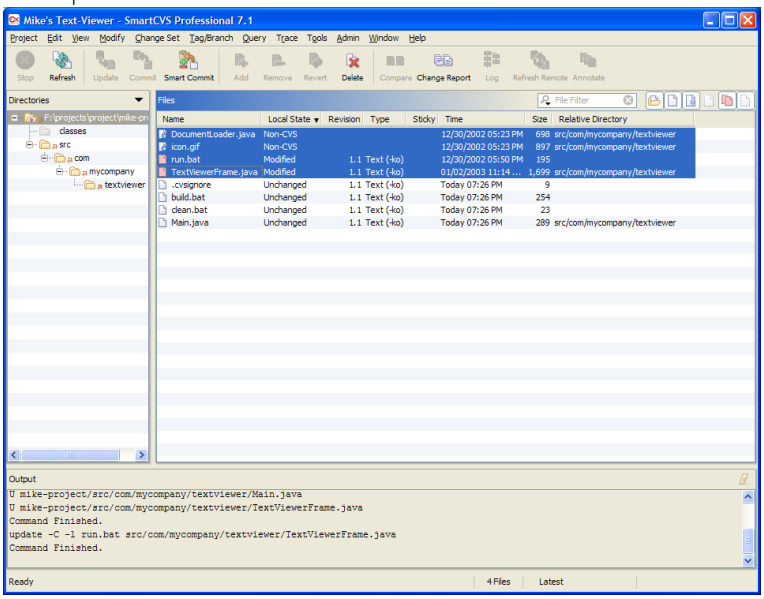
After a few moments the Change Report window will be opened. It shows four changed files and by selecting a file you can see the in-file changes which have been performed (except for binary files like the `icon.gif`). You also can use the arrow tool bar buttons to step through all changes of all files.

After reviewing the changes, close the Change Report window.

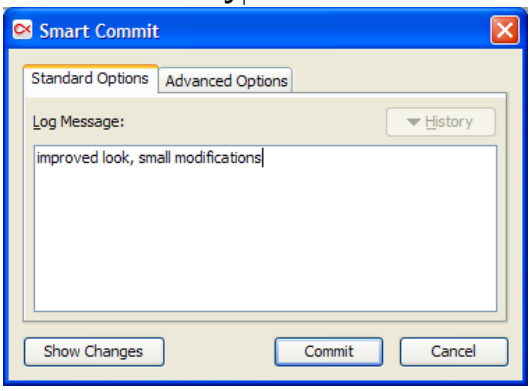
5.4 Smart Commit

Mike created two new files and modified two others. It would be possible to add the new files with the Add command as Susan did and commit all four files afterwards, but SmartCVS offers the ability to do this in one step.

Select the new, non-CVS files and the modified files. This can be done very fast using the **Edit|Select Committable Files** or the associated short-cut <*Ctrl*>+<*Shift*>+<*A*>.

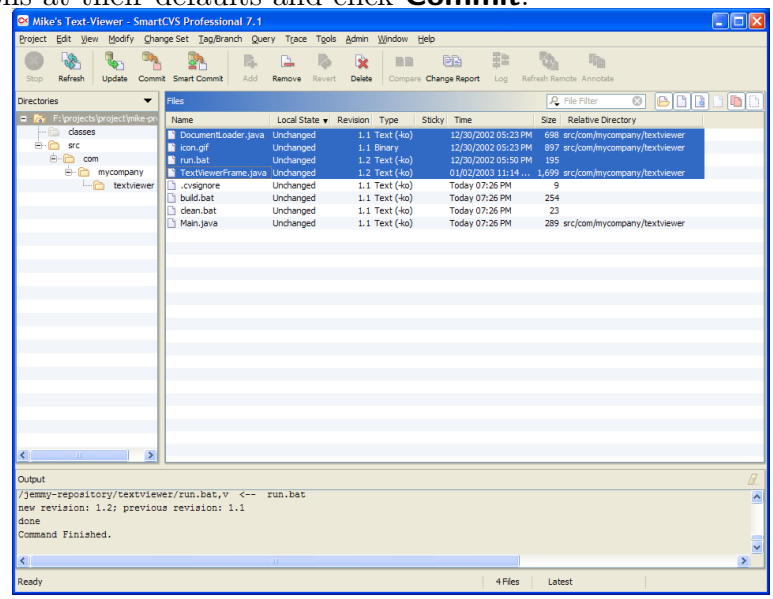


Click the **Modify|Smart Commit** menu item or the **Smart Commit** tool bar button.



The Smart Commit command does not support all options of the Add commands. Instead it uses the default values which can be configured in the preferences.

Enter Mike's log message `improved look, small modifications`, leave all other options at their defaults and click **Commit**.



SmartCVS adds the non-CVS files and then commits all files. One advantage is, that you don't need to worry about the file types, as you would need with other CVS clients. SmartCVS scans each file you want to add and detects, whether it must be added as a text file or as a binary file. Looking at the **Type** table column, you can see, that SmartCVS added the file `icon.gif` as a binary file. The revision of the modified and committed files changed from `1.1` to `1.2`.

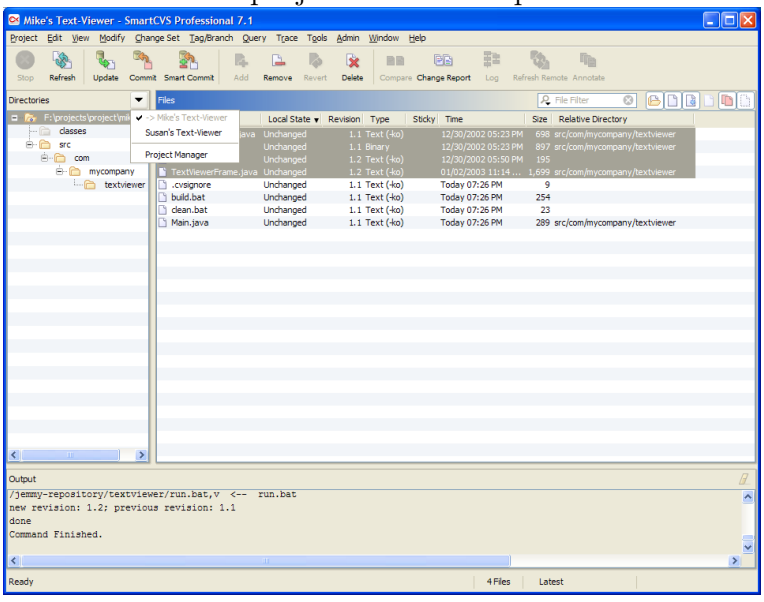
Chapter 6

Susan Fixes a Bug

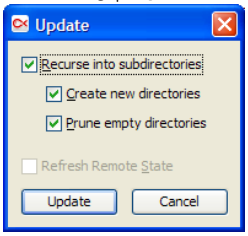
6.1 Get Mike's Changes

Susan needs to get Mike's changes in her project. This is done with the *Update* command.

