4 juin 1996

- Vous êtes ingénieur dans à l'agence spatiale européenne.
- Depuis 9 ans, vous travaillez sur les différents modules embarqués dans la fusée Ariane 5
- Aujourd'hui est le grand jour de lancement de la fusée :
  - Kourou / Guyane
  - 370 Millions de $ d'investissement

4 juin 1996

- tout le monde retient son souffle...

Ariane : 4 juin 1996

- bug dans le système de guidage d'Ariane 5 reposant sur des accéléromètres et des gyroscopes.

- Les valeurs excessives d'un accéléromètre produisent un dépassement de capacité dans l'unité de calcul.

Le petit point sur PG106

Vous savez

#QDLE#S#AB#20#

- Expliquer les différentes étapes de compilation ?

A) OUI B) NON

Vous savez

- Expliquer les différentes étapes de compilation ?

- Pré-compilation (gcc -E)
- Compilation (-c)
- Edition de lien

Vous savez

#QDLE#S#AB#20#

- Créer une bibliothèque statique ?

A) OUI B) NON

Vous savez

- Créer une bibliothèque statique ?

- Précompilation + compilation
- Archivage des fichiers objets :

```
ar rcs libtoto.a obj1.o obj2.o obj3.o ...
```

Vous savez

#QDLE#S#AB#20#

- Créer une bibliothèque dynamique ?

A) OUI B) NON

Vous savez

- Créer une bibliothèque dynamique ?

- Précompilation
- Compilation avec l'option -fPIC
- Édition de lien :

```
gcc -shared -o libtoto.so obj1.o obj2.o ...
```

Vous savez

#QDLE#S#AB#20#

- Générer un programme qui utilise une bibliothèque statique ou dynamique ?

A) OUI B) NON

Vous savez

- Générer un programme qui utilise une bibliothèque statique ou dynamique ?

- Statique : lors de l'édition de lien, ajout du chemin de la bibliothèque :

  gcc prog.o libtoto.a

- Dynamique : options -l -L

  gcc prog.o -ltoto -L/chemin/vers/toto

  + utilisation de LD_LIBRARY_PATH à l'exécution si la bibliothèque n'est pas dans les répertoires par défaut.

Vous savez

#QDLE#S#AB#20#

- Nommer, expliquer et utiliser les trois modes de réservation mémoire ?

A) OUI B) NON

Vous savez

- Nommer, expliquer et utiliser les trois modes de réservation mémoire ?

- Statique
- Automatique
- Dynamique

Vous savez

#QDLE#S#AB#20#

- Expliquer ce qu'est un processus, une page mémoire et nommer les différentes zones mémoires ?

A) OUI B) NON

Vous savez

- Expliquer ce qu'est un processus, une page mémoire et nommer les différentes zones mémoires ?

- Processus : programme chargé en mémoire et exécuté
- Zones :
  - Données (statique)
  - Code
  - Bibliothèques dynamiques
  - Pile
  - Tas

Vous savez

#QDLE#S#AB#20#

- Compiler un programme pour utiliser gdb, expliquer comment utiliser gdb, donner les commandes de bases ?

A) OUI B) NON

Vous savez

- Compiler un programme pour utiliser gdb, expliquer comment utiliser gdb, donner les commandes de bases ?
- Options : -g -O0
- Commandes :
  - List - next
  - Break - step
  - Print - condition
  - Display - watch
  - Run - backtrace

Vous savez

#QDLE#S#AB#20#

- Compiler un programme pour utiliser valgrind, expliquer comment utiliser valgrind, donner les erreurs de bases que peut détecter valgrind ?

A) OUI B) NON

Vous savez

- Compiler un programme pour utiliser valgrind, expliquer comment utiliser valgrind, donner les erreurs de bases que peut détecter valgrind ?
- Options : -g -O0
- Erreurs :
  - Accès mémoire en lecture non initialisé
  - Accès mémoire en écriture sur de la mémoire non réservée
  - Fuites mémoires

...

#QDLE#S#ABC#30#

- Pendant les vacances j'ai programmé :

  A) Non

  B) une fois

  C) plusieurs fois

...

#QDLE#S#ABCDE#30#

- La programmation :

  A) J'adore

  B) J'aime bien

  C) Sans plus

  D) J'aime pas trop

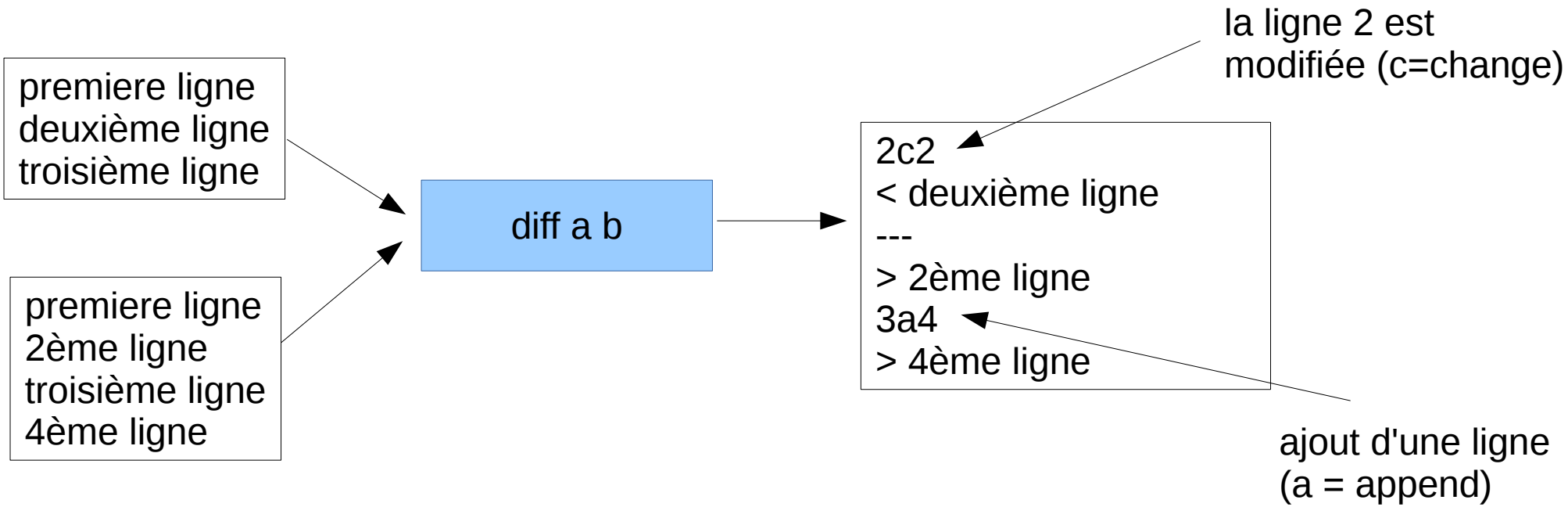
  E) Je déteste

La suite :

- Diff & patch
- Les gestionnaires de sources (svn, etc.)
- Intégration Continue
- Les tests
- La performance

diff

- diff est un outil d'analyse de texte qui compare deux fichiers entre eux et produit le nombre minimum d'édition à faire sur le premier fichier pour obtenir le second :



diff : side by side

- on peut afficher les deux fichiers cote à cote :

```
diff -y a b
```

```
premiere ligne                                  premiere ligne
deuxième ligne                                | 2ème ligne
troisième ligne                                 troisième ligne
                                              >     4ème ligne
```

- diff permet la comparaison récursive de deux arborescences.

diff : recursif

- Je souhaite modifier le code source d'un projet :
  1. je fais un copie de sauvegarde des sources d'origine
  2. j'effectue mes modifications
  3. à tout moment, je visualise mes modifications avec diff : `diff -r projet projet_new`

diff : recursif

- Je souhaite modifier le code source d'un projet :
  1. je fais un copie de sauvegarde des sources d'origine
  2. j'effectue mes modifications
  3. à tout moment, je visualise mes modifications avec diff : `diff -r projet projet_new`

```
diff -r GenTaskLib/TaskParser.cpp GenTaskLib_new/TaskParser.cpp
15c15
<         throw gtl::InvalidArgumentException("json parser error: expected a dictionnary at each task");
---
>         throw gtl::InvalidArgumentException("json parser error: expected a dictionnary for each task");
diff -r GenTaskLib/TaskParser.hpp GenTaskLib_new/TaskParser.hpp
36a37,38
>   /*! TaskParser: build a task from a json description.
>    */
```

Ajout d'un commentaire

diff : algorithme

- Le programme diff repose sur un problème classique d'algorithmique du texte : La plus longue sous-séquence commune

  MOTELS ARE NOT HELL!

  MIROIR TU ES LA!

