# Logiciel libre, une introduction

Roberto Di Cosmo

Université Paris Diderot
UFR Informatique
Laboratoire Preuves, Programmes et Systèmes

roberto@dicosmo.org

27 Mars 2014



Part IV

# Outils de développement pour le logiciel libre



Outils collaboratifs
- Build/Configure/I18N/L10N
- Diff et Patch
- Version Control Systems
- Bug Tracking Systems
- Forges
- Intégration Continue



## Support pour le développement collaboratif

Un logiciel libre *qui a du succès* fedère des utilisateurs et des développeurs qui travaillent de façon *collaborative*.
Développer *efficacement* un logiciel de façon collaborative est une tâche difficile, et nécessite des outils pour:

- construire le logiciel à partir des sources
- échanger des modifications au logiciel entre développeurs
- 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 développeurs souhaitaient isoler leur travail de ces détails

Dans les environnements Unix (avec du support pour Mac OS X et Windows), il y a des outils qui peuvent aider:

- autoconf : identification automatique des fonctionnalités `http://www.gnu.org/software/autoconf/autoconf.html`
- automake : génération automatique de Makefiles portables `http://www.gnu.org/software/automake/automake.html`
- gnulib : collection de modules sources portables
- libtool : génération de librairies partagées portables

Une référence utile est le livre
`http://sources.redhat.com/autobook/`



## Exemple I

```
$git clone git://gitorious.org/parmap/parmap.git
Cloning into 'parmap'...
remote: Counting objects: 440, done.
remote: Compressing objects: 100% (434/434), done.
remote: Total 440 (delta 267), reused 0 (delta 0)
Receiving objects: 100% (440/440), 148.58 KiB, done.
Resolving deltas: 100% (267/267), done.

$cd parmap
$aclocal -I m4
$autoconf
$autoheader
$./configure
checking for ocamlc... ocamlc
```



## Exemple II

```
OCaml version is 3.13.0+dev11 (2012-01-26)
OCaml library path is /usr/local/lib/ocaml
checking for ocamlopt... ocamlopt
checking for ocamlc.opt... ocamlc.opt
versions differs from ocamlc; ocamlc.opt discarded.
checking for ocamlopt.opt... ocamlopt.opt
version differs from ocamlc; ocamlopt.opt discarded.
checking for ocaml... ocaml
checking for ocamldep... ocamldep
checking for ocamlmktop... ocamlmktop
checking for ocamlmklib... ocamlmklib
checking for ocamldoc... ocamldoc
checking for ocamlbuild... ocamlbuild
checking for camlp4... camlp4
```

## Exemple III

```
checking for camlp4boot... camlp4boot
checking for camlp4o... camlp4o
checking for camlp4of... camlp4of
checking for camlp4oof... camlp4oof
checking for camlp4orf... camlp4orf
checking for camlp4prof... camlp4prof
checking for camlp4r... camlp4r
checking for camlp4rf... camlp4rf
checking for ocamlfind... ocamlfind
checking for gcc... gcc
checking whether the C compiler works... yes
checking for C compiler default output file name... a.out
checking for suffix of executables...
checking whether we are cross compiling... no
```



## Exemple IV

```
checking for suffix of object files... o
checking whether we are using the GNU C compiler... yes
checking whether gcc accepts -g... yes
checking for gcc option to accept ISO C89... none needed
checking how to run the C preprocessor... gcc -E
checking for grep that handles long lines and -e... /bin
checking for egrep... /bin/grep -E
checking for ANSI C header files... yes
checking for sys/types.h... yes
checking for sys/stat.h... yes
checking for stdlib.h... yes
checking for string.h... yes
checking for memory.h... yes
checking for strings.h... yes
```



## Exemple V

```
checking for inttypes.h... yes
checking for stdint.h... yes
checking for unistd.h... yes
checking sched.h usability... yes
checking sched.h presence... yes
checking for sched.h... yes
checking whether sched_setaffinity is declared... yes
checking mach/thread_policy.h usability... no
checking mach/thread_policy.h presence... no
checking for mach/thread_policy.h... no
checking whether thread_policy_set is declared... no
ocamlbuild does not exists or it does not support -use-ocamlf
configure: creating ./config.status
config.status: creating Makefile_3.11
```



## Exemple VI

```
config.status: creating config.h
$make
ocamlfind ocamlc -package "unix␣bigarray␣" -c bytearray.
ocamlfind ocamlc -package "unix␣bigarray␣" -c parmap.mli
ocamlfind ocamlc -package "unix␣bigarray␣" -c setcore.ml
ocamlfind ocamlc -package "unix␣bigarray␣" -c bytearray.
ocamlfind ocamlc -package "unix␣bigarray␣" -c parmap.ml
ocamlc -c bytearray_stubs.c
ocamlc -c -cc "gcc␣-D_GNU_SOURCE␣-o␣setcore_stubs.o␣-fPI
ocamlmklib -o parmap bytearray.cmo parmap.cmo bytearray_
ocamlfind ocamlopt -package "unix␣bigarray␣" -c bytearra
ocamlfind ocamlopt -package "unix␣bigarray␣" -c parmap.m
ocamlmklib -o parmap bytearray.cmx parmap.cmx bytearray_
```



## I18N, L10N

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

I18N : I*nternationalisation*; préparer un logiciel pour pouvoir fonctionner avec les conventions de plus d'un pays (langue, date, monnaie, etc.)

