Gestion de la configuration des logiciels (SCM)

Aziz Salah, Prof. Dépt. d'informatique, UQAM (hiver 2005)
Guy Tremblay, Prof. Dépt. d'informatique, UQAM (été 2005)

Plan

Remarque : Dans cette présentation, pour alléger le texte, les termes « gestion de la configuration du logiciel » seront souvent remplacés par « SCM », où SCM est une abréviation des termes anglais « *Software Configuration Management* ».

Qu'est-ce que la gestion de la configuration du logiciel?

> *[Configuration management] is the discipline of developing uniform descriptions of a complex product at discrete points in its life cycle with a view to controlling systematically the manner in which the product evolves.*
>
> (A.A. Takang & P.A. Grubb, « *Software Maintenance – Concepts and Practice* », Thomson Computer Press, 1996)

- La gestion de la configuration du logiciel consiste à développer et appliquer des standards et procédures qui vont permettre de gérer *l'évolution* d'un logiciel. Ainsi, on veut être capable de gérer l'existence de diverses variantes (versions) du logiciel, de construire la version la plus à jour, de retracer des anciennes versions, de savoir quelles sont les modifications qui ont été effectuées et pourquoi, etc.
- Tous les produits associés au processus de développement du logiciel (des dizaines, des centaines, parfois des milliers de documents...) peuvent et doivent être gérés par la SCM :
  - Spécifications;
  - Documents de conception;
  - Programmes;
  - Plans de tests et jeux d'essai;
  - Manuels d'utilisation.
  - etc.

Deux concepts clés

- Une *version* est une instance d'un système qui est strictement distincte des autres instances (au niveau fonctionnel, performance, plate-forme cible, etc.).
- Un *« release »* est une version d'un système qui est destinée à la distribution à l'extérieur de l'équipe de développement (par exemple, pour des clients externes).

  Le nombre de *releases* est toujours inférieur au nombre de versions, puisque des versions internes ou privées à l'équipe peuvent exister (pour tests, corrections de bogues, etc.). Un *release* est aussi une version qui satisfait certains critères de qualité, autrement on ne le rendrait pas public.

  Un *release* pour un client n'inclut pas seulement le code exécutable mais peut aussi inclure des fichiers de configuration, des fichiers de données, des programmes d'installation, de la documentation, etc.

Principales activités de la SCM

- Définir les types de documents à gérer ainsi qu'une convention pour nommer et identifier les documents.
- Désigner qui prend en charge la responsabilité des procédures de la SCM.
- Définir des politiques pour contrôler les changements et gérer les versions.
- Définir et maintenir à jour une base de données pour la SCM.
- Sélectionner les outils qui vont être utilisés dans le processus de la SCM.
- Définir le processus d'utilisation des outils.

Note : Des standards officiels (par exemple, les standards IEEE la SCM) existent pour définir et encadrer ces différentes activités. De tels standards sont généralement utilisés dans les organisations qui utilisent des processus structurés et formels.

Dans des organisations avec un processus structuré, le processus de gestion des changements peut lui aussi être très formel. Dams ce cas, ce processus entre en jeu à partir du moment où on désire modifier un item qui a été mis sous le contrôle de la SCM. Ainsi, lorsqu'un problème est identifié, une demande formelle de changement doit être complétée (« *change request form* »), cette demande doit être évaluée et formellement acceptée par un « *Change Control Board* ». Ensuite, une fois que la modification appropriée a été effectuée, celle-ci doit encore être approuvée pour assurer que la correction n'a pas créé de nouveaux problèmes, etc.

CVS : un outil pour le contrôle des versions

> Le *contrôle de version* (ou le *contrôle de révision*) consiste à maintenir des informations sur l'évolution d'un projet de façon à pouvoir retrouver des versions antérieures de fichiers, suivre les modifications et (souvent le plus important) coordonner les efforts d'une équipe de développement.
>
> (G.N. Purdy, « CVS précis & concis », Éditions O'Reilly, Paris 2004)

- CVS = *Concurrent Versions System*
- Permet la gestion des versions de documents (textes principalement) et de leurs historiques de développement.
- Permet à plusieurs développeurs de travailler sur les même fichiers en parallèle : les changements qui ne créent pas de *conflit* sont automatiquement acceptés, les autres doivent être approuvés par l'usager.
- La « version maître » est conservée dans un dépôt (une archive, en anglais « *a repository* ») centralisé. Chaque développeur travaille sur une copie locale.
- CVS est seulement un outil. La conception et la planification de la SCM est la responsabilité des utilisateurs.

Une introduction à CVS

Dans cette introduction à CVS, on va supposer que l'équipe de développement est composée de trois personnes, identifiées par leur code usager Unix : `salah`, `ahmed` et `nadia`.

Initialisation pour l'utilisation de CVS :

- Pour utiliser CVS, il faut tout d'abord qu'un « administrateur » crée et initialise le dépôt (le répertoire qui va contenir la copie maître). (Notons qu'ici il ne s'agit pas nécessairement d'un administrateur système, mais simplement de la personne qui aura la responsabilité de gérer le dépôt pour un groupe de projets.)

- Pour ce faire, l'administrateur utilise la commande « `cvs init` » : cette commande doit être exécutée une seule fois, par l'administrateur, pour créer le dépôt qui sera partagé par l'ensemble des développeurs (pour un ou plusieurs projets).

