Part V

Outils de dévéloppement pour le logiciel libre

Support pour le dévéloppement collaboratif

Un logiciel libre *qui a du succès* fedère une communauté d'utilisateurs et de dévéloppeurs, qui travaillent de façon *collaborative*.
Dévélopper *efficacement* un logiciel de façon collaborative est une tâche difficile, et nécéssite des outils pour::

- construire le logiciel à partir des sources
- échanger des modifications au logiciel entre dévéloppeurs
- suivre les modifications du logiciel:
  - ce qui a changé
  - qui l'a changé
  - quand le changement a été fait
  - à quel état se refère le changement
  - auditer le logiciel (revenir à un état donné)

Ces concepts sont pris en compte par la discipline du "software configuration management".

Construire un executable

Construire un executable à partir des sources peut nécéssiter des opérations complexes

- configuration sur la plateforme de deployement
- internationalisation/localisation
- compilation et édition des liens
- installation sur la plateforme de deployement

Beaucoup des outils et des bonnes pratiques qu'on retrouve dans le libre viennent de la tradition Unix.

Configuration

Le problème:

- la periode des “Unix Wars” avait produit une galaxie de variantes des API Unix, et des conventions de localisation des librairies et des binaires
- les dévéloppeurs souhaitaient isoler leur travail de ces détails

Les outils:

- autoconf
- automake

I18N, L10N

Le logiciel libre est dévéloppé souvent par des équipes transnationales, il faut donc des outils pour:

I18N : I$\underbrace{\textit{nternationalisatio}}_{18}$n; préparer un logiciel pour pouvoir fonctionner avec les conventions de plus d'un pays (langue, date, monnaie, etc.)

L10N : L$\underbrace{\textit{ocalisatio}}_{10}$n; spécialiser le logiciel pour un pays donné (langue, date, monnaie, etc. forment le `locale`)

Les outils:

- la librairie `gettext` de GNU fournit le nécéssaire pour cela (la librairie est disponible pour un très grand nombre de langages)
- les LC (locale categories) plus connues sont: LC_CTYPE et LC_TIME

Gettext: le programmeur

- encapsule toute chaîne de caractères qui peut être traduite dans un appel à `gettext` , d'habitude on trouve ça dans du code C

```
#include <libintl.h>
#define _(String) gettext (String)
. . .
printf(_("The file named %s is missing.\n"), fn);
```

- il appelle ensuite `xgettext` pour produire un template (extension `.pot`) qui servira aux traducteurs

Gettext: le traducteur

- le traducteur initialise sa traduction avec

```
msginit --locale=fr --input=logiciel.pot
```

  cela produit le fichier `fr.po` qui doit être traduit
- il édite `fr.po` et traduit la ligne `msgstr` de chaque entrée

```
#: toto.c:36
msgid "The file named %s is missing.\n"
msgstr "The file named %s is missing.\n"
```

- ensuite il compile `fr.po` avec `msgfmt` pour produire `fr.mo`, qui est alors prêt à être utilisé.

Make

Un outil pervasif que vous connaissez déjà; il permet de

- décrire la procédure de compilation de façon *déclarative* via des *dépendances* et des *règles* implicites ou explicites
- recompiler seulement les parties qui le necessitent (algorythme de tri topologique)

Tout est dans un fichier `Makefile`.
Le standard est désormais le `make` de GNU.

Pour Java, on trouve `ant`.

Le couteau suisse: diff et patch

Pour collaborer entre dévéloppeurs, dans le cadre souvent laxe de l'organisation d'un logiciel libre, deux outils sont très utilisés:

- `diff`: calcule la différence `D` entre un fichier `A` et un fichier `B`
- `patch`: applique une différence `D` (calculée par `diff`) à un fichier `A` pour produire `B`

Exemple de diff et patch: le contributeur

Scenario typique: il corrige un erreur, ou ajoute une fonctionnalité à un logiciel, en modifiant `f.c`:

```
tar xzvf logiciel-v2.1.tar.gz
cp -a logiciel-v-2.1 logiciel-v-2.1_work
cd logiciel-v-2.1_work
xemacs f.c
make . . .
```