- Sur cet exemple « MO T » est une sous séquence commune.

diff : algorithme

#QDLE#Q#ABC*D#45#

- Le programme diff repose sur un problème classique d'algorithmique du texte : La plus longue sous-séquence commune

  MOTELS ARE NOT HELL!

  MIROIR TU ES LA!

- Taille de la plus longue sous séquence commune ? (les espaces comptes).

  A. 6 B. 8 C. 9 D. 10

diff : algorithme

- Le programme diff repose sur un problème classique d'algorithmique du texte : La plus longue sous-séquence commune

  MOTELS_ARE_NOT_HELL!

  MIROIR_TU_ES_LA!

diff : algorithm

- Chaque fichier est découpé en ligne
- Les lignes sont comparées entre elles (LCS entre chaque ligne)
- Puis l'ensemble des lignes sont comparées entre elles (LCS où chaque symbole représente une ligne).

diff et communication

- Ainsi, si l'on souhaite communiquer une modification, il suffit d'envoyer le résultat d'un diff récursif : `diff -rupN original new > patch`
- On appel ce fichier un patch.
- Le destinataire peut lire ce fichier et comprendre vos modifications
- Il peut également appliquer ces modifications en local grâce au programme patch

- *les options upN sont nécessaire au fonctionnement de patch, elles ajoutent des informations de contexte (u), de fonction (p), d'ajout de fichier (N)*

patch

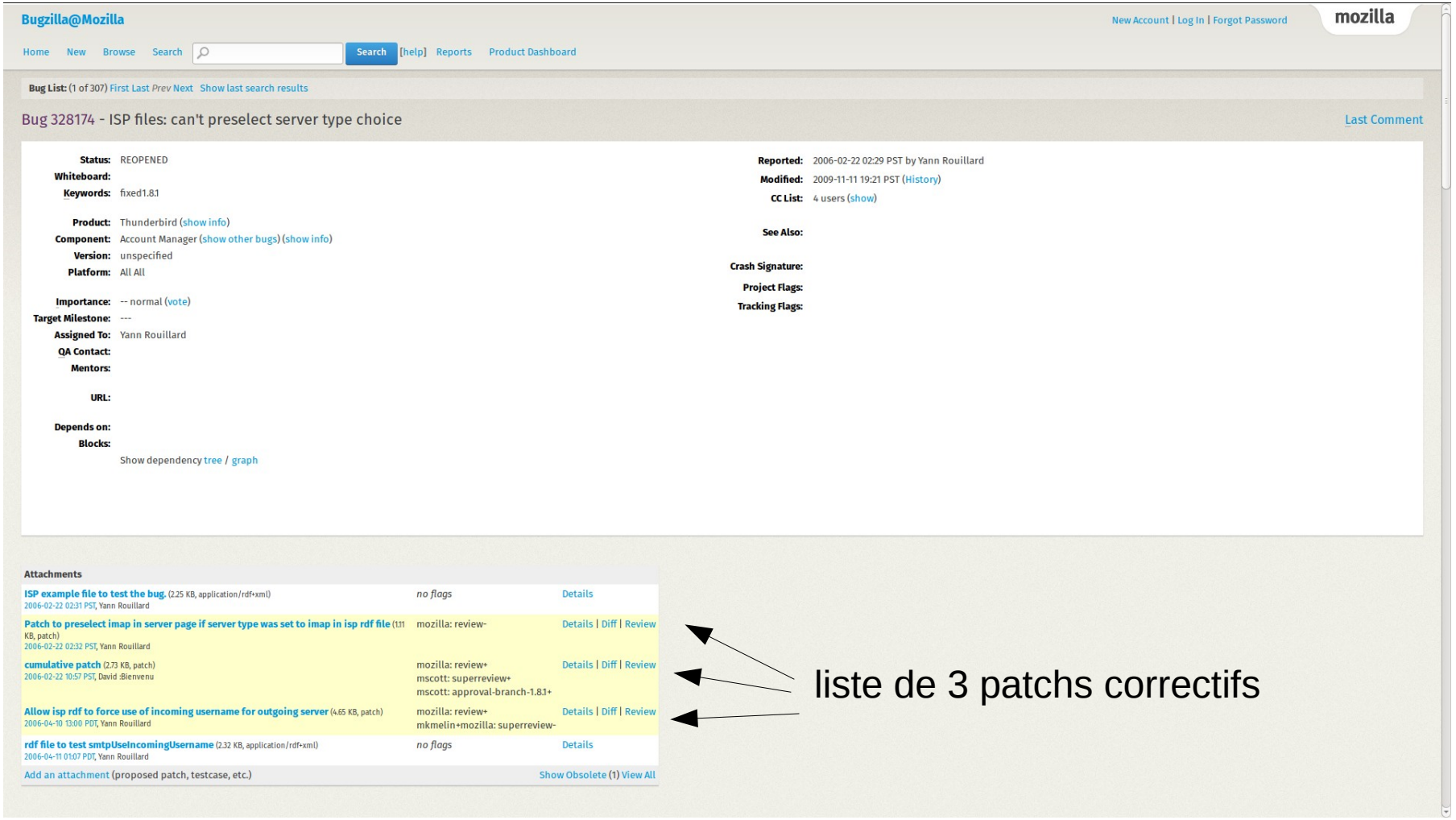
- le programme patch permet d'appliquer les modifications identifiées par diff.
- Ainsi sur l'exemple précédent je peux faire :

```
$ cp -R GenTaskLib GenTaskLibMod
$ cd GenTaskLibMod
$ patch < ../patch
patching file TaskParser.cpp
patching file TaskParser.hpp
$ cd .. ; diff -r GenTaskLib GenTaskLibMod
```

```
diff -r GenTaskLib/TaskParser.cpp GenTaskLibMod/TaskParser.cpp
15c15
<         throw gtl::InvalidArgumentException("json parser error: expected a dictionnary at each task");
---
>         throw gtl::InvalidArgumentException("json parser error: expected a dictionnary for each task");
diff -r GenTaskLib/TaskParser.hpp GenTaskLibMod/TaskParser.hpp
36a37,38
>    /*! TaskParser: build a task from a json description.
>     */
```

diff & patch

- Dans le monde open source, diff et patch sont extrêmement utilisés



gestion de source

- Un gestionnaire de source permet de conserver l'historique des modifications apportées à un ensemble de fichier.
- Il permet à un ensemble d'utilisateurs d’interagir sur un même code source.
- Tous les gestionnaires de codes sources sont basés sur les principes de diff et patch
- Il existe deux grandes catégories de gestionnaires :
  - les centralisés
  - les décentralisés

Les gestionnaires centralisés

- Historiquement, cvs (concurent versioning system) et son successeur svn (subversion)
- Les gestionnaires centralisés reposent sur un serveur central qui archive toutes les modifications apportées au code.
- Chaque modification incrémente un numéro de révision
- Les commandes de base de svn :
  - checkout, update, infos, status, commit, diff, revert

svn

- Chaque utilisateur interagit avec le serveur, il n'y a pas d'échange direct entre deux utilisateurs.
- Une méthodologie est associée à l'utilisation de svn, vous retrouverez cet méthodologie dans tout projet de développement (open source ou entreprise).

svn : méthodologie

- Il est possible d'utiliser SVN juste pour le répertoire de développement principal.
- Seulement, comment faire si on a un « gros » développement à produire: si on transmet les modifications intermédiaires, le programme devient instable (ne compile plus par exemple...)
- Comment faire également pour gérer les versions:
  - On sort une version 1.0 qui évolue en 1.1 puis 1.2
  - On sort la version 2.0 (« casse » la compatibilité avec 1.x)
  - Un bug est trouvé dans la version 2.0: il faut le corriger dans la version courante mais également dans la version 1.x!

svn : méthodologie

- Aussi, il est nécessaire de maintenir plusieurs « répertoires » de développement parallèle.
- Une méthodologie classique organise les sources ainsi:
  - / « racine »
  - /trunk : contient la version en cours des sources (dev.)
  - /branches/: des copies de trunk (« svn copy »)
  - /branches/1.0 : copie de trunk pour release (tests), retour de modif dans le trunk avec « svn merge » si compatible
  - /tags/1.0.0: version **figée** d'une branche, sert de référence, est diffusée. La branche correspondante est etiquetée (tag). **pas de commit/modifs dans ce répertoire**
  - /branches/modif_allali_155/: copie temporaire pour de « grosses » modification avec retour dans le trunk par « merge ». Synchronisation régulière depuis le trunk (« merge »)

svn : la création de dépôt

- Le création d'un dépôt se fait à l'aide de la commande « svnadmin create nom_de_depot »
- possiblité d'ajouter l'execution de script avant/pendant et après les « commits »
- possiblité d'envoie de mails lors des commits
- facile à mettre en place sur son compte:
- cd ~/.depots/ ; svnadmin create SVN
- cd ~/; svn co file:///$HOME/.depots/SVN
- via ssh:
- svn co svn+ssh://allali@ssh.enseirb.fr/.depots/SVN

Les gestionnaires dé-centralisés

- Dans ce cas, il n'y a pas de dépôt centrale.
- Chaque utilisateur gère son propre dépôt.
- Un protocole permet l'échange de modification (commit) entre deux utilisateurs.
- Exemple : git
- Commandes de base :
  - clone, add, commit, push, pull, checkout

git : les commits, pull et push

- Dans git, les commits sont locaux (il n'y a pas de dépôt centrale).
- On peut transmettre un ensemble de commit à un autre utilisateur avec la commande push
- On peut réceptionner un ensemble de commit depuis un autre utilisateur avec la commande pull.