Par exemple, nous allons créer le dépôt partagé par `salah`, `ahmed` et `nadia` dans le sous-répertoire `CVS_REP` du compte de l'usager `salah` comme suit:

```
salah@arabica> ls
salah@arabica> cvs -d /info2/salah/CVS_REP init
salah@arabica> ls
CVS_REP
salah@arabica> ls -R
```

```
.:
CVS_REP

./CVS_REP:
CVSROOT

./CVS_REP/CVSROOT:
checkoutlist    config,v       Emptydir  modules,v  taginfo
checkoutlist,v  cvswrappers    history    notify     taginfo,v
commitinfo      cvswrappers,v  loginfo    notify,v   val-tags
commitinfo,v    editinfo       loginfo,v  rcsinfo    verifymsg
config          editinfo,v     modules    rcsinfo,v  verifymsg,v

./CVS_REP/CVSROOT/Emptydir:
```

On remarque que CVS a créé un grand nombre de sous-répertoires et de fichiers: ce sont tous des fichiers de gestion de CVS qui ne doivent jamais être modifiés directement. Au fur et à mesure où de nouveaux modules seront ajoutés au dépôt, des sous-répertoires propres à ces modules seront créés dans CVS_REP.

Format général des commandes CVS

Usage: `cvs [cvs-options] command [command-options-and-arguments]`

- *`cvs-options`* : représente des options de CVS
- *`command`* : représente l'une des commandes que CVS peut exécuter, entre autres :
  - `init`
  - `import`
  - `checkout (ou co)`
  - `update (ou upd)`
  - `add`
  - `commit (ou ci)`
  - `export`
  - `log`
  - `status`
  - `diff`
- *`Command-option-and-arguments`* : options et arguments spécifiques à la commande

La commande `cvs init` utilisée précédemment illustre l'utilisation de l'option '`-d`' (nom du dépôt), laquelle option peut être utilisée avec n'importe quelle commande CVS et qui doit alors apparaître juste avant le nom de cette commande (e.g., `init, add, commit`, etc.) et qui permet d'indiquer le dépôt sur lequel la commande s'applique.

Règle générale, on travaille souvent sur un même dépôt pendant une longue période, et il peut donc devenir lourd de devoir spécifier explicitement le dépôt à chaque fois qu'on exécute une commande. Par contre, il est possible d'indiquer, via des variables d'environnement, le dépôt par défaut.

Commandes pour configurer les variables d'environnement :

- CVS utilise un certain nombre de variables d'environnement :
  - `CVSROOT` : indique l'emplacement du dépôt.
  - `CVSEDITOR` : indique l'éditeur de texte à utiliser pour fournir des commentaires.
  - Autres variables d'environnement: voir documentation CVS.

Chaque développeur doit configurer les variables d'environnement de CVS dans son propre compte. Par exemple, supposons que l'on utilise le dépôt créé le sous-répertoire `CVS_REP` de l'usager `salah` (`/info2/salah/CVS_REP`).
Selon le *shell* utilisé, chaque développeur `X` peut donc faire la commande suivante, pour indiquer ce dépôt:

- pour csh :

```
X@arabica> setenv CVSROOT /info2/salah/CVS_REP
```

- pour bash :

```
X@arabica> set CVSROOT=/info2/salah/CVS_REP
X@arabica> export CVSROOT
```

Dans ce qui suit, on va dorénavant supposer que chaque utilisateur a bien spécifié le dépôt tel qu'indiqué avec `CVSROOT`.

La commande import

La commande `import` permet au développeur d'indiquer à `CVS` que les fichiers et sous- répertoires du répertoire courant doivent être importés dans le dépôt `CVS`, donc doivent être mis sous le contrôle de CVS.

`salah` veut faire mettre les fichiers de son répertoire mon`_tutoriel_cvs` dans le dépôt, donc sous le contrôle de CVS, avec comme nom de projet «`tutoriel_cvs`» -- on utilise aussi le terme de «*module*».

```
salah@arabica> cd mon_tutoriel_cvs
salah@arabica> ls
rep1      rep2      main.c     makefile

salah@arabica> cvs import -m "Version initiale" tutoriel_cvs UQAM_TUTORIEL_CVS VERSION_INITIALE
N tutoriel_cvs/main.c
N tutoriel_cvs/makefile
cvs import: Importing /info2/salah/CVS_REP/tutoriel_cvs/rep1
N tutoriel_cvs/rep1/code_f1.c
N tutoriel_cvs/rep1/code_f2.c
N tutoriel_cvs/rep1/makefile
cvs import: Importing /info2/salah/CVS_REP/tutoriel_cvs/rep2
N tutoriel_cvs/rep2/code_f4.c
```

```
N tutoriel_cvs/rep2/code_f5.c
N tutoriel_cvs/rep2/code_f3.c
N tutoriel_cvs/rep2/makefile

No conflicts created by this import
```

On remarque que `tutoriel_cvs` est le nom que l'on veut utiliser pour le projet (le module) dans le dépôt, et que c'est donc le nom spécifié lors de l'appel à `import`. Le symbole `UQAM_TUTORIEL_CVS` est utilisé seulement comme une étiquette (un *tag*). Plus précisément, deux étiquettes doivent être spécifiées : la première indique généralement un nom lié à l'organisation, alors que le deuxième (`VERSION_INITIALE`) indique qu'il s'agit d'un module nouvellement créé.