Quand il est satisfait, il extrait ses modifications dans un "patch"

```
cd ..
diff -u logiciel-v-2.1/f.c logiciel-v-2.1_work/f.c > mesmodifs
```


et les envoye au dévéloppeur originaire:

```
mail author@somewhere
Subject: Fix for the I/O bug in funcion foo of f.c
Dear Author,
I found a way to prevent the core dump resulting
from oversized inputs in function foo of f.c by using
a guarded input loop. See the attached proposed patch.

Yours sincerely

Newbie

~r mesmodifs
"mesmodifs" 9/606
```

Attention

Il est de bonne pratique de

- ajouter dans votre message une explication claire et précise de votre modification,
- ajouter dans le code un commentaire avec une explication similaire

Regardez `http://tldp.org/HOWTO/Software-Release-Practice-HOWTO/patching.html` pour des conseils avisés.

Exemple de diff et patch: l'auteur originaire

Scenario typique: il reçoit le message et regarde la modification il décide de l'essayer, et sauve le message dans un fichier `/tmp/foo` puis applique le changement avec la commande `patch`

```
cd myprojects
cp -a logiciel-v-2.2 logiciel-v-2.2-test
cd logiciel-v-2.2-test
patch -p 1 < /tmp/foo
patching file f.c
Hunk #1 succeeded at 8 (offset 5 lines).
make . . .
```

S'il est satisfait, il accepte la modification.
N.B.: ici le patch est appliqué sur une version plus recente du logiciel! `patch` a utilisé le contexte produit par `diff -u` pour retrouver les lignes à modifier (décalées 5 lignes plus bas).

Un exemple simple: poor man's version control.

Le dossier contenant le projet d'un étudiant en Licence ressemble souvent à ça:

```
lucien> ls
a.out
projet.ml
projet-save.ml
projet-hier.ml
projet-marche-vraiement.ml
projet-dernier.ml
```

Quelle différence entre les cinq fichiers source?
Quelle relation de dépendence les lie entre eux?
Sans disposer d'outils spécifiques, il est très difficile de repondre.

Un exemple simple: poor man's version control, bis

Si on vous permet de réaliser le projet à plusieurs, cela devient vite pire:

```
lucien> ls ~joel/projet
a.out
module.ml
module-de-julien-qui-marche.ml
projet.ml
projet-save.ml
projet-hier.ml
projet-marche-vraiement.ml
projet-dernier.ml
```

```
lucien> ls ~julien/projet
a.out
module.ml
projet.ml
projet-recu-de-joel.ml
module-envoye-a-joel.ml
```

Quelle est la bonne combinaison de `projet.ml` et `module.ml` pour passer l'examen?

Echange de fichiers, creation de *patches*

Pour échanger les fichiers `projet.ml` et `module.ml`, Joel et Julien utilisent l'e-mail, `diff` et `patch`.

Julien

```
lucien> diff -Nurp projet-hier.ml projet.ml > mescorrections
lucien> mail -s "Voici mes modifs" joel@lucien < mescorrections
```

Joel

```
lucien> mail
Mail version 8.1.2 01/15/2001.  Type ? for help.
>   1 julien@lucien  Fri Sep 13 20:06   96/4309  Voici mes modif
& s 1 /tmp/changes
& x
lucien> patch < /tmp/changes
```

Maintenant, les modifications de Julien entre projet-hier.ml et projet.ml sont appliquées au fichier projet.ml de Joel.

Problèmes

Pourtant, le jour de l'examen, rien ne marche, alors que tout fonctionnait la veille.

Dans la panique, vous cherchez à comprendre

- ce qui a changé
- qui l'a changé
- quand le changement a été fait
- à quel état se refère le changement
- comment revenir à l'état qui fonctionnait

En bref, vous avez besoin d'un *système de contrôle de versions*.

Principes

Un système de contrôle de versions:

- gêre des unités de programmes (des fichiers, des dossiers, des arborescences, etc.)
- est capable de memoriser les changements effectués (notion de "version"):
  - qui a fait le changement
  - par rapport à quel état
  - pour quelle raison
  - à quelle date
- est capable de montrer les modifications entre deux versions
- est capable de restaurer l'état correspondant à une version ou date donnée
- peut gêrer l'intervention concurrente de plusieurs programmeurs

RCS

RCS = Revision Control System.
Auteur: Walter F. Tichy et plus tard Paul Eggert.

