Subversion (SVN) Introduction

Objectives:

- Learn the basics of Subversion source code management.
- Learn how to use Eclipse's SVN integration.

Part I (Command-line usage)

In this part of the lab you will be creating and accessing a repository from the command line. For the following examples, the blue text contained within a box are the commands that you should execute. The remaining text is what should be outputted by Subversion or the shell.

Creating the Repository

First, SSH into your CIS account then type the following commands.

```
viper~% svnadmin create ~/svnrepository
viper~% ls -lp ~/svnrepository/
README.txt  conf/  dav/  db/  format  hooks/  locks/
```

The **`svndamin create`** command creates a new directory `~/svnrespository` (the tilde is an alias for your home directory) which contains your repository.

Importing a Project

Now type the following commands to download and extract a tarball containing a sample project for this lab:

```
viper~% wget -q
http://people.cis.ksu.edu/~harmon/cis540f09/cis540-lab4.tar.gz
viper~% tar xzvf cis540-lab4.tar.gz
viper~% ls -l ~/cis540/
```

The sample project contains one file: `HelloWorld.java`. Now we are ready to import this sample project into the repository. In the following, replace *<abpath>* with the absolute path of your home directory. For most this will be '`/home/ugrads/<cis_userid>`'

```
viper~% svn import ~/cis540 file:///<abpath>/svnrepository/cis540
-m "initial import"
Adding         <abpath>/cis540/src
Adding         <abpath>/cis540/src/HelloWorld.java

Committed revision 1.
```

Now the repository contains a top-level directory named `cis540`, which in turn contains the two files that make up our project. Note that '`-m "initial import"`' defines the comment we want to associate with this import. If we didn't include this line then Subversion would start your default text editor (likely vi) and wait for you to exit it before continuing.

Note that the original `~/cis540` directory is unchanged by the import. Subversion is unaware of it. Since the project is in the repository, we could even delete the directory

A typical Subversion repository often holds the files (or source code) for several projects; usually, each project is a subdirectory in the repository's filesystem tree. You would do an **`svn import`** for each project that you want to Subversion to manage.

Creating a Working Copy

In order to start manipulating the repository data, you need to create a new "working copy" of the data, a sort of private workspace. Now ask Subversion to "checkout" two working copies of the svnlab directory in the repository calling them `svnlab_bob` and `svnlab_sue`. You do this using the **`svn checkout`** command.

```
viper~% svn checkout file://<abpath>/svnrepository/cis540 svnlab_bob
A    svnlab_bob/src/HelloWorld.java
Checked out revision 1.
```

```
viper~% svn checkout file://<abpath>/svnrepository/cis540 svnlab_sue
A    svnlab_sue/src/HelloWorld.java
Checked out revision 1.
```

In the above output we can see that subversion created two directories called `svnlab_bob` and `svnlab_sue` and populated each with the current version of the `cis540` project. We will use these two working copies to simulate two different users named Bob and Sue who are working on the same project.

Now switch to the Bob's `svnlab_bob` directory and do a directory listing

```
viper~% cd svnlab_bob/
viper~/svnlab_bob% ls -l
./  ../  .svn/ src/
```

Notice that the working copy has a subdirectory called '`.svn`'. This is Subversion's administrative directory. The files in each administrative directory help Subversion recognize which files contain unpublished changes, and which files are out-of-date with respect to others' work. It also stores the location (URL) of the repository so you no longer have to specify it at the command line.

Manipulating the Working Copy

In the `svnlab_bob` directory type the following :

```
viper~/svnlab_bob% echo "INSTALL file for svnlab" > INSTALL
viper~/svnlab_bob% svn add INSTALL
A         INSTALL
```

The first command created a new file called `INSTALL`. The second command instructs Subversion to add `INSTALL` to Bob's working copy and schedule it for addition to the repository. Note that if we didn't add it to the working copy then Subversion would simply ignore it (i.e. it would not be under version control).

