7

Painless Code Sharing Using Team Support

IN THIS CHAPTER

It's the source control, stupid.

—Anonymous

Eclipse Support for CVS

No matter what your views may be about the use of an IDE versus a text editor, the one thing developers can agree on is the use of source control. Once your source code has found its way onto the filesystem, it is imperative that you save it in some kind of repository when you feel it has reached a level of functionality you can snapshot. Test-driven development techniques allow you to feel confident about the state of your code at any given milestone, whereas the use of source control gives you the knowledge that, if you had to, you can always roll back to an earlier version. Don't let test-driven development make you cocky; always use source control.

The advantage of source control is that you can work on your copy of the source for as long as you like, and when you decide to check the code into the repository, CVS will let you know that either your code is acceptable or someone else changed the code since you checked it out and now you need to resynchronize your code with the repository. Let's look at Eclipse support for CVS and how it will assist you in importing code for you to work on as well as exporting source code when you are finished.

Eclipse support for source control is at the same level of support as Java development. The CVS perspective contains views of the available CVS repositories, including a repositories view, an editor view, and resource history. Unfortunately, Eclipse does not support local CVS repositories. The only way Eclipse can perform its source control tasks is through the use of a CVS server available on some port on some machine.

CVS Perspective

The CVS perspective is displayed by either going to the main menu and selecting Window, Open Perspective or by clicking the Open Perspective button in the shortcut bar at the top right of the workbench. In either case, the CVS perspective appears as shown in Figure 7.1. The first time you use the CVS perspective, it will not display any repositories. Eclipse waits for you to enter a server and repository name so it can connect to the remote server and display whatever CVS information the server is willing to supply. You can add or remove repositories at any time, but be careful: Eclipse does not confirm the removal of a repository from the CVS Repositories view unless you have imported code into a project. This does not affect the actual repository in any way; it only removes it from the view.

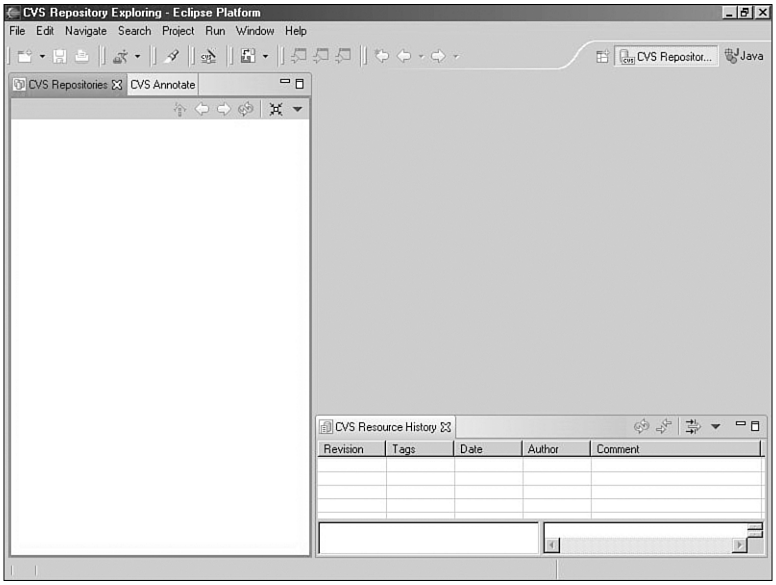


FIGURE 7.1 The CVS perspective with no displayable CVS repositories.

Creating a New Repository Entry

Let's look at the various views and how they support each other by connecting to a public CVS repository on the Web.

To illustrate creating a new repository entry, let's go to SourceForge and use Eclipse to browse the CVS repository of a SourceForge project. You will start by going to `http://sourceforge.net` and selecting one of the projects under the Most Active list in the left navigation bar. At the time of this writing, JBoss was the fourth most active project on the site. After selecting the JBoss.org project (`http://sourceforge.net/projects/jboss/`), you should see the "Project: JBoss.org: Summary" page appear. Select the CVS link from the headers list directly below the Summary bar. The CVS link leads you to the "Project: JBoss.org: CVS" page, which briefly discusses CVS, anonymous CVS access, and developer CVS access. The information you are looking for is located in the Anonymous CVS Access section. When I accessed the CVS page for JBoss.org, the Anonymous CVS Access section displayed the following CVS information:

```
cvs -d:pserver:anonymous@cvs.sourceforge.net:/cvsroot/jboss login
```

This line tells us three things:

- The CVS server name is `cvs.sourceforge.net`.
- The CVS root path to the project is `/cvsroot/jboss`.
- The CVS user login name is `anonymous`.