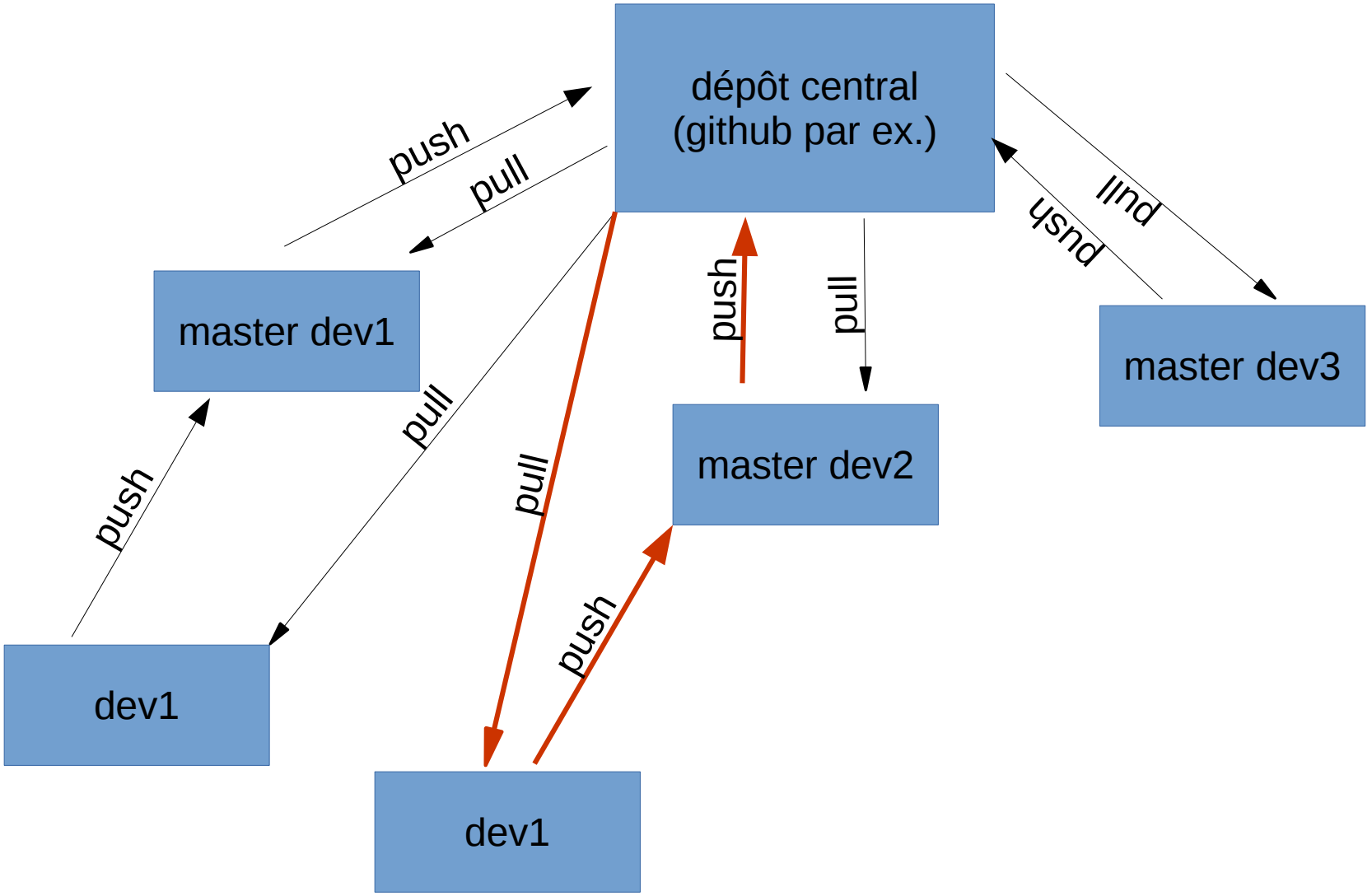
git

- git intègre une gestion de branches :
  - création :
    - git branch b2
    - git checkout b2
    - ou bien git checkout -b b2
  - la fusion :
    - on bascule dans la branche qui doit recevoir les modifs
    - git checkout master ; git merge b2
    - Les modifications doivent avoir été commité dans b2
  - la délétion : git branch -d b2
- La branche par défaut s'appelle master

re-centralisation

- L'avantage d'être en décentraliser et de pouvoir faire des « commit » sans connexion à un serveur.
- Pour la plus part de projet, il est cependant nécessaire d'avoir une référence : on utilise alors un dépôt git comme tel (github par exemple).
- On peut ensuite mettre en place un système de propagation hiérarchique des « commits ».

git


dépôt central
(github par ex.)
push
pull
master dev1
push
pull
dev1
push
pull
master dev2
push
pull
dev1
llnd
ysnd
master dev3

contrôle dans svn

- Il est aussi possible d'avoir cette approche hiérarchique dans svn en subdivisant le répertoire « branches » en répertoire et en ajustant les droits :
  - seuls les masters peuvent commiter dans « trunk »
  - les dev travaillent dans des branches
- Différences majeures entre svn et git est :
  - centralisé / dé-centralisé
  - support natif du système de branches dans git
  - commit locaux dans git

Les autres gestionnaires

	Open Source	Centralisé	Décentralisé
CVS	●	●	
SVN	●	●	
GIT	●		●
SourceSafe		●	
Mercurial	●		●
Bazaar	●		●
BitKeeper			●
Team Foundation Server		●	

Intégration Continue

L'Intégration Continue

- Lors que l'on développe, on est sur un système particulier :
  - type de système (unix, linux, windows, macosx, etc.)
  - version du compilateur
  - version des bibliothèques
  - environnement général (ressources…)
- Avant de transmettre une modification (commit), le développeur doit s'assurer que ses modifications fonctionnent pour l'ensemble des systèmes/configurations cibles.

L'Intégration Continue

- Pour cela, on dispose d'un ensemble de machines.
  - Solution 1 : avant de transmettre mes modifications, je me connecte sur chacune des machines et je testes.
  - Solution 2 : j'utilise un système qui fait cela automatiquement pour moi !

    ⇒ C'est ce que l'on appelle l'Intégration Continue.

  - l'IC *garantie* une stabilité des développements au fur et à mesure. Cela permet de contrôler certaines dettes techniques.

L'Intégration Continue

- Il existe plusieurs plateformes d'intégration continue :
  - Jenkins (successeur de Hudson, java-open source)
  - TeamCity (JetBrain, commercial)
  - CruiseControl (java-open source)
  - Team Foundation Server (Microsoft, commercial)
  - Travis IC (online IC for github projects).
  - ...

L'Intégration Continue

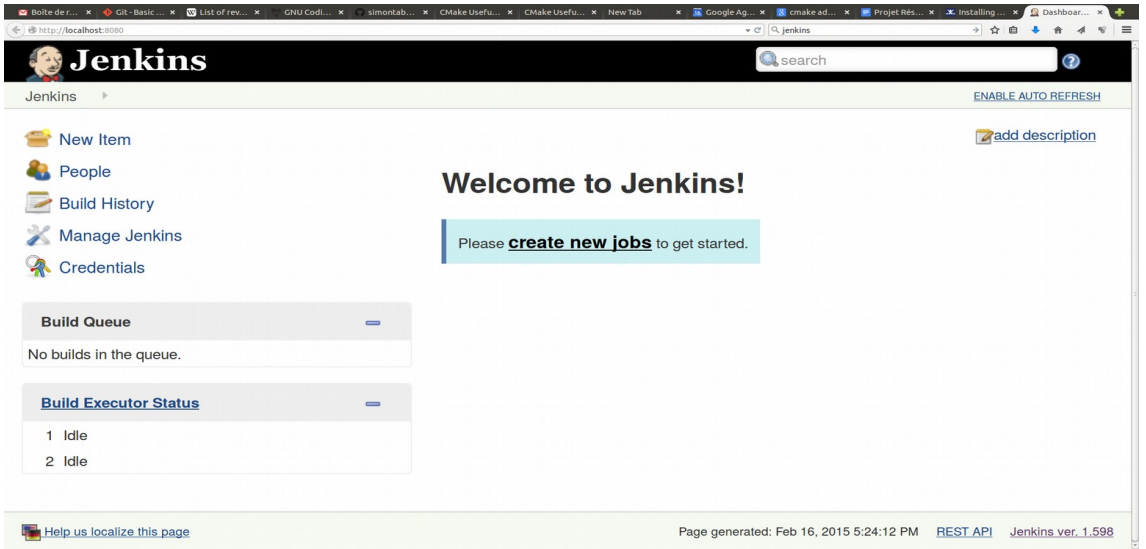
- Un serveur d'intégration continue va se synchroniser avec un dépôt
- A intervalle régulier, il va vérifier que le dépôt est à jour. Si une mise à jour est intervenue, il va effectuer une série de taches (compilation par exemple).
- En fonction du résultat des taches, le serveur va indiquer l'état du projet et possiblement transmettre des alertes.
- Afin de gérer plusieurs environnements, le serveur d'IC va piloter des clients sur lesquels il lancera les taches (via ssh par exemple)

