

Introduction to Eclipse



CVS with Eclipse

What is CVS?



- *CVS (Concurrent Versioning System)*
 - is a version control tool
 - has been around since the 80s
 - has become the de facto standard for open source development
- CVS is a client/server system
- Most of the CVS documentation you'll find online talks about the command line client
 - e.g. the cvs command installed on most Unix/Linux systems or the cvsnt port for Windows.
- Eclipse provides a nice CVS GUI which is tightly integrated with the Java IDE and is *much* easier to use than the command line client

CVS Concepts - The Repository

- The CVS server manages a *repository* containing a number of *modules*.
- A module contains directories and *versioned files*.
- A versioned file represents the history of single file and contains a number of *revisions*.
 - Whenever a change is made to the file, a new revision is created
 - For text-based files (.java, .xml, .properties etc) each revision just contains the text that has changed since the previous revision
 - For binary files (.exe, .jar, .pdf, etc), each revision contains the complete file contents
- CVS also records the user who created the revision, their comment for the revision, and a revision number.
 - Revision numbers are automatically updated; in most cases
 - ✓ the first revision of a file is 1.1, the second 1.2, and so on.

Repository Example

```
repository
  moduleA/
    versionedFile1
      (revision number="1.1" user="dave" comment="Created")
      (revision number="1.2" user="mike" comment="New cache
        implementation")
      (revision number="1.3" user="lucy" comment="Fixed defect
        D113")
    versionedFile2
      (revision number="1.1" user="dave" comment="Created")
      (revision number="1.2" user="dave" comment="Updated after
        feedback from Lucy")
    directory1/
      versionedFile3
        (revision number="1.1" user="lucy" comment="Initial
          version")
      ...
  moduleB/
    ...
```

Repository Connection and Permissions



- CVS clients can use various protocols to connect to the server
 - **pserver**, a simple CVS-specific protocol which is fine for most purposes
 - **ssh**, which uses SSL for extra security but usually requires more setup, e.g. creating certificates.
- Access permissions are set on a per-module basis.
 - Unix like permissions (Login, Read, Write) for each user

Working with CVS



- Check out a working copy
 - Once connected to the repository, you can *check out* one or more modules
 - This copies the latest module contents from CVS to your machine, giving you a *working copy* of the module
- Optimistic locking
 - Some version control tools enforce *strict locking* of files which are checked out for editing.
 - ✓ This means that nobody else can edit the file before you've checked it back in
 - A drawback of strict locking is that you're often waiting for another developer to finish editing a file
 - ✓ It's also easy to accidentally leave files locked even though you're not editing them
 - CVS uses a strategy called *optimistic locking* in which locks aren't required
 - ✓ Developers simultaneously edit checked out files, relying on CVS to detect *conflicts*
 - ✓ When conflicts occur, CVS provides tools to help resolve them
 - ✓ If you want to avoid conflicts, CVS also provides a way of tracking who is editing what

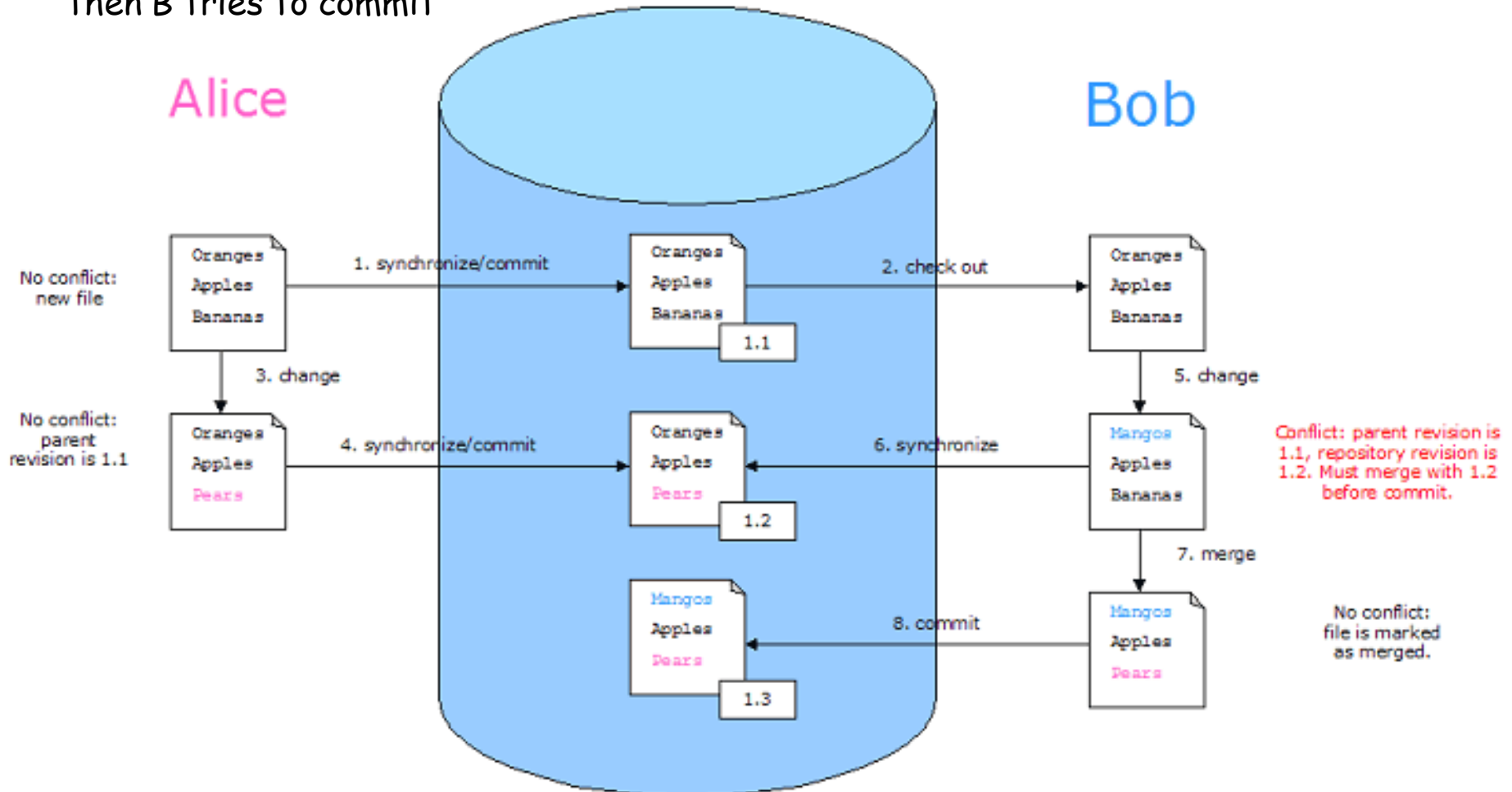
Working with CVS



- Synchronize, update, commit
 - You can use your CVS client to *synchronize* your working copy with the repository
 - This shows you differences between your working copy and the latest module contents in CVS. These differences can be classified as:
 - ✓ *Incoming changes* - changes which have been made in the repository, but not in your working copy
 - ✓ *Outgoing changes* - changes which you've made in your working copy, but not the repository
 - The client can *update* your working copy with some or all of the incoming changes
 - If you have write permission for the module, you can also *commit* (a.k.a *check in*) some or all of the outgoing changes

Working with CVS

- CVS automatically detects when incoming and outgoing changes are in *conflict*
- Conflicts arise when developers A and B both edit the same file, A commits first, and then B tries to commit



Working with CVS

o Merging conflicts

- When a conflict is detected for a file, developer B must *merge* the conflicting changes before the file is committed
- CVS can automatically merge changes which affect different lines in the file (for example the changes above)
- If the changes affect the same lines, a *manual merge* must be performed
- Eclipse provides an excellent merge GUI

o Keeping track of editors

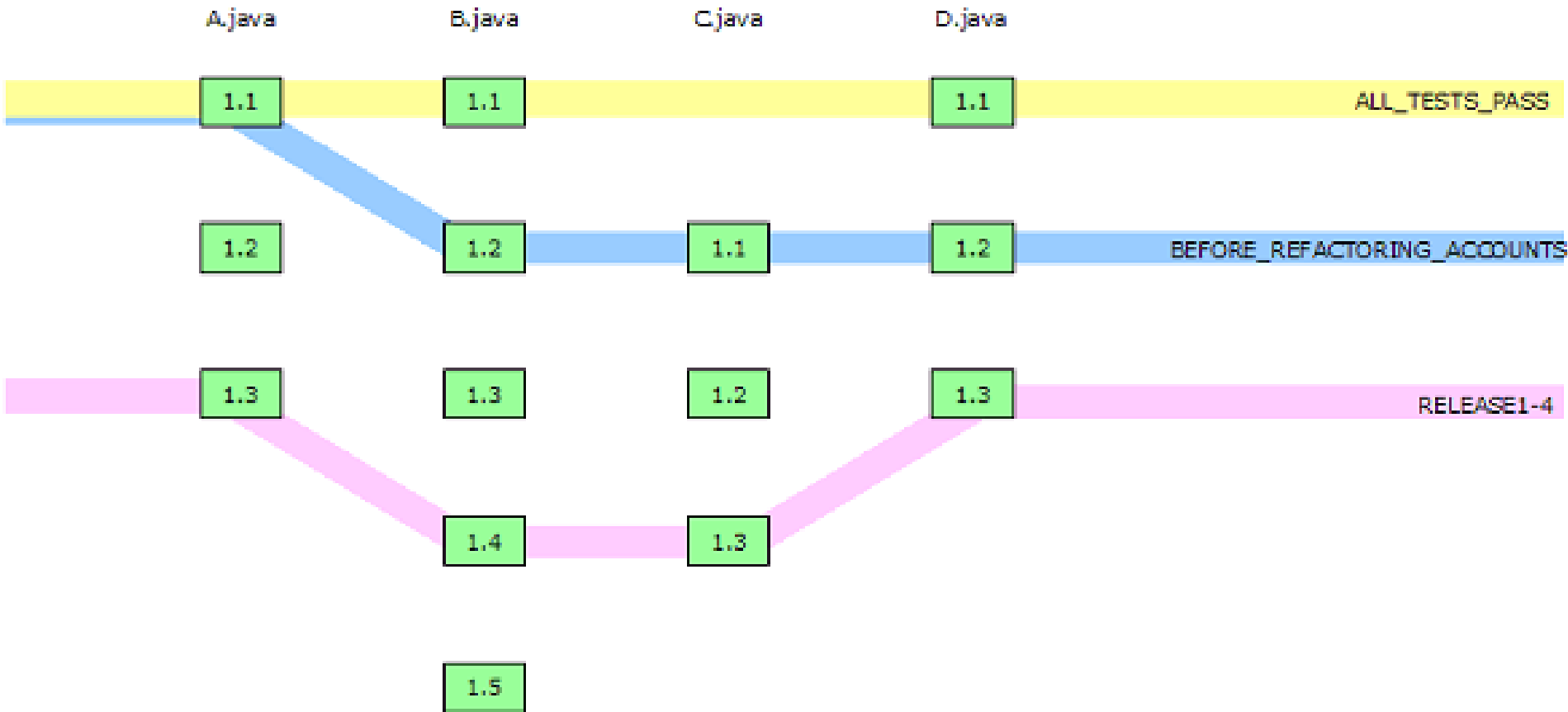
- Although CVS uses an optimistic locking model, you can still keep track of who is editing what
- Your client can be configured to notify the server when you start editing a file
 - ✓ If anyone else tries to edit the file, they'll be given a warning that you are already editing it
 - useful for avoiding conflicts.