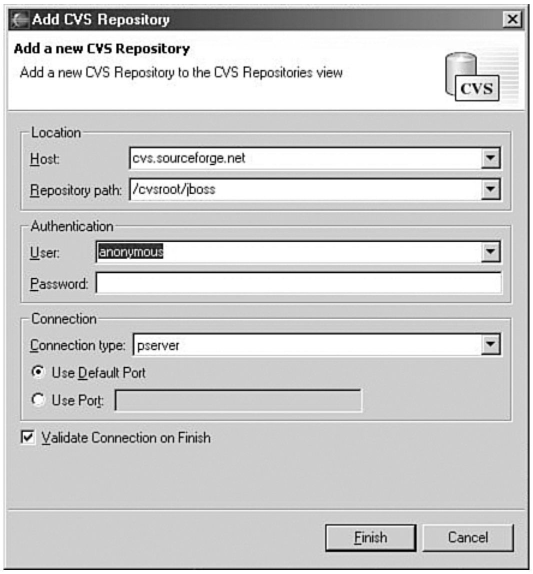
Something else the JBoss.org CVS page tells us is that user `anonymous` has no password. Using this information, the next few steps will connect you to the SourceForge CVS server and allow Eclipse to display the CVS repository for the JBoss.org project.

CVS, Firewalls, and Connectivity

If the CVS server you need to access is not located on your local machine, you need to be set up to connect either across the Internet or on your local network. Connecting to SourceForge is a trivial exercise that gives you a rich selection of code from which to choose. Remember, however, that this will not work if you are situated behind a firewall or are not connected to the Internet.

First, right-click in the CVS Repositories view and select New, Repository Location to display the Add CVS Repository dialog (see Figure 7.2). Once you enter all the required information, this dialog will take care of finding the machine on the network and displaying its repository in the CVS Repositories view located on the left side of the workbench.

In the Add CVS Repository dialog, enter **cvs.sourceforge.net** into the Host field, **/cvsroot/jboss** into the Repository Path field, and **anonymous** as the user. The Connection Type field should be set to pserver; if it is not, change it to pserver. The CVS servers at SourceForge use the standard CVS password server to authenticate any users coming into the repositories, so you must set the connection type to pserver. Use Default Port should be the selected radio button, and Validate Connection on Finish should be checked. Figure 7.2



Add CVS Repository

Add a new CVS Repository

Add a new CVS Repository to the CVS Repositories view

Location
Host: cvs.sourceforge.net
Repository path: /cvsroot/jboss

Authentication
User: anonymous
Password:

Connection
Connection type: pserver
(●) Use Default Port
( ) Use Port:

[✓] Validate Connection on Finish

FIGURE 7.2 The Add CVS Repository dialog with all the information necessary to connect to the SourceForge CVS server.

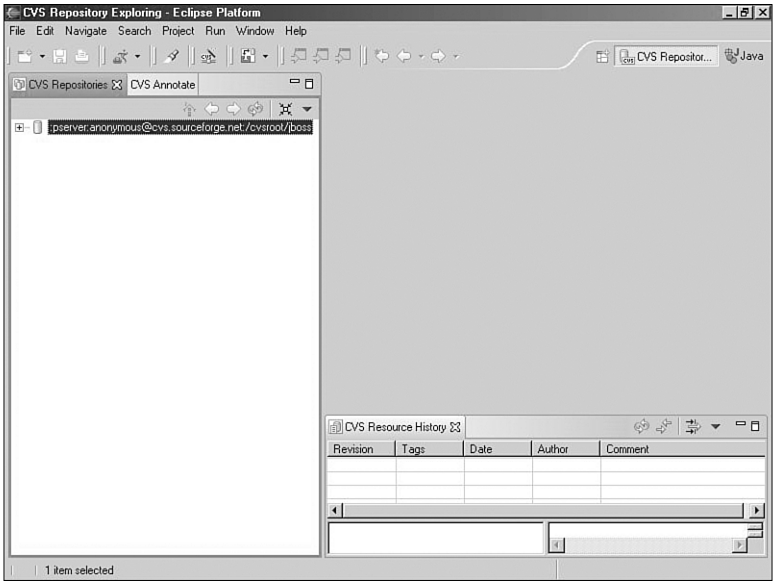
shows all the information discussed here. Click Finish. A Progress Information dialog should briefly appear. When it closes, the CVS Repositories view should display the JBoss.org CVS source tree (see Figure 7.3).

A Brief Tour of the Various Views

Let's look at the various parts of the CVS perspective using the JBoss CVS repository.

The default CVS perspective is made up of three views:

- The view to the left is the CVS Repositories view.
- Directly to the right of the CVS Repositories view is an editor area.
- The area below the editor is the CVS Resource History.