La commande `checkout` (ou `co`)

La commande `checkout` permet de créer une copie de travail locale à partir d'un module contenu dans le dépôt.

```
salah@arabica> cd ..
salah@arabica> ls
CVS_REP  mon_tutoriel_cvs
```

Ensuite, effectuons le `checkout` du module `tutoriel_cvs`, mais dans le répertoire `copie_tutoriel`:

```
salah@arabica> cvs checkout -d copie_tutoriel tutoriel_cvs
cvs checkout: Updating copie_tutoriel
U copie_tutoriel/main.c
U copie_tutoriel/makefile
cvs checkout: Updating copie_tutoriel/rep1
U copie_tutoriel/rep1/code_f1.c
U copie_tutoriel/rep1/code_f2.c
U copie_tutoriel/rep1/makefile
cvs checkout: Updating copie_tutoriel/rep2
U copie_tutoriel/rep2/code_f3.c
U copie_tutoriel/rep2/code_f4.c
U copie_tutoriel/rep2/code_f5.c
U copie_tutoriel/rep2/makefile
```

Ainsi, `salah` a créé une copie locale du module `tutoriel_cvs` dans le répertoire `copie_tutoriel`. En effet, l'option `-d` de la commande `checkout` permet de spécifier un répertoire où la copie locale doit être stockée – remarquons que cette option `-d` apparaît après le nom de la commande, contrairement à l'autre qui spécifiait le nom du dépôt.

On peut alors constater, en utilisant la commande `diff`, que les deux répertoires sont bien identiques, à l'exception des fichiers des sous-répertoires `CVS` qui sont utilisés par CVS pour gérer la copie locale – une fois assuré que la copie locale est correcte, on peut alors effacer le répertoire d'où a été effectué l'`import`.

```
salah@arabica> diff -r copie_tutoriel mon_tutoriel_cvs
Seulement dans copie_tutoriel/: CVS
Seulement dans copie_tutoriel/rep1: CVS
Seulement dans copie_tutoriel/rep2: CVS
salah@arabica> rm -r -f mon_tutoriel_cvs
salah@arabica> ls
copie_tutoriel CVS_REP
```

Supposons maintenant que `salah` veut modifier un fichier, par exemple `main.c`.
Une chose possible à faire lorsque plusieurs usagers peuvent vouloir travailler sur les mêmes fichiers est que `salah` informe les autres usagers qu'il désire travailler sur ce fichier (à l'aide de la commande `cvs edit`). Toutefois, bien que ceci soit conseillé pour certains gros projets à plusieurs développeurs, cela n'est pas vraiment nécessaire, car cela n'empêche aucunement d'autres usagers de quand même travailler sur ce fichier. Dans CVS, contrairement à d'autres systèmes de gestion de configuration, il n'y aucune façon de verrouiller un fichier pour en obtenir l'accès exclusif. Nous allons donc ignorer cette commande.

La commande `status`

La commande `cvs status` permet de consulter le statut d'un ou de plusieurs fichiers.

Avant de modifier le fichier `main.c`, `salah` consulte le statut du fichier :

```
salah@arabica> cvs status main.c
===================================================================
File: main.c            Status: Up-to-date

   Working revision:    1.1.1.1 Mon Feb 14 17:38:34 2005
   Repository revision: 1.1.1.1 /info2/salah/CVS_REP/tutoriel_cvs/main.c,v
   Sticky Tag:          (none)
   Sticky Date:         (none)
   Sticky Options:      (none)
```

Supposons maintenant que `salah` ajoute un commentaire au début du fichier `main.c` (commentaire à la première ligne), et qu'ensuite il vérifie à nouveau le statut:

```
salah@arabica> cvs status main.c
===================================================================
File: main.c            Status: Locally Modified

   Working revision:    1.1.1.1 Mon Feb 14 17:38:34 2005
   Repository revision: 1.1.1.1 /info2/salah/CVS_REP/tutoriel_cvs/main.c,v
   Sticky Tag:          (none)
   Sticky Date:         (none)
   Sticky Options:      (none)
```

On remarque que CVS indique bien que la copie locale (de travail) de `main.c` a effectivement été modifiée (`Locally Modified`).

La commande update

Une autre façon de constater que la copie locale a été modifiée est lorsqu'on utilise la commande `update` :

```
salah@arabica>cvs update
cvs update: Updating .
M main.c
cvs update: Updating rep1
cvs update: Updating rep2
arabica>
```

De façon plus générale, la commande `cvs update` effectue la mise à jour des copies locales (le répertoire de travail), c'est-à-dire que les copies locales des fichiers sont mises à jour *en leur fusionnant les modifications les plus récentes du dépôt*. Si la copie locale de travail d'un fichier a été modifiée, alors le nom du fichier est précédé d'un « `M` » dans le cas où les modifications du dépôt ont pu être fusionnées sans conflit.

Autre exemple : supposons que l'on supprime le fichier `makefile` du répertoire courant. La mise à jour de la copie locale détectera l'absence du fichier et ira alors chercher le fichier en question dans le dépôt – le dépôt peut donc aussi servir de répertoire pour conserver les copies de sauvegarde:

```
salah@arabica> rm makefile
salah@arabica> cvs update
cvs update: Updating .
M main.c
cvs update: warning: makefile was lost
U makefile
cvs update: Updating rep1
cvs update: Updating rep2
```

Voici les différentes possibilités d'indication sur les fichiers lors d'une commande `update`:

- `M` : Soit la copie locale est modifiée par rapport à la version du dépôt, soit les deux fichiers, la version du dépôt et la version locale, ont été modifiées et que CVS pourrait les fusionner sans conflit.
- `A` : le fichier a été mis sous le contrôle de CVS (cf. `cvs add`) mais le fichier n'a pas encore été ajouté au dépôt
- `R` : la copie locale du fichier a été effacée et a été marquée pour suppression (cf. `cvs remove`), mais le fichier n'a pas encore été supprimé du dépôt
- `C` : un conflit existe entre la copie locale et la copie du dépôt
- `U` : la copie locale du fichier vient d'être mise à jour par celle du dépôt (cf. `cvs upd`)
- `P` : la copie locale vient d'être mise à jour par celle du dépôt (avec une rustine = *patch*)
- `?` : le fichier est inconnu, i.e., n'est pas sous le contrôle de CVS et, donc, n'existe pas non plus dans le dépôt