Working with CVS

○ Tags

- When developing, you often reach different milestones such
 - ✓ *"Basic system working, all tests pass"*
 - ✓ *"Before refactoring account processing"*
 - ✓ *"After refactoring account processing"*
 - ✓ *"Release 1.4"*
 - ✓ *"Fixed defect D113"*
- Marking these milestones is essential for tracking changes and ensuring that you can revert code to a previous state if required.
- In CVS, you mark milestones by *tagging* the module. The tag captures a single revision for *each file in the module*
 - ✓ The tag name should be relevant to the milestone, for example `ALL_TESTS_PASS`

Working with CVS

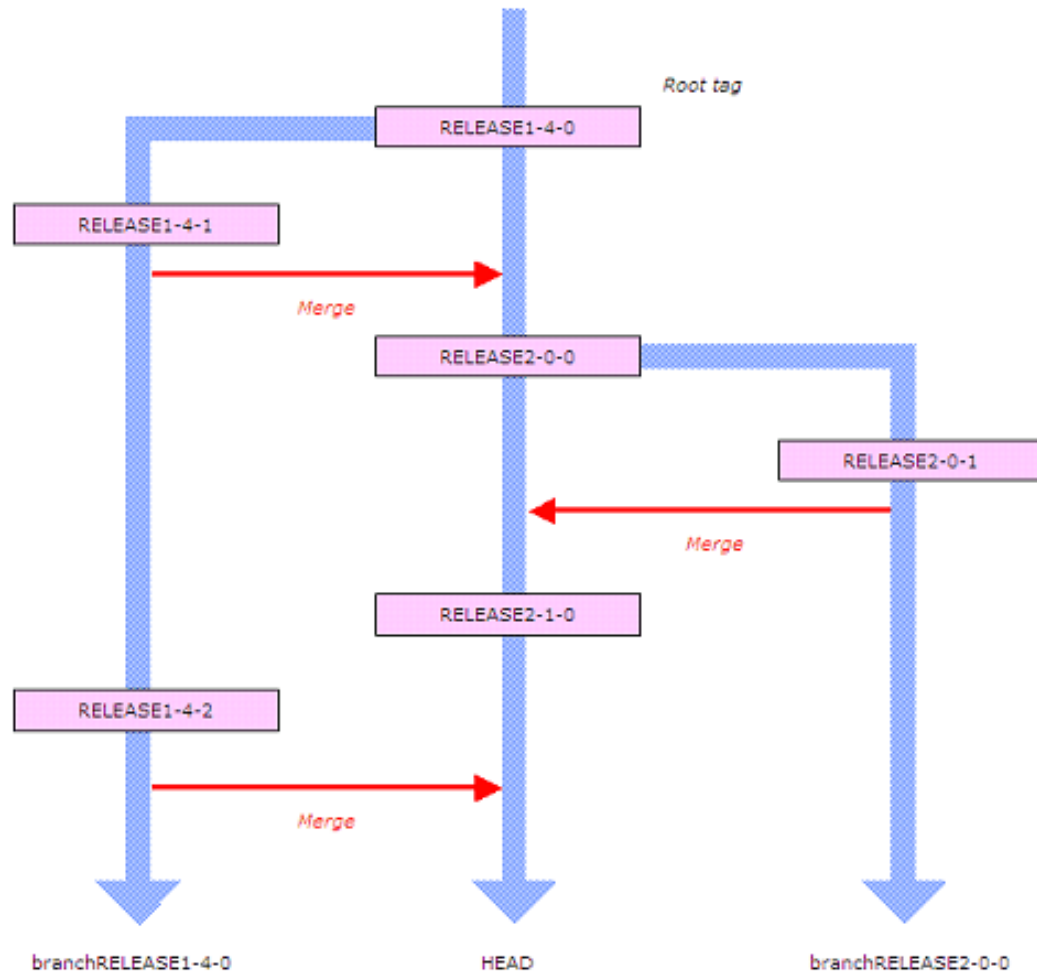


Working with CVS

o Branches and the HEAD

- *Branches* are used to isolate streams of development.
- Unless otherwise specified, checkouts and commits operate on a default branch called *HEAD* (sometimes also called *the trunk*)
- You can create new branches when required
 - ✓ Branches must always start from a tag, called the *root tag*
 - ✓ Any tag can be used as a root tag
- The most common use for branches is to manage parallel development of releases
- E.g. you've released version 1.4.0 of your application and development for 2.0.0 has started on the trunk. If a user reports a critical bug in 1.4.0, you can create a branch using `RELEASE1-4-0` as the root tag

Working with CVS



- When you start working in the branch, the module contents will be exactly the same as they were for release 1.4.0
 - Your updates and commits will act on the branch only
- Tags can be defined on branches in the same way that they're defined on the trunk
- Branches can be created from branches
 - although it's best to stick to one level of branching if possible

Working with CVS



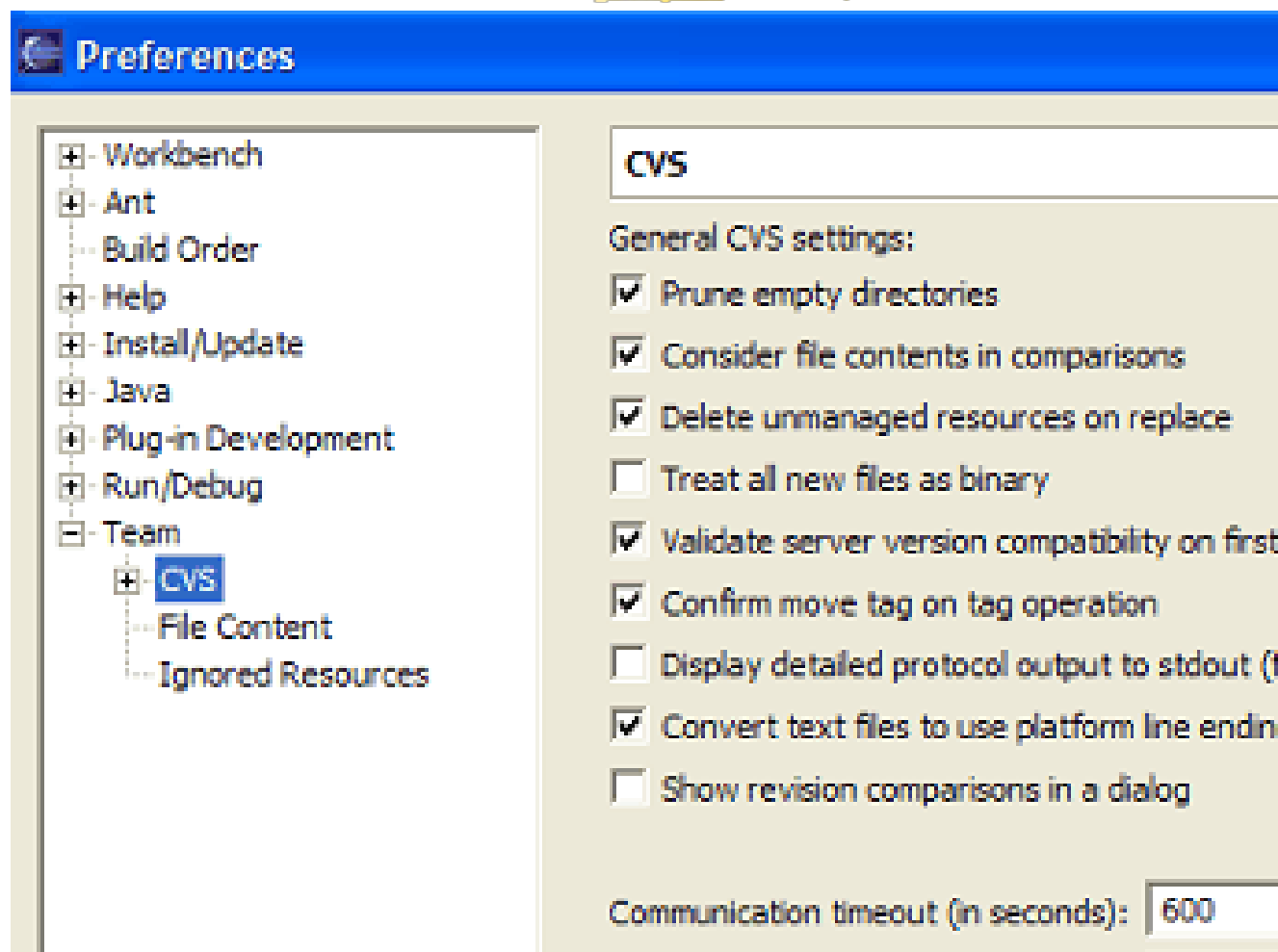
o Merging branches

- You can *merge* changes from one branch to another when required.
- For example, in the diagram above we merge the bugfix changes from release 1.4.1 back into the trunk so that they're included in release 2.0.0
- Merging is a similar process to synchronising; there can be incoming changes (from branch A to branch B), outgoing changes, and conflicts
- The Eclipse GUI makes branch merging easy

