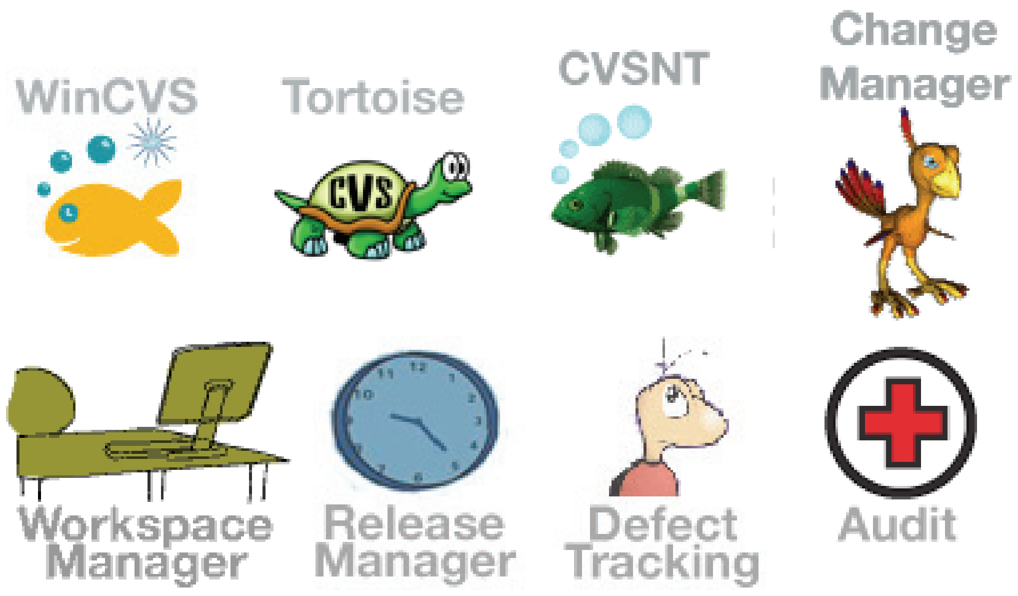
Workflow, Defect Tracking and Email

CVS SUITE AND CM SUITE **2009** Build 3701 **February 2010**



Optimized for Microsoft Visual Studio

March Hare Software Ltd

Legal Notices

march-hare.com

sales@march-hare.com

Table of Contents

Part I – Theory

Promotion model

A promotional model for managing changing documents or software is very common in many organizations. Using a promotion model it is easy to ensure that the correct people only "see" the documents and objects that are at the appropriate level of the promotion process.

Example 1

A government department is required to draft a new piece of legislation. The document evolves using a clearly defined model:

- Draft
- Legal Review
- Ministerial Approval
- Parliament

The document may go through several revisions during this entire process, however each time it is *promoted* to the next level it cannot be changed at that level except by authorised people. For example the public servant who authors the document cannot change the *Legal review* copy.

Frequently the document is only ever changed at the lowest level of the hierarchy, however meta data may be added at different levels. For example the legal review may wish to tag certain paragraphs as needing changes, or they may make the changes themselves.

It is important that the department that prints the legislation to table before parliament cannot accidentally print a copy that has not been through the entire promotion process.

Example 2

A software development company releases software four times a year. The software for each release evolves using a clearly defined model:

- Development
- Review
- Test
- Integration Test
- Production

The software goes through several revisions during this entire process, however each time it is *promoted* to the next level it cannot be changed at that level except by authorised people. For example the programmer who authors the bug fix cannot change the *Test* copy.

The software is only ever changed at the lowest level of the hierarchy, however meta data may be added at different levels. For example the code review may wish to tag certain functions as needing changes to comply with company coding standards, or they may make the changes themselves.

It is important that the users only run the Production version of the software and do not accidentally run another version. If a CD is produced and shipped out, the distribution department must have a fail safe way to ensure that they cannot accidentally deploy an untested release.

What are Branches, Magic Branches and Vendor Branches

Branches, Magic Branches and Vendor Branches facilitate very powerful ways of organising the planned development of your software projects. Often software development managers indicate that they do not require branches in their solution. Take the time to carefully read this section since once you understand what branches, magic branches and vendor branches are and what they can facilitate you may find that it is very useful to you.

When are Branches, Magic Branches and Vendor Branches Used

VendorBranches, Magic Branches and Branches are used whenever the evolution of changes to the documents is not sequential.

Example 1

A manager writes the outline of the current product specification for what the business does and gives it to the marketing department so they can develop a "what we do" document for sales people. However the manager has been instructed to also prepare for a new venture in the near future so after sending the document to the marketing department the manager begins to modify it again to bring it up to date with the new plans.

While the manager updates the document, the marketing department begin to change the wording of the document to make it more suitable for a lay audience, adding pictures and changing the formatting. The same document now has *two streams of development*.

Example 2

A software company releases version 1 of their software and immediately begins work on version 2. However the sales department sell the software to a company who discovers that a part of the application has a bug. Version 2 will not be ready for weeks yet, however the customer requires a fix for the broken function much sooner. The software source code now requires *two streams of development*.

Example 3

A freight company uses a software package to track cargo around the country, however the software uses terminology that is different to what the company uses. The names are stored in configuration files that the freight company modified to change the names to more appropriate ones.

However the software company release version 2 with many new features that the freight company want, but it has a new configuration files with a lot of new information.

The configuration files have two independent *streams of evolution*.

Example 4

A web designer manages seven web sites that are identical to each other except for a few files on each. About 90% of the web sites share the same PHP code. The web site has a single stream of development with *multiple variations*.

Example 5

A software vendor manages configuration files for their software as used by the most important 20 customers. The configuration files have a single stream of development with *multiple variations*.

What are the benefits to using Branches, Magic Branches and Vendor Branches

In each of the above examples the organization finds that the documents do not have a sequential evolution of development but there are *more than one stream of development* (or a primary stream and multiple *variations*). CVS uses branches to track how these changes are made.

However using branches can also offer your organization some advantages because CVS can automate some of your business activities.

Making the same changes twice

One of the benefits to using good configuration management tools is that if you need to re-apply textual changes made to a document in one stream of development to another stream then it can be automated.

For instance, in example 2 above the software developer can make the “bug fix” in “version 2” and use automated techniques to re-apply the same change to the “maintenance version”.

In the example 3 the freight company can use a vendor branch to apply the vendors changes to the configuration files that they are using and therefore keep their labels.

Changes to binary files (eg: word documents or pictures) cannot be replicated automatically.

Ensure security

If a document is evolving on more than one stream of development then it is possible that one or more have different security requirements. For instance a software vendor may develop enhancements to their application for two customers who are competitors. In this case it may be necessary for the developers making changes for customer A not to be able to see customer B’s changes.

In example 1 above, the developers from the “version 2” team may not be allowed to make changes to the more stage “version 1”, so it is possible to secure the access of the two groups based on the branch.

Technical limitations on Branches – no deleting or renaming

It is not possible to delete or rename a branch created using CVSNT Server except in special circumstances (eg: the branch was created and never used – no revisions are on the branch).

CM Suite

EVS Server has the ability to track changes to branch properties and attributes (eg: the name) over time, however there is no tool provided in the current release of CM Suite to rename branches stored in EVS Server. If you are upgrading to EVS Server and this feature is critical to successful completion of the upgrade please contact your technical account manager who can assist you obtaining upgraded tools.

Choose branch names carefully – they may not be deleted or changed

Due to these technical limitations we strongly recommend that you choose branch names carefully:

We recommend these guidelines for branch names, e.g.:

– avoid names whose meaning will expire, eg: *stable*

– avoid personal nouns, eg: joeys_branch

– it is good to use numbers which are meaningful to the business, eg: stable_ver_2_0_1

What is different between a Branch and a Magic Branch

Magic branches are used where the documents as a collection exist as several *variations*, however as individual documents the majority are all identical. This is common for configuration files or data (or configuration) driven web sites. Two Active Server Pages web sites may exist that are identical except for a few graphics files and the `locals.inc`, which sets the title for the pages and the name of the database to get the data from.

Magic Branches organise development so that the site-specific configuration can be developed separately to the web site framework. Provided that the web site framework is never modified on the branch the magic branch always will contain the same files as the trunk.

Introduction to CVS Meta Data and Deltas

CVS stores three things for each of your documents that it is managing:

– The latest version
– The history as differences to the latest version
– Meta data about each version d

Meta data that CVS keeps is very valuable and combined with the *latest version* and *differences* are the key to understanding how branching works.

Based on the above information which CVS stores - if there are two streams of evolution for a document there is a basic problem. The first version created is given a *label* 1.1 (in the meta data). The second version is given version label 1.2. However if Person B makes parallel changes to version 1.1 then it cannot be named 1.3 because you would lose the changes between 1.1 and 1.2.

This is not a problem if Person B wanted version 1.3 to contain both their changes and Person A's changes, or person A's changes were a once off that can now be lost.

However in our above examples what we need are two parallel development streams:

Person A 1.1 -> 1.2 -> 1.3

Person B 1.1 -> 1.1.1.1 -> 1.1.1.2 -> 1.1.1.3

In this case person B wants to create a branch (or vice versa).

What creating a branch really does

When Person B creates the branch there are now two "latest versions" stored by CVS in the repository. This allows differences to be kept for both. So there is now a "latest version" for Person B's branch and a "latest version" for Person A. Differences are kept for both

So if Person A decides that they need Person B's changes between 1.1.1.2 and 1.1.1.3 but not the other oncs it is easy to re-apply those using CVS to 1.3 to create version 1.4.

The repsoistory only ever stores one complete copy of the versioned file, and that is the tip of the Trunk. Therefore performing work on branches is never as efficient as on the Trunk – for this reason all of the following development models discussed assume that the majority of changes will be performed on the Trunk, and then migrated as necessary to other branches.

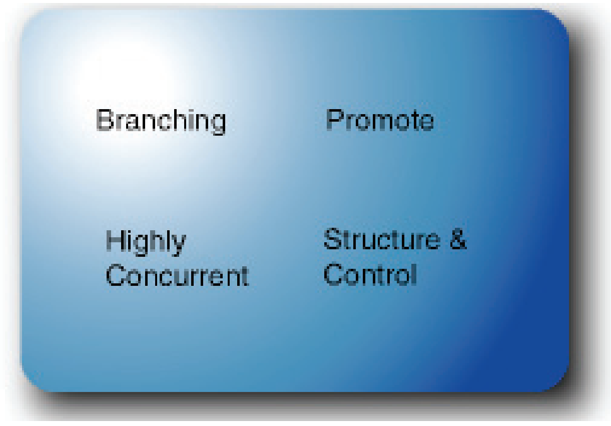
Promotion model versus Branches

A mixture of branches and promotion models may be used, however it is common for organizations to primarily use one or another for a particular project.

Within a single CVS repository it is allowable to use different branches and promotion models on different projects.

Highly concurrent / Highly available vs. Structure and Control

The more your CM process relies on the Promotion Model the greater structure and control, the more your CM process relies on Branching the more highly concurrent your development will become.



A Branching Intensive Development Model

CVS was written originally with the requirements of open source developers in mind. There are several things that are unique about open source development:

- Developers work geographically separated from one another
- Developers work concurrently on the same files but achieve a single unified result
- Development never ceases even during a software release

Consequently CVS has evolved very strong Branching techniques, since this allows for the highly concurrent nature of Open Source development. For commercial software development manager the last point in the above list may be surprising. In Open Source development the software never reaches a stable state to promote to “Code Review” or to “Test”.

To cater for this – the software is branched at an arbitrary point in time and a test build made. Incomplete development is taken off the branch and *build errors* are corrected. When the build process is stable on the branch then testing can begin and volunteer programmers test and fix bugs on the branch. Once the release candidate is stable only then are the changes merged back to the trunk (if at all).

Most commercial software development will not find this model suitable – however some aspects of it may be helpful – in particular the ability to have the staff programmers most productive on developing new features not hindered by the requirement to slow development in preparation for release.

A promotion intensive "secure" model

CM systems that do not support any branching use a promotion intensive model:

- Developers work closely with the build and deployment team
- Only one developer works on a single source at a time and that developer usually is working on only one task
- Development on that object often ceases until it has been released to the customer

Software development using these techniques can be slow, however the releases are stable and management have a great deal of control over what can be delivered to customers at a given point in time.

Mixed model

Mixing up the development models

Most organizations implementing CVS will want to use a mixture of the branching and promotional models. For example:

- Develop Version 1 on the Trunk
- When Version 1 is feature complete Branch Version 1 Maintenance
- Begin work on Version 2 on the Trunk
- Finalise Version 1 on the Branch
- Promote Version 1 Branch to Test
- Fix Version 1 bugs on the Version 1 Branch and promote to test again

Using a mixture of the techniques can lead to a balance with the strengths of both systems.

Regardless of the choices you make CVS is always capable of reapplying the changes between any two revisions to another revision – whether it is on the same branch or a different one.

Patch management – getting fixes to customers

The CM model that you choose may be heavily influenced by business concerns such as needing to deliver fixes to current software while also allowing development to continue on newer versions. This is known as *patch management*.

Service Packs

Often organizations deliver stable combinations of patches to customers as a service pack – or a point release. For example, version 1 is released and several bugs are found and fixed, and three months after the first release version 1.1 (or version 1 service pack 1) is released containing all bug fixes.

This service pack example can also be described in a time line similar to:

- Version 1
- Fix bug 1
- Fix bug 2
- Fix bug 3
- Release Version 1.1

Patches

What happens if one of those bugs seriously effects the customer's current day-to-day operations? In this case it may be necessary to release one of the bug fixes immediately. Since most customers are not affected the release version 1.1 is not created earlier – but a patch is produced.

This patching example can also be described in a time line similar to:

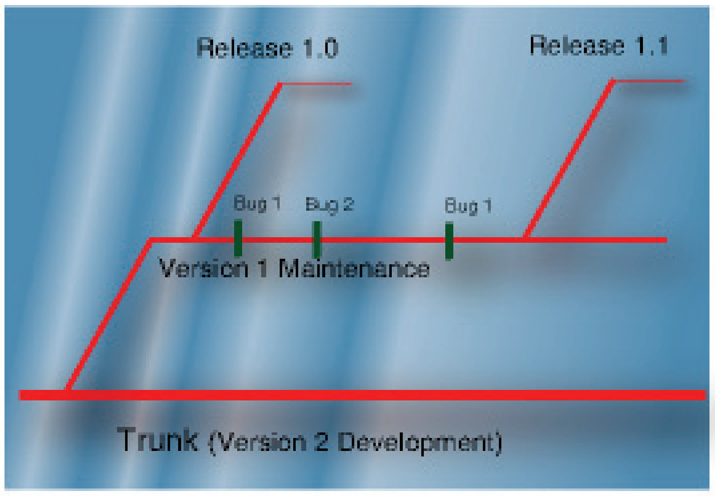
- Version 1
- Fix bug 1
- Fix bug 2
- Release Patch 1
- Fix bug 3
- Release Version 1.1

However the customer does not want any changed functionality *except* their bug fix. In the list above it can be seen that bug fix 1 and bug fix 2 have already been completed and a combined Release Patch 1 contains both fixes. A typical development environment may have made 50 changes, and the customer does not want the responsibility of testing all those fixes to get their environment working again.

The software company needs to balance the customer's requirements with their own. Specifically they need to ensure that the change is only made once but it is then applied to release 1.1 and also later to release 2.0.

Careful consideration of the business requirements is necessary to design an effective CM process. CVS is technically capable of supporting all these decisions.

In this particular example the choices the software organization would make would depend largely on the frequency and the billing methods. If these *individual patch releases* are rare or are charged to the customer then they will be designed as an exception. If they are common then the SCCM solution will be designed with some level of automation for reliability and reproducibility.



File and Directory Architecture

The physical format of the files that you version will affect the CM process.

– File types (Exports; Binary / Unicode / ASCII)
– File naming
– Include Files
– Size / Contents and the relationship to project activity

File Types

CM systems and CM theory largely require the actual file (or thing) being modified to be versioned (managed) by the tools. The most common scenario for a CM system to version something other than the actual file involved is in a database application. The tables themselves (or the meta file) are not versioned, but a script containing the SQL instructions to create that meta data.

If a copy of the thing (or file) is versioned then additional manual procedures must ensure:

– The *actual file* is updated when the CM system is updated
– The CM system is updated when the *actual file* is updated
– If the *actual file* is modified outside of the system and the CM system attempts to overwrite a newer version with an older one that some system prevents this
– If the *actual file* is being modified that a lock or notification is placed on the CM system so other users wanting to (or are) working on that file are informed.

File Naming

Names of file, or its location in a directory tree of files should not contain important information about the nature or function of the contents. The correct place for that information is in the CM system.

For example: `2005/June/common.h` is not a good name for a file since it is likely to need to change often. Instead the file `common/fin.h` can have many branches or many tags to differentiate different financial periods.

Poor filenames can lead to unnecessary refactoring of the repository (ie: name changes).

Include Files and Common Files

Files that are used by several other files (eg: a document template or an include file) should reside in a separate directory.

Some configuration management systems encourage the users to put the common file in the same directory as the file that uses it and have several links to this file.

This can lead to confusion as to the true nature of the file (ie: used by many – not specific to this document or project). Instead store common files in a separate module (or directory) and use branches for different variations used by different projects or users.

Often these common files will also require different security measures – eg: regular developers can change projects, but only administrators can change the `fin.h`. include file. CVSNT has ACL’s that allow repository administrators to set different levels of access for users and groups. These ACL’s are designed to be applied to directories instead of individual files.

Size and contents and the relationship to project activity

Large text or Unicode files in a well designed CM system are rare. If you have a large document or file check that it cannot be better tailored to the project structures you use. These guidelines can help you find if the document or file would be better divided:

– Different sections of the file / document are regularly maintained or modified by different people. Eg: Joe always makes modifications to section A, and Bill always makes changes to the Glossary. In this case the Glossary and each section are most likely candidates for separate versioning. Then Bill and Joe can easily make their changes at the same time as each other.
– The document or file is frequently modified or frequently the subject of contention (several people or projects wanting to modify it at the same time). If the file has smaller divisions this can reduce contention.
– The document or file is frequently modified and then unmodified by the same two or three teams or people. Eg: Team A set the `fin.h` file to `#define YEAR 2005`, then team B change `fin.h` to `#define YEAR 2006` then Team A set the `fin.h` to `#define YEAR 2005`. This indicates that Team A and Team B should be using different branches of the same file (branch “teamA” and branch “teamB”) then the conflicts will not occur.

