CHAPTER 3

Revision Control: Subverting Your Code

At one point or another in your days as a developer, chances are you've mistakenly deleted an important file. It happens. Sometimes the problem is worse than you thought; for instance, you might not even be sure what files you deleted. Or perhaps your brilliant idea for a new feature has horribly broken the code base. Making matters worse, your changes were spread across multiple files, and now it's unclear how to return to the previous state. Version control can remedy all of these problems by coordinating the life cycle of all files in your project, allowing you to not only recover mistakenly deleted code, but actually revert back to earlier versions.

Version control can go well beyond simple file management and recovery, though; it also plays a crucial role in managing changes made in environments where multiple developers might be simultaneously working with the code. Sure, each of you could make copies of the code base and yell over the cubicle wall, "Hey, I'm working on `tools.py` right now, don't touch it." But sooner or later, you'll nonetheless overwrite each other's changes. It gets worse when you're not within earshot, or even the same time zone.

Revision control helps this situation by acting as a moderator and a single source of truth. Either by gating access or merging changes, it prevents you from stepping on each other's toes. Revision control keeps track of what changes were made, and further, it keeps track of who made them.

Revision control also lets you work on multiple versions of the code at the same time, allowing you to test out that ambitious new feature without interfering with the stable version. This encourages all sorts of efficiencies, allowing one developer to add new features for an upcoming release while another developer works on security fixes for the current release. When you are ready, the changes can be merged back together.

The benefits of coordination aren't limited to humans, though. You can configure your build process to execute against the source repository and cause the build to begin anew any time somebody checks in new code.

You can also use revision control to enforce policy. For instance, you can prevent users from checking in changes to certain branches of the tree, analyze code before allowing it to be submitted, ensure that all Python code has proper whitespacing, or require that all Python files are syntactically correct. All of this is made possible by revision control.

Subversion is one of the most widely used revision control systems available. In this chapter, I'll show you how to use Subversion to manage your code on your local machine, both from the command line and from within Eclipse via the Subversive plug-in. The examples

include such common operations as adding, editing, and removing files, but they also include operations that don't immediately spring to mind. Among these are comparing your local changes with those in the revision system, retrieving others' work from the repository, and resolving conflicts between changes you have made and changes that others have made.

Revision Control Phylum

We can look at revision control systems in a couple of broad aspects. The most significant of these is distributed vs. centralized. Another is availability. Is the repository available locally or remotely? I'm not even going to mention revision control systems that are local. Many of the practices in this book are intended to scale up to multiple machines, so a local repository just doesn't work for us.

Centralized revision control systems have been around forever. They access a single logical repository that is physically stored on one or more systems. Most commercial systems are centralized, and centralized systems seem to be the most mature. Examples of centralized revision control systems are CVS, Visual SourceSafe, Subversion, Perforce, and ClearCase.

Distributed revision control systems are the new kid on the block. To date, their most highly visible implementations have been related to operating system kernel development. Both Linux and Solaris use the distributed repository Git, which was created to support development of Linux. Examples of distributed revision control systems are Darcs (`darcs.net/`), BitKeeper (`www.bitkeeper.com/`), Mercurial (`www.selenic.com/mercurial/wiki/`), Git (`git.or.cz/`), and Bazaar (`bazaar-vcs.org/`). They're pretty cool in some conceptual ways, but many release engineering professionals look on them warily. Despite their complexities, kernels are still simple and well-understood entities compared to many enterprise systems, and distributed version control systems have yet to prove themselves in the more complicated enterprise environments.

If you look on the Web, you'll see vociferous arguments about which kind of revision control system is better. Much of this seems to be driven by people's experience with CVS. Advantages are touted for distributed revision control that when examined closely boil down to "Our software doesn't suck like CVS." Claims are made about branch creation, labeling, or merging that boil down to "CVS did it like this. CVS is a centralized revision control system. Therefore, all centralized revision control systems do it like this." Examples of where this logic is applied include branching and merging. Almost never are the free systems compared to the commercial systems.

The commercial systems are impressive. In general, they're more mature and feature rich than the free systems. They offer administrative controls and reporting that is missing from the free systems. They do branching and merging well, too. Perforce particularly shines in this area, and its integration tools are impressive. However, we're not going to be using Perforce in this book.

We'll be using Subversion. The choice is driven by a number of factors. First, Subversion is widely used. As with Eclipse, there is a large ecology of tools associated with it. The tools we've worked with and will be working with later easily integrate with it. And it's free.

Subversion supports atomic commits of multiple files. There is a global revision counter allowing you step back to any specific point in the repository's history. It supports labeling and branching. If some of these terms don't make sense to you right now, they will shortly.

What Subversion Does for You

Subversion stores your code on a central server in a repository. The repository acts much like a filesystem. Clients can connect to the filesystem and read and write files. What makes the repository special is that every change ever made to any file in the repository is available. Even information such as renaming files or directories is tracked.

Clients aren't limited to looking at the most recent changes. They can ask for specific revisions of a file, or information like, "who made the third change last Thursday?" This is where the real utility of a revision control system comes from.

A user checks out code from the repository, makes changes of one sort or another, and then submits those changes back to the repository. Multiple users can be doing this at the same time. Two or more users can check out the same file and edit it, and when the file changes are submitted, they'll have to resolve any conflicts. This resolution is called *merging*.

The overall process is called *edit-and-merge*. Contrast this with the other approach, called *exclusive locking*. In this scheme, only one person gets to have a file open for edit at any time. While it saves the possible work of merging changes, it can bring development to a halt. It turns out that in practice, edit-and-merge is the least disruptive.

What happens if two users try to submit changes at the same time? One goes first. In Subversion, groups of files are submitted together. The submissions are a single atomic action. While CVS has interfaces that allow you to submit multiple files at once, each file is an individual submit. Two users can submit sets of files, and their changes will be interleaved. This can never happen with Subversion.

Subversion maintains a global revision counter that is incremented with every submission. It increases monotonically, and it can be thought of as describing the state of the repository at any point in time. While it may not seem like much, having this counter is remarkably useful for labeling builds and releases.

Subversion stores working copies of the files on your disk. It stores the the information describing these working copies on your local system too. This contrasts with other systems that store this state on a server. Subversion doesn't need to contact the server to find out the current state of your files, allowing you to work remotely without a network connection. The bad news is that you must be connected to rename or copy files, which takes away from the joy.

The local state is stored in directories named `.svn` (just like CVS uses `.cvs` directories). There is one in every directory checked out from Subversion. Many refer to these directories as "droppings." The `.svn` directories carry virgin copies of all files in your working copy. This way, the more frequently invoked commands, such as diff and revert, can be run without accessing the central repository.

Frequently, there is a need to work on multiple differing copies of a project. Consider a software product that has an installed base of users. At most points in a software product's life, there will be multiple activities going on. Some developers will be working on new features for upcoming releases. Other developers will be working on high-priority repairs for customers who have already installed the product. The new features will destabilize the codeline and often mask the bugs that are reported by customers. They'll also introduce many new bugs, particularly early in the development cycle. High-priority bug fixes must be made to code that mirrors the release code as closely as possible so that the customer doesn't receive a version of the product that is broken in yet more new and interesting ways.

Sadly, both the new development and bug fixes must be performed simultaneously. This is done by creating copies of the program. One copy is used for the new work, and the other is used only for the bug fixes. These copies are referred to as *branches*. Branches are independent but related copies of a program. A new branch can be made whenever simultaneous but conflicting changes must be made to a program.

In practice, managing branching is one of the primary jobs of a revision control system. As branches proliferate, it is necessary to have some way of referring to them. This is done with *labels*. Labels are names attached to branches at a particular point in time. They let you precisely and concisely specify a version of a program.

The new release will require the bug fixes from the maintenance branch, so the branches will need to be recombined. This process is called *merging*. This is an important part of branch management. Merging takes the changes from one branch and combines them with another branch. A surprisingly large part of the process can be automated, and the results work a surprisingly large percentage of the time, but ensuring that they work requires good tests, and the process almost always requires some developer intervention.

Subversion supports branching—that's the good news. The bad news is that merging support is very new. It was just added in version 1.5, and it has yet to be widely deployed.

That brings us to labeling. Subversion supports labeling. Kinda. Labeling is just branching to a different place. The good news is that we have the global revision counter, which allows us to bypass labeling to some degree.

Getting Subverted

The first step is installing Subversion. Subversion is available from `http://subversion.tigris.org/`. If you're running on Linux and you installed your system with development tools included, then the odds are good that you've already got Subversion installed. If Subversion is not installed, chances are that packaged binaries can be located for your system at `http://subversion.tigris.org/project_packages.html`, and if worse comes to worst, the source code is also available there.

Once Subversion is installed, the first step in creating your repository is initializing the database on your Subversion server:

```
phytoplankton:~ jeff$ svnadmin create /usr/local/svn/repos
```

This creates the Subversion database in the directory `/usr/local/svn/repos`. There are two ways of storing this information. One is on the filesystem, and the other is in Berkeley DB database files. The default is within the filesystem, and unless you have good reason to do otherwise, I suggest taking the default. You can find more information in *Practical Subversion, Second Edition*, by Daniel Berlin and Garret Rooney (Apress, 2006). The directory structure that will be created looks something the following:

```
$ ls -F /usr/local/svn/repos
```

```
README.txt      dav/            format          locks/
conf/           db/             hooks/
```

■ **Note** You may need to create the directory `/usr/local/svn` before you can run this command, and you may also need to set your permissions appropriately. I had to change ownership to my own account. If I were running this in production, it would be owned by the `svn` user.

Subversion repositories can be accessed in multiple ways. The path to and within the repository is specified using a URL (see Figure 3-1). The URL scheme (the part before the first colon) specifies the access protocol. This can be through the local filesystem, HTTP or HTTPS, SSH, or Subversion's own protocol.

The scheme you use will depend on the server that you're accessing. The easiest is the file protocol. It can only be used when you're on the same machine as the Subversion server. The HTTP and HTTPS protocols require the use of Apache. You gain a huge amount of flexibility in access control by using Apache, but the setup is more complex. The Subversion protocol is somewhere in between. It uses a dedicated server that is very easy to set up, and it offers some level of access control. The protocol is faster than using HTTP or HTTPS for large projects. We're going to be using the file protocol for the examples in this section of the book.