L'Intégration Continue

- La qualité qu'offre l'IC va dépendre principalement de deux facteurs :
  - la nature et la diversité des clients (environnement de validation)
  - la complexité des taches à réaliser :
    - de la compilation
    - à l'exécution de taches complexes de validation
- Le serveur d'IC peut également rendre compte de facteurs comme les ressources utilisées (cpu, temps, mémoire).

Exemple avec Jenkins

- répertoire projet :
  - projet/ :
    - makefile
    - main.c
- installation de jenkins et connexion au port 8080 sur localhost :



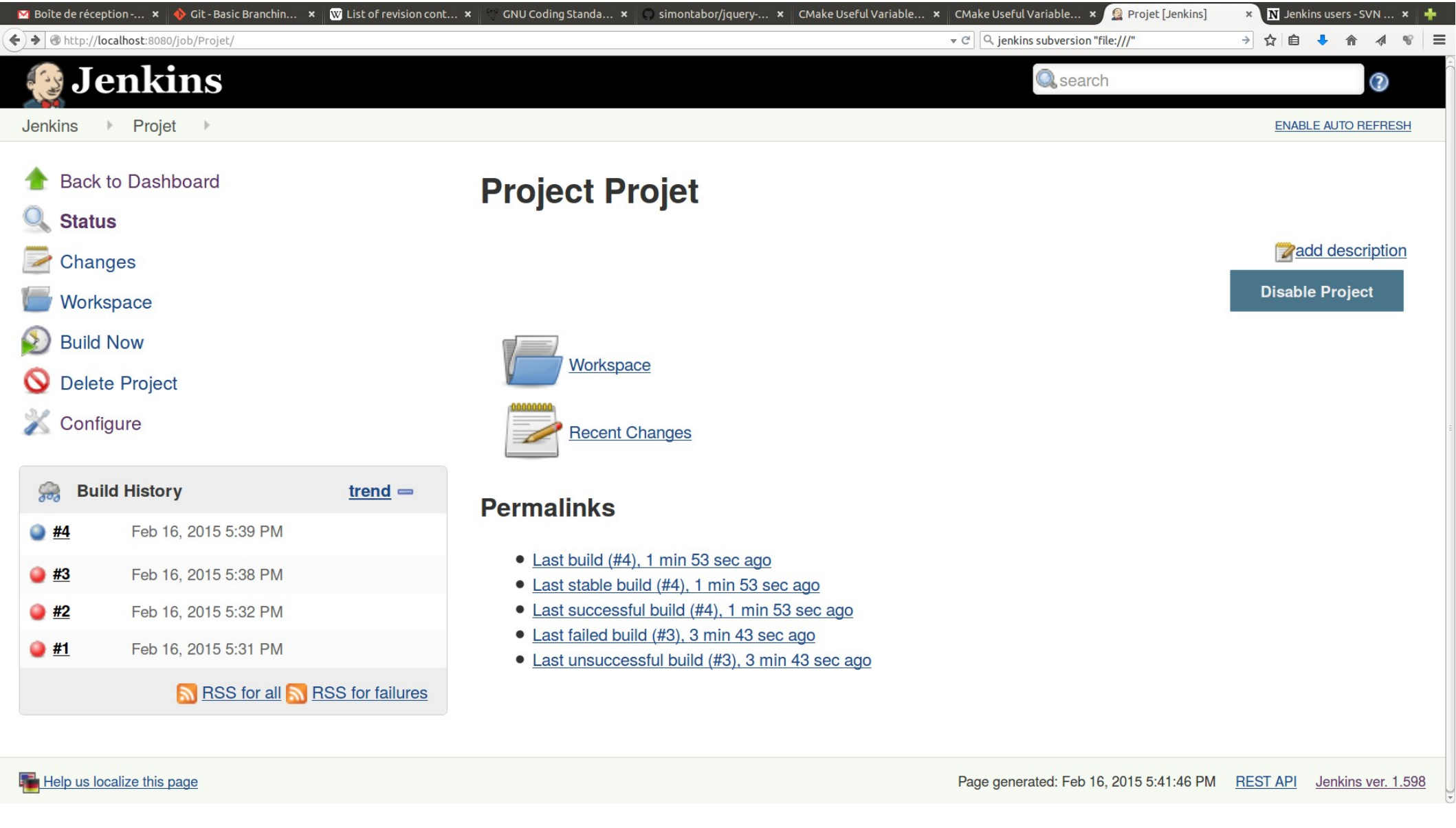
Jenkins : exemple

- création d'un dépôt local avec les sources :

  svnadmin create /tmp/projet

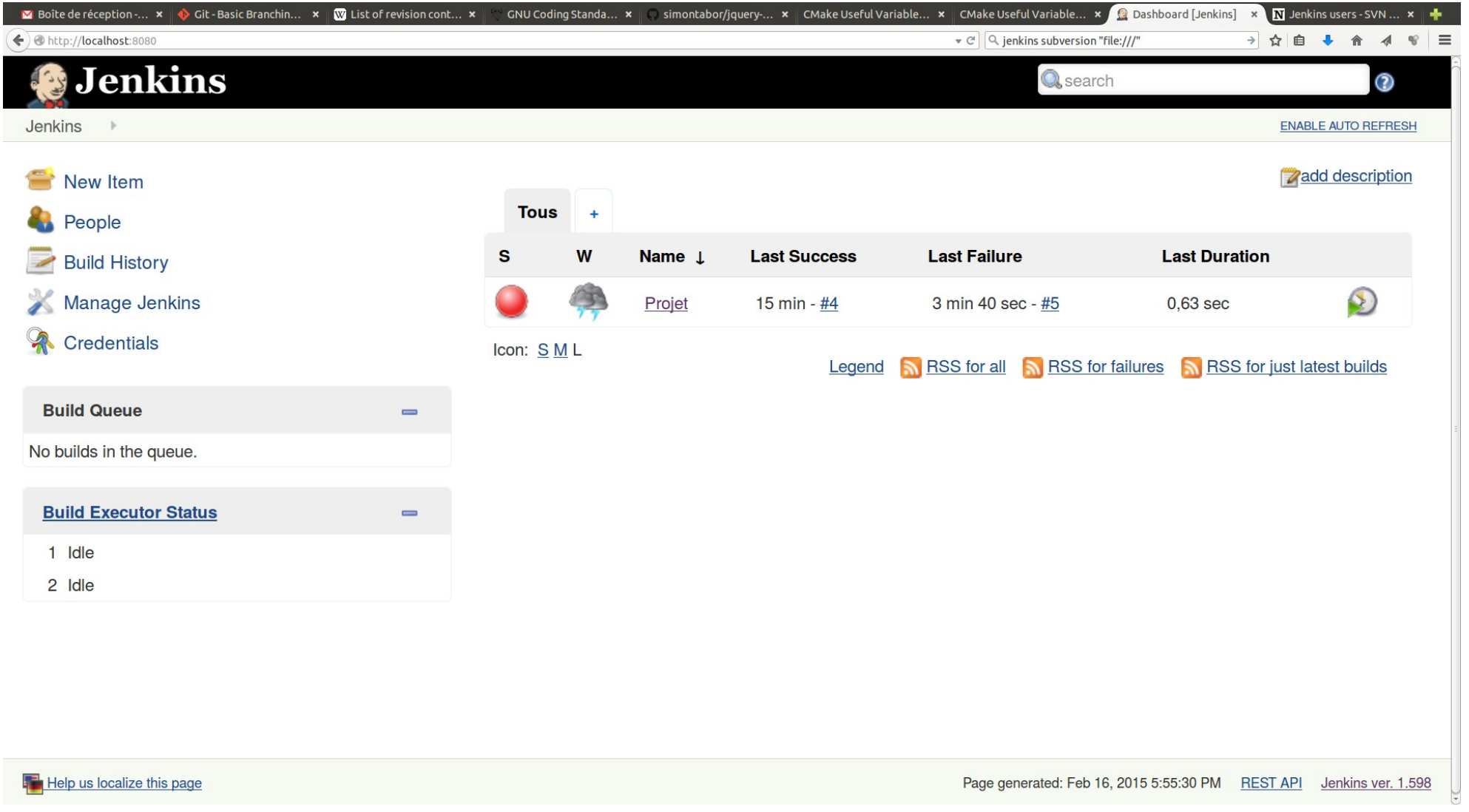
  checkout + ajout des sources + commit
- Ajout du dépôt dans Jenkins en utilisant comme url `svn+ssh://localhost/tmp/projet/`
- Ajout comme commande de build : make
- Lancement d'un build dans jenkins