- un des plus anciens (1980)
- l'unité est le *fichier*
- tout l'historique est stoqué dans un repertoire RCS local
- par defaut, pour modifier un fichier il faut prendre un verrou: modèle *pessimiste* ne permettant pas de *modifications* en parallèle

RCS

Modèle de versions arborescent, numérotation conventionnelle:

- 1.1, 1.2, 2.3 ce sont des versions "principales"
- 1.1.1.1, 2.3.2.4 ce sont des versions "sur une branche"

Opérations courantes:

- sauver une version: `ci projet.ml`
- ressortir une version en lecture seule: `co projet.ml`
- ressortir une version en écriture: `co -l projet.ml`
- voir les différences (deltas) entre deux versions: `rcsdiff -r1.2 -r1.3 projet.ml`
- incorporer dans le "trunk" des changements fait sur une branche: `rcsmerge -r1.2.1.1 -r1.2.1.3 projet.ml`

CVS

CVS= Concurrent Versions System.
Auteurs: Dick Grune (1986), puis Brian Berliner (1989).

- construit comme un ensemble de scripts sur RCS
- l'unité recherchée est le *projet* (une arborescence de repertoires)
- tout l'historique est stoqué dans un repertoire CVS central, éventuellement via le réseau
- par défaut, on peut modifier les fichiers sans prendre des verrous: modèle *concurrent* et *optimiste*
- il existe la possibilité de travailler en réseau (`pserver` ou via `ssh` [recommandé])

CVS: modèle de versions

Il est assez alambiqué:

- chaque *fichier* garde ses versions RCS (1.1, 1.2, 2.3, 1.1.2.4, etc.)
- un état donné d'un *repertoire* peut être identifié par un `tag` qui est posé sur chaque fichier individuellement
- la gestion de branches se fait par des commandes spécifiques avec des tags

CVS: Opérations courantes

- ressortir un projet: `cvs checkout nomduprojet`
- rapatrier dans le CVS les modifications locales: `cvs commit`
- mettre à jour le workspace par rapport au CVS:
  `cvs update -d`
- poser un `tag` sur le repertoire CVS:
  `cvs rtag RELEASE_1_0 nomduprojet`
- voir les différences (deltas) entre deux versions d'un fichier:
  `cvs diff -r1.2 -r1.3 projet.ml`
- voir les différences (deltas) entre deux versions du projet:
  `cvs diff -c -r RELEASE_1_0 -r RELEASE_1_1`
- voir les différences (deltas) entre le workspace et la derniere mise à jour du CVS: `cvs diff`

CVS: Limitations

- impossible de renommer, deplacer, effacer un fichier
- presque aucune gestion des métadonnées (attributs des fichiers)
- impossible de gerer des liens ou des copies
- commit non atomique
- gestion des branches rigide avec des commandes ad hoc
- versions des fichiers disjointes des versions du projet
- repertoire centralisé, impossible de faire un miroir distant et de le fusionner après
- gestion des droits sur le serveurs basé sur les accounts Unix[29]

[29] pour donner un accès en écriture à un comtributeur, il faut lui créér un compte!

Subversion

Projet lancé par `CollabNet` en 2000, version 1.0 en Février 2004.
Idée: remplacer CVS en lévant des limitations.

- metadonnées arbitraires sur fichiers et dossiers
- fichiers et dossiers sont versionnés, leur métadonnées aussi
- notion de lien symbolique prise en compte
- les opérations de copie, renommage, effacement sont efficaces et gardent l'historique
- commit atomique
- numéro de version unique dans l'ensemble du projet
- gestion des droits interne (pas besoin de creer un utilisateur Unix sur le serveur)
- module Apache avec support WebDAV
- documentation très détaillée en ligne (`http://www.tigris.org`)

Subversion: gestion des branches

Subversion est basé sur un modèle centralisé (comme CVS, il faut un référentiel unique), *mais* il relâche énormement les contraintes de CVS sur les branches.