La commande `commit`

La commande `cvs commit` permet de sauvegarder dans le dépôt les modifications qui ont été faites localement :

```
salah@arabica> cvs commit -m "Modification pour illustrer commit" main.c
Checking in main.c;
/info2/salah/CVS_REP/tutoriel_cvs/main.c,v  <--  main.c
```

```
new revision: 1.2; previous revision: 1.1
done
```

Tel qu'illustré par la commande, on a un fichier spécifique à sauvegarder dans le dépôt. Si aucun nom de fichier n'est indiqué, alors la mise à jour du dépôt se fait avec tous les fichiers de l'espace local qui ont été modifiés. On doit aussi spécifier un commentaire (avec l'option « `-m "..."` ») qui donne l'explication, qui justifie cette modification, donc qui donne la *raison* pour laquelle elle a été effectuée – si cette option n'est pas spécifiée, l'éditeur sera appelé pour effectuer l'entrée du commentaire.

On remarque dans la réponse de `CVS` qu'un nouveau numéro de révision a été associé au fichier `main.c` : 1.2. Lorsqu'une nouvelle version est mise à jour dans le dépôt, la partie décimale du numéro de version est augmentée de 1. Lorsqu'une prochaine version sera créée pour ce fichier, elle sera dénotée par 1.3, puis la suivante 1.4, etc. Soulignons que cette numérotation se fait de façon indépendante pour chaque fichier.

Par exemple, supposons maintenant que la participation d'`ahmed` dans le projet en cours soit restreinte au sous-répertoire `rep1` du module `tutoriel_cvs`. Ainsi, `ahmed` désire créer une copie de travail de `rep1` connue localement sous le nom `mon_rep_f1`.

```
ahmed@arabica> cvs checkout -d mon_rep1 tutoriel_cvs/rep1
cvs checkout: Updating mon_rep1
U mon_rep1/code_f1.c
U mon_rep1/code_f2.c
```

```
U mon_rep1/makefile
```

Supposons qu'ensuite `ahmed` modifie `code_f1.c` en ajoutant un commentaire au début du fichier. Ensuite, il met à jour le dépôt avec `commit`.

```
ahmed@arabica> cvs commit -m "Ajout ligne pour illustrer numero revision"
Checking in code_f1.c;
/info2/salah/CVS_REP/tutoriel_cvs/rep1/code_f1.c,v  <--  code_f1.c
new revision: 1.2; previous revision: 1.1
done
```

`ahmed` modifie encore `code_f1.c` en ajoutant un commentaire en fin du fichier puis met à jour :

```
ahmed@arabica> cvs commit -m "Ajout ligne pour illustrer numero revision"
cvs commit: Examining .
Checking in code_f1.c;
/info2/salah/CVS_REP/tutoriel_cvs/rep1/code_f1.c,v  <--  code_f1.c
new revision: 1.3; previous revision: 1.2
done
ahmed@arabica>  cat code_f1.c
// Commentaire debut (ahmed)

int f2 (int,int);

int f1(int a,int b)
```

```
{
  return (f2(1,3));
}

// Commentaire fin (ahmed)
```

La commande add

La commande `cvs add` sert à signaler que des nouveaux répertoires ou fichiers doivent être mis sous le contrôle de CVS.

Par exemple, supposons qu'`ahmed` a développé un nouveau fichier `code_f6.c` conservé dans le sous-répertoire `rep1_sous_rep` et qu'il veut ajouter ce sous-répertoire et son fichier au projet.

Tout d'abord, il doit signaler son intention de mettre ce répertoire et ce fichier sous le contrôle de CVS – dans l'exemple, le premier `update` sert donc à vérifier qu'un tel fichier ou répertoire n'a pas déjà été ajouté et permet donc de confirmer (signe '`?`') que le répertoire n'est pas encore sous le contrôle de CVS, alors que le second confirme que le fichier a bien été marqué pour ajout. Quant à la commande `ls`, elle permet simplement de voir que lorsqu'un répertoire est ajouté à CVS, un sous-répertoire CVS est automatiquement ajouté.

```
ahmed@arabica ls
code_f1.c      code_f2.c      CVS          makefile       rep1_sous_rep
ahmed@arabica cvs update
```

```
cvs update: Updating .
? rep1_sous_rep
ahmed@arabica cvs add rep1_sous_rep

Directory /info2/salah/CVS_REP/tutoriel_cvs/rep1/rep1_sous_rep added to the
repository

ahmed@arabica ls rep1_sous_rep
rep1_sous_rep:
code_f6.c  CVS

ahmed@arabica> cvs add -m "Ajout de repertoire pour add" rep1_sous_rep/*
cvs add: scheduling file `rep1_sous_rep/code_f6.c' for addition
cvs add: cannot add a `CVS' directory
cvs add: use 'cvs commit' to add this file permanently
ahmed@arabica> cvs update
cvs update: Updating .
cvs update: Updating rep1_sous_rep
A rep1_sous_rep/code_f6.c
```

Rappelons que l'indication '`A`' avant `rep1_sous_rep/code_f6.c` veut dire que CVS sait que `rep1_sous_rep/code_f6.c` est marqué pour ajout mais que l'ajout effectif au dépôt n'a pas encore été effectué.