Notification of what is committed / added

CVSNT contains trigger libraries that can notify you when changes are committed to the repository. You can choose either or both of:

– Email trigger.
– Defect Tracking trigger (Bugzilla, Mantis or Atlassian Jira).

What other choices do you need to make about CM

For Configuration Management to be effective it must be implemented in such a way that is consistent with your corporate culture and is designed to deliver results according to your management objectives. Therefore it is not usually possible for one company to mimic another company’s successful implementation and also achieve a successful implementation.

Management Objectives

In order to evaluate the success of the implementation of CM it is necessary to understand the objectives of management when they agreed to pay the cost. Simply because CVS is free does not imply that there is zero cost to the implementation, and if money/time is being spent then there are always expectations.

Analysis of these objectives early may lead to the conclusion that CVS is not technically capable of delivering on the requirements. In this case either the project must achieve “buy in” from management for different objectives or select an alternative tool that is technically capable of delivering on the requirements.

Company Culture

The following two sections deal with technical architecture choices that a person designing a new CM solution must make. These choices are guided overall by the management objectives, but should then be guided by the company culture. Specifically the designer should avoid using personal preference as a guide, or at least observe if that personal preference is in keeping with the opinion of others in the organization.

The company culture is most apparent in the workflow that a developer follows when they are working on source code. Is it methodical and planned or is it disorganised and unstructured? Are flexible work practices used including job sharing and working from home? Finally what is "the most important thing" – delivery or development? This last issue is often very different for an organization maintaining legacy applications versus a start-up without any current customers.

Reserved / Unreserved – Centralised / De-Centralised

These are the most fundamental choices in designing the workflow that your team will use with a new CM solution.

Reserved / Unreserved

Does development "reserve" a file for editing by a single user, or can multiple users all edit the same version of the same file at once?

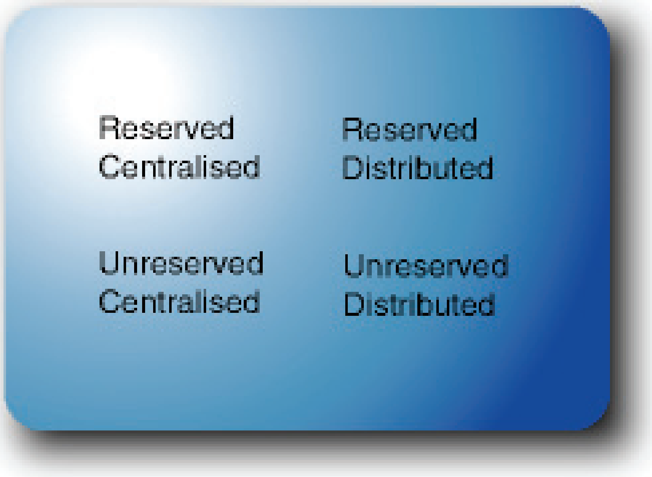
Centralised / De-Centralised

Is a central *reference copy* of the source code always available, or do users work more autonomously on their own *personal copy*?

There is no correct answer to this question. Some CVS and some CM textbooks suggest that one technique or the other is the only right one, whereas the truth is that there is probably only one right answer for your requirements.

Traditionally CVS is assumed to only be capable of delivering *De-Centralised Unreserved* source code management. This is also simply not true. Even older versions of CVS were capable of providing any of the four available choices. CVSNT has been enhanced to make using it in these different ways more reliable and intuitive.

These choices are best understood as a matrix:



Reserved Centralised

A version of the source code for a particular release is stored in a central location. Developers lock a file that they are working on. Other users can see the changes as they are made, but are unable to alter the changes. Users working on other releases do not see the changes. When the Developer finishes the changes the file is unlocked.

Reserved Distributed

Each developer has a version of the source code for a particular release stored locally. Developers lock a file that they are working on and other developers with their own copies cannot make changes to the file in the same release while it is locked. Other workers do not see the changes made to this version until it is returned and they update their local copies.

Unreserved Centralised

A version of the source code for a particular release is stored in a central location. These may be edited at any time by the developers. After files have been changed they may be "rolled back" or "committed". Developers usually work in teams. All developers can see all the changes (committed and uncommitted) all of the time.

Unreserved Distributed

Each developer has a version of the source code for a particular release stored locally. These may be edited at any time by the developers. After files have been changed they may be "rolled back" or "synchronised and committed". Developers usually work individually. Several developers can work on the same file in the same release at the same time. The first developer who "commits" can do so without making any other unrelated changes. Other developers must "synchronise and commit" to add the first developers changes to their own.

Reserved Centralised / Reserved Distributed Mix

A version of the source code for a particular release is stored in a central location. Developers lock a file that they are working on and it is copied into their personal work area. Other workers do not see the changes made to this version until it is returned. Developers may work individually or in teams. As soon as work is returned all other developers see the changes. This model usually requires specific support in the development tools. CVS Suite does not directly support this method, however CVS Professional (Level 2 and Higher) can contact their sales representative regarding this option.

Communication versus Insulation

Each of the abovementioned quadrants in the reserved/unreserved – distributed/centralised matrix offers some degree of Communication and Insulation.

Effective configuration management requires a balance between these two priorities.

Communication

Developer Andrew is working on release two of the software, and Mark is working on an urgent bug fix to the same software for a customer. It is important that the fact that Mark has made changes to the same "base code" is communicated to Andrew – so that the new version is tested to ensure the same problem doesn't exist when that version ships out.

Insulation

Developer Mark needs to make an urgent fix to the Invoice Maintenance screen for a customer – but Andrew is currently re-writing it because it is knows to have a large number of inconsistencies. Mark must be insulated from Andrews's development so that the urgent changes can occur immediately.

Sources / Objects

Every software project is made up of a combination of *source code* and *object code*. Source code generates object code usually using a compiler, e.g.: `hello.c` generates `hello.o` and `hello`; `ADDCUST.XML` generates `addcust.frm`; and `install.doc` generates `install.pdf`.

Object code can always be re-generated as long as the source code is available. The process that generates the objects from the source is usually referred to as the build – and build management tools such as MAKE or ANT can be integrated with your source code controller such as CVS. Build scripts and *makefiles* are also considered source code since the objects cannot be created without them.

Different source code management philosophies disagree on whether you should version only the source code or the object code as well. The truth again is that it depends entirely on your requirements and company culture.

For instance if the process which generates the objects is long, or requires software licences that are not generally available – then storing the objects can be justified as simply convenient. Alternatively it may simply mean an additional complexity that your organization does not require.

Another strong reason to version your objects is that it can make reproducing a "release" of your software much simpler. If you use CVSNT to do your release management (i.e.: your production software runs compiled code checked out of CVSNT) then you can perform an "upgrade" by simply doing a "cvs update". This is akin to the "Windows Update" technology.

Part II – Practical

Installing server integration

To install an integrated server that includes CVSNT, defect tracking integration, auditing and automated builds you will require the March Hare integration DLLs. These DLL’s are a part of the CVS Suite Server installation package available from the customer downloads area.

Before configuring these components you should have successfully completed the section *Installing Bugzilla and MySQL for CVSNT Integrations* and in particular *Installing server components*. That section includes step by step instructions for installing Perl, MySQL and Bugzilla on a Windows 2000/2003 server (XP Professional Workstation can also be used provided that simple networking is switched off and the number of users will not exceed the pre-defined limits of XP).

Integration with Email

When a user commits, tags or begins work on a file (with cvs edit) it is possible to automatically send an e-mail to people who are “watching” that file.

More complex Email integration is possible using CVSMailer a separate (free) product.

Default Behaviour

When a user commits, tags or begins work on a file (with cvs edit) all users . It allows you to put any contents in the emails, but the output format is fairly simple - it is no substitute for a purpose designed notification program.

Email sending is disabled by default. To configure it for use you must do the following.

Configure the commit support files

commit_email, tag_email and notify_email

The commit support files *commit_email*, *tag_email* and *notify_email* contain the names of the template files to use for commit, tag and edit respectively. Each line in these files is a regular expression followed by a filename. The filename is always relative to the CVSROOT directory and may not be an absolute path for security reasons.

The first matching line for each directory committed is used. If there is no match the DEFAULT line is used.

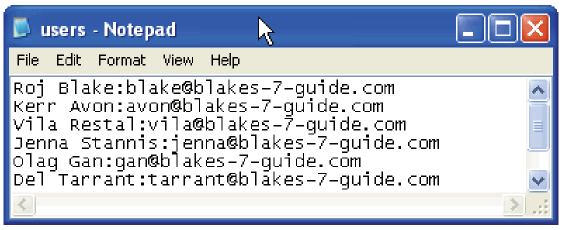
template and checkoutlist

The name of the template file should also be listed in the *checkoutlist* administration file so that it is available for the script to use.

users and checkoutlist

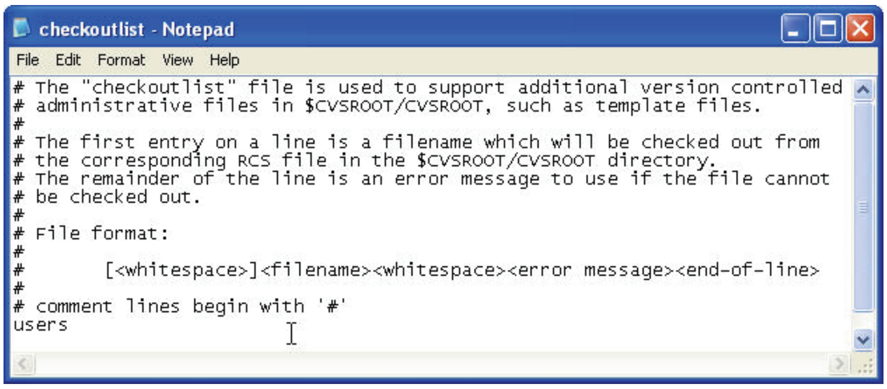
The administration file *users* is used to lookup the username -> email mapping. This file is a list of colon separated username/email pairs. If this file does not exist or the username is not listed the default domain name set in the global configuration is used.

Create a file named users (no file name extension) that will map between users login names (ie: Active Directory names) and their e-mail addresses:



```
Roj Blake:blake@blakes-7-guide.com
Kerr Avon:avon@blakes-7-guide.com
Vila Restal:vila@blakes-7-guide.com
Jenna Stannis:jenna@blakes-7-guide.com
Olag Gan:gan@blakes-7-guide.com
Del Tarrant:tarrant@blakes-7-guide.com
```

Add the users file to the checkoutlist file:



```
# The "checkoutlist" file is used to support additional version controlled
# administrative files in $CVSROOT/CVSROOT, such as template files.
#
# The first entry on a line is a filename which will be checked out from
# the corresponding RCS file in the $CVSROOT/CVSROOT directory.
# The remainder of the line is an error message to use if the file cannot
# be checked out.
#
# File format:
#
#       [<whitespace>]<filename><whitespace><error message><end-of-line>
#
# comment lines begin with '#'
users
```

Write the template

The template file is a text file listing the exact text of the email to send including headers. The To:, From:, Cc: and Bcc: lines are used by the sending software to determine the addresses to use.

An example commit template is:

```
From: [email]
To: cvsnt_users@mycompany.com

Subject: Commit to [module]

CVSROOT:      [repository]
Module name:  [module]
Changes by:   [email]           [date]
On host:      [hostname]

[begin_directory]
Directory: [directory]

[begin_file]
[change_type] [filename]   [tag]   [old_revision] -> [new_revision]
[bugid]
[end_file]
[end_directory]

Log message:
[message]
```

A number of replacements are done on the file to format it for final sending. This differs for each file, and is listed below.

Configure the server

There are two ways that CVSNT can send email. The simplest is to set the SMTP Server and default domain in the global configuration (CVSNT Server Control Panel in Windows, `/etc/cvsnt/PServer` in Unix) and let the internal SMTP client send the emails.

This will not work in the case where authentication is required or the server is not capable of SMTP. In these cases you should set the Email Command. This command should take a list of 'to' addresses as parameters, and a raw email as its standard input.

A suitable configuration for Unix systems is

```
      /usr/sbin/sendmail -i
```

Similar programs exist for Windows.

Integration with Bugzilla, Mantis or Jira Defect Tracking

CVS Suite **2009** Build 3701 is the third release of CVS Suite which March Hare supports linking CVS with Defect Tracking systems like Bugzilla, Mantis and JIRA. If you are upgrading from version 2.0.x please read this section carefully and then follow the upgrade instructions in the appendix.

Supported defect tracking systems and versions

Bugzilla

The server integration is designed to support the Bugzilla schemas:

- 2.18 (which includes Bugzilla 2.17 and 2.20) and
- 2.22 (which includes Bugzilla 2.22, 3.0 and 3.2)

Mantis

The server integration is designed to support the Mantis schemas 1.1 and has been tested with Mantis 1.1.1. Mantis can use any of several databases for data storage however the integration supports only MySQL.

Jira

The server integration is designed to support the Jira schemas 3.12 and has been tested with Jira 3.12.2. Jira can use any of several databases for data storage however the integration supports only MySQL.

A Jira ‘bug number’ typically comprises a project key and a number. You can specify a default project key in the defect tracking integration configuration (eg: PROJ) and then any numbers will automatically be assigned to that key (eg: bug 1 will be ‘PROJ-1’).

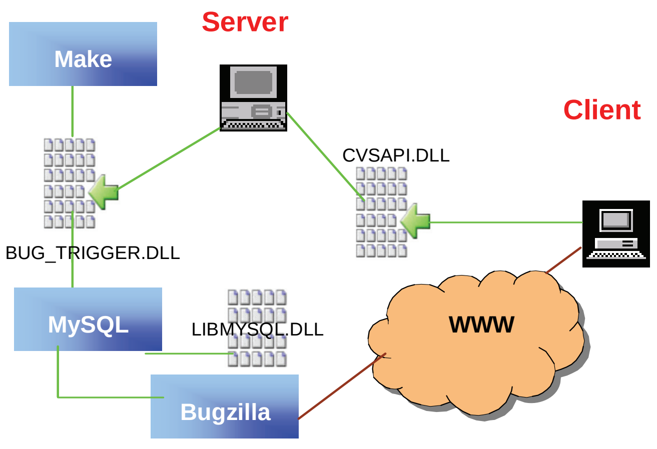
How the Bugzilla Integration Works

Version control without defect tracking may limit the benefits available to an organisation. March Hare have designed CVSNT to integrate seamlessly with defect tracking systems at the server. Furthermore this interface is designed to be transparent to the client user.

This section provides an overview of how the integration between CVSNT and Bugzilla is operated.

Communications

The triggers DLL works in conjunction with the CVSAPI to communicate between the client and server and fire the events to trigger recording of bug information.



Default Behaviour

The trigger DLL will be used if the triggers administrative file is configured to activate it. The trigger is pre programmed with the following behaviour:

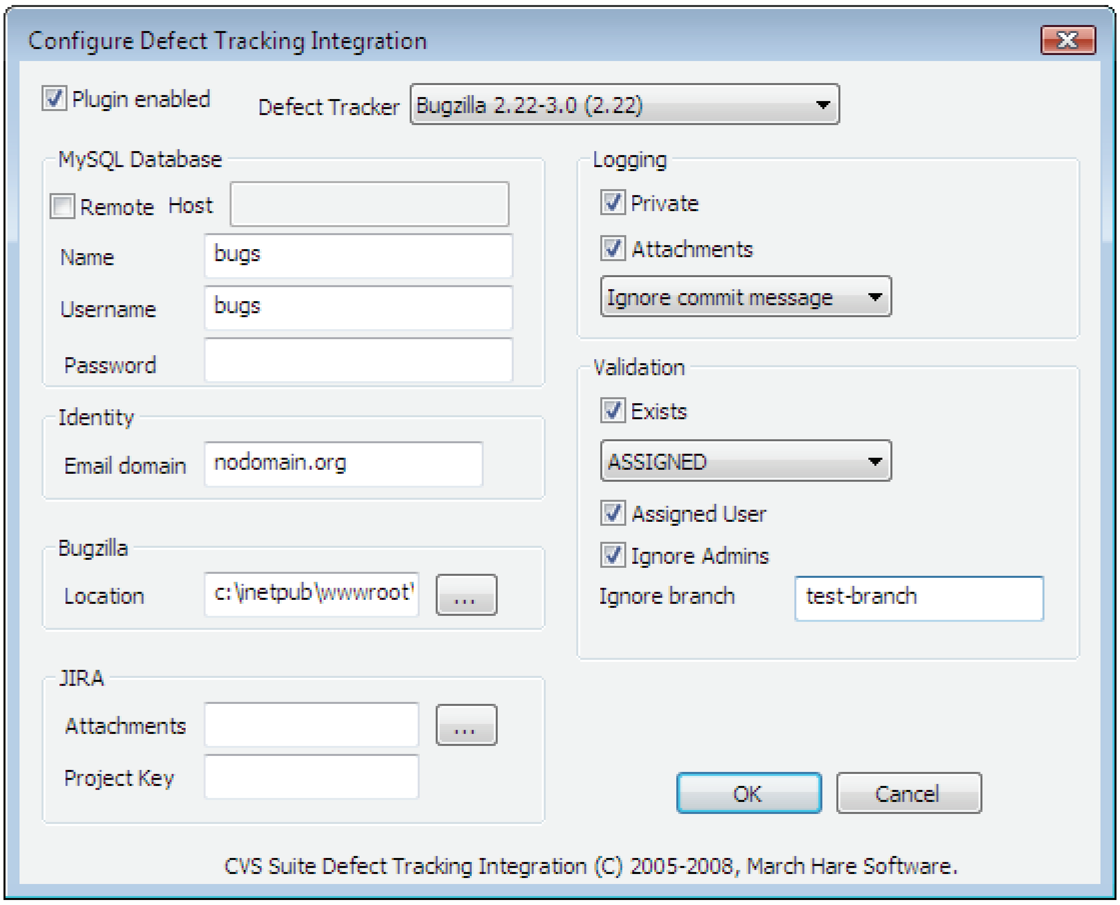
- On completion of a commit with the –B *bugid* switch the bug identified with *bugid* will receive the comment.

Installing Bugzilla on Windows

This section assumes you have already installed a compatible version of Bugzilla such as 2.18, 2.20, 3.0, 3.2 etc. For instructions on setting up Bugzilla on Windows please refer to the section on *Installing Server Components*.