To switch back to Susan's project, click the triangle button in the top-right of the **Directories** tree component and select her project. Select **Current Window**, when asked in which window the project should be opened.

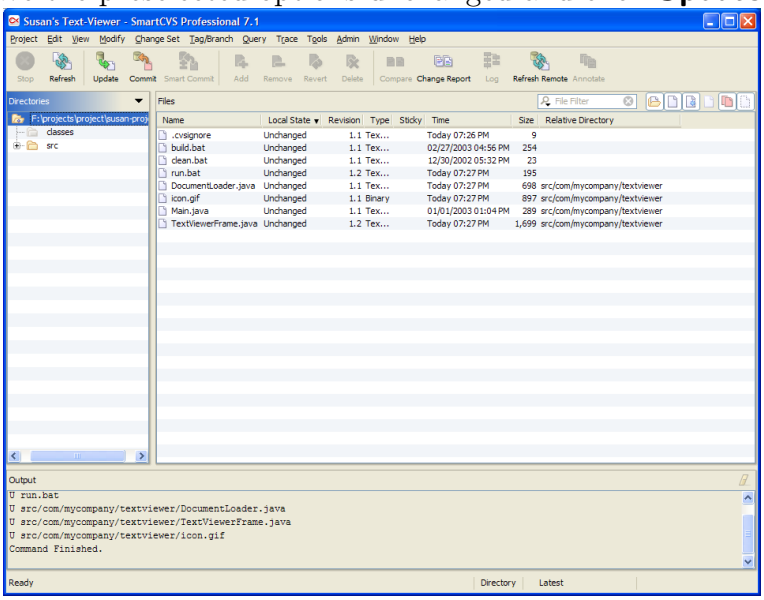


To get the latest sources, ensure, that the project root directory is selected and click the **Modify**|**Update** menu item or the **Update** tool bar button.



This is the light-weight version of the update command, which only has a few options to get the latest changes from the repository and is sufficient for 90% of the daily work.

Leave the preselected options unchanged and click **Update**.



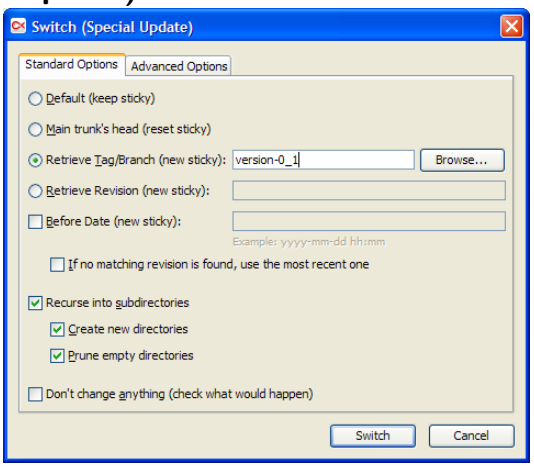
As you can see, Mike's added and changed files will be fetched.

Note	In contrast to some other version control systems, not only new files will automatically be fetched, but also obsolete files will be deleted.

6.2 Creating a Branch

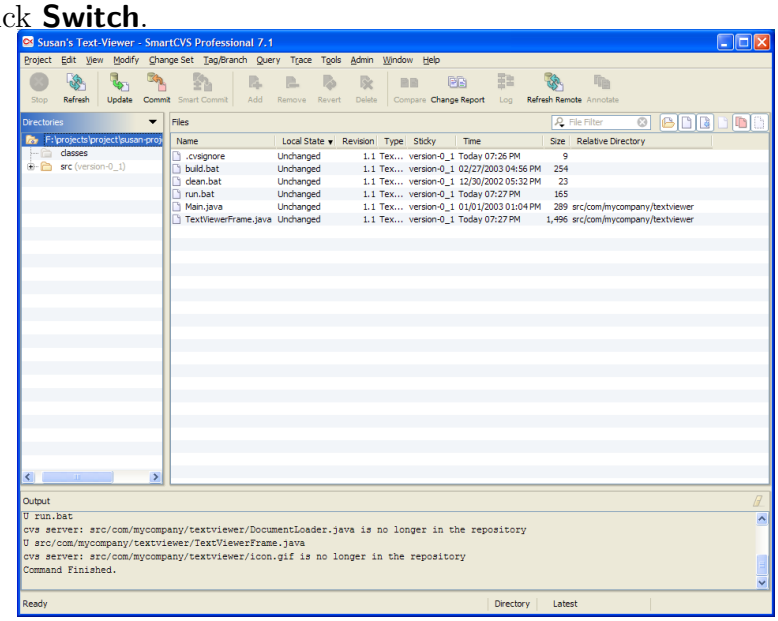
Before Susan can proceed in improving the Text-Viewer project, she receives a bug report about a program crash, when no file is specified on the command line. Mike says, that his modifications aren't tested enough to use it as a base for the bug-fix. Happy, that she has added a tag to her first release, she wants to fetch the first release's files back and fix the bug independent of Mike's changes.

Ensure, that the project root directory is selected and click the **Modify|Switch (Special Update)** menu item.



This is the powerful version of the update command, which also provides the possibility to switch to different tags, branches, revisions or to revert back erroneousy committed

files. Susan needs to fetch a project state, that was marked with a tag, so select the option **Retrieve Tag/Branch (new sticky)**. Enter `version-0_1` in the input field and click **Switch**.