Now, modify Bob's `HelloWorld.java` so that it outputs "`Hello everyone`" instead of "`Hello World!`" Save your changes to the file and do the following:

```
viper~/svnlab_bob% svn status
A       INSTALL
M       src/HelloWorld.java
```

This command compares the metadata contained in the `.svn` directory with the working copy. The '`A`' indicates that `INSTALL` has been scheduled for addition into the repository. The '`M`' indicates that the contents of `HelloWorld.java` have been modified.

Now commit your changes back to the repository using the following command

```
viper~/svnlab_bob% svn commit -m "Added INSTALL and modified
helloworld"
Adding          INSTALL
Sending         src/HelloWorld.java
Transmitting file data ..
Committed revision 2.
```

The **`svn commit`** command sends all of your changes to the repository. When you commit a change, you need to supply a log message, describing your changes. Your log message will be attached to the new revision you create. If your message is short then you can type it at the command line using the '`-m`' switch as in the example above. If your message will contain multiple paragraphs then leave off the '`-m`' switch and Subversion will open your default text editor (likely vi) and wait till you exit before proceeding.

Handling Conflicts

Now switch to the Sue's `svnlab_sue` directory using the following command:

```
viper~/svnlab_bob% cd ../svnlab_sue/
```

Modify the contents of Sue's `HelloWorld.java` so that it outputs "`Hello all`" instead of "`Hello world`".
Try committing these changes as follows:

```
viper~/svnlab_sue% svn commit -m "modified helloworld "
Sending        src/HelloWorld.java
svn: Commit failed (details follow):
svn: Out of date: '/cis540/src/HelloWorld.java' in transaction '2-1'
```

The output is informing us the commit failed (atomically) because our `HelloWorld.java` file is older than the one currently stored in the repository. To learn more, we can run **`svn status`** but this time supply the show updates switch `-u`

```
viper~/svnlab_sue% svn status -u
       *            INSTALL
M      *        1   src/HelloWorld.java
Status against revision:      2
```

The `-u` switch instructed Subversion to contact the repository and add information about things that are out-of-date. In the sample output above, the first column tells the status of a file or directory and/or its contents. The third column indicates what version Sue's working copy files are at. `INSTALL` does not have an entry in the first and second columns since is not in Sue's working copy. The asterisks in the second column tell us that both `INSTALL` and `HelloWorld.java` are out of date.

We now have to fix those problems. To do this, type the following:

```
viper~/svnlab_sue% svn update
A    INSTALL
C    src/HelloWorld.cpp
Updated to revision 2.
```

`svn update` brings changes from the repository into your working copy. In the sample output, the 'A' indicates that `INSTALL` has been added to our working copy. The 'C' indicates Sue's changes to HelloWorld.java overlap with the changes from the server, and now you have to manually choose between them.

For every conflicted file, Subversion places up to three extra unversioned files in your working copy:

`filename.mine`
This is your file as it existed in your working copy before you updated your working copy—that is, without conflict markers. This file has your latest changes in it and nothing else. (If Subversion considers the file to be unmergeable, then the `.mine` file isn't created, since it would be identical to the working file.)

`filename.rOLDREV`
This is the file that was the BASE revision before you updated your working copy. That is, the file that you checked out before you made your latest edits.

`filename.rNEWREV`
This is the file that your Subversion client just received from the server when you updated your working copy.

To see this in action, do a directory listing for Sue.

```
viper~/svnlab_sue% ls
INSTALL  HelloWorld.java       HelloWorld.java.r1
HelloWorld.java.mine  HelloWorld.java.r2
```

`HelloWorld.java.mine` would output "Hello all" (Sue's changes),
`HelloWorld.java.r1` would output "Hello world" (version 1), and
`HelloWorld.java.r2` would output
"Hello everyone" (Bob's changes aka version 2).

At this point we could do one of three things to fix the conflict

1. Copy one of the temporary files on top of your working file.
2. Run **`svn revert HelloWorld.java`** to throw away all of your local changes.
3. Merge the conflicted text "by hand" by examining and editing the conflict markers within the file.