Configuring Integration on Windows

Use the CVSNT Control Panel to configure the plugin. Navigate to the "plugins" tab and select the Bugzilla Integration plugin and press the Configure button.



Configure Defect Tracking Integration

- [✓] Plugin enabled
- Defect Tracker: Bugzilla 2.22-3.0 (2.22)

MySQL Database
- [ ] Remote Host:
- Name: bugs
- Username: bugs
- Password:

Identity
- Email domain: nodomain.org

Bugzilla
- Location: c:\inetpub\wwwroot\

JIRA
- Attachments:
- Project Key:

Logging
- [✓] Private
- [✓] Attachments
- Ignore commit message

Validation
- [✓] Exists
- ASSIGNED
- [✓] Assigned User
- [✓] Ignore Admins
- Ignore branch: test-branch

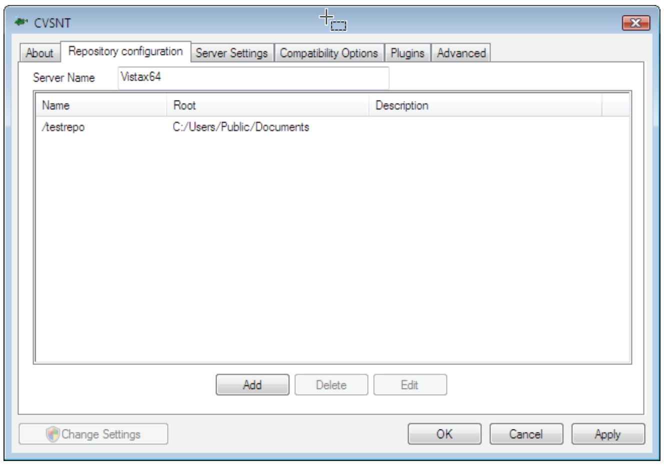
CVS Suite Defect Tracking Integration (C) 2005-2008, March Hare Software.

Configuring on Unix

The Bugzilla integration is configured in the file `/etc/cvsnt/PServer` and is enabled using the file `/etc/cvsnt/Plugins` .

Installation of Integration

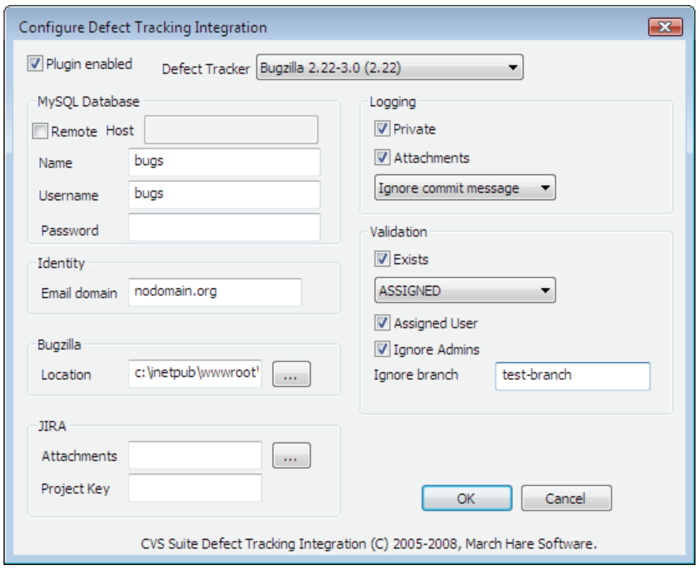
1. If you have not already created a repository, create one using the CVSNT Server windows control panel:



Server Name: Vistax64

Name	Root	Description
/testrepo	C:/Users/Public/Documents	

2. Use the CVSNT Control Panel to configure the plugin. Navigate to the “plugins” tab and select the Bugzilla Integration plugin and press the Configure button.



Configure Defect Tracking Integration

✓ Plugin enabled Defect Tracker: Bugzilla 2.22-3.0 (2.22)

MySQL Database
- ☐ Remote Host:
- Name: bugs
- Username: bugs
- Password:

Identity
- Email domain: nodomain.org

Bugzilla
- Location: c:\inetpub\wwwroot\

JIRA
- Attachments:
- Project Key:

Logging
- ✓ Private
- ✓ Attachments
- Ignore commit message

Validation
- ✓ Exists
- ASSIGNED
- ✓ Assigned User
- ✓ Ignore Admins
- Ignore branch: test-branch

CVS Suite Defect Tracking Integration (C) 2005-2008, March Hare Software.

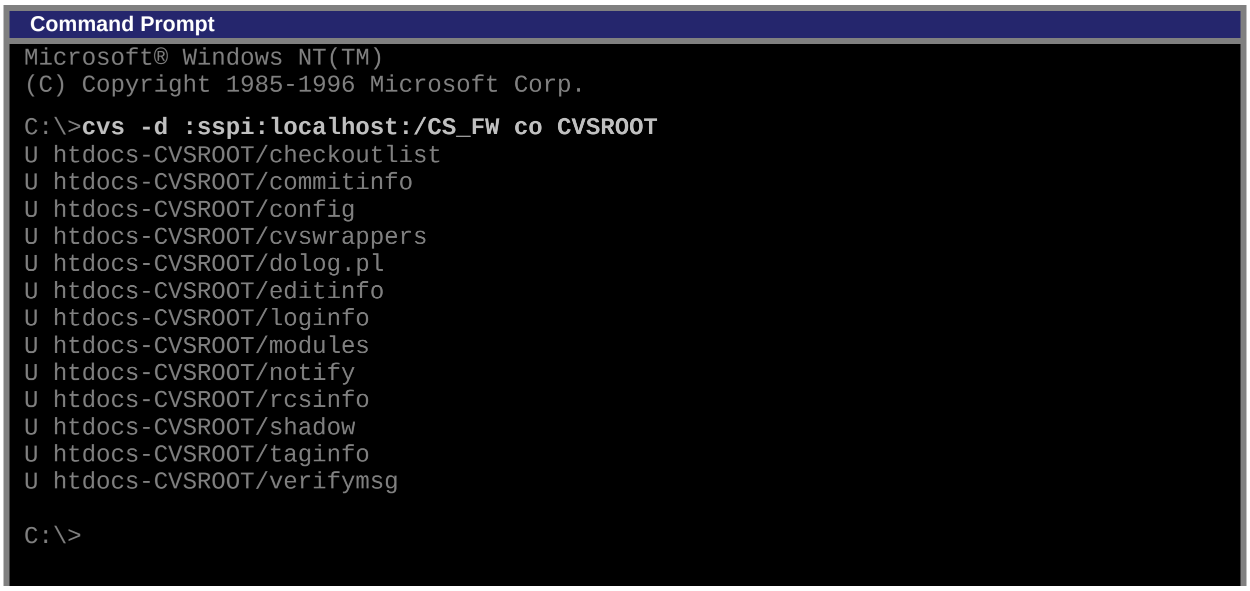
Ensure that the plugin is enabled and enter the following additional information:
- Database Name
- Database User
- Database Password
- Default user domain (for where no translation exists in the CVSROOT/users file)
- Location of Bugzilla (to trigger automatic e-mails)

And choose which options you want enabled:
- Mark commit comments as private (note: automatic e-mails are not sent for private comments)
- Store commit deltas as attachements in patch format

You can choose validation additional options that you want enabled:
- Bug must exist
- Bug must be in the state specified
- Bug must be assigned to user

3. The administration file *users* is used to lookup the username -> email mapping. This file is a list of colon separated username/email pairs. If this file does not exist or the username is not listed the default domain name set in the global configuration is used.

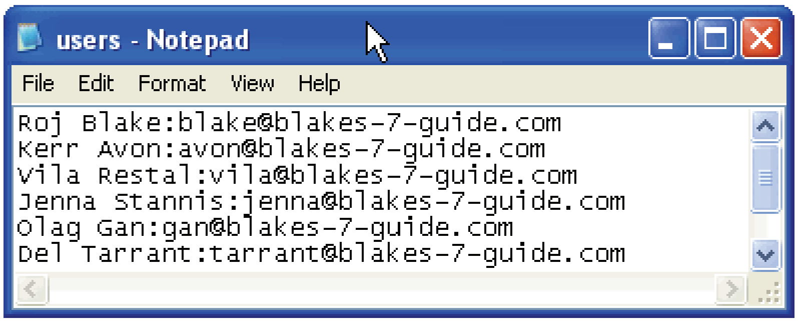
   Check the CVSROOT module:



Command Prompt

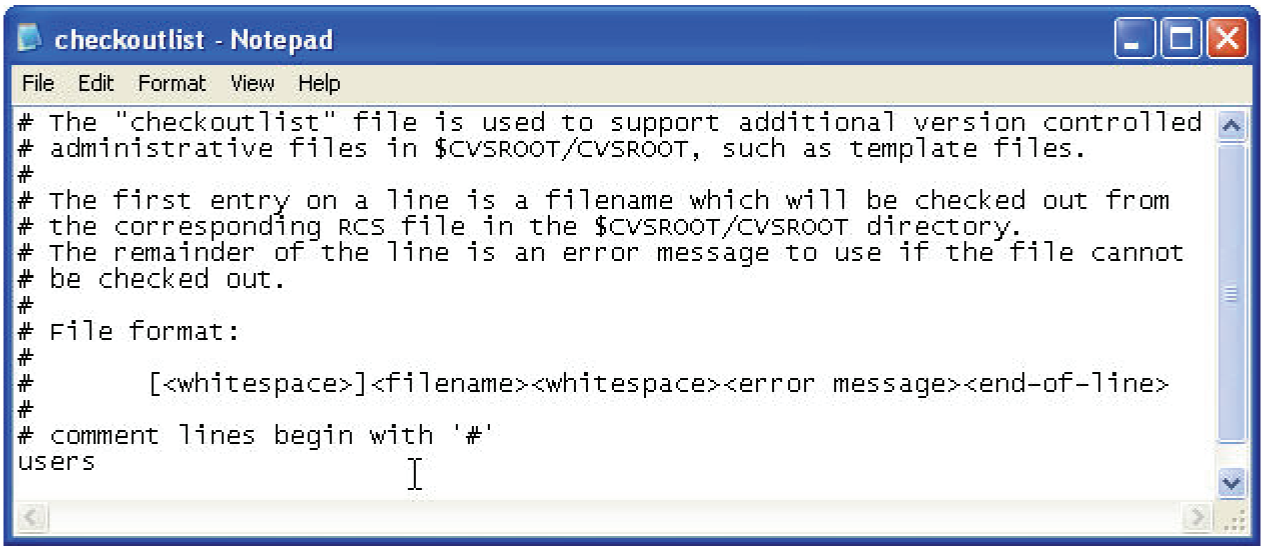
```
Microsoft® Windows NT(TM)
(C) Copyright 1985-1996 Microsoft Corp.

C:\>cvs -d :sspi:localhost:/CS_FW co CVSROOT
U htdocs-CVSROOT/checkoutlist
U htdocs-CVSROOT/commitinfo
U htdocs-CVSROOT/config
U htdocs-CVSROOT/cvswrappers
U htdocs-CVSROOT/dolog.pl
U htdocs-CVSROOT/editinfo
U htdocs-CVSROOT/loginfo
U htdocs-CVSROOT/modules
U htdocs-CVSROOT/notify
U htdocs-CVSROOT/rcsinfo
U htdocs-CVSROOT/shadow
U htdocs-CVSROOT/taginfo
U htdocs-CVSROOT/verifymsg

C:\>
```

4. Create a file named users (no file name extension) that will map between users login names (ie: Active Directory names) and their e-mail addresses:



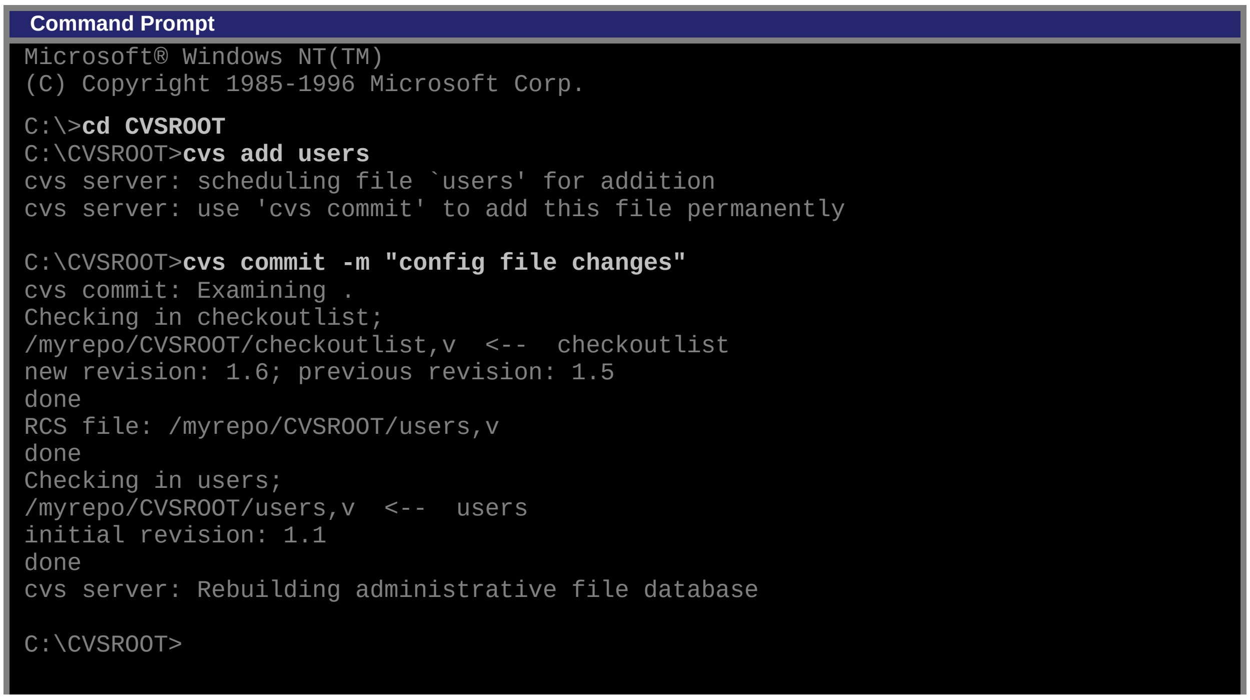
```
Roj Blake:blake@blakes-7-guide.com
Kerr Avon:avon@blakes-7-guide.com
Vila Restal:vila@blakes-7-guide.com
Jenna Stannis:jenna@blakes-7-guide.com
Olag Gan:gan@blakes-7-guide.com
Del Tarrant:tarrant@blakes-7-guide.com
```

5. Add the *users* file to the *checkoutlist* file:



```
# The "checkoutlist" file is used to support additional version controlled
# administrative files in $CVSROOT/CVSROOT, such as template files.
#
# The first entry on a line is a filename which will be checked out from
# the corresponding RCS file in the $CVSROOT/CVSROOT directory.
# The remainder of the line is an error message to use if the file cannot
# be checked out.
#
# File format:
#
#       [<whitespace>]<filename><whitespace><error message><end-of-line>
#
# comment lines begin with '#'
users
```

Add the file *users* to the CVSROOT and commit both the *checkoutlist* and *users* files:

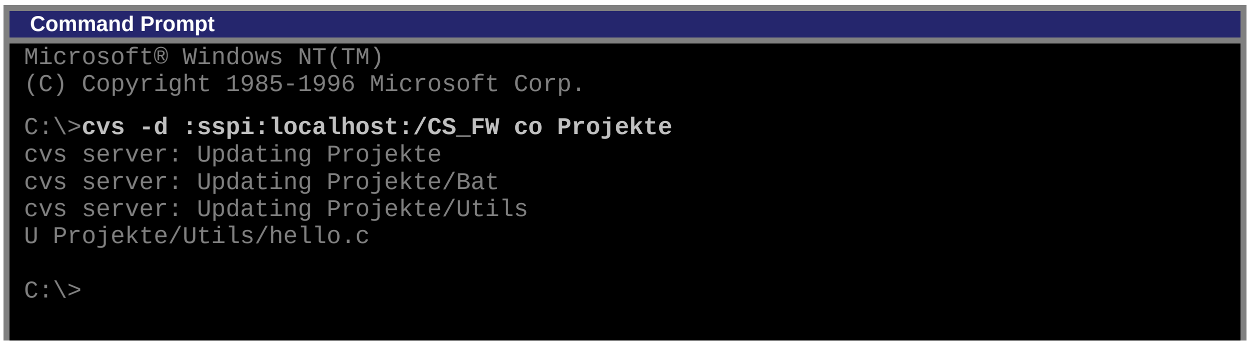


Command Prompt

```
Microsoft® Windows NT(TM)
(C) Copyright 1985-1996 Microsoft Corp.

C:\>cd CVSROOT
C:\CVSROOT>cvs add users
cvs server: scheduling file `users' for addition
cvs server: use 'cvs commit' to add this file permanently

C:\CVSROOT>cvs commit -m "config file changes"
cvs commit: Examining .
Checking in checkoutlist;
/myrepo/CVSROOT/checkoutlist,v  <--  checkoutlist
new revision: 1.6; previous revision: 1.5
done
RCS file: /myrepo/CVSROOT/users,v
done
Checking in users;
/myrepo/CVSROOT/users,v  <--  users
initial revision: 1.1
done
cvs server: Rebuilding administrative file database

C:\CVSROOT>
```

Testing of Integration (command line)

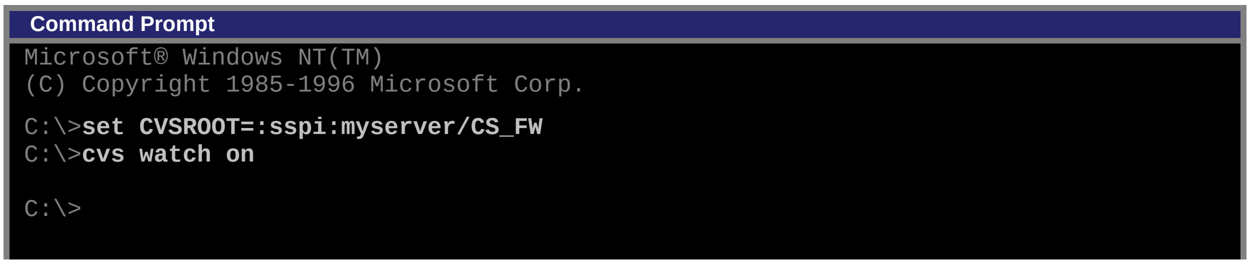
1. Check out a module:



Command Prompt

```
Microsoft® Windows NT(TM)
(C) Copyright 1985-1996 Microsoft Corp.

C:\>cvs -d :sspi:localhost:/CS_FW co Projekte
cvs server: Updating Projekte
cvs server: Updating Projekte/Bat
cvs server: Updating Projekte/Utils
U Projekte/Utils/hello.c

C:\>
```

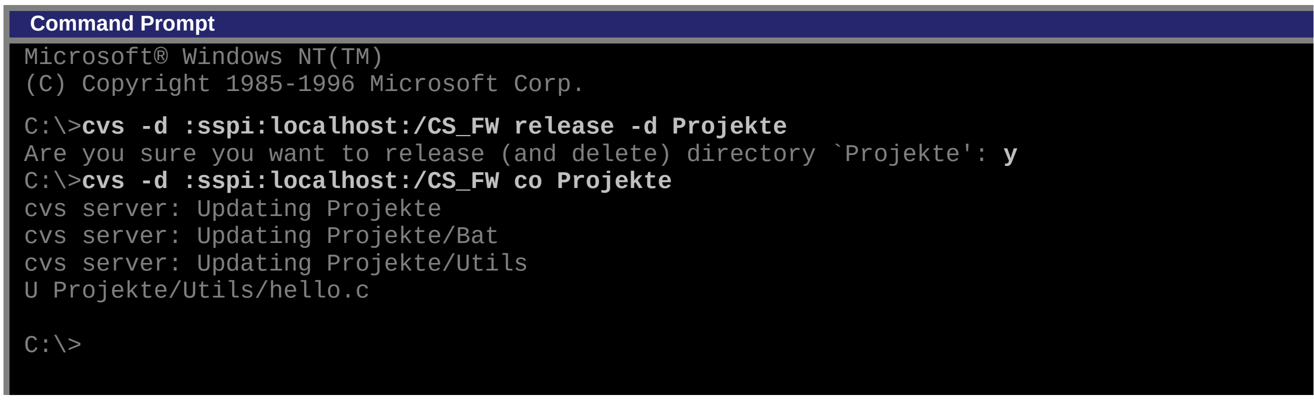
2. Enable watches



Command Prompt

```
Microsoft® Windows NT(TM)
(C) Copyright 1985-1996 Microsoft Corp.

C:\>set CVSROOT=:sspi:myserver/CS_FW
C:\>cvs watch on

C:\>
```

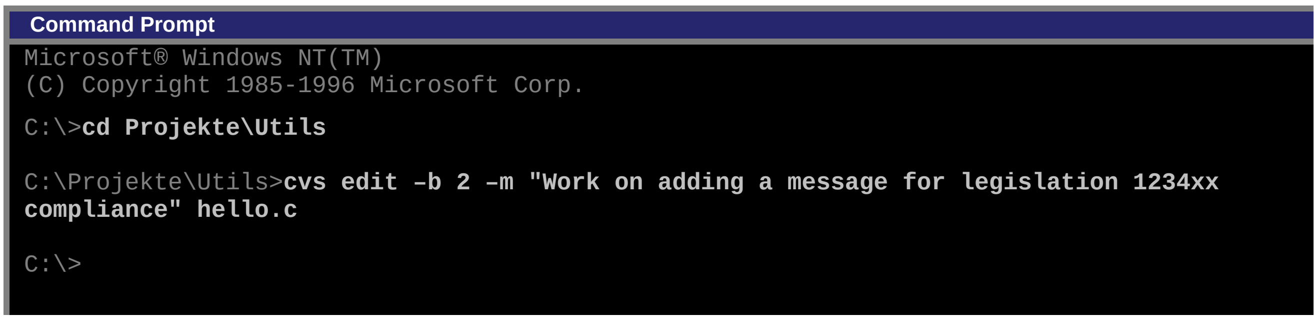
3. Release then check out a module:



Command Prompt

```
Microsoft® Windows NT(TM)
(C) Copyright 1985-1996 Microsoft Corp.

C:\>cvs -d :sspi:localhost:/CS_FW release -d Projekte
Are you sure you want to release (and delete) directory `Projekte': y
C:\>cvs -d :sspi:localhost:/CS_FW co Projekte
cvs server: Updating Projekte
cvs server: Updating Projekte/Bat
cvs server: Updating Projekte/Utils
U Projekte/Utils/hello.c

C:\>
```

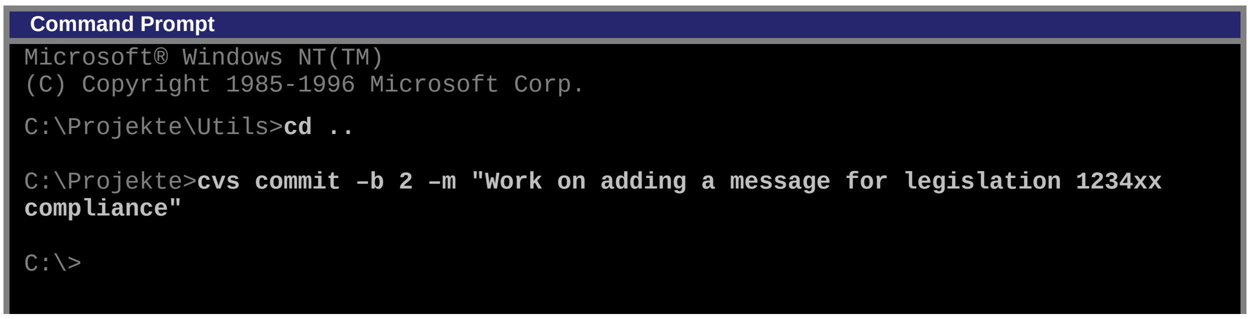
4. Create a bug using Bugzilla (use a web browser). Note down the bug number created in Bugzilla.
5. Begin work on a file using a bug number:



Command Prompt

```
Microsoft® Windows NT(TM)
(C) Copyright 1985-1996 Microsoft Corp.

C:\>cd Projekte\Utils

C:\Projekte\Utils>cvs edit –b 2 –m "Work on adding a message for legislation 1234xx
compliance" hello.c

C:\>
```

6. Make the changes to the source code hello.c
7. Commit all changes for this bug:



Command Prompt

```
Microsoft® Windows NT(TM)
(C) Copyright 1985-1996 Microsoft Corp.

C:\Projekte\Utils>cd ..

C:\Projekte>cvs commit –b 2 –m "Work on adding a message for legislation 1234xx
compliance"

C:\>
```

The comment from the commit – and the name of the file being committed is logged in the bug in Bugzilla.

Testing of Integration (CVS Suite Tortoise)

The CVS Suite version of TortoiseCVS include a "Use Bug" field on the edit dialog, and both "Use Bug" and "Mark Bug" field on the commit dialog.

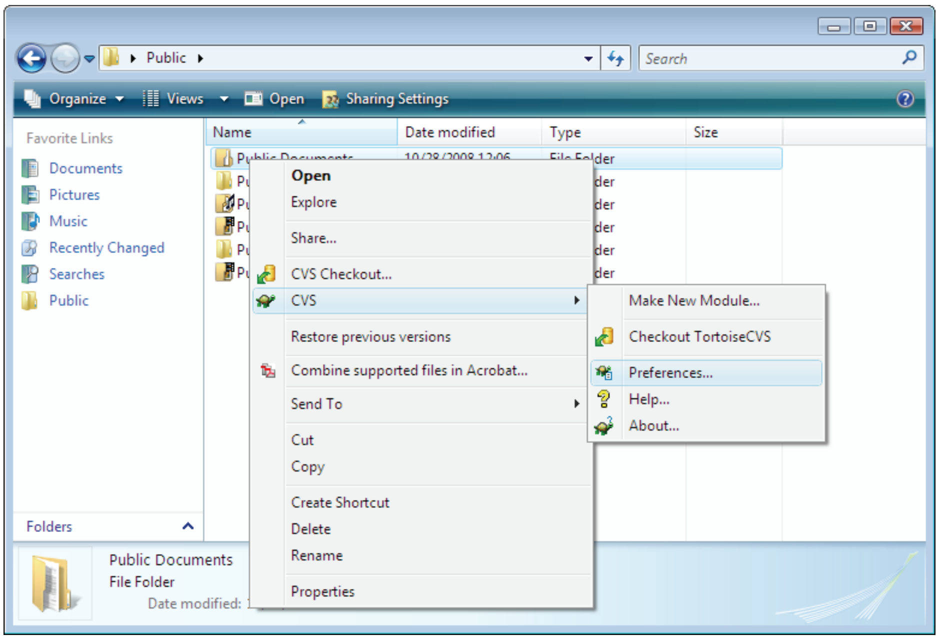
The "Use Bug" field is the most common method of working with change sets:

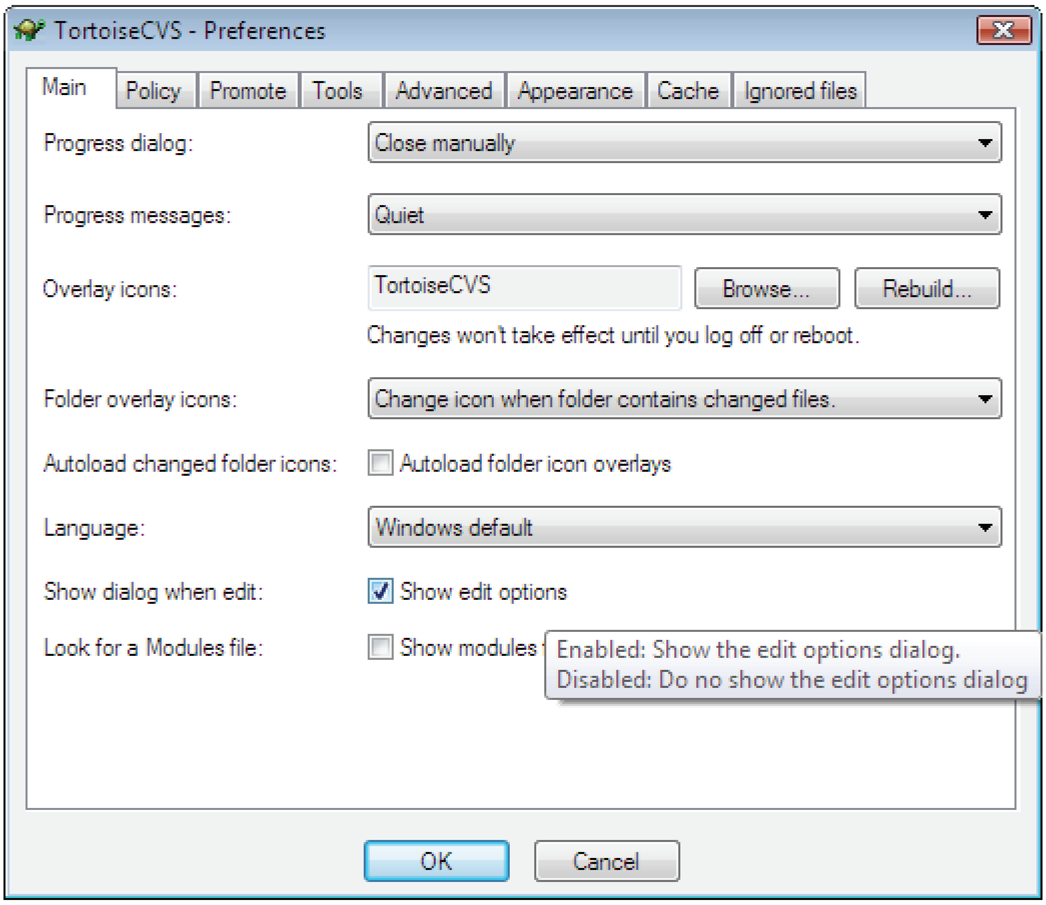
– Supply a bug number when you begin work on a file
– Can have several Bug numbers in use at the same time
– Can commit files selectively based on the bug number
– Patches and Checkin Comment can be attached to Bug in Defect Tracking System (Bugzilla)

The “Mark Bug” field is an alternative method of working with change sets where the bug number (or numbers) are supplied at the time of check in / commit only. The patch and comment are still applied to the bug in the defect tracking system (Bugzilla).

Firstly perform the steps 1 to 4 of “Testing of Integration (command line)” above.

1. Set the Tortoise Preference:





Progress dialog: Close manually

Progress messages: Quiet

Overlay icons: TortoiseCVS

Changes won't take effect until you log off or reboot.

Folder overlay icons: Change icon when folder contains changed files.

Autoload changed folder icons: ☐ Autoload folder icon overlays

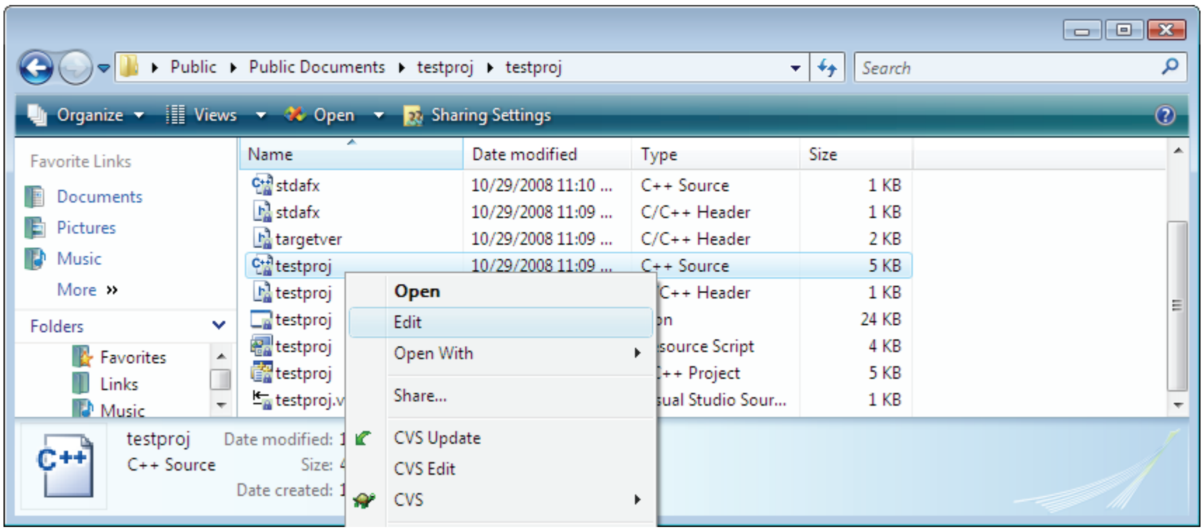
Language: Windows default

Show dialog when edit: ☑ Show edit options

Look for a Modules file: ☐ Show modules

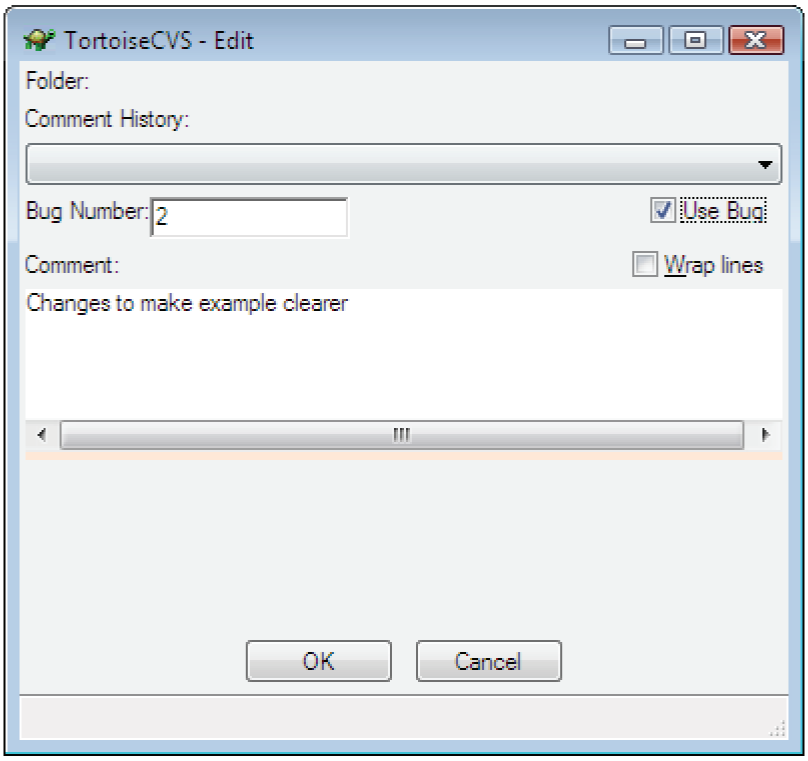
Enabled: Show the edit options dialog.
Disabled: Do no show the edit options dialog