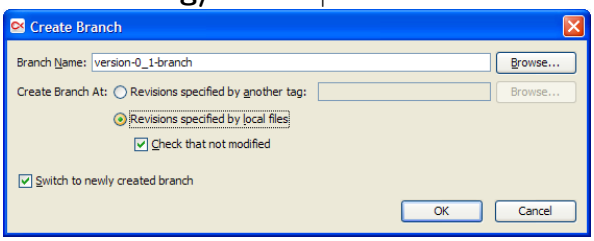


After a few moments Susan gets all files that were tagged with `version-0_1`, as you can see from the values of the **Sticky** table column. If a table column is too small, double click at the right side of its table header.

Note	Mike's newly added files `icon.gif` and `DocumentLoader.java` will be automatically removed, because they were not present in the "version 0.1".

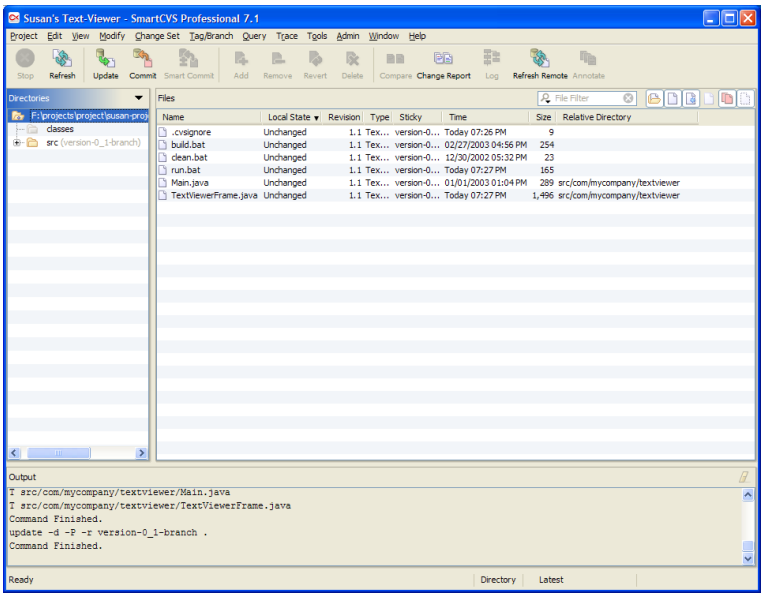
It is very convenient to fix bugs without impact on the ongoing work. You can do this with the help of branches. With CVS a branch is a special kind of tag, a so-called *branch-tag*.

To create a branch on all the revisions of the local files, select the project root directory and click the **Tag/Branch**|**Create Branch** menu item.



To distinguish between branch-tags and normal tags, it's always a good idea to name them differently, for example with a `-branch` suffix.

Enter the branch name `version-0_1-branch`, ensure, that the options **Revisions specified by local files** and **Update Branch** are selected, and click **OK**.

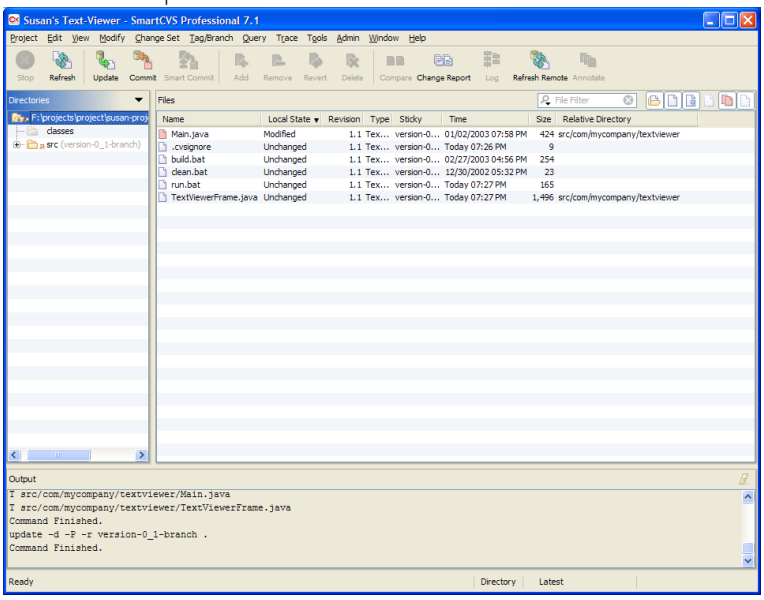


SmartCVS creates the branch-tag at the revisions of the local files and then updates with the newly created branch. As you can see from the revisions, no file was changed locally, but the Sticky value changed from `version-0_1` to `version-0_1-branch`. Also take a look at the versioned directories. Each one will display the branch name in gray braces. When a new file is added, it will be automatically added to the branch of its containing directory.

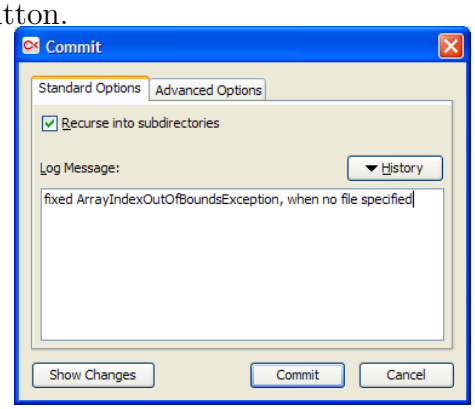
6.3 Fixing the Bug in the Branch

Susan modifies the file `Main.java` to show an error message on the command line, if no file was specified.

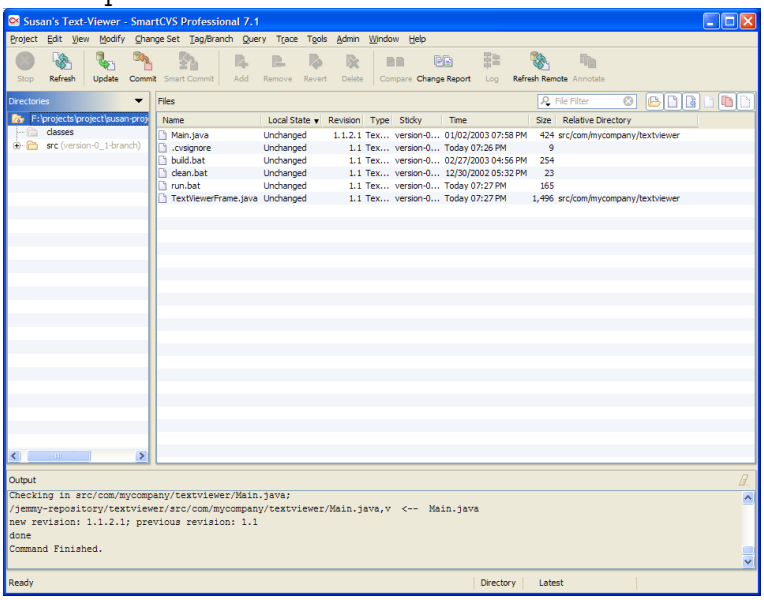
Change the file as shown in Section 9.3 or unpack it from the zip-file `susans-bugfix.zip`. Click the **View**|**Refresh** menu item or the **Refresh** tool bar button to see her changes.