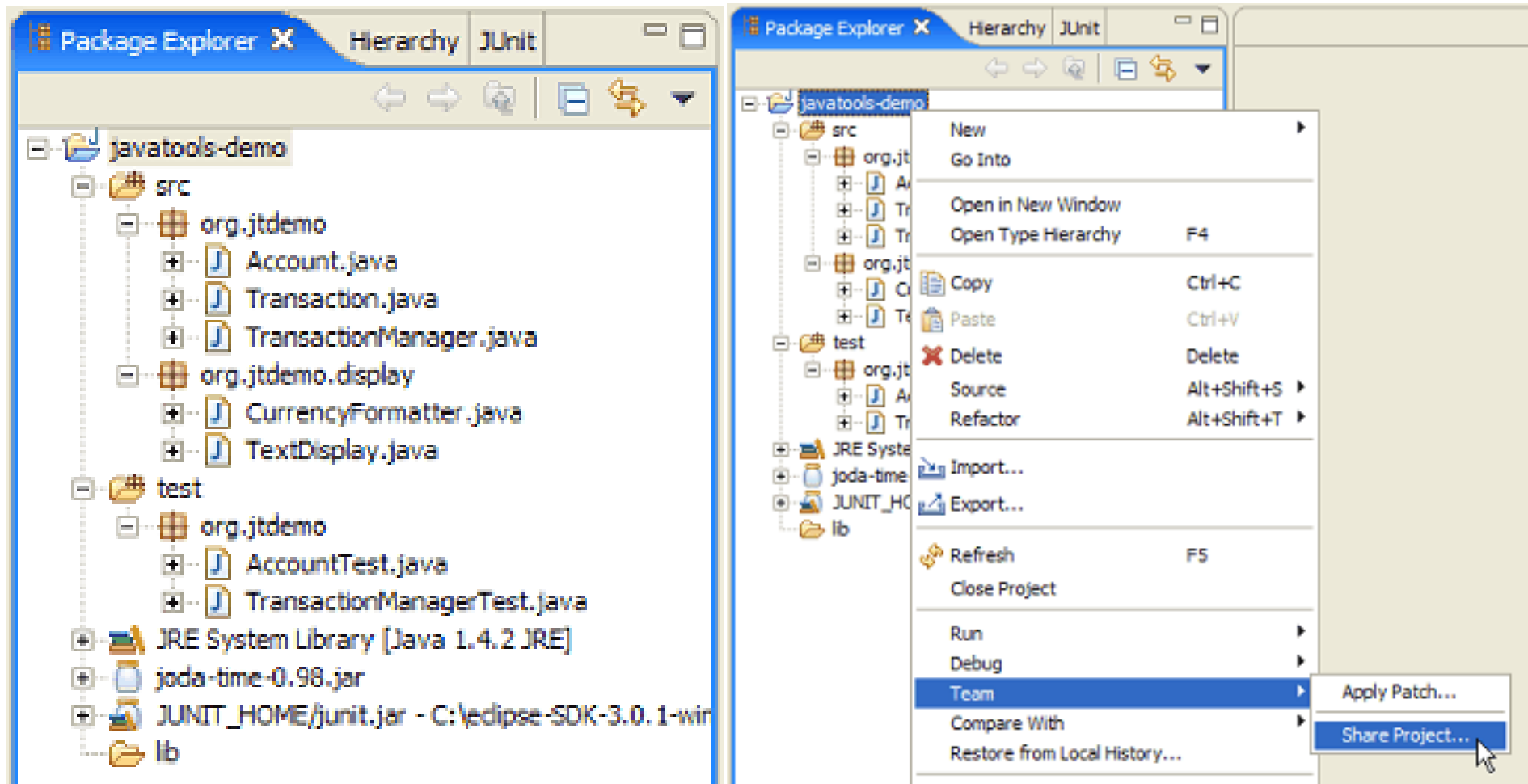
o Viewing history and differences

- CVS provides various ways of viewing the revision history of individual files or file sets
- Allows you to view differences between revisions, tags, and branches.

CVS and Eclipse - Setup



Committing a project to CVS for the first time (project owner)



Committing a project to CVS for the first time (project owner)

Share Project

Enter Repository Location Information

Define the location and protocol required to connect with an existing CVS repository.

Location

Host: cvs.dev.java.net

Repository path: /cvs

Authentication

User: martinplu

Password: *****

Connection

Connection type: pserver

Use Default Port

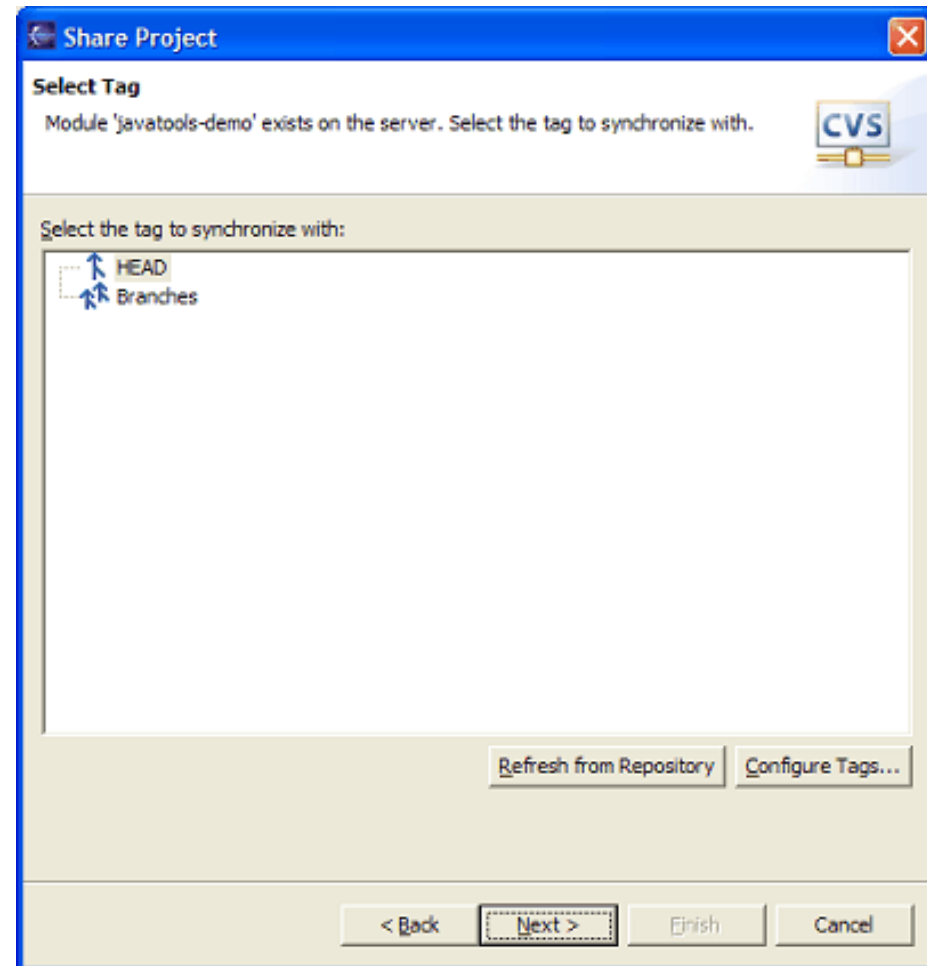
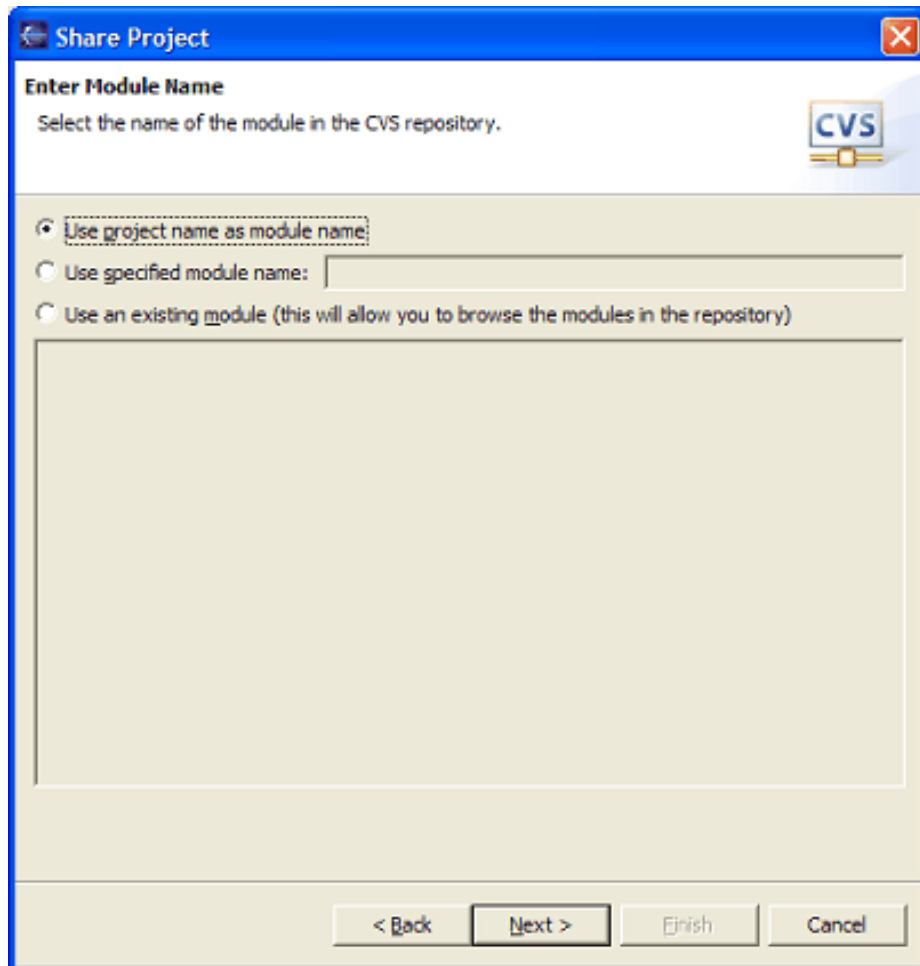
Use Port: _____