Usually you won't want to just delete the conflict markers and Bob's changes—that user is going to be awfully surprised when they next refresh their working copy and suddenly the program is outputting "`Hello all`" instead of "`Hello everyone`". So this is where you pick up the phone or walk across the office and discuss the conflict with Bob. Once you've agreed on the

changes you will check in, edit your file and remove the conflict markers.

To make it easy on ourselves, let's just accept Bob's changes.

```
viper~/svnlab_sue% cp HelloWorld.java.r2 HelloWorld.java
```

Now that you've resolved the conflict, you need to let Subversion know by running **`svn resolved`**. This removes the three temporary files and Subversion no longer considers the file to be in a state of conflict. Note that you have to run **`svn resolved`** regardless of which of the three options you had chosen to fix the conflict.

```
viper~/svnlab_sue% svn resolved HelloWorld.java
Resolved conflicted state of 'HelloWorld.java'
```

Now that you know how to handle conflicts, it is import to note that they do not occur very often in practice. It is unlikely that you will have two or more people modifying the exact same portion of a file.
After running **`svn update`** you will most likely see a '`U`' or a '`G`' next to the filenames. The `G` stands for merGed, which means that the file had local changes to begin with, but the chrepository didn't overlap with the local changes.

Checking-out Historical Revisions

One useful feature of versioning systems is the ability to "go back in time" and examine old versions of a project. For example, if we include the `-r` switch and a reversion number when performing a checkout then Subversion will create a working copy that contains the project as it existed at the specified revision.
For our example we could checkout revision 1 into a directory called `svnlab_r1` as follows.

```
viper~/svnlab_bob% cd ~
viper~% svn checkout -r 1 file://<abpath>/svnrepository/cis540
svnlab_r1
```

Other Useful Commands

svn diff - Displays the differences between two paths.

In the following example we will ask Subversion to compare reversion 1 of `HelloWorld.java` against Bob's current reversion.

```
viper~% cd ~/svnlab_bob
viper~/svnlab_bob% svn diff -r 1 src/HelloWorld.java
```

The output shows which lines differ between the two revisions. Lines prefixed with a '−' are from revision 1 and line prefixed with a '+' are from Bob's current version. Note that Sue could have used **`svn diff -r2 HelloWorld.java`** before she did **`svn update`** in order to see how her copy differed from the current revision located on the server.

Making changes to the repository

`svn delete` - delete an item from a working copy or the repository. For example, we could use **`svn delete INSTALL`** which would schedule `INSTALL` for deletion.

`svn copy` - Copy a file or directory in a working copy or in the repository. In Part II of this lab we will see how to use this command to 'tag' or 'branch' a project.

`svn move` - This command moves a file or directory in your working copy or in the repository. For example, we could use **`svn move HelloWorld.java hw.java`** which would schedule hw.java for addition and HelloWorld.java for deletion.

At this point indicate to the TA that you are finished with Part I of the lab. Do not proceed to Part II until the TA checks your work.