To commit the changes, click the **Modify|Commit** menu item or the **Commit** tool bar button.



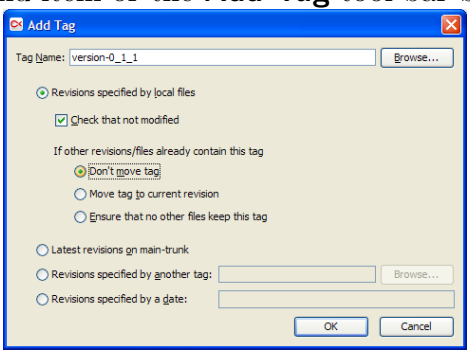
Enter a meaningful log message like `fixed ArrayIndexOutOfBoundsException, when no file specified` and click **OK**.



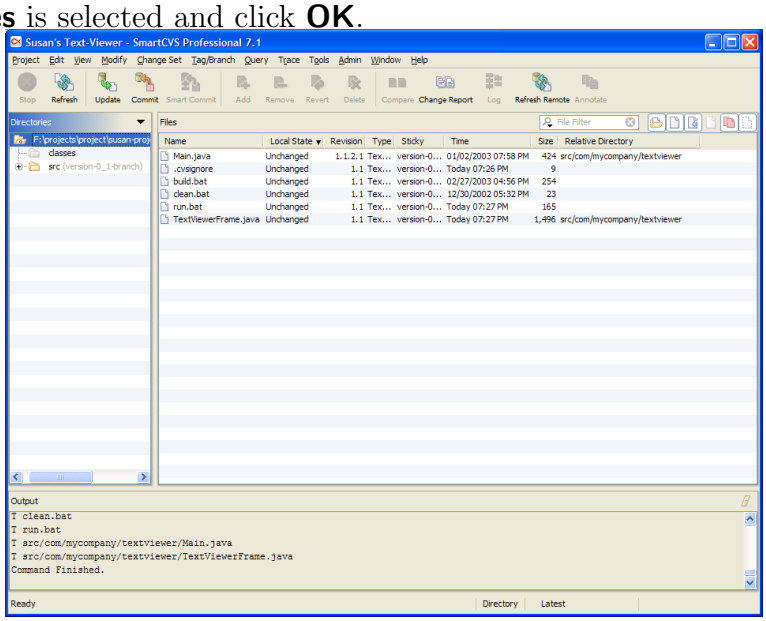
The revision of the file `Main.java` changed from `1.1` to `1.1.2.1` to indicate, that the new revision was created in the first branch of the revision 1.1.

After some testing, the version works fine and Susan releases the Text-Viewer project as "version 0.1.1". To reflect this by a tag, she tags the project with `version-0_1_1`.

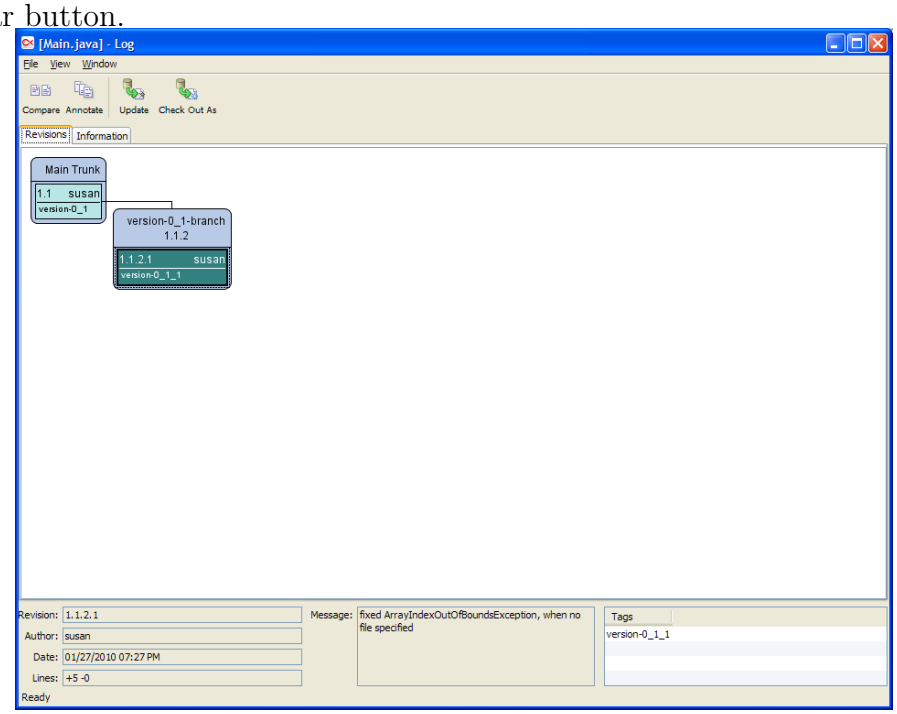
Ensure, that the project root directory is selected and click the **Tag/Branch|Add Tag** menu item or the **Add Tag** tool bar button.



Enter the tag name `version-0_1_1`, ensure, that option **Revisions specified by local files** is selected and click **OK**.



To see, what has happened in detail to the file `Main.java`, take a look at its log. Select the file `Main.java` and click the **Query|Log** menu item or the corresponding tool bar button.