Save Password

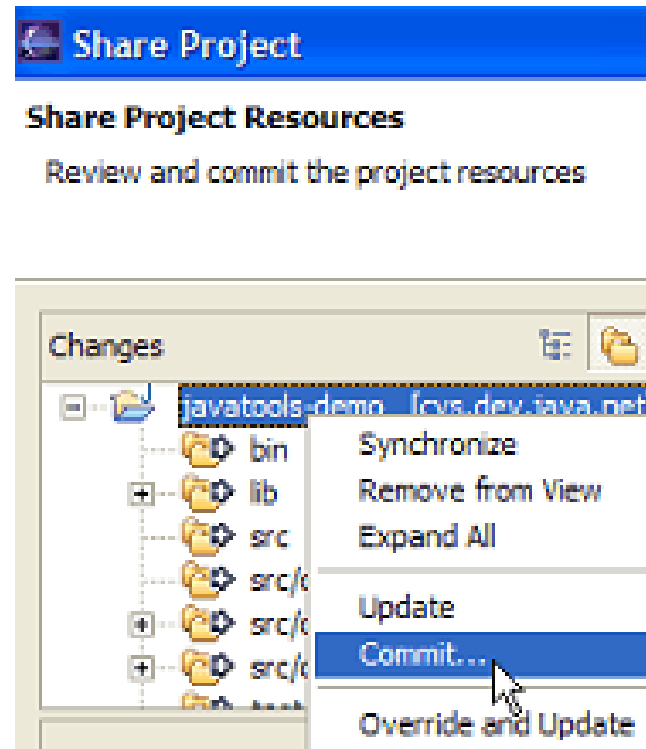
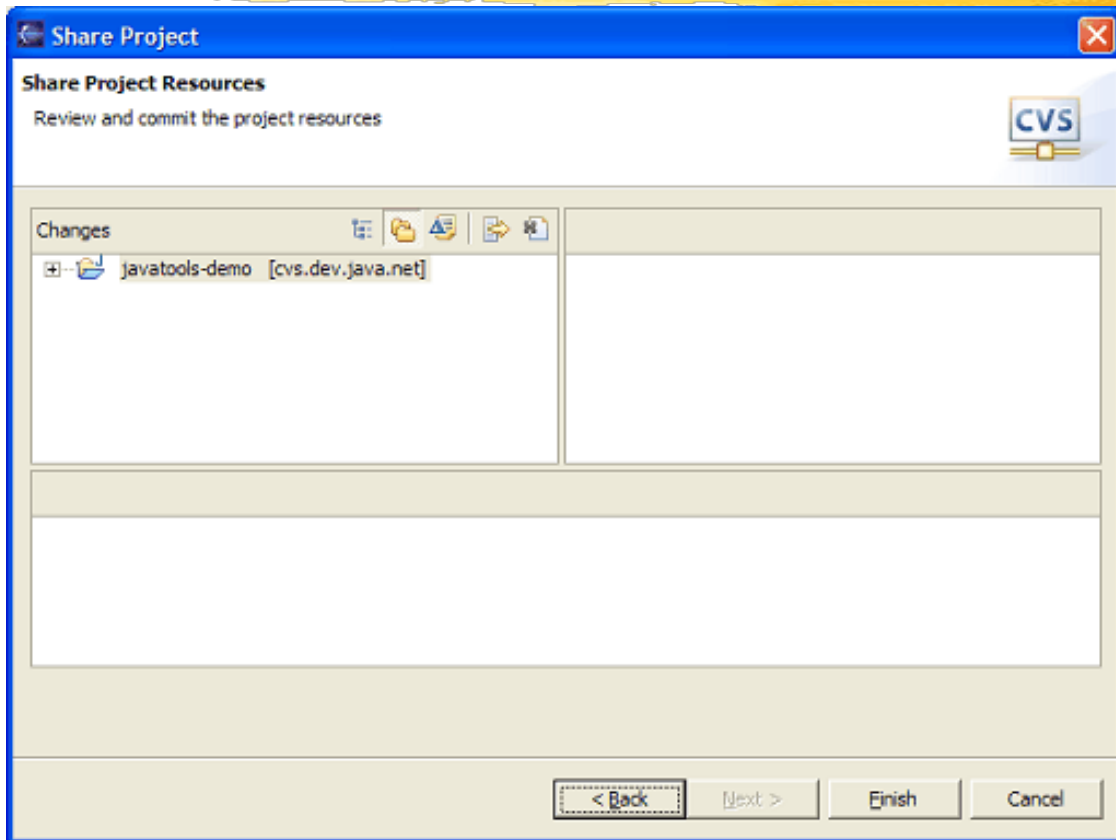
⚠ Saved passwords are stored on your computer in a file that's difficult, but not impossible, for an intruder to read.

< Back Next > Finish Cancel

Committing a project to CVS for the first time (project owner)

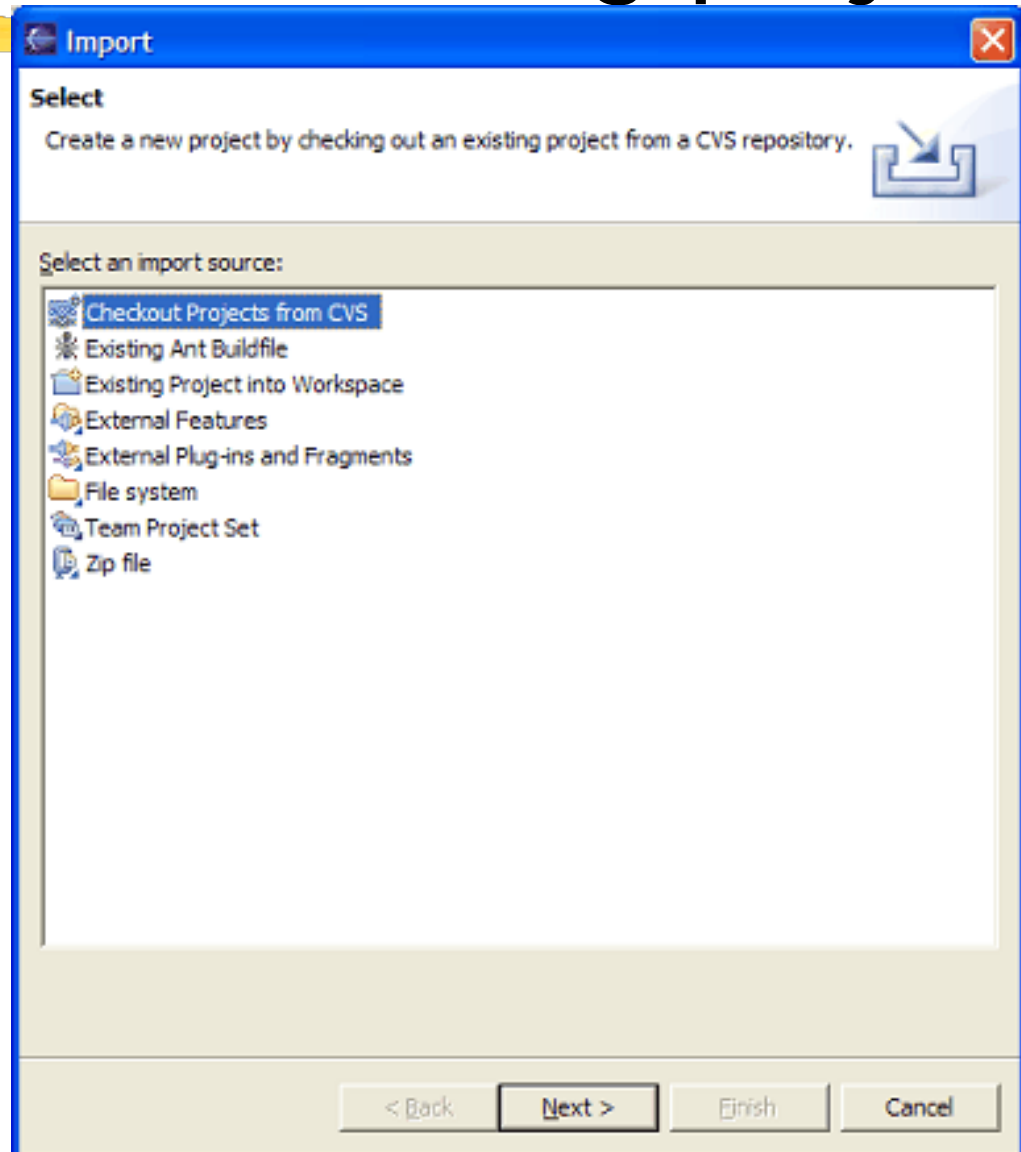
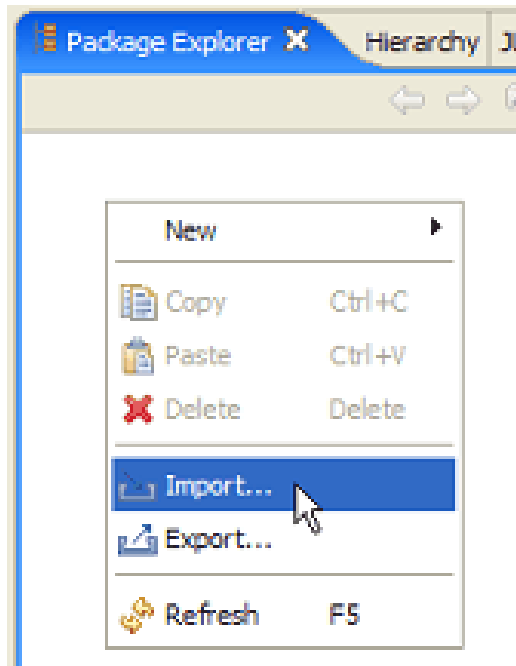


Committing a project to CVS for the first time (project owner)



- Remember to commit your project!

Checking out an existing project



Checking out an existing project

Checkout from CVS

Enter Repository Location Information
Define the location and protocol required to connect with an existing CVS repository.