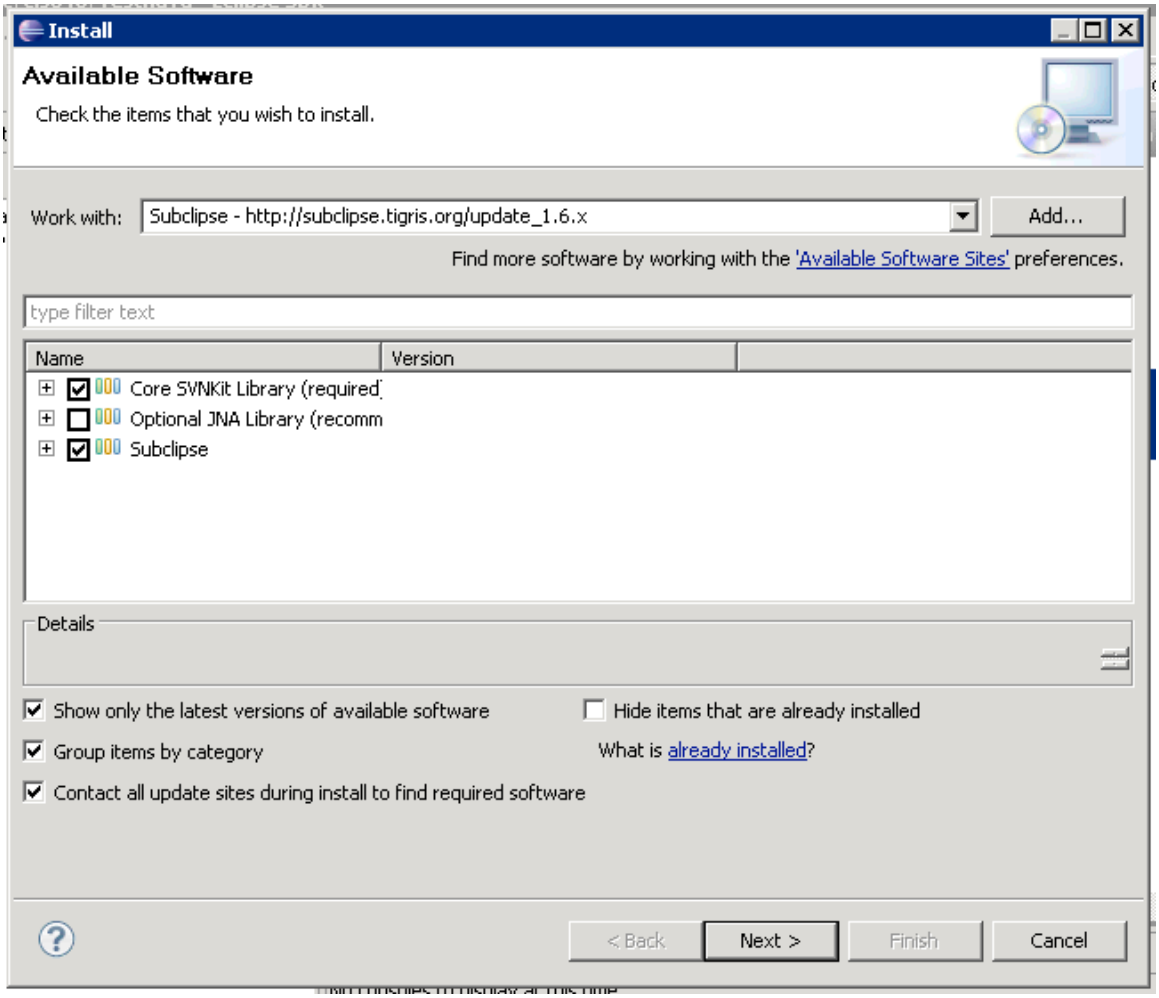
Part II: Eclipse SVN Use

Eclipse does not come with built in SVN support. Thus, for this section, we will first install a popular SVN plugin for eclipse and then go through setting up eclipse to use your projects account.

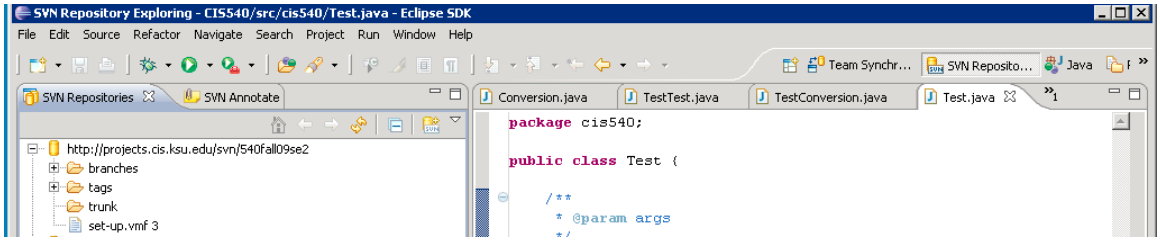
Installing Subclipse Plugin

The update site for subclipse is: http://subclipse.tigris.org/update_1.6.x

Go to *Help->Install New Software*.