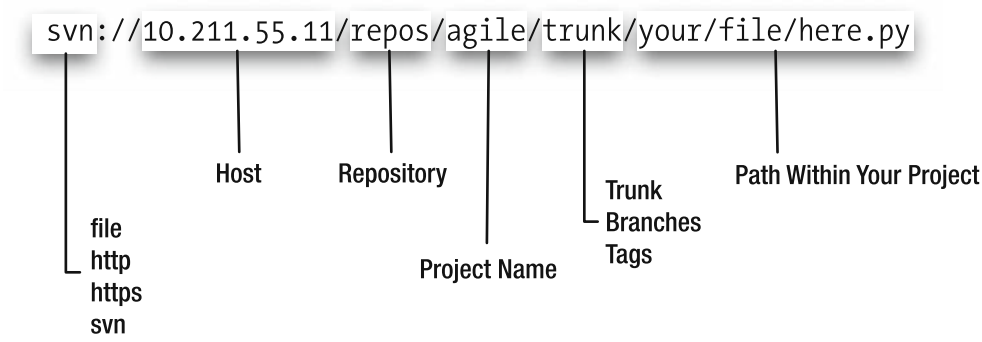

Figure 3-1. *Parts of a Subversion URL*

The process of loading a project into Subversion involves several steps. The first is the creation of a repository, which you've already done. A repository can hold any number of projects, and these projects can be organized in any number of ways. You have to decide how you're going to do that. Then you have to create those directories, and finally you'll be able to import the project into Subversion.

In most working environments, there are multiple projects within a single repository. This requires some level of organization. Generally, these projects have a mainline containing the gold version of the code. They have a number of branches where conflicting work is performed, and they have a place for tags. (*Tag* is Subversion's term for a label.) By convention, the main codeline is stored in a directory called `trunk`, branches are stored in a directory named `branches`, and tags are stored in a directory named `tags`. We'll stick with that convention.

There are two common conventions for organizing projects. One is *project major*, and the other is *project minor*. In project major, each project has its own `trunk`, `branches`, and `tags` directories. In project minor, the repository has top-level `trunk`, `branches`, and `tags` directories. Beneath each of these is a directory for each project, as shown in Figure 3-2.

Project Major Organization	Project Minor Organization
/path/to/repository	/path/to/repository
/path/to/repository/project1	/path/to/repository/trunk
/path/to/repository/project1/trunk	/path/to/repository/trunk/project1
/path/to/repository/project1/branches	/path/to/repository/trunk/project2
/path/to/repository/project1/tags	/path/to/repository/branches
/path/to/repository/project2	/path/to/repository/branches/project1
/path/to/repository/project2/trunk	/path/to/repository/branches/project2
/path/to/repository/project2/branches	/path/to/repository/tags
/path/to/repository/project2/tags	/path/to/repository/tags/project1
	/path/to/repository/tags/project2

Figure 3-2. *Project major and project minor organization*

I prefer project major organization. It makes it easy to identify what belongs to a project, it makes access control easier to manage, and it allows you to move your project about with very few commands. Our project is named agile. With project major organization, our directories structure will look like this:

```
/usr/local/svn/repos/agile
/usr/local/svn/repos/agile/trunk
/usr/local/svn/repos/agile/branches
/usr/local/svn/repos/agile/tags
```

You create this with the command `svn mkdir`. Once you've created the directories, you can look at them with the `svn list` command:

```
phytoplankton:~ jeff$ svn mkdir file:///usr/local/svn/repos/agile \
  -m "creating the internal organization for the project 'agile'"
```

```
Committed revision 1.
```

```
phytoplankton:~ jeff$ svn list file:///usr/local/svn/repos/agile
phytoplankton:~ jeff$ svn mkdir \ file:///usr/local/svn/repos/agile/trunk ➥
file:///usr/local/svn/repos/agile/branches \ file:///usr/local/svn/repos/agile➥
/tags -m "creating the internal organization for the project 'agile'"
```

```
Committed revision 2.
```

```
phytoplankton:~ jeff$ svn list file:///usr/local/svn/repos/agile
```

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

Now you can import the mainline into the depot. This is done with the `import` command. The `import` command takes three arguments:

- `Agile` is the imported directory.
- The `file:` URL is the destination in `trunk`.
- The `-m` option is the `commit` comment.

The contents of the directory `agile` will be loaded into the Subversion trunk. The directory `agile` itself will be omitted:

```
phytoplankton:~ jeff$ cd ~/ws
phytoplankton:~/ws jeff$ svn import agile \
 file:///usr/local/svn/repos/agile/trunk \
 -m "Initial import of our the 'agile' trunk"
```

```
Adding         agile/.project
Adding         agile/src
Adding         agile/src/examples
Adding         agile/src/examples/__init__.py
Adding         agile/src/examples/greetings
Adding         agile/src/examples/greetings/__init__.py
Adding         agile/src/examples/greetings/standard.py
Adding         agile/.pydevproject

Committed revision 3.
```

```
phytoplankton:~/ws jeff$ svn list \ file:///usr/local/svn/repos/agile/trunk
```

```
.project
.pydevproject
src/
```

You have imported the `.project` and `.pydevproject` files that Eclipse created. These files are as important as any other source files. As you create larger and more complicated projects, these files will contain more and more information that you don't want to lose. When a developer checks out a file from Subversion the first time, they will be able to import it directly into Eclipse. They'll be working on the code rather than figuring out how get the code to build under Eclipse.

Working with Your Subverted Code

At this point, you've imported your code into Subversion, but you don't have a working version on your local machine. You can't add new files, edit files, delete files, or update from the repository until you get a local copy.

Your local copy can't be pulled directly into your workspace directory. Subversion will detect that the files already exist. You need to do one of two things: either move aside your current project directory or pull the code down into your existing directory. In this case, choosing a new project is what you'll do next:

```
phytoplankton:~/ws jeff$ svn checkout \ file:///usr/local/svn/repos/agile/trunk ➥
hello
```

```
A    hello/.project
A    hello/src
A    hello/src/examples
A    hello/src/examples/__init__.py
A    hello/src/examples/greetings
A    hello/src/examples/greetings/__init__.py
A    hello/src/examples/greetings/standard.py
A    hello/.pydevproject
Checked out revision 3.
```

```
phytoplankton:~/ws jeff$ ls -la hello
```

```
total 16
drwxr-xr-x   6 jeff  jeff  204 Oct  2 18:51 .
drwxr-xr-x   5 jeff  jeff  170 Oct  2 18:51 ..
-rw-r--r--   1 jeff  jeff  359 Oct  2 18:51 .project
-rw-r--r--   1 jeff  jeff  307 Oct  2 18:51 .pydevproject
drwxr-xr-x   8 jeff  jeff  272 Oct  2 18:51 .svn
drwxr-xr-x   4 jeff  jeff  136 Oct  2 18:51 src
```

The first thing to notice is the `.svn` directory. Each directory checked out from Subversion will contain one. This is where Subversion stores information describing the state of your local system. It contains a record of each file that has been checked out and a copy of that file.

You've already seen how to perform a few common operations. You've made directories in the repository; you've listed the contents of a directory; and you've looked at the contents of a file. I'll run through the rest of the operations you'll routinely perform with Subversion. These are the operations that every user needs. They include the following:

- Examining your working copy
- Adding a file
- Deleting a file
- Reverting a file
- Committing changes
- Editing a file
- Comparing a file against the repository
- Updating your working copy
- Resolving conflicts during a submission

These operations form the core of what you'll be doing from day to day. They will carry over almost directly to the Eclipse interface. We'll start by examining the files.

Examining Files

There are two commands that are used to examine the state of your workspace. They are `svn info` and `svn status`. `svn info` works on individual files and directories. `svn status` works on your workspace as a whole. `svn status` is used more frequently than `svn info`, but there are times when you need information that is only available through `svn info`, so we'll start there.

```
phytoplankton:~/ws jeff$ cd hello
phytoplankton:~/ws/hello jeff$ svn info
```

```
Path: .
URL: file:///usr/local/svn/repos/agile/hello
Repository Root: file:///usr/local/svn/repos
Repository UUID: 74a71bd7-8c3b-0410-b727-f8ad94e0a8f0
Revision: 3
Node Kind: directory
Schedule: normal
Last Changed Author: jeff
Last Changed Rev: 3
Last Changed Date: 2007-10-02 18:46:37 -0700 (Tue, 02 Oct 2007)
```

```
phytoplankton:~/ws/hello jeff$ svn info .project
```

```
Path: .project
Name: .project
URL: file:///usr/local/svn/repos/agile/hello/.project
Repository Root: file:///usr/local/svn/repos
Repository UUID: 74a71bd7-8c3b-0410-b727-f8ad94e0a8f0
Revision: 3
Node Kind: file
Schedule: normal
Last Changed Author: jeff
Last Changed Rev: 3
Last Changed Date: 2007-10-02 18:46:37 -0700 (Tue, 02 Oct 2007)
Text Last Updated: 2007-10-02 18:51:35 -0700 (Tue, 02 Oct 2007)
Checksum: 97703150e87f434355444a9f07b6750b
```

Notice that Subversion tracks the directory itself. This is reported in the `Node Kind` field. This differs from some other version control systems that only track files. The really important field here is `Revision`. It lets you know what edition of a file the system thinks you have. You can get this information for all files using the `svn status` command.

Run without arguments, `svn status` reports changed files that have not been committed. You have no changed files at this moment, so it would report nothing. You're interested in seeing the verbose output, which shows all files. You turn this on with the `-v` flag:

```
phytoplankton:~/ws/hello jeff$ svn status -v
```

```
                3        3 jeff         .
                3        3 jeff         .project
                3        3 jeff         src
                3        3 jeff         src/examples
                3        3 jeff         src/examples/__init__.py
                3        3 jeff         src/examples/greetings
                3        3 jeff         src/examples/.../__init__.py
                3        3 jeff         src/examples/.../standard.py
                3        3 jeff         .pydevproject
```

You can't tell easily, but there are a number of blank fields ahead of the first numbers. The four remaining fields are the working revision, the head revision, the author committing that head revision, and finally the path to the file. This information will become more interesting as you work. Now that you know how to look at your workspace, you can move on to making changes.

Adding Files

Suppose that you've created a new file named `src/examples/common.py`, and you want to add this file to the repository. You do this with the `svn add` command. It works pretty much as you'd expect. We'll look at its effects using the `svn status` command:

```
phytoplankton:~/ws/hello jeff$ svn add src/examples/common.py
```

```
A         src/examples/common.py
```

```
phytoplankton:~/ws/hello jeff$ svn status
```

```
A      src/examples/common.py
```

```
phytoplankton:~/ws/hello jeff$ svn status -v
```

```
...
                3        3 jeff         src/examples
A               0       ?   ?          src/examples/common.py
                3        3 jeff         src/examples/__init__.py
...
```

Notice that `status -v` shows an `A`, which denotes a file to be added. It shows that the current revision is `0`, which denotes that there's no revision on the client and that the head revision and head author don't exist. This demonstrates something important about Subversion. Adding a file doesn't immediately add the file to the repository. It adds it to the list of pending changes. In SVN parlance, this is known as *scheduling an add for commit*. You have to use `svn commit` to complete the addition.

```
phytoplankton:~/ws/hello jeff$ svn commit -m "Adding common code for all greetings"
```

```
Adding         src/examples/common.py
Transmitting file data .
Committed revision 4.
```

```
phytoplankton:~/ws/hello jeff$ svn status -v
```

```
...
                3        3 jeff         src/examples
                4        4 jeff         src/examples/common.py
                3        3 jeff         src/examples/__init__.py
...
```

Now that the change is committed, you can see that the file has been added to the repository. The file was committed in revision 4, and you have that revision in your working copy.