Location
Host: cvs.dev.java.net
Repository path: /cvs

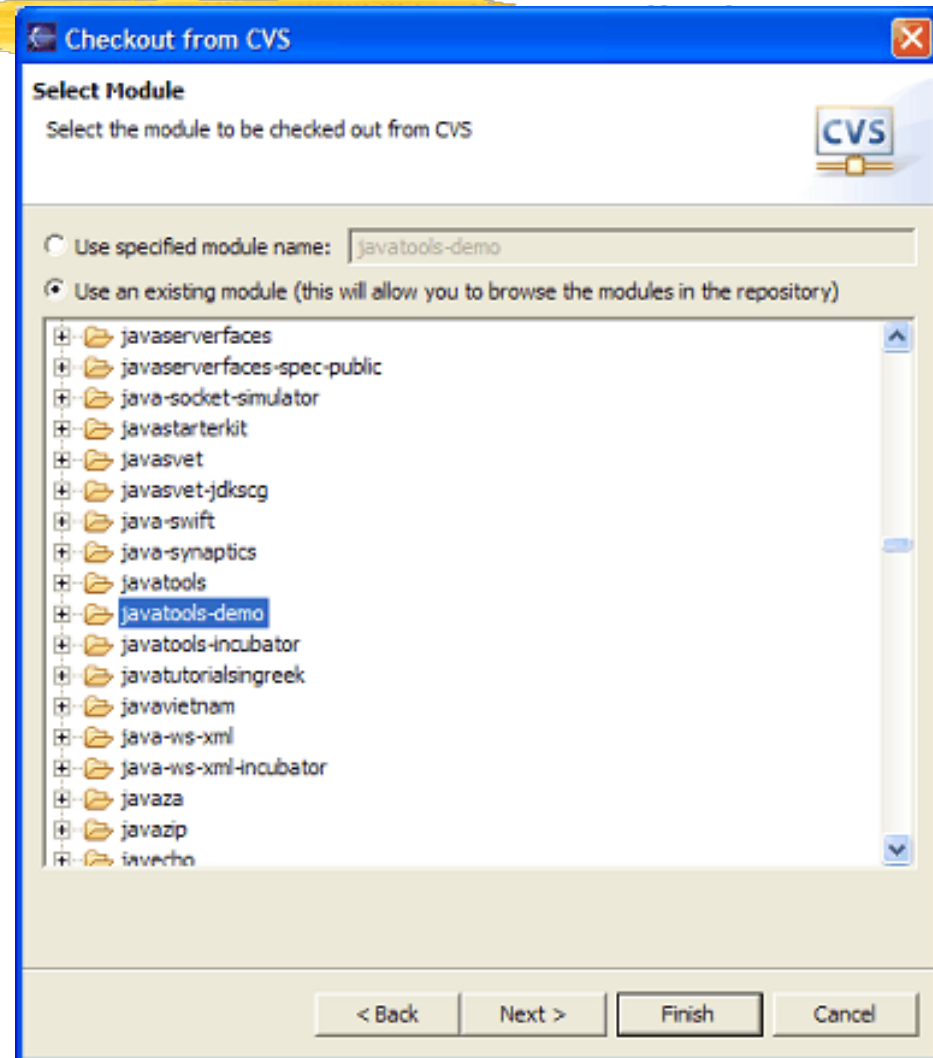
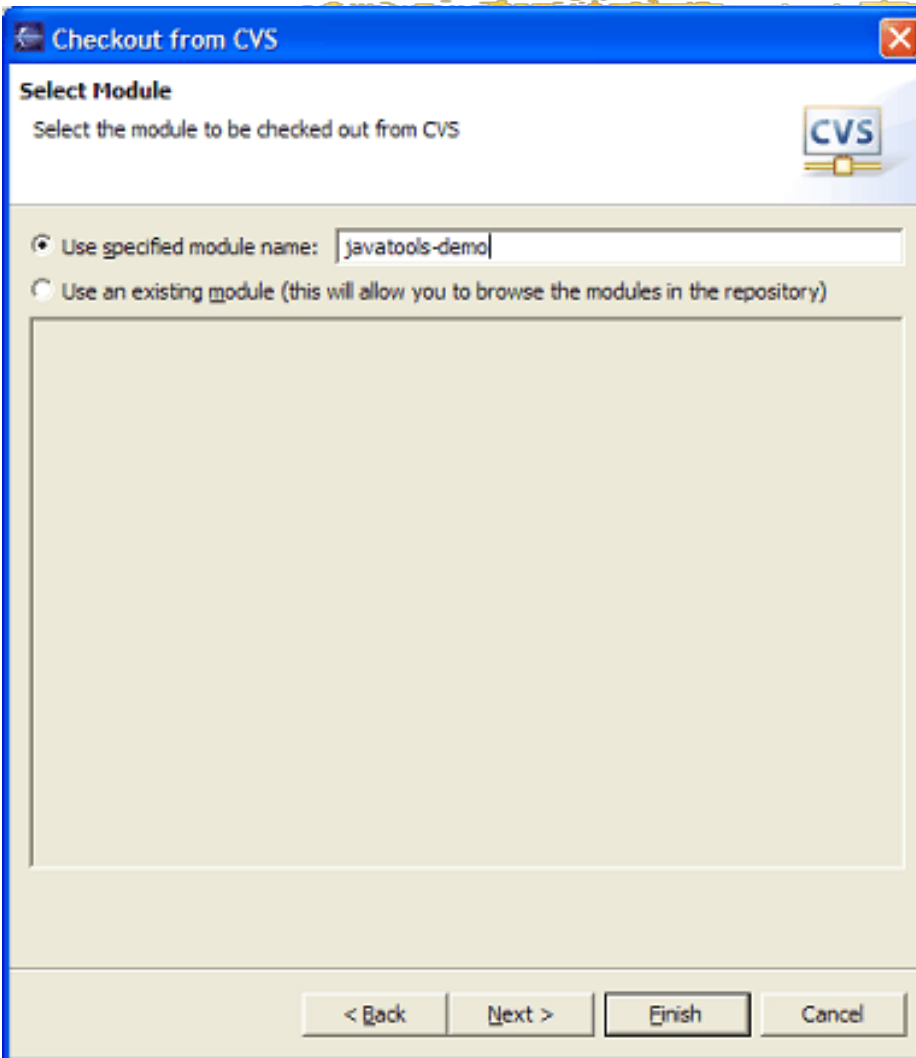
Authentication
User: martinplu
Password: *****

Connection
Connection type: pserver
 Use Default Pgrt
 Use Port:

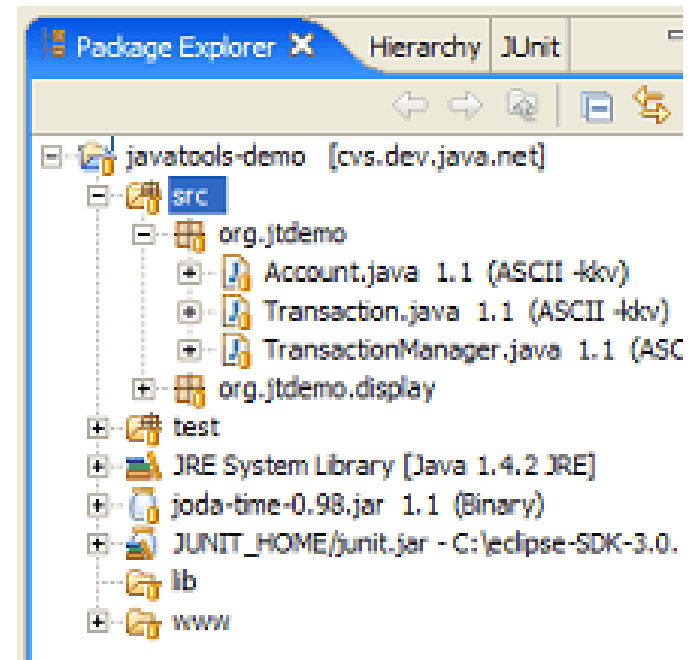
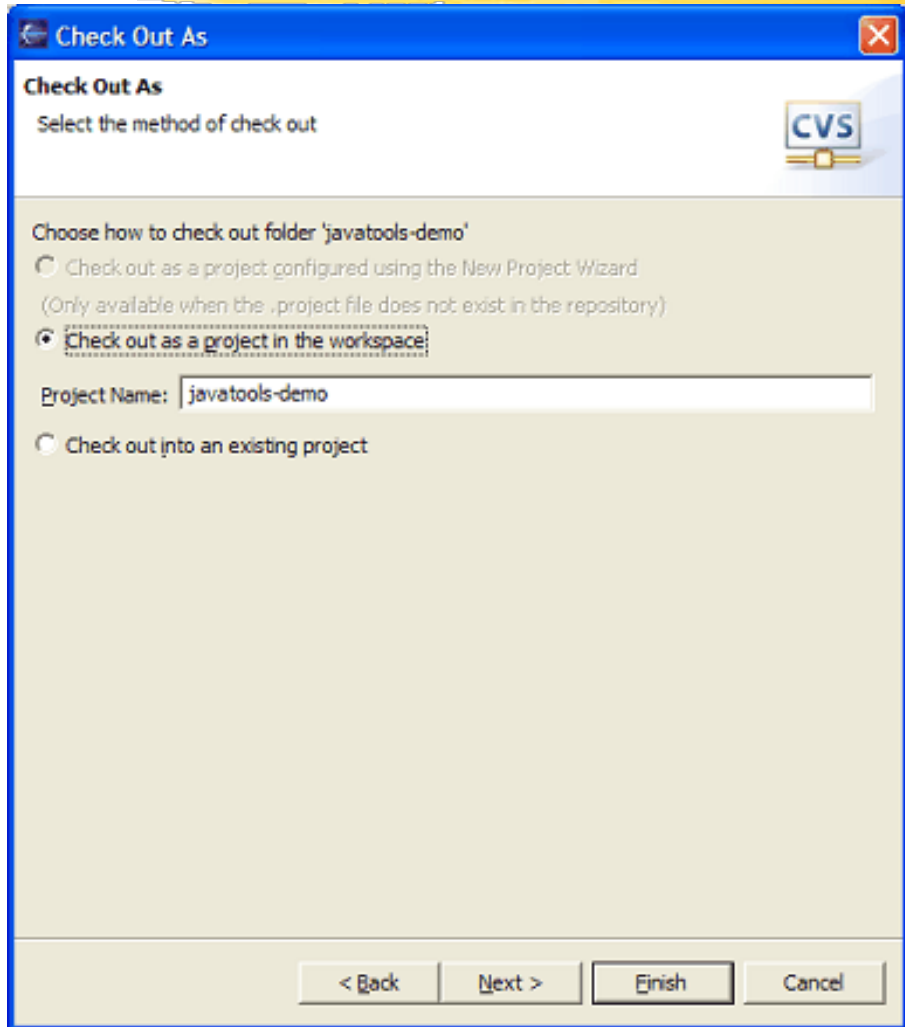
Save Password
 Saved passwords are stored on your computer in a file that's difficult, but not impossible, for an intruder to read.