CVS Repository Exploring - Eclipse Platform

:pserver:anonymous@cvs.sourceforge.net:/cvsroot/jboss

Revision	Tags	Date	Author	Comment

1 item selected

FIGURE 7.3 The CVS Repositories view with the JBoss.org CVS repository being displayed.

The CVS Repositories view is where CVS repositories are displayed in a tree view (see Figure 7.4). The CVS Repositories view can display zero-to-many CVS repositories. Double-clicking the repository name, or clicking once on the plus sign, will open the view, revealing the HEAD, Branches, and Version nodes.

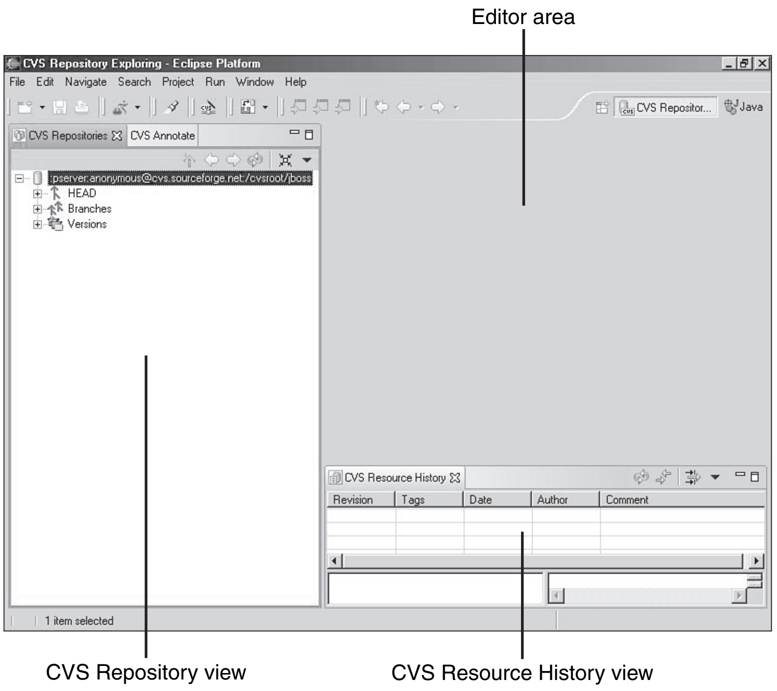

Editor area
CVS Repository Exploring - Eclipse Platform
File Edit Navigate Search Project Run Window Help
CVS Repositor...
Java
CVS Repositories
CVS Annotate
:pserver:anonymous@cvs.sourceforge.net:/cvsroot/jboss
HEAD
Branches
Versions
CVS Resource History
Revision
Tags
Date
Author
Comment
1 item selected
CVS Repository view
CVS Resource History view

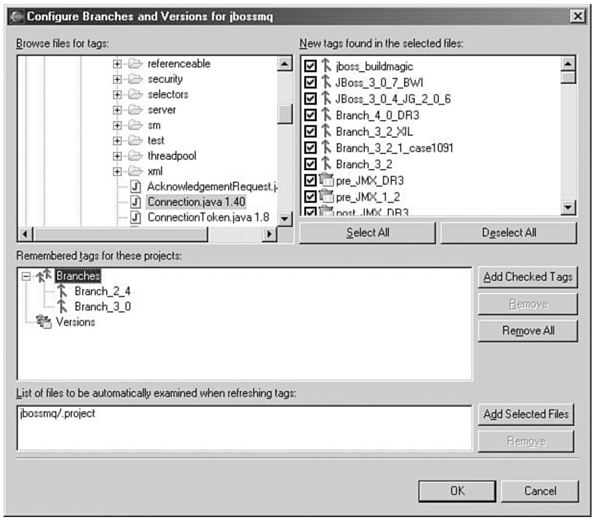
FIGURE 7.4 The CVS perspective with its two views and one editor area.

The HEAD node is the main branch of the CVS tree for that repository. It represents the main branch or trunk of the repository tree. Double-clicking HEAD, or clicking once on the plus sign next to the HEAD branch, will display everything in the main branch of the repository, which includes directories and files. Double-click HEAD to close the node.

Double-clicking the Branch node, or clicking once on the plus sign next to the Branch node, will reveal that there are no branches currently available. What does this mean? Well, you would create a branch in a CVS repository when you need a safe place to make changes that will end up in the HEAD branch. If you were to decide to make changes to the JBoss.org source (assuming you had developer privileges), you would create a branch where the files you wanted to work on would be associated. After the file changes were complete, you would check to see if the files had changed since you took them out. If they had changed, you would have to reconcile the new changes with your current changes. If no changes were made, you could safely check your files into HEAD.

