

UTILISATION DE CVS "Concurrent Version System"

Maurice Libes -libes@com.univ-mrs.fr

v0.1 - Mai 1999

Contents

1	Préambule	2
2	Introduction: qu'est ce que CVS?	2
2.1	Ce que CVS n'est pas et ne fait pas...	3
3	Pour commencer à utiliser CVS	3
3.1	Création d'un répertoire d'archivage : le Repository!	4
3.2	Pour créer une archive "from scratch" ("à partir de rien" ;-)	4
3.3	Problèmes des permissions d'écriture	6
4	Une session CVS standard : récupérer - modifier - mettre à jour	7
4.1	Comment récupérer le contenu d'une archive?	7
4.2	comment modifier et mettre à jour l'archive dans le Repository?	8
4.3	Comment rajouter-détruire des fichiers dans l'archive?	8
4.3.1	rajouter quelques fichiers	8
4.3.2	Comment rajouter toute une hiérarchie de répertoires	9
4.3.3	Comment détruire des fichiers d'une archive?	9
4.3.4	Comment détruire des répertoires entiers	10
4.3.5	Comment déplacer ou renommer des répertoires	10
4.4	Comment récupérer un fichier détruit par erreur?	10
4.5	Comment récupérer une version antérieure particulière ?	11
4.5.1	extraction avec un label ("sticky tag")	11
4.5.2	extraction avec un numéro de version	12
5	La gestion des versions par CVS	12
5.1	l'incrémentatation automatique des numéros de version	12
5.2	forcer un numéro de version	12
5.3	donner un label ("tag") à une certaine révision	13
6	Les différentes branches d'un développement	13
6.1	Pour créer une branche nouvelle de développement	14
6.1.1	à partir de la copie locale	14

6.1.2	à partir d'une ancienne version	14
6.2	Pour accéder à une branche de développement	15
6.3	La Fusion des branches ("merging")	15
6.4	Pour détruire une branche	15
7	Comment travailler à plusieurs sur une archive ?	15
7.1	Philosophie du développement à plusieurs avec CVS	16
7.2	Sur quelle version travaille t'on? mettre les fichiers à jour	16
7.3	gestion des conflits	17
7.4	Consultation de l'historique d'un projet - gestion des événements -supervision d'un projet . .	18
7.4.1	consultation des messages de trace de commit	18
7.4.2	consultation de l'historique d'un projet	18
7.5	consigner les événements	18
7.6	être averti des tentatives de modifications par "les autres"	18
7.7	Savoir qui travaille sur les fichiers de l'archive?	19
7.8	informer les autres développeurs qu'on travaille sur un fichier	20
8	Utiliser CVS en client-serveur à travers un réseau	20
9	Lexique	20
10	Quick Reference	21
11	Joli schéma	22

1 Préambule

Ce document se veut être une aide à l'utilisation de CVS. C'est un document original que j'ai écrit avec de larges emprunts au document "Version Management with CVS" de Per Cederqvist et al. mais qui n'en est pas une traduction littérale. Bien sur ce document ne prétend pas être aussi complet que la documentation de base présente dans /usr/doc/cvs-1.10.6/. Je ne suis pas un utilisateur spécialiste de CVS, je me suis intéressé aux possibilités de développement offertes par cet outil de gestion de versions, afin de prévoir son utilisation sur de gros codes de développement dans mon laboratoire. Le lecteur se sentira libre de me contacter pour apporter toute correction d'erreurs ou d'imprécisions qui se seraient subtilement cachées dans le texte.

2 Introduction: qu'est ce que CVS?

CVS est un système logiciel permettant de gérer et suivre l'évolution des différentes versions qui jalonnent le développement d'un document informatisé... Document qui peut être bien sur un programme informatique écrit en "C", Fortran, Perl, Tcl/Tk ou dans n'importe lequel des langages, mais encore un simple document de texte tel qu'un article ou un livre qui évolue peu à peu au fil du temps. CVS permet de gérer dans le temps chaque modification apportée à un ensemble de documents informatiques. CVS permet alors de suivre

l'évolution et l'historique du projet, comparer des versions, uniformiser des développements concurrents faits par d'autres développeurs etc...

Quel développeur ne s'est pas dit un jour "je comprends pas...? j'ai rien touché et ça marche plus ! " ;-) En analysant les modifications portées entre 2 versions, CVS peut permettre de ne plus jamais se dire cela en écrivant un programme.

2.1 Ce que CVS n'est pas et ne fait pas...

Attention tout de même à ne pas croire au miracle! CVS est un outil de gestion de version de documents! Il ne remplace pas un "chef de projet" qui décide d'une politique de gestion de l'évolution des versions comme:

- Quand mettre à jour les programmes dans l'archive de CVS?
- Comment gérer les conflits de développement entre plusieurs développeurs?
- doit-on avertir les autres développeurs qu'on travaille sur une version "x" d'un programme?
- etc...

Comme le dit l'auteur de CVS (en anglais dans le texte, chacun traduira à sa manière):

- "CVS is not a substitute for management" et
- "CVS is not a substitute for developer communication"

Moralité : "...il faut jouer collectif au niveau du mental (A. Jacquet)"

3 Pour commencer à utiliser CVS

Le principe de base de CVS tient en 3 points :

1. un répertoire central, le "Repository", contient une copie complète de tous les fichiers d'un projet. Au fil du temps, on archive dans le "Repository" toutes les modifications et évolutions apportées au projet.
 2. pour travailler, chaque utilisateur extrait du "Repository" une copie locale de ce projet, et travaille et modifie cette copie locale (le Repository n'est pas impliqué dans le travail local).
 3. Lorsque une version devient stable ou intéressante, et qu'elle mérite d'être sauvegardée, l'utilisateur choisit de recopier sa version modifiée localement, dans le "Repository" central.
- Ainsi, CVS est conçu pour permettre à plusieurs utilisateurs de travailler en parallèle sur un même projet! Chaque développeur va en effet travailler sur une copie **personnelle et locale** du projet. Chaque développeur peut en outre travailler sur des versions différentes d'un projet, pouvant de pas être au même stade de développement. Un utilisateur U1 peut travailler sur la version 1.6 du module a.c pendant que l'utilisateur U2 travaillera sur la version 1.5 de ce même module ! CVS possède tous les outils pour gérer ces différents états de développement et les uniformiser, quand cela devient nécessaire.

On verra comment CVS gère l'historique de l'évolution des versions de même que les conflits résultant d'un état de développement hétérogène entre différents utilisateurs.

Dans tous les cas CVS représente une aide au développement d'un projet grâce aux multiples commandes d'administration et de gestion du projet qui permettent de savoir quand un fichier a été modifié, par qui? quelles ont été les modifications apportées etc...

