Concurrent Versions System (CVS)

CVS

- Serves two main purposes
  - tracks changes to text files
  - supports collaboration between developers
    - allows concurrent modification of the same files
- Free software under the GNU General Public License (GPL)
- Built on top of Revision Control System (RCS)
  - created by Walter Tichy
- Formerly maintained by Cyclic Software
- Now maintained by SourceGear Corporation
  - www.sourcegear.com
  - they have taken over the cyclic.com domain and maintain that site
    - www.cyclic.com forwards to www.sourcegear.com/CVS

Development Model

- Many other source control tools use a model where files are
  - locked
  - modified
  - unlocked (checked in)

one developer at a time

- CVS uses a model where files are
  - copied
  - modified
  - merged

multiple developers can work on the same file concurrently

Repositories and Working Copies

- Repository
  - stores change history for text files
  - can store files for multiple projects in a single repository
  - the latest version of each file is stored
  - "diffs" for returning to each preceding revision are also stored

This information is stored in a single `,v` repository file for each file that is imported or added to CVS.

- Working Copy
  - one per developer
  - stores a snapshot of projects from a repository
    - could be the latest revisions of all the files or the revisions at a certain point in time
  - changes are made to files in working copies
  - testing is performed on working copy
  - once tested, changes are committed back to the repository

Checkout, Update and Commit

- Checkout
  - creates a working copy from a given project in a repository
- Update
  - updates the files in a working copy from the repository
  - changes committed in the repository since the last update
    are merged into files in the working copy,
    even if the files in the working copy have been modified
- Commit
  - copies changes to files in the working copy to the repository
  - creates new revisions in the repository

Typical Scenario

- Here's a typical scenario for how CVS is used
  - `cvs checkout`
    - to create a working copy (only done once)
  - `cvs update`
    - to merge change other developers have committed to the repository
      into your working copy
    - useful if time has passed since checkout or last update
  - `cvs edit`
    - makes given files in your working copy writable
    - optionally notifies other developers that you intend to modify given files
  - make changes and test in your working copy
  - `cvs commit`
    - if other developers have changed the same files you changed since the last update,
      CVS will ask you to merge those changes into your local copy
      before allowing you to commit your changes

CVS Options

- Two types of options
- Global options
  - appear between `cvs` and the command name
  - apply to most commands
  - examples are `-q` (quiet) and `-Q` (very quiet) which reduce the amount of output from CVS commands
- Command options
  - appear after the command name
  - specific to each command
- CVS command syntax

```
> cvs [global-options] command [command-options] [parameters]
```

CVS Command File Parameters

- Many CVS commands accept a list of files on which to operate
- When omitted they typically operate on all files in the current directory and all files in subdirectories of the current directory (recursive)
- Examples
  - `cvs update` updates all files in or below the current directory
  - `cvs commit` commits changes to all files in or below the current directory

Log Messages

- CVS commands `add`, `commit` and `import`
  accept log messages specified using the `-m` command option
- If `-m` is not specified, an editor will be launched for entering a log message
  - under UNIX the editor used is determined by the EDITOR environment variable
  - under Windows the default editor is Notepad
- Can modify bad log messages

  `adm` and `rcs` are synonyms

  ```
  > cvs admin -m revision-number:"new-log-message" file-name
  ```
  - can't change log messages for multiple files with one command
    since they each could have a different revision number

Dates

- Many CVS commands accept command options that specify a date
- Examples of valid date specifications
  - `"16 Apr"`
  - `"16 Apr 2000"`
  - `"16 Apr 2000 13:20"`
  - `16/4/2000`
  - `2000-04-16`
  - `"2000-04-16 13:20"`
  - `now`
  - `yesterday`
  - `"2 days ago"`
- Example usage

  ```
  > cvs update -D yesterday
  ```

Repository Login

- When only using one repository, set CVSROOT environment variable to point to the repository
  - under Windows, do this in the System control panel or from a script
  - under UNIX, do this in a startup script
  - Windows command-line examples
    ```
    > set CVSROOT=:pserver:userid@hqipbpu3:/www/cvsroot
    > set CVSROOT=:local:C:\Volkmann\CVS\myRepository
    ```
  - pserver method for accessing remote repositories is common but there are others
- Command

  same as used to create the repository; not needed if CVSROOT is set

  ```
  > cvs -d %CVSROOT% login
  ```

  `logon` and `lgn` are synonyms
- Only required once per repository
- Login information is stored in .cvspass file in home directory for use by subsequent CVS commands
  - `cvs logout` removes login information for current repository from .cvspass

Creating a Repository

- Command

  can use `$CVSROOT` (UNIX) or `%CVSROOT%` (Windows) here

  ```
  > cvs -d repository-path init
  ```
- Repositories are assumed to be remotely accessible unless *`repository-path`* is preceded with "`:local:`"
- CVSROOT directory is created under *`repository-path`*
  - contains administrative files such as cvswrappers, notify, users and modules (maps names to repository directories; see next page)

  cvswrappers, notify and users are covered later
- UNIX
  - example remote repository path
    - `/usr/local/myRepository`
- Windows
  - not considered an ideal platform for hosting remote repositories
  - example local repository path
    - `:local:C:\Volkmann\CVS\myRepository`