Opening the Version node displays version tags associated with various directories and files for a particular version. For example, selecting the version tag `jbossmq` and right-clicking it will display the pop-up menu. Choosing Configure Branches and Versions causes a dialog to appear that allows you to select branches and/or versions of a directory associated with a particular branch or version (see Figure 7.5).

Let's walk through an interesting JBoss directory and add it to the Branch node. Once that directory has created a new branch, you will add the directory's contents as a project within Eclipse. Open the Version node in the CVS Repositories view, scroll down, and right-click `jbossmq`. Select Configure Branches and Versions from the pop-up menu. When the dialog appears, open the `src` folder by clicking the plus sign next to it and then open the main folder by clicking the plus sign next to `main`. Open the `org` directory, followed by `jboss` and then `mq`. Select the `Connection.java` file. The list box to the right now displays the various branches and versions associated with that file. Click the Deselect All button and check the Branch_2_4 and Branch_3_0 boxes. Click Add Checked Tags, and a plus sign will appear next to the Branches node in the tree view in the center of the dialog. If you open that Branch node, you will see that Branch_2_4 and Branch_3_0 have been added to it. The dialog and all the preceding information is displayed in Figure 7.5. Click OK.


Configure Branches and Versions for jbossmq

Browse files for tags:
- referenceable
- security
- selectors
- server
- sm
- test
- threadpool
- xml
- AcknowledgementRequest.j
- Connection.java 1.40
- ConnectionToken.java 1.8

New tags found in the selected files:
- ✓ jboss_buildmagic
- ✓ JBoss_3_0_7_BWI
- ✓ JBoss_3_0_4_JG_2_0_6
- ✓ Branch_4_0_DR3
- ✓ Branch_3_2_XIL
- ✓ Branch_3_2_1_case1091
- ✓ Branch_3_2
- ✓ pre_JMX_DR3
- ✓ pre_JMX_1_2
- ✓ post_JMX_DR3

Remembered tags for these projects:
- Branches
  - Branch_2_4
  - Branch_3_0
- Versions

List of files to be automatically examined when refreshing tags:
jbossmq/.project

FIGURE 7.5 The Configure Branches and Versions dialog for jbossmq displaying the two selected branches.

The Branch node in the CVS Repositories view now has a plus sign next to it. Open the Branch node and you will see two branches associated with it: Branch_2_4 and Branch_3_0. Open either branch and you will see that `jbossmq` has been added as a branch associated with the JBoss.org repository. These branches have no connection with the public CVS repository with which we started. However, if you right-click the `jbossmq` folder, you'll see the pop-up menu item Check Out As Project (the second item in the list). If you select Check Out As

Project, the CVS plug-in will extract all the code in the folder and create an Eclipse project named jbossmq made up of the code from the particular branch of JBoss.org you selected, which in this case is not the latest branch.

The Editor view is used to display a read-only view of a selected file. Because the file is not associated with an Eclipse project, you are not allowed to change the file in any way. Select Branches, Branch_2_4, jbossmq Branch_2_4, src, main, org, jboss, mq and then double-click `Connection.java`. The file will appear in the editor view fully syntax-colored and with a moveable cursor. Cutting and pasting from the editor into another destination is supported.

The final view is the CVS Resource History. This view displays the CVS revisions, tags, time-stamps, author, and comments associated with a file. Right-clicking the `Connection.java` file used earlier will display the selection Show in Resource History in the pop-up menu. Select that item, and the current known CVS information for `Connection.java` will appear. Of course, all the CVS information is displayed, not just the information up to the version you selected.

The two text areas below the CVS Resource History table will display detail data for a chosen row. Select one of the rows of CVS data from the CVS Resource History. The left window will display version tags, if any are available, and the right window will display the full comment.

Accessing CVS

An interesting point to note is that the CVS perspective does not care if you are looking at a repository of Java code or a repository of COBOL code. Source control is source control. CVS is CVS. The JBoss.org repository consists of Java code, but the information we will be viewing pertains to anything under the control of the repository.

Checking In

Now that you have seen how easy it is to connect to an external CVS server, create working branches from versions within the CVS repository, and create projects from CVS folders, let's create a new project and make it available to the rest of our team. For the sake of argument, assume that we have a project called Greeter with one package and one class, called `GreetingFactory`, in the package. You will export `GreetingFactory` so others can use the class.

If you do not have a CVS server to which you can safely connect and you are running on a Windows machine, you can download and install CVSNT from `http://www.cvsnt.org/`. You'll find plenty of instructions and help at the CVSNT site to get you going. However, I ran into problems the first time I configured CVSNT, so be sure to read Appendix B, "Setting Up and Running a Local CVS Server," which goes into some of the setup issues involved in getting CVSNT up and running so that Eclipse can connect to it. Be aware that some of the functionality may not work correctly due to incompatibilities between Eclipse and CVSNT.