Copying and Moving Files

Unlike several other revision control systems, Subversion has simple commands for copying and moving files. These commands maintain revision history and ancestry between the source and destinations. We'll copy `common.py` to `shared.py`:

```
$ cd src/examples
$ svn copy common.py shared.py
```

```
A         shared.py
```

```
$ svn status
```

```
A  +   shared.py
```

```
$ svn commit -m "Copying common.py to shared.py"
```

```
Adding         examples/shared.py

Committed revision 5.
```

You'll notice that `svn status` returns `A +`. The `+` indicates that revision history is being maintained from the original to the copy. A similar process happens with a move:

```
$ svn move shared.py unshared.py
```

```
A         unshared.py
D         shared.py
```

```
$ svn status
```

```
A  +   unshared.py
D      shared.py
```

```
$ svn commit -m "Moving shared.py to unshared.py"
```

```
Deleting       examples/shared.py
Adding         examples/unshared.py

Committed revision 6.
```

In this case, there are two changes that are performed. The line beginning with `A +` indicates that `unshared.py` was added while maintaining history, and the line beginning with `D` indicates that the original file `shared.py` was deleted.

This is also the first time you've seen multiple changes at once. Unlike CVS, both of these changes are performed in a single atomic transaction. At no point is there a moment where both files exist. To the outside world, it is as if the copy and delete happened simultaneously.

Deleting Files

The `svn delete` command schedules files for removal. The `svn status` command shows these prefixed with `D`. These changes become permanent when they are committed.

```
$ svn delete common.py unshared.py
```

```
D         common.py
D         unshared.py
```

```
$ svn status
```

```
D      common.py
D      unshared.py
```

```
$ svn commit -m "Removing common.py and unshared.py"
```

```
Deleting       examples/common.py
Deleting       examples/unshared.py

Committed revision 7.
```

Reverting Changes

Now is a good moment to examine what is happening on the file system when we delete a file. We're going to delete `__init__.py`. Don't worry too much, though—we're going to resurrect it.

```
$ ls
```

```
__init__.py     greetings
```

```
$ svn delete __init__.py
```

```
D         __init__.py
```

```
$ svn status
```

```
D      __init__.py
```

```
$ ls
```

```
greetings.py
```

The important thing to notice at this point is that the operation has already taken place on the filesystem. Subversion makes the changes to the working copy before they are committed to the repository. Your working copy is what the repository will look like after you commit your changes. Now we're going to undo those changes:

```
$ svn revert __init__.py
```

```
Reverted '__init__.py'
```

```
$ svn status
$ ls
```

```
__init__.py     greetings
```

As you can see, `__init__.py` has been restored to the working copy. This resurrected copy was pulled from the `.svn` directory contained within the working directory. The delete was also removed from the pending changes listed by `svn status`. `revert` works for all kinds of local changes, including adds, copies, moves, deletes, and modifications.

Modifying a File

Making changes to existing files is the real meat of daily work. It is not necessary to explicitly open a file in Subversion. All files are considered to be fair game for editing. We've made some changes to the file `src/examples/greetings/standard.py`. `svn status` shows that we've modified the file:

```
$ svn status
```

```
M      greetings/standard.py
```

The `M` indicates that the file has been modified. This is determined by comparing the working copy with the stored revision in one of the `.svn` directories. Because it is performed against a locally stored copy, you can run this even if you're disconnected from the server. You can find out what changes were made by using the `svn diff` command:

```
$ svn diff greetings/standard.py
```

```
Index: greetings/standard.py
--- greetings/standard.py       (revision 7)
+++ greetings/standard.py       (working copy)
@@ -1,6 +1,7 @@
#!/usr/bin/python
    class HelloWorld(object):
        """Simple hello world example"""

        def main(self):
           print "Hello World!"
```

The diff shows that the comment `"""Simple hello world example"""` was added. As with the status request, the diff is done against the locally stored copy, and it can be performed even when disconnected from the server. If you were dissatisfied with the changes, you could revert them using `svn revert`, but you're satisfied, so you commit it:

```
$ svn commit -m "Adding doc string to HelloWorld"
```

```
Sending        examples/greetings/standard.py
Transmitting file data .
Committed revision 8.
```

Updating Your Working Copy

Outside of your local development environment there will be multiple people working with the repository. The code will be changing. The longer your project stays out of the trunk, the further it will diverge from the code in the repository. It is important to get these changes into your working copy. It is best to do this before committing changes. This is done with the `svn update` command.

Now suppose that someone has edited the file `standard.py` since you did. Another developer modified the file and it was committed as revision 9. You can find this out using the command `svn status -u`:

```
$ svn status -u
```

```
       *        8   src/examples/greetings/standard.py
Status against revision:      9
```

This shows that your working copy of `standard.py` is out of date. This is indicated by the * in the first column. The 8 indicates that you have revision 8, and the line `Status against revision: 9` indicates that revision 9 is the most recent revision.

You can look at the differences using `svn diff -r BASE:HEAD`. This shows all the differing files reported in `svn status -u`.

```
$ svn diff -r BASE:HEAD
```

```
Index: src/examples/greetings/standard.py
===================================================================
--- src/examples/greetings/standard.py  (working copy)
+++ src/examples/greetings/standard.py  (revision 8)
@@ -4,6 +4,8 @@
     """Simple hello world example"""

     def main(self):
+        """Someone else added a comment here"""
+
         print "Hello World!"

 if __name__ == '__main__':
```

You can pull down the most recent revision with the `svn update` command. With no arguments, this pulls down all updates to your working copy.

```
$ svn update
```

```
U    src/examples/greetings/standard.py
Updated to revision 9.
```

Conflicting Changes

Now I'll make a change to `standard.py`. It will return an exit code upon completion. The new lines are displayed in bold.

```
#!/usr/bin/python

import sys

class HelloWorld(object):
    """Simple hello world example"""
```

```
    def main(self):
        """Print message and terminate with exit code 0"""

        print "Hello World!"
        sys.exit(0)

if __name__ == '__main__':
    HelloWorld().main()
```

While this change was made, another developer submitted revision 10. Revision 10 changes the doc string for `main()`.

```
$ svn commit -m "Exit codes are explicitly returned"
```

```
Sending        src/examples/greetings/standard.py
svn: Commit failed (details follow):
svn: Out of date: '/agile2/trunk/src/examples/greetings/standard.py' in ➥
transaction '10-1'
```

This is the usual way that you'll discover something has changed. You'll try to submit and it will fail. Nothing has changed on the filesystem, though. You've just been warned that the commit couldn't happen. You can use the commands `svn status -u` and `svn diff -r BASE:HEAD` to see what has changed.

There is another command that lets you look at the changes to be committed. This command is `svn log -r BASE:HEAD`. It shows the changes between the base revision (from your last update) and the head revision in the repository:

```
phytoplankton:~/ws/agile jeff$ svn log -r BASE:HEAD
```

```
------------------------------------------------------------------------
r9 | doug | 2007-10-09 13:08:23 -0700 (Tue, 09 Oct 2007) | 1 line

Added doc string to HelloWorld.main()
------------------------------------------------------------------------
r10 | doug | 2007-10-09 13:08:25 -0700 (Tue, 09 Oct 2007) | 1 line

Updated doc string for HelloWorld.main()
------------------------------------------------------------------------
```

`svn status` will show that `standard.py` is the only file that changed, and `svn diff` will show that the comment is correct. Now you have to merge the changes together. You do this with the commands `svn update` and `svn merge`:

```
phytoplankton:/tmp/am1/src/examples/greetings jeff$ svn update
```

```
C    standard.py
Updated to revision 10.
```

This brings down the most recent changes, as before—but notice the status line for `standard.py`. It begins with `C`, which indicates a conflict. You have to resolve the changes. Subversion has created four versions of the conflicting file that will be helpful in this process.

```
phytoplankton:/tmp/am1/src/examples/greetings jeff$ ls
```

```
__init__.py             standard.py             standard.py.mine
standard.py.r10         standard.py.r9
```

- `standard.py` is the candidate merge.
- `standard.py.mine` is the version that I just made.
- `standard.py.r9` is the virgin working copy before I made my changes in `standard.py.mine`.
- `standard.py.r10` is the conflicting head revision.

The really important file here is `standard.py`, the candidate merge. The other files exist for use with external diff tools.

In the candidate merge, Subversion has spliced together your version and the head revision. Lines that have changed in one but not the other have been added to the file. The changed lines replace the unchanged lines. Generally, changes that don't overlap lines don't overlap in functionality, so simply splicing in the changed sections is a surprisingly effective algorithm for automatically merging code. The resulting code functions in most cases. In fact, it's eerie how often the merged code results in a functioning program.

The problem arises with lines that have changed in both files. There's no automatic way to merge together these conflicting blocks. When this happens, Subversion defers to the developer's judgment. The conflicting blocks of lines are both included in the merge candidate `standard.py`. They are separated with markers indicating their source. Your copy is first, and the head revision is second. It is up to you to make the appropriate changes.

```
$ more standard.py
```

```
#!/usr/bin/python

import sys

class HelloWorld(object):
    """Simple hello world example"""

    def main(self):
<<<<<<< .mine
        """Print message and terminate with exit code 0"""
=======
        """Someone updated the doc string"""
>>>>>>> .r10
```

```
        print "Hello World!"
        sys.exit(0)

if __name__ == '__main__':
    HelloWorld().main()
```

You'll edit `standard.py` until it looks like you want it to, and then you'll tell Subversion that the merge is complete using the command `svn resolved`. Once Subversion knows that you've resolved the conflicting files, you can submit the changes.

```
$ vi standard.py
```

```
[... resolve conflict manually...]
```

```
$ more
```

```
#!/usr/bin/python
...

    def main(self):
        """Print message and terminate with exit code 0"""

        print "Hello World!"
...
```

```
$ svn resolved standard.py
```

```
Resolved conflicted state of 'standard.py'
```

```
$ svn commit -m "Exit codes are explicitly returned"
```

```
Sending        greetings/standard.py
Transmitting file data .
Committed revision 11.
```

Merging code can be one of the more onerous tasks. The longer between merges, the more changes accumulate. The more changes that accumulate, the more likely conflicts are to arise. The more conflicts you have at any one time, the more work to be done when merging. The more changes that have been made, the likelier functionality is to break, too.