AGe-Connect `modules` file

module name | repository directory

```
ageconnect_bin   isp/ageconnect/bin
ageconnect_class isp/ageconnect/class
ageconnect_debug isp/ageconnect/debug
ageconnect_devl -a ageconnect_src ageconnect_doc ageconnect_lib ageconnect_class \
                   ageconnect_bin ageconnect_tools ageconnect_debug
ageconnect_doc   isp/ageconnect/doc
ageconnect_lib   isp/ageconnect/lib
☆ ageconnect_src   isp/ageconnect/src
ageconnect_tools isp/ageconnect/tools
```

Adding a Project to the Repository

- Command

  `> cvs import` *`project-name vendor-tag release-tag`*

  `im` and `imp` are synonyms

  especially useful for managing third party software
- Common options

  `-m "`*`log message`*`"`
- Run from top-level project directory
- Example

```
> cvs import -m "initial import" \
  SecurityService AGEdwards V1
```

- Imports ALL files in the current directory and its subdirectories
  - if these contain files that should not be part of the project or do not require version control (such as .class files), move them before performing the import
  - creates `,v` files in the repository that will be used to store revision info.

  the initial revision number of all files will be 1.1.1.1
- Does NOT change the current directory to a working copy (coming up)

@ Character

- @ is the field delimiter in RCS files which are used by CVS
- All occurrences of @ characters in files must be replaced by @@
- CVS does this automatically when files are added and committed
- CVS automatically changes them back to a single @
  when files are checked out or updated
- You shouldn't have to hand edit `,v` files but if you ever do …
  - use @@ instead of @

Creating a Working Copy

causes working copy files to be read-only

can be directory names in the repository (specified in import) or modules names defined in the modules file

- Command

```
> cvs [-r] checkout project-names
```

`co` and `get` are synonyms

- Creates a directory below the current directory for each project
- Each directory within the project will contain a CVS directory
  which can be ignored
- Safety precaution
  - if you just imported the project it is a good idea to move the local copies of those files before creating a working copy in the same directory to insure that the import worked correctly
- Common options

  `-r` *`tag`*

  more on this later

  - gets specific tagged revisions of the files
  - future updates will only merge files with that tag (sticky tag)

Deleting a Working Copy

same as used to create the repository; not needed if CVSROOT is set

`re` and `rel` are synonyms

- Command

```
> cvs [-d repository-path] release [-d] [projects]
```

- Generally only do this when you don't anticipate working on the project again
  - recorded in history file
- Must execute this from the top of the the working copy directory
- Doesn't work if working copy contains uncommitted changes
- To have it delete the working copy directory for you
  - use `-d` command option after "`release`"

Committing Changes to Repository

- Command

```
> cvs commit -m "log message" [files]
```

`ci` and `com` are synonyms

- Common options

```
-m "log message"
```

- Only files that have been modified will receive a new revision number
- Out-of-date files
  - other developers may have committed changes to the repository
    for the same files you have modified and wish to commit
  - those changes must be merged into your working copy
    before CVS will allow your changes to be committed
  - CVS will tell you if this needs to be done and will refuse the commit until it is

Sample Session

(under Windows)

```
$ set CVSROOT=:local:C:\Volkmann\CVS\repository
$ cvs -d %CVSROOT% init
$ cd \SecurityService
$ cvs import -m "initial import" SecSvc AGEdwards V1
$ cd working-copy-directory
$ cvs -r checkout SecSvc
$ cvs watch on
$ cvs edit TestGUI.java
edit the file
$ cvs commit -m "added comments"
$ cvs edit TestGUI.java
edit the file
```

covered later

The top directory of the project is C:\SecurityService.
The name given to the project is SecSvc.

Subsequent example output is based on this session.
Note that the last change hasn't been committed yet.

Revision Numbers

- Importing a project into the repository
  - creates two revisions for each file, 1.1.1.1 and 1.1
- Adding a file to the repository creates revision 1.1.1.1
- After adding a file it must be committed; this creates revision 1.1
- Subsequent commits increment the decimal portion of the revision number
  - 1.2, 1.3, …, 1.9, 1.10, 1.11, …, 1.99, 1.100, 1.101, …
- It is never necessary to change the integer portion of the revision number
  - if desired, can change with cvs commit
  - example
    ```
    > cvs commit -m "incrementing major revision" -r 2.0
    ```
  - for files that haven't changed in the working copy this creates new revisions with identical contents
  - typically want to clear sticky tags after this
    ```
    > cvs update -A
    ```

Updating Working Copy

- Command
  ```
  > cvs update [files]
  ```
  `up` and `upd` are synonyms
- Merges changes committed to the repository since checkout or last update with your working copy
- Reports conflicts if any
  - must be resolved manual
  - conflict markers (covered soon) are placed in the files of your working copy to assist

Updating Working Copy (Cont'd)

- Common options

  `-d`
  - adds directories that are in the repository but not in your working copy

  `-P`
  - prunes empty directories (often directories that have been deleted from CVS)

  `-A`
  - clears sticky tags so the latest revisions can be obtained
  - merges the latest repository revisions of the files with your working copy

  `-r` *tag*
  - merges specific tagged revisions of the files with your working copy
  - future updates will only merge files with that tag (sticky tag)

  `-p`
  - sends file to stdout
  - can also be used to get an old revision without setting a sticky tag
    ```
    > cvs update -p -r revision-number filename > filename
    ```