Because you are creating this project for the first time, we will create a new repository location and then commit the project files into it.

Select the CVS perspective using either the Perspective button or by selecting Window, Open Perspective, CVS Repository Exploring from the main menu. Once the perspective is visible, right-click in the CVS Repositories view and select New, Repository Location. When the Add CVS Repository dialog appears, enter the name of the machine where you installed CVSNT, the absolute path to the repository, and the username and password to allow you to log in to the repository. Click Finish to complete the connection to your CVS server. If you have a repository path problem, make sure you unchecked the Repository Prefix box in the CVS for NT admin program.

Before you can check in the Greeter project, you need to create it. Go to the Java perspective, create a project named Greeter and create a new class, `GreetingFactory`, in the package `example`. The code for this class should look something like this:

```
package example;

public class GreetingFactory {

    public String greetingsTo(String name) {
        return "Hello, " + name + "!";
    }
}
```

After saving the class, right-click the project name in the Package Explorer view. From the pop-up menu, select Team, Share Project. The Share Project dialog will appear, asking you to choose a repository type for the shared code to reside in. Select CVS and click Next. The next panel in the dialog, the Share Project with CVS Repository panel, needs to know if the code will be shared in an already existing CVS repository or if you are going to add a new repository location. Part of the process of configuring CVSNT is the registration of CVS repositories. If you did not create any repositories, go to Appendix B and read the section titled "Registering a Repository."

Once you have a repository registered, you can do one of two things: You can register the repository location in the CVS perspective ahead of time and then, within the Share Project dialog, select Use Existing Repository Location, or, again from the Share Project dialog, you can select Create a New Repository Location.

If you select Create a New Repository Location, the panel you used to enter the repository information (host, repository path, username, and password) is displayed. In addition, once you've entered the repository information, the dialog's Enter Module Name panel will request a module name under which the project will reside. It will try to best guess your intention by using the existing project name as the module name, but you are also allowed to enter your own module name. For this example, you should use the project name. At this point, clicking Next or Finish accomplishes the same goal: connecting to a repository and starting the first step in checking in the new project.

However, if you chose Use Existing Repository Location instead, clicking Next will take you directly to the Enter Module Name panel, and you can either use the existing project name or enter a new name after selecting Use Specified Module Name. Again, for this example, use the existing project name for clarity.

Clicking Next or Finish will now set up an initial connection between your project and the CVS repository. If you have not clicked Finish, do so now. In addition, the CVS plug-in will compare your incoming code to code that may already be in the repository. In this case, there is nothing to compare, so right-click the project folder in the Synchronize view and select Commit from the pop-up menu. The Add to CVS Version Control dialog appears asking if you want to add the files to CVS because the resources do not appear to be under its control. Click Yes and enter a comment, such as "My first CVS commit." And click OK Select the CVS perspective and open the repository location you selected as the destination for your code. Open the HEAD node and you will see a folder with your project name (unless you chose to name your project something else, in which case the folder will have that name). Open the project folder, also known as the *module* folder, and you will see the same directory structure as your project. The only piece that appears to be missing is the JRE System Library component of your project. In fact, in your actual project, the Package Explorer view combines the JAR files listed in your classpath with your source code to give you a consolidated view of your work. The JAR files are not actually located in your project directory.

The process works the same if you want to check in an arbitrary number of individual files, with only one caveat: Before you check in your file, you should synchronize your changes with the current contents of the branch you are committing to.

For example, someone else can update the very basic `GreetingFactory` you committed with additional code while you have been working on it as well. This would imply the following development steps when using CVS:

1. Check out, or create, a project.
2. Make changes.
3. Synchronize with the CVS repository.
4. Update any code that is in conflict.
5. Commit the files once any conflicts have been resolved.

Return to the Java perspective. Change the `GreetingFactory` code by adding a comment:

```
/**
 * This is a comment.
 */
```

The code should now look like this:

```
package example;

/**
 * This is a comment.
 */
public class GreetingFactory {

    public String greetingsTo(String name) {
        return "Hello, " + name + "!";
    }
}
```