2. Begin work on a file using a bug number:



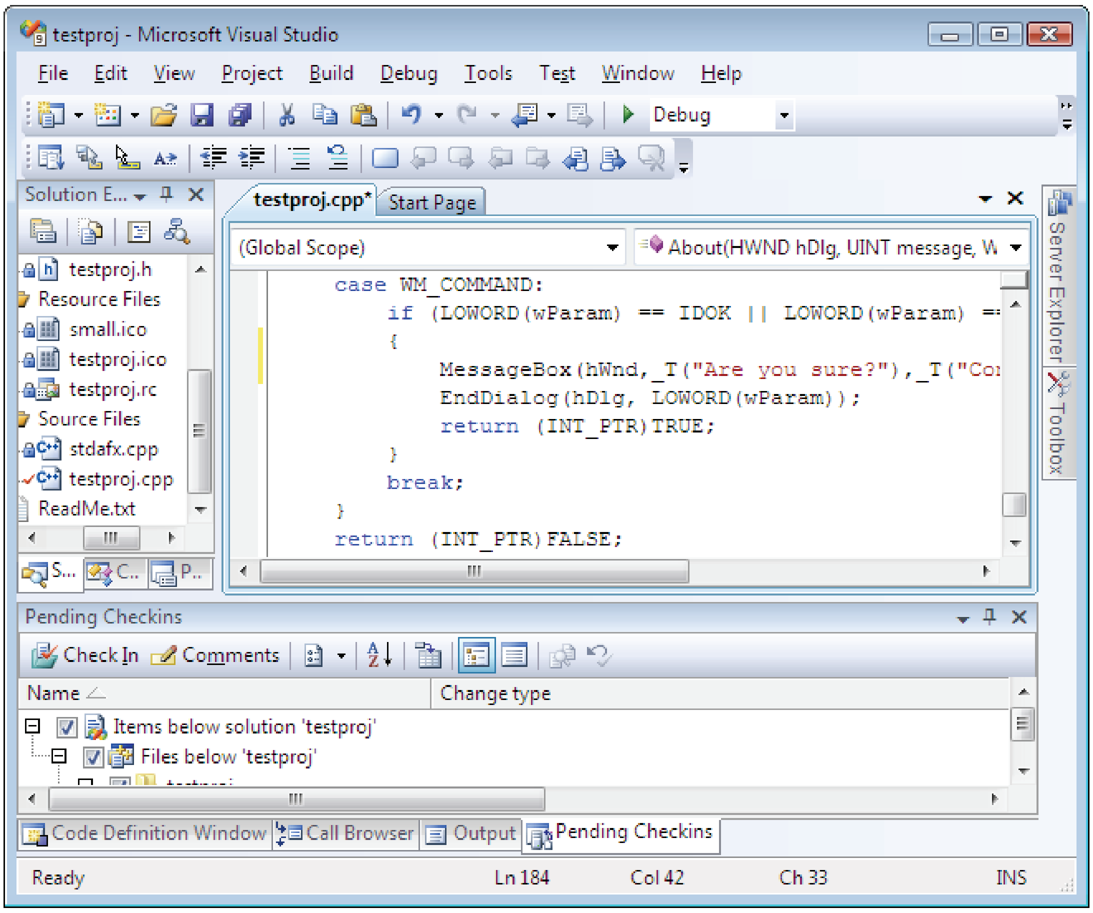
Name	Date modified	Type	Size
stdafx	10/29/2008 11:10 ...	C++ Source	1 KB
stdafx	10/29/2008 11:09 ...	C/C++ Header	1 KB
targetver	10/29/2008 11:09 ...	C/C++ Header	2 KB
testproj	10/29/2008 11:09 ...	C++ Source	5 KB
testproj		C++ Header	1 KB
testproj		n	24 KB
testproj		source Script	4 KB
testproj		++ Project	5 KB
testproj.v		ual Studio Sour...	1 KB

testproj Date modified: 1
C++ Source Size: 4
Date created: 1



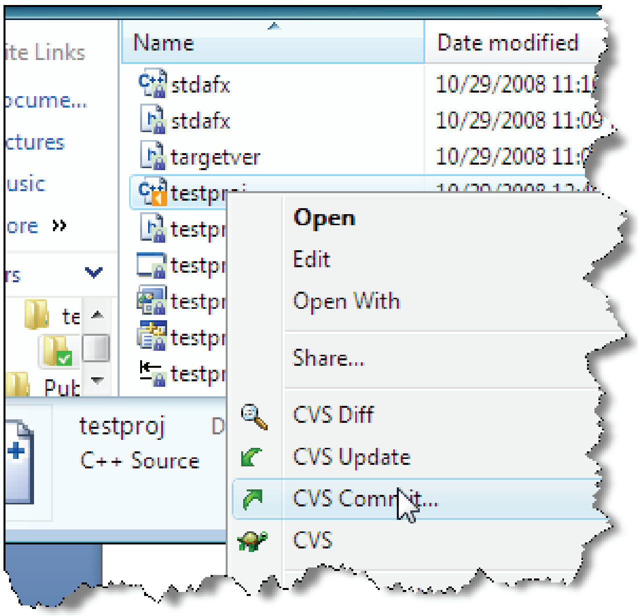
Folder:
Comment History:
Bug Number: 2
[✓] Use Bug
Comment:
[ ] Wrap lines
Changes to make example clearer

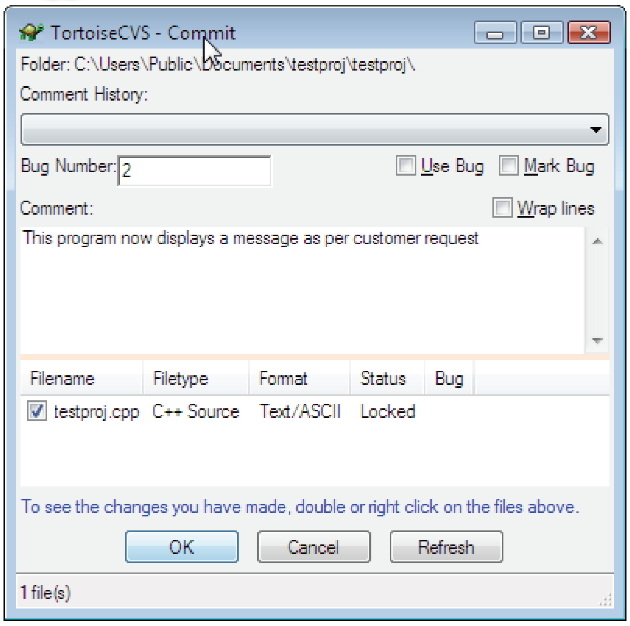
3. Make the changes to the source code testproj.cpp



```
    case WM_COMMAND:
        if (LOWORD(wParam) == IDOK || LOWORD(wParam) ==
        {
            MessageBox(hWnd,_T("Are you sure?"),_T("Co
            EndDialog(hDlg, LOWORD(wParam));
            return (INT_PTR)TRUE;
        }
        break;
    }
    return (INT_PTR)FALSE;
```

4. Commit all changes for this bug:





Folder: C:\Users\Public\Documents\testproj\testproj\

Bug Number: 2

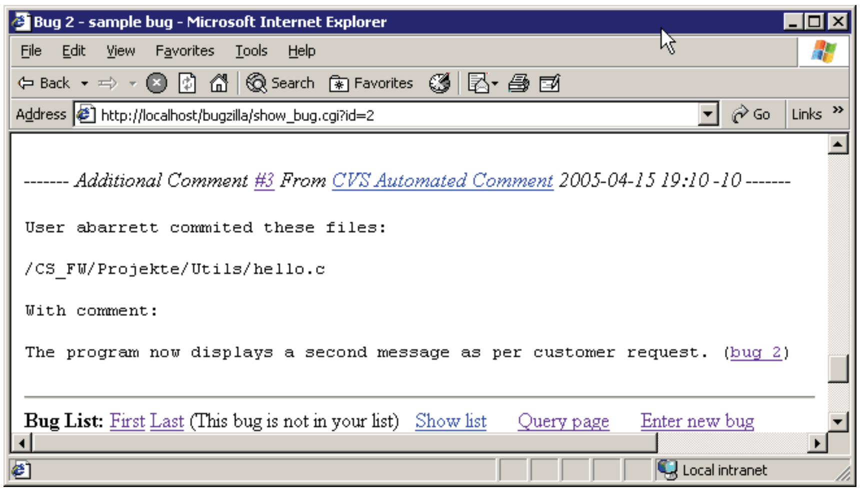
Comment:

This program now displays a message as per customer request

Filename	Filetype	Format	Status	Bug
testproj.cpp	C++ Source	Text/ASCII	Locked	

To see the changes you have made, double or right click on the files above.

1 file(s)


Bug 2 - sample bug - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Address http://localhost/bugzilla/show_bug.cgi?id=2
Go Links
Local intranet

------- *Additional Comment #3 From CVS Automated Comment* 2005-04-15 19:10 -10 -------

```
User abarrett commited these files:

/CS_FW/Projekte/Utils/hello.c

With comment:

The program now displays a second message as per customer request. (bug 2)
```

Bug List: First Last (This bug is not in your list) Show list Query page Enter new bug

Creating Branches

Branches may be created:

- By running the CVS command *tag* with the *–b tag* option on a checked out copy (sandpit)
- on the server using rtag
- Using the CVS Suite Studio graphical client

Also see the section *Advanced Client Functions* for information on this topic.

Creating Promotion Levels

CVS uses branches to implement a promotion level. The difference is simply how you use them.

See the section *Advanced Client Functions* for information on this topic.

Promoting

Promote may be done using:

- the CVS command *update* and *commit* on a designated promotion level (sandpit)
- Using the CVS Suite Studio and TortoiseCVS graphical client (Promote menu)

There are two ways to promote a revision from the Trunk (or a branch) to a promotion level: Move and Merge. See the section *Advanced Client Functions* for information on the Promote Move and Promote Merge topics.

Merging-in Branches

Promote may be done using:

- the the CVS command *update* with the *-j branchname* flag (sandpit)
- Using the CVS Suite Studio and TortoiseCVS graphical client (Merge menu)

See the section *Advanced Client Functions* for information on Merging-in Branches and Using Mergepoints.

Bug ID's (User Defined Change Sets)

Most CVS commands now support a a user defined change set, bug or defect identifier. The identifier can be used to group or mark files, and the identifier can also be used to operate on files with that identifier. See the *Client* section for more information on how to use this feature in the client, and the *Integration with Bugzilla* heading in this section for how to use the bug number on the server.

Access Control Lists

With Access Control Lists you can define complex rules for which users or groups of users can perform different tasks on parts of your CVS repository. Access Control Lists are most commonly used to prevent unauthorised merging, update or delete of objects from test or production branches.

This security information exists outside of the permissions on the individual files, and therefore is kept in place when the repository is restored from backup, moved between servers or moved between server platforms (eg: Windows to Unix).

Access Control Lists are primarily designed to control access to CVS modules and directories. It is possible to set different access control levels to different files in a directory however the results of the "none" keyword may not be intuitive. You should always assume that if a person has permission to read a directory that contains a file that they also have permission to read that file.

Chacl command

Using CVS command *chacl* you can define access control lists to manage security on the CVS repository.

```
cvs chacl [-R] [-r branch] [-u user] [-j branch] [-n] [-p priority] [-m message]
[no]{read|write|create|tag|control}[,...]]
[-d] [file or directory...]
        -a access       Set access
        -d              Delete ACL
        -j branch       Apply when merging from branch
        -m message      Custom error message
        -n              Do not inherit ACL
        -p priority     Override ACL priority
        -r branch       Apply to single branch
        -R              Recursively change subdirectories
        -u user         Apply to single user
```

The *chacl* command operates with a locally checked out copy of your files.

Access Roles

CVSNT defines the following access roles:

– Read	- able to read this file
– Write	- able to add revisions to this file
– Create	- able to add a file to the repository
– Tag	- able to apply tags to files and create branches
– Control	- able to administer this file and change ACL's

Access by Groups

The Access Control Lists by default assign permissions to a user. Any groups defined in the group administration file can also be used as a username. See the section on the group administration file for more information.

Typically if you are using the SSPI protocol on a Windows server the groups are the Windows Active Directory Groups, not the groups defined in the group administration file.

Default ACL's

Every object stored in the CVSNT repository has a default ACL. This can be set using the following command:

```
cvs chacl -a access -R
```

If no access is specifically set, then the default is that all users can *read*, *write*, *tag* and *create*. The *control* permission is reserved for the members of the *admin* group (Administrators on windows).

Branch ACL's

Each branch will use the repository default ACL unless set otherwise. The branch access can be set using the following command:

```
cvs chacl -r branch -a access -R
```

Commit ID's

Every commit transaction in a CVS repository has a unique commit ID. This ID can then be used to re-action the exact same change set at a later date.

Commit ID's are also available in the administrative files as a private variable.

For example, to rollback a commit that you just performed:

```
cvs update –j @commitid –j "@<commitid" filename or modulename
```

"@<commitid"	revision before that commit
@commitid	revision of that commit

Release tags

tag=tag

To allow customers with multiple deployment environments for their applications it is now possible to base one tag on another.

So the production environments can checkout the branch *Production* rather than *prod_1_2_2*. When release 1.2.3 is ready for production the administrator sets *Production=prod_1_2_3* and the next time a CVS *update* command is issued in the production environment the new production version will be installed.

Branch alias

Branch x is also known as y

```
cvs tag –A –r existing-tag new-tag
```

Branch point in time alias

Branch x at this point in time is known as y

```
cvs tag –r existing-tag new-tag
```

Implementing Methodologies

In the first chapter we discussed four methodologies for versioning, and other decisions related to your management objectives and company culture.

This section provides a guide to how to implement those choices with server side configuration.

Some configuration is also required in how the client is used which is discussed in the Clients section.

Implementing the methodology is generally a lot simpler than deciding what it should be. It is strongly recommended that you should decide what versioning methodology will best meet your business objectives and fit within your company culture before installing and configuring CVSNT server or any clients.

Reserved / Unreserved

By default CVSNT server uses the unreserved model. To enforce use of a reserved model you will need to make configuration changes to the server, and also to the client.

Reserved

CVSNT has four implementations of the reserved model:

Administration Reserved – Not recommended

The administrator can reserve all versions and all branches of a file. This is a very basic lock and its use is not recommended.

All reserved modes use watches

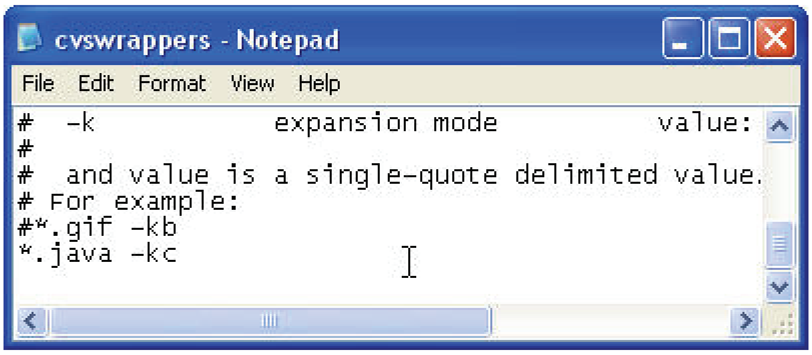
To use any of the reserved modes it is necessary on the client to enable watches using "cvs watch on" in the working directory (where the repository is checked out). Clients such as Eclipse and Tortoise have settings that enforce this automatically. See the Clients section for more information.

Reserved for Communication Only

This is not really reserved at all however is worth mentioning. Using this technique all users can edit the same version of the same file at the same time, however each is informed that other users are editing it. To enable this requires no changes on the server.

Reserved Co-operatively

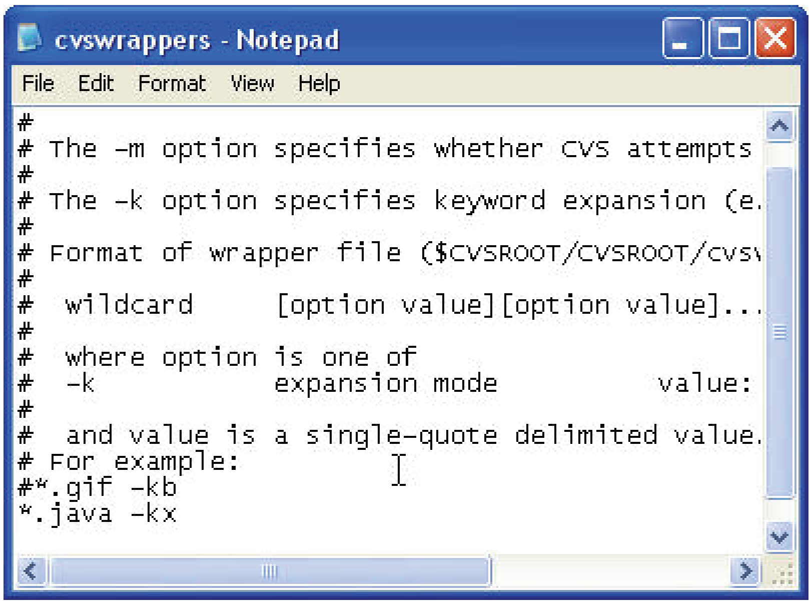
This is traditional reserved editing while still allowing users to override it and work in an unreserved manner. To enable this edit the *cvswrappers* administration file, set the file types which should use this mode to –kx (or –kvx etc).



```
#   -k             expansion mode          value:
#
#   and value is a single-quote delimited value.
# For example:
#*.gif -kb
*.java -kc
```

Reserved Exclusive

This is traditional reserved editing and will prevent CVSNT working in an unreserved manner. To enable this edit the *cvswrappers* administration file, set the file types which should use this mode to –kx (or –kvx etc). By setting this in the administration file it is possible to allow some repositories to use reserved exclusive and some to use other modes.



```
#
# The -m option specifies whether CVS attempts
#
# The -k option specifies keyword expansion (e.
#
# Format of wrapper file ($CVSROOT/CVSROOT/cvsw
#
#  wildcard     [option value][option value]...
#
#  where option is one of
#  -k           expansion mode          value:
#
#  and value is a single-quote delimited value.
# For example:
#*.gif -kb
*.java -kx
```

Distributed / Centralised

By default CVSNT clients are designed to work in a distributed mode. If you are working distributed then you will checkout an entire module to your local drive and work on it there.

If you do not have support with March Hare Software Ltd then you will not receive any assistance with attempting to use CVSNT clients on a network share.

Checking out single files or only a portion of the source code to the local disk and the remainder of the files being available in a central shared area requires setting up a trigger on the server.

When a file is modified and committed to the CVSNT server then the relevant directory will be updated with the new *reference copy*. For example:

Location	***branch***
All Files and Directories	
/usr/opt/devel	TRUNK
/usr/opt/test_1_0	Test_1_0
/usr/opt/prod_1_0	Prod_1_0
/usr/opt/test_1_1	Test_1_1
/usr/opt/prod_1_1	Prod_1_1

Updating Centralised Copy on Windows Server

The CVSNT Server for Windows includes a plugin for build management. When the commit has completed the file `build_make.bat` in the CVSROOT will be executed with the module name and branch as parameters.

A simple DOS batch script can be used to run **cvs update** commands on the correct directory for each branch.

Updating Centralised Copy on Unix, Linux or Mac OS X Server

When the commit has completed the server will execute the shell scripts or perl scripts specified in the **postcommand** administration file.

Create a shell script or perl script to update the correct reference directory based on your own rules. Alternatively if you want to use a simpler interface then you can use the build management plugin and a `build_make.sh` script in CVSROOT.

Store Object Code

By default CVSNT will store all code, though some clients such as Eclipse have settings that can ignore object code overriding this.

If you decide to store object code then you will have to decide the point in time that the code should be checked in. Generally this is either the same time as the source is checked in (from the developer) or at build time.