CVS : évolution des versions et des branches exclusivement à travers des commandes spécifiques utilisant les `tag` (horrible hack sur la couche RCS sousjacente)

Svn : les opérations de copies sont très légères, donc la création de branches est purement une *convention* entre les dévéloppeurs

Un exemple, pris de la présentation de Blair Zajac

Subversion: exemple de conventions

```
orcaware
        branches
                blair-super-orca2-working-on-rehat-9.0
        tags
                orcaware
                super-orca2
        trunk
                orcaware
                super-orca2
                        build
                        docs
                        emacs-style
                        ...
```

Subversion: exemple de processus

Un programmeur copies le “trunk” sur une nouvelle branche:

```
% svn cp -m "Creating branch for working on super orca 2" \
  https://svn.orcaware.com/orcaware/trunk/super-orca2 \
  https://svn.orcaware.com/orcaware/branches/blair-working-on-redhat-9.0
% svn log -v https://svn.orcaware.com/orcaware/branches/blair-working-on-redhat-9.0
# Prints revision number, say 123
% svn co https://svn.orcaware.com/orcaware/branches/blair-working-on-redhat-9.0
```

Quand il a fini son travail, il “update” sa branche

```
% svn merge -r 123:HEAD https://svn.orcaware.com/orcaware/trunk/super-orca2 .
# Test new code.
% svn commit -F message_file1
# Revision 256.
```

Ensuite l'administrateur incorpore les changements dans le trunk.

```
% cd /tmp
% svn co https://svn.orcaware.com/orcaware/trunk/super-orca2
% cd super-orca2
% svn merge . https://svn.orcaware.com/orcaware/branches/blair-working-on-redhat-9.0
% svn commit -F message_file2
```


Subversion: conventions

Attention, cet exemple *n'est pas normatif*:

- `trunk` est un nom de dossier comme un autre
- `tags` est un nom de dossier comme un autre
- `branches` est un nom de dossier comme un autre

Ce sont les communautés de dévéloppeurs qui fixent leur propres usages, l'outils n'impose rien (à part le référentiel centralisé).

Darcs = David's Advanced Revision Control System

Auteur: David Roundy, première version publique en 2003
Originalité:

- plus de référentiel central: chaque copie est un référentiel complet (vous pouvez tout faire dans le RER)
- l'unité de base n'est plus le fichier, mais le "patch"
- le n. de version est simplement une collection de "patches"
- plusieurs méthodes pour transferer des updates (e-mail, http, etc.)

Les utilisateurs de `darcs` sont libres de fixer leur propres conventions.
Attention: certaines opérations sur les "patches" sont très complexes, et des opérations naturelles comme la copie de fichier ne sont pas natives.

Quelques critères pour classer les VCS

Modèle:

- histoire: snapshot/changeset
- store: centralised/distributed
- collaboration: lock/merge
- unité: projet/fichier
- finesse des versions: fichiers/repertoires

VCS	unité	histoire	store	collab	versions
RCS	fichier	changeset	local	lock	fichier
CVS	projet	changeset	central	merge	projet[30]
Svn	projet	changeset	central	merge	global timeline
Darcs	changeset	changeset	distributed	merge	changeset[31]

[30] Hybride, avec tags et versions RCS
[31] Darcs est basé sur les patches, pas les fichiers!

Quelques critères pour classer les VCS

Features :

- permissions fines sur l'arborescence
- possilité de copier des parties de l'arborescence
- n. de revision indépendent du référentiel (important pour les VCS distribués)
- possibilité de travailler seulement sur un sous-repertoire
- annotation des dernières contributions à un fichier ligne par ligne
- messages de log per-fichier

BTS = Bug Tracking System

Un VCS est un important, mais ce n'est pas tout: il faut un support pour organiser le travail des dévéloppeurs

- soumettre une description de bug
- assigner le bug à un dévéloppeur
- indiquer quand le bug est corrigé
- faire de même pour les demandes de nouvelles fonctionnalités (features)

Quelques exemples

Certaines communautés utilisent simplement une mailing list, mais on peut retrouver des outils plus sophistiqués:

- BugZilla
- GNATS
- Mantis
- RT
- ...

Tout mettre ensemble: les forges

Une "forge logicielle" est un instrument qui intégre un ensemble d'outils:

- VCS
- BTS
- Mailing Lists
- ...

L'archetype est `SourceForge`, créé en 1999 par VA Software, mais on en trouve plusieurs aujourd'hui

- GForge (descendant de SourceForge)
- LibreSource (Inria)
- Trac (plus adapté à la gestion d'un seul projet)