L10N : L*ocalisation*; 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



q



## 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 "On doit traduire 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 développeurs, 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 développeur 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 modifs
& 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[35]

[35] pour donner un accès en écriture à un contributeur, il faut lui créer 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 développeurs

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 développeurs qui fixent leur propres usages, l'outils n'impose rien (à part le référentiel centralisé).



## Subversion: limitations

- les copies ne sont pas chères sur le serveur (on fait des liens), mais elle sont très chères sur le client, qui ne garde pas le partage!
- il fonctionne en mode serveur: on ne peut pas avoir une copie de l'historique chez soi; si vous n'avez pas de réseau, vous êtes perdus!
- il rend les forks assez simples, avec la creation de branches, mais il n'a pas d'outils efficaces pour faire des merge!



## 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.

## GIT

Auteur: Linus Torvalds (du 3 Avril 2005 au 26 Juillet 2005; Junio Hamano après)

Raison: problème de licence avec BitKeeper, utilisé jusque là

Objectifs:

- flot de travail distribué (comme Darcs, Hg, Monotone, Bazaar etc.)
- facilité des merge
- très bonne résistance contre les erreurs (accidentels ou intentionnels)
- très *rapide*



## Quelques caractèristiques de GIT

- on garde dans le repertoire .git une copie de chaque *objet* (fichier, repertoire, commit, etc.)
- chaque objet est identifié par un hash
- la synchronization entre différentes instances du repository se fait en envoyant seulement les objets différents (facile avec les hash)
- on ne garde pas des diffs (c'est une notion dérivée)
- Git dispose d'un panoplie touffue d'algorithmes de *merge*
- il est possible de *modifier* l'histoire des commits

Tout ceci a rendu Git très populaire, au delà du noyau Linux.
Attention La courbe d'apprentissage est assez raide.



## Quelques critères pour classer les VCS

- 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[36] |
| Svn | projet | changeset | central | merge | global timeline |
| Darcs | changeset | changeset | distributed | merge | changeset[37] |
| Git | projet | snapshot[38] | distributed | merge | hash |

[36] Hybride, avec tags et versions RCS
[37] Darcs est basé sur les patches, pas les fichiers!
[38] Git ne garde pas de diff!



## 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



## Quelques fonctionnalités avancées: bisect/trackdown

Quand on dispose d'un VCS, et on l'utilise correctement (pas de fichiers zombie inconnues du VCS, pas de commits monolithiques), il est possible de repondre à la question: "quel changement a introduit cet erreur?" (et ... "qui est coupable?").
La plupart des VCS populaires, comme Subversion, Git, Darcs et Mercurial, fournissent une fonctionnalité connue comme `bisect` ou `trackdown`.
Voyons un exemple (DEMO).



## 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)

et plusieurs forges spécialisées pour `git`

- GitHub
- Gitorious (en libre, moins sophistiqué)
- GitLab (clone libre de GitHub)

## Le futur

On est quand même encore loin de ce qu'on voudrait.
On aimerait des forges qui permettent:

- *le choix du VCS*
- *la gestion fine des droits*
- l'intégration profonde entre BTS et VCS (closes/fixes #35 dans le commit ferme le bug #35 dans GitHub)
- l'intégration profonde avec les outils de IM (IRC, etc.)
- l'annotation sémantique des bugs et commits
- la migration entre forges



## CI = Continuous Integration

Quand il y a beaucoup de dévéloppeurs, et une activitée soutenue, on peut être confrontés à des problèmes d'*intégration* qui se manifestent au moment de fusionner les modifications faites par nous avec celles dévéloppées indépendemment par d'autres personnes entretemps.

- *conflits* au moment du *merge* (responsabilité du dévéloppeur)
- *fonctionnalités cassées* par des modifications incompatibles (pas forcément en conflit)

Afin de réduire le coût (élévé) de ces problèmes, on a recours à l'Intégration Continue.



## Premier ingrédient: les tests

Afin de mettre en place de l'intégration continue, la première étape est de constituer un jeu de tests. Cela peut aller du simple test que la compilation du code aboutit (la moindre des choses), à des tests sophistiqués visant à controller les *régressions*.

- Tests unitaires (on teste chaque module / fonctionnalité)
- Tests d'intégration (on teste des assemblages de modules)
- Tests de validation (on valide le logiciel complet)



## Deuxième ingrédient: les *buildbots*

La deuxième étape est constitué par des robots de compilation/construction des artefacts logiciels, qui ont pour mission de lancer la construction du code, et d'exécuter les tests.

Cela peut arriver à intevalles régulières (toutes les nuits), ou sur la base d'événements (à chaque commit).



## Troisième ingrédient: le *tableau de bord*

Quand on doit tester un logiciel complexe sur un ensemble de configurations, il est important de présenter les résultats de façon concise aux mainteneurs.

On se sert alors d'un tableau de bord qui résume l'état des exécutions des différentes tâches des robots.



## Un exemple

Voyons un exemple sur le cas de `gihub` avec `travis`.