SHOP TALK

Practicing Safe Source Control

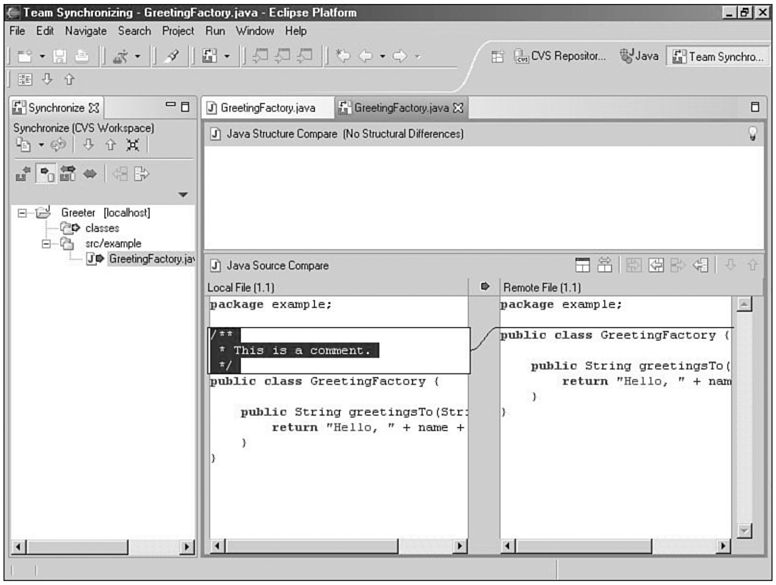
A search on various search engines for “source control best practices” does not yield much in the way of direction for the use of source control in environments where *continuous integration* is the watchword. In practice, if developers on a team can be convinced that source control is useful—and there are many who pay lip service to it but in fact don’t use it or use it badly—then how often it is used becomes the driving force in your process. If your team wants to test/integrate your system every day, then enough code has to churn into the source control system to trigger the checkout of the system so that it can be deployed and tested.

Consider the following when you extract code from source control to perform a bug fix or a feature addition/extension:

- Make sure you also take out the test associated with the class you are about to modify. Add whatever tests you need to confirm that your changes work.
- Although this might be too obvious, you should push only working code back into source control. The code might still break when your scripts perform a full system test, but it should have worked when it left your machine.
- If you are fixing a bug, push back all the files associated with the change, including the tests.
- Consider branching HEAD to make your changes rather than making the changes to HEAD. Although this is a little extra trouble, it allows others the opportunity to work the main source trunk if needed.
- If you are adding/completing functionality, push back all the files/tests for the working functionality. Consider granularity at a use-case level until you find a finer or coarser granularity that you and your team are comfortable with.

Save the file and notice the greater-than signs (>) that appear next to all the names in the project hierarchy in the Package Explorer. The greater-than sign tells you that you have checked out the changed file and, by implication, you are going to check it in at some later point in time.

Before you check in the code, you have to synchronize your version of the file with whatever happens to be out in the branch. You are still using the HEAD branch, so right-click the file in the Package Explorer and go to Team, Synchronize with Repository. The Synchronize CVS Workspace dialog opens, displaying the available resources that can be synchronized. Select the Greeter project and click Finish. A confirmation dialog opens, asking for permission to switch to the Team Synchronizing perspective. Click Yes to open the new perspective. When the Team Synchronizing perspective opens, another dialog opens, reporting the results of the synchronization. In this case, it found two outgoing changes, no incoming changes, and no conflicting changes. Click OK to close the reporting dialog. To the left of the perspective is the Synchronizize view. In this view, open Greeter, `src/example`, `GreetingFactory.java`. Double-click `GreetingFactory.java` to open the Compare editor (see Figure 7.6).



```
package example;

/**
 * This is a comment.
 */
public class GreetingFactory {

    public String greetingsTo(Str
        return "Hello, " + name +
    }
}
```

```
package example;

public class GreetingFactory {

    public String greetingsTo(
        return "Hello, " + nam
    }
}
```

FIGURE 7.6 The Team Synchronizing perspective displaying the Synchronize view and the Compare editor.

The Compare editor is showing you three different views of the file about to go into CVS. The Java Structure Compare view, located at the top of the Compare editor, displays the class structure and the differences between the current structure and the structure of the class coming from the branch to be merged. Any selected elements will appear in Java Source Compare. Any elements flagged in red are conflicts that must be resolved. Because there are no structural changes, the window is empty.

The Java Source Compare view, the bottom half of the Compare editor, displays the workbench file to the left and the branch file to the right. All changes go from right to left. In this example, this is not a problem, but if you had been trying to synchronize a branch, it might be a problem (more on branches later). However, because the change in the workbench code is going to overwrite whatever is in the HEAD branch, you can safely ignore the code differences.

The toolbar on the title bar of the Java Source Compare view can be used to copy conflicting code from right to left for an incoming change, from left to right on an outgoing change, or to display a direction-insensitive file compare. In this case, you don't need to do anything to update the file to contain the changes because they are in the file you are about to commit. The fourth button from the left copies all nonconflicting changes from right to left, and the sixth button copies the current change from the right file to the left file.