Checking the object code in from the build environment ensures that the object code is compiled using a pre-defined environment not subject to developer preferences.

Keeping object code, from the developer

If you decide to keep the developers copy of the source code there is nothing more to do.

Keeping object code, from the build environment

The following needs to be configured:

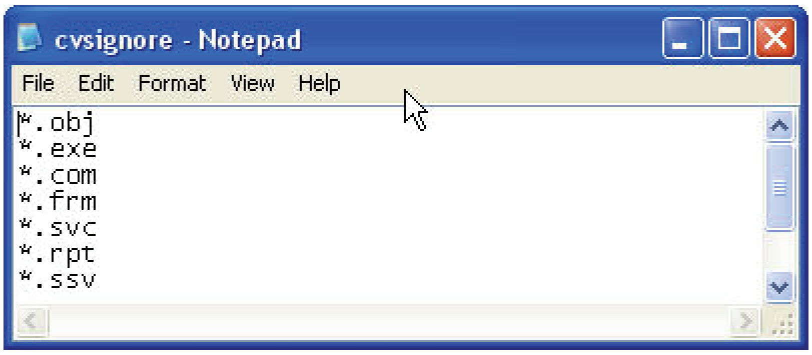
– Restrict access permissions on the directories containing the object code so that developers cannot commit changes to these files.

– Structure the repository so that developers can easily commit source code and not object code (eg: /src /bin /lib).

– Use the build manager plugin to begin a build when changes are committed. At completion of the build a commit should run from the "reference area" which includes the build log file. Optionally e-mail the build log to the author.

Ignoring object code

To configure CVSNT to ignore object code set each "file extension" for the object code files in the *cvsignore* administration file, eg:



```
*.obj
*.exe
*.com
*.frm
*.svc
*.rpt
*.ssv
```

Promotion Model

Whilst promotion levels are logically managed on the server, the creation and manipulation of them occurs on the client.

It is usual to configure promotion branches so that developers cannot commit changes. This is usually reserved for QA staff.

Insulation and Communication

Insulation

The level of insulation between users is a factor of:

– Branching / which "reference area"
For configuring this please see the section on setting up centralised areas using triggers.

> If two users share a branch and a "reference area" then when one completes their work then the other will immediately see the same changes. If they work on separate branches then they will have different "reference areas" and will not see each others changes until there is an explicit merge into their branch.

– Shared "working directories"
If two users share a working directory on a network drive then they will have no insulation.

Communication

Automated communication is set up using the Email Integration or CVSMailer. See the earlier sections on Email integration and/or CVSMailer for more information.

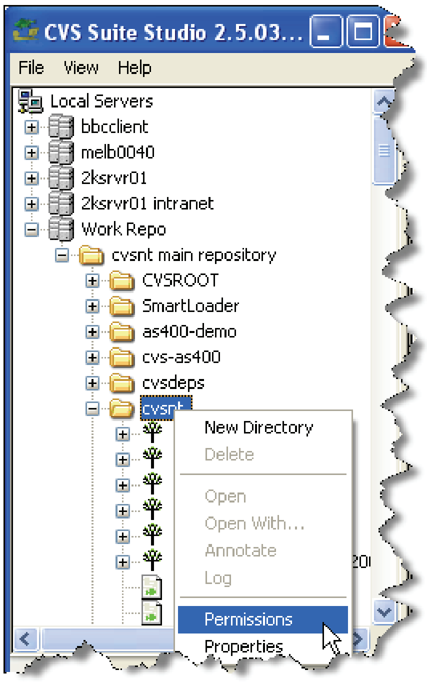
Part III – Client

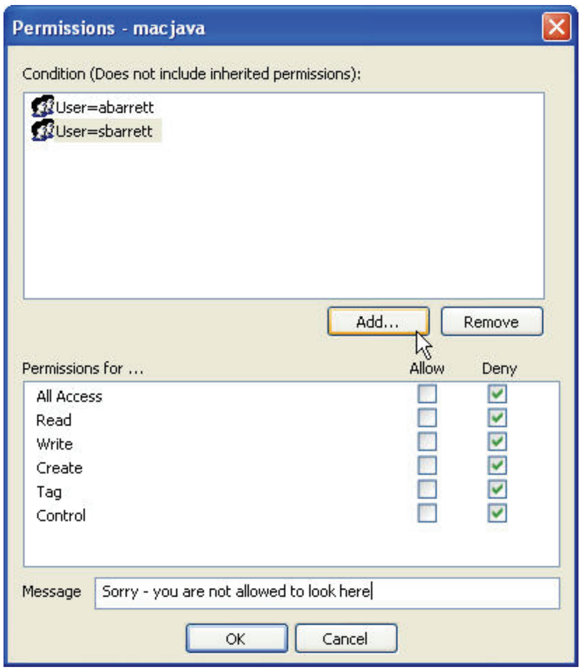
Setting up CVS Suite Studio Client on Windows

CVS Suite Studio is the recommended tool graphical front end for CVSNT and provides a range of powerful commands for performing the most common CVSNT tasks in a clear and simple user interface.

Set and Manage ACL's

To use ACL's you first must configure the ACLMode, see the Administration section for more information. To deny access by default set `aclmode = normal` (if you do not set this CVSNT will allow access by default). Once the repository wide default is set, you can then use CVS Suite Studio to add or remove permissions to folder on the server:





Condition (Does not include inherited permissions):

User=abarrett
User=sbarrett

Permissions for ...

	Allow	Deny
All Access	☐	✓
Read	☐	✓
Write	☐	✓
Create	☐	✓
Tag	☐	✓
Control	☐	✓

Message: Sorry - you are not allowed to look here

CVSNT Workflow

This section is intended to give the practical workflow when working with one of the popular methodologies discussed in the theory section. This section assumes that your repository, triggers (eg: keeping a read only "reference" copy of the source), integrations and access controls are already set up.

Workflow Definition

There are many techniques to define the workflow of your organisation in CVSNT, however these are not usually implemented in the client – but in the server using rules. There are many places on the server that rules can be defined depending on how they affect the versioning process. See the *Administration* section earlier in this book or *Implementing Methodoligies* below for more information about defining the workflow.

Version Numbers

The internal version numbers used by CVSNT do not specifically correlate to the version identifiers used by you or your organisation to identify individual file revisions or the revision of a group of files (such as a book or a software application).

Therefore it is important to distinguish between the version numbers used internally by CVSNT and external numbering schemes used by you. These external numbering schemes can and should be recorded in CVSNT as tags or labels.

Bug ID's

Most CVS commands now support a bug or defect identifier. Often this is much more useful to the end user than the physical or internal version number of the file. The bug identifier can be used to group or mark files, and the identifier can also be used to operate on collections of files already marked with that identifier, including merge and commit.

Edit

-B	Mark this file with this identifier. A working revision can only have a single identifier.

Unedit

-b	Unedit files with this identifier

Update (Merge)

-b	When merging development branch (or Trunk) into the release branch it will take the lowest revision with that bug number and the highest revision with that bug number and merge the differences into the current branch. This is not a mergepoint.

Commit

-b	Commit files with this identifier.
-B	Commit files and mark the new version with this identifier.

Creating Branches

Creating Promotion levels and branches is not a part of the regular workflow – these are administration functions. Merging changes up (promoting) is also an administration function. Generally this section does not describe these activities since they are not generally performed from the graphical clients. These administration activities are best performed using the command line CVS tool. See the Administration section for more information.

Creating Sandboxes

Creating sandboxes is not a regular part of the users workflow. If you are using the Distributed model of CM then each user generally has a complete "checked out" copy of the source from the server in a sandbox. This sandbox usually resides on the local workstations hard drive, but may exist on a network share (see system requirements). This checkout is generally done once and rarely deleted (released) and started again.

If you are using the Centralised Reserved model then each user checks out and begins work on a file or set of files in one action.

CVS Suite includes the CVS Suite Studio which is the ideal tool for creating workspaces. TortoiseCVS and WinCVS can also be used to create workspaces.

Implementing Methodologies

In the first chapter we discussed four methodologies for versioning, and other decisions related to your management objectives and company culture.

This section provides a guide to how those choices map to use on the client.

Reserved / Unreserved

The majority of graphical CVS clients are designed to allow you to work in unreserved mode. Some clients allow you to also work in reserved mode to varying degrees.

To nominate a particular file type as one that should use "reserved" mode, define it in the *cvswrappers* administrative file with **–kc** (for cooperative reserved, or reserved for communication only) or **–kx** (for traditional reserved). More information on the administrative files can be found in the administration section on *cvswrappers*.

If you have a file type that should be reserved in some projects but not in others – create separate repositories for these two types of project.

There is nothing to set to enable unreserved mode.

Reserved

Working in reserved mode requires two steps in a "checkout process":

– Checkout (copy the file(s) to a work area)
  use the **cvs checkout** command.
– Reserved (allow this file to be worked on by the user)
  use the **cvs edit** command.

All reserved modes use watches

To enforce the need to "cvs edit" the file before changing it there must be a mechanism for CVSNT to know that the files should be checked out read-only. This is usually achieved by setting watches on. Note: if you are using –kx or -kc defined in the cvswrappers then this is not required.

To set watches on, checkout a module from the cvs repository and issue the command: *cvs watch on* to activate. In future all workspaces created for that module will be created read-only.

Reserved for Communication Only

This method employs the use of watches (as described above) but not –kc or –kx locking modes. In this way the files will be read only, but there is no hindrance to how many files and how many users of a file can access it at a time.

Reserved Co-operatively

Co-operative reserved uses the -kc *cvswrappers* option to ensure that files are not committed unless they are first editied. Since the watch mechanism merely makes files read-only, a smart user could simply remote the read-only attribute of the file using the unix *chmod* or DOS *attrib* commands.

If you are using co-operative edit's enforced in cvswrappers then when that changed file is committed the CVSNT server checks that it has already received an edit notification. If it has not then the user cannot commit the changes until the *cvs edit* command is used. More than one user can still edit a file at the same time, but they are always informed of this (and can be alerted via e-mail with the Email plugin).

Reserved Exclusive

Exclusive reserved uses the -kx *cvswrappers* option to ensure that files are not committed unless they are first editied, and prevents more than one user at a time having edit access to the file. Since the watch mechanism merely makes files read-only, a smart user could simply remote the read-only attribute of the file using the unix *chmod* or DOS *attrib* commands.

If you are using exclusive edit's enforced in cvswrappers then when that changed file is committed the CVSNT server checks that it has already received an edit notification. If it has not then the user cannot commit the changes until the *cvs edit* command is used. Since only one user at a time can have a file edited then it enforces the exclusivity of the lock.

Distributed / Centralised

The majority of graphical CVS clients are designed to allow you to work in distributed mode. Some clients allow you to also work in centralised mode to varying degrees. Implementing the Centralised model requires a read-only copy (shadow copy) of the workspace (or a branch of the workspace) available at all times. See the Section *Distributed / Centralised* in the *Administration* section.

Store Object Code

By default CVSNT will import and version all files in a directory. To exclude object code then set up the *cvsignore* administrative file.

Keeping object code, from the build environment

To keep the objects from the build environment only then change the access control list to allow developers to check out object code but not check it in. The build process should then “login” to cvs as an authorised user.

Promotion Model

See the section *Advanced Client Tasks* for more information about using the promotion model from the client.

Insulation and Communication

Insulation

Insulation is implemented by having separate workspaces for users who should be insulated form each others changes. If the projects are sufficiently different then they should also use different branches. See *CVS Suite Studio* for details on creating workspaces.

Communication

Communication is implemented by configuring the E-mail notification integration on the server. If two developers are both watching the same files then they will be alerted by e-mail when changes occur. Also see the headings *Find out about other peoples changes* and *All reserved modes use watches* in this section.

Unreserved Distributed Workflow

For a developer or an author working with source code files or documents controlled by CVSNT you will need to:

- Create a sandbox if one does not already exist
- Inform CVSNT that you have began working on a file (optional)
- Check your changes
- Synchronise your local file with the repository (update) or add a new local file to the repository
- Store your changes into the repository (commit)

The final stage also requires that you enter a description of your changes.

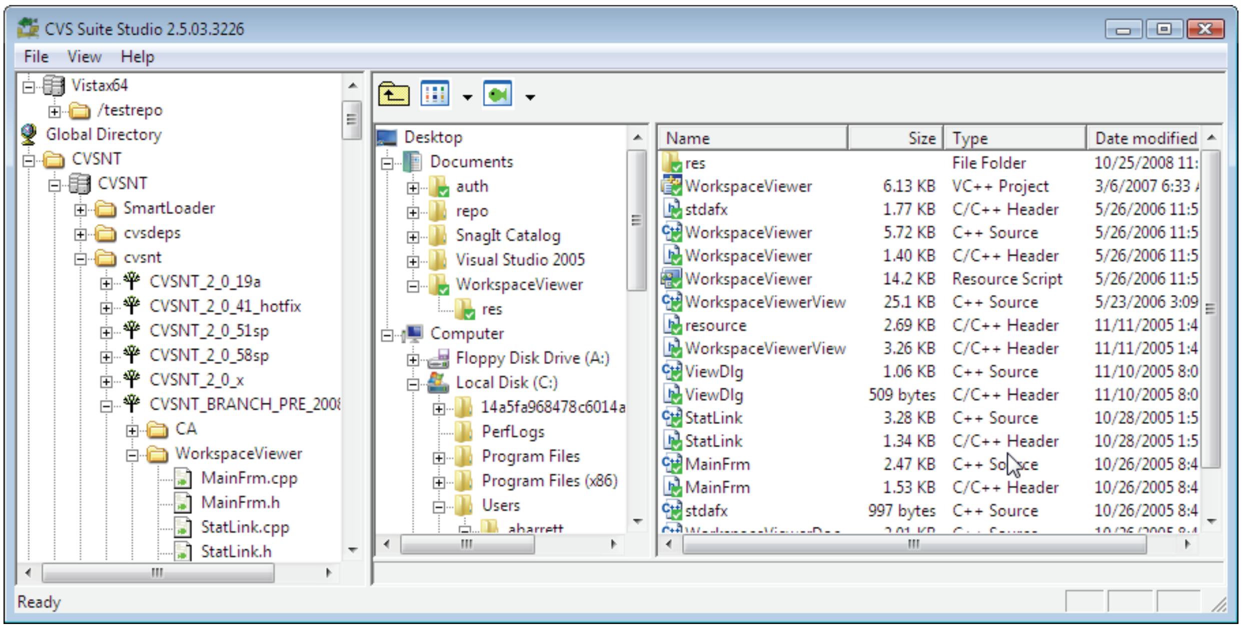
This workflow can be completed using any style of interface, we will look at Tortoise and WinCVS below.

CVS Suite Studio

CVS Suite Studio is the primary client graphical user interface of CVS Suite. CVS Suite Studio allows you to browse CVSNT repositories and to create workspaces containing individual files or entire modules and directories from either the Trunk or any branch or tag.

Browse

Open CVS Suite Studio and it will automatically discover any servers on your local network. It will discover servers located behind a router or on the other side of a VPN.



```
Vistax64
  /testrepo
Global Directory
CVSNT
  CVSNT
    SmartLoader
    cvsdeps
    cvsnt
      CVSNT_2_0_19a
      CVSNT_2_0_41_hotfix
      CVSNT_2_0_51sp
      CVSNT_2_0_58sp
      CVSNT_2_0_x
      CVSNT_BRANCH_PRE_2008
        CA
        WorkspaceViewer
          MainFrm.cpp
          MainFrm.h
          StatLink.cpp
          StatLink.h
```

```
Desktop
  Documents
    auth
    repo
    SnagIt Catalog
    Visual Studio 2005
    WorkspaceViewer
      res
  Computer
    Floppy Disk Drive (A:)
    Local Disk (C:)
      14a5fa968478c6014a
      PerfLogs
      Program Files
      Program Files (x86)
      Users
        abarrett
```

Name	Size	Type	Date modified
res		File Folder	10/25/2008 11:
WorkspaceViewer	6.13 KB	VC++ Project	3/6/2007 6:33
stdafx	1.77 KB	C/C++ Header	5/26/2006 11:5
WorkspaceViewer	5.72 KB	C++ Source	5/26/2006 11:5
WorkspaceViewer	1.40 KB	C/C++ Header	5/26/2006 11:5
WorkspaceViewer	14.2 KB	Resource Script	5/26/2006 11:5
WorkspaceViewerView	25.1 KB	C++ Source	5/23/2006 3:09
resource	2.69 KB	C/C++ Header	11/11/2005 1:4
WorkspaceViewerView	3.26 KB	C/C++ Header	11/11/2005 1:4
ViewDlg	1.06 KB	C++ Source	11/10/2005 8:0
ViewDlg	509 bytes	C/C++ Header	11/10/2005 8:0
StatLink	3.28 KB	C++ Source	10/28/2005 1:5
StatLink	1.34 KB	C/C++ Header	10/28/2005 1:5
MainFrm	2.47 KB	C++ Source	10/26/2005 8:4
MainFrm	1.53 KB	C/C++ Header	10/26/2005 8:4
stdafx	997 bytes	C++ Source	10/26/2005 8:4

Browse the server by pressing the + icon to open modules and directories. If you wish to see branches and tags then select those options form the pulldown menu. The options in the pulldown menu only affect modules and directories not already opened with the + icon.

Create a workspace

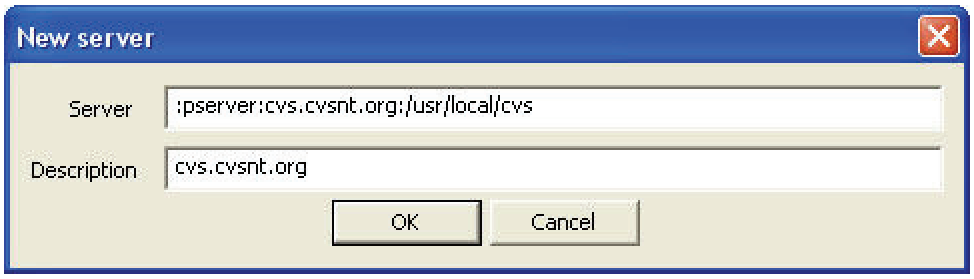
Create a workspace on your local PC drive by dragging a file or folder from the CVS Suite Studio panel on the left to a directory on the right.