3.1 Création d'un répertoire d'archivage : le Repository!

CVS stocke tous ses fichiers de travail dans un répertoire central appelé le "Repository". Il faut donc en tout premier lieu créer ce répertoire spécial, dont CVS va se servir pour archiver les projets et contrôler leurs évolutions . En pratique, le Repository sera un ensemble de répertoires liés à l'administration et à la gestion des archives CVS.

La toute première commande nécessaire pour utiliser CVS, et créer le Repository, est donc :

```
$ cvs -d $CVSROOT init
```

où:

-d

cette option commune à toutes les commandes CVS indique le chemin vers le repository init crée la structure initiale du repository Dans cet exemple -d est superflu puisque CVS se serait basé sur la valeur de la variable CVSROOT (identique à l'option -d)

init

est l'option cvs pour créer un Repository initial !

Toutes les commandes CVS nécessitent de connaître l'emplacement du Repository. Pour cela CVS s'appuie sur une variable d'environnement appelée **CVSROOT** qui indique quel sera le lieu d'archivage des projets.

Pour utiliser CVS, il faut donc initialiser la variable d'environnement **CVSROOT** avec le répertoire qui contiendra tous les fichiers d'administration des archives à gérer.

```
export CVSROOT=/usr/local/CVS ou export CVSROOT=/home/cvs
```

Après avoir lancé cette commande, on aura créé la hiérarchie de répertoires suivants:

```
$ ls -l /home/cvs
```

```
total 1
```

```
drwxrwxr-x 2 momo cvs 1024 May 16 16:30 CVSROOT
```

```
$ ls /home/cvs/CVSROOT/ checkoutlist config,v history notify taginfo,v checkoutlist,v cvswrappers  
loginfo notify,v verifymsg commitinfo cvswrappers,v loginfo,v rcsinfo verifymsg,v  
commitinfo,v editinfo modules rcsinfo,v config editinfo,v modules,v taginfo
```

Je ne m'étendrai pas sur le contenu et la fonction de ces divers répertoires! (Si quelqu'un veut l'écrire ça fera la version 0.2 ;-)

NB : En pratique on n'a peu de choses à aller faire dans les répertoires du Repository. On peut toutefois comme on le verra plus loin modifier les options par défaut de CVS et aller inscrire certaines directives dans les fichiers d'administrations. Un Repository pourra contenir une ou plusieurs archives de projets différents, gérés par des chefs de projets différents... On peut également selon les besoins créer différents Repositoires sur une machine.

3.2 Pour créer une archive "from scratch" ("à partir de rien" ;-)

On a désormais initialisé la variable d'environnement CVSROOT et on a créé un Repository dans /home/cvs (par exemple... mais vous pouvez le créer ou bon vous semble)

On veut désormais faire gérer les différentes versions relatives à l'évolution d'un projet, par CVS. La première étape est donc d'intégrer pour la toute première fois un ensemble de fichiers relatifs à un projet, dans une archive du Repository.

Pour cela, il est nécessaire de se placer DANS le répertoire contenant les fichiers d'un projet (textes ou programmes)

```
$ cd essaicvs
```

et taper la commande:

```
$ cvs import -m "Projet de Doc cvs 1.0" essaicvs momo start
```

A l'issue de cette commande toute la hiérarchie de fichiers et répertoires sous jacents est **récurivement** intégrée à une nouvelle archive située dans \$CVSROOT

dans cette commande ci dessus:

-m

est l'option permettant de donner une chaîne de caractères identifiant l'état du développement. En l'absence de cette option, CVS vous place dans un éditeur de texte, afin de taper la chaîne de caractères qui vous convient.

essaicvs

est le nom de l'archive qui sera placée dans le Repository (\$CVSROOT/essaicvs)

momo

est ce que la doc appelle le "vendor tag". Un label "libre" identifiant l'origine des sources qu'on intègre.

start

est un exemple de "release tag" (voir plus bas) qui est un label marquant le départ de la constitution de l'archive doccvs dans le Repository. On aurait pu mettre autre chose que "start" comme label (qu'il ne faut pas considérer ici comme un mot réservé de CVS!)

On voit ci dessous dans le Repository, le répertoire "essaicvs" qui correspond au nom qu'on a donné dans la ligne de commande ci dessus "cvs import". On aurait

```
$ ls -l /home/cvs/essaicvs/
```

```
total 18
```

```
-r-r-r- 1 momo cvs 13412 May 16 18:45 doccvs.lyx,v
```

```
-r-r-r- 1 momo cvs 2481 May 16 16:32 doccvs.momo,v
```

A l'issue de l'intégration d'un projet dans le Repository, on peut tout à fait détruire la version originale du projet que l'on vient d'intégrer ou renommer le répertoire original afin qu'il n'interfère pas avec CVS.

```
$ cd ..
```

```
$ rm -fr essaicvs
```

ou bien

```
$ mv essaicvs essaicvs.orig
```

(Les plus "paranos" et méfiants auront fait une sauvegarde préalable)

3.3 Problèmes des permissions d'écriture

Achtung ! ;-) ...Il est nécessaire que les différents développeurs qui vont travailler sur le projet "P" puissent tous avoir accès en écriture au Repository. Ainsi si on le crée dans un répertoire protégé comme /usr/local/cvsroot, par défaut, seul le (super) user "root" aura les permissions d'écriture pour créer le Repository ainsi que les archives qui vont le composer. Cela est peu compatible avec des groupes d'utilisateurs ordinaires qui travailleraient sur un projet commun nécessitant des permissions d'écriture partagées.

Aussi, il est sans doute préférable :

1. de créer un Repository dans un endroit accessible à plusieurs utilisateurs travaillant sur un même projet. (ou de créer autant de repositories qu'il y a de projets à gérer).
2. de confier la création d'un repository pour un projet "P" à un utilisateur "U" chef de projet du projet "P"
3. de créer un groupe Unix rassemblant tous les acteurs du projet P (disons un groupe "grpcvs" par exemple)
4. de donner les permissions d'écriture sur le projet "P" créé aux membres de ce groupe particulier "grpcvs" par exemple

Ce qui se traduit en Unix par:

- création d'un user "cvsuser" dont le HOME directory /home/cvs par exemple, servira de Repository
\$ adduser cvsuser -d /home/cvs
- création d'un nouveau groupe Unix dans /etc/group "cvsgrp" **\$ groupadd cvsgrp**
- faire appartenir le Repository au groupe de travail "cvsgrp", et donner la permission d'écriture de ce Repository au groupe de travail "cvsgrp" **\$ chgrp -R cvsgrp /home/cvs**
\$ chmod -R 770 /home/cvs
- rajouter les utilisateurs momo et zaza , qui vont travailler ensemble sur ce même projet dans le groupe "cvsgrp"

cvsgrp:x:504:cvs,momo,zaza (ligne extraite de /etc/group)