The key to keeping merges simple is to merge often. The agile practice of continuous integration is based on this observation. Updating your code from the code base should be done daily if not more often. Your changes to the code base should also be committed daily if not more often. This eliminates the painful and error-prone integration phase from development.

There are times when those merges, no matter how small, will result in incorrectly functioning code. A comprehensive automated test harness can catch these errors. The agile practice of comprehensive unit testing provides this safety net.

Merging using a text editor can be one of the more confusing things to be done, particularly with more than one or two conflicts. Along with reporting status information, this is an area where GUI tools and slick interfaces come into their own.

Subverting Eclipse

Eclipse talks to revision control systems. In Eclipse terminology, a project under revision control is a *shared project*. Revision control plug-ins are referred to as *team providers*. The team provider we'll be using is called Subversive. The Subversive web site is located at `www.polarion.org/index.php?page=overview&project=subversive`, and the update site is located at `www.polarion.org/projects/subversive/download/1.1/update-site/`. There are several optional components in this package that depend upon other plug-ins that you may not have installed. By default, they are selected. In order to install Subversive, you must either install these plug-ins or deselect the optional components.

■ **Note** Subversive is likely to become the standard Subversion team provider for Eclipse, and by the time you read this, it may ship with Eclipse.

Sharing Your Subverted Project

There are several ways of getting your project into Eclipse:

- Importing directly from Subversion.
- Importing a project that has already been checked out via the command line.
- Sharing a project that has already been checked out.
- Exporting your project directly to Subversion.
- Adding sharing to a project that has not been yet been checked out. (Sadly, this is broken for `file:///` URLs in Subversive.)

You're going to import your project directly from Subversion. This is the most frequent way that you'll operate. It ensures that you have a clean environment, and it's easy to do. In Chapter 2, you set up Eclipse with your test project, named agile. You imported that project into Subversion, checked it out in another location, and made a number of changes. Those changes are in Subversion, but they're not in the workspace for agile. When you import the project, you're going to choose to overwrite that project.

You can import your project directly from Subversion because the `.project` file is checked in. The `.project` file contains the name of the project. This makes it a little trickier to import multiple versions of the same project, but it goes a long way toward ensuring that every developer has a consistently named environment. There is often a deep desire to customize project names, but consistent naming becomes important in projects where many developers work together. This is particularly true with pair programming. In such situations, developers will end up working on someone else's machine at least half of the time. Having to figure out the local namings adds unnecessary hassle and often subtly frustrates one of the pair.

Importing from Subversion

Once you've installed Subversive and restarted Eclipse, you can import from the repository. Select File ➤ Import, which will bring up the Import project window, shown in Figure 3-3.

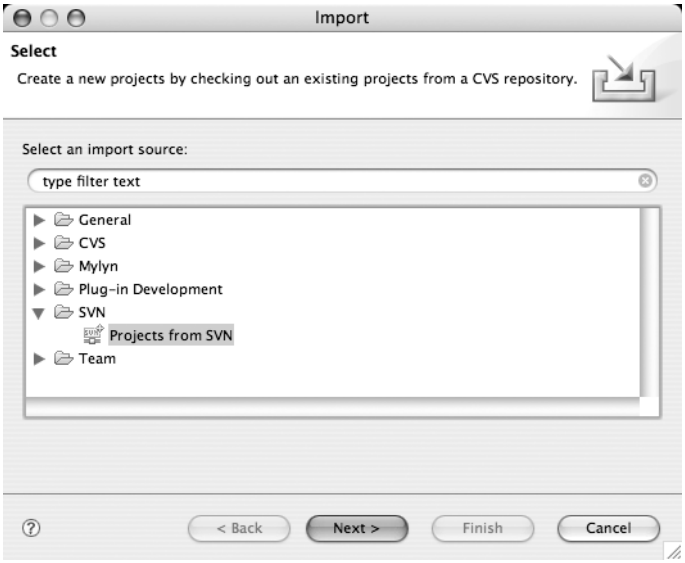

Figure 3-3. *Importing an existing project*

Select SVN ➤ Projects from SVN, and then click the Next button. This will take you to the repository selection screen, shown in Figure 3-4.

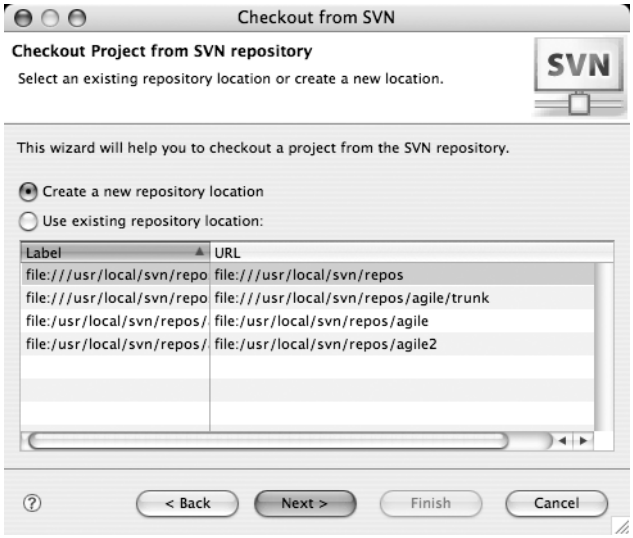

Figure 3-4. *Checkout from SVN*

If a repository had already existed, then you could select it from the list. Since you've never accessed this repository location before, you'll have to create a new one. Choose the "Create a new repository location" radio button, and click Next. This takes you to the screen shown in Figure 3-5, where you'll define the repository location.

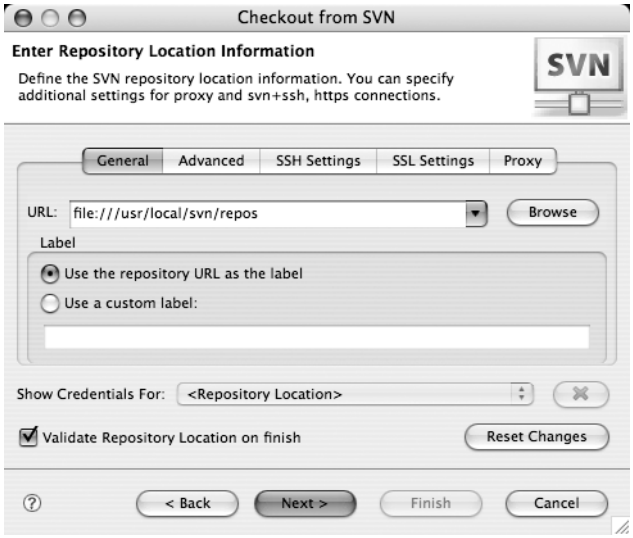

Figure 3-5. *Defining the repository location*

You should fill in the URL with the path to the local repository, which is `file:///usr/local/svn/repos`. File repositories require almost no additional information, so no extra work needs to be done. If you were using SSH, HTTP, or HTTPS as transports, then you could configure authentication information at this point. For HTTP and HTTPS, a proxy server can also be defined. The one relevant tab is labeled Advanced. It allows you to configure settings for repository structure determination.

Once you've filled in the URL as pictured, click Next, which will take you to a repository browser, as shown in Figure 3-6.

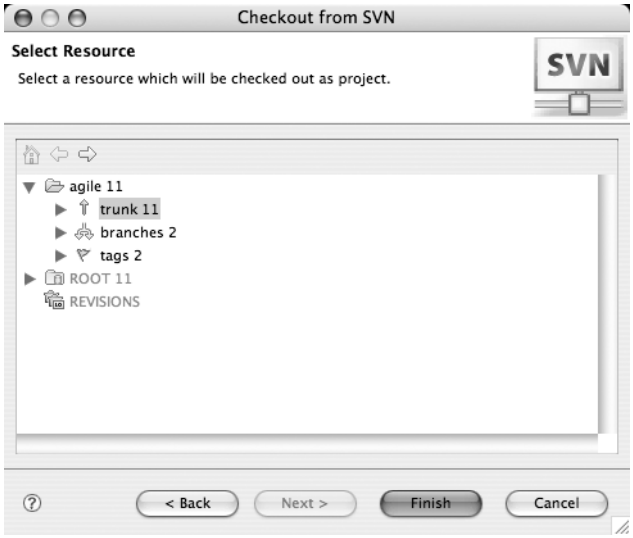

Figure 3-6. *Selecting the agile project's trunk*

Note the glyphs beside "trunk," "branches," and "tags." Subversive understands the common conventions used with Subversion. If you're importing and exporting projects, Subversion often makes the correct guess about project major vs. project minor organization. The revision numbers are beside each node.

You're going to import the trunk. Select "agile" ➤ "trunk," and then click Finish. Eclipse now gives you an opportunity to decide how you're going to import your project. The window is shown in Figure 3-7.

There are two options available. One is checking out as a project into an existing folder. This might make sense in a project where multiple repositories are being used. You might use this at a company where documentation and source code are kept in different parts of the repository or in different repositories. (I'm not a fan of separating code from documentation, but I've seen it done on more occasions than not.)

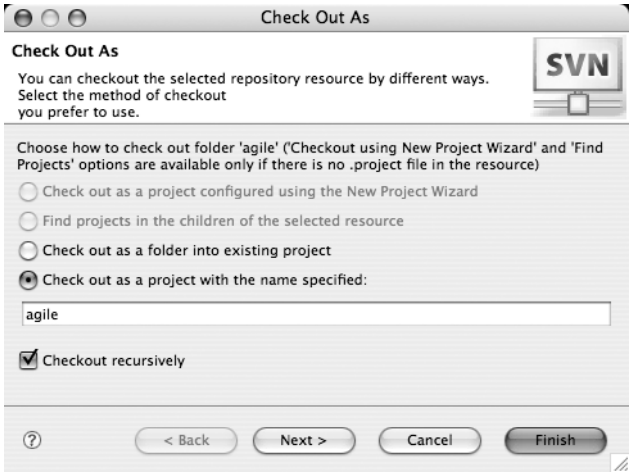

Figure 3-7. *Checking out the project*

The other option is checking out the project as a new project with the specified name. At this point, you could rename the project, but you won't be doing that. You'll choose the default name "agile" and clobber the existing project. As noted earlier, this name is extracted from the `.project` file in Subversion. Click `Finish`. Subversive detects the impending clobbering, and gives you an opportunity to back out. The open window is shown in Figure 3-8.