In the Log window you can see two revisions (green background color), each within a branch (blue background color), for the file `Main.java`. Branches are completely independent change-lines and there is always a default branch, the so-called *Main Trunk*. The numbering of branches (except of the Main Trunk) starts with their initial revision followed by an even number, e.g., `1.1.2`, `1.1.4`, `1.1.6` would be the first, second and third branch of revision `1.1`. Normally you don't have to worry about branch numbers, because you cannot create a branch without a branch-tag. Branches might contain their

own revisions. Their revision numbers consist of the branch revision plus a positive number. For instance, `1.1.2.1`, `1.1.2.2`, `1.1.2.3` would be revisions in the first branch of revision `1.1`. The Main Trunk's revision numbers are two numbers, separated by a period, e.g., `1.1`, `1.2` or `1.157`.

The branch were revisions will be created, can be changed by running the Switch (Special Update) command with the option **Retrieve Tag/Branch (new sticky)** selected and the appropriate branch-tag.

Tip	The Main Trunk can be accessed from a branch with the tag name `HEAD`.

Chapter 7

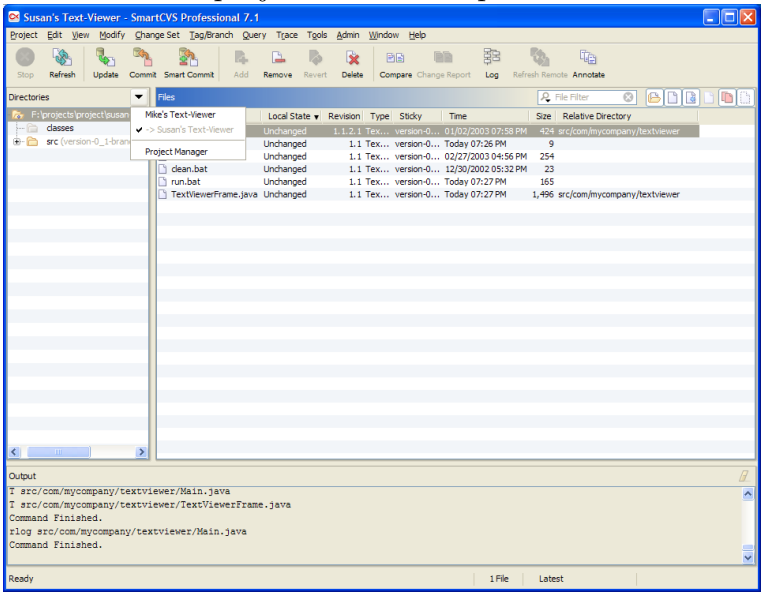
Mike Merges Susan's Bug-Fix

Susan fixed the bug only in the branch `version-0_1-branch`. Therefore Mike's code, which only exists on the Main Trunk, is not influenced by Susan's changes and still contains the same bug. There are two solutions to also fix the bug in the Main Trunk. Mike can do it himself or he can let CVS help him, which is easier and more robust.

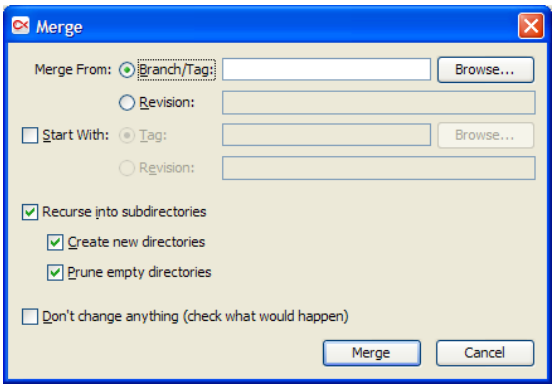
CVS records the changes for all revisions. Hence it knows, what lines were added, deleted or modified from revision to revision, and can apply the changes between two revisions to another revision of the same file. This process is called *merging*. Mike decides to let CVS do the merge for him.

7.1 Merging Changes Between Branches

Switch back to Mike's project by clicking the triangle button in the top-right corner of the **Directories** tree and choose Mike's project. Select **Current Window**, when asked in which window the project should be opened.



Ensure, that the project root directory is selected, and click the **Modify**|**Merge** menu item.

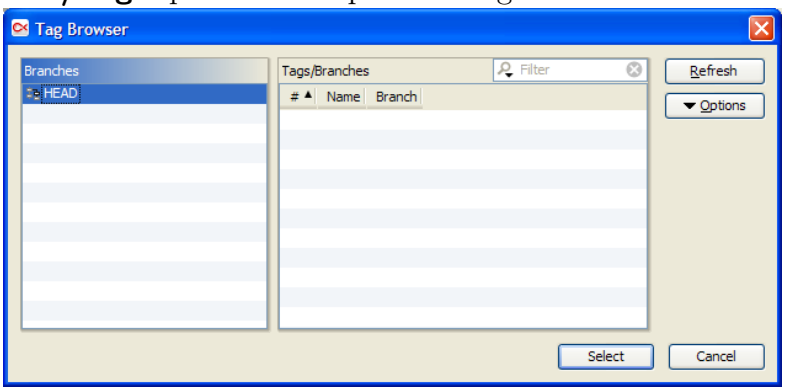


The Merge command allows you to merge changes between different branches. The target branch must be already checked out (see the **Sticky** column of the file table) and the source branch can be entered. Because it is possible to merge multiple times from one branch, it is necessary to merge only the changes made between a start- and an end-position, either specified with a (branch) tag or a revision. Although you don't merge multiple times in the Text-Viewer tutorial project, you should make use of this possibility.

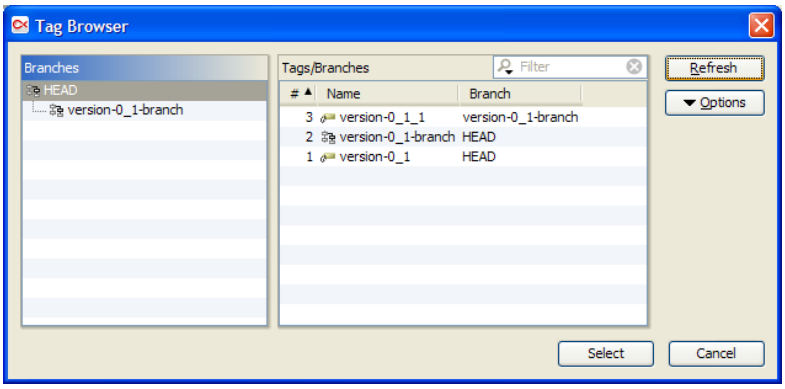
You need to merge the changes, that Susan made between the "version 0.1" and "version 0.1.1" into the Main Trunk. Mike's project is already on the Main Trunk (empty **Sticky** table column).