Import files and directories to the repository

Import files and directories from your local PC drive to the repository by dragging a folder from the CVS Suite Studio panel on the right to the server on the left. The directory on the PC is unaffected by this operation. To create a "versioned workspace" on the client PC follow the above section *Create a Workspace*.

Add a repository that is not already listed

If your repository is not listed automatically in the CVS Suite Studio then you can add it by selectign File->new from the pulldown menu. Do not specify logon details in the name – if a login is required CVS Suite Studio will prompt you.

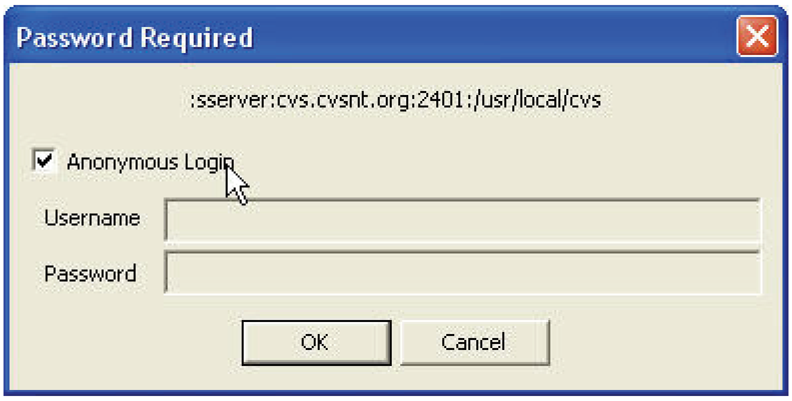


Server: :pserver:cvs.cvsnt.org:/usr/local/cvs

Description: cvs.cvsnt.org

Browse anonymously or specify a username and password

If a login is required to browse a repository then CVS Suite Studio will prompt you. If you want to check files into the repository or create a workspace that will allow *commit* then you would normally specify a user and password. If you are creating a work area for viewing only, or viewing and update then the anonymous user will usually be sufficient:



Operations on your workspace: Add, Commit etc

You can add files, commit (checkin) changes to your workspaces listed in the left hand window by right clicking on the folder. The actions available are identical to the Tortoise actions listed below.

Find out about other peoples changes

You can reserve a file for edit using CVSNT so that you are the only one who will have access to it while you are using it. However there are times that you do not want to lock people out – but need to know when changes are made. For example if one program depends on another.

It is possible to request that CVS notify you when other users begin work on a file or finish work on a file using the *notify* command.

Notify

For Notify to work your repository administrator must have already enabled e-mail notification via the *Notify* configuration file using a tool such as CVSMailer.

To enable watches you should use the CVSNT command line.

Use the *cvs watch on* command to enable watches and then *cvs watch add* each file that you want to be informed about.

Good comments

When using configuration management the kind of comments you enter and where you enter them becomes important.

If you do not have a tool such as CVS to automatically track who has changed what then you often add comments in the source code to indicate your initials and the change date. You may want to continue this practice however CVS will now automatically track this same information for you.

What is important are the comments you make when returning the file through version control. If in six months time you are trying to determine why a change was made a comment such as "fixes for September" is of little value – the date of the revision tells you that.

Good comments include:

– Bug number(s)
– References to other people / sponsors
– References to other documentation
– A succinct explanation of the purpose of the change

Here is an example of a good comment:

> *Bug 1234. Changes detailed in the functional spec h:\docs\fs23-1234.doc. Joe explained that the customer component should always display a warning before changes are written to the database. This is a first draft and probably needs some work.*

Here is a poor example of a comment:

> *Changes by Arthur. I made some changes to the WRITE trigger. Used askmess. A part of the September work.*

If you have a defect tracking system it is possible for the CVS repository administrator to have your comments at checkin automatically added to the defect tracking database.

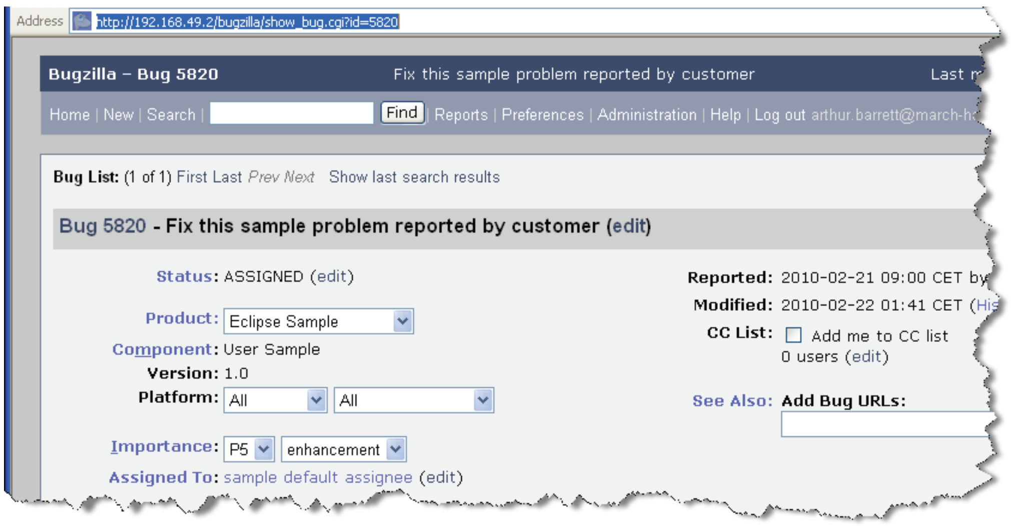
Bug fixing workflow with Promote

A typical request from QA managers is how to track what files have changed for each job or bug. The following section describes an example of this:

– Make a change to source code using Eclipse and commit to repository.
– View changes via Bugzilla
– View changes via SQL Query on Audit database or Bugzilla database
– View changes via CVS Suite command line client

Make changes to source code using Eclipse and commit to repository

In a typical commercial software development environment a programmer is given a task number, job number or bug number before beginning work. This bug number is then used to track the changes from requirements gathering through to release:



Address http://192.168.49.2/bugzilla/show_bug.cgi?id=5820

Bugzilla – Bug 5820 Fix this sample problem reported by customer Last m

Home | New | Search | Find | Reports | Preferences | Administration | Help | Log out arthur.barrett@march-h

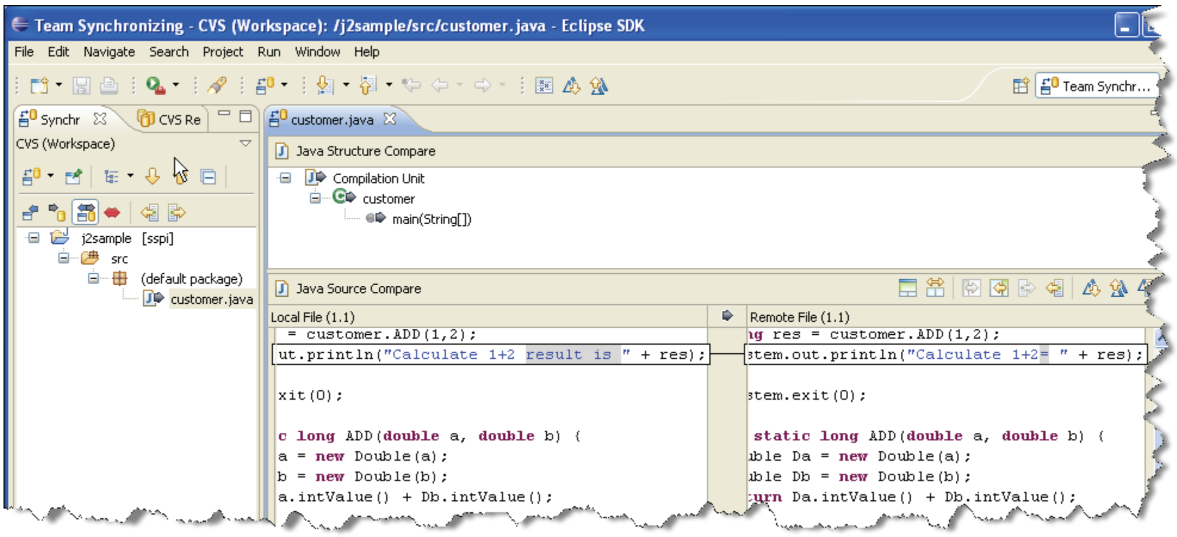
Bug List: (1 of 1) First Last Prev Next Show last search results

Bug 5820 - Fix this sample problem reported by customer (edit)

Status: ASSIGNED (edit)
Product: Eclipse Sample
Component: User Sample
Version: 1.0
Platform: All All
Importance: P5 enhancement
Assigned To: sample default assignee (edit)

Reported: 2010-02-21 09:00 CET by
Modified: 2010-02-22 01:41 CET (His
CC List: Add me to CC list
0 users (edit)
See Also: Add Bug URLs:

The developer can use the synchronize view to visually compare the changes in the eclipse workspace with the CVS Suite Server repository:



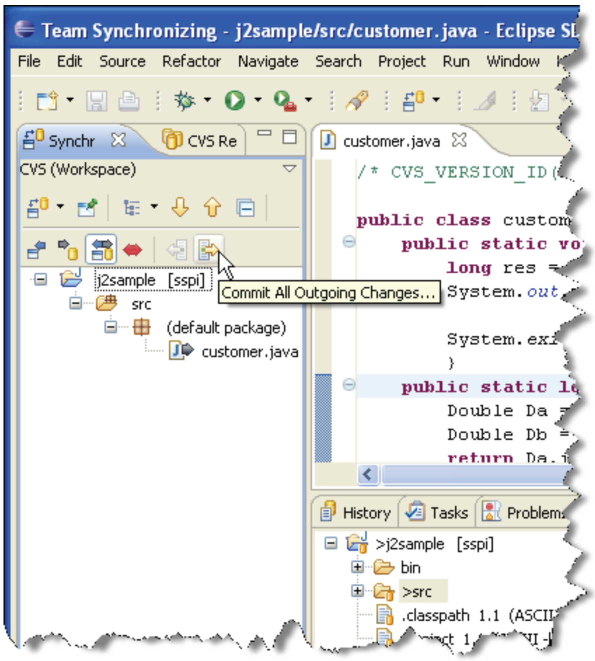
Local File (1.1)

```
 = customer.ADD(1,2);
ut.println("Calculate 1+2 result is " + res);

xit(0);

c long ADD(double a, double b) {
a = new Double(a);
b = new Double(b);
a.intValue() + Db.intValue();
```

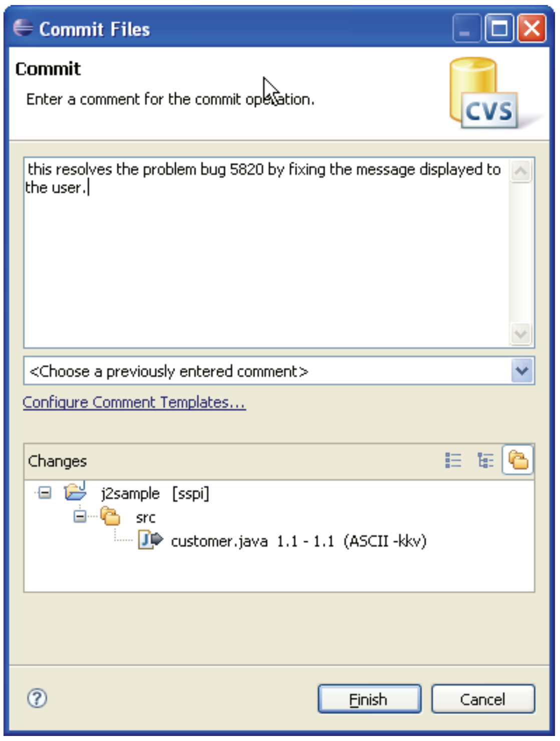
Remote File (1.1)

```
ng res = customer.ADD(1,2);
stem.out.println("Calculate 1+2= " + res);

stem.exit(0);

 static long ADD(double a, double b) {
uble Da = new Double(a);
uble Db = new Double(b);
urn Da.intValue() + Db.intValue();
```

When the developer is finished with the changes they can commit them back to the repository and link them to the job / bug using the bug number:


Team Synchronizing - j2sample/src/customer.java - Eclipse SI
File Edit Source Refactor Navigate Search Project Run Window
Synchr
CVS Re
CVS (Workspace)
j2sample [sspi]
src
(default package)
customer.java
Commit All Outgoing Changes...
History
Tasks
Problems
>j2sample [sspi]
bin
>src
.classpath 1.1 (ASCII

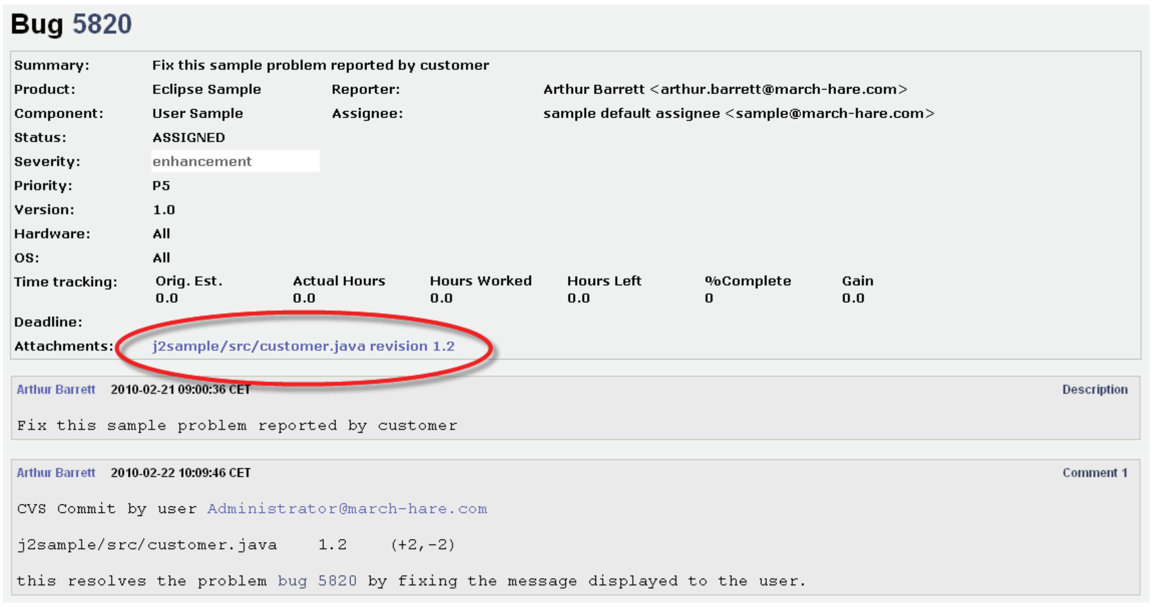
```
/* CVS_VERSION_ID(

public class custom
    public static vo
        long res =
        System.out

        System.exi
    }
    public static lo
        Double Da =
        Double Db =
        return Da.i
```


Commit Files
Commit
Enter a comment for the commit operation.
CVS
<Choose a previously entered comment>
Configure Comment Templates...
Changes
j2sample [sspi]
src
customer.java 1.1 - 1.1 (ASCII -kkv)
Finish
Cancel

this resolves the problem bug 5820 by fixing the message displayed to the user.

View changes in Bugzilla

The QA manager can see at a glance in Bugzilla all the files modified by any bug:



Bug 5820

Summary:	Fix this sample problem reported by customer		
Product:	Eclipse Sample	Reporter:	Arthur Barrett <arthur.barrett@march-hare.com>
Component:	User Sample	Assignee:	sample default assignee <sample@march-hare.com>
Status:	ASSIGNED		
Severity:	enhancement		
Priority:	P5		
Version:	1.0		
Hardware:	All		
OS:	All		

Time tracking:	Orig. Est.	Actual Hours	Hours Worked	Hours Left	%Complete	Gain
	0.0	0.0	0.0	0.0	0	0.0

Deadline:

Attachments: j2sample/src/customer.java revision 1.2

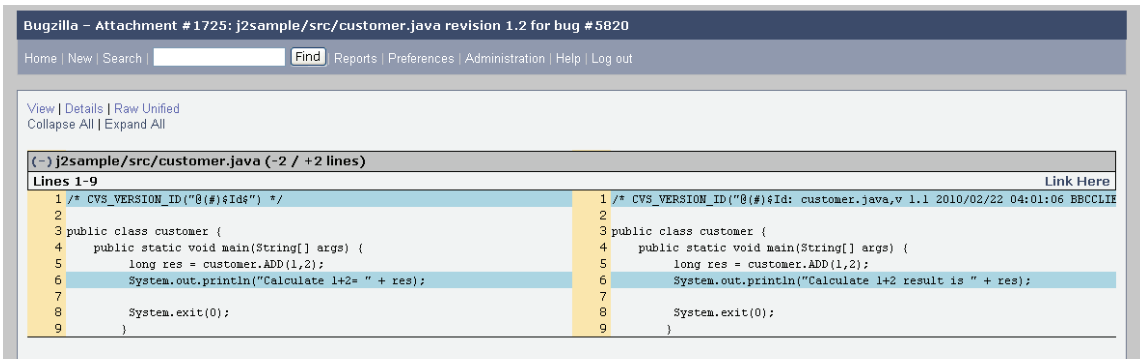
Arthur Barrett 2010-02-21 09:00:36 CET Description

```
Fix this sample problem reported by customer
```

Arthur Barrett 2010-02-22 10:09:46 CET Comment 1

```
CVS Commit by user Administrator@march-hare.com

j2sample/src/customer.java    1.2    (+2,-2)

this resolves the problem bug 5820 by fixing the message displayed to the user.
```

The diffs can even be code reviewed directly in Bugzilla:



Bugzilla – Attachment #1725: j2sample/src/customer.java revision 1.2 for bug #5820

Home | New | Search | Find | Reports | Preferences | Administration | Help | Log out
View | Details | Raw Unified
Collapse All | Expand All

(-) j2sample/src/customer.java (-2 / +2 lines)

Lines 1-9 Link Here

```
/* CVS_VERSION_ID("@(#)$Id$") */

public class customer {
    public static void main(String[] args) {
         long res = customer.ADD(1,2);
         System.out.println("Calculate 1+2= " + res);

         System.exit(0);
       }
```

```
/* CVS_VERSION_ID("@(#)$Id: customer.java,v 1.1 2010/02/22 04:01:06 BBCCLIE

public class customer {
    public static void main(String[] args) {
         long res = customer.ADD(1,2);
         System.out.println("Calculate 1+2 result is " + res);

         System.exit(0);
       }
```

View changes via SQL Query on Audit database or Bugzilla database

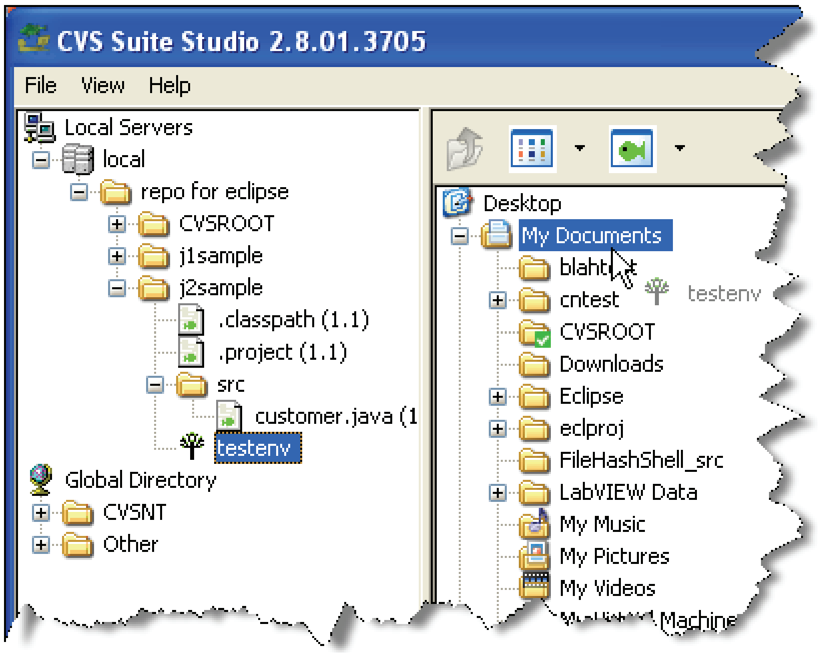
An SQL client (eg: Microsoft Excel) can query the audit database or the Bugzilla database to find all the files changed by a bug or all of the bugs affected by a file.