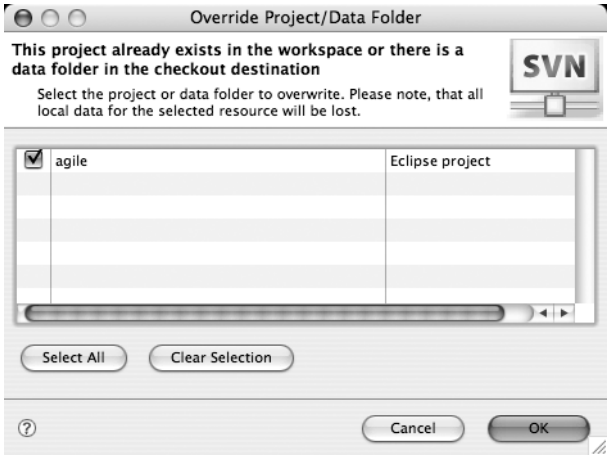

Figure 3-8. *Verifying that you want to overwrite the agile project*

At this point, you definitely want to overwrite the existing agile project. Check the "agile" check box and complete the importation by clicking OK. For a few seconds, you'll see a progress bar as the projects are reshuffled and the new project is imported. You'll then return to the workbench, as shown in Figure 3-9.

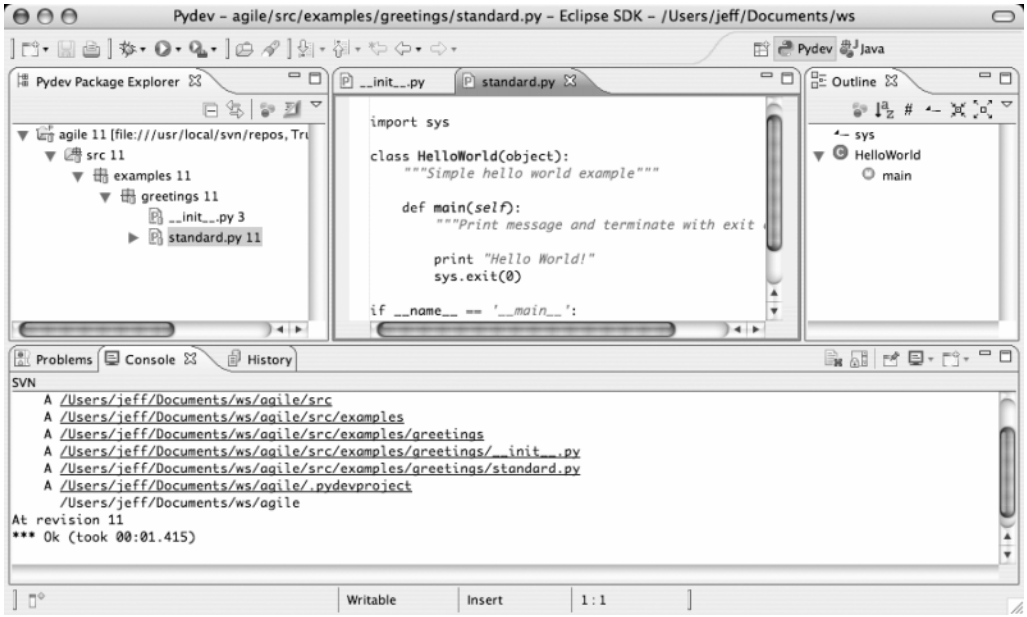

Figure 3-9. *The workbench with the agile project imported*

Activity has happened in the Console view and in the Pydev Package Explorer view. The Console view shows the output from the Subversion checkout. Subversive gives a verbose accounting of its actions. It shows both the command-line equivalent of the operation it performed and the output from the operation. You could replicate its operations on the command line if you wanted to.

The Pydev Package Explore looks different than before. Beside the project name is the URL of its repository. Beside each node is the revision number, and each icon has a small yellow glyph that indicates that the node is shared from Subversion. Other team providers use other glyphs. Additional glyphs show up to indicate other status changes. Those will be covered in the next section of this chapter.

Working with a Subverted Eclipse

Many operations in Eclipse are directly tied to Subversion. Deleting a module or package under Subversion control will delete the file from Subversion. Copying or renaming a module or package will cause the corresponding copy or move.

Surprisingly, some operations that you'd expect to be tied into Subversion are not. Creating a new file, module, or package does not automatically add the file to Subversion. More perplexingly, revert is only half done. Reverting operations that create new files will leave the newly created files in your workspace while restoring the old files. Move and rename both do this as well. These issues might be fixed by the time this you read this, however.

It's useful to be able to see what state your working copy is in with respect to the repository, and Subversive excels at this. This is done through the *team repository view*.

The Team Repository View

The team repository view supplants most of the Subversion status operations we looked at earlier in the chapter. It shows how the repository is different from your working copy, and how your working copy is different from the repository. It can combine those views showing all changes, or it can show only files with conflicts. It can show the aggregate changes, or it can break the differences down by revision.

Open the repository view by choosing the menu item Window ➤ Open View ➤ Other. This will bring up the window shown in Figure 3-10.

Figure 3-10. *Selecting a view*

Select Team ➤ Synchronize from the menu and click OK. This takes you back to the workbench, where you'll see Synchronize in the bottom pane, as in Figure 3-11. It's blocking the Console view. You can toggle back and forth between the two by clicking the tabs, but for the upcoming examples, you'll want to see both views simultaneously. Fortunately, all views in Eclipse can be moved.

Figure 3-11. *The newly opened Synchronize view*

You manipulate views by grabbing their named tabs and dragging them to a desired location. As you do this, a bounding box will show where the view will be repositioned. The other

panes in the workbench will be resized to accommodate the change. If you drag it into a list of other tabs, it will join the set. If you drag a view onto the desktop, it will become a free-floating window.

You're going to split the lower view in two. Grab the Synchronize tab and drag it all the way to the right edge. At some point, the display will show a box that splits the pane in two, and the cursor will turn into an arrow pointing to the right. Let go of the tab at that point, and you should have two panes, as shown in Figure 3-12.

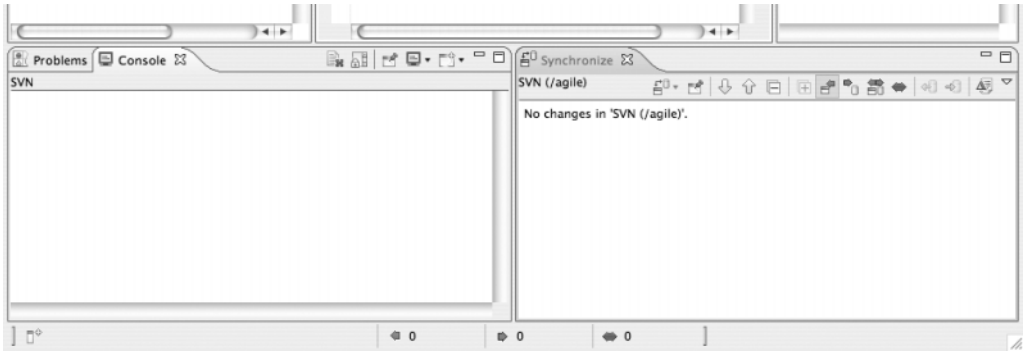

Figure 3-12. *Now the views are side by side.*

You can further adjust the proportions by grabbing the divider between the two views and dragging it left or right. You can adjust the height of both sets of views by grabbing the upper dividing bar and moving it up or down.

Notice the three colored arrows at the bottom of the screen. These only show up when the Synchronize view is active. They relate to the number of changes that have been made since the last update. The number beside the blue arrow indicates changes that have been made in the repository. The number beside the gray arrow indicates changes in your working copy. The number beside the red arrow indicates the number of places in which conflicting changes exist in both the repository and your local working copy.

The view's main area contains a tree browser showing the outstanding changes, as shown in Figure 3-13.

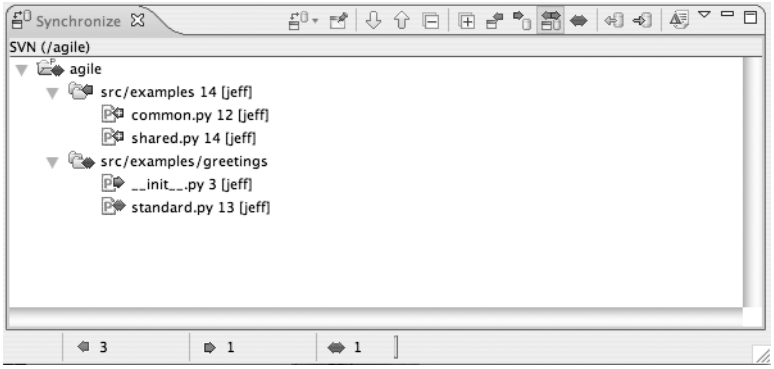

Figure 3-13. *The Synchronize view with outstanding changes*

Double-clicking a file node will bring up a diff viewer among the editor windows. The viewer shows the differences between your local copy and the repository copy. Along the top side of the bar there are a number of icons. I'll run through them from left to right:

Workbench-between-repository: This icon is entitled Synchronize SVN (/agile). When clicked, it updates the view with the most recent information from both the local working copy and the repository. The first time you do this, it also asks if you want to open the Team Synchronizing perspective. You'll be using the Synchronize view within the Pydev view, so choose No. Make sure to choose "Remember my decision" so that you won't be asked this every time you want to see what has changed. You can always open the Team Synchronizing perspective manually.

Pushpin: When active, this icon pins the window in place. Pinning is a generic Eclipse feature. Normally, a view with new information spontaneously pops to the front. Pinned views stay on top even if other views have new information.

Down arrow: This icon advances to the next displayed change. When it advances, it opens up a diff view among the editors. This view shows two versions of a file side by side. The differences between the two files are highlighted with bounding boxes. We'll look at the diff view when we get to merging later in this chapter.

Up arrow: This icon advances to the previous displayed change. Other than that, it works the same as the down arrow.

Boxed minus: This collapses all of the tree nodes in the main area of the view.

Boxed plus: This expands all of the tree nodes in the main area of the view.

Left arrow pointing to a workbench glyph: This limits the main area to new changes in the repository. When selected, the main area shows the changes that have been committed to the repository but have not been updated into the local copy. These are referred to as incoming changes.

Right arrow pointing to a repository glyph: This limits the main area to changes on the local copy. When selected, the main area shows only those changes that have been made locally but have not been committed to the repository.

Left and right arrows over workbench and repository glyphs: This icon shows all the changes that must be made. This includes both incoming changes from the repository and uncommitted local changes.

Double-ended red arrow: This indicates that you want to limit the view to files with conflicts.

Green arrow pointing away from a repository glyph: This pulls down all incoming changes.