Jenkins : statut du projet

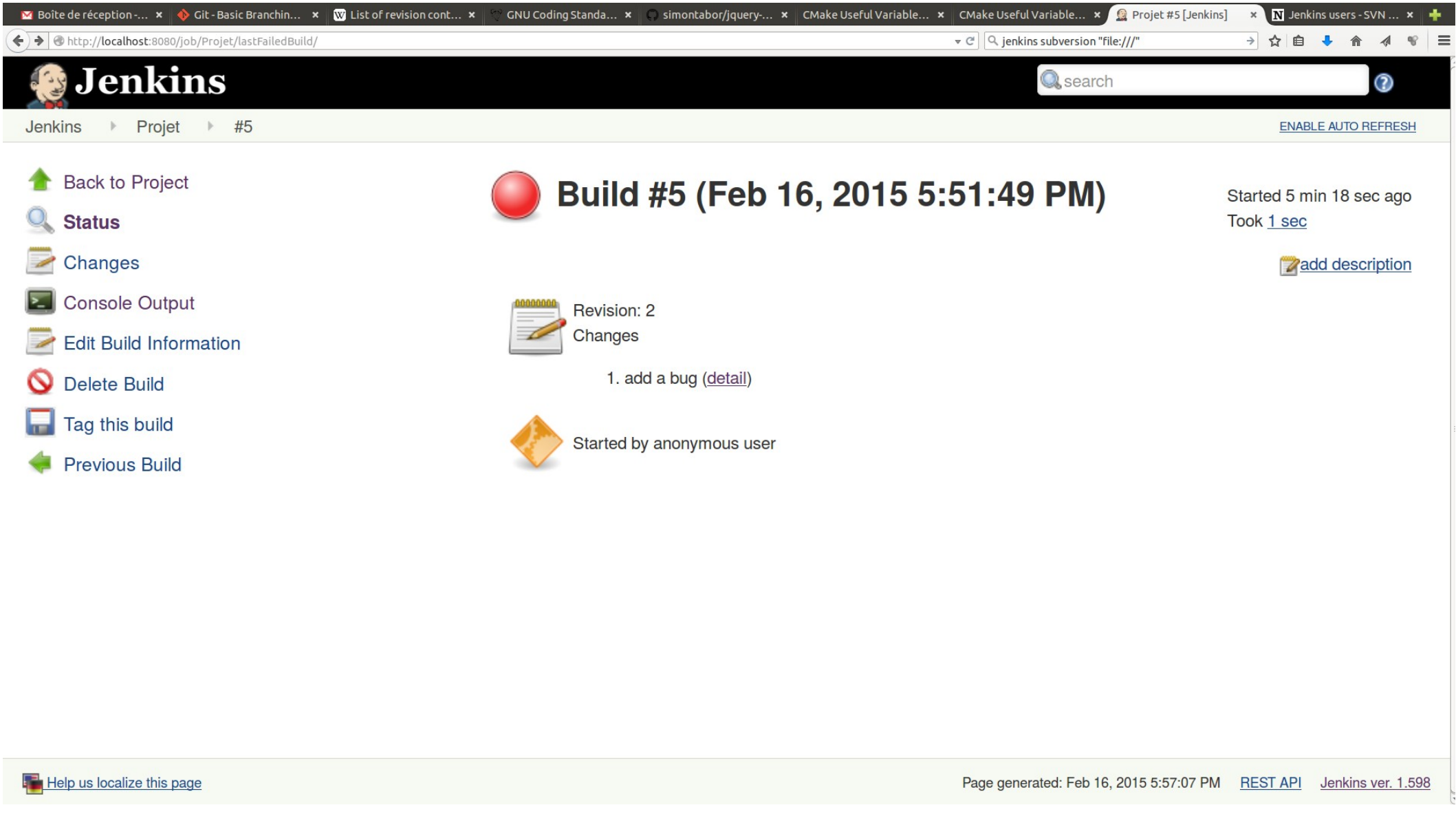


Jenkins : commit

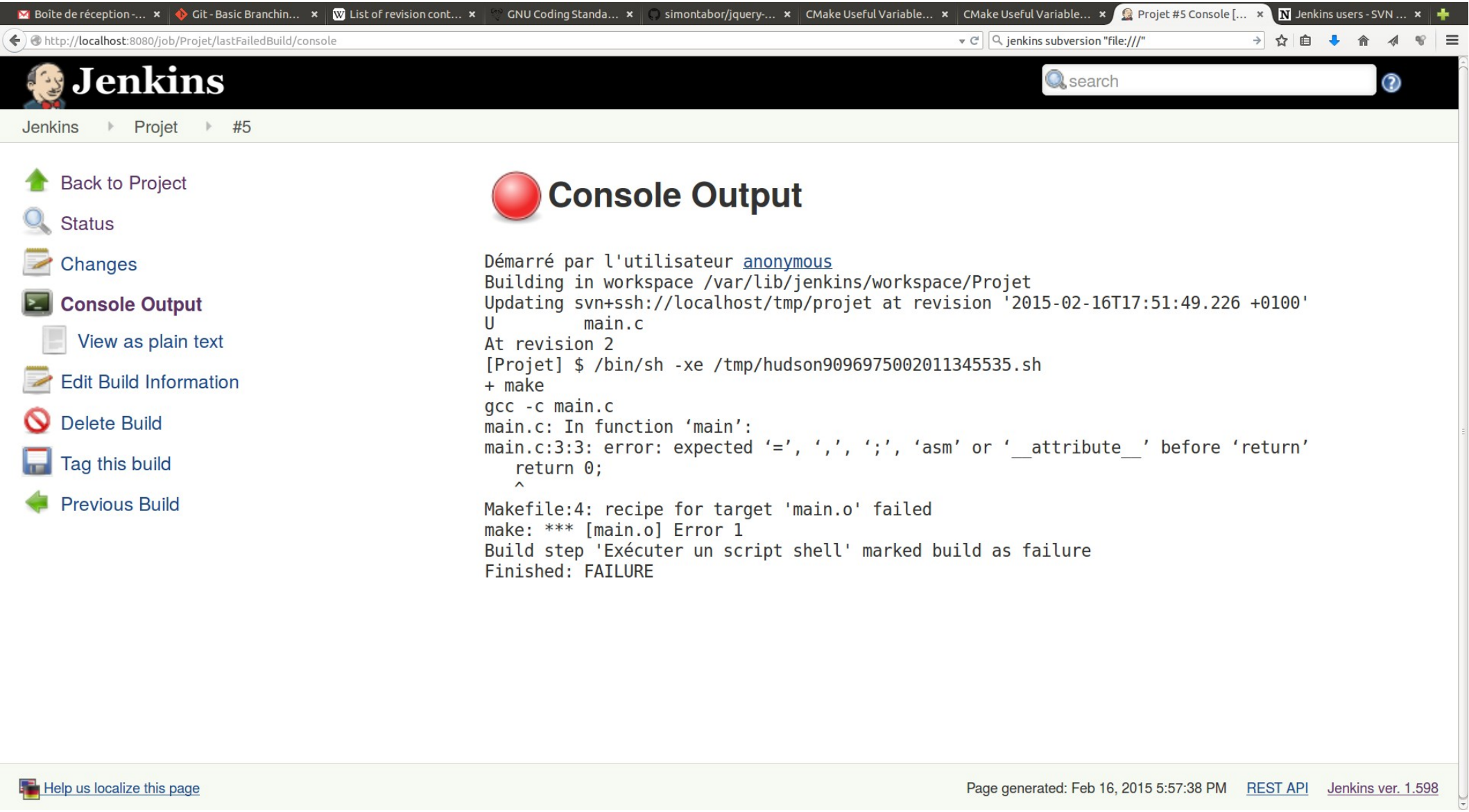
- Ajout d'un bug, commit dans le dépôt



Jenkins : bug


Boîte de réception -...
Git - Basic Branchin...
List of revision cont...
GNU Coding Standa...
simontabor/jquery-...
CMake Useful Variable...
CMake Useful Variable...
Projet #5 [Jenkins]
Jenkins users - SVN ...
http://localhost:8080/job/Projet/lastFailedBuild/
jenkins subversion "file:///"
Jenkins
search
Jenkins
Projet
#5
ENABLE AUTO REFRESH
Back to Project
Status
Changes
Console Output
Edit Build Information
Delete Build
Tag this build
Previous Build
Build #5 (Feb 16, 2015 5:51:49 PM)
Started 5 min 18 sec ago
Took 1 sec
add description
Revision: 2
Changes
1. add a bug (detail)
Started by anonymous user
Help us localize this page
Page generated: Feb 16, 2015 5:57:07 PM
REST API
Jenkins ver. 1.598