Update Codes

- `cvs update` lists information about about its actions and the state of your working copy
- These codes precede each listed file
  - `U` - not modified in working copy but modified in repository; merged successfully
  - `M` - modified in working copy and possibly in repository; merged successfully
  - `C` - modified in working copy and repository; merge resulted in conflict
  - `A` - added to working copy (with `cvs add`) but not committed to repository
  - `?` - in working copy but haven't added (with `cvs add`) or committed to repository
  - `R` - removed from working copy (with `cvs remove`) but not committed to repository
  - `P` - like `U` but CVS server sent a patch instead of entire file (same result as `U`)

adding and removing files is covered later

Conflict Markers

- Placed in files when `cvs update` is unable to merge repository changes into the working copy
- Format

may be useful in a grep

```
<<<<<<< filename
  changes in working copy
=======
  changes in repository
>>>>>>> repository-revision-number
```

- For each conflict marker
  - choose the changes to keep or create a combination of them
  - delete the conflict markers
- Commit the new version
  - could get more conflicts if someone else committed changes to the file while you were resolving conflicts

Determining Differences

- Command
  `> cvs diff [files]`

  usually one file is specified

  `di` and `dif` are synonyms
- By default, between working copy and latest repository revision
- Common options
  - `-c`
    - to show differences within a **context** of three preceding and following lines
  - `-r revision-number|tag-name`
    - to see difference between working copy and a specified repository revision
    - can have two of these to compare two repository revisions
- Context diff output (`-c`)
  - lines preceded by + have been added
  - lines preceded by - have been removed
  - lines preceded by ! differ

Use **`cvs rdiff`** to find differences between repository revisions when no working copy is available. Synonyms are `pa` and `patch`. `rdiff` can also be used to create “patch files” for updating released files without releasing entire new copies.

`cvs diff` Example

```
C:\Volkmann\CVS\SecSvc>cvs diff TestGUI.java
Index: TestGUI.java
===================================================================
RCS file: C:\Volkmann\CVS\repository/SecSvc/TestGUI.java,v
retrieving revision 1.2
diff -r1.2 TestGUI.java
7a8  <-- added a line between lines 7 and 8
> // Added a second comment to test CVS.
```

`cvs diff -c` Example

```
C:\Volkmann\CVS\SecSvc>cvs diff -c TestGUI.java
Index: TestGUI.java
===================================================================
RCS file: C:\Volkmann\CVS\repository/SecSvc/TestGUI.java,v
retrieving revision 1.2
diff -c -r1.2 TestGUI.java
*** TestGUI.java        2000/04/12 17:51:48     1.2
--- TestGUI.java        2000/04/12 17:55:47
***************
*** 5,10 ****    <- lines 5 to 10 in repository revision 1.2
--- 5,11 ----    <- lines 5 to 11 in working copy
  import javax.swing.*;

  // Added this comment to test CVS.
+ // Added a second comment to test CVS.    <- an added line shown in context
  public class TestGUI extends JFrame {

      private static final String HOST = "hqecomdu4"; // "hqistgu3";
```

Getting Status of Working Copy Files

- Command
  - `> cvs status [files]` `st` and `stat` are synonyms
- Status of each file
  - "Up-to-date" means the working copy is identical to the latest version in the repository OR the revision associated with the sticky tag
  - "Locally Modified" means the working copy has been modified by someone else
  - "Needs Patch" means changes have been committed in the repository and the working copy needs to be updated
- Sticky tag
  - if a file was obtained using `cvs update` or `cvs checkout` together with the `-r` or `-D` command options (gets specific tagged revisions), it will have a sticky tag
  - prevents updates from revisions newer than the tag from being merged with the working copy
  - use `cvs update -A` to remove sticky tags from working copy

`cvs status` Example

```
C:\Volkmann\CVS\SecSvc>cvs status TestGUI.java
===================================================================
File: TestGUI.java      Status: Locally Modified

   Working revision:    1.2     Wed Apr 12 17:51:25 2000
   Repository revision: 1.2     C:\Volkmann\CVS\repository/SecSvc/TestGUI.java,v

   Sticky Tag:          (none)
   Sticky Date:         (none)
   Sticky Options:      (none)
```

Viewing Change History

- Command
  `> cvs log [files]`

  `log` and `rlog` are synonyms
- Shows information about each revision of each file including
  - revision number
  - date/time of commit
  - author
  - log message
- Common options

  `-w[users]`
    - to show information about revisions committed by given users
    - if *`users`* is omitted then the current user is assumed
    - multiple users are separated by commas with no spaces

`cvs log` Example

```
C:\Volkmann\CVS\SecSvc>cvs log TestGUI.java

RCS file: C:\Volkmann\CVS\repository/SecSvc/TestGUI.java,v
Working file: TestGUI.java
head: 1.2
branch:
locks: strict
access list:
symbolic names:
        V1: 1.1.1.1
        AGEdwards: 1.1.1
keyword substitution: kv
total revisions: 3;     selected revisions: 3
description:
----------------------------
revision 1.2
date: 2000/04/12 17:51:48;  author: VOLKMAMM;  state: Exp;  lines: +1 -0
added comments
----------------------------
revision 1.1
date: 2000/04/12 17:48:09;  author: VOLKMAMM;  state: Exp;
branches:  1.1.1;
Initial revision
----------------------------
revision 1.1.1.1
date: 2000/04/12 17:48:09;  author: VOLKMAMM;  state: Exp;  lines: +0 -0
initial import
=============================================================================
```