< Back Next > Finish Cancel

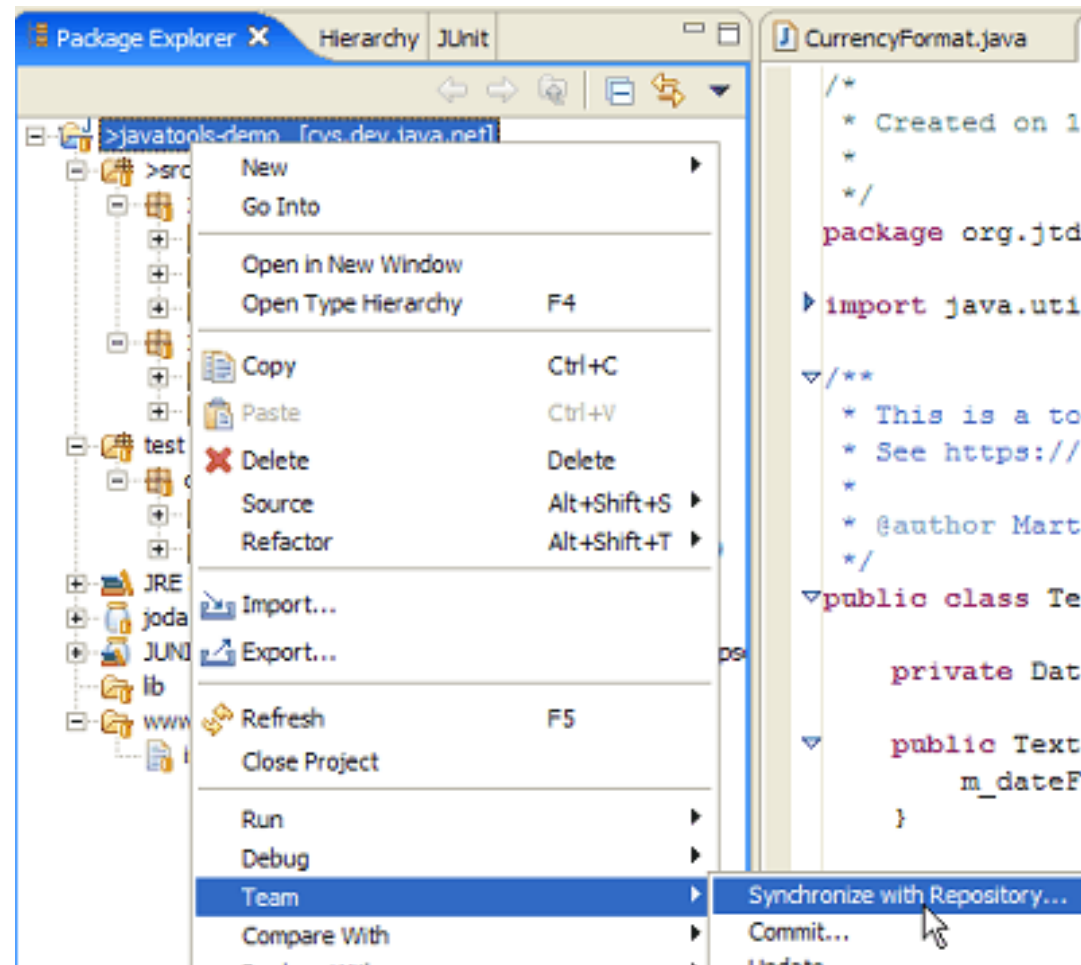
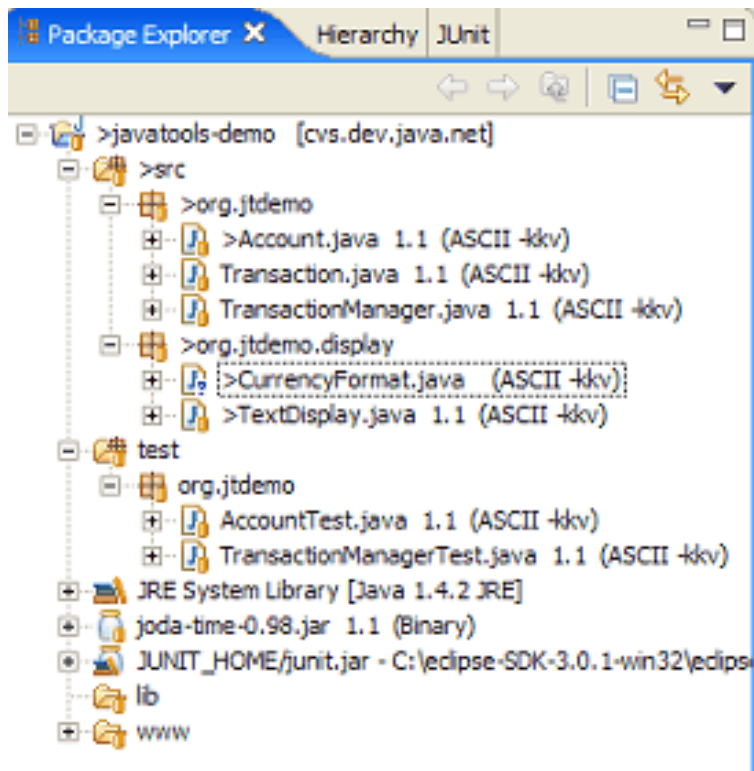
Checking out an existing project



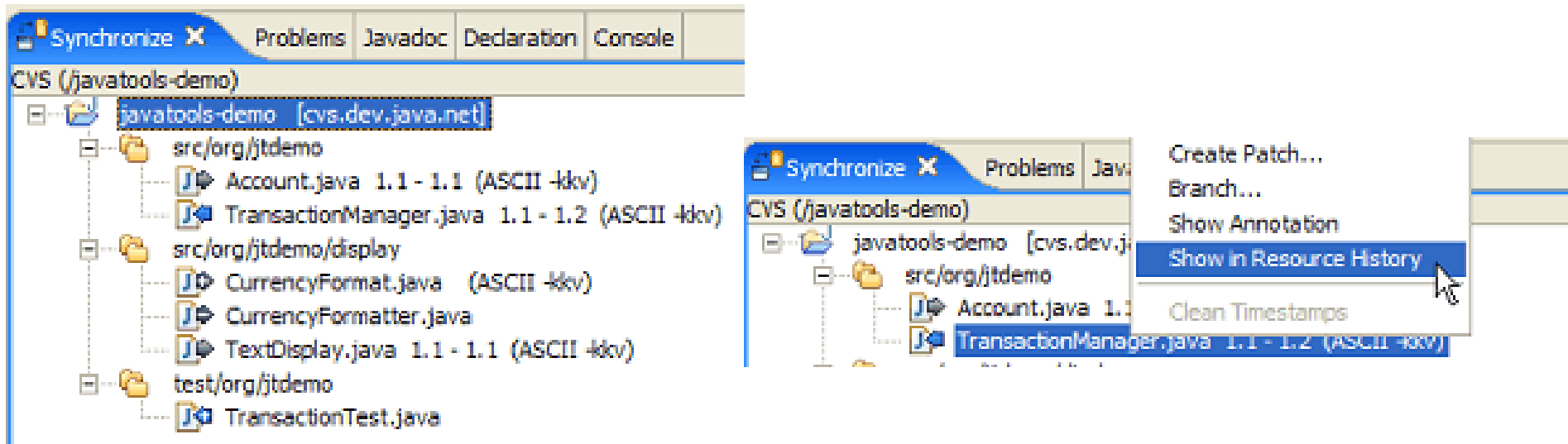
Checking out an existing project



Synchronizing with the repository - no conflicts



Synchronizing with the repository - no conflicts



The screenshot shows the 'CVS Resource History' window for 'TransactionManager.java'. The table displays the following data:

Revision	Tags	Date	Author	Comment
1.2		19/02/05 12:07	spiffy	Added getter for accounts
*1.1		14/02/05 20:33	martinplu	Created

Synchronizing with the repository - no conflicts

The screenshot displays the 'Java Source Compare' window in an IDE, comparing a local file with a remote file. The window is titled 'TransactionManager.java' and shows the 'Compilation Unit' structure for 'TransactionManager' with the method '@getAccounts()'. The 'Local File (1.1)' pane contains the following code:

```
@return Returns the day.
*/

public DateTime getDay() {
    return m_day;
}

/**
 * Get a List of all transactions which were executed on the cu
 * @return Returns the todaysTransactions.
 * @see #getDay()
 */

public List getTodaysTransactions() {
    return m_todaysTransactions;
}
}
```

The 'Remote File (1.2)' pane contains the following code:

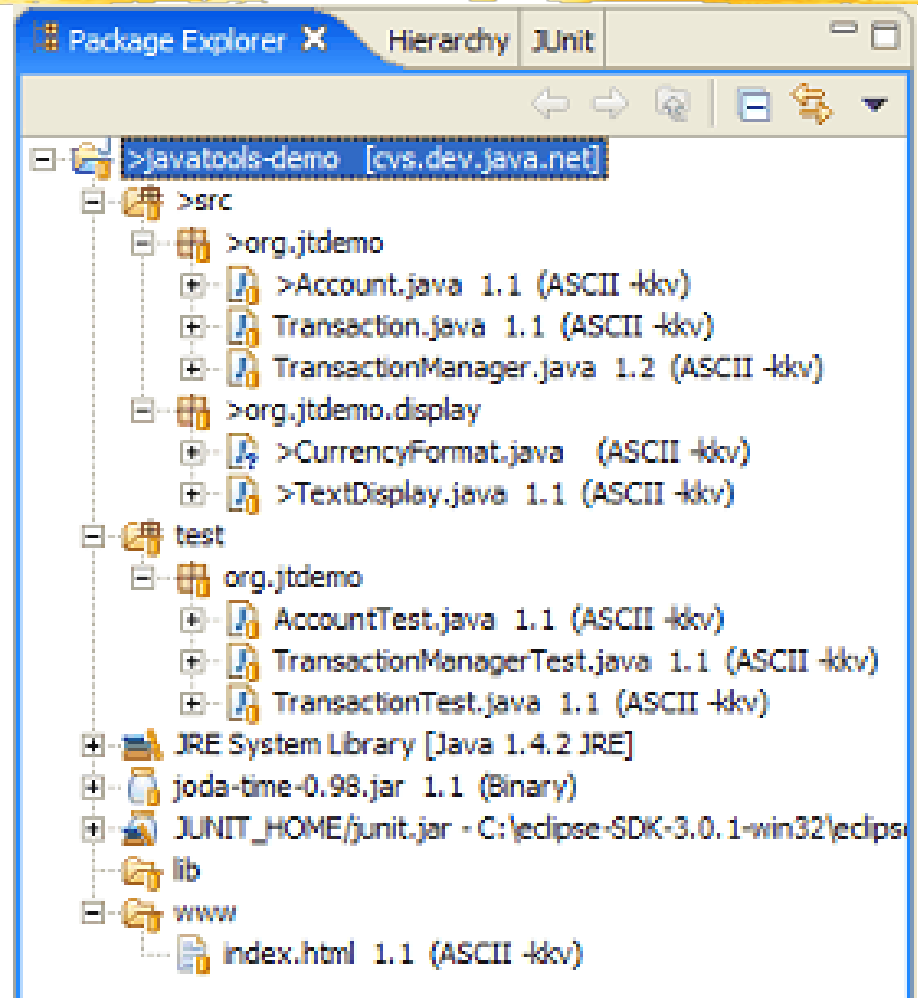
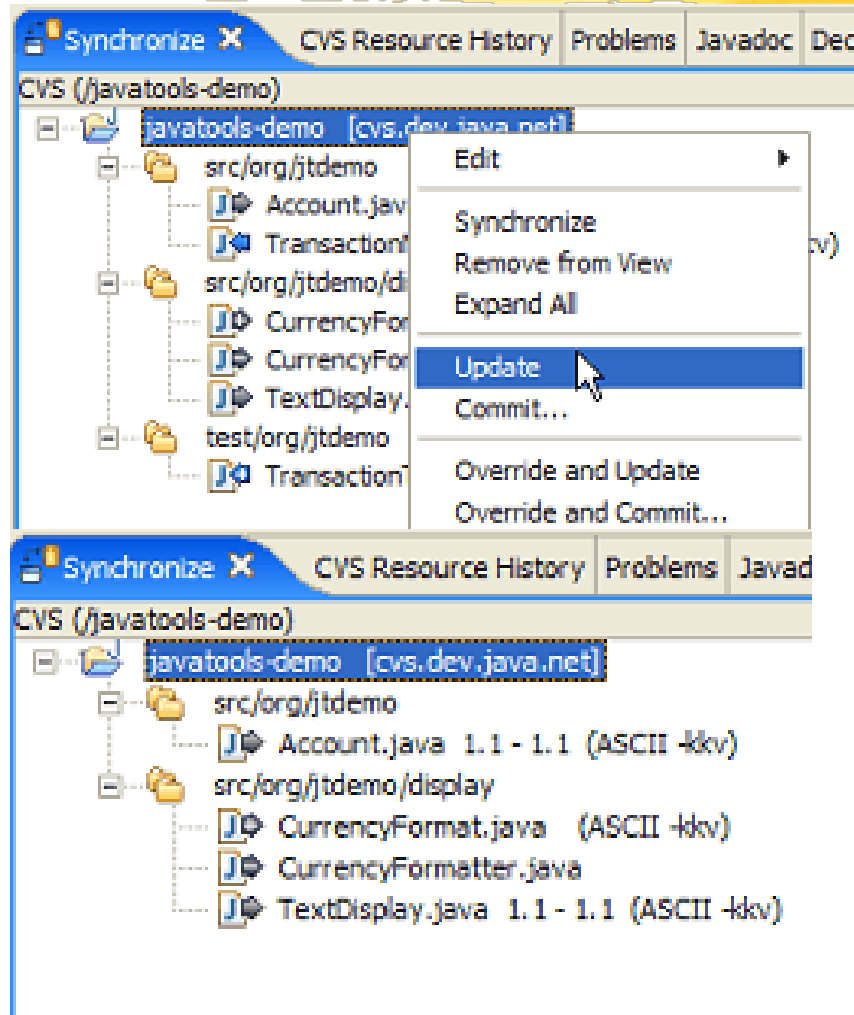
```
/**
 * Get a List of all transactions which were executed on the
 * @return Returns the todaysTransactions.
 * @see #getDay()
 */

public List getTodaysTransactions() {
    return m_todaysTransactions;
}

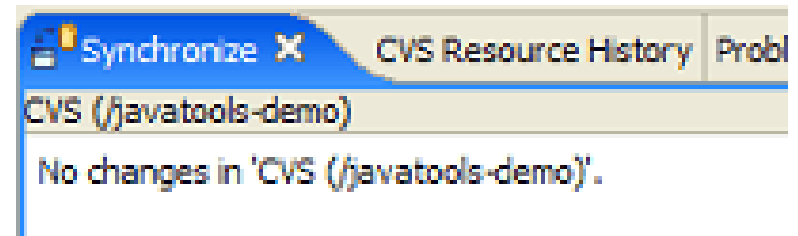
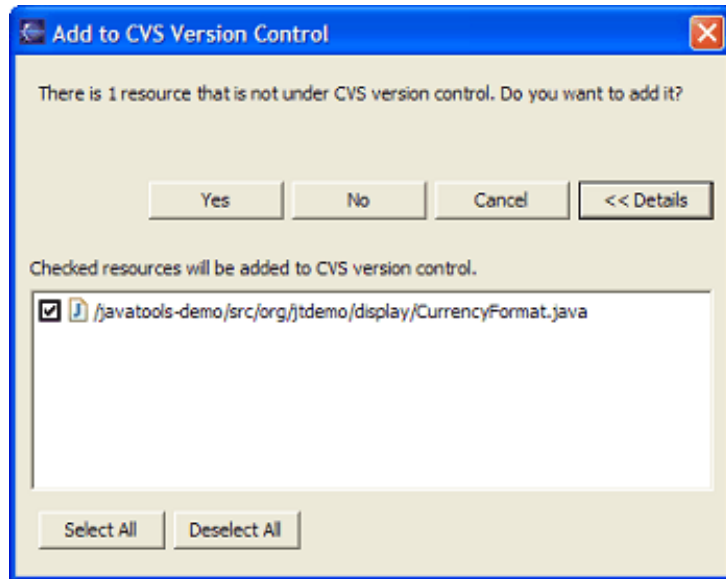
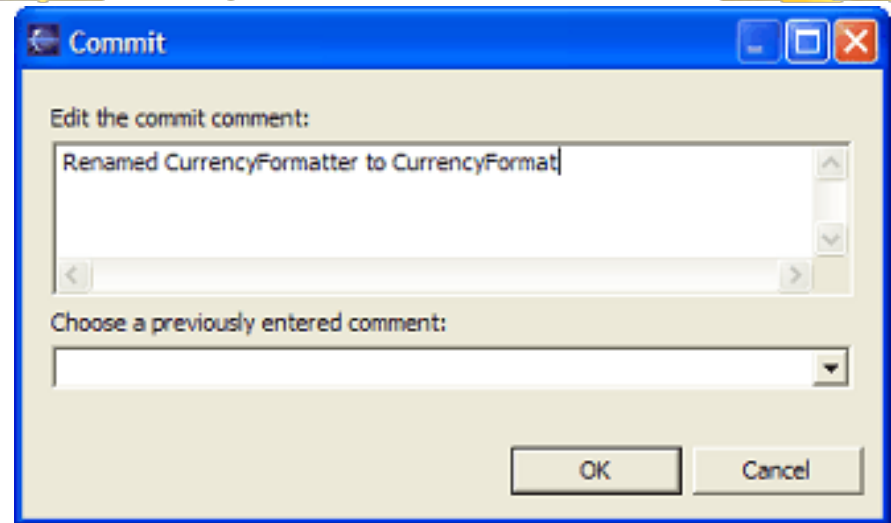
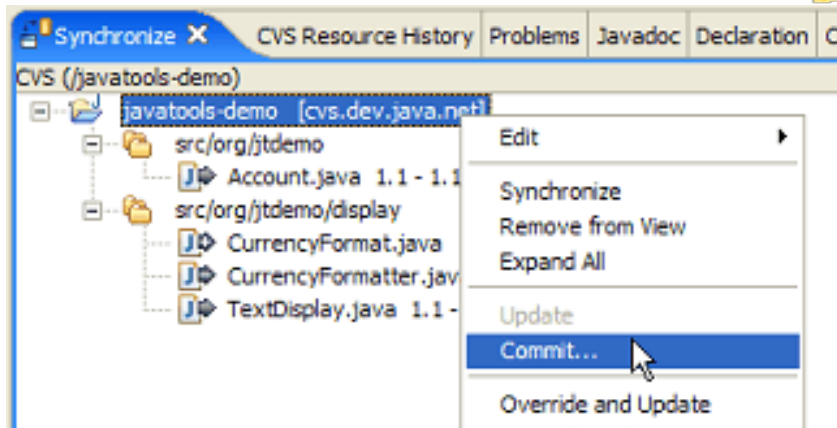
/**
 * @return Returns the accounts.
 */
public Map getAccounts() {
    return m_accounts;
}
}
```

A blue box highlights the `getAccounts()` method in the Remote File pane.

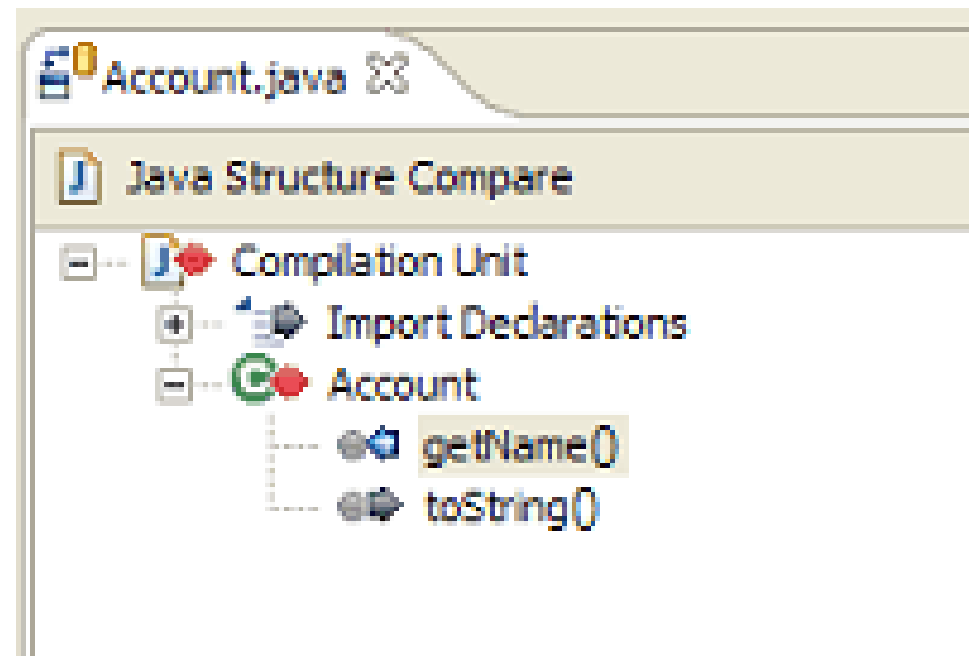
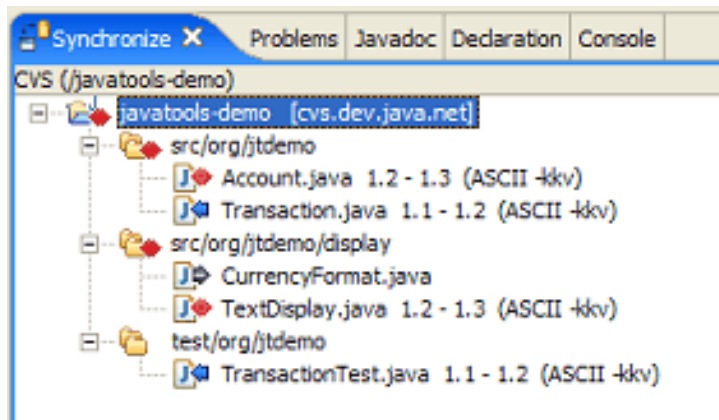
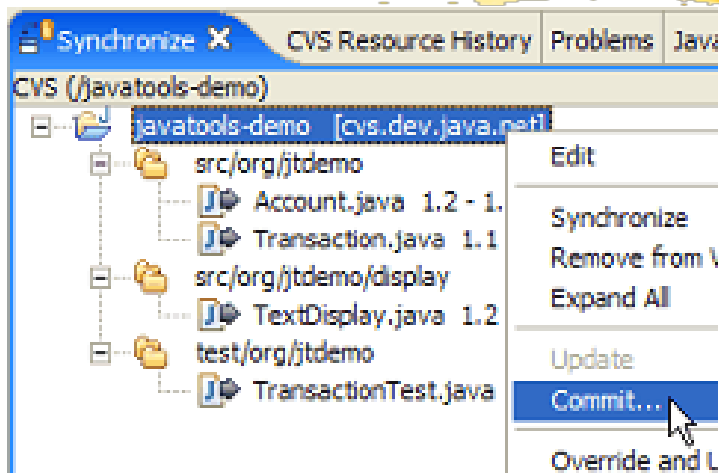
Synchronizing with the repository - no conflicts