Jenkins : bug


Boîte de réception -...
Git - Basic Branchin...
List of revision cont...
GNU Coding Standa...
simontabor/jquery-...
CMake Useful Variable...
CMake Useful Variable...
Projet #5 Console [...
Jenkins users - SVN ...
http://localhost:8080/job/Projet/lastFailedBuild/console
jenkins subversion "file:///"
Jenkins
search
Jenkins
Projet
#5
Back to Project
Status
Changes
Console Output
View as plain text
Edit Build Information
Delete Build
Tag this build
Previous Build
Console Output
Démarré par l'utilisateur anonymous
Building in workspace /var/lib/jenkins/workspace/Projet
Updating svn+ssh://localhost/tmp/projet at revision '2015-02-16T17:51:49.226 +0100'
U main.c
At revision 2
[Projet] $ /bin/sh -xe /tmp/hudson9096975002011345535.sh
+ make
gcc -c main.c
main.c: In function ‘main’:
main.c:3:3: error: expected ‘=’, ‘,’, ‘;’, ‘asm’ or ‘__attribute__’ before ‘return’
return 0;
^
Makefile:4: recipe for target 'main.o' failed
make: *** [main.o] Error 1
Build step 'Exécuter un script shell' marked build as failure
Finished: FAILURE
Help us localize this page
Page generated: Feb 16, 2015 5:57:38 PM
REST API
Jenkins ver. 1.598

Intégration Continue

- Tester que le programme compile sous plusieurs environnements est bien mais cela n'offre que peu de garantie quand à l'état fonctionnel du projet.
- Pour *garantir* une qualité tout au long du développement, il faut ajouter des **tests**

Les Tests

Les Tests

- Il existe de nombreux types de tests
- Les tests ont pour objectifs de valider votre code :
  - au niveau d'une fonction : tests unitaires
  - au niveau d'un module : tests fonctionnels
  - entre plusieurs modules : tests d'intégration
  - au niveau général, applicatif : tests de recette

TDD : test driven development

- La méthodologie TDD repose sur l'écriture d'abord de tests puis du code validant les tests.
- La méthode XP (extreme programming) repose en partie sur partie sur TDD.
- TDD repose sur des cycles courts consistant :
  - à écrire un test fonctionnel
  - vérifier que le test plante
  - à écrire un test unitaire
  - vérifier que le test plante
  - écrire le code minimal pour que le test fonctionne
  - vérifier que le test passe

TDD

- A la fin de l'écriture d'un code et lorsque tout les tests passent, on peut vouloir refactoriser votre code (copier/coller, simplification, unification, …)
- Dans ce cas, on ne touche surtout pas aux tests et on remanie le code jusqu'à ce que à nouveau il valide l'ensemble des tests.

TDD par l'exemple : bowling

- Supposons que l'on souhaite écrire un module de calcul de feuille de score de bowling :

exemple grandement inspirée de la présentation « TDD in C » par Olve Maudal (dispo. slideshare)

bowling

- Le joueur a 10 sets
- Pour chaque set, le joueur lance la boule une ou deux fois:
  - Si le joueur élimine les 10 quilles du premier coup, il fait strike et il ne joue pas de 2ème boule
  - Si le joueur élimine les 10 quilles au deuxième coup, il fait spare
- Le score d'un set fait :
  - le score précédent + la somme des deux lancés si pas de strike ni de spare
  - le score précédent + 10 + le score du premier lancé du prochain set si spare

bowling

- La spécification client est d'avoir un module BowlingGame avoir les méthodes suivantes
  - void roll(BowlingGame *,int nbPinsDown) : enregistre un nouveau lancé
  - int score(BowlingGame *) : renvoie le score actuel

bowling

- Tout d'abord il nous faut une structure pour modéliser une partie :
  - struct Game
- puis une structure pour modéliser un set
  - struct Set
- Une partie est composée de 10 sets :

```
struct Game {
    struct Set sets[10] ;
}
```

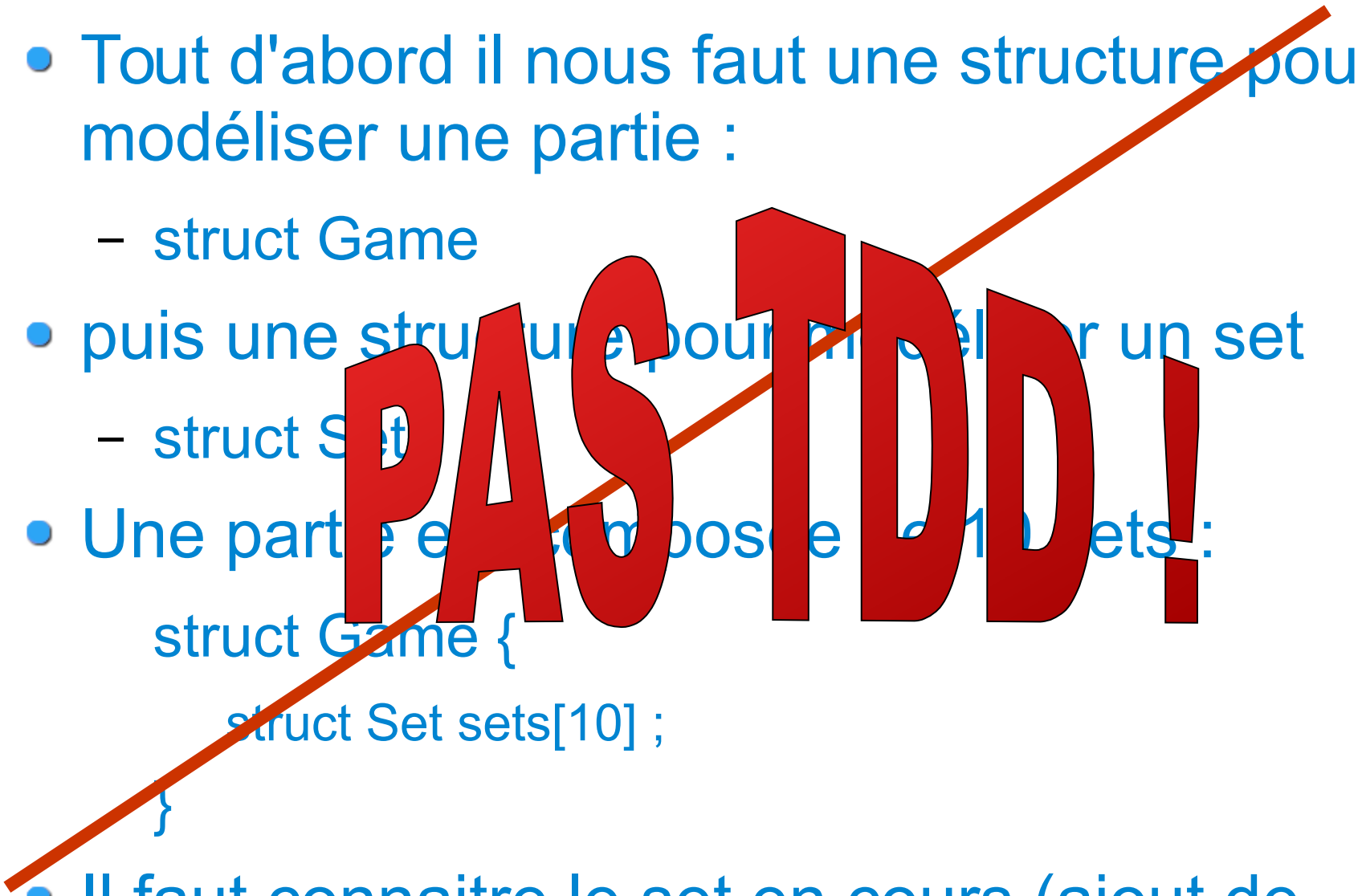
- Il faut connaitre le set en cours (ajout de currentSet dans Game)

bowling

- Tout d'abord il nous faut une structure pour modéliser une partie :
  - struct Game
- puis une structure pour modéliser un set
  - struct Set
- Une partie est composée de 10 sets :

```
struct Game {
    struct Set sets[10] ;
}
```