Viewing Project History

- Command
  `> cvs history`
  `hi` and `his` are synonyms
- Lists all uses of these commands
  - checkout, update, commit, tag, rtag and release
- Each line contains
  - code date username revision file-name repository-info
- Common options
  - `-e` to show uses of all commands above (defaults to only showing checkouts)
  - `-x codes` to show only the indicated commands
  - `-a` to show commands issued by all users (defaults to only showing current user)
  - `-u` *`user`* to show only commands issued by a given user
  - `-l` to show only the last time a user issued the command

Codes

```
O - checkout
T - tag
F - release
W - update to remove
U - update, no user changes
G - update, user changes
C - update, conflicts
M - commit, modified file
A - commit, added file
R - commit, removed file
E - export (for packaging)
```

`cvs history` Example

```
C:\Volkmann\CVS\SecSvc>**cvs history**
**O** 04/12 17:49 +0000 VOLKMAMM SecSvc =SecSvc= ~Volkmann\CVS/*

C:\Volkmann\CVS\SecSvc>**cvs history -a -e**
**O** 04/12 17:48 +0000 VOLKMAMM SecSvc =SecSvc= ~Volkmann\CVS/*
**O** 04/12 17:49 +0000 VOLKMAMM SecSvc =SecSvc= ~Volkmann\CVS/*
**M** 04/12 17:51 +0000 VOLKMAMM 1.2 TestGUI.java SecSvc == ~Volkmann\CVS\SecSvc
```

defaults to only showing checkouts by the current user

mapping of project name to repository directory

Viewing Line-by-Line File Changes

- Command
  - `> cvs annotate [`*`files`*`]`
  - `ann` is a synonym
  - – usually run on a single file
- For each line in each file this lists
  - – the revision number of the last change to the line
    - defaults to showing only trunk changes, not branch changes
  - – the user who changed the line
  - – the text of the line
- Example output line
  - – `1.19 (VOLKMAMM 16-Apr-00): import java.util.*;`
- Common options
  - `-r` *`revision-number`* to see only changes up to that; can also use a tag name
  - `-D date` to see only changes on or before then

`cvs annotate` Example

```
C:\Volkmann\CVS\SecSvc>cvs annotate TestGUI.java
Annotations for TestGUI.java
***************
1.1          (VOLKMAMM 12-Apr-00): import com.agedwards.security.*;
1.1          (VOLKMAMM 12-Apr-00): import java.awt.*;
1.1          (VOLKMAMM 12-Apr-00): import java.awt.event.*;
1.1          (VOLKMAMM 12-Apr-00): import javax.naming.NamingException;
1.1          (VOLKMAMM 12-Apr-00): import javax.swing.*;
1.1          (VOLKMAMM 12-Apr-00):
1.2          (VOLKMAMM 12-Apr-00): // Added this comment to test CVS.
1.1          (VOLKMAMM 12-Apr-00): public class TestGUI extends JFrame {
```

remaining output omitted

Reverting Changes

- If the latest revision is deleted from the repository there will be no history of it
- A better alternative
  - retrieve an old revision and recommit it as the newest revision
  - one way to do this

    ```
    > cvs -Q update -p -r revision-number file > file
    > cvs commit -m "reverted to revision-number" file
    ```
  - explanation
    - `-p` causes `update` to send its output to stdout
    - `-r` retrieves a given revision
    - > redirects this output to the working copy of the file
    - the selected revision is then committed to the repository a new revision

Adding Files and Directories

- Command
  ```
  > cvs add files|directories
  ```
  `ad` and `new` are synonyms
- Used to add files to a project that has already be imported
- Use `cvs import` to add an entire directory of files
- When a directory is added, the files in the directory are not automatically added
  - must specify the files to be added
  - `cvs add` doesn't recursively descend subdirectories
- Adding files is actually a two-step process
  - must commit the files after adding them
  - not necessary for directories
- Common options
  ```
  -m "log message"
  ```

Removing Files

- To remove files
  - remove files from the working copy using OS commands
  - execute these commands
    ```
    > cvs remove [files]
    > cvs commit -m "removed" [files]
    ```
    `rm` and `delete` are synonyms
- Places the file and its revision history in the repository Attic directory
  - creates the Attic directory if it doesn't exist
- Creates a new revision with a state of "dead"
  - view state with `cvs status`

Restoring Removed Files

- To restore a file that was removed

```
> cvs -Q update -p -r revision-number file-name > file-name
> cvs add file-name
> cvs commit -m "restored" file-name
```

- This process is identical to reverting changes (discussed earlier)

Removing Directories

- To remove a directory and all subdirectories
  - remove all files from the working copy of the directory and its subdirectories using OS commands
  - execute these commands from within the directory

```
> cvs remove
> cvs commit -m "removed"
> cd ..
> cvs update -P (prunes empty directories)
```

  - CVS removes the working copy directory, not you

Renaming Files

- To rename a file
  - rename the file using OS commands
  - execute these commands

```
> cvs remove old-name
> cvs add new-name
> cvs commit -m "renamed old-name to new-name" \
  old-name new-name
```