Mike can remember, that Susan tagged her initial project with `version-0_1`. This is the point from where he wants to start merging. Select the **Start With** option and enter `version-0_1` in the (second) **Tag** input field.

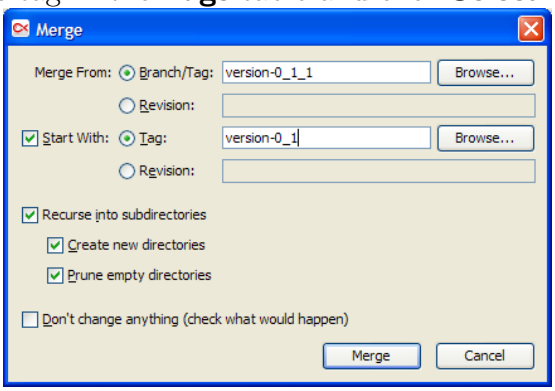
Mike does not know the tag, which Susan used to fix the reported bug. Therefore he will use the Tag Browser to select this tag. Click the **Browse** button right to the (first) **Branch/Tag** input field to open the Tag Browser.



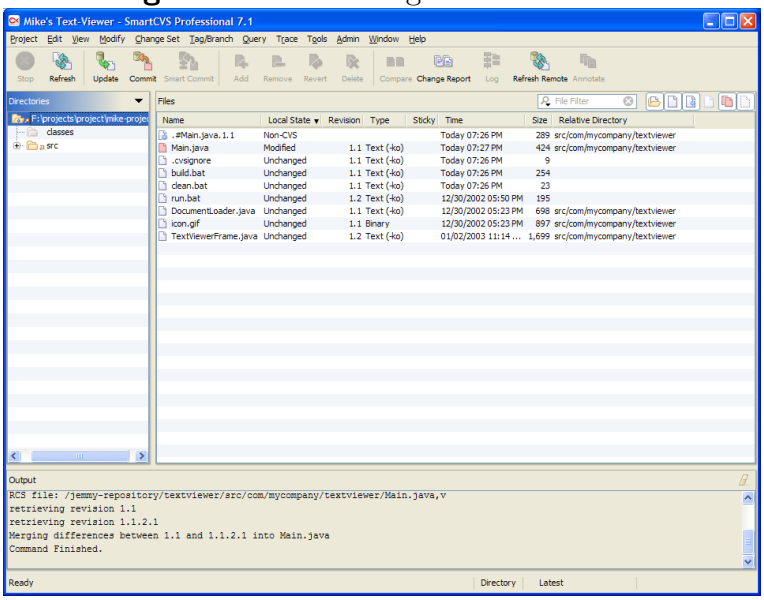
Click on the **Refresh** button to reload the tags from the repository.



Now Mike can figure out that Susan has tagged her bug-fix with `version-0_1_1`. Select this tag in the **Tags** table and click **Select**.



Click **Merge** to start the merge.

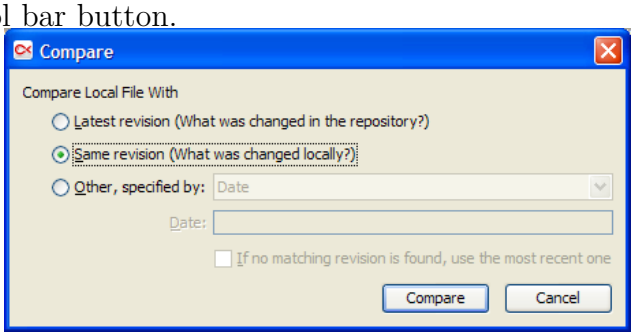


In contrast to other version control systems, CVS merges the changes locally (you merge *from*, not *to*), so you can test, whether the merge was successfully before committing the results. For all files where merging was necessary, the original local file is automatically backed up as file `.#<file-name>.<local-revision>`. Usually you can ignore and therefore delete such files after you have verified, that the merge was successful.

7.2 Comparing Files

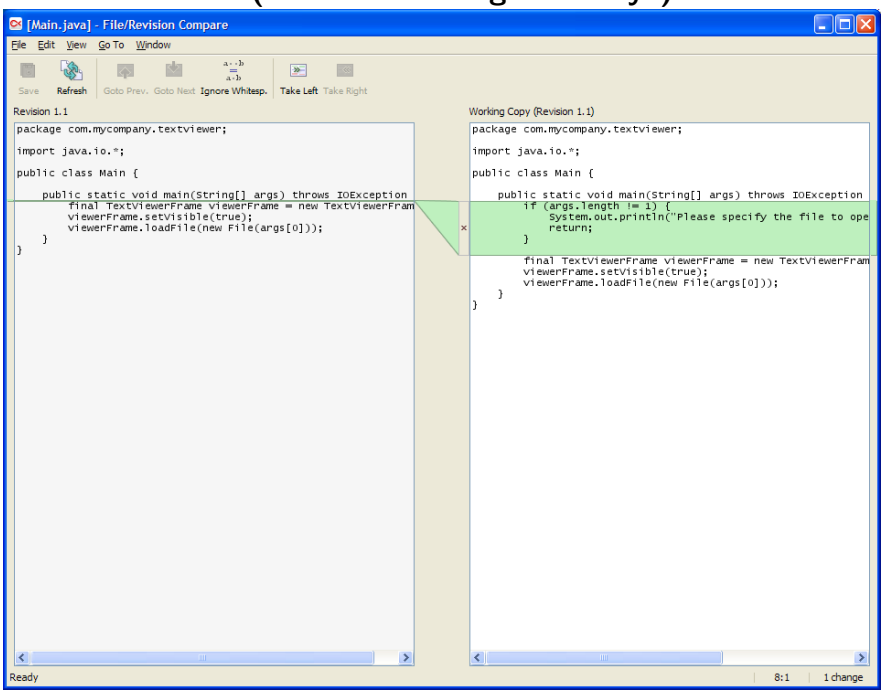
Mike wants to review the changes made by Susan. As only one file has been changed, he decides to use the built-in File Compare for this purpose.