- Il faut connaitre le set en cours (ajout de currentSet dans Game)



bowling : en TDD

- En TDD on décrit ce que doit faire le système plutôt que comment le faire
- On commence « basique » :

bowling : en TDD

- En TDD on décrit ce que doit faire le système plutôt que comment le faire
- On commence « basique » :

```
#include<assert.h>
#include<stdbool.h>

int main(){
	assert(false && « c'est parti ») ;
}
```

ok, le système de test fonctionne !

```
$ make bow
gcc -Wall  bow.c   -o bow
$ ./bow
bow: bow.c:5: main: Assertion `0 && "c'est parti"' failed.
```

bowling : cas vide

- Commençons par le cas vide

```
#include<assert.h>
#include<stdbool.h>

void test_empty(){
  int i;
  struct BowlingGame *game=bg_init();
  for(i=0;i<20;++i)
    bg_roll(game,0);
  assert(bg_score(game)==0 && "test empty");
  bg_free(game) ;
}

int main(){
  test_empty();
}
```

```
$ gcc -Wall bow.c
bow.c: In function ‘test_empty’:
bow.c:6:10: warning: implicit declaration of function ‘bg_init’ [-Wimplicit-function-declaration]
   struct BowlingGame *game=bg_init();
          ^
bow.c:6:28: warning: initialization makes pointer from integer without a cast
   struct BowlingGame *game=bg_init();
```

Ajout de bowling.h

```
#ifndef BOWLING_H
#define BOWLING_H


struct BowlingGame;

struct BowlingGame *bg_init();
void bg_roll(struct BowlingGame *,int );
int bg_score(struct BowlingGame *);
void bg_free(struct BowlingGame *) ;

#endif
```

bowling : cas vide

```
$ gcc -Wall bow.c
/tmp/ccTstSlJ.o: In function `test_empty':
bow.c:(.text+0xe): undefined reference to `bg_init'
bow.c:(.text+0x2c): undefined reference to `bg_roll'
bow.c:(.text+0x42): undefined reference to `bg_score'
bow.c:(.text+0x6b): undefined reference to `bg_free'
collect2: error: ld returned 1 exit status
```


Ajout de bowling.c

bowling : cas vide

```
$ gcc -Wall bow.c
/tmp/ccTstSlJ.o: In function `test_empty':
bow.c:(.text+0xe): undefined reference to `bg_init'
bow.c:(.text+0x2c): undefined reference to `bg_roll'
bow.c:(.text+0x42): undefined reference to `bg_score'
bow.c:(.text+0x6b): undefined reference to `bg_free'
collect2: error: ld returned 1 exit status
```

Ajout de bowling.c

```
#include "bowling.h"
#include <stdlib.h>

struct BowlingGame{};

struct BowlingGame *bg_init(){
  return NULL;
}
void bg_roll(struct BowlingGame *g,int s){
}
int bg_score(struct BowlingGame *g){
  return -1;
}
void bg_free(struct BowlingGame *g){}
```

```
$ gcc -Wall bow.c bowling.c
allali@hebus:/tmp/bowling$ ./a.out
a.out: bow.c:10: test_empty: Assertion `bg_score(game)==0 && "test empty"' failed.
```

bowling : cas vide

```
$ gcc -Wall bow.c bowling.c
allali@hebus:/tmp/bowling$ ./a.out
a.out: bow.c:10: test_empty: Assertion `bg_score(game)==0 && "test empty"' failed.
```

Ajout du score

```
$ gcc bowling.c bow.c
-Wall
$ ./a.out
$ valgri
ok
```

```
#include "bowling.h"
#include <stdlib.h>

struct BowlingGame{
        int score ;
};

struct BowlingGame *bg_init(){
  struct BowlingGame * g=malloc(sizeof(* g)) ;
  g->score=0 ;
  return g ;
}
void bg_roll(struct BowlingGame *g,int s){
}
int bg_score(struct BowlingGame *g){
  return g->score;
}
void bg_free(struct BowlingGame *g){
  free(g) ;
}
```

bowling : test tout à 1

```
#include<assert.h>
#include<stdbool.h>

void test_empty(){ … }

void test_all_ones(){
  int i;
  struct BowlingGame *game=bg_init();
  for(i=0;i<20;++i)
    bg_roll(game,1);
  assert(bg_score(game)==20 && "test all ones");
  bg_free(game) ;
}

int main(){
  test_empty();
  test_all_ones() ;
}
```

```
$ ./a.out
a.out: bow.c:19: test_all_ones: Assertion `bg_score(game)==20 && "test all ones"' failed.
```

bowling : test tout à 1

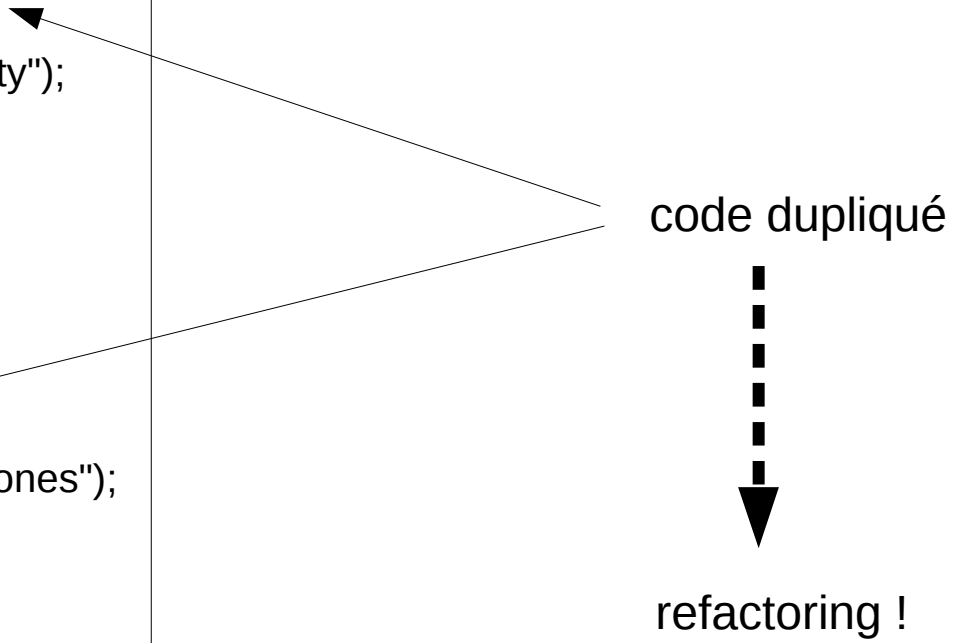
```
$ ./a.out
a.out: bow.c:19: test_all_ones: Assertion `bg_score(game)==20 && "test all ones"' failed.
```

```
$ gcc bowling.c bow.c -Wall
$ ./a.out
$
```

```
#include "bowling.h"
#include <stdlib.h>

struct BowlingGame{
        int score ;
};

struct BowlingGame *bg_init(){
  struct BowlingGame * g=malloc(sizeof(* g)) ;
  g->score=0 ;
  return g ;
}
void bg_roll(struct BowlingGame *g,int s){
   g->score+=s ;
}
int bg_score(struct BowlingGame *g){
  return g->score;
}
void bg_free(struct BowlingGame *g){
  free(g) ;
}
```

bowling : code smell...

```
void test_empty(){
  int i;
  struct BowlingGame *game=bg_init();
  for(i=0;i<20;++i)
    bg_roll(game,0);
  assert(bg_score(game)==0 && "test empty");
  bg_free(game);
}

void test_all_ones(){
  int i;
  struct BowlingGame *game=bg_init();
  for(i=0;i<20;++i)
    bg_roll(game,1);
  assert(bg_score(game)==20 && "test all ones");
  bg_free(game);
}

int main(){
  test_empty();
  test_all_ones();
  return 0;
}
```



bowling : code smell...

```
void test_empty(){
  int i;
  struct BowlingGame *game=bg_init();
  for(i=0;i<20;++i)
    bg_roll(game,0);
  assert(bg_score(game)==0 && "test empty");
  bg_free(game);
}

void test_all_ones(){
  int i;
  struct BowlingGame *game=bg_init();
  for(i=0;i<20;++i)
    bg_roll(game,1);
  assert(bg_score(game)==20 && "test all ones");
  bg_free(game);
}

int main(){
  test_empty();
  test_all_ones();
  return 0;
}
```

```
void rolls(struct BowlingGame *game,int n, int v){
  int i;
  for(i=0;i<n;++i)
    bg_roll(game,v);
}

void test_empty(){
  struct BowlingGame *game=bg_init();
  rolls(game,20,0);
  assert(bg_score(game)==0 && "test empty");
  bg_free(game);
}

void test_all_ones(){
  struct BowlingGame *game=bg_init();
  rolls(game,20,1);
  assert(bg_score(game)==20 && "test all ones");
  bg_free(game);
}

int main(){
  test_empty();
  test_all_ones();
  return 0;
}
```

bowling : un spare

```
void test_one_spare(){
  struct BowlingGame *game=bg_init();
  bg_roll(game,5);
  bg_roll(game,5);
  bg_roll(game,3);
  rolls(game,17,0);
  assert(bg_score(game)==16 && "test one spare");
}
```

```
$ gcc bow.c bowling.c -Wall
allali@hebus:~/SVN_LaBRI/ENSEIRB/PG106/Cours$ ./a.out
a.out: bow.c:31: test_one_spare: Assertion `bg_score(game)==16 && "test one spare"' failed.
```