Install Core SVNKit Library and Subclipse.

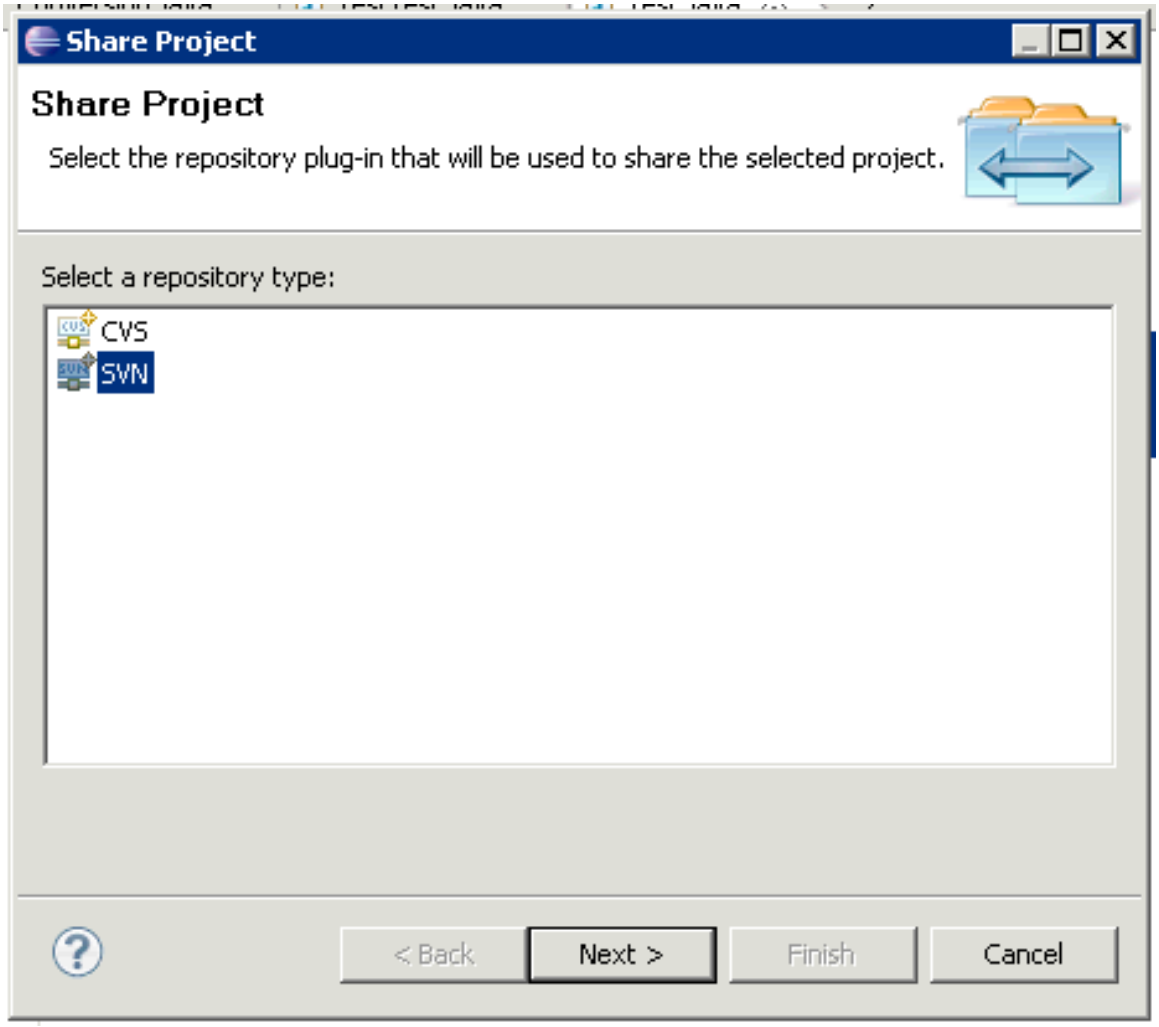


Go to the *SVN Repository Exploring* Perspective. Add new repository. For the URL use: http://projects.cis.ksu.edu/svn/<team>