- If only the case of the file is being changed
  - on the cvs server (can login using telnet)
    - cd to the repository directory containing the file
    - rename the `,v` file
  - on the local hard drive
    - delete the old file
    - delete the reference to the old file from the entries file in the CVS subdirectory
    - do a `cvs update` on the directory

Renaming Directories

- To rename a directory
  - create a directory with the new name in the working copy

```
> cvs add new-dir-name
```

  - copy all files from the old directory to the new one
  - remove all files from the old directory
  - from the new directory, execute these commands

```
> cvs add
> cd ..
> cvs commit -m "moved from old-dir-name to new-dir-name"
> cvs update -P (prunes empty directories)
```

Default Options

- Commonly used options can be used by default
- Create or edit the `.cvsrc` file in your home directory
- Each line in this file contains a CVS command followed by the default command options to be used
- To specify global options, use “`cvs`” as the command
- Example `.cvsrc` file

`checkout -P`	prune empty directories
`cvs -q`	quiet
`diff -c`	context diff
`update -P`	prune empty directories

Tagging Files

- Tagging all the files in a project makes it possible to retrieve revisions of the files as they were when they were tagged
- Tags are recorded in the repository
- Commands
  - `> cvs tag` *`tag-name`*   `ta` and `freeze` are synonyms
    - applies tag to revision numbers in working copy
  - `> cvs rtag` *`tag-name`*   `rt` and `rfreeze` are synonyms
    - applies tags to newest revision numbers in repository

  delete tags with `-d` command option
- Tag names
  - must start with a letter
  - can contain letters, digits, hyphens and underscores
- Tagging doesn’t modify files, it just associates the tag name with certain file revisions
  - note that the tagged files typically have different revision numbers

Retrieving Files By Tag

- To replace all files in your working copy with file revisions that have a given tag name

```
> cvs update -r tag-name
```

  - all the files will now be “sticky”
    - can’t update to more recent revisions
    - can’t commit changes until sticky tag is removed (`cvs update -A`) and all repository changes are merged into working copy (`cvs update`)
- To retrieve file revisions with a given tag name into directory other than the working copy
  - create the directory and cd to it

```
> cvs checkout -r tag-name project-name
```

  - useful when there are uncommitted changes in the working copy or you wish to avoid making the working copy files “sticky”

Branches

- Branches allow parallel development
- Useful for bug fixes in released software and experimental coding
- Typical scenario
  - project is tested and ready for production release
  - all files are tagged with “*`proj-name`*`-v1`”
  - project is released to production
  - development of features for next release proceeds
  - some changes are committed
  - bugs are discovered in production release
  - a branch from tag “*`proj-name`*`-v1`” is created called “*`proj-name`*`-v1-fix1`”
  - changes are made to files on this branch
  - changes are tested and ready for release to production
  - all files on the branch are tagged with “*`proj-name`*`-v1-fix1`”
  - files on branch are used to build bug-fix release
  - changes are merged back to “trunk” (composed of the non-branch file revisions)

Creating Branches

note that quotes are not required around option values on this page

- Steps
  - if branch is to be created from previous revisions, retrieve those revisions
    - `> cvs update -r` *`tag-name`*
      - useful when the working copy revisions are not needed
    - `> cvs checkout -d` *`branch-directory`* `-r` *`tag-name`*
      - useful when there are uncommitted changes in the working copy or you wish to avoid making the working copy files "sticky"
      - remember to cd to *`branch-directory`* to work on the branch files
  - mark the retrieved files as a branch
    - `> cvs tag` **`-b`** *`branch-name`*
    - the `-b` option causes a branch to be created
    - applies the branch name to all files in and below the current directory
- `cvs status`
  - indicates whether file revisions in working copy are from a branch
- Branches can be created from branches but this is rarely needed

Commiting Changes to a Branch

- After creating a branch, commits occur on the branch until the branch is merged back to the trunk
- Branch file revision numbers have four pieces
  - first is the major revision number from the trunk revision
  - second is the minor revision number from the trunk revision
  - third is the branch number (always even)
  - fourth is the branch file revision
  - example
    - if revision number is `1.19` before the branch is created, the first commit after creating the branch will be `1.19`**`.2.1`**
    - if a previous branch had been created, the first commit will be `1.19`**`.4.1`**

The branch number can be used to get the latest revision of a file on the branch.
For example,
`> cvs update -r 1.19`**`.2`** *`file`*
Note that the second number can be different for each file on the branch.

Merging Branch Changes Into Trunk

- **IMPORTANT:** If there is any chance that additional changes will be made later on the branch, tag the branch before merging it
  - necessary to merge branch into trunk multiple times (see next page)
- Three steps to merge
  - 1 if the branch was created in the same directory as the trunk working copy
    - stop using branch revisions (remove sticky tags),
      revert to trunk revisions and
      merge changes other developers may have made to trunk
      `> cvs update -A` (performs all three actions)
  - else
    - cd to top directory of trunk working copy
  - 2 join (merge) branch changes with trunk revisions in working copy
    ```
    > cvs update -j branch-name
    ```
  - 3 commit working copy
    ```
    > cvs commit -m "merged from branch branch-name"
    ```