Ainsi, supposons que salah mette à jour sa copie de travail (à partir du niveau supérieur du répertoire `copie_tutoriel`).

```
salah@arabica> ls
CVS      main.c    makefile  rep1     rep2
salah@arabica> cvs update
cvs update: Updating .
U rep1/code_f1.c
```

`salah` obtient alors une copie à jour de `code_f1.c` qui a été modifié par `ahmed`; toutefois, on n'observe encore aucune trace du fichier `code_f6.c.`

Ensuite, supposons qu'`ahmed` mette à jour le dépôt, ce qui effectuera effectivement l'ajout du fichier `code_f6.c` au dépôt.

```
ahmed@arabica> cvs commit -m "Ajout pour illustrer nouveau repertoire"
cvs commit: Examining .
cvs commit: Examining rep1_sous_rep
RCS file: /info2/salah/CVS_REP/tutoriel_cvs/rep1/rep1_sous_rep/code_f6.c,v
done
Checking in rep1_sous_rep/code_f6.c;
/info2/salah/CVS_REP/tutoriel_cvs/rep1/rep1_sous_rep/code_f6.c,v  <--  code_f6.c
initial revision: 1.1
done
```

La commande log

Supposons que `salah` veuille voir les diverses versions qui existent pour le fichier `main.c` et voir les modifications qui ont été faites. Il peut alors consulter alors le journal (le *log file*) du fichier `main.c`

```
salah@arabica> cvs log main.c

RCS file: /info2/salah/CVS_REP/tutoriel_cvs/main.c,v
Working file: main.c
head: 1.2
branch:
locks: strict
access list:
symbolic names:
        VERSION_INITIALE: 1.1.1.1
        UQAM_TUTORIEL_CVS: 1.1.1
keyword substitution: kv
total revisions: 3;     selected revisions: 3
description:
----------------------------
revision 1.2
date: 2005/02/14 18:54:50;  author: salah;  state: Exp;  lines: +2 -0
Modification pour illustrer commit
```

```
----------------------------
revision 1.1
date: 2005/02/14 17:38:34;  author: salah;  state: Exp;
branches:  1.1.1;
Initial revision
----------------------------
revision 1.1.1.1
date: 2005/02/14 17:38:34;  author: salah;  state: Exp;  lines: +0 -0
Version initiale
=============================================================================
```

Scénario : supposons maintenant que `nadia` ait obtenu une copie locale du module `tutoriel_cvs` juste après qu'il ait été initialement créé, par exemple, à l'aide de la commande «`cvs co tutoriel_cvs`» (auquel cas elle avait alors obtenu la version 1.1). Ensuite, elle a modifié sa copie locale du fichier `main.c`. Elle veut maintenant sauvegarder ses modifications dans le dépôt.

Tout d'abord, elle s'assure d'avoir la version la plus à jour des fichiers.

```
nadia@arabica> cvs update
cvs update: Updating .
RCS file: /info2/salah/CVS_REP/tutoriel_cvs/main.c,v
retrieving revision 1.1
retrieving revision 1.2
Merging differences between 1.1 and 1.2 into main.c
rcsmerge: warning: conflicts during merge
cvs update: conflicts found in main.c
C main.c
U makefile
cvs update: Updating rep1
U rep1/code_f1.c
U rep1/code_f2.c
U rep1/makefile
cvs update: Updating rep2
```

Lors du `update`, les modifications effectuées au fichier `main.c` par `salah` (version 1.2) seraient intégrées à la copie locale de `nadia` (version 1.1). Toutefois, dans ce cas-ci, les deux modifications portaient sur une même partie du fichier (au début du fichier): il y a donc des conflits, tel qu'indiqué par CVS avec le '`C`' devant le nom du fichier et tel qu'indiqué par les lignes annotées avec '`<<<<<<<`', '`=======`' et '`>>>>>>>`' dans le fichier `main.c`:

```
nadia@arabica> more main.c
<<<<<<< main.c
# Commentaire de nadia
=======
#commentaire debut
>>>>>>> 1.2
```

`nadia` doit donc corriger le conflit avant de pouvoir faire son `commit`, c'est-à-dire qu'elle doit décider, parmi les deux versions du même code, laquelle est la bonne et ensuite doit faire les modifications appropriées au fichier.

Une fois les lignes en conflit corrigées, le `commit` sur `main.c` fonctionne sans problème:

```
nadia@arabica> cvs commit -m "Correction du conflit concernant commentaire"
cvs commit: Examining .
cvs commit: Examining rep1
cvs commit: Examining rep2
Checking in main.c;
/info2/salah/CVS_REP/tutoriel_cvs/main.c,v  <--  main.c
new revision: 1.3; previous revision: 1.2
done
```

Un point à souligner concernant la commande `update` exécutée un peu plus haut : on peut remarquer qu'aucune mention n'était faite du sous-répertoire `rep1_sous_rep` créé par `ahmed`. Par défaut, la commande `update` n'obtient pas les nouveaux sous-répertoires. Pour les obtenir, il faut plutôt utiliser la commande `update -d`, qui crée et met à jour les nouveaux répertoires:

```
nadia@arabica> cvs upd -d
cvs update: Updating .
cvs update: Updating rep1
cvs update: Updating rep1/rep1_sous_rep
U rep1/rep1_sous_rep/code_f6.c
cvs update: Updating rep2
```

Les commandes `tag` et `rtag`

- Les commandes `tag` et `rtag` permettent de donner une étiquette (un *nom symbolique*, un *tag*) à une configuration, i.e., à un groupe spécifique de versions.
- La commande `tag` opère sur la configuration constituée par la copie de travail courante.
- La commande `rtag` opère directement sur le dépôt et n'a pas besoin d'une copie locale de travail. Par contre, son utilisation nécessite la spécification d'un numéro de révision ou bien d'une date, et d'un nom de fichier ou d'un répertoire donné comme paramètre.
- Une configuration est constituée d'un fichier ou d'un ensemble de fichiers.
- Une configuration représente, en général, une version qui fonctionne ou qui marque un changement majeur durant le développement d'une application.