- définir la variable CVSROOT contenant le lieu de création du Repository pour chacun des utilisateurs du groupe de travail (dans leur ~/.bashrc par exemple) **\$ export CVSROOT=/home/cvs**
- Créer effectivement le Repository dans /home/cvs **\$ cvs -d \$CVSROOT init**
- Je suis le User "momo" chef de projet du projet de documentation cvs en français ;-). Je vais créer l'archive d'un projet "doccvs" dans le Repository. Il est INDISPENSABLE DE SE PLACER DANS le répertoire de travail que l'on veut archiver dans le Repository!! **\$ cd /home/momo/doccvs**
- Créer l'archive du projet "doccvs" dans le Repository **\$ cvs import -m "Projet de Doc cvs 1.0" doccvs momo start**

```
$ ls -l /home/cvs/
```

```
total 2
```

```
drwxrwxr-x 2 momo cvs 1024 May 16 16:30 CVSROOT
```

```
drwxrwxr-x 2 momo cvs 1024 May 16 16:32 doccvs
```

A l'issue de cette création d'archive, on retrouvera les fichiers composant le répertoire de travail doccvs dans la hiérarchie "doccvs" du Repository:

```
$ ls -l /home/cvs/doccvs
```

```
total 21
```

```
-r--r--r-- 1 momo cvs 225 May 16 23:33 Afaire,v
```

```
-r--r--r-- 1 momo cvs 16256 May 16 23:38 doccvs.lyx,v
```

```
-r--r--r-- 1 momo cvs 2481 May 16 16:32 doccvs.momo,v
```

L'archive est créée! CVS gère désormais toutes modification des fichiers du répertoire courant "doccvs".

4 Une session CVS standard : récupérer - modifier - mettre à jour

4.1 Comment récupérer le contenu d'une archive?

Après avoir créé un Repository central et y avoir déposé tous les fichiers d'un projet initial par la commande **\$ cvs import -m "Projet de Doc cvs 1.0" doccvs momo start** on a créé dans le Repository, l'archive ayant pour nom "doccvs"

Une fois que cela est fait on peut (et on doit) totalement détruire les répertoires du projet original (**\$ rm -fr ~/monbeauprojet** (...oui je sais, ça fait peur)) afin de s'en remettre à la gestion par CVS. Evidemment on aura sauvegardé la version originale quelque part, au cas où!... mais la distribution originale ne devra plus du tout servir et être utilisée. Tout le travail ultérieur est composé de l'extraction des fichiers à partir du Repository, puis de l'intégration des versions modifiées!

Il est **nécessaire** par la suite, d'extraire ce projet du Repository, afin de pouvoir travailler **sur une copie locale** de tous les fichiers de ce projet.

L'extraction d'un ensemble de fichiers du Repository central se fait par la commande :

```
$ cvs checkout
```

à laquelle il faut donner comme argument tout ou partie du projet que l'on veut récupérer localement.

```
$ cvs checkout [sans argument]
```

```
cvs [checkout aborted]: must specify at least one module or directory
```

Se placer au dessus du répertoire que l'on veut créer.

(**crée le répertoire local doccvs et y place le fichier monfichier.txt, extrait du repository**)

```
$ cd ..
```

```
$ cvs checkout doccvs/monfichier.txt
```

ou, extraction de la totalité

```
$ cd ..
```

```
$ cv checkout doccvs
```

Attention pour extraire un projet complet (toute une hiérarchie de répertoires) du repository il faut se placer à l'**extérieur** du répertoire que l'on souhaite récupérer. C'est à dire que pour récupérer et extraire du Repository le projet "monprojet", il faudra auparavant quitter le répertoire courant "monprojet" (cd ..) , sans quoi l'extraction du Repository va créer le répertoire "monprojet" dans le répertoire "monprojet", et on aura les répertoires ~/monprojet/monprojet

Par défaut, la commande `cvs checkout` extrait du Repository la dernière version de l'archive (version de tête).

4.2 comment modifier et mettre à jour l'archive dans le Repository?

Je résume les quelques étapes que nous avons vu:

- l'archive a été créée dans le Repository (`cvs import ...`)
- l'archive doit avoir été extraite sous forme d'une copie locale, dans un répertoire de travail local (`cvs checkout`)
- Après avoir récupéré le projet dans un répertoire local, on y travaille dessus... en travaillant on aura modifié certains fichiers, on en aura rajouté ou détruit (`cvs add ; cvs remove`)
- Au passage, une commande "`cvs status`" que l'on verra plus loin, permet de savoir si quelque chose a bougé entre la version présente dans l'archive et la copie locale

```
cvs status doccvs.lyx =====  
File: doccvs.lyx Status: <bf>Locally ModifiedWorking revision: 2.22 Tue May 25 20:59:25 1999 Repository
```

Lorsqu'on est satisfait d'une version d'un fichier sur lequel on a travaillé, on range tout ou partie du projet dans le Repository par la commande

\$ `cvs commit` (range tout le projet en cours, récursivement, dans le Repository)

ou

\$ `cvs commit <le_fichier_dont_on_est_satisfait.c>` (range uniquement le fichier mentionné)

```
Checking in doccvs.lyx; /home/cvs/essai cvs/doccvs.lyx,v <- doccvs.lyx new revision: 2.23; previous revision: 2.22 done
```

Le numéro de version est alors incrémenté d'une unité dans le Repository! Une version qui avait le numéro de tête 2.3 par exemple, aura le numéro 2.4 après l'intégration dans le Repository par la commande `cvs commit`.

L'option `-m` de la commande `cvs commit` permet de placer un commentaire attaché à la version que l'on vient de ranger dans le Repository

Si on ne donne pas l'option `-m`, `cvs` lance alors un éditeur de texte (`vi` par défaut) permettant d'écrire une description sommaire de quelques lignes, correspondant à la nouvelle version que l'on veut archiver dans le Repository.

....

[...écrire ici une description de la version...]

```
CVS:----- CVS: Enter Log. Lines beginning with 'CVS:
```

4.3 Comment rajouter-détruire des fichiers dans l'archive?

4.3.1 rajouter quelques fichiers

C'est bien beau, j'ai intégré tout un ensemble de fichiers d'un projet à une archive CVS, mais au cours de mon travail je dois rajouter de nouveaux fichiers, et même de nouveaux répertoires.