Merging Additional Branch Changes Into Trunk

- Only the join step is different
  ```
  > cvs update -j current-branch-tag-name
                -j previous-branch-tag-name
  ```
  - merges changes in files tagged with *current-branch-tag-name*
    made since revisions tagged with *previous-branch-tag-name*
- Example
  ```
  > cvs update -j proj-name-v1-fix2 -j proj-name-v1-fix1
  ```

Branching Tips

- Minimize number of active branches to reduce likelihood of conflicts
- Avoid branching from branches
- Choose a naming convention for
  - tagging releases
  - tagging branches
  - tagging after merging branches back to trunk
- Don't work on a branch too long without merging changes back to trunk
  - reduces the number of changes made on the trunk that will have to be merged with branch revisions before they can be be committed onto trunk
- After merging changes back to trunk, consider creating a new branch for additional changes instead of continuing work on the existing branch
  - same rationale as previous tip

"Sticky" Properties

- Three types: sticky tag, sticky date and sticky options
- Values are displayed by `cvs status`
- Sticky Tag
  - set when `cvs update` or `cvs checkout` is used with the `-r` *`rev`* command option
  - *`rev`* can be a revision number, a tag name, or a branch name
  - can't update to a newer revision
  - when a tag or branch name is used, that name is associated with a specific revision for each file
- Sticky Date
  - set when `cvs update` or `cvs checkout` is used with the `-D` *`date`* command option
  - can't update to revisions newer than this date
  - stored in Greenwich mean time (GMT), not local time

“Sticky” Properties (Cont’d)

- Sticky Options
  - set when `cvs update` or `cvs checkout` is used with the `-k` *`mode`* command option
  - affects RCS keyword substitution
  - common modes
    - `kv` - substitutes the keyword and the value (ex. $Revision: 1.19$)
    - `k` - substitutes just the keyword (ex. $Revision$)
    - `v` - substitutes just the value (ex. 1.19)
    - `b` - suppresses the keyword and the value
      - good for binary files
      - also prevents line end conversions
- Clear all sticky properties using `cvs update -A`
  - also updates files to latest repository revision
- It is common for all files in a directory of the working copy to have the same sticky properties but this is not necessary

RCS Keywords

- Strings that are expanded to meaningful information when files are checked out or updated
- Surrounded by `$` characters
- Normally used in commented source lines
- Commonly used RCS keywords
  - `$Author$` - user who committed current revision
  - `$Date$` - date of commit
  - `$Id$` - filename, revision, date and author
  - `$Revision$` - revision number
  - `$RCSfile$` - repository filename (includes `,v`)
  - `$Source$` - path to repository file including filename
  - `$Log$` - all log messages for the file

Handling Binary Files

- CVS was not designed to maintain a change history for binary files
- By default CVS performs these operations on all files
  - expands RCS keyword strings
  - converts line endings to the type needed for the current platform
    - UNIX, Windows or Macintosh
- These operations should be turned off for binary files
- Two ways to do this
  - on add command
    ```
    > cvs add -kb file
    ```
  - in cvswrappers file
    - specifies special file handling based on file names
    - contains lines like this
      - `*.gif -k b`
      - says that all .gif files are binary which prevents the operations described above

Handling Binary Files (Cont'd)

- Concurrent changes to binary files cannot be merged
- To minimize manual merging, lock binary files before editing
- To lock a file
  ```
  > cvs admin -l file
  ```
- To unlock a file
  ```
  > cvs admin -u file
  ```
- Still have to use `cvs edit` to make file writable
- Still must commit changes
  - committing also unlocks so only have to unlock with `admin -u` when changes will not be committed
- Other users are prevented from locking the file if it is already locked
- They are not prevented from editing the file but they are prevented from committing changes

Watches

- Watches enable email notification of three types of events for files users have asked to watch
  - `> cvs` **`edit`** *`file-name`*
    - notifies others that you intend to edit the file
  - `> cvs` **`unedit`** *`file-name`*
    - notifies others that you are no longer editing the file and do not intend to commit changes
    - reverts changes in your working copy
  - `> cvs` **`commit`** *`file-name`*
    - notifies others that changes have been committed
- You are not notified of your own events

Email notifications are **not currently being sent** at A.G. Edwards due to security restrictions on using sendmail.

Enabling Watches

the CVS administrator may have already done this

- To enable watches
  - checkout CVSROOT/notify from the repository
  - uncomment the following line
    - `#ALL mail %s -s "CVS notification"`
  - commit this change
- User email addresses may need to be added
  - stored in CVSROOT/users
  - format of each line is
    - *`cvs-username`*`:`*`email-address`*
  - modify and commit changes

Encouraging Use of “`cvs edit`”

- Making working copy files read-only serves as a reminder to use “`cvs edit`” before editing files
  - setup
    - run “`cvs watch on`” from the top of your working copy
    - make all files in the working copy read-only
  - impact of using “`cvs edit`” after this
    - makes file writeable
    - sends email notification to watchers
  - impact of using “`cvs commit`” after this
    - commits changes as usual
    - changes file back to read-only
    - sends email notification to watchers
  - impact of using “`cvs update`” after this
    - updates as usual
    - doesn’t change writable status

Can get around this by manually changing file permissions. **DON’T DO THIS!**

All developers should do this from their own working copy.