Select the file `Main.java` and click the **Query|Compare** menu item or the **Compare** tool bar button.



You can decide, with what repository revision the local file should be compared. If the file is locally modified, it is most likely, that you want to see the local changes and therefore SmartCVS automatically selects **Same revision (What was changed locally?)**. If the file is up-to-date, it is most likely, that you want to see, what might have changed in the repository and therefore **Latest revision (What was changed in the Repository?)** is automatically selected.

Because Mike wants to see, what changes the merge produced locally, keep the default value **Same revision (What was changed locally?)** selected and click **Compare**.



After a few moments the File/Revision Compare window shows up. Between both file versions, the "change-stripe" to the left and right shows you at one glance all changes in the file. Green indicates inserted, blue deleted and red modified lines. If you have performed a compare on a local file, the right text area is editable, as it is the case for this example. This is for instance convenient, if you want to revert some of your changes

before committing the file.

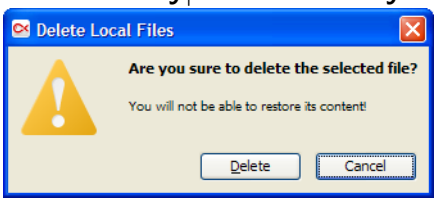
Mike decides to rewrite the error message, which has been introduced by Susan in case of a missing file name.

Click in the right area of the File/Revision Compare window and replace Susan's text `Please specify the file to open as the one and only argument.` with `Usage: run filename` which sounds more professional to Mike. Then click **File**|**Save** and close the window.

Note	You can also configure external file comparators for different file patterns. Take a look at the Preferences dialog (**Edit**|**Preferences** menu item), if you want to make use of this possibility.

7.3 Committing Merged Files

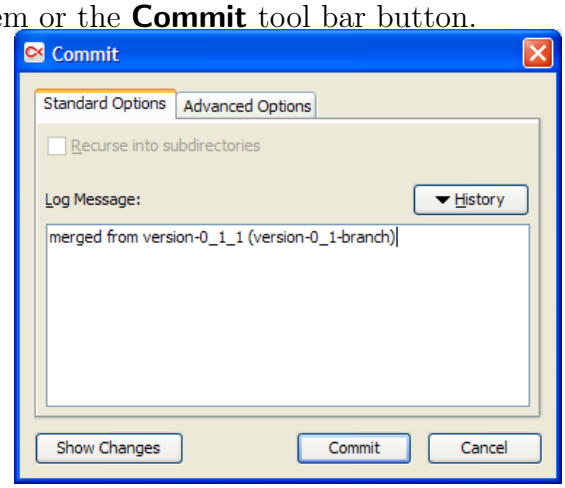
Mike has verified the successful merge. Therefore select the file `.#Main.java.1.1` and click the **Modify**|**Delete Locally** menu item or the **Delete** tool bar button.



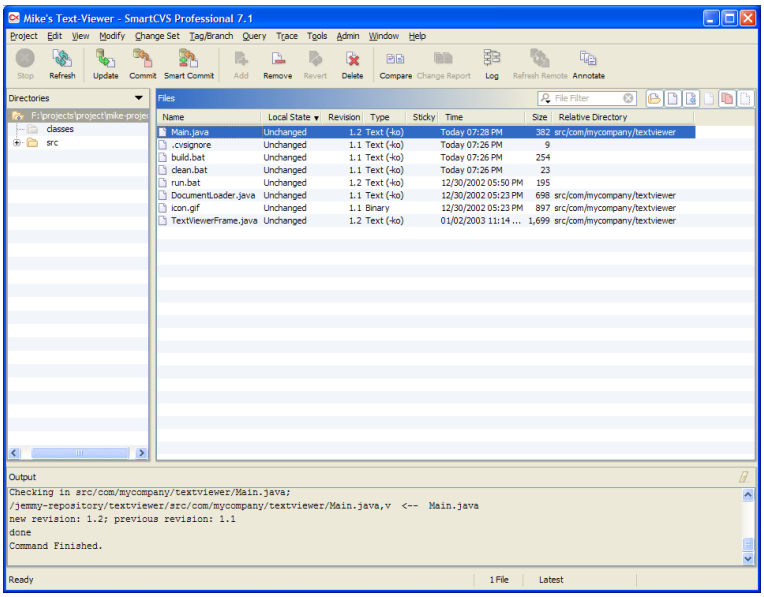
Click **Delete** to confirm the deletion of the file.

Warning!	You will not be able to restore the deleted file(s), e.g, from the Windows Trash, so be careful before clicking **Delete**!

Now Mike can commit the changes. No files need to be added or removed. Therefore it is sufficient to select the project root directory and click the **Modify**|**Commit** menu item or the **Commit** tool bar button.



Enter a meaningful log message like `merged from version-0_1_1 (version-0_1-branch)` and click **Commit**.



The `Main.java`'s revision increased to `1.2`, indicating, that a new revision was created on the Main Trunk. The Main Trunk now contains Mike's improvements and Susan's bug-fix.

Note	Susan could now switch back to work together with Mike in the Main Trunk by performing an update with the option **Main trunk's head (Reset Sticky)** selected.

Chapter 8

Summary

Obviously, this Text-Viewer project can be more improved, but you can delegate this to Susan and Mike.

In this tutorial you have learned, how to use SmartCVS to import projects into the repository, how to check out working areas. You have seen, how to get latest sources, commit changes or work with branches, so it should be possible for you to start with SmartCVS on a daily base. SmartCVS contains a lot more features for successful development and maintainance of software projects or other projects for which a change-history is useful. Find them out by yourself!

Good luck and happy versioning!