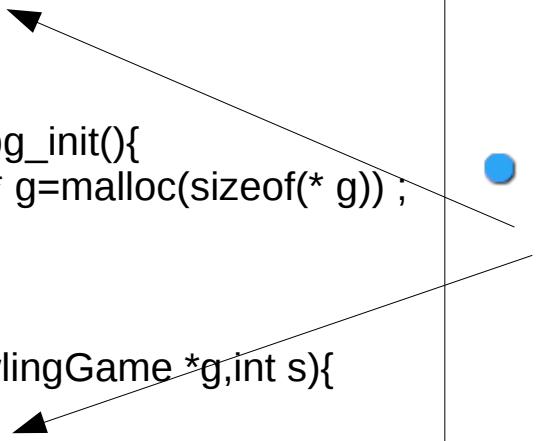
bowling : conception

```
#include "bowling.h"
#include <stdlib.h>

struct BowlingGame{
        int score ;
};

struct BowlingGame *bg_init(){
  struct BowlingGame * g=malloc(sizeof(* g)) ;
  g->score=0 ;
  return g ;
}
void bg_roll(struct BowlingGame *g,int s){
   g->score+=s ;
}
int bg_score(struct BowlingGame *g){
  return g->score;
}
void bg_free(struct BowlingGame *g){
  free(g) ;
}
```

- Pour gérer un spare, il faut connaitre le coup d'avant.

bowling : conception

```
#include "bowling.h"
#include <stdlib.h>

struct BowlingGame{
        int score ;
};

struct BowlingGame *bg_init(){
  struct BowlingGame * g=malloc(sizeof(* g)) ;
  g->score=0 ;
  return g ;
}
void bg_roll(struct BowlingGame *g,int s){
   g->score+=s ;
}
int bg_score(struct BowlingGame *g){
  return g->score;
}
void bg_free(struct BowlingGame *g){
  free(g) ;
}
```



- Pour gérer un spare, il faut connaitre le coup d'avant.
- On pourrait ajouter un temporaire pour cela

bowling : conception

```
#include "bowling.h"
#include <stdlib.h>

struct BowlingGame{
        int score ;
};

struct BowlingGame *bg_init(){
  struct BowlingGame * g=malloc(sizeof(* g)) ;
  g->score=0 ;
  return g ;
}
void bg_roll(struct BowlingGame *g,int s){
   g->score+=s ;
}
int bg_score(struct BowlingGame *g){
  return g->score;
}
void bg_free(struct BowlingGame *g){
  free(g) ;
}
```

- Pour gérer un spare, il faut connaitre le coup d'avant.
- On pourrait ajouter un temporaire pour cela
- il y a un problème de conception :
  - roll : calcule le score mais ne devrait pas
  - score : doit calculer le score mais ne le calcul pas

bowling : conception

```
#include "bowling.h"
#include <stdlib.h>

struct BowlingGame{
        int score ;
};

struct BowlingGame *bg_init(){
  struct BowlingGame * g=malloc(sizeof(* g)) ;
  g->score=0 ;
  return g ;
}
void bg_roll(struct BowlingGame *g,int s){
   g->score+=s ;
}
int bg_score(struct BowlingGame *g){
  return g->score;
}
void bg_free(struct BowlingGame *g){
  free(g) ;
}
```

Refactoring !

- Pour gérer un spare, il faut connaitre le coup d'avant.
- On pourrait ajouter un temporaire pour cela
- il y a un problème de conception :
  - roll : calcule le score mais ne devrait pas
  - score : doit calculer le score mais ne le calcul pas

bowling : refactoring

- On revient en arrière :

```
int main(){
  test_empty();
  test_all_ones();
  // test_one_spare() ;
  return 0;
}
```

```
$ gcc bow.c bowling.c -Wall
$ ./a.out
$
```

- On modifie le code : ajout d'un tableau de score, du coup en cours et mise à jour de la fonction de calcul

bowling : refactoring

```
#include "bowling.h"
#include <stdlib.h>

struct BowlingGame{
        int rolls[21] ;
        int current;
        int score ;
};

struct BowlingGame *bg_init(){
  struct BowlingGame * g=malloc(sizeof(* g)) ;
  g->current=0 ;
  return g ;
}
void bg_roll(struct BowlingGame *g,int s){
  g->score+=s ;
   g->rolls[g->current++]=s ;
}
int bg_score(struct BowlingGame *g){
  int score=0,i ;
  for(i=0;i<g->current;++i) score+=g->rolls[i] ;
  return score ;
 return g->score;
}
void bg_free(struct BowlingGame *g){
  free(g) ;
}
```

```
$ gcc bow.c bowling.c -Wall
$ ./a.out
$
```

```
int main(){
  test_empty();
  test_all_ones();
  test_one_spare() ;
  return 0;
}
```

```
$ gcc bow.c bowling.c -Wall
$ ./a.out
a.out: bow.c:31: test_one_spare: Assertion `bg_score(game)==16
&& "test one spare"' failed.
```

bowling : one spare (back)

```
int bg_score(struct BowlingGame *g){
  int score=0,i ;
  for(i=0;i<g->current;++i) score+=g->rolls[i] ;
  return score ;
}
```

```
int bg_score(struct BowlingGame *g){
  int score=0,i ;
  for(i=0;i<g->current;++i) {
      if (g->rollls[i]+g->rolls[i+1]==10){
            // this is a spare...
            score= ... ; // ?
       }
      score+=g->rolls[i] ;
  }
  return score ;
}
```

ca ne marchera pas car il faut compter par set. On doit encore faire un refactoring !

```
int main(){
  test_empty();
  test_all_ones();
  //test_one_spare() ;
  return 0;
}
```

```
$ gcc bow.c bowling.c -Wall
$ ./a.out
$
```

bowling : refactoring (again)

```
int bg_score(struct BowlingGame *g){
  int score=0,i ;
  for(i=0;i<g->current;++i) score+=g->rolls[i] ;
  return score ;
}
```

```
struct BowlingGame *bg_init(){
  int i ;
  struct BowlingGame * g=malloc(sizeof(* g)) ;
  g->current=0 ;
  for(i=0;i<21;++i) g->rolls[i]=0 ;
  return g ;
}

int bg_score(struct BowlingGame *g){
  int score=0, frame;
  for(frame=0;frame<10;++frame) {
        score+=g->rolls[2*frame]+g->rolls[2*frame+1] ;
  }
  return score ;
```

```
$ gcc bow.c bowling.c -Wall
$ ./a.out
$
```

bowling : one spare (again)

```
$ gcc bow.c bowling.c -Wall
$ ./a.out
a.out: bow.c:31: test_one_spare: Assertion `bg_score(game)==16
&& "test one spare"' failed.
```

```
int bg_score(struct BowlingGame *g){
  int score=0, frame, hits;
  for(frame=0;frame<10;++frame) {
        hits=g->rolls[2*frame]+g->rolls[2*frame+1] ;
        score+=hits ;
        if (hits==10) // spare
              score+=+g->rolls[2*frame+2] ;
  }
  return score ;
}
```

```
$ gcc bow.c bowling.c -Wall
$ ./a.out
$
```

bowling : TDD

- Et ainsi de suite :
  - ajout d'un test avec un strike
  - cas pour la fin de partie
  - …
- Le cycle à suivre en TDD est :
  - écriture d'un test
  - le test ne passe pas : ROUGE
  - écriture du code
  - le test passe : VERT
- Lorsqu'on ré-écrit des tests, on ne touche pas au code jusqu'à ce que ça repasse au vert

Tests

- Il existe plusieurs types de tests.
- Les plus importants sont :
  - Les tests unitaires
  - Les tests fonctionnels
  - Les tests d'intégration
  - Les tests de recette

Les tests unitaires

- Les tests unitaires ont pour objet de valider le fonctionnement d'une fonction.
- Pour qu'un test unitaire soit correct, il faut tester le fonctionnement « normal » ainsi qu'aux limites (cas NULL, domaine de valeur).
- Un test unitaire doit tester une fonction le plus indépendamment possible du reste du code : comment faire si la fonction utilise d'autres fonctions ?