Marking Files to be Watched

- Command

```
> cvs watch add|remove [files]
```

  - omit list of files to add or remove watches for all files in the current directory and below
- This adds you as a “permanent watcher”
  - will continue watching until explicitly removed
- For files you are not permanently watching
  - `cvs edit` adds you as a “temporary watcher”
  - `cvs commit` and `cvs unedit` remove the temporary watch

Who’s editing? Who’s watching?

- To determine who is editing a file
  ```
  > cvs editors [files]
  ```
- To determine who is watching a file
  ```
  > cvs watchers [files]
  ```
  – for each file/user combination, indicates which events are being watched
    - don’t have to watch `edit`, `unedit` and `commit`
    - **`t`**`edit`, **`t`**`unedit` and **`t`**`commit` indicate temporary watches
- Omit file names to list editors or watchers
  for all files in the current directory and below

Troubleshooting

- Most problems are caused by
  – inconsistent working copies
  – bad repository file permissions
- To determine if the problem is in the repository or a working copy,
  see if other developers are having the same problem with the same files
  – if so, suspect a repository problem

Getting Help

- Online CVS Manual
  - the "Cederqvist" (after the original author of CVS, Per Cederqvist)
  - see http://durak.org/cvswebsites/doc/ and http://www.loria.fr/~molli/cvs-index.html
- UNIX-specific help
  - info cvs
  - man
- To get a list of CVS commands

```
> cvs --help-commands
```

- To get help on a given command

```
> cvs -H command-name
```

- To get a list of command synonyms (abbreviated names for commands)

```
> cvs --help-synonyms
```

Windows Installation

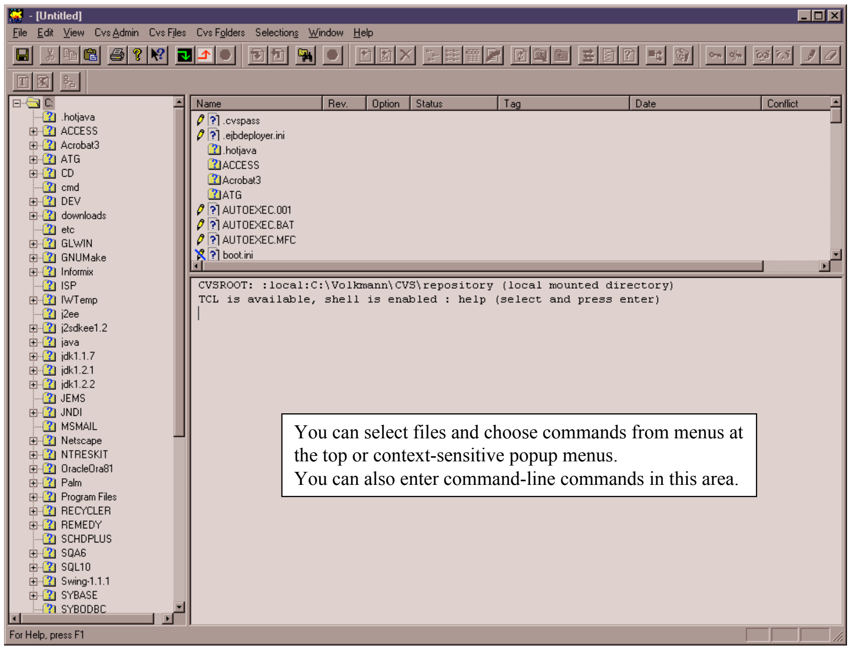
- Download from internet or shared drive (WinCvs includes this)
  - internet
    - http://download.cyclic.com/pub/cvs-1.10/windows/cvs-1.10-win.zip
  - shared drive
    - L:\Ageconnect Development\Software\Install\CVS\cvs-1.10-win.zip
- Unzip
  - double-click the zip file to launch WinZip and extract the contents
- Install
  - place cvs.exe in a directory listed in the PATH environment variable or add the current directory to PATH
- To determine CVS version

```
> cvs -version
```

WinCvs

- Download from internet or shared drive
  - internet
    - http://www.wincvs.org/
  - shared drive
    - double-click L:\Ageconnect Development\Software\Install\CVS\WinCvs\setup.exe
- WinCvs includes CVS so a separate download and install is not required
  - by default, cvs.exe will be installed in C:\Program Files\GNU\WinCvs
  - add this directory to PATH environment variable to use from command line
- Can use command-line commands within WinCvs
  - see large window in the lower-right corner of screen shot on next page
- Don't name the directory containing your working copy CVS!
  - WinCvs won't let you access files in directories with this name
- These notes apply to version WinCvs 1.0.6

WinCvs

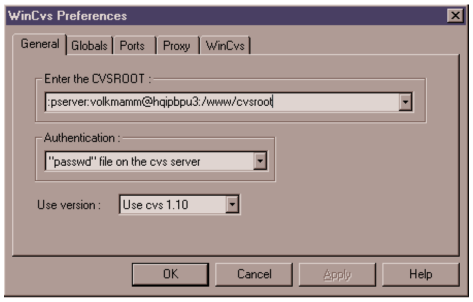


```
CVSROOT: :local:C:\Volkmann\CVS\repository (local mounted directory)
TCL is available, shell is enabled : help (select and press enter)
```

You can select files and choose commands from menus at the top or context-sensitive popup menus.
You can also enter command-line commands in this area.