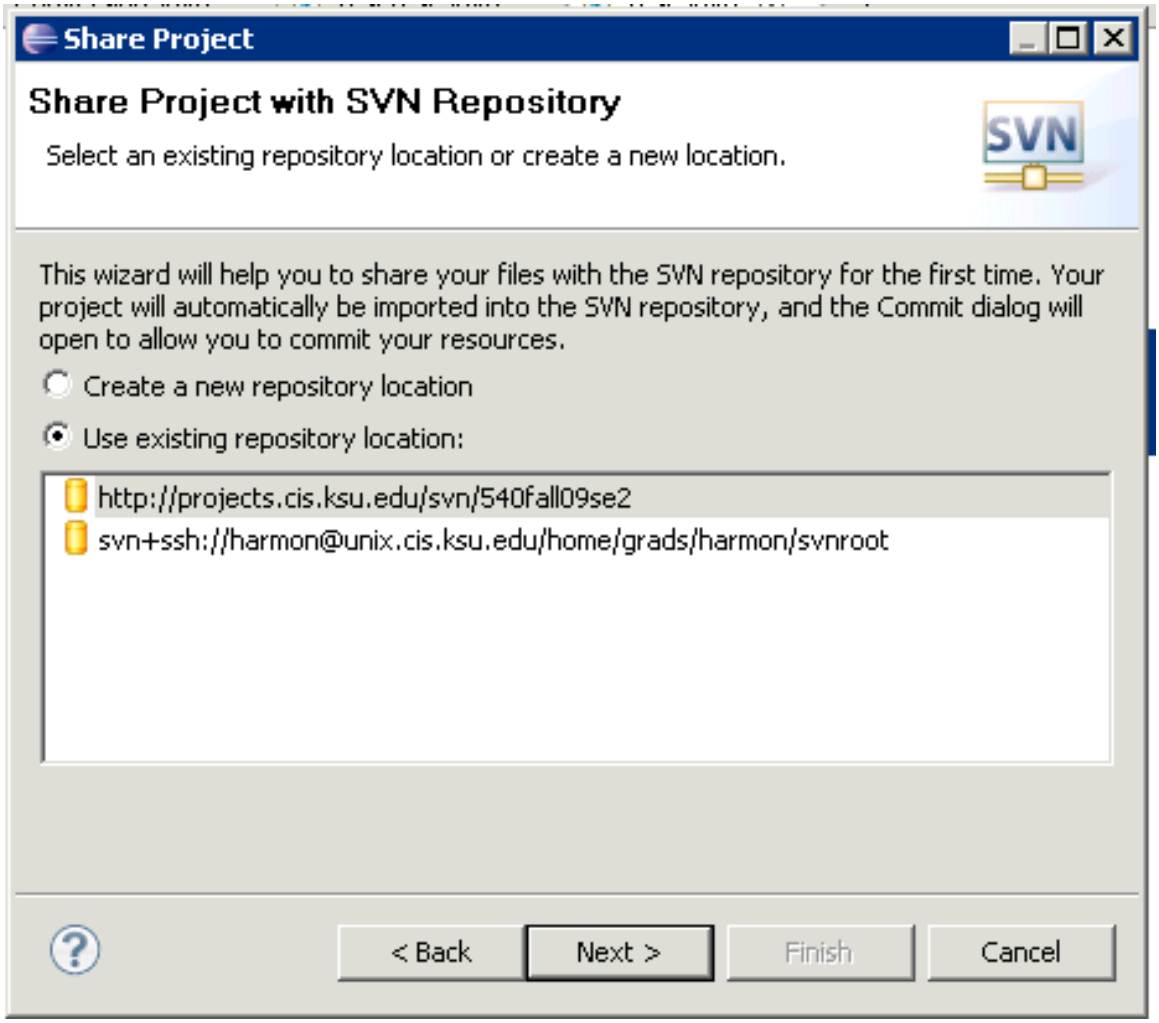
Where <team> is 540fall09se2 for team 2.