Promote to test or production by bug number

The CVS Suite tools all understand the bug number, and you can promote to any defined promotion level using a bug number, eg: promoting to test:

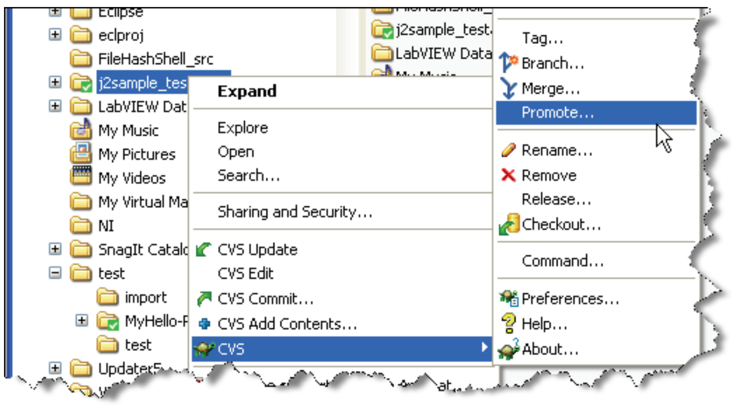
Firstly the QA manager creates a disk area to perform the promote:

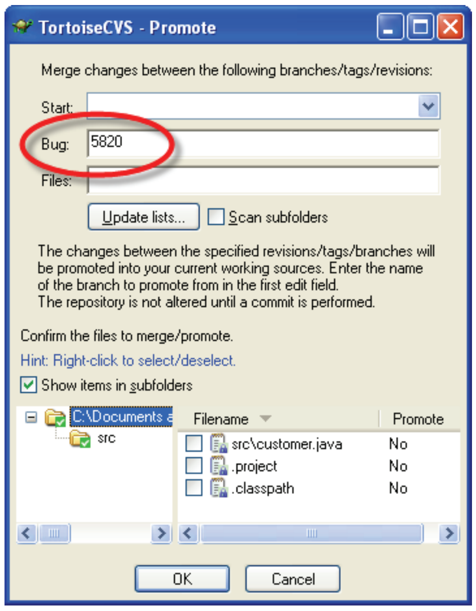
Drag the promotion level to a secure location:



Secondly use the right click promote menu

Right click the new promotion directory and choose the bug number to promote – there is no need to know what files are affected by the bug – all files affected are promoted:





Merge changes between the following branches/tags/revisions:

Start:

Bug: 5820

Files:

The changes between the specified revisions/tags/branches will be promoted into your current working sources. Enter the name of the branch to promote from in the first edit field.
The repository is not altered until a commit is performed.

Confirm the files to merge/promote.

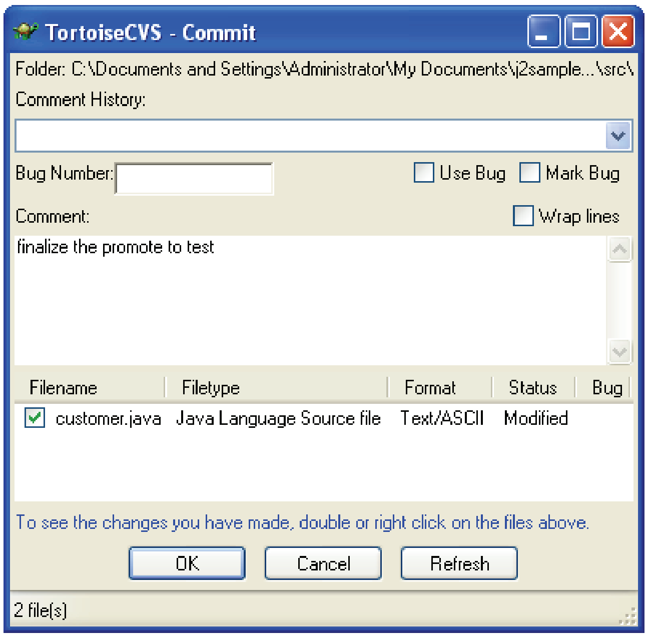
Hint: Right-click to select/deselect.

☑ Show items in subfolders

Filename	Promote
src\customer.java	No
.project	No
.classpath	No

Lastly use the right click commit menu

Right click the new promotion directory and choose commit – you can use the same bug number, or a new bug number (eg: a new bug that combines several other bug fixes into a release or service pack):



Folder: C:\Documents and Settings\Administrator\My Documents\j2sample...\src\

Bug Number:

Comment:

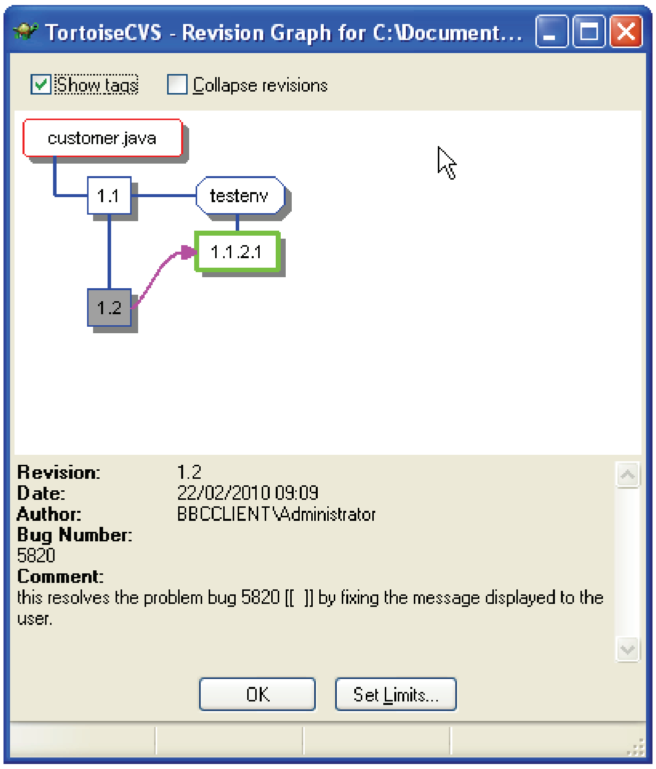
finalize the promote to test

Filename	Filetype	Format	Status	Bug
✓ customer.java	Java Language Source file	Text/ASCII	Modified	

To see the changes you have made, double or right click on the files above.

2 file(s)

You can use the revision graph feature to visually confirm the promotion:



Revision: 1.2
Date: 22/02/2010 09:09
Author: BBCCLIENT\Administrator
Bug Number:
5820
Comment:
this resolves the problem bug 5820 [[ ]] by fixing the message displayed to the user.

Advanced client functions

This section is intended to give the administrators a quick overview as to how to use CVS clients to perform general administrative functions such as creating branches and doing promotes (merges).

It is recommended that administrators be familiar with using the command line tool for performing administrative functions.

Creating branches

To create a branch use the **cvs tag** command. Branches are designed to allow development on the same files by teams with different objectives to occur at the same time. The most common reason to create a branch will be in a software project when version 1 of the product is almost complete and version 2 is beginning work.

At this stage the development of version 2 will continue on the trunk and a branch will be created for finishing version 1. The test and release promotion levels will be branched of the version 1 development branch.

There are two ways to create a branch:

- With a sandbox of the “from” files (eg: creating a branch “from” the trunk you begin with a sandbox of the trunk. Creating a branch does not alter the working directory ad sandbox in any way.
- Without a sandbox. In this case the command takes two parameters for the “from” and new branch names. To ensure that the files you expect to be tagged are the ones that are tagged it is necessary to ensure that noone commits any changes to the repository. Eg: you decide to create dev1 branch and between you making the decision and running the rtag command someone commits some work which breaks the build.

Creating a branch from a sandbox

Use the following command on the sandbox:

cvs tag –b *branch-name*

after this command the local sandbox will not be altered in any way. If you want to make changes to this branch you will need to check out the newly created branch using the checkout command.

Creating a branch without a sandbox

You can create a branch without using a sandbox, however if another person commits changes while the branch is being created then it will not be immediately clear which versions of the files received the branch. This is because branch creation and commits are not atomic. This problem does not apply to checkouts (because a checkout is atomic).

To create a branch without a sandbox use:

cvs rtag –b *branch-name*

Creating Promotion Levels

CVS uses branches to implement a promotion level. The difference is simply how you use them. Because the promotion level is usually only accessed by a QA manager there are typically two additional steps to setting up a promotion level once the branch is created:

Change the default access so that developers cannot merge changes into the promotion level or commit changes onto it.

Change the default locking more to unreserved. This can be done using the cvs update command on a checked out sandbox of the new promotion level:

```
cvs update -k-x
```

```
cvs commit
```

Promoting

CVS uses branches to implement a promotion level. The difference is simply how you use them. There are two ways to promote a revision from the Trunk (or a branch) to a promotion level: Move and Merge. With the Move method you will not keep a history of the previous versions also moved to that promotion level, with the merge method CVSNT will keep track of the history using Mergepoints and these will be viewable using CVSGraph in TortoiseCVS or WinCVS.

Move Method

On the Trunk or branch you can push a version to a promotion branch (promote) using the CVS *tag* command with the options *-F -B -b tag*.

This is unsafe to do with a normal branch because you will lose the changes that have occurred on the branch between it's creation and the promotion.

Merge Method

You will usually merge changes onto a promotion level by using bug identifiers. On a promotion branch versions are moved onto the promotion branch (promoted) using the CVS *update* command with the options *-B bug –j 1 –e newbug*. The *newbug* is a bug or task identifier for the tasks of promoting the bug to the new level.

Merging-in Branches

You can merge changes made on a branch into your working copy by giving the CVS command *update* with the *-j branchname* flag. With one *-j branchname* option it merges the changes made between the point where the branch was last merged and newest revision on that branch (into your working copy).

If you wish to revert to the older CVS behaviour of merging from the point the branch forked, specify the -b option.

If you are updating from an Unix CVS server of older CVSNT server that doesn't support merge points, then the merge will always be done from the branch point.

The -j stands for "join".

An example

Consider this revision tree:

```
+-----+    +-----+    +-----+    +-----+
! 1.1 !----! 1.2 !----! 1.3 !----! 1.4 !      <- The main trunk
+-----+    +-----+    +-----+    +-----+
                ! 
                !
                !   +---------+    +---------+

Branch R1fix -> +---! 1.2.2.1 !----! 1.2.2.2 !

                    +---------+    +---------+
```

The branch 1.2.2 has been given the tag (symbolic name) R1fix. The following example assumes that the module mod contains only one file, *m.c*.

```
$ cvs checkout mod                # Retrieve the latest revision, 1.4

$ cvs update -j R1fix m.c         # Merge all changes made on the branch,
                                  # i.e. the changes between revision 1.2
                                  # and 1.2.2.2, into your working copy
                                  # of the file.

$ cvs commit -m "Included R1fix" # Create revision 1.5.
```

A conflict can result from a merge operation. If that happens, you should resolve it before committing the new revision.

If your source files contain keywords, you might be getting more conflicts than strictly necessary.

The CVS command *checkout* also supports the *-j branchname* flag. The same effect as above could be achieved with this:

```
$ cvs checkout -j R1fix mod
$ cvs commit -m "Included R1fix"
```

It should be noted that the CVS command *update* with *-j tagname* will also work but may not produce the desired result.

Merging-in Branches using MergePoint

What is a mergepoint, and how does one use it?

Mergepoints help CVS find the common ancestor when trying to diff a file, which greatly reduces the effort required to merge in branches. It is automatically saved by CVS when you merge changes from one branch to another. Just make sure you commit after merging (before performing any other merges) and CVS will save the mergepoint field with the update.

Note that mergepoints are specific to CVSNT, and require that CVSNT is running on both client and server.

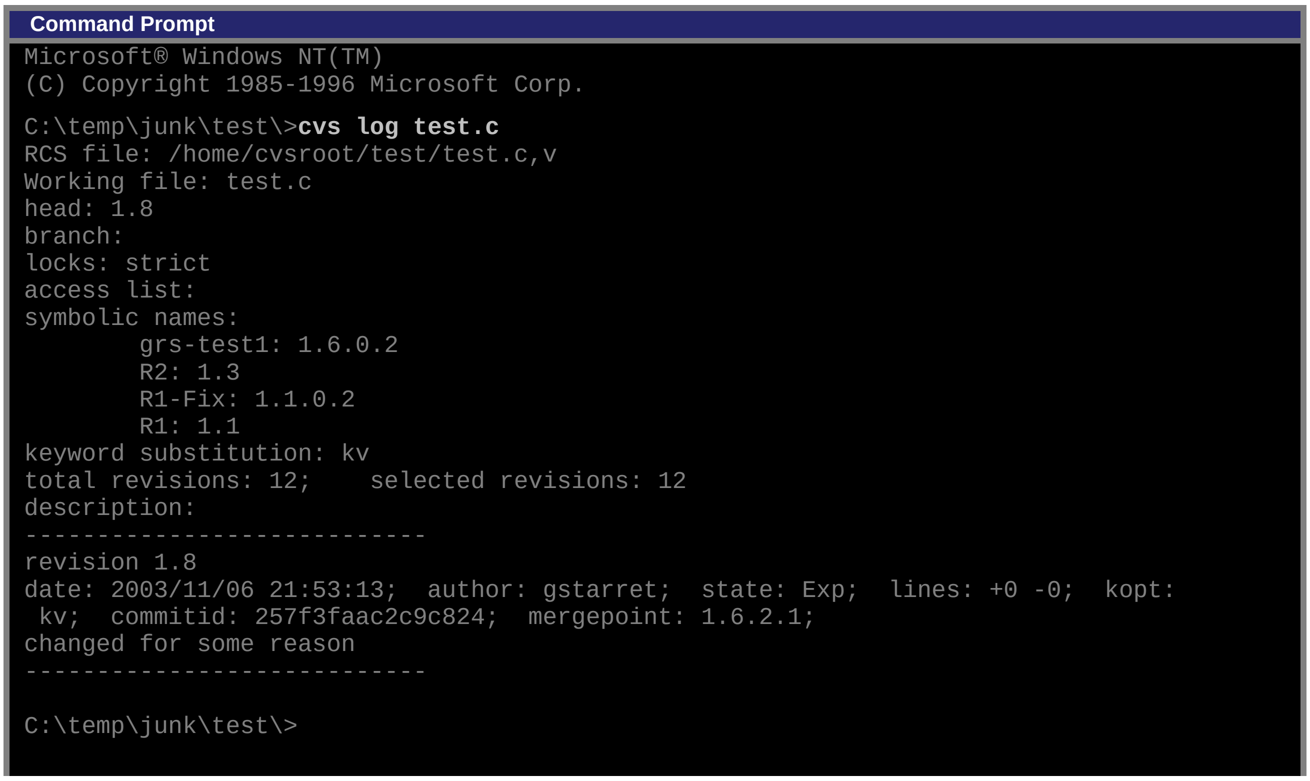
When you merge back and forth from dev branch to HEAD, it is a very simple operation--just specify the branch to merge from, and CVS takes care of the rest. If you read about merging in the regular CVS documentation, it looks like a big effort to *tag* before, *merge*, *commit*, retag... lots of tagging and remembering those tag names.

With mergepoint, merging is done simply by using:

cvs update -j branch-tag

Then you can correct any conflicts, test your integration, and commit without worrying about a lot of tagging operations.

You can see the mergepoint records in the output of the log command (look at rev 1.8, the last field before the comment):



```
Microsoft® Windows NT(TM)
(C) Copyright 1985-1996 Microsoft Corp.

C:\temp\junk\test\>cvs log test.c
RCS file: /home/cvsroot/test/test.c,v
Working file: test.c
head: 1.8
branch:
locks: strict
access list:
symbolic names:
        grs-test1: 1.6.0.2
        R2: 1.3
        R1-Fix: 1.1.0.2
        R1: 1.1
keyword substitution: kv
total revisions: 12;    selected revisions: 12
description:
----------------------------
revision 1.8
date: 2003/11/06 21:53:13;  author: gstarret;  state: Exp;  lines: +0 -0;  kopt:
 kv;  commitid: 257f3faac2c9c824;  mergepoint: 1.6.2.1;
changed for some reason
----------------------------

C:\temp\junk\test\>
```

That mergepoint is in the record from the commit after I performed the merge.