Usage des commandes `tag` et `rtag`

```
cvs [cvs-options] tag [command-options] tagname [filenames]
```

```
cvs [cvs-options] rtag [command-options] tagname [filenames]
```

Exemple d'étiquettes pour les commandes `tag` et `rtag` – seules les lettres, chiffres, tirets et soulignés sont permis:

- PRE_VERSION_BETA-0-1
- VERSION_ALPHA-0-4
- RELEASE-2-0

Étiquettes (*tags*) spéciales prédéfinies:

- HEAD: étiquette qui représente l'ensemble des révisions les plus récentes des divers fichiers dans le dépôt
- BASE: étiquette des révisions qui correspondent aux fichiers obtenus pour la copie locale de travail

Chaque entreprise possède sa propre politique concernant les événements qui donnent lieu à des *tags*. Par exemple, les événements suivants peuvent être considérés comme devant donner lieu à une version identifiée par une étiquette spéciale :

- Après la complétion d'une fonctionnalité ou d'un service
- Après chaque phase majeure de développement
- Juste avant la suppression d'une fonctionnalité ou d'un service
- Juste avant le début des tests
- Avant de faire des changements d'un programme qui fonctionne
- Juste avant la création d'une branche
- Juste après la fusion d'une branche

Il est important d'utiliser des noms significatifs (sans caractères spéciaux tels '.', ',', ';', '$', ':', '@'; par contre on peut utiliser '_'ou '-') et ayant un format fixe qui contient l'information essentielle. Dans une entreprise avec une grosse équipe, des noms formels comme les suivants peuvent être utilisés:

VERSION-ALPHA-TEST-0-1
VERSION-BETA-TEST-0-2
VERSION-FINAL-1-2
etc.

Pour une utilisation dans un cours de programmation, des noms symboliques significatifs en lien avec le cours ou avec le processus de remise des travaux peuvent être utilisés, tel qu'illustré plus bas.

Par exemple, `salah` veut identifier une version du tutoriel sur CVS qui pourra être mis à la disposition des étudiants du cours INF3135 du trimestre d'automne 2005:

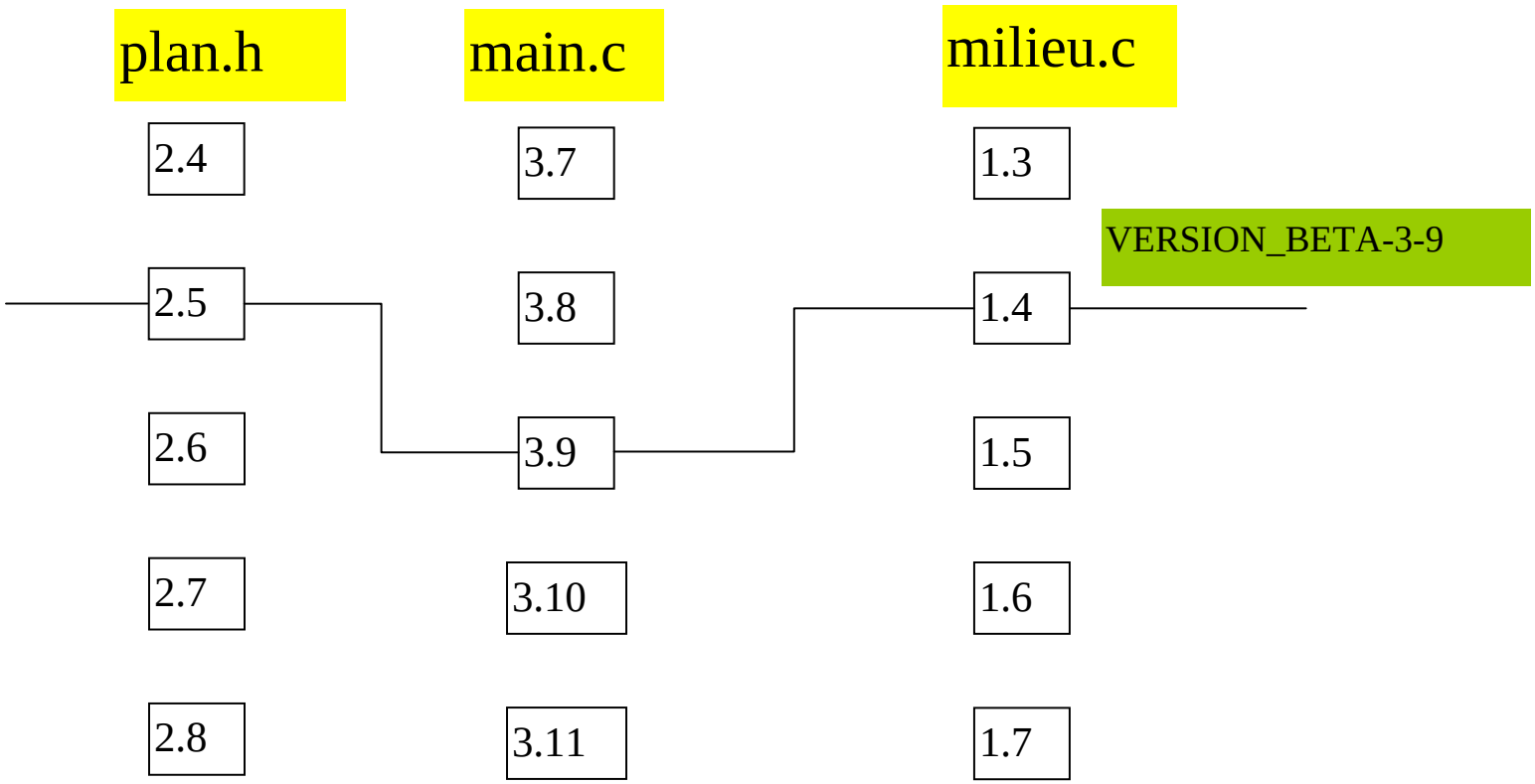
```
salah@arabica> cvs tag VERSION_INF3135_AUT05
cvs tag: Tagging .
T main.c
T makefile
cvs tag: Tagging rep1
T rep1/code_f1.c
T rep1/code_f2.c
T rep1/makefile
cvs tag: Tagging rep1/rep1_sous_rep
T rep1/rep1_sous_rep/code_f6.c
cvs tag: Tagging rep2
T rep2/code_f3.c
T rep2/code_f4.c
T rep2/code_f5.c
T rep2/makefile
```