Red arrow pointing into a repository glyph: This triggers a commit of all outstanding changes in the local copy.

Delta over a list: This alters the presentation of the view. Normally, all incoming changes are bundled together into one list. When this icon is active, the view is organized by revision number.

Upside-down white triangle: This is a standard Eclipse icon. It indicates that this view has a menu. You select the menu by clicking the icon. The menu contains a number of selections, but the two most interesting are Presentation and Schedule. Presentation allows you to select the format in which changes are presented (the default is compressed tree format). Schedule allows you to select how frequently Subversive will update the view. The default is to never update automatically. I use the Schedule option to update several times an hour for local projects and once a day for remote public projects.

Adding a File

You've already learned how to create a new Python module in Chapter 2. Create one now called `examples.common`. It doesn't matter what's in the module at the moment. Your Pydev Package Explorer should look something like Figure 3-14.

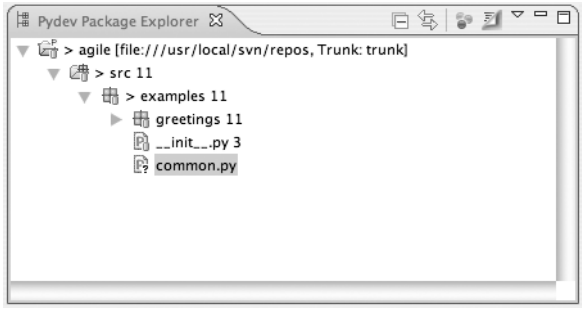

Figure 3-14. *examples.common.py created and added, but not committed*

The question mark glyph indicates that Subversion doesn't know about this file yet. You should take a look at the Synchronize view, though. You'll see that it shows up as an uncommitted change. Which of these is right? The answer is that both are right. Is there a bug? Possibly.

Subversive recognizes that the new file exists, and it assumes that you want to commit it. The Synchronize view reflects what Subversive thinks will happen. The Pydev Package Explorer reflects what Subversion reports.

If you commit from Eclipse, then the new file will be included. If you submit from the command line, then the file will not be included in the submission. My personal feeling is that Subversive should perform the add for you when it sees the new file, so that `svn commit` will check in the same files as Subversive does.

If you're just working within Eclipse, then Subversive's behavior works well. If you switch between Eclipse and the command line, then Subversive's behavior can lead to problems. Command-line tools won't know about the new files you intend to add. In that case, you should add the files explicitly.

Bring up the context menu by right-clicking `common.py` in the Package Explorer. Select Team ➤ Add to Version Control. This will bring up a window presenting a list of files to add, as shown in Figure 3-15.

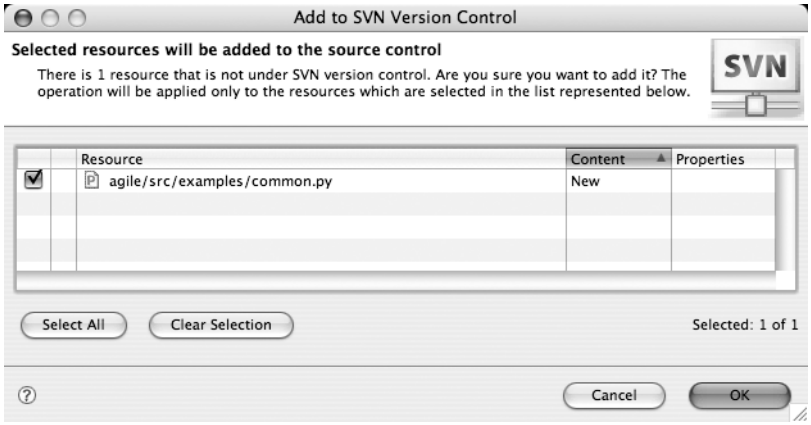

Figure 3-15. *Adding a file to Subversion*

Click the OK button. Eclipse will add all the checked files from this window; in this case, just `common.py`. The Console view should show something akin to the following messages:

```
*** Add to Version Control
svn add "/Users/jeff/ws/agile/src/examples/common.py"
A         /Users/jeff/ws/agile/src/examples/common.py
*** Ok (took 00:00.121)
```

The Pydev Package Explorer should have been altered, and should look something like Figure 3-16. There is a clock glyph on `common.py`. There is also a > preceding the name. The clock glyph indicates that the file has been scheduled for commission, and the > indicates that the node contains a change. The clock applies to just this file, but the > ripples up through the directory tree. If a directory contains a file or a directory with a change, then it is marked, too.

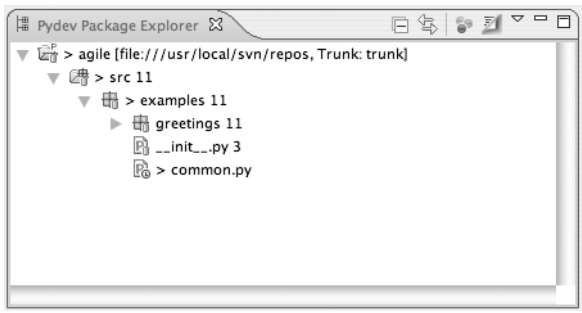

Figure 3-16. *Subversion now knows about common.py.*

Committing Changes

There are two primary ways of committing changes to Subversion. The first is by selecting individual files and using the context menu. The second is through the Synchronize view. Both paths will take you to the Commit window (shown in Figure 3-17).

Individual file selection can be done in any place that shows files. Typically, this will be through the Pydev Package Explorer or the Synchronize view. You'll use the Package Explorer for this example. Select `common.py` from the Package Explorer, right-click to bring up the context menu, and select Team ➤ Commit. This will bring up the Commit window.

The selected files will be shown in the lower portion of the window. When checked, they will be included in the commit. Instead of choosing individual files, you can also select a package or folder from the view. All new files contained within that package or folder will be selected for addition. All files contained in any subpackages or subfolders will be similarly selected.

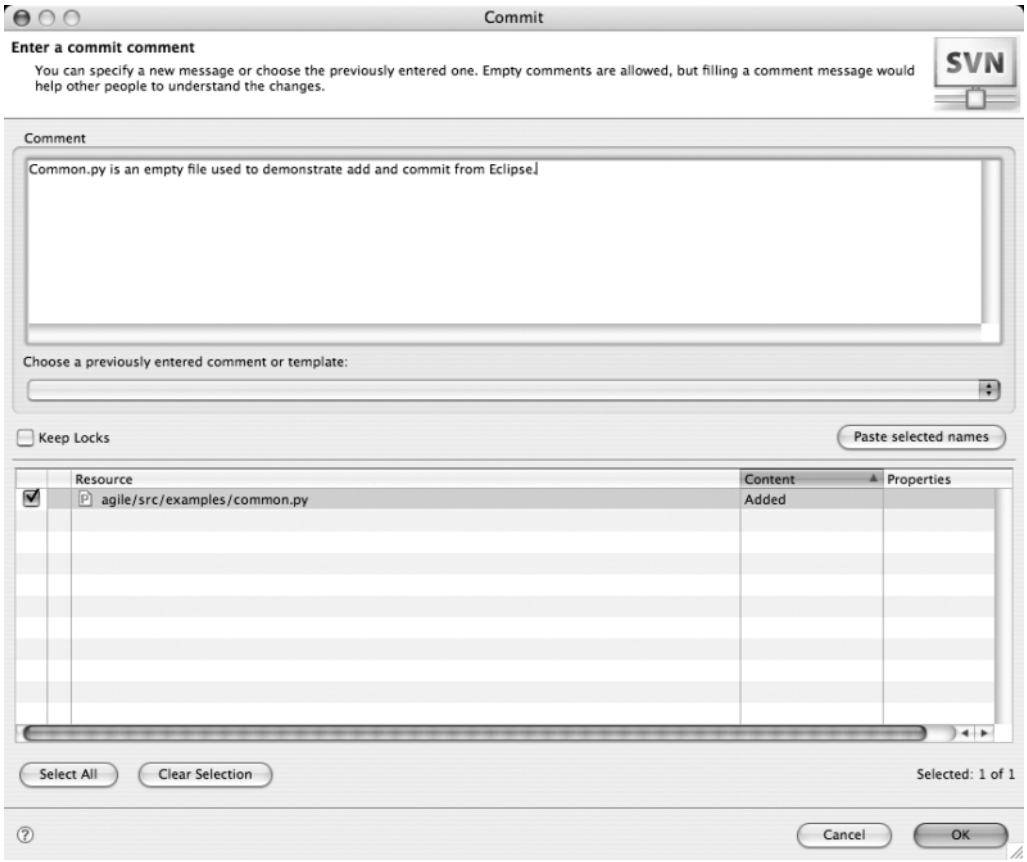

Figure 3-17. *The Commit window*

The upper portion of the window contains the commit message. You can't commit without one. There are a number of options to help with generating the message. The menu bar below allows you to choose from previously entered comments or from predefined templates. You will find the previously defined messages useful when resubmitting failed commits. Templates are useful when setting up default messages for common tasks.

Below that is the "Paste selected names" button. Clicking it will copy in the names of any files selected in the lower window. It copies those names into the comment field, one name per line. It's a real time and effort saver when you're putting together a commit message, and it helps to prevent misspellings from retyping.

When you're done composing the commit message, click OK. The console should spew out something close to the following message:

```
*** Commit
svn commit "/Users/jeff/ws/agile/src/examples/common.py" -N ➥
-m "The empty common.py file is being used to demonstrate ➥
adding and reverting."
A         ws/agile/src/examples/common.py
Transmitting file data: ws/agile/src/examples/common.py
Committed revision 12
*** Ok (took 00:01.054)
```

You can also submit all pending changes through the Synchronize view. On the left-hand side of the menu bar is a red arrow pointing at a repository glyph. Clicking this icon will bring up the Commit window (shown in Figure 3-17), but this time all scheduled changes will be included in the file list.

Editing a File

Editing is the easiest operation. Just open an editor and go to town. The complications come when it is time to submit. Subversive and the Synchronize view make it easy to anticipate conflicts, though.

You can see this by making changes to the previously added `common.py` file. Adding a doc string like `"""A sample edit"""` will suffice. Notice that when you type, there are no changes in the synchronization window. This is because you haven't saved the changes to the filesystem yet. That's indicated by a little * in the editor tab and just to the left of the file name. Once you save the file, the marker will go away, and the screen should look something like Figure 3-18.