Chapter 9

Files

9.1 Susan's First Version

This is the directory structure.

```
[.]
 + [classes]
 | - [com]
 |   - [mycompany]
 |     - [textviewer]
 |       * Main.class
 |       * TextViewerFrame.class
 + [src]
 |  - [com]
 |     - [mycompany]
 |        - [textviewer]
 |           * Main.java
 |           * TextViewerFrame.java
 * build.bat
 * clean.bat
 * run.bat
```

9.1.1 File `build.bat`

```
@echo off

set JAVA_HOME=C:\jdk1.3.1

set CLASSES=classes
set SOURCES=src
set MAIN_FILE=%SOURCES%\com\mycompany\textviewer\Main.java

if not exist %CLASSES% mkdir %CLASSES%
```

```
%JAVA_HOME%\bin\javac -d %CLASSES% -sourcepath %SOURCES% %MAIN_FILE%
```

9.1.2 File clean.bat

```
rmdir classes /s /q
```

9.1.3 File run.bat

```
@echo off

set JAVA_HOME=C:\jdk1.3.1

set CLASSES=classes
set MAIN_CLASS=com.mycompany.textviewer.Main

%JAVA_HOME%\bin\java -cp %CLASSES% %MAIN_CLASS% %1
```

9.1.4 File src/com/mycompany/textviewer/Main.java

```
package com.mycompany.textviewer;

import java.io.*;

public class Main {

  public static void main(String[] args) throws IOException {
    final TextViewerFrame viewerFrame = new TextViewerFrame();
    viewerFrame.setVisible(true);
    viewerFrame.loadFile(new File(args[0]));
  }
}
```

9.1.5 File src/com/mycompany/textviewer/TextViewerFrame.java

```
package com.mycompany.textviewer;

import java.awt.*;
import java.io.*;
import javax.swing.*;
import javax.swing.text.*;

class TextViewerFrame extends JFrame {

  private final JTextArea textArea;
```

```
public TextViewerFrame() {
  super("TextViewer");

  setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

  textArea = new JTextArea();
  textArea.setFont(new Font("Monospaces", Font.PLAIN, 12));
  textArea.setEditable(false);

  final JScrollPane scrollPane = new JScrollPane(textArea);
  scrollPane.setPreferredSize(new Dimension(400, 300));
  setContentPane(scrollPane);

  pack();

  final Dimension screenSize = Toolkit.getDefaultToolkit().getScreenSize();
  final Dimension size = getSize();
  setLocation((screenSize.width - size.width) / 2,
              (screenSize.height - size.height) / 2);
}

public void loadFile(File file) throws IOException {
  final BufferedReader reader = new BufferedReader(new FileReader(file));
  try {
    final Document document = new PlainDocument();

    for (String line = reader.readLine();
         line != null;
         line = reader.readLine()) {
      document.insertString(document.getLength(), line, null);
      document.insertString(document.getLength(), "\n", null);
    }

    textArea.setDocument(document);
  }
  catch (BadLocationException ex) {
    ex.printStackTrace();
  }
  finally {
    try {
      reader.close();
    }
    catch (IOException ex) {
      // if this fails, we really have a problem
    }
```

```
    }
  }
}
```

9.2 Mike's Changes

Following, only the changed files are listed.

9.2.1 File run.bat

```
@echo off

set JAVA_HOME=C:\jdk1.3.1

set CLASSES=classes
set SOURCES=src
set MAIN_CLASS=com.mycompany.textviewer.Main

%JAVA_HOME%\bin\java -cp %CLASSES%;%SOURCES% %MAIN_CLASS% %1
```

9.2.2 File src/com/mycompany/textviewer/DocumentLoader.java

```
package com.mycompany.textviewer;

import java.io.*;
import javax.swing.text.*;

class DocumentLoader {

  public void load(Document document, File file) throws IOException {
    final BufferedReader reader = new BufferedReader(new FileReader(file));
    try {
      for (String line = reader.readLine();
           line != null;
           line = reader.readLine()) {
        document.insertString(document.getLength(), line, null);
        document.insertString(document.getLength(), "\n", null);
      }
    }
    catch (BadLocationException ex) {
      ex.printStackTrace();
    }
    finally {
      try {
```

```
        reader.close();
      }
      catch (IOException ex) {
        // if this fails, we really have a problem
      }
    }
  }
}
```

9.2.3 File src/com/mycompany/textviewer/TextViewerFrame.java

```
package com.mycompany.textviewer;

import java.awt.*;
import java.awt.event.*;
import java.io.*;
import javax.swing.*;
import javax.swing.text.*;

class TextViewerFrame extends JFrame {

  private final JTextArea textArea;

  public TextViewerFrame() {
    super("TextViewer");
    setIconImage(new ImageIcon(getClass().getResource("icon.gif")).getImage());

    textArea = new JTextArea();
    textArea.setFont(new Font("Monospaces", Font.PLAIN, 12));
    textArea.setEditable(false);

    init();
  }

  private void init() {
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    setJMenuBar(createMenuBar());

    final JScrollPane scrollPane = new JScrollPane(textArea);
    scrollPane.setPreferredSize(new Dimension(400, 300));
    setContentPane(scrollPane);

    pack();
```

```
    final Dimension screenSize = Toolkit.getDefaultToolkit().getScreenSize();
    final Dimension size = getSize();
    setLocation((screenSize.width - size.width) / 2,
                (screenSize.height - size.height) / 2);
  }

  private JMenuBar createMenuBar() {
    final JMenu fileMenu = new JMenu("File");
    fileMenu.setMnemonic('f');
    fileMenu.add(createExitAction());

    final JMenuBar menuBar = new JMenuBar();
    menuBar.add(fileMenu);
    return menuBar;
  }

  private AbstractAction createExitAction() {
    return new AbstractAction("Exit") {
      public void actionPerformed(ActionEvent e) {
        processWindowEvent(new WindowEvent(TextViewerFrame.this,
                                           WindowEvent.WINDOW_CLOSING));
      }
    };
  }

  public void loadFile(File file) throws IOException {
    final PlainDocument document = new PlainDocument();
    new DocumentLoader().load(document, file);
    textArea.setDocument(document);
  }
}
```

9.3 Susan's Bug Fix

9.3.1 File src/com/mycompany/textviewer/Main.java

```
package com.mycompany.textviewer;

import java.io.*;

public class Main {

  public static void main(String[] args) throws IOException {
    if (args.length != 1) {
      System.out.println("Please specify the file to open as the one and" +
```

```
                          " only argument.");
      return;
    }

    final TextViewerFrame viewerFrame = new TextViewerFrame();
    viewerFrame.setVisible(true);
    viewerFrame.loadFile(new File(args[0]));
  }
}
```