Qu'à cela n'Etienne! voici ce qu'il faut faire :

```
$ cvs add fic1 fic2
```

puis

```
$ cvs commit fic1 fic2
```

Les fichiers fic1 et fic2 doivent évidemment exister au préalable dans le répertoire courant. En pratique, les fichiers "fic1" et "fic2" ne sont effectivement inscrits dans le Repository qu'après que la commande la commande **cvs commit** ait été lancée. Il faut donc faire un cvs commit pour mettre effectivement à jour l'archive dans le Repository. Ce qui est clairement indiqué par les messages d'information des commandes cvs.

Exemple si je crée le fichier "Afaire" et que je veux l'intégrer dans le Repository

```
$cvs add Afaire
```

```
cvs add: scheduling file 'Afaire' for addition cvs add:
```

```
use 'cvs commit' to add this file permanently
```

```
$ cvs commit -m "rajout fichier Afaire" Afaire
```

```
RCS file: /home/cvs/essaicvs/Afaire,v done
```

```
Checking in Afaire; /home/cvs/essaicvs/Afaire,v <- Afaire initial revision: 1.1 done
```

4.3.2 Comment rajouter toute une hiérarchie de répertoires

Je n'ai pas un seul fichier à rajouter mais tout une hiérarchie de fichiers dans un sous répertoire!

Pour rajouter une branche complète de répertoire contenant des fichiers à l'archive, il est préférable d'utiliser la commande **cvs import** comme dans le cas de la création initiale de l'archive dans le Repository.

```
$ mkdir Rep1
```

```
$ touch Rep1/fic1rep3
```

```
$ cvs import -m "ajout de Rep1" Rep1 momo start
```

```
I Rep1/CVS N Rep1//doccvs.lyx
```

```
N Rep1/Afaire I Rep1//doccvs.lyx~
```

```
N Rep1/doccvs.momo cvs import: Importing /home/cvs/Rep1
```

```
N Rep1/Rep1/fic1rep1
```

```
N Rep1/Rep1/fic2rep1
```

```
No conflicts created by this import
```

```
$ cvs add -m "ajout de Newrep3/fic1rep3 " Newrep3/fic1rep3
```

```
$ cvs commit -m "ajout Newrep3 " Newrep3
```

4.3.3 Comment détruire des fichiers d'une archive?

- il faut tout d'abord détruire le fichier **dans** le répertoire de travail local

```
$ rm fic1
```

- puis indiquer à CVS que l'on veut supprimer un fichier d'une archive par la commande "cvs remove". Le fichier est marqué pour être détruit, mais n'est pas encore détruit de l'archive tant que l'on a pas effectué la commande "cvs commit"
- A ce stade, si on s'est trompé de fichier à détruire (mettons aa.c) , on peut encore le récupérer par la commande "cvs add aa.c" car il n'a pas été détruit de l'archive.

```
$ cvs remove fic1
```

```
cvs remove: scheduling 'fic1' for removal
```

```
cvs remove: use 'cvs commit' to remove this file permanently
```

- puis le détruire effectivement de l'archive

```
$ cvs commit -m "destruction fic1"
```

```
cvs commit: Examining .
```

```
Removing fic1; /home/cvs/essai/cvs/fic1,v <- fic1 new revision: delete; previous revision: 1.1 done
```

4.3.4 Comment détruire des répertoires entiers

On veut détruire un certain répertoire de notre répertoire de travail, et le faire disparaître du Repository! mais on veut garder la trace de ce répertoire dans le Repository (en tant que version "n").

Dans ce cas il faut :

- détruire tous les fichiers du répertoire concerné, SANS détruire le répertoire lui-même, du répertoire local de travail. (soit `$rm -f mon_rep_bidon/*`)
- invoquer l'option `-P` des commandes `cvs update` , `cvs checkout`

Cette option `-P` provoque la destruction des répertoires vides dans le répertoire de travail.

4.3.5 Comment déplacer ou renommer des répertoires

[page 51]

4.4 Comment récupérer un fichier détruit par erreur?

```
$ rm truc.c
```

Oops! je me suis trompé... à ce stade, on a détruit un fichier du file system Unix, mais l'archive CVS du projet n'est absolument pas modifiée. Aussi il suffit de mettre à jour le répertoire courant avec la commande "cvs update" pour retrouver le fichier détruit ... mmâââggic!. Le fichier détruit par erreur "truc.c" sera extrait de l'archive pour mettre à jour le répertoire courant.

```
$ cvs update
```

Si on a détruit le fichier par erreur, **après** la phase "cvs remove", il suffit de lancer tout de suite la commande "cvs add" pour récupérer l'erreur

exemple:

```
$rm doccvs.momo
```

```
$ cvs remove doccvs.momo
```

```
cvs remove: scheduling 'doccvs.momo' for removal
```

```
cvs remove: use 'cvs commit' to remove this file permanently
```

```
$ cvs add doccvs.momo
```

```
U doccvs.momo cvs add: doccvs.momo, version 1.1.1.1, resurrected
```

4.5 Comment récupérer une version antérieure particulière ?

4.5.1 extraction avec un label ("sticky tag")

Par défaut la version extraite du repository par la commande "cvs checkout" est la dernière archivée (version de tête). Cependant, on peut extraire de l'archive une version antérieure particulière, (pas nécessairement la dernière en cours) en spécifiant un label ("tag") de la version souhaitée. Il aura fallu bien évidemment mettre en place ce label à un certain moment du développement, par la commande "cvs tag" (voir 5.3).

Ainsi, la commande

```
$ cvs checkout -r "mardi-1805" essaicvs
```

```
cvs checkout: Updating essaicvs
```

```
U essaicvs/doccvs.lyx
```

```
cvs checkout: Updating essaicvs/Rep1
```

extraira du Repository la révision de l'archive "essaicvs" dont le tag qui a été inscrit grâce à la commande cvs tag, (voir plus bas 5.3) est "mardi-1805".