For this example, you want to take all the changes from your workbench code (the window on the left) and commit them to the repository. Right-click the `GreetingFactory.java` file in the Synchronize view and select Commit from the pop-up menu. When the Commit dialog appears, enter the comment "My second commit." Click OK. Return to the CVS perspective, right-click in the CVS Repositories view and select Refresh View, and then open your local repository location. Open HEAD, Greeter, src, example. The `GreetingFactory.java` file will be version 1.2. Close all open editors.

Checking Out

In order to proceed through the steps of checking out an existing project, you will now delete your project from the Java perspective and check out as a project the code you committed into the repository. You will then modify your file, check it back in as an individual file, and view the file in CVS to ensure that the check-in was successful.

Keyboard Shortcuts Considered Useful

Remember that Ctrl+F8 will take you to the next open perspective on your workbench. Pressing Ctrl+F8 will toggle you between your current perspective and your last perspective.

Return to the Java perspective. Select project Greeter and delete the project, either by pressing the Delete key or by right-clicking the project name and selecting Delete from the pop-up menu. When the Confirm Delete Project dialog appears, select Also Delete Contents Under [your project path here]? and click Yes. Your code is now gone from the local filesystem.

Return to the CVS perspective. Right-click the Greeter folder located under HEAD and select Check Out As Project from the pop-up menu. Because the project does not exist in the workspace, the checkout proceeds without any more prompting from the system. Had the project already existed and you chose Check Out As Project, the CVS plug-in would have prompted you for permission to delete the destination project so that the code checkout could proceed. If you go back to the Java perspective (press Ctrl+F8), you will find the Greeter project is back safe and sound.

Also notice that the project name has a greater-than sign (>) as its prefix. This means that the project was taken out of source control.

Now that you have successfully checked in and checked out your project, bear in mind that all the work you have accomplished so far has been to the HEAD branch of your repository. At some point you will need to make changes to your code that you can commit without affecting HEAD. This is where branching comes in.

Branching

In the Java perspective, right-click the Greeter project and select Team, Branch. Enter the branch name **Greeter_1_0**, leave the check box Start Working in the Branch checked, and press Enter or click the OK button. The string "Greeter_1_0" and the name of your CVS server appearing within square brackets should now follow your project name. Return to the CVS perspective. Anywhere within the CVS Repositories view, right-click and select Refresh View from the pop-up menu, or press F5. A plus sign will appear next to the Branches node. Open the Branches node and you will find a branch called Greeter_1_0. If you open branch Greeter_1_0, you will find a folder named after your project with the branch name displayed next to it.

The Use Of Branch Naming Conventions

Not to belabor the obvious, but branch naming conventions are important. At the very least, enter a branch name that will not conflict with any other branch name in your repository.

Branch Greeter_1_0 is where your code will go when you update it and commit it to the repository. Return to the Java perspective. Add the following code to `GreetingFactory`:

```
private String _greeting = "Hello, ";

/**
 * @param string
 */
public void setGreeting(String string) {
    _greeting = string;
}
```

You should insert the code above `greetingsTo()`. Save the code, right-click the filename in the Package Explorer, and select Team, Commit from the pop-up menu. When the Commit dialog appears, enter the comment "Added setGreeting()." Click OK.

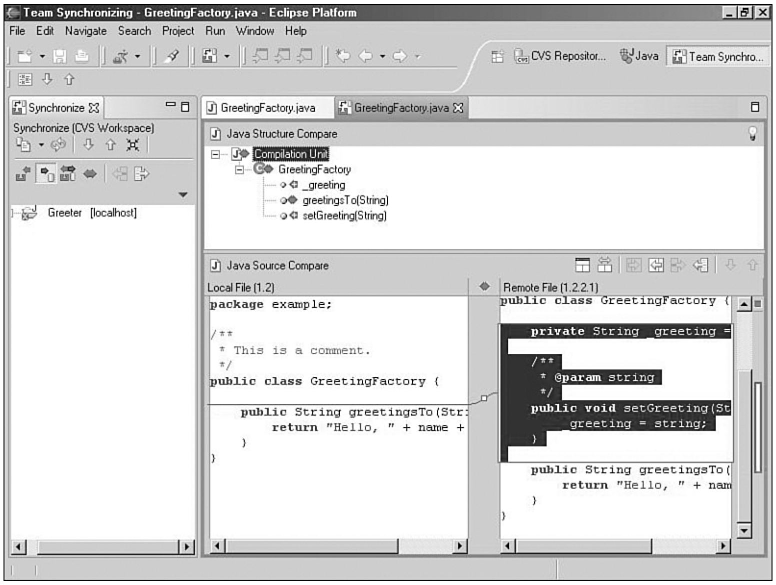
Press Ctrl+F8 to return to the CVS perspective. Refresh the view by pressing F5. Open Branch, Greeter_1_0, Greeter Greeter_1_0, src, example. The version number for `GreetingFactory.java` is different (higher) from the version number of `GreetingFactory.java` under HEAD.