Sharing an Eclipse Project

Within Eclipse's *Project Browser*, either create a new, or use an existing project. Right click the project and go to Team->Share Project.

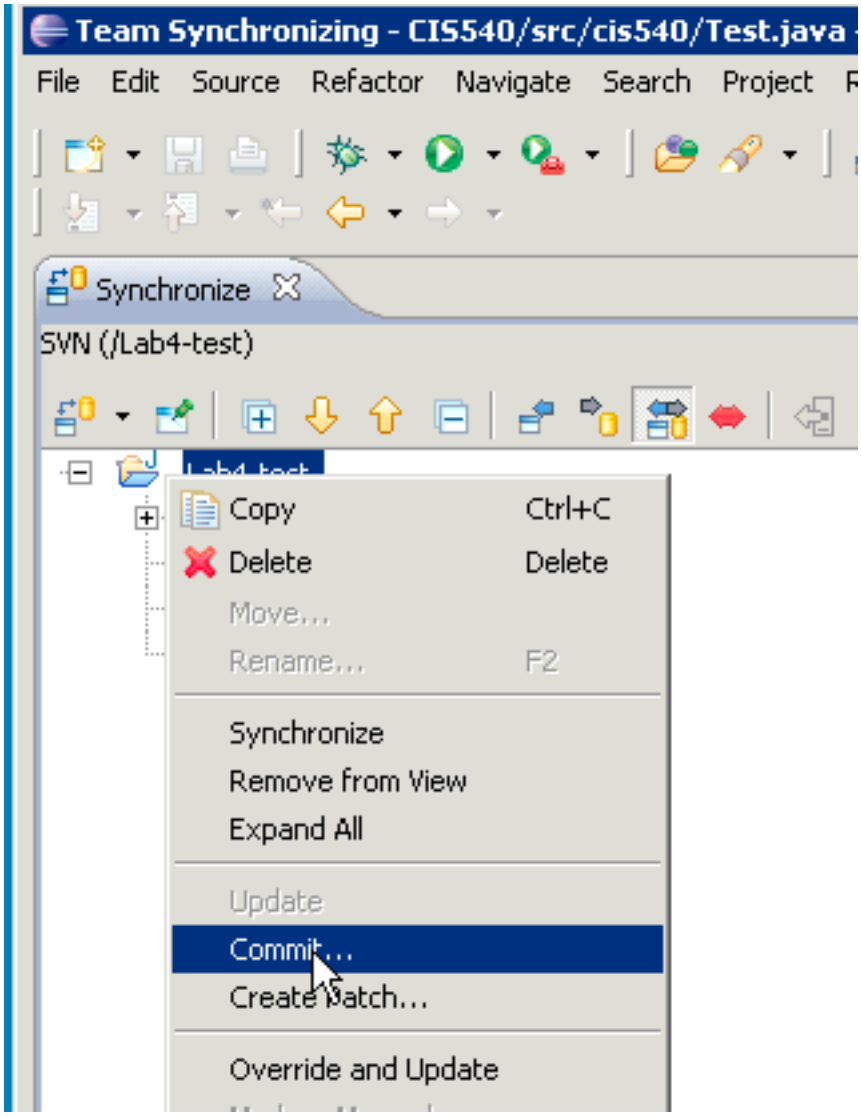

Share Project
Share Project
Select the repository plug-in that will be used to share the selected project.
Select a repository type:
CVS
SVN
< Back
Next >
Finish
Cancel

Choose SVN.



Select the repository location you created in the previous step.

Continue through the wizard.



After opening the Synchronizing Perspective, right click on the project and select Commit.

Now team members may check out the project in their eclipse and make changes.

For more info please see: http://subclipse.tigris.org/