On aura donc extrait dans son répertoire local une version antérieure du projet qui n'EST PAS la dernière version courante (version de tête)!

```
$ cvs status
```

```
cvs status: Examining essaicvs
```

```
=====
```

```
File: doccvs.lyx Status: Up-to-date
```

```
Working revision: 2.6 Mon May 17 22:32:27 1999
```

```
Repository revision: 2.6 /home/cvs/essaicvs/doccvs.lyx,v
```

```
Sticky Tag: mardi-1805 (revision: 2.6)
```

```
Sticky Date: (none)
```

```
Sticky Options: (none)
```

La version extraite avec son "sticky tag" **restera dans le répertoire local tant que on n'aura pas mis à jour l'archive avec la commande "cvs update -A"**

La commande **cvs update -A** permet de retrouver la version de tête de l'archive, en oubliant les sticky tag précédemment pris en compte..

```
$ cvs update -A
```

```
cvs update: Updating .
```

```
U Afaire U doccvs.lyx
```

```
U doccvs.momo
```

U ficzaza

Le but d'un label (sticky tag) de version est d'identifier un certain état de développement. On se sert des "sticky tag" (label collant) lorsqu'on veut extraire une certaine version de l'archive, et qu'on ne veut pas que celle ci soit modifiée par des mises à jour "cvs update" ultérieurs. Une version récupérée avec un sticky tag est en effet **insensible** à des mises à jour par "cvs update". Il faut faire la commande "cvs update -A" pour retrouver la dernière version de tête de l'archive extraite du Repository. Avec un sticky tag, le développement local est alors isolé de toute modification faite dans le Repository par d'autres collègues développeurs.

4.5.2 extraction avec un numéro de version

Hormis les labels de version, on peut également extraire une certaine version du Repository en spécifiant un numéro de version avec l'option -p de la commande update

Ainsi

```
$ cvs update -p -r 2.0 doccvs.lyx > doccvs.lyx
```

ressortira du Repository la version 2.0 du document doccvs.lyx

5 La gestion des versions par CVS

5.1 l'incrémentation automatique des numéros de version

Lors de chaque intégration d'une nouvelle révision par la commande "cvs commit" d'un document dans le Repository, CVS attribue automatiquement un nouveau numéro de version, incrémenté d'une unité par rapport à la version précédente.

La version de base a le numéro 1.1. A partir de celle ci, chaque nouvelle révision aura successivement les numéros 1.2 puis 1.3, puis 1.4 etc.... CVS conserve donc le premier numéro et incrémente le second (tant qu'on n'a pas forcé un changement de numéro de version majeur).

Si on ajoute un nouveau fichier dans l'archive (cvs add) ce fichier aura pour numéro de version principale x.y. Où "y" sera toujours égal à 1, et "x" sera le numéro de version le plus élevé dans l'archive. Par exemple, si lorsque j'ajoute le nouveau fichier aa.c , et que un des fichiers de l'archive a le numéro 4.36, alors le numéro de version de aa.c sera 4.1

5.2 forcer un numéro de version

Dans la majeure partie des cas, l'utilisateur n'a pas vraiment à se soucier de l'attribution des numéros de versions que réalise CVS.

Il peut arriver toutefois que, dans un certain état de développement, ou de stabilité, l'on veuille forcer un numéro de version dans l'archive. Cela se fait avec l'option "-r" de la commande cvs commit/

Ainsi

```
$ cvs commit -r 2.0
```

permettra de forcer les numéros de révision de tous les fichiers de l'archive à 2.0 A partir de cette nouvelle base, CVS incrémentera automatiquement les nouvelles révisions comme indiqué ci dessus : 2.0, 2.1, 2.2 etc...

5.3 donner un label ("tag") à une certaine révision

Enfin, de la même manière que l'on peut vouloir, comme ci dessus, forcer un numéro de version dans un certain état de développement, on peut vouloir également donner un label à un certain état de développement du projet.

Le fait de placer un label sur une archive au cours de son développement est comme prendre une "photo instantanée" du projet à un temps "t" de son cycle de développement. Le label placé sur un projet, permettra de retrouver (extraire) quelques temps plus tard tous les fichiers dans l'état où ils étaient lorsqu'on à placé le label!

Cela se fait avec la commande

```
$ cvs tag "le label" <fichier>
```

exemple:

```
$ cvs tag "mardi-1805" doccvs.lyx
```

```
T doccvs.lyx
```

```
$ cvs status -v doccvs
```

```
=====
```

```
File: doccvs.lyx Status: Up-to-date
```

```
Working revision: 2.6 Mon May 17 22:32:33 1999
```

```
Repository revision: 2.6 /home/cvs/essaicvs/doccvs.lyx,v
```

```
Sticky Tag: (none)
```

```
Sticky Date: (none)
```

```
Sticky Options: (none)
```

```
Existing Tags:
```

```
mardi-1805 (revision: 2.6)
```

```
start (revision: 1.1.1.2)
```

```
momo (branch: 1.1.1)
```

Il y a peu de raison de tagger un seul fichier particulier de l'archive. En général on préfère tagger une archive complète dans un état stratégique du développement.

6 Les différentes branches d'un développement

Comme on le sait désormais, CVS permet d'enregistrer et gérer des versions successives d'un développement. Par défaut ces versions sont, gérées (peu importe comment) dans le Repository, dans une hiérarchie principale appelée le "tronc principal" ("main trunk"). Les révisions successives sont gérées séquentiellement à partir d'une version de base 1.1, avec un numéro de révision qui s'incrémente de 1 lors de chaque nouvelle intégration dans le Repository.

Cependant au cours du développement d'un projet, il arrive parfois que l'on doive (ou que l'on veuille) s'écarter des versions contenues dans le tronc principal. On crée alors une branche qui va représenter une différence dans le développement par rapport au tronc principal. Les raisons peuvent en être multiples:

- correction d'un bug trouvé dans une vieille version : imaginons que la version courante d'un programme soit la 2.10 en cours de développement et que l'on découvre un bug dans la version 1.5 en cours d'exploitation... Pour corriger le bug de la version 1.5 on peut alors créer une branche 1.5.1 et corriger le bug dans cette

branche, sans perturber le développement en cours de la 2.10. Bien sur la correction du bug sera intégrée par la suite à la 2.10.

- développement concurrentiel en parallèle par 2 équipes de développeurs

Le développement dans les branches du projet n'apparaît pas dans la version de tête. Branches et tronc principal sont indépendants. Toutefois, on verra plus loin comment les modifications effectuées dans une branche peuvent par la suite être intégrées au tronc principal.

6.1 Pour créer une branche nouvelle de développement

6.1.1 à partir de la copie locale

c'est l'option tag -b suivi d'un nom symbolique (label du tag) qui crée une branche.

\$ cvs tag -b correction-1.5

La commande crée donc une branche nouvelle 1.5.1 (dans cet exemple) dans le Repository et ce, à partir des versions présentes dans le répertoire de travail courant! . Cette nouvelle branche porte le label (tag) "correction-1.5". La commande doit être tapée dans le répertoire courant qui contient la copie locale de l'archive.

Attention : ce n'est pas parcequ'on vient de créer une nouvelle branche, que les fichiers du répertoire courant (copie locale) vont nécessairement appartenir à cette nouvelle branche dans le Repository!. Une branche est créée dans le Repository, PAS dans la copie locale de travail.