Merging a Branch

Your next task is to take the code change you made to `GreetingFactory` and move it into HEAD. In order to accomplish the move to HEAD, you are going to overwrite the code in the workbench and merge it with the code for your custom branch. This is a safe procedure because the updated code is, in fact, in branch Greeter_1_0.

Return to the Java perspective. Right-click the project name in the Project Explorer view and go to the Replace With, Another Branch or Version item. When the Select with Branch or Version dialog appears, select HEAD and click OK. The CVS plug-in has now overwritten the changes you made, but the changes are still in Greeter_1_0. Now that the workbench reflects the code from HEAD, you will update the workbench code with the changes from the branch.

Right-click `GreetingFactory.java` and select Team, Merge from the pop-up menu. Select as the starting point Root_Greeter_1_0 and click Next. In the next panel, open the Branches node, select Greeter_1_0, and click Finish. The Confirm Perspective Switch dialog opens, asking to switch to the Team Synchronizing perspective. Click Yes. In the Editor view of the Team Synchronizing perspective, the Compare editor will be open with information displayed in all three Compare editor windows (see Figure 7.7). The Java Structure Compare window, the one at the top of the Compare editor, allows you to compare the file in the branch with the source in the workbench, which matches the source in HEAD. To make it easier to compare the files, double-click the title bar of the Compare editor to fill the screen.



Local File (1.2):

```
package example;

/**
 * This is a comment.
 */
public class GreetingFactory {

    public String greetingsTo(Str
        return "Hello, " + name +
    }
}
```

Remote File (1.2.2.1):

```
public class GreetingFactory {

    private String _greeting =

    /**
     * @param string
     */
    public void setGreeting(St
        greeting = string;
    }

    public String greetingsTo(
        return "Hello, " + nam
    }
}
```

FIGURE 7.7 The Team Synchronizing perspective's Compare editor with the HEAD and branch versions of `GreetingFactory.java`.

The Java Structure Compare window shows you the differences in the structure of your class definition. In this case, the blue arrows with the embedded plus signs mean that the branch has changes that need to be incorporated into the workbench code. Once again, the Java Source Compare window is directly below the Structure Compare window. The left window is the workbench code, and the right window is the branch code, which contains the changes you made. In this case, the HEAD code is to the left and the branch code is to the right. However, bear in mind that the changeable code is *always* going to be in the left window. It is because of this that you overwrite the workbench code with the target branch that does not contain your changes, and you compare it to the branch that contains the actual changes. The color code used by the two windows is as follows:

- Incoming changes are blue.
- Outgoing changes are gray.

Simply update the workbench files with the changes committed to the branch. Click the sixth button, Copy Current Change from Right to Left, to update the workbench file with your changes. If more changes were available, you would continue adding changes to the workbench file until no more changes were available. Save the file and shrink the editor window back to normal by double-clicking the title tab.

The final step in updating the HEAD branch is the actual commit to the repository. This is no different from what you did earlier when you checked in the project. However, in this example you are only going to enter one file. Right-click `GreetingFactory.java` and select Commit. Enter a comment into the Commit dialog and click OK. Return to the CVS perspective and press F5 to refresh the CVS server view. Note that `GreetingFactory` has had its version number updated.

In Brief

The CVS perspective is a quick-and-easy way to connect to a CVS server. The sharing of files is a trivial task, and support for combining out-of-sync files is intuitive.

You were able to accomplish a number of CVS-related tasks:

- The creation of a new repository entry is handled by using the Add CVS Repository Wizard.
- The CVS Repositories view lists the various CVS servers with whose repositories you can interact. The editor area will use an appropriate editor to display a file, but the file is read-only. The CVS Resource History allows you to see the CVS audit trail of any given file.
- The check-in process involves synchronizing with the repository to ensure that no one else has changed the file since you took it out and, after you correct any changes, committing the file into the repository after you enter a descriptive comment about any changes made.

- The check-out process involves finding a module of folder in the CVS repository and selecting Check Out As a Project.
- Branches are created by selecting a project or a file and choosing Team, Branch from the Java perspective Package Explorer view.
- File merges are accomplished by committing your current changes to the non-HEAD branch, checking out the HEAD branch, and selecting the branch and HEAD as the merge source and target, respectively. After reconciling all file differences, you can safely commit your files to HEAD.