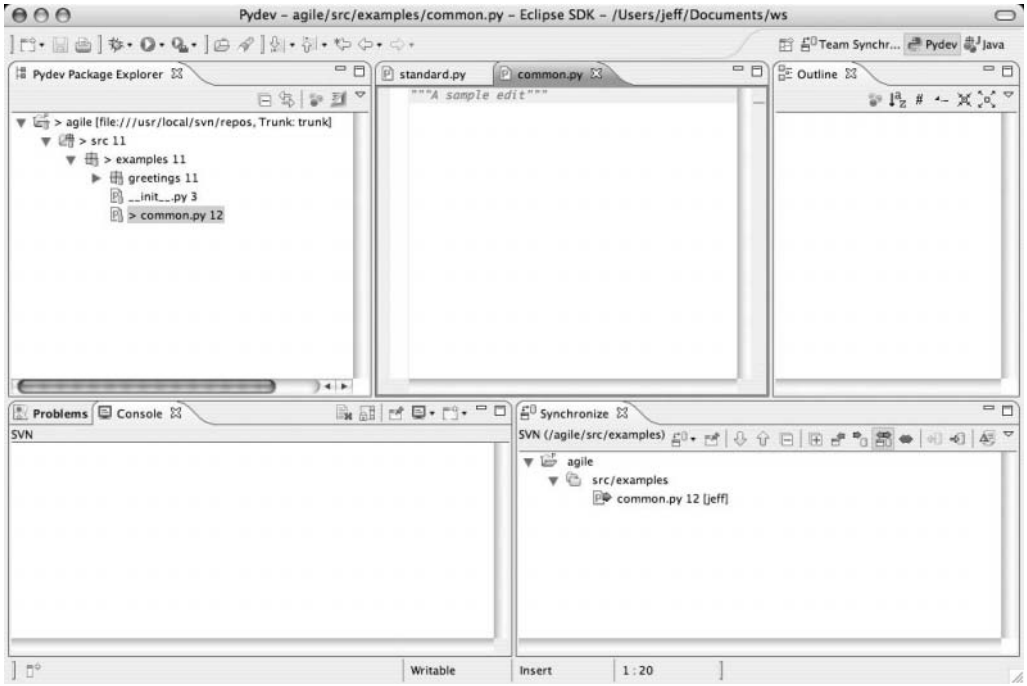

Figure 3-18. *A simple sample edit*

Within instants after you save the change, the Synchronize view will update with the modifications. Subversive also updates the Pydev Package Explorer with change markers cascading from `common.py` up through all of the containers.

Reverting Changes

You have already seen how files can be committed using the context menu. You've seen how this works from different windows and how an entire tree can be selected by choosing the parent container. The same user interface mechanisms hold true for reverting files.

You're going to revert the edits made previously. Select `common.py` from either the Pydev Package Explorer or the Synchronize view, or just right-click in the editor for `common.py`. Bring up the context menu by right-clicking, and select Team ➤ Revert. This will bring up the Revert window, shown in Figure 3-19.

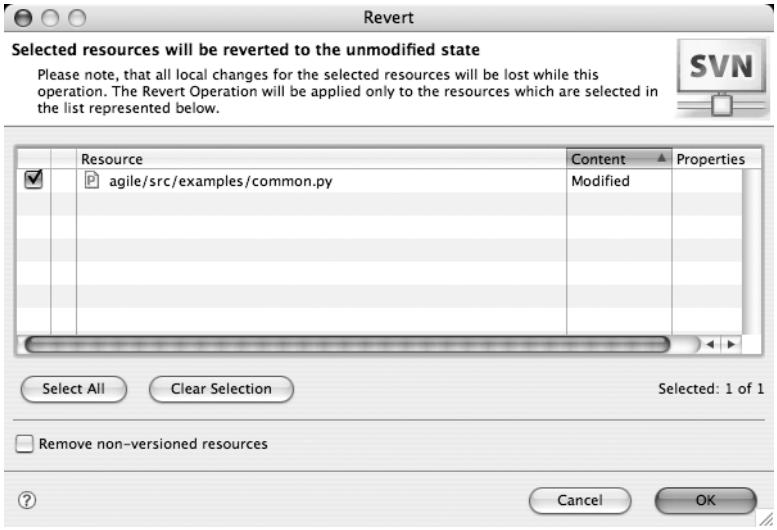

Figure 3-19. *The Revert window*

At this point, you can deselect any files that you erroneously chose. This might happen if you selected a directory containing many files. In this case, only the file you chose should be selected. Click OK, and you should see a message similar to the following:

```
*** Revert
svn revert "/Users/jeff//ws/agile/src/examples/common.py" -R ➥
Reverted /Users/jeff/ws/agile/src/examples/common.py
*** Ok (took 00:01.003)
```

Your change should be gone, and the change markers should vanish from the Pydev Package Explorer. The `common.py` editor should be blank, and the Synchronize view should report the following: `No changes in 'SVN (/agile/src/examples)'`.

Resolving Conflicts

Suppose that someone else has made a change while you were editing `standard.py`. Once again, you've both changed the comment line for `standard.HelloWorld.main()`. This time, the other user has committed their change before you have. When you attempt to submit, you see the window shown in Figure 3-20.

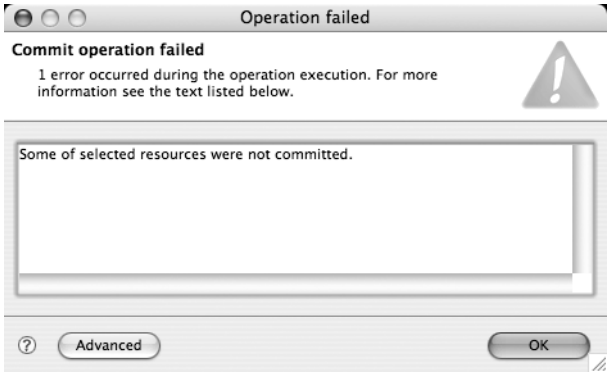

Figure 3-20. *The commit operation failed.*

Clicking the Advanced button will show the details of the failure. The message that follows shows up more or less identically in both the failure details and the Console view:

```
*** Commit
svn commit "/Users/jeff/ws/agile/src/examples/greetings/standard.py" ➥
 -m "Updating doc string."
M       /Users/jeff/ws/agile/src/examples/greetings/standard.py
Transaction is out of date
svn: Commit failed (details follow):
svn: Out of date: '/agile/trunk/src/examples/greetings/standard.py' ➥
in transaction '13-1'

*** Error (took 00:01.122)
```

Here, you can see that you're in conflict with transaction 13. You can get more information from the Synchronize view, but you need to refresh it. You can do this by clicking the leftmost icon in the view's toolbar, after which the view will update and show one conflict (see Figure 3-21).

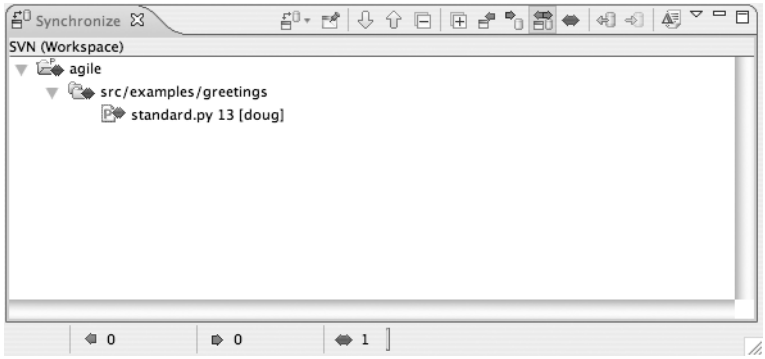

Figure 3-21. *One conflict shown in the Synchronize view*

The conflicting file is revision 13, and you can see that it was committed by doug. Double-clicking the file name will bring up the Text Compare editor, as shown in Figure 3-22. You can see from Figure 3-22 that there is a difference in whitespacing on one line.

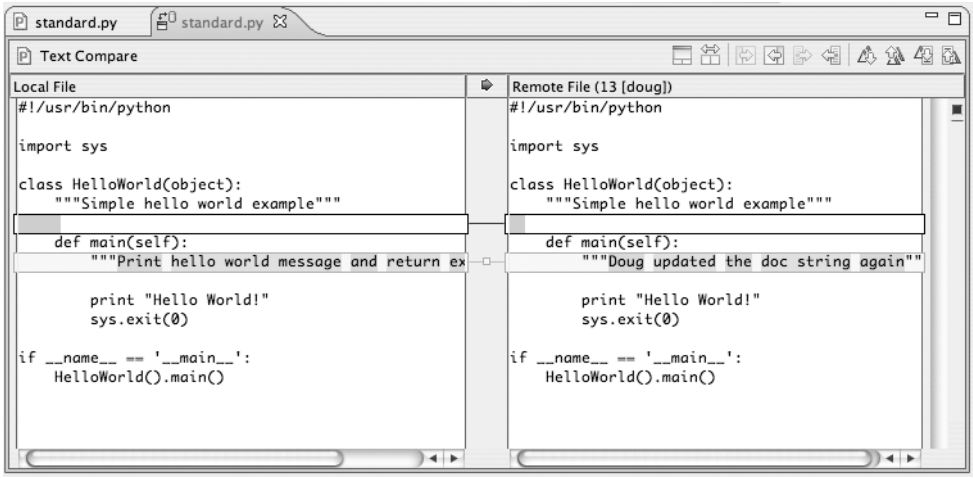

Figure 3-22. *Showing conflicts between files*

You can update your code in one of two ways. You can choose a directory tree from an explorer and then select Team ➤ Update from the context menu to bring over a subset of changes, or you can use the update button in the Synchronize view to bring over all the changes. The update button is the third icon from the right. The icon is a little green arrow pointing away from a repository glyph. After you click the button, you'll be asked to confirm that you really want to bring over these changes. You do, so you can agree. A lot happens at this point. First, the update log messages appear in the console:

```
*** Update
svn update "/Users/jeff/ws/agile/src/examples/greetings/standard.py" -r HEAD
C         /Users/jeff/ws/agile/src/examples/greetings/standard.py
At revision 13
*** Warning (took 00:00.584)
```

Subversion creates the four versions of the changed file, as in the command-line example. These show up in the Pydev Package Explorer view. In that same view, the repository glyph next to `standard.py` turns red to indicate that the file is in conflict. The Text Compare editor loads the candidate merge of `standard.py`, and you can see the conflict markers. The conflict markers prevent the file from parsing as legal Python, so you'll see a chain of red error marker glyphs on the lower left-hand corners of each node in the Synchronize view. You can see all of this in Figure 3-23 (if you look closely).