\$ cvs commit -m "après branche correction 1.5"

cvs commit: Examining .

cvs commit: Examining Rep1

Checking in doccvs.lyx; /home/cvs/essaicvs/doccvs.lyx,v <- doccvs.lyx

new revision: 2.16; previous revision: 2.15 done

Ainsi dans cet exemple, après avoir créé une branche 1.5.1, un nouveau "cvs commit" dans le répertoire courant continue à intégrer l'archive dans le tronc principal 2.16

6.1.2 à partir d'une ancienne version

On peut créer une branche à partir d'une ancienne version (et non plus à partir de la copie locale) déjà intégrée dans le Repository, avec la commande "cvs rtag". Il faut en outre mentionner dans la ligne de commande le tag (label) de l'ancienne version à partir de laquelle on veut créer la branche!

Ainsi

\$ cvs rtag -b -r mardi-1805 modif-mardi-1805 essaicvs

où:

rtag -b option pour créer une nouvelle branche à partir d'une ancienne version

-r mardi-1805 : indique que la nouvelle branche sera créée à partir de la version qui porte le label "mardi-1805"

modif-mardi-1805 est le label identifiant la nouvelle branche créée

essaicvs est le nom du module (archive) dans laquelle on veut créer la branche

Je vais le dire autrement : avec cette commande, une nouvelle branche portant le label "modif-mardi-1805" sera créée dans l'archive essaicvs à partir de la version dont le label est "mardi-1805"

6.2 Pour accéder à une branche de développement

On peut récupérer une branche particulière de 2 manières différentes:

i) Soit en la retirant de l'archive (Repository) par la commande

```
$ cvs checkout -r <label branche> <module>
```

ainsi

```
$ cvs checkout -r modif-mardi-1805 essaicvs
```

va extraire du Repository les fichiers de la branche portant le label "modif-mardi-1805" de l'archive "essaicvs"

ii) Soit en demandant une mise à jour de la copie locale

```
$ cd essaicvs
```

```
$ cvs update -r modif-mardi-1805
```

Une fois qu'on a extrait une branche dans le répertoire local, le répertoire local **reste "attaché" à cette branche là...** tant qu'on n'a pas dit explicitement qu'on voulait rejoindre le tronc principal ou une autre branche! Cela signifie que toute commande `cvs commit` qui vont suivre vont mettre à jour les versions de fichiers **DANS** la branche de l'archive qu'on a précédemment extraite. Le tronc principal ou les autres branches ne'en sont pas affectées! (chaque branche et le tronc sont indépendants)

Pour savoir sur quelle branche on se trouve (c'est à dire de quelle branche proviennent les fichiers qui sont dans mon répertoire local), on peut utiliser la commande

```
$cvs status
```

```
File: tga-liste.c Status: Up-to-date
```

```
<code>Working revision: 2.1 Wed Sep 22 21:49:53 1999 Repository revision: 2.1 /home/cvs/tga-erosion/t
```

Il faut regarder la ligne Sticky tag pour voir sur quelle branche on se trouve (i.e de quelle branche provient le fichier présent dans mon répertoire local, que je suis en train de vérifier). Dans l'exemple ci dessus; la ligne Sticky Tag mentionne "none" , ce qui signifie que la version du fichier que je vérifie provient du tronc principal (j'aurais pu choisir une autre illustration)

6.3 La Fusion des branches ("merging")

6.4 Pour détruire une branche

La documentation originale de Cederqvist et al. ne mentionne pas cet aspect. Y a t'il intérêt à détruire une branche de développement? sans doute aucun!? La branche peut bien rester inexploitée dans le Repository. L'intérêt est sans doute de savoir qu'elle a existé, pas de l'effacer!? [si le lecteur a des idées la dessus, qu'il s'exprime!...]

7 Comment travailler à plusieurs sur une archive ?

Dans le développement en commun, si on 'y porte pas cas : "L'enfer ca peut rapidement être les autres"

- Comment récupérer chez moi la dernière version d'un projet mise à jour par mes collègues de travail? (`$cvs update`)

- Comment Gérard récupèra-t'il les gèniales modifications que j'ai faites sur les programmes buggés qu'il a écrit? (`$cvs update`)
- Comment saura t'il que j'ai fait des modifications et que j'en suis à la révision 1.19 alors qu'il débogue encore sa version 1.1? (`$cvs status $cvs log $cvs commit`)
- est ce que je ne vais pas écraser les modifications qu'il aura effectuées avant moi?
- et si je modifie les mêmes lignes de codes que Gérard..au même moment! comment se fait la gestion des conflits? maaaaan!!!.....

7.1 Philosophie du développement à plusieurs avec CVS

Quand on développe à plusieurs personnes, il peut y avoir des conflits! Deux personnes peuvent éditer et apporter des modifications au même fichier en même temps! Dans certains projets, il arrive que l'on veuille éviter cela. Deux solutions sont possibles pour éviter/résoudre ces conflits potentiels :

- verrouiller le fichier de travail (on parle d'extraction réservée "reserved checkout"). Cela signifie que seul le premier des développeur accède au fichier convoité. Une seule personne à la fois est autorisée à éditer et modifier le fichier! Pour réaliser un tel verrouillage avec CVS, il faut utiliser :
 - soit la commande **`cvs admin -l`**
 - soit l'ensemble des commandes **`cvs watch`** (voir plus bas) Dans ce cas de figure le second développeur se verra refuser l'accès au fichier en cours d'édition par un autre développeur.
- "ne pas verrouiller les fichiers de travail" (on parle "d'extraction non réservée" ou "Unreserved Checkout"). Ce choix est la philosophie de base par défaut de CVS! Dans ce cas de figure, tous les développeurs sont autorisés à accéder à un même fichier de l'archive en même temps! Ils récupèrent donc une copie de travail locale dans leur répertoire de travail! (jusque là, on ne fait du mal à personne). Par la suite, il peut arriver qu'un des développeur mette à jour l'archive et donc crée la version "n+1" (`$cvs commit`) pendant qu'une autre personne est en train d'éditer la version "n" et s'apprete à l'archiver. Dans ce cas les autres développeurs ne pourront pas mettre à jour l'archive avec leur version "n" modifiée! Ils devront utiliser les commandes CVS de mise à jour de leur répertoire local (`$cvs update`) pour amener leur répertoire de travail local en conformité avec l'état du Repository! En faisant `$cvs update` CVS mettra à jour leur copie locale en intégrant les différences entre leur version "n" et la version "n+1" dernièrement intégrée.

7.2 Sur quelle version travaille t'on? mettre les fichiers à jour

Il nous faut pouvoir tester si on a dans notre répertoire de travail la version la plus à jour de l'archive (version de tête) ?