WinCvs Configuration

- Configuration steps
  - select `Preferences`... from the `Cvs Admin` menu
  - enter the following for CVSROOT
    - `:pserver:`*`username`*`@hqipbpu3:/www/cvsroot`
  - change Authentication to
    - `"passwd" file on the cvs server`
  - click “OK” button

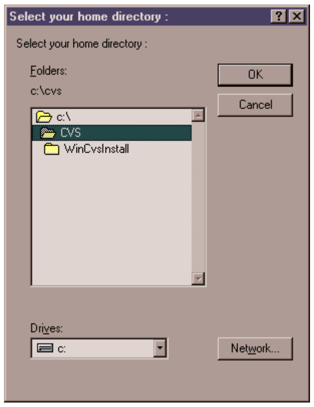


Enter the CVSROOT : :pserver:volkmamm@hqipbpu3:/www/cvsroot

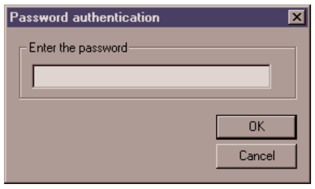
Authentication : "passwd" file on the cvs server

Use version : Use cvs 1.10

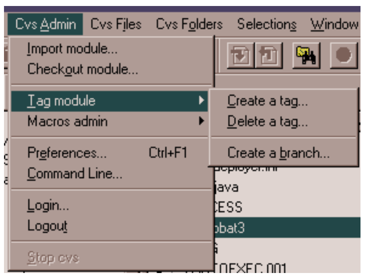
WinCvs Login

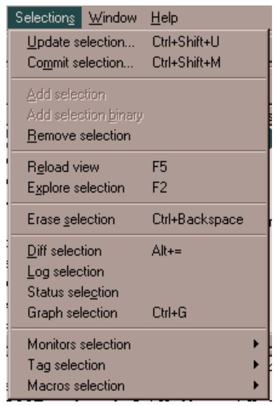


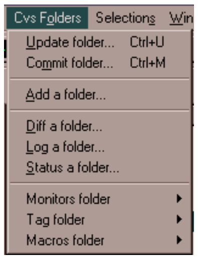
- Login steps
  - select `Login`... from the `Cvs Admin` menu
  - if this is the first time you’ve logged in you’ll be prompted to select the directory where your working copy will be stored
  - next you’ll be prompted to enter your password

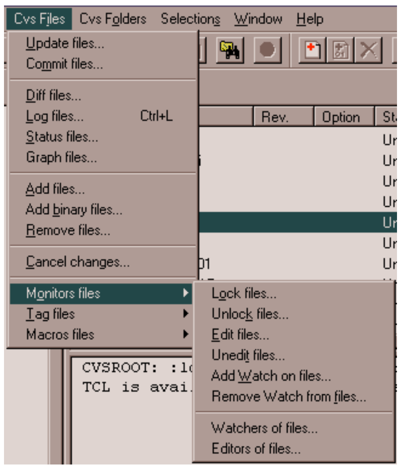


WinCvs Menu Commands









AGe-Connect CVS Commenting Standards

- CVS provides the ability to include a brief comment at the time a file (or group of files) are committed to the repository
- The AGe-Connect project requires that a meaningful string be entered for comments
- All files related to implementation of a feature or a defect fix shall be commented with an identical string
  - at the developer's discretion, tags may also be used to distinguish file revisions with changes for specific features or defect fixes
- If a unique identifier has been established for the feature or defect, the comment shall utilize this as a prefix with a colon (:) as a separator character from the remainder of the comment text
- In general, comment text shall not exceed 60 characters in length
  - for example, "`Defect_453: Fixed NullPointerException`"

AGe-Connect CVS Tagging Standards

- Always tag trunk files before creating branches
- Always tag branch files before merging onto trunk
- Always tag trunk files after merging from branches
- When tagging, tag all files in the repository,
  not just files for a specific track
- Each track is responsible for tagging their files for inclusion in the Milestone at the time of each Milestone integration
- The format of this tag shall be "`MILESTONE_###`"
  - for example, `MILESTONE_001`
- Defect fixes to files included in the milestone shall be implemented on a separate branch named `MILESTONE_###_FIX`
- All fixes by all tracks for the Milestone shall be performed on that branch

AGe-Connect CVS Tagging Standards (Cont'd)

- Development will take place on the main CVS branch except in special circumstances
  - for example, when a developer is required to investigate impacts of new tool versions
- Each track will place the "STABLE" tag on their code periodically to establish a set of files for that track which compile, pass unit tests, and pass track-level integration tests
  - to do this, commit all changes and tag all the files belonging to the track
    - cd to the track directory

```
> cvs commit -m "log message"
> cvs tag -F STABLE
```

`-F` forces reassignment of the tag name when it already exists on another revision

  - placement of a MILESTONE tag should immediately follow the placement of a STABLE tag

AGe-Connect CVS Tagging Standards (Cont’d)

- Developers in each track will initially create their working copy using the following commands

```
> cvs update -r STABLE isp/ageconnect  ← top repository directory
> cd src/com/agedwards/ageconnect/<trackname>
> cvs update -A
```

- This will establish “sticky” files for other tracks while allowing updates to files for their track
- Developers can periodically obtain STABLE updates from other tracks (and the latest versions of their track) by performing a “`cvs update`” at the top of their working copy