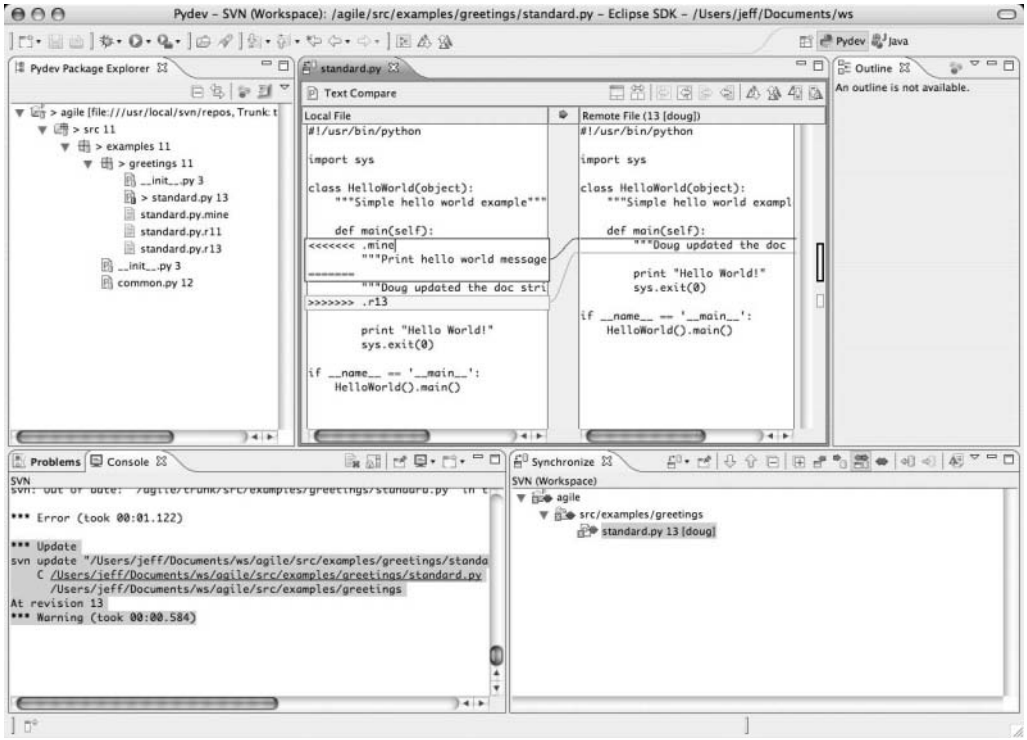

Figure 3-23. *The workbench with conflicting files*

Subversive understands that the three additional files are related to the conflict. It does not treat them as outbound changes, and they do not show up in the Synchronize view.

Open up an editor for `standard.py` and make the desired changes. After the changes are made, it's time to mark the file as resolved. This can be done from the context menu in an explorer view or from the context menu in the Synchronize view. From an explorer, the menu option is Team ➤ Mark as Merged, and from the Synchronize view, the menu option is Mark as Merged—either choice works. After one is selected, Eclipse will grind away for a second or two. You'll see a progress bar, and afterward the conflicts will disappear from the Synchronize view to be replaced by a normal pending update marker.

At this point, you can safely commit the changes. Your previous commit comment will be accessible from the drop-down menu on the Commit window, so there is no need to retype it, although you will have an opportunity to edit it.

Deleting Files

Files can be selected for deletion using pretty much any tree browser or through the file's editor window. Selecting a directory will delete all of its contents, too. File selection is exactly the same as with adding, reverting, or committing. It's only a little different when you choose to delete the selection.

Deleting files can be done in three ways. One is to bring up the context menu and select Delete. Another is to select Edit ➤ Delete from the main menu. Finally, you can press the Delete key.

Once you confirm your deletion, the files will be removed. You should see a log message in the console similar to this:

```
*** Delete
svn delete "/Users/jeff/ws/agile/src/examples/common.py" ➥
--force
D         /Users/jeff/ws/agile/src/examples/common.py
*** Ok (took 00:00.099)
```

At this point, the selection is scheduled for deletion, and it should show up in the Synchronize view. The icon beside the file name indicates the scheduling. It is a little outbound arrow glyph containing a minus sign indicating that the file will be removed.

Moving Files

Files and directories can be moved from one directory to another. This is done by selecting the candidate files from an explorer and either choosing Move from the context menu or Edit ➤ Move from the main menu. This will bring up a file browser to select the destination directory.

When Eclipse moves the files, Subversive will schedule a series of adds and deletes to perform the move. The message for moving a single file should look similar to the following:

```
*** Move
A         /Users/jeff/ws/agile/src/examples/greetings/common.py
D         /Users/jeff/ws/agile/src/examples/common.py
*** Ok (took 00:01.093)
```

The add operation maintains history between the original files and the new files. Each add and delete will show up in the Synchronize view.

Renaming Files

Only one file at a time can be renamed. You can select a file from an explorer view, and then you can choose either Rename from the context menu or Edit ➤ Rename from the main menu. This brings up a window that allows you to select a new name. Subversive treats the rename exactly as it treats a move; a similar message shows up in the console window:

```
** Move
A         /Users/jeff/ws/agile/src/examples/uncommon.py
D         /Users/jeff/ws/agile/src/examples/common.py
*** Ok (took 00:00.152)
```

The difference between copying and renaming is just the interface to the command.

Copying Files

Normally in Eclipse, you use the copy and paste operations to copy files. This is a no-no when using Subversion and many other source control systems. Subversion needs to maintain the history of a copied file. Because copy and paste are separate operations, that information is lost. Subversive gets around this by providing a special copy operation in the Team menu.

Copying files from one directory to another is much like moving files. The files to be copied are selected from an explorer, and then the copy operation is selected. From the context menu, you select Team ➤ Copy To, which brings up the screen shown in Figure 3-24.

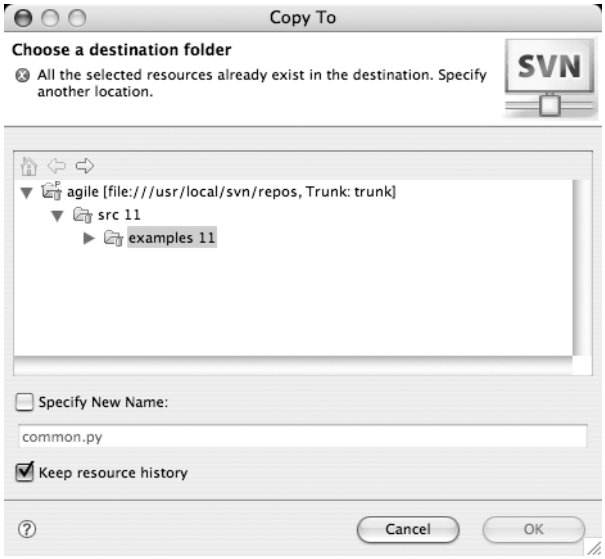

Figure 3-24. *Copying a file*

The destination is selected from the tree browser as when moving files, but there are several other options. As with move, you select a destination directory. Subversive normally uses the Subversion `copy` command, which tracks history. You can see this in the console:

```
*** Copy
svn copy "/Users/jeff/ws/agile/src/examples/common.py" "/Users/jeff/ws/agile/➥
src/common.py"
A         /Users/jeff/ws/agile/src/common.py
*** Ok (took 00:01.330)
```

You can specify a new name at this point. If you are copying a single file and you rename it, then the destination file will be placed into the destination directory with the new name. If you are copying more than one file and you specify a new name, then something else happens. A directory with the new name is created in the destination directory. The files are then copied into this new subdirectory. You can see this in the console as `src/examples/__init__.py` and `src/examples/common.py` are copied into `src` and renamed to `stuff`:

```
*** Copy
svn add "/Users/jeff/ws/agile/src/stuff" -N ➥
A         /Users/jeff/ws/agile/src/stuff
svn copy "/Users/jeff/ws/agile/src/examples/__init__.py" ➥
"/Users/jeff/ws/agile/src/stuff/__init__.py"
A         /Users/jeff/ws/agile/src/stuff/__init__.py
svn copy "/Users/jeff/ws/agile/src/examples/common.py" ➥
"/Users/jeff/ws/agile/src/stuff/common.py"
A         /Users/jeff/ws/agile/src/stuff/common.py
*** Ok (took 00:01.878)
```

You can look at this as an obscure way of copying files into a new directory.

You can also turn off resource history. Normally, Subversive tracks the parentage between the original and its copy. Most of the time you want to keep this information, but there are times when you don't; for example, you might be using a piece of example or demo code as a starting point for real work. You don't really care that you started with the example code. In these cases, you can turn off the resource history.

If you copy a single file without revision history, then the file is added to the filesystem in the new destination. You will see no Subversion messages in the console. If you rename a single file without revision history, then it will be added to the filesystem with the new name. You will see no Subversion messages in the console. If you try to rename multiple files without revision history, then something bad happens. You'll see an attempted copy message and an error message in the console. As of Subversive 1.1.7, copying multiple files while renaming without revision history is broken. Hopefully, this case will be fixed by the time you read this.

Reverting Moves, Renames, and Copies

Reverting a move, rename, or a copy doesn't work the way you might expect it to. Reverting a moved, renamed, or copied file will only undo the Subversion operations creating that file. It won't remove the file from the local filesystem. You'll have to delete the new files manually, or else they will be added during the next commit. Reverting the new file also won't undo the operations affecting the original file or files either. For a rename or move, you'll have to revert the deletes manually. Reverting an entire directory or using the Synchronize view makes things easier.

Summary

There is no reason not to use a revision control system. There is no reason to lose code. Revision control systems are common and many are free. They provide a shared repository that allows you to look at your code at any point in time. They should serve as the shared repository for all development sources on any project. All developers submit their changes to the repository, and they receive one another's changes through the repository.

Although there are many revision control systems out there, I've focused on Subversion. Subversion is a free revision control system based around the edit-and-merge paradigm. It supports network access, a topic that will be examined in more detail in Chapter 5. Commits

are atomic and ordered through a global, monotonically increasing revision number. All work in Subversion is done in a local working copy until the changes are committed.

You saw how to work with Subversion from the command line and through Eclipse. This included setting up a repository, obtaining an initial set of files, and performing day-to-day operations such as adding, editing, deleting, copying, and moving files. In addition, you learned how to maintain consistency between the repository and the working copy on your local machine. Differences between the two can be examined, conflicts can be resolved, and undesired changes can be reverted.

Now that you know how to work with Subversion and Eclipse, it is time to start building a real project. There will be many problems to address. The project will need to be built, deployed, and packaged. The packages need to be tracked and versioned, and unit tests must be run over and over again. After all this, the code is deployed into the development environment to see how it interacts with the rest of the Python installation.

This could be done from scratch. Reams of Python code could be written. It might even make a fun project if you're into that sort of thing. Fortunately, though, it has already been done with a package called Setuptools. In the next chapter, I'll be showing you how to use it. It accomplishes all the things described here and more.