La commande "**`cvs status`**" permet de comparer la version que l'on possède dans son répertoire de travail avec la version contenue dans l'archive! Elle peut être appliquée à un fichier particulier ou à l'archive complète.

Les diagnostics de cette commande `cvs status` peuvent être:

- Up-to-date : le répertoire de travail est en conformité avec la version de tête du Repository
- Locally modified : des fichiers du répertoire de travail ont été modifiés... ces modifications n'ont pas été intégrées au Repository

- Locally added : On a ajouté des fichiers dans le répertoire de travail..ceux ci n'ont pas encore été intégrés au Repository (`$cvs commit`)
- Locally Removed : (idem avec des fichiers détruits localement)
- Needs checkout : quelqu'un intégré une nouvelle version dans le Repository... il faut faire un `cvs update` pour mettre le répertoire local en conformité
- Needs patch : idem qu'au dessus, mais CVS extrait un patch pour mettre a jour les fichiers, plutot que les fichiers complets

```
$<bf> cvs status doccvs.lyx ===== File: doccvs.lyx Statu
```

A priori dans l'exemple ci dessus l'utilisateur U possède dans son répertoire de travail la version 1.6 du fichier `doccvs.lyx` alors que l'archive (le Repository) contient la version 1.7. Cette dernière version a sans doute été mise à jour par une tierce personne du groupe de travail. Il est donc nécessaire de mettre à jour sa version courante du répertoire de travail soit par la commande "`cvs checkout`" soit par la commande "`cvs update`"

Les modifications élaborée par un développeur NE SONT JAMAIS PERDUES! même si entre temps un autre développeur modifié le Repository. La commande `cvs update` va intégré (merge) les modifications apportées à la version de tête DANS mon fichier que je suis en train de modifier localement.

Ainsi, dans l'exemple ci dessous, quelqu'un à mis à jour le repository avec la version 2.48. Pendant ce temps j'étais en train de modifier la version 2.46 . Je ne peux pas mettre à jour (`cvs commit`) le Repository puisque ma copie locale n'est pas en conformité ave la version de tête. Je suis obligé de me mettre en conformité : `cvs update` va incorporer TOUS les CHANGEMENTS entre la version 2.46 et 2.48 DANS mon fichier de travail du répertoire local

```
$ cvs update cvs update: Updating . RCS file: /home/cvs/essaicvs/doccvs.lyx,v
retrieving revision 2.46 retrieving revision 2.48 <bf>Merging differences between 2.46 and 2.48 into d
```

```
$ cvs checkout essaicvs
```

```
cvs checkout: Updating essaicvs
```

```
U essaicvs/Afaire
```

```
U essaicvs/doccvs.lyx
```

```
U essaicvs/doccvs.momo
```

Si on n'avait pas obtenu les dernières versions des sources dans son répertoire courant, aurait-il été possible de mettre à jour l'archive avec une version courante [1.6] inférieure à la version de l'archive CVS [1.7] ? la réponse est NON !

```
$cvs commit (pour intégrer la v1.6, sans avoir fait l'update préalable)
```

```
cvs commit: Examining . cvs commit: Up-to-date check failed for 'doccvs.lyx'
```

```
cvs [commit aborted]: correct above errors first!
```

7.3 gestion des conflits

7.4 Consultation de l'historique d'un projet - gestion des événements - supervision d'un projet

7.4.1 consultation des messages de trace de commit

Quand on intègre une nouvelle version dans le Repository, par la commande **cvs commit -m "message"**, CVS demande un message. Ce message de trace peut être consulté après coup par la commande

\$ cvs log [fichier]

Cet affichage permet de consulter les diverses intégrations dans le Repository, couplé à un message d'explication.

7.4.2 consultation de l'historique d'un projet

Un fichier d'historique peut être géré par CVS pour consigner tous les événements de la vie d'un projet. Ce fichier se trouve dans `$CVSROOT/CVSROOT/history`. Il permet par exemple de savoir quels fichiers ont été modifiés, à quelle date et par qui?...et donc de punir les coupables! ;-)

Cette consultation est possible par la commande

\$ cvs history

Les options de cvs history sont importantes

`cvs history -c` permet de connaître chaque fois qu'un commit a été fait (Repository modifié)

`cvs history -u <user>` permet de connaître toutes les actions qu'a fait un user <user>

`cvs history -e` extrait toutes les informations du fichier history

`cvs history -l` montre la dernière modification

`cvs history -a` montre les événements relatifs à chaque utilisateur ayant travaillé sur le projet

....[consulter la doc pour connaître toutes les options]

7.5 consigner les événements

CVS utilise une base de données d'événements. On peut choisir les événements que l'on veut tracer et gérer: par exemple inscrire chaque fois que quelqu'un a fait un cvs commit dans le repository...savoir qui a fait une branche?, qui a récupérer l'archive? etc...

On peut configurer CVS pour enregistrer différents types d'événements grâce à des scripts qui vont s'exécuter lorsque ces événements auront lieu.

Les fichiers d'événements se trouvent dans `$CVSROOT/CVSROOT`. Le plus important est le fichier "module" qui est consulté lorsque les commandes CVS (`cvs commit`, `cvs add`) seront exécutées.

7.6 être averti des tentatives de modifications par "les autres"

Pour savoir qui est en train d'éditer ou modifier un fichier dans son répertoire local, il faut utiliser

1. **cvs watch on <fichier>** : pour indiquer que l'on veut surveiller l'édition de certains fichiers
`cvs watch off` : pour désactiver cette surveillance
2. **cvs watch add | remove -a all <fichier>**: Cette commande ajoute l'utilisateur qui l'a tapée dans la liste des utilisateurs qu'il faut avertir lorsque le fichier en question est touché/modifié. Pour ôter

l'utilisateur de la liste des utilisateurs à avertir **\$ cvs watch remove**. L'option "-a" permet d'indiquer le type d'événements dont on souhaite être notifié. Le fichier d'action de notification se trouve dans **\$CVSROOT/CVSROOT/notify** Par défaut, on trouve dans ce fichier une notification par mail, mais rien n'empêche d'écrire son propre script de notification.

```
ALL mail %s -s "CVS notification" ALL echo "CVS notification %s" >> /home/cvs/00NOTIFCVS
```

Par défaut le mot clé ALL indique de consigner tous les événements (ALL = edit et commit) et d'en inform

- Si 2 utilisateurs X et Y sont sur un répertoire géré par CVS (inclus dans \$CVSROOT) .
- Si les actions de surveillance ont été mises en place (cvs watch on ; cvs watch add)
- Si l'utilisateur X est en train d'éditer (cvs edit) ou d'intégrer un fichier dans le repository (cvs commit) , alors l'utilisateur Y sera averti selon l'action inscrite dans le fichier "notify"... et ceci inversement si c'est Y qui modifie, c'est X qui sera averti automatiquement

Lorsqu'on demande à mettre en place une notification par cvs watch, CVS fait passer les fichier à surveiller en "lecture seule" (readonly)!! Ainsi les utilisateurs ne peuvent plus modifier localement leur fichier ! (on peut utiliser chmod mais c'est triché...!)