On peut obtenir l'information sur les *tags* à l'aide de la commande `status -v` – on remarque alors que les divers fichiers qui sont associés à une même étiquette symbolique, par exemple, `VERSION_INF3135_AUT05`, peuvent avoir des numéros de révision différents:

```
salah@arabica> cvs status -v main.c
===================================================================
File: main.c            Status: Up-to-date

   Working revision:    1.3     Mon Feb 16 19:15:04 2005
   Repository revision: 1.3     /info2/salah/CVS_REP/tutoriel_cvs/main.c,v
   Sticky Tag:          (none)
   Sticky Date:         (none)
   Sticky Options:      (none)

   Existing Tags:
        VERSION_INF3135_AUT05           (revision: 1.3)
        VERSION_INITIALE                (revision: 1.1.1.1)
        UQAM_TUTORIEL_CVS               (branch: 1.1.1)

salah@arabica> cvs status -v makefile
===================================================================
File: makefile          Status: Up-to-date

   Working revision:    1.1.1.1 Mon Feb 14 17:38:34 2005
   Repository revision: 1.1.1.1 /info2/salah/CVS_REP/tutoriel_cvs/makefile,v
   Sticky Tag:          (none)
   Sticky Date:         (none)
   Sticky Options:      (none)

   Existing Tags:
        VERSION_INF3135_AUT05           (revision: 1.1.1.1)
        VERSION_INITIALE                (revision: 1.1.1.1)
        UQAM_TUTORIEL_CVS
```

Un développeur ou un client qui voudrait obtenir l'ensemble des fichiers associés identifié par cette étiquette pourrait alors procéder simplement comme suit:

```
client@arabica> cvs co -rVERSION_INF3135_AUT05 tutoriel_cvs
cvs checkout: Updating tutoriel_cvs
U tutoriel_cvs/main.c
U tutoriel_cvs/makefile
cvs checkout: Updating tutoriel_cvs/rep1
U tutoriel_cvs/rep1/code_f1.c
U tutoriel_cvs/rep1/code_f2.c
U tutoriel_cvs/rep1/makefile
cvs checkout: Updating tutoriel_cvs/rep1/rep1_sous_rep
U tutoriel_cvs/rep1/rep1_sous_rep/code_f6.c
cvs checkout: Updating tutoriel_cvs/rep2
U tutoriel_cvs/rep2/code_f3.c
U tutoriel_cvs/rep2/code_f4.c
U tutoriel_cvs/rep2/code_f5.c
U tutoriel_cvs/rep2/makefile
```


plan.h
main.c
milieu.c
2.4
3.7
1.3
VERSION_BETA-3-9
2.5
3.8
1.4
2.6
3.9
1.5
2.7
3.10
1.6
2.8
3.11
1.7

Remarque: La figure qui précède montre l'idée générale qu'une étiquette (par exemple, l'étiquette `VERSION_BETA-3-9`), celle-ci permet d'identifier un groupe spécifique de versions de fichiers – donc crée une sorte d'image instantanée *(snapshot)* de l'état du système à un instant donné. Bien qu'il soit possible que d'autres versions du fichier aient été créées par la suite, on peut toujours référer aux anciennes versions *taggées* grâce à l'étiquette qui leur est associée, grâce au nom symbolique utilisé pour identifier le groupe de fichier de la version.

Quelques options de la commande tag:

- `-c` : avant d'associer une étiquette, vérifier que la copie de travail est inchangée par rapport à la dernière version du dépôt
- `-d` : supprimer l'étiquette indiquée
- `-l`: locale et non récursive pour les sous-répertoires
- `-r` : suivi d'un numéro de révision ou d'un nom de *tag*, permet la sélection de fichiers selon ce qui est spécifié
- `-D` : suivi d'une date (CVS prend la dernière révision avant la date spécifiée), ce qui permet la sélection de fichiers
- `-b` : suivi d'un nom de tag, permet la création d'une branche

Les commandes diff et rdiff

Ces commandes sont très utiles pour comprendre les différences qui existent entre la copie locale et le dépôt ou encore entre deux versions d'un fichier identifié par des numéros de révision spécifiques. La comparaison entre les fichiers se fait comme pour la commande `diff` d'Unix, donc comparaison ligne par ligne.