Synchronizing with the repository - no conflicts



Synchronizing with the repository - conflicts



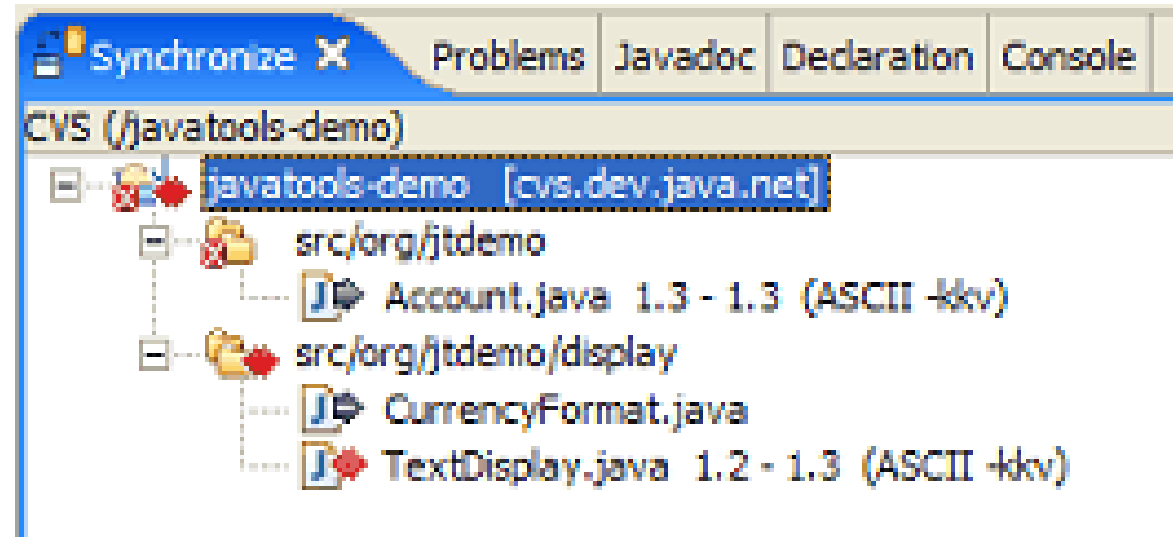
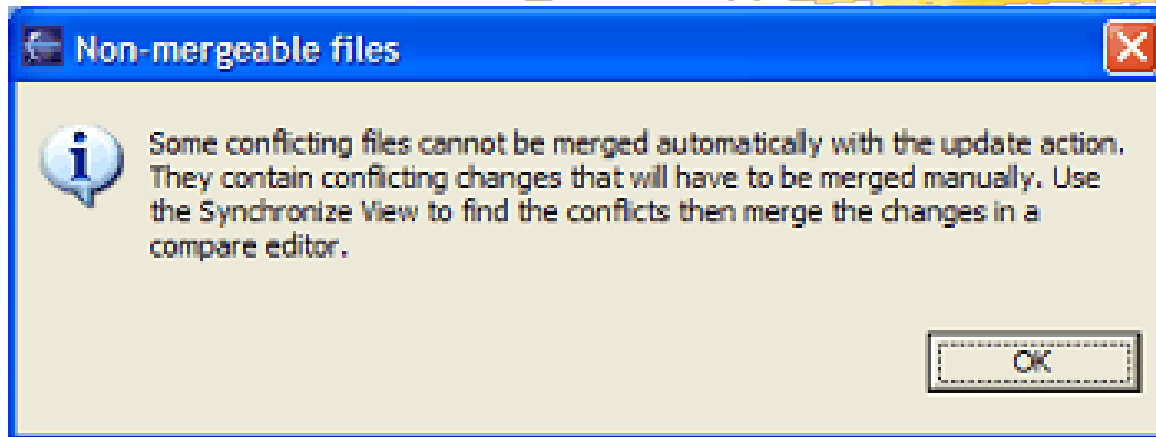
Synchronizing with the repository - conflicts

The screenshot shows an IDE window titled "TextDisplay.java" with a "Java Structure Compare" view on the left and a "Java Source Compare" view on the right. The "Java Structure Compare" view shows a tree structure with "Compilation Unit", "Import Declarations", "TextDisplay", and "update(Observable, Object)". The "Java Source Compare" view shows two versions of the "update" method. The "Local File (1.2)" version has a red highlight on the line: `System.out.println("\t" + NumberFormat.getCurrencyInstance().format(tx.getAmount()));`. The "Remote File (1.3)" version has a blue highlight on the corresponding line: `System.out.println("\t" + tx.to3Str());`.

```
Local File (1.2)
/* (non-Javadoc)
 * @see java.util.Observer#update(java.util.Observable, java.lang.Object)
 */
public void update(Observable o, Object arg) {
    TransactionManager mgr = (TransactionManager)o;
    DateTime day = mgr.getDay();
    System.out.println(m_dateFormat.print(day) + ":");
    List txns = mgr.getTodaysTransactions();
    for (Iterator iter = txns.iterator(); iter.hasNext();) {
        Transaction tx = (Transaction) iter.next();
        System.out.println("\t" + NumberFormat.getCurrencyInstance().format(tx.getAmount()));
    }
}

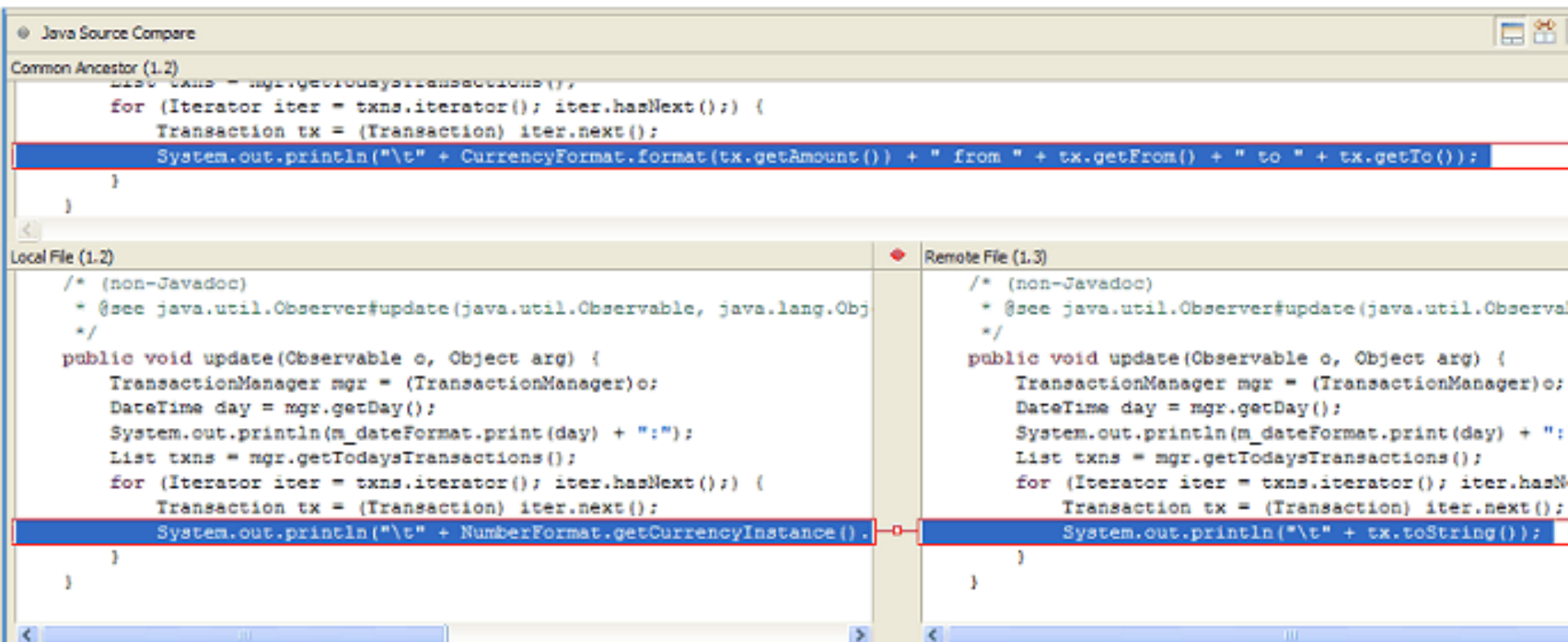
Remote File (1.3)
/* (non-Javadoc)
 * @see java.util.Observer#update(java.util.Observable, java.lang.Object)
 */
public void update(Observable o, Object arg) {
    TransactionManager mgr = (TransactionManager)o;
    DateTime day = mgr.getDay();
    System.out.println(m_dateFormat.print(day) + ":");
    List txns = mgr.getTodaysTransactions();
    for (Iterator iter = txns.iterator(); iter.hasNext();) {
        Transaction tx = (Transaction) iter.next();
        System.out.println("\t" + tx.to3Str());
    }
}
```

Synchronizing with the repository - conflicts



Synchronizing with the repository - conflicts

- The Show Ancestor Pane button  in the Java Source Compare viewer can be used to show the common ancestor of the two changed files.



The screenshot shows the Java Source Compare viewer interface. At the top, the 'Common Ancestor (1.2)' pane displays the shared code from the common ancestor. Below it, the 'Local File (1.2)' and 'Remote File (1.3)' panes show the differences between the local and remote versions. The local file uses `NumberFormat.getCurrencyInstance()` for formatting, while the remote file uses `tx.toString()`. A red vertical bar in the center indicates the line-by-line comparison between the two files.

```
Common Ancestor (1.2)
    List txns = mgr.getTodaysTransactions();
    for (Iterator iter = txns.iterator(); iter.hasNext();) {
        Transaction tx = (Transaction) iter.next();
        System.out.println("\t" + CurrencyFormat.format(tx.getAmount()) + " from " + tx.getFrom() + " to " + tx.getTo());
    }
}

Local File (1.2)
/* (non-Javadoc)
 * @see java.util.Observer#update(java.util.Observable, java.lang.Obj
 */
public void update(Observable o, Object arg) {
    TransactionManager mgr = (TransactionManager)o;
    DateTime day = mgr.getDay();
    System.out.println(m_dateFormat.print(day) + ":");
    List txns = mgr.getTodaysTransactions();
    for (Iterator iter = txns.iterator(); iter.hasNext();) {
        Transaction tx = (Transaction) iter.next();
        System.out.println("\t" + NumberFormat.getCurrencyInstance().
    }
}

Remote File (1.3)
/* (non-Javadoc)
 * @see java.util.Observer#update(java.util.Observable, java.lang.Obj
 */
public void update(Observable o, Object arg) {
    TransactionManager mgr = (TransactionManager)o;
    DateTime day = mgr.getDay();
    System.out.println(m_dateFormat.print(day) + ":");
    List txns = mgr.getTodaysTransactions();
    for (Iterator iter = txns.iterator(); iter.hasNext();) {
        Transaction tx = (Transaction) iter.next();
        System.out.println("\t" + tx.toString());
    }
}
```

Synchronizing with the repository - conflicts