Cela pour forcer les utilisateurs à utiliser la commande cvs edit afin

La commande **cvs edit** a 2 actions:

1. elle rend le fichier à surveiller modifiable pour l'utilisateur qui l'a tapée (rajout de la permission "w" sur le fichier en questin)
2. elle informe/notifie les autres utilisateurs qui ont demandés à être informés qu'on s'apprete à éditer et modifier un fichier sous surveillance. La notification, rappelons le se faisant selon l'action inscrite dans \$CVSROOT/CVSROOT/notify

7.7 Savoir qui travaille sur les fichiers de l'archive?

On peut savoir qui travaille ou a travaillé sur les fichiers de l'archive par les commandes

\$ cvs watchers [fichier]

affiche les utilisateurs qui ont demandé d'être notifiés des modifications apportées aux fichiers (ceux qui ont tapés cvs watch on ; et cvs watch add)

```
$ cvs watchers doccvs.lyx
```

```
zaza tedit tunedit tcommit doccvs.momo momo tedit tunedit tcommit
```

(dans ce cas zaza est un utilisateur qui a demandé a être notifié par cvs)

\$ cvs editors [fichier]

affiche les utilisateurs qui sont en train de travailler sur les fichiers en questions

```
$ cvs editors doccvs.lyx
```

```
zaza Thu Nov 4 15:52:35 1999 GMT pcmaison /home/zaza/essaicvs doccvs.momo
```

```
momo Thu Nov 4 11:05:58 1999 GMT pcmaison /home/momo/essaicvs
```

7.8 informer les autres développeurs qu'on travaille sur un fichier

Comme on l'a vu, tout ce système est fait pour faciliter la communication et l'information entre développeurs travaillant sur un même projet. Le but est de surveiller les modifications de fichiers sensibles par des tierces personnes (par `cvs watch on` et `cvs watch add`). Le fait d'avoir tapé ces commandes entraîne le passage des fichiers à surveiller en mode "lecture seule" dans le répertoire local de l'utilisateur. Ainsi, le développeur est obligé de rendre son fichier "writable" pour pouvoir le modifier. Cette modification se fait par la commande:

\$ cvs edit

Lorsqu'on utilise **cvs edit** CVS fait passer les fichiers en mode "writable", et en outre on notifie les autres développeurs que l'on s'apprete à modifier un fichier (si tant est que l'on ait demandé de consigner ces événements par `cvs watch on` ; et `cvs watch add`).

Bien sur personne ne peut vous forcer à utiliser éditer `cvs edit` plutôt que "`chmod +w le fichier`" ... Disons qu'utiliser `cvs edit` est fait pour informer les autres!

Le fichier passe en mode "writable" jusqu'à ce qu'on ait tapé la commande

\$ cvs unedit

ou bien qu'on ait intégré les modifications à l'archive par **\$ cvs commit**

8 Utiliser CVS en client-serveur à travers un réseau

Pour des projets mobilisant des développeurs situés sur des lieux ou des machines différentes, on peut faire en sorte que le Repository central soit placé sur une machine particulière d'un réseau. Les développeurs peuvent alors accéder à une machine qui est un serveur de Repository, depuis des machines cliente.

Les développeurs doivent alors accéder au Repository central non plus en se basant sur la variable `CVSROOT` [à terminer.....page 18 de la doc]

9 Lexique

Repository

lieu où sont stockés tous les fichiers d'administration de CVS ainsi que les différentes versions successives d'un projet.

Module

Tags

un label qui identifie une certaine état "t" d'un développement. En d'autre termes, un tag est aussi un nom symbolique que l'on donne à une révision. La RedHat 5.2 noyau 2.0.36 ne s'appelait il pas "Apollo"?

stickyTag

VendorTag

ReleaseTag

Revision

s'applique à un fichier : Une révision d'un fichier est une modification qui a conduit CVS à incrémenter son numéro de version, en le rangeant dans le Repository. La numérotation se fait de façon incrémentale à partir d'une version de base 1.1 -> 1.2 -> 1.3 etc...

Release/Version

s'applique à un logiciel complet : numéro de version d'un produit logiciel

main_trunk

le tronc principal ;-). C'est à dire la succession de révisions séquentielles des fichiers d'un projet, à partir de la version de base 1.1.

branch

une branche ;-)) C'est à dire une séparation du développement par rapport au tronc principal (voir au dessus tronc principal). Les révisions de fichiers appartenant à une branche n'apparaissent nulle part ailleurs que dans cette branche.

head_version

version de tête de l'archive dans le Repository. C'est la dernière version courante, celle dont le numéro de révision est le plus élevé dans le "main trunk".

10 Quick Reference

Les commandes de CVS sont:

add

pour ajouter un nouveau fichier dans le Repository. IL faut valider tout de suite après par cvs commit.

admin

Interface d'administration pour rsc.

annotate

Show last revision where each line was modified checkout

checkout

extraît un projet entier ou des fichiers du Repository, et les place dans le répertoire de travail local

commit

intègre les fichiers dans le Repository

diff

montre les différences entre la version de tête du Repository et sa propre version qu'on possède dans son répertoire local

edit

Get ready to edit a watched file editors See who is editing a watched file

export

Exporte un ou tous les fichiers du Repository CVS dans le répertoire local de travail. Cela est identique a cvs checkout

history

montre les accès aux différents Repositoires gérés par CVS

import

Importe (archive) tous les fichiers d'un projet dans un Repository CVS. C'est l'étape initiale de constitution d'une archive dans sa première version 1.1. La commande "import" utilise les "vendor branches"

init

création initiale d'un Repository de CVS

log

Print out history information for files login Prompt for password for authenticating server.

logout

Removes entry in .cvspass for remote repository.

rdiff

création d'un fichier de différence entre 2 fichiers du Repository. Le fichier produit est au format "patch" et peut donc être utilisé comme un fichier de patch

releases

release Indicate that a Module is no longer in use

remove

Remove an entry from the repository rtag Add a symbolic tag to a module

rtag

Add a symbolic tag to a module

status

Display status information on checked out files

tag

Add a symbolic tag to checked out version of files unedit Undo an edit command

update

Bring work tree in sync with repository

watch

Set watches

watchers

See who is watching a file

11 Joli schéma