Par défaut, sans aucun argument, la commande `diff` compare la copie locale avec la version du dépôt d'où elle a été extraite initialement. En d'autres mots, `diff` sans argument permet de déterminer si des modifications locales ont été effectuées au fichier par rapport aux fichiers qui avaient été obtenus. Par exemple, dans le cas de `salah`, on obtiendrait ce qui suit, puisque `salah` n'a effectué aucune modification récente sur ses fichiers locaux (qui n'ont pas été sauvegardées dans le dépôt):

```
salah@arabica> cvs diff
cvs diff: Diffing .
cvs diff: Diffing rep1
cvs diff: Diffing rep2
```

Toutefois, nous savons par contre que `nadia` a effectué des modifications sur `main.c` et qu'elle les a sauvegardées dans le dépôt. On peut examiner dans quelle mesure la copie locale diffère de la version la plus à jour dans le dépôt (donc pas nécessairement la version à partir de laquelle la copie locale a été obtenue) en indiquant que la comparaison doit s'effectuer par rapport à la version la plus à jour du dépôt, version dénotée par `HEAD`:

```
salah@arabica> cvs diff -rHEAD
cvs diff: Diffing .
Index: main.c
===================================================================
RCS file: /info2/salah/CVS_REP/tutoriel_cvs/main.c,v
retrieving revision 1.3
retrieving revision 1.2
diff -r1.3 -r1.2
1c1
< # Commentaire de nadia
---
> #commentaire debut
cvs diff: Diffing rep1
cvs diff: Diffing rep2
```

De façon générale, il est possible de spécifier des numéros spécifiques de versions, par exemple :

```
salah@arabica> cvs diff -r1.1 -r1.2 main.c
Index: main.c
===================================================================
RCS file: /info2/salah/CVS_REP/tutoriel_cvs/main.c,v
retrieving revision 1.1
retrieving revision 1.2
diff -r1.1 -r1.2
0a1
> #commentaire debut
```

CVS et fichiers binaires

Une particularité importante de `CVS` est que le dépôt ne contient pas directement l'ensemble du texte des différentes révisions. La sauvegarde des nouvelles versions se fait plutôt de façon *incrémentale*, c'est-à-dire que seules les *différences* entre les fichiers sont sauvegardées. Ces différences sont calculées à l'aide de `diff`, donc la comparaison se fait de façon textuelle, ligne par ligne. `CVS` peut donc reconstruire n'importe quelle version ancienne en se basant sur les modifications effectuées pour obtenir cette version.

Un avantage important de cette approche basée sur `diff` est de minimiser l'espace nécessaire pour conserver les différentes révisions d'un fichier. Le désavantage est que ça ne fonctionne pas très bien pour la manipulation de fichiers non textuels. Dans le cas d'un fichier binaire (ce qui inclut les documents Word), il est donc préférable d'indiquer explicitement lors de l'ajout (avec la commande `add`) que le fichier est binaire, et ce à l'aide de l'option `'-kb'` :

```
cvs add –kb fich1.doc
```

La commande `remove`

La commande `cvs remove` permet d'indiquer qu'un fichier doit être supprimé du dépôt. Par exemple, supposons que `salah` désire supprimer le fichier `rep1/code_f1.c`. L'exemple qui suit illustre le fait que pour être supprimé du dépôt, le fichier doit tout d'abord être supprimé de l'espace local, et qu'il n'est pas véritablement supprimé tant qu'un `commit` (`ci`) n'a pas été effectué:

```
salah@arabica> cvs remove rep1/code_f1.c
cvs remove: file `rep1/code_f1.c' still in working directory
cvs remove: 1 file exists; remove it first
salah@arabica> rm -f rep1/code_f1.c
salah@arabica> cvs remove rep1/code_f1.c
cvs remove: scheduling `rep1/code_f1.c' for removal
cvs remove: use 'cvs commit' to remove this file permanently
salah@arabica> cvs update rep1/code_f1.c
R rep1/code_f1.c
salah@arabica> cvs ci -m "Suppression pour illustrer effet implicite"
cvs commit: Examining .
cvs commit: Examining rep1
cvs commit: Examining rep1/rep1_sous_rep
cvs commit: Examining rep2
Removing rep1/code_f1.c;
/info2/salah/CVS_REP/tutoriel_cvs/rep1/code_f1.c,v  <--  code_f1.c
new revision: delete; previous revision: 1.3
done
```

Observons maintenant ce qui se passe si `nadia` met ensuite à jour sa copie locale :

```
nadia@arabica> ls rep1/code_f1.c
rep1/code_f1.c
nadia@arabica> cvs upd
cvs update: Updating .
cvs update: Updating rep1
cvs update: rep1/code_f1.c is no longer in the repository
cvs update: Updating rep1/rep1_sous_rep
cvs update: Updating rep2
nadia@arabica> ls rep1/code_f1.c
rep1/code_f1.c: Aucun fichier ou répertoire de ce type.
```

La commande `export`

La commande `cvs export` permet la création d'une copie du projet prête pour la distribution (à l'externe). Elle est essentiellement équivalente à la commande `cvs checkout`, à la différence que les fichiers de gestion spécifique à `CVS` (dans le sous-répertoire `CVS`) *ne sont pas exportés*.

```
compte_pour_export@arabica> cvs export -rHEAD tutoriel_cvs
cvs export: Updating tutoriel_cvs
U tutoriel_cvs/main.c
U tutoriel_cvs/makefile
cvs export: Updating tutoriel_cvs/rep1
U tutoriel_cvs/rep1/code_f1.c
U tutoriel_cvs/rep1/code_f2.c
U tutoriel_cvs/rep1/makefile
cvs export: Updating tutoriel_cvs/rep1/rep1_sous_rep
U tutoriel_cvs/rep1/rep1_sous_rep/code_f6.c
cvs export: Updating tutoriel_cvs/rep2
U tutoriel_cvs/rep2/code_f3.c
U tutoriel_cvs/rep2/code_f4.c
U tutoriel_cvs/rep2/code_f5.c
U tutoriel_cvs/rep2/makefile
{compte_pour_export@linux} ls -R tutoriel_cvs
tutoriel_cvs:
main.c  makefile  rep1/  rep2/

tutoriel_cvs/rep1:
code_f1.c  code_f2.c  makefile  rep1_sous_rep/

tutoriel_cvs/rep1/rep1_sous_rep:
code_f6.c

tutoriel_cvs/rep2:
code_f3.c  code_f4.c  code_